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

Amy Tai*§, Michael Wei*, Michael J. Freedman§, Ittai Abraham*, Dahlia Malkhi*

*VMware Research, §Princeton University
SQL vs NoSQL

SQL
- Rich querying model
- Transaction support
- Poor performance

NewSQL
- Rich querying model
- Transaction support
- High performance

NoSQL
- Simple, key-based querying
- No or limited transaction support
- Scalable

Ex: CockroachDB, Yesquel, H-Store
Ex: HyperDex, Cassandra
SQL vs NoSQL

SQL
- Rich querying model
- Transaction support
- Poor performance

NewSQL
- Rich querying model
- Transaction support
- High performance

NoSQL
- Simple, key-based querying
- No or limited transaction support
- Scalable

Ex: CockroachDB, Yesquel, H-Store
Key-value stores
Ex: HyperDex, Cassandra
Replex enables richer queries without sacrificing shared-nothing scale-out
NoSQL Scales with Shared-Nothing Partitioning
NoSQL Scales with Shared-Nothing Partitioning

Partitions
(with respect to the primary index)
Key Observations

1. Indexing enables richer queries
   (searches, joins, etc.)
Approach 1: Local Indexing

Index stored locally at each partition

insert (r[0] r[1] r[2] ... r[c])

Index updates are local
Approach 1: Local Indexing

Each partition builds a local index

```
select * where col2 = "foo"
```

- Index updates are local
- Index lookups must be broadcast to all partitions
- Potential synchronization across partitions
Approach 2: Global Indexing

Distributed data structure spans all partitions

Insert \((r[0] \ r[1] \ r[2] \ldots \ r[c])\)

- Index updates could be multi-hop
- Index lookups executed on a single data structure
Key Observations

1. Indexing enables richer queries (searches, joins, etc.)

2. Indexes more efficient if data is partitioned according to that index
Partitioning by Index $\rightarrow$ Storage Overheads

Partitioning by index $\rightarrow$ must store data again
Partitioning by Index → Storage Overheads

Ideally, build all indexes over a single copy of data
Partitioning by Index ➔ Storage Overheads

Ideally, build all indexes over a single copy of data

Then, replication also replicates indexes
Partitioning by Index → Storage Overheads

Instead, to make indexes durable, must 3-way replicate each partitioning of data.
Partitioning by Index $\rightarrow$ Storage Overheads

Overheads!

HyperDex
Replex solves the indexing problem by combining indexing and replication
Replex

New replication unit:

**replex** -- data replica partitioned and sorted with respect to an associated index

Serves data **replication** and **indexing**
Replex

New replication unit:

- Replace data replicas with replexes
- Indexing comes free during replication
- Serves data replication and indexing
System Architecture
Inserts in Replex

Replex uses a modified chain replication protocol
Which chain?

All pairs of partitions are potential chains
Inserts in Replex

\[ \text{insert} (r[0] \ r[1] \ r[2] \ \ldots \ r[c]) \]
Partition Determined by a replex’s Index

\[(r[0] \ r[1] \ r[2] \ldots \ r[c])\]
Partition Determined by a replex’s Index

\[
\text{insert} \quad (r[0] \ r[1] \ r[2] \ldots r[c])
\]
Partition Determined by a replex’s Index

insert
(r[0] r[1] r[2] . . . r[c])
Propagating a Commit Bit

\[
\text{insert} \quad (r[0] \ r[1] \ r[2] \ \ldots \ r[c])
\]
Commit bits can be aborts

\[\text{insert}\]

\[(r[0] \ r[1] \ r[2] \ldots \ r[c])\]

Uniqueness constraint on green replex can cause abort
Commit bits can be aborts

insert
(r[0] r[1] r[2] ... r[c])
Indexing is **Free**

insert

(r[0] r[1] r[2] . . . r[c])
Index Reads in Replex

1. Check for a commit bit
2. Only if the bit is true, can row be returned

```
select * where col2="foo"
```
Partition Failures

1. Partition Recovery
2. Client Requests

Index is unavailable, **data is available**
Partition Failures

Where is the data?

Index is unavailable, data is available
Finding Data After Failure

Parallel Recovery Requests must be multicast.
Lookups reduce to linear scan at each partition.

Partition
Parallel
Recovery
Time

Request Amplification

How to explore this tradeoff?
Hybrid Replex

Hybrid replexes constrain potential data chains → More targeted recovery
Recovery time vs Request Amplification

Recovery is less parallel... but lower request amplification
Hybrid Replex: Definition

- Shared across $r$ replexes
- Not associated with an index
- Partitioning function dependent on these $r$ replexes
Replication Chains with Hybrid Replex

\[ \text{insert} \ (r[0] \ r[1] \ r[2] \ \ldots \ r[c]) \]
Replication Chains with Hybrid Replex

\((r[0] \ r[1] \ r[2] \ldots \ r[c])\)
Replication Chains with Hybrid Replex

\((r_0 \ r_1 \ r_2 \ \ldots \ r_c)\)
Replication Chains with Hybrid Replex

(insert \( r[0] r[1] r[2] \ldots r[c] \))
Hybrid Replex: More Properties

1. Recovery time vs request amplification tradeoff
2. Improve failure availability of multiple replexes
3. Storage vs recovery performance tradeoff
4. Graceful failure degradation
Hybrid Replex: More Properties

1. Recovery time vs request amplification tradeoff
2. Improve failure availability of multiple replexes
3. Storage vs recovery performance tradeoff
4. Graceful failure degradation
Improving Failure Availability w/o Hybrid Replexes

2x Storage!
Improve failure availability of multiple replexes
Improve failure availability of multiple reflexes
Storage vs. recovery performance

1.5x Storage! ... but worse client requests
Implementation

• Built on top of HyperDex, ~700 LOC
• Implemented partition recovery and request re-routing on failure
• Implemented variety of hybrid replex configurations
Evaluation

1. What is impact of reflexes on steady-state performance?

2. How do hybrid reflexes affect failure performance?
Evaluation Setup

• Table with two indexes
• 12 server machines, 4 client machines
• All machines colocated in the same rack, connected via 1GB top-of-rack switch
• 8 CPU, 16GB memory per machine
Systems Under Evaluation

Replex-2

Replex-3

HyperDex
Steady State Latency

Reads

- Replex-3
- HyperDex

Inserts

- Replex-3
- HyperDex

Reads against either index have similar latency, but we report reads against primary index.

Replex-2 not included because it has a lower fault tolerance threshold.
Single Failure Performance

Experiment
- Load with 10M, 100 byte rows
- Split reads 50:50 between each index
Recovery Time

1. HyperDex recovers slowest because 2-3x more data
2. Replex-2 recovers fastest because least data, parallel recovery
Single Failure: Failure Throughput

1. Replex-2 low throughput because of high request amplification
2. Replex-3 has throughput comparable to HyperDex
Summary

1. Rethink the replication paradigm

2. Replacing replicas with reflexes decreases index storage AND maintenance overheads

3. Hybrid reflexes introduce rich tradeoff space for failure SLAs
Questions?