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

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

*VMware Research, §Princeton University
How to distribute a data store

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>. . .</th>
<th>colₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r[0]</td>
<td>r[1]</td>
<td>r[2]</td>
<td>. . .</td>
<td>r[c]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- rows
- col₀
- col₁
- col₂
- . . .
- colₙ

Database distribution model
Partition based on default key

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>colₖ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

rows

Default partitioning key, \( k \)
Partition based on default key

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>colₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitioning Function, PF(k):
returns the partition a row belongs to
Partitioning functions vary widely

• Typically, $PF(k)$ will include a hash: $PF(k) = PF'(h(k))$
• $PF(k)$ can range-partition, for locality
• $PF(k)$ can hash-partition, to eliminate hot spots
• $PF(k)$ can be consistent hashing, for elasticity

Main takeaway: All $PF$s compute on a fixed key
Partitioning functions vary widely

• Typically, $PF(k)$ will include a hash: $PF(k) = PF'(h(k))$
• $PF(k)$ can range-partition, for locality
• $PF(k)$ can hash-partition, to eliminate hot spots
• $PF(k)$ can be consistent hashing, for elasticity

Main takeaway: This is a problem
Fixed partitioning keys make secondary indexing hard
Secondary indexing

- Secondary indexing: providing fast access to data based on a set of columns *other than* the default partitioning key

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>colₙ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rows</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Table diagram](image)
Secondary indexing

- Secondary indexing: providing fast access to data based on a set of columns *other than* the default partitioning key
Secondary indexing enables richer queries

- Secondary indexing: providing fast access to data based on a set of columns *other than* the default partitioning key
- Queries based on arbitrary columns
- Joins
- Range queries
Lookups by the **primary** index:

```sql
select * where k=4
```

- **Identify partition**: PF(4) will return a partition, p
- At p, use a local index to find relevant rows

Lookups by the **secondary** index:

```sql
select * where other_attr=4
```

- **Identify partition**: PF(?)

Without k, all partitions must be visited
Problem:

Data is partitioned by a (primary) index different from the desired (secondary) index
Approach 1: Local secondary indexes

Index stored locally at each partition

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

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>col_c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Index updates are local
Approach 1: Lookup visits all partitions

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 secondary index

Distributed data structure spans all partitions

<table>
<thead>
<tr>
<th>col_0</th>
<th>col_1</th>
<th>col_2</th>
<th>...</th>
<th>col_c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Implementation varies:
- Cached at clients
- Each tree node sits on a different partition
- Replicated at each partition
Approach 2: Global secondary index

Distributed data structure spans all partitions

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>colₖ</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Insert (r₀ r₁ r₂ ... rₖ)

Index updates could be multi-hop

Index lookups do not need to visit all partitions
Problem:

Data is partitioned by an (primary) index different from the desired (secondary) index

What if we partition data according to every desired index?
Each partitioning also builds local indexes, so search within a partition is efficient
Each partitioning must store data again...

default primary key

<table>
<thead>
<tr>
<th>col_0</th>
<th>col_1</th>
<th>col_2</th>
<th>...</th>
<th>col_c</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
... creating storage overheads

default primary key
Redundant with data replication

<table>
<thead>
<tr>
<th>col₀</th>
<th>col₁</th>
<th>col₂</th>
<th>...</th>
<th>colₙ</th>
</tr>
</thead>
</table>

Data is now stored 5x!
Availability requirements of the partitionings...

To make each partitioning available, must 3-way replicate
... creates extreme storage and network overheads

To make each partitioning available, must 3-way replicate
Problem:
Data is partitioned by an index different from the desired index

What if we partition data according to every desired index?

... then the overheads limit scalability
Modern data stores trade off functionality for scalability

Replex eliminates this trade-off by combining indexing and replication
Replex

New replication unit:

**replex** -- data replica partitioned and sorted with respect to a desired 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

![Diagram of system architecture with a table and database icons.]
Inserts in Replex

Replexes are replicas => each replex must store all rows

Which partition replicates a specific row?
Inserts in Replex

\[
\text{insert} \\
(\text{r}[0] \ r[1] \ r[2] \ \ldots \ r[c])
\]
Par++on determined by a replex’s PF

\[ PF(r[0]) \]

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

Insert
Inserts in Replex

insert

(r[0] r[1] r[2] \ldots r[c])
Inserts in Replex

\[
\text{insert (r[0] r[1] r[2] \ldots r[c])}
\]

Similar to chain replication...
But each partition in CR is identical
Chain replication: identical partitions

\[
\text{insert} (r[0] \ r[1] \ r[2] \ \ldots \ r[c])
\]
Replex replication: different partitions...

Insert

\[(r[0] \ r[1] \ r[2] \ldots \ r[c])\]
... require a commit bit

insert
(r[0] r[1] r[2] ... r[c])
Commit bits can be aborts

insert

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

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

```sql
create unique index green_index on table
```
Commit bits can be aborts

\[
\text{insert} \quad (r[0], r[1], r[2], \ldots, r[c])
\]
Indexing is *free*

The replication protocol is enough to replicate *and* index data
Index reads in Replex

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

```sql
select * where col2="foo"
```
Partition failures

When a partition fails:
1. Rebuild the partition
2. Respond to client requests
Data is available, index is unavailable

When a partition fails:
1. Rebuild the partition
2. Respond to client requests
Data is available, index is unavailable

How to find the data if the index is unavailable?

When a partition fails:
1. Rebuild the partition
2. Respond to client requests
Recovery edges help find lost data

**Recovery edge**: connects two partitions that might *share data*

\[
(*, *, 'A'-'F', *) \}

\[
\begin{align*}
[A-'F'] & [G-'M'] & [N-'T'] & [U-'Z']
\end{align*}
\]
Complete graph of recovery edges
Using recovery edges to find data after failure
Using recovery edges to measure failure performance

Rebuild the partition

Respond to requests

Measured by:

Recovery Time

Request Amplification

Replex as-is:

Parallel Recovery

Requests must be broadcast to all partitions
Using recovery edges to measure failure performance

Rebuild the partition

Respond to requests

Measured by: Replex as-is

Recovery time

Request amplification

How to explore tradeoff between recovery time and request amplification?
Hybrid reflexes enable flexible recovery edge graphs
Hybrid reflexes enable flexible recovery edge graphs
Hybrid replexes enable flexible recovery edge graphs
Each recovery edge graph is a point in the tradeoff spectrum between recovery time and request amplification.
Hybrid replex: definition

replex: data replica partitioned and sorted with respect to a desired index

hybrid replex:
• Not associated with an index
• Shared across $r$ replexes
• Partitioning function dependent on these $r$ replexes (recovery edge graph is dependent on these $r$ replexes)
Inserts with a hybrid replex

(insert (r[0] r[1] r[2] ... r[c]))
Inserts with a hybrid replex

\[
insert (r[0], r[1], r[2], \ldots, r[c])
\]
Inserts with a hybrid replex

insert
(r[0] r[1] r[2] ... r[c])
Inserts with a hybrid replex

\[ \text{insert} \ (r[0] \ r[1] \ r[2] \ \ldots \ r[c]) \]
Inserts with a hybrid replex

\( \text{insert } (r[0] \ r[1] \ r[2] \ldots \ r[c]) \)
Inserts with a hybrid replex

\[
\text{insert (r[0] \ r[1] \ r[2] \ldots \ r[c])}
\]
Hybrid replex properties

1. Recovery time vs request amplification tradeoff
2. Improve failure availability of multiple replexes
3. Storage vs recovery performance tradeoff
Recovery time vs request amplification tradeoff

Recovery is less parallel... but lower request amplification
Improving failure availability w/o hybrid replexes
Improving failure availability w/o hybrid replexes

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

1.5x Storage! ... but worse client requests
Benchmarking Replex

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

2. How do hybrid replexes affect failure performance?
Benchmark 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 benchmarked

Replex-2

Replex-3

HyperDex
Steady state latency

**Reads**

![CDF graph for Reads](#)

**Inserts**

![CDF graph for Inserts](#)

- **Replex-3**
- **HyperDex**

3 network hops → 6 network hops
Experiment

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

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

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?