Replex: A Scalable, Highly Available Multi-Index Data Store

Amy Tai[⋆†], Michael Wei[⋆‡], Michael J. Freedman[†], Ittai Abraham[⋆], Dahlia Malkhi[⋆]

[⋆]*VMWare Research,* [†]*Princeton University,* [‡]*University of California, San Diego*

Abstract

The need for scalable, high-performance datastores has led to the development of NoSQL databases, which achieve scalability by partitioning data over a single key. However, programmers often need to query data with other keys, which data stores provide by either querying every partition, eliminating the benefits of partitioning, or replicating additional indexes, wasting the benefits of data replication.

In this paper, we show there is no need to compromise scalability for functionality. We present Replex, a datastore that enables efficient querying on multiple keys by rethinking data placement during replication. Traditionally, a data store is first globally partitioned, then each partition is replicated identically to multiple nodes. Instead, Replex relies on a novel replication unit, termed *replex*, which partitions a full copy of the data based on its unique key. Replexes eliminate any additional overhead to maintaining indices, at the cost of increasing recovery complexity. To address this issue, we also introduce *hybrid replexes*, which enable a rich design space for trading off steady-state performance with faster recovery. We build, parameterize, and evaluate Replex on multiple dimensions and find that Replex surpasses the steady-state and failure recovery performance of HyperDex, a state-of-the-art multi-key data store.

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[1, 3, 14, 9, 15, 7] have emerged that can support queries on multiple keys through the use of secondary indexes. 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 [12], 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). Replex actually recovers from failure faster than other NoSQL systems because of storage savings during replication.

We implement (§ 4) and evaluate (§ 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× while maintaining performance equivalent with HyperDex. In addition, we show that Replex can recover from one or two failures 2-3× 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.

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.

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 indexes. 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 indexes of and replicating the table.

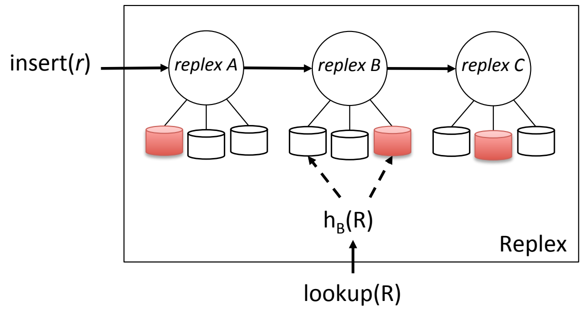


Figure 1: Every replex stores the table across of a number of partitions. This diagram shows the system model for a table with 3 indexes. When a row r is inserted, h_A, h_B, and h_C determine which partition (shaded) in the replex stores r. Similarly, a lookup on a replex is broadcast to a number of partitions based on h.

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 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 [27] 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 1; in Section 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

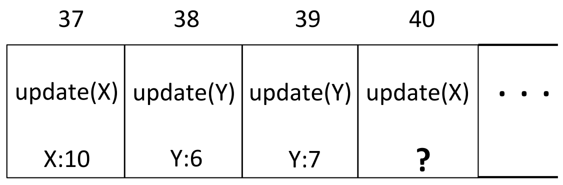


Figure 2: Consider storing every log entry in a Replex table. For linearizability, a local timestamp cannot appear to go backwards with respect to the global timestamp. For example, tagging in last entry with local timestamp X:9 violates the semantics of the global timestamp.

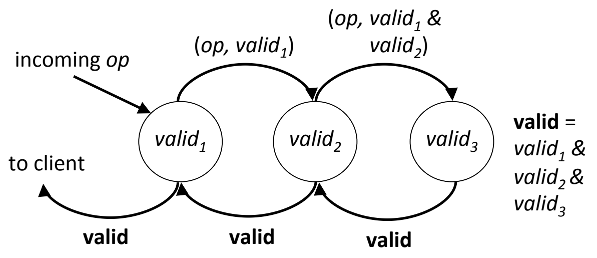


Figure 3: Each node represents an index. This modified replication protocol has two phases: 1) Top phase: propagates the operation to all relevant partitions and collects each partition's decision. 2) Bottom phase: the last partition aggregates these decisions into the final **valid** boolean, which is then propagated back up the chain. When a replex receives **valid**, it knows to commit or abort the operation

individual index structures, which translates to reductions in network traffic, operation latency, and storage inflation.

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 replexes 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 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 [4], Hyder [6], and CalvinFS [24].

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).

Datastores without global secondary indexes do not have this validity problem, because a key is only sorted by a single index. Datastores with global secondary indexes employ a distributed transaction for update operations, because an operation must be atomically replicated as valid or invalid across all the indexes [11]. Because replexes are similar to global secondary indexes, a distributed transaction can do the job. But to use a distributed transaction for every update operation would cripple system throughput.

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 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 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 [16]. 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 [23]. 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.

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 indexes 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 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 Hybrid Replexes

Suppose a user schema specifies a single table with two indexes, *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 4. Then after one failure, reads to replex *A* do not see any failure amplifcation. However, adding another

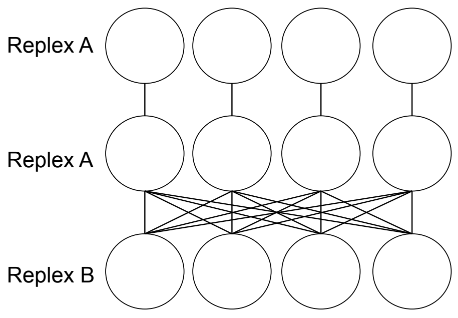


Figure 4: In graph depictions of replexes, nodes are partitions and edges indicate two partitions might share data. For example, because replexes *A* and *B* have independent sharding functions, it is possible for all combinations of nodes to share data. This graph shows a simple solution to reduce the failure amplification experienced by replex *A*, which is to replicate *A* again.

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.

3.1 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 Section 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 5 is

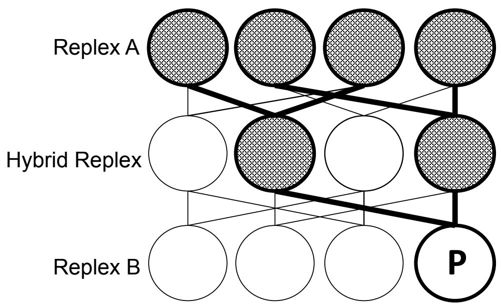


Figure 5: Each node is connected to exactly 2 nodes in another replex. This means that partitions in both replexes will see only 2x failure amplification after a single failure.

numbered from left to right from 0-3. Then:

$$h_{hybrid}(r) = 2 \cdot (h_A(r) \pmod 2) + h_B(r) \pmod 2 \quad (1)$$

The graph in Figure 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 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 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 Section 6, erasure

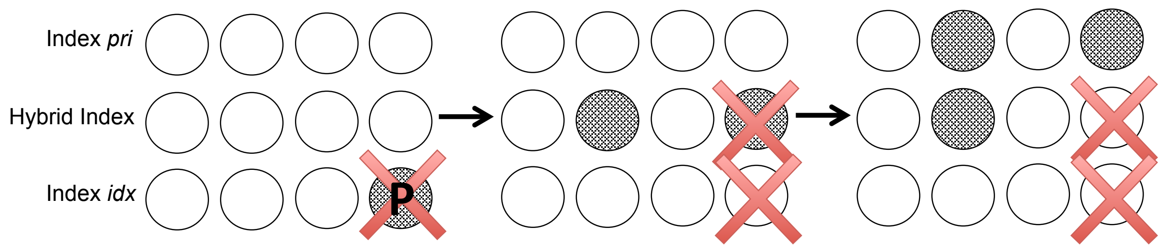


Figure 6: Graceful degradation. Shaded nodes indicate the nodes that must be contacted to satisfy queries that would have gone to partition P. As failures occur, Replex looks up replacement partitions for the failed node and modifies reads accordingly. Instead of contacting an entire replex after two failures, reads only need to contact a subset.

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 6.

In Figure 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.

3.2 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:

$$h_{hybrid}(r) = n_2 \cdot (f_A(r) \pmod{n_1}) + f_B(r) \pmod{n_2} \quad (2)$$

Applying this to Figure 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 More Extensions

In this section, we discuss intuition for further generalizing hybrid replexes. Explicit construction requires defining complex h_{hybrid} that is beyond the scope of this paper.

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 ex-

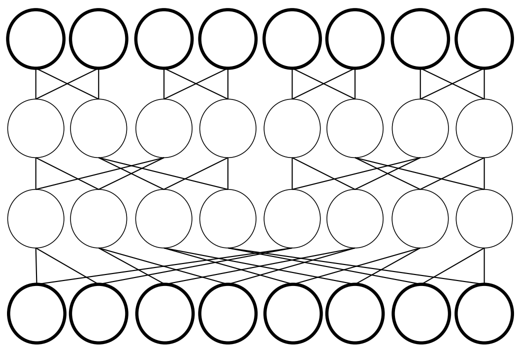


Figure 7: Inserting two hybrid replexes in between two replexes (in bold). Each node in the graph has outcast 2, which means after any partition fails, failure amplification will be at most 2x. After two failures, amplification will be 3x; after three, it will be 4x.

pected, if more replexes share a hybrid replex, improvement over $O(p)$ failure amplification becomes smaller.

For example, suppose a table requires 4 indexes, 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 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 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.

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 indexes for its data, including RDBMSs such as MySQL Cluster, as well as any NoSQL system. 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 Section 3. We reuse the chain replication that HyperDex provides to replicate across subspaces.

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 Section 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 $f = 0$.

5 Evaluation

Our evaluation is driven by the following questions:

- How does Replex's design affect steady-state index performance? (§ 5.1)
- How do hybrid replexes enable superior recovery performance? (§ 5.2)
- How can generalized 2-sharing allow a user to tune failure performance? (§ 5.3)

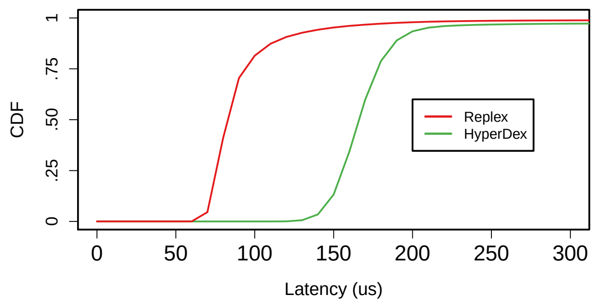


Figure 8: Insert latency microbenchmark CDF

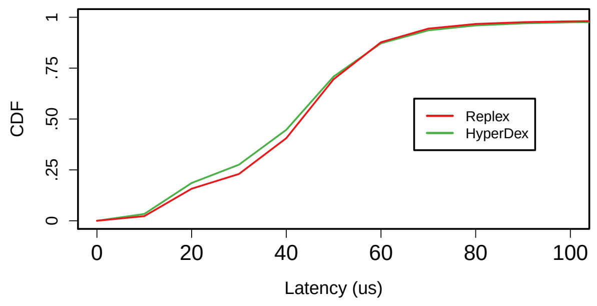


Figure 9: Read latency microbenchmark CDF

- How do hybrid replexes enable better resource tradeoffs with r-sharing? (§ 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.

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 indexes: 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 2. 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 9. More importantly, the insert latency for Replex is consistently

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 1: YCSB workloads.

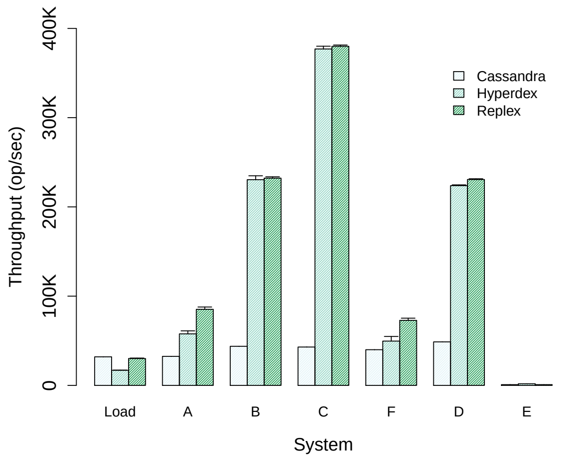


Figure 10: Mean throughput for full YCSB suite over 3 runs. Error bars indicate standard deviation. Results grouped by workload, in the order they are executed in the benchmark.

2x less than the latency of a HyperDex insert, as in Figure 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 2. In fact, the more indexes a user builds, the larger the factor of difference in latency inserts. This helps to demonstrate Replex's scalability compared to HyperDex.

Figure 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 [10]. 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.

Replex-3's lower latency insert operations translate to higher throughput on the load portion and Workloads A, F than both HyperDex and Cassandra; 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. Cas-

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

Table 2: Systems evaluated.

sandra has comparable load throughput 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.

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 indexes. Reads are split 50:50 between the two indexes. We kill a server after 25 seconds. Figure 11 shows the read throughput in the system as a function of time, and Tables 3 and 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 total amount of data across all replicas; recovery times in Tables 3 and 4 are approximately proportional to the Replication Factor column in Table 2. 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, 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 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 dimensions $n_1 = 3, n_2 = 4$. Second, only 3 partitions are responsible for reconstructing the

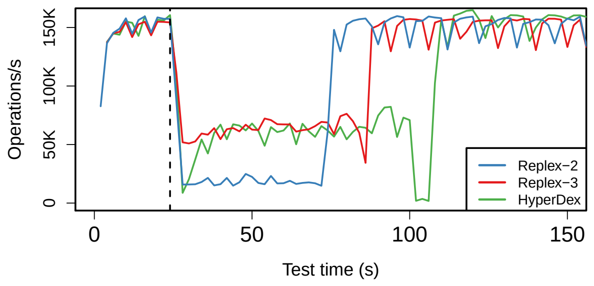


Figure 11: We crash a server at 25s and report read throughput for Replex-2, Replex-3, and Hyperdex. Systems are loaded with 10 million, 100 byte rows. All three systems experience a dip in throughput right before returning to full functionality due to the cost of reconfiguration synchronization, which introduces the reconstructed node back into the system configuration.

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: Recovery statistics of one machine failure after 25 seconds. 10 million, 100 byte records. Results reported as average time ± 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 4: Recovery statistics of one machine failure after 25 seconds. 1 million, 1KB records. Results reported as average time ± standard deviation of 3 runs.

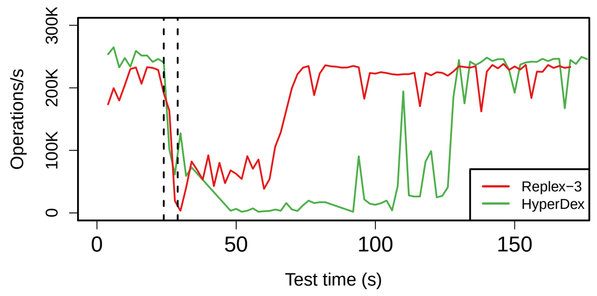


Figure 12: Read throughput after two failures. We crash one server at 25s and then a second at 30s. Request pile-up because throughput is used for recovery is responsible for the jumps in HyperDex throughput.

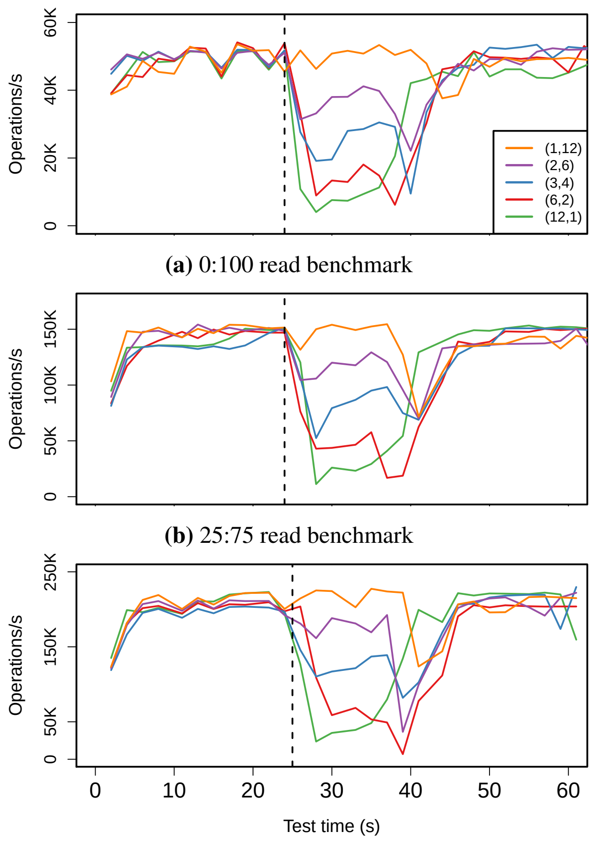
System	Recovery Time (s)	Recovery Throughput (op/s)
Replex-3	37.6 ± 1.2	$60{,}844 \pm 27{,}492$
HyperDex	98.0 ± 11	$30{,}220 \pm \ \ 8{,}104$

Table 5: Recovery statistics of two machine failures at 25s and 30s. Results reported as average time ± standard deviation of 3 runs. Recovery time is measured from the first failure.

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.

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 experiment. Figure 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 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.



(a) 0:100 read benchmark

(b) 25:75 read benchmark

(c) 50:50 read benchmark

Figure 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.

5.3 Parametrization of the Hybrid Replex

As discussed in Section 3.2, any hybrid replex $\mathscr{H}$ can be parametrized as (n_1,n_2). Consider the Replex-3 setup, which replicates operations to replexes in the order $A \rightarrow \mathscr{H} \rightarrow B$. If $\mathscr{H}$ 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 13 shows the throughput results

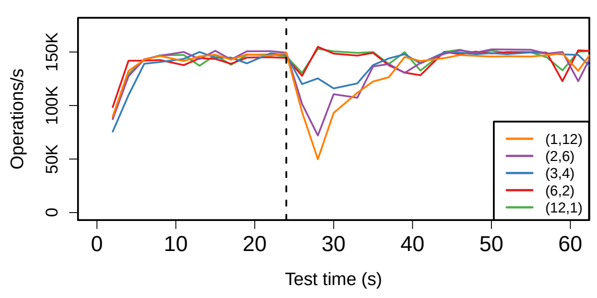


Figure 14: Replex-3 throughput with a 25:75 read benchmark. We crash a machine in replex A at 25s.

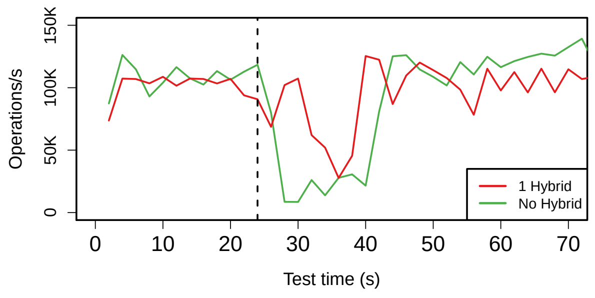


Figure 15: Read throughput after a failure at 25s with a 33:33:33 benchmark.

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 $\mathscr{H}$ 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 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.

5.4 Evaluating 3-Sharing

In the previous sections, all systems evaluated assumed 3-way replication. In particular, in Replex, if the number of indexes 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.

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

Table 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 ± standard deviation of 3 runs.

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 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 Section 3.3. Of course, depending on the importance of the three original indexes, 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.

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 indexes. 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 indexes: as the number of indexes grows, the fractional cost of adding a hybrid replex decreases, but the hybrid replex can still provide enormous gains during recovery.

6 Related Work

6.1 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 [25], LT [19], Online [18], Raptor [22], Reed-Solomon [20], Parity [8] and Tornado [17] 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 [13]. Similarly, hybrid replex construction allows fine tuning given a workload and failure model.

6.2 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 indexes. Cassandra [1], CouchDB [3], Hypertable [14], MongoDB [9], Riak [15] and SimpleDB [7] 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 [12] builds a global secondary index for each index, enabling efficent query of secondary indexes. However, each index is also replicated to maintain fault tolerance, which comes with a significant storage overhead. As we saw in Section 5, this leads to slower inserts and significant rebuild times on failure.

6.3 Relational (SQL) Databases

Traditional relational databases build multiple indexes and auxillary data structures, which are difficult to partition and scale. Sharded MySQL clusters [21, 10] 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 [26, 10]. For example, a query which involves a secondary index must contact each shard, just as with a multi-index datastore.

Yesquel[2] provides the features of SQL with the scalability of a NOSQL system. Like Hyperdex, however, Yesquel separately replicates every index.

6.4 Other Data stores

Corfu [4], Tango [5], and Hyder [6] 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 indexes 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.

7 Conclusion

Programmers need to be able to query data by more than just a single key. For many NoSQL systems, supporting multiple indexes 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 indexes.

Replex reconsiders multi-index data stores from the bottom-up, showing that implementing secondary indexes 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.

References

[1] *Cassandra*. `http://cassandra.apache.org/`.

[2] Marcos K Aguilera, Joshua B Leners, Ramakrishna Kotla, and Michael Walfish. Yesquel: scalable SQL storage for Web applications. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, 2015.

[3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide*. O'Reilly Media, Inc., 2010.

[4] Mahesh Balakrishnan, Dahlia Malkhi, John D Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. CORFU: A distributed shared log. *ACM Transactions on Computer Systems*, 31(4), 2013.

[5] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[6] Philip A Bernstein, Colin W Reid, and Sudipto Das. Hyder-a transactional record manager for shared flash. In *Proceedings of the Conference on Innovative Datasystems Research*, 2011.

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

[8] Peter M Chen, Edward K Lee, Garth A Gibson, Randy H Katz, and David A Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 1994.

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

[10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, 2010.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3), 2013.

[12] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4), 2012.

[13] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure storage. In *Proceedings of the USENIX Annual Technical Conference*, 2012.

[14] Ankur Khetrapal and Vinay Ganesh. HBase and Hypertable for large scale distributed storage systems. *Department of Computer Science, Purdue University*, 2006.

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

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

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

[18] Petar Maymounkov. Online codes. Technical report, New York University, 2002.

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

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

[21] Mikael Ronstrom and Lars Thalmann. MySQL cluster architecture overview. *MySQL Technical White Paper*, 2004.

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

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

[24] Alexander Thomson and Daniel 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.

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

[26] Bogdan George Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. In *Proceedings of Roedunet International Conference*. IEEE, 2011.

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