
# “All In" with Determinism for Performance and Testing in Distributed Systems

jhugg@voltdb.com

@johnhugg

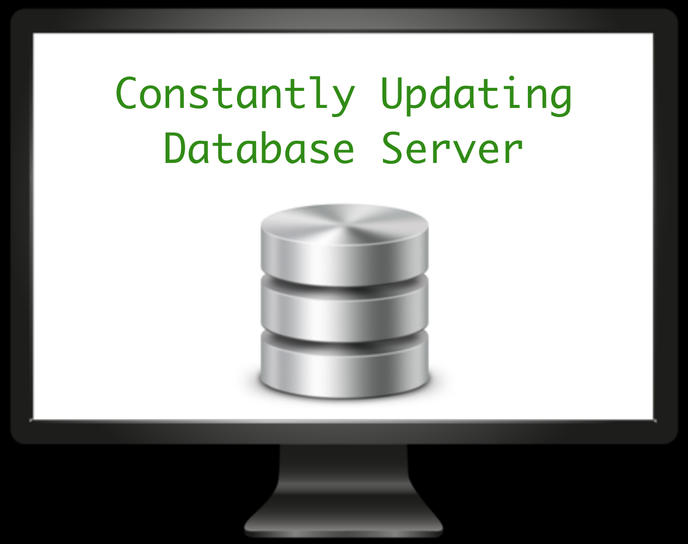
VOLTDB

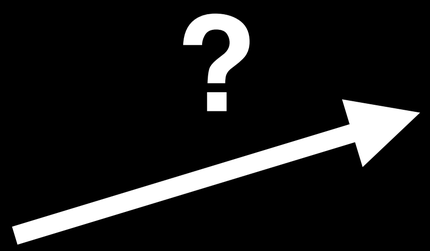
SMART DATA FAST.™

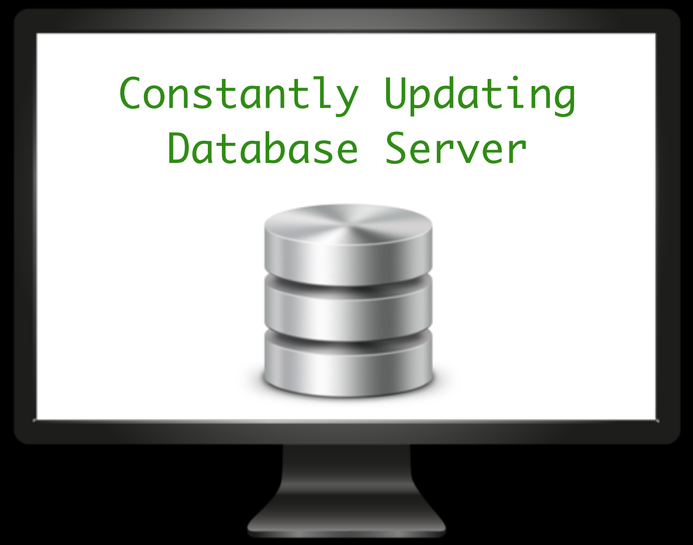
September 25th, 2015

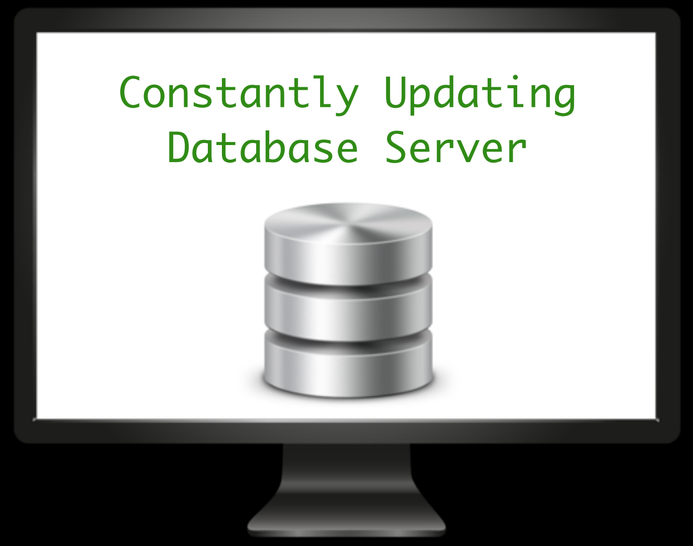
StrangeLoop, St. Louis

# So you need a replicated setup?



# Options

- Could run a primary and replica server pair and send a changelog to a replica (async or sync)

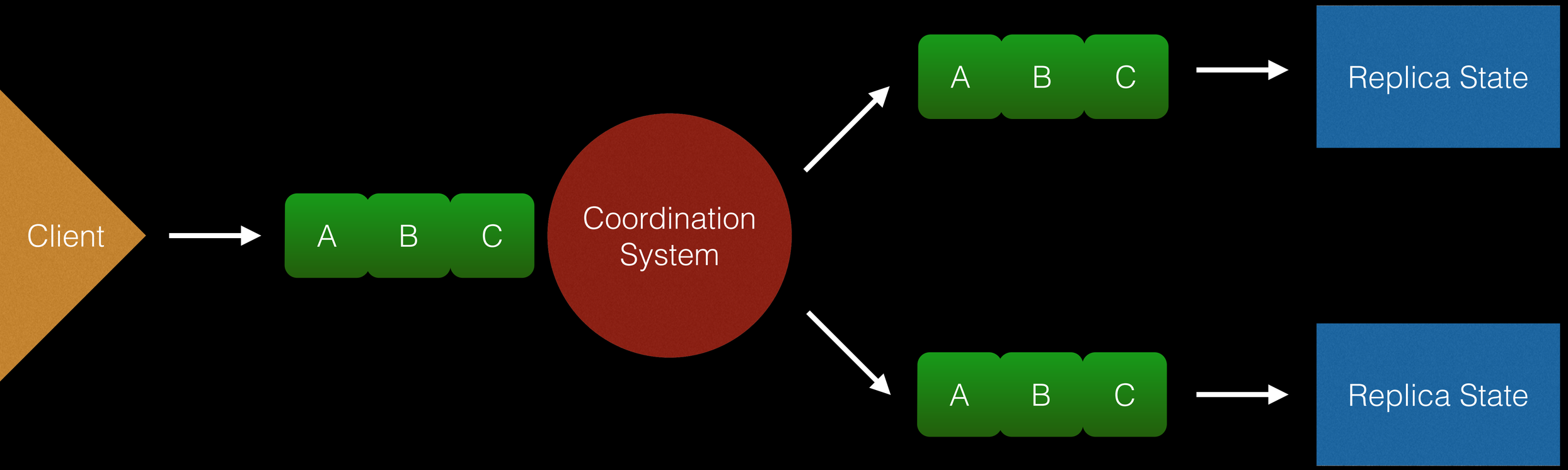
- Could allow writes to two servers, do conflict detection and repair (last write wins?)

# Meh

# Active-Active in Theory


# Active-Active in Theory


# Active-Active in Theory



Good


Connect Four
THE VERTICAL FOUR-IN-A-ROW
CHECKERS GAME
Ages 7 and Up

Bad


PLINKO

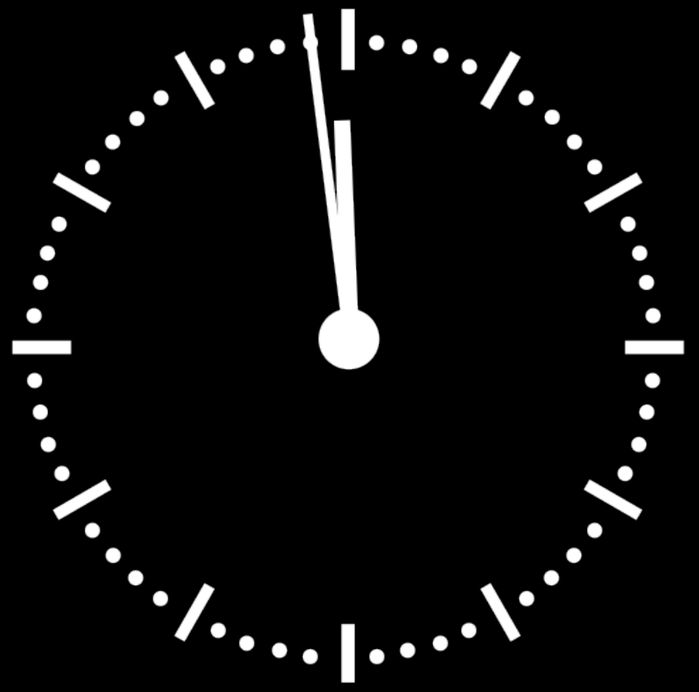

# This is a logical log

Do the same things in the same order to two copies of the same state… the order is a log!

- Can replicate this log over the network for active-active concurrent application to state.
- Can write this log to disk to an append-only, uhm… log… for disk recovery purposes.
- Can send this log over the network to another cluster in another datacenter for async replication.

# Sources of Non-Determinism

(aka “Pure Evil”)



```
insert into table
values (rand(), now());
```

Proc code too…

```
DELETE FROM BDAYS WHERE ID IN
(SELECT ID FROM BDAYS ORDER BY D LIMIT 1);
```

| BDAYS | |
|---|---|
| 1 | 1943-10-11 |
| 2 | 1944-08-17 |
| 3 | 2013-01-31 |
| **4** | **2013-01-31** |

| BDAYS | |
|---|---|
| 1 | 1943-10-11 |
| 2 | 1944-08-17 |
| 4 | 2013-01-31 |
| **3** | **2013-01-31** |

# External Systems

```
Begin Tranaction
   Read my id
   Ping the NOAA for the weather
   Write (id, wind speed) to db
Commit Transaction
```

# Non-User Sources of Non-Determinism

- Bad Memory
- Libraries that are more secure, randomize things to avoid attack, and break your database


Safety?

# Deterministic SQL

- VoltDB's SQL planner understands determinism.
- 100% of DML is deterministic. No limits on DML.
- If a query is in a RW transaction, sometimes a row scan will be swapped for a tree-index scan to achieve determinism.
- We warn you when you load a procedure that could be non-deterministic.

# Deterministic APIs

- VoltDB PRNG uses the same seed for all applications of a procedure, whether for active-active replication or for crash recovery replay.
- VoltDB can tell you what time it's supposed to be, which is better than asking your OS what time it actually thinks it is. Your OS is only guessing anyway.

# No Divergence Allowed

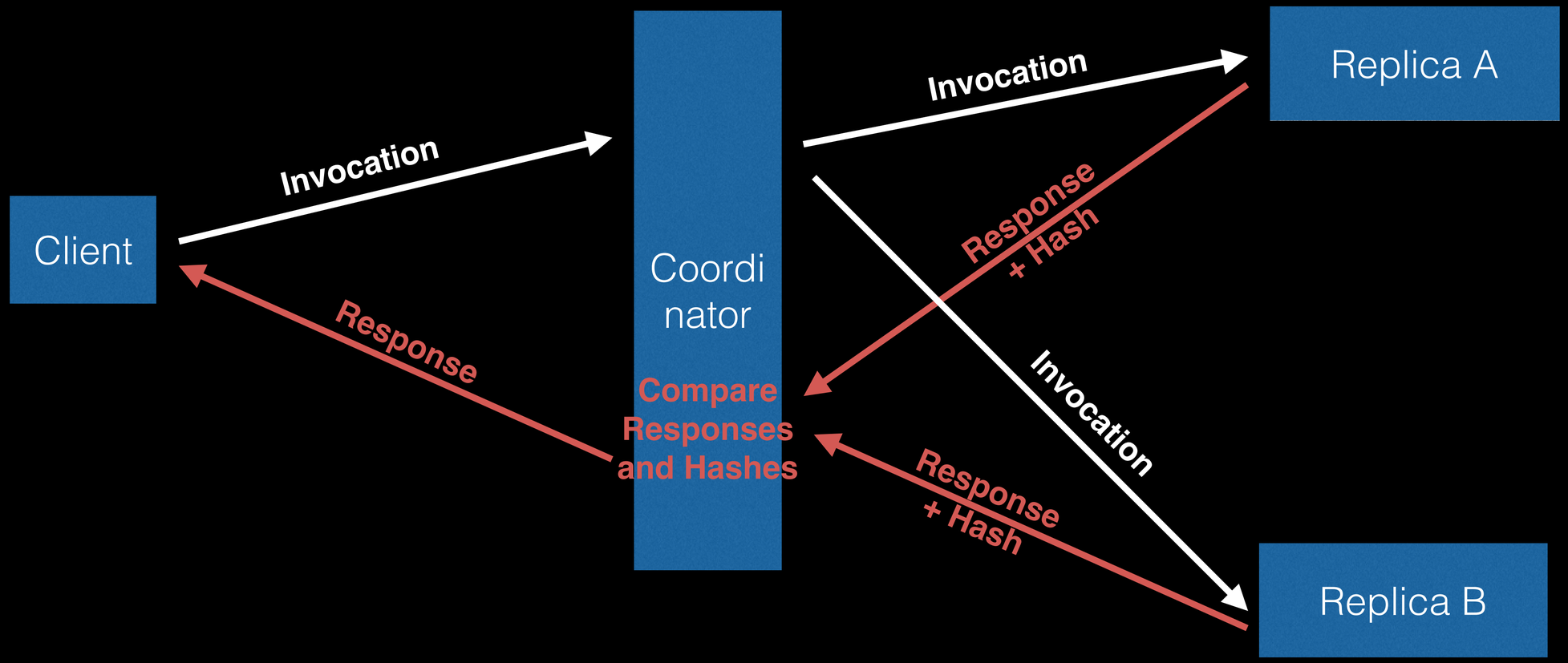
```
procedure foo(param1, param2…) {
    logic
    read sql
    logic
    write sql
    logic
    read sql
    logic
    write sql
    return response
}
```

# No Divergence Allowed

```
procedure foo(param1, param2…) {
    logic
    read sql
    logic
    write sql
    logic
    read sql
    logic
    write sql
    return response
}
```

Hash these up
(include any params)

# No Divergence Allowed



# Belt & Suspenders

- Can also check replica state is identical by hashing values while making a transactional snapshot.
- We could do more…
    - Sometimes we've made conscious decisions to let someone shoot themselves in the foot in the name of performance or memory use.
    - Sometime we just haven't gotten there yet.

# Performance

```
pizza(large, [ham, olives]);
```

Knead and toss dough

Spread sauce on dough

Spread cheese on sauce

Sprinkle oil and oregano on cheese

Add sliced ham on top

Add olives on top

Heat at 450° until cheese melts and crust turns slightly brown

Put in pizza box

# Why Deterministic Logical Log for Synchronous Replication?

- **Replicate Faster:** Can do tasks simultaneously, rather than one and then the other.
- **Persist Faster:** Can start logging when the operation is ordered, rather than when it completes.
- **Bounded size:** If the operations fit on a gigabit connections as they come in, you can log them at <= gigabit rates.

# So what?

- 100-500k ACID Transactions / Server (100% writes)
- Millions of SQL Statements / Second / Server (DML or DQL)
- Add more logic + SQL to each stored proc nearly for free
- Linear Scale measured to 30 nodes, probably goes higher
- Synchronous replication with minimal impact on latency
- Synchronous disk persistence with a huge head start
  (As low as 1ms average latency with a great tail profile)

# Boring Key-Value Note

- Key-Value CRUD is trivially deterministic.
- In fact, no difference between logical and binary logging.
- Theme: Dist-Sys is easier when your system only supports KV-CRUD.
- Seems like most interesting KV work is for local stores like LevelDB, RocksDB, etc…

# Tradeoffs n’ Compromises?

# Tradeoff #1

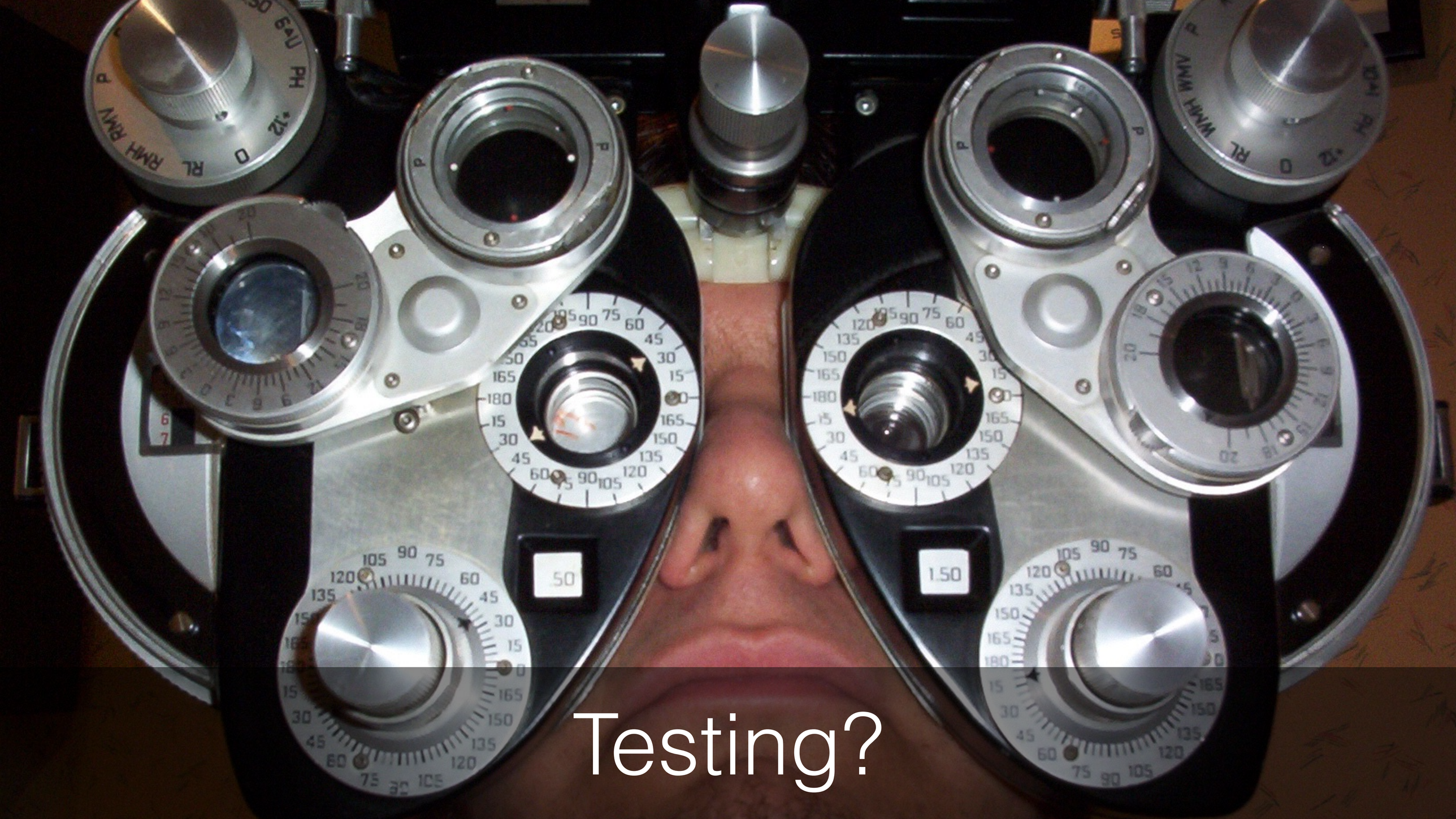
It’s more work


# Tradeoff #2

Running mixed versions is scary

# Tradeoff #3

- If we trip the non-determinism safety checks…


Testing?

# A Multi-Pronged Approach


- Test Distributed Consistency (ACID)
- Test SQL correctness
- Test Integrations
- Test Performance (in all conditions)
- Test Operational Procedures
- Test Clients

# ACID Review

| | |
|---|---|
| **ATOMIC** | The whole operation happens or not.<br>No partial failure. |
| **CONSISTENT** | Enforces schema constraints.<br>Not related to the “C” is “CAP”. |
| **ISOLATED** | No interference between concurrent operations. |
| **DURABLE** | Once an operation commits, it can’t be rolled back.<br>(It can be overwritten) |

# Isolation Levels

| | |
|---|---|
| **SERIALIZABLE** | **Any pair of operations must have a possible logical ordering. You must be able to say one happened before the other.** |
| REPEATABLE-READ | Explicit reads and writes are cool, but queries that filter may return different results if run more than once (phantom reads). |
| SNAPSHOT-READ | All reads against a logical point-in-time snapshot of the past. Usually that time is when the operation starts. |
| READ-COMMITTED | Only committed data is read. Data can change if read twice. |
| READ-UNCOMMITTED | No attempt at isolation is made. |

strangeloop

Sept 17-19, 2014 - St. Louis, MO
http://thestrangeloop.com

# Deterministic Simulation Testing of Distributed Systems

For Great Justice

FOUNDATIONDB

will.wilson@foundationdb.com

@FoundationDB

2014

# We went a different way…

- Lots of respect for what they did.
- Exhaustive states + Compressing time is awesome.
- We have more states. Way more.
- Simulation can’t find bugs outside of core state machines.
- We believe we get more value from our engineering resources our way.
- Still fully confident in the result.

# How Do We Test ACID?

- Message / State Machine Fuzzing
- Unit tests.
- Smoke tests.
- **A Self-Checking Workload**

# Leveraging Internal Checking

- SQL & parameters for SQL writes are hashed.
- Can we leverage this?
- So always feed values you read into SQL writes and VoltDB will ensure the reads are the same across replicas, replays and recoveries.
- “When it doubt, write it out!”

# Plan: Build a Nefarious App

TransactionId Selfcheck 2

# I is for isolation

Run concurrently

```
BEGIN
  Read Value X
  Increment X
  Read Value X for verification
COMMIT
```

# A is for atomic

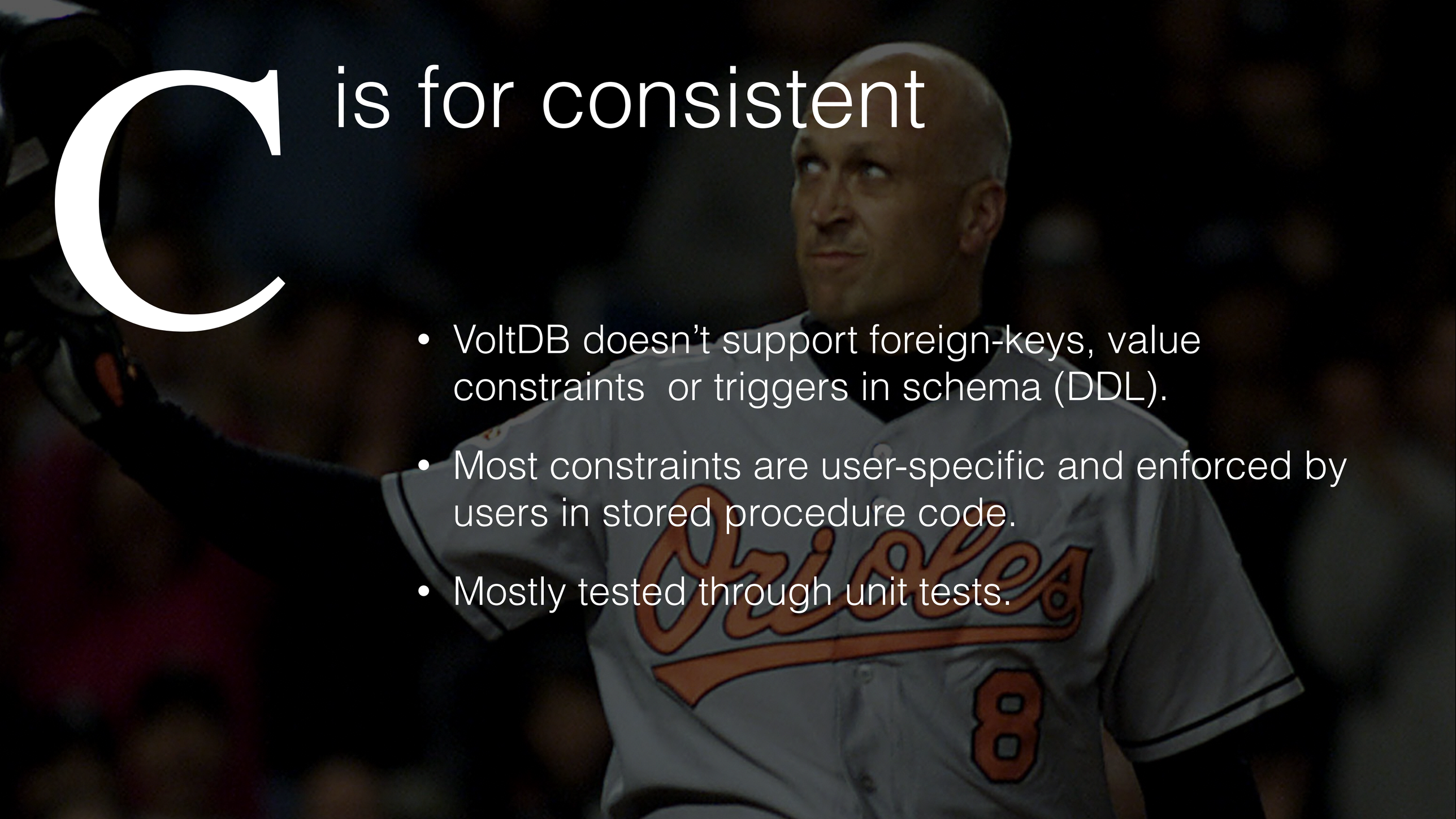
```
BEGIN
  Read Value X
  Increment X
  Maybe abort and roll back?
  Increment X
  Read Value X for verification
  Maybe abort and roll back?
  Read Value X for verification
COMMIT
```

Run concurrently

X should never be odd?

# D is for durable

- Build a data verifier that can inspect state and give 👍 or 👎.
- Build a committed tuple checker to verify that constraints are met on recovery.



# C is for consistent

- VoltDB doesn't support foreign-keys, value constraints  or triggers in schema (DDL).
- Most constraints are user-specific and enforced by users in stored procedure code.
- Mostly tested through unit tests.

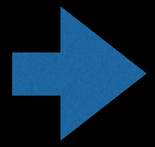
# C is for consistent

- Vo
  co
- Mo
  us
- Mo

# Workload Must Be Nasty

- The transactions presented in the past few slides are good… but not great. Let's make them better.
- Example:

```
BEGIN
  Read Value X
  Increment X
  Read Value X (verify)
COMMIT
```

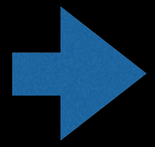


```
BEGIN
  Do 10 times:
    Read Value X
    Increment X
    Read Value X (verify)
COMMIT
```

# Workload Must Be Nasty

- The transactions presented in the past few slides are good… but not great. Let's make them better.
- Example:

```
BEGIN
  Read Value X
  Increment X
  Read Value X (verify)
COMMIT
```

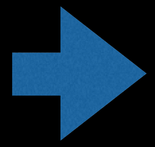


```
BEGIN
  Do 10 times:
    Read Value X
    Set X = hash(X)
    Read Value X (verify)
COMMIT
```

# Workload Must Be Nasty

- The transactions presented in the past few slides are good… but not great. Let's make them better.
- Example:

```
BEGIN
  Read Value X
  Increment X
  Read Value X (verify)
COMMIT
```

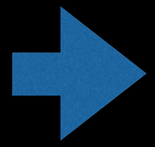


```
BEGIN
  Do 10 times:
    Read Value X, Y
    Set Y = X, X = hash(X)
    Read Value X, Y (verify)
COMMIT
```

# Workload Must Be Nasty

- The transactions presented in the past few slides are good… but not great. Let's make them better.
- Example:

```
BEGIN
  Read Value X
  Increment X
  Read Value X (verify)
COMMIT
```

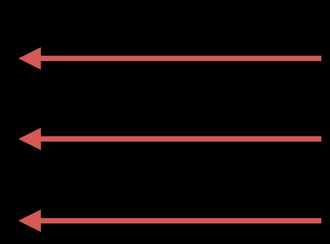


Actual transactions are much worse…

# Simplified Main Transaction

# Schema & Idea

```
CREATE TABLE t (
  cid          tinyint,
  rid          bigint,
  txnid        bigint,
  ts           bigint,
  prevtxnid    bigint,
  cidallhash   bigint,
  cnt          bigint,
  PRIMARY KEY (cid, txnid),
  UNIQUE ( cid, rid )
);
PARTITION TABLE partitioned ON COLUMN cid;
```



Client ID
Row ID within Client ID
Unique Number from VoltDB API
Timestamp from VoltDB API
See next slide
Hash of all rows with current cid
Incrementing counter for Client ID

```
CREATE TABLE t (
  cid         tinyint,
  rid         bigint,
  txnid       bigint,
  ts          bigint,
  prevtxnid   bigint,
  cidallhash  bigint,
  cnt         bigint
);
```

```
procedure p (cid, rid) {
  allrows = “SELECT * FROM t
             WHERE cid = ?
             ORDER BY rid DESC;”

  txnid = volt.getUniqueId()
  ts = volt.getTimestamp()
  prevtxnid = allrows[0].get(“txnid”)
  cidallhash = hash(allrows)
  cnt = allrows[0].get(“cnt”) + 1

  insert new row
  if cnt > 10, delete lowest rid row
}
```

**Concurrent!**

```
CREATE TABLE t (
  cid         tinyint,
  rid         bigint,
  txnid       bigint,
  ts          bigint,
  prevtxnid   bigint,
  cidallhash  bigint,
  cnt         bigint
);
```

```
procedure p (cid, rid, shouldRollback) {
  allrows = “SELECT * FROM t
             WHERE cid = ?
             ORDER BY rid DESC;”
  ensure all “cnt” values are consecutive

  txnid = volt.getUniqueId()
  ts = volt.getTimestamp()
  prevtxnid = allrows[0].get(“txnid”)
  cidallhash = hash(allrows)
  cnt = allrows[0].get(“cnt”) + 1

  insert new row
  if cnt > 10, delete lowest rid row
  ensure all “cnt” values are consecutive

  if shouldRollback, rollback
}
```

# Constraints

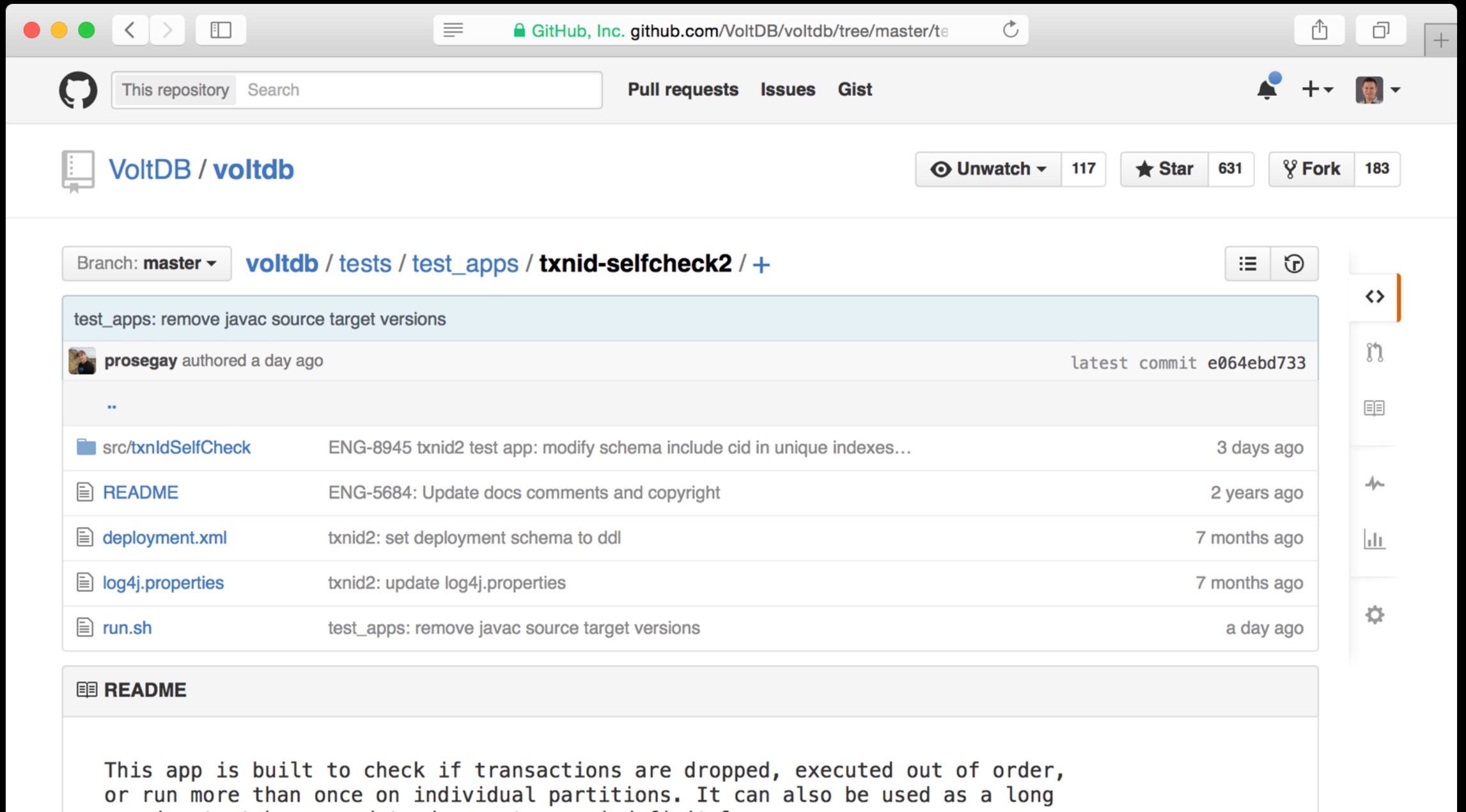
- There is a max number of rows per client.
- All rows for a client have consecutive row ids and counters
- Timestamps follow row id order
- rowA.rid + 1 = rowB.rid => rowA.txnid = rowA.prevtxnid

- Also test VoltDB APIs return same values across replicas and replays.

# Workload Tweaks

- Run on all permutations of partitioned and replicated data — and partitioned and global transactions.
- Add huge blobs to row here and there to slow things down.
- Add “mayhem threads” that run single-statement ad-hoc SQL that does updates.
- Add joins and dimension tables because this found bugs that I don’t understand.

# 1 of 35 workloads in /tests/test_apps



# Environment Tweaks

- Run on different operating systems, java versions, VM hosts, cloud PaaSs.
- Run on different hardware/networking. *Get weird.*
- Inject latency (network/disk). Inject failures.
- MIX ALL OF THE ABOVE!!!

# Committed Tuple Checker

A LOT OF WORK

- Client knows last sent transaction and last acknowledged transaction (oracle here)
- When running with synchronous durability, checker makes sure recovered data contains all acknowledged transactions.
- When running asynchronous, verify that fewer than a specific number of transactions are missing, and/or roughly a specific window of time.

A LOT OF WORK

# Big Advantage:
# Anyone Can Extend

- We updating our bulk ingestion frameworks with deeper hooks into coordination.
- As part of shipping that feature, we added threads that use bulk ingestion to TxnIdSelfcheck2. Then we made writes based on that data in other procedures, tripping any determinism checks.
- This found bugs.
- This found bugs immediately.
- We do this for all of our features that affect transactional guarantees / coordination / agreement.

# Big Advantage: Anyone Can Extend

- We’re rolling out multi-datacenter active-active replication between clusters this fall.
- TxnIdSelfcheck2 has been running successfully in this environment for MORE THAN A WEEK!
- Real workloads are easy to port to new topologies. We did a similar thing when we rolled out elastic expansion in 2013.

# Found a Bug. Now What?

- Two classes of bug here:
  - Things that happen fast.
  - Things that happen slow or randomly.
- A big chunk of the things that happen fast are easy to identify and fix.
- You get a decent amount of context.
- Often there’s a logical log on disk you can replay.
- But the tools could get better.

# Conclusion & Thanks

- Goal was to explain some of the benefits and tradeoffs we've run into when exploiting determinism and building tests that leverage that decision.
- Happy to answer whatever questions I can. Reach out to me.
- VoltDB is hiring, FWIW.


jhugg@voltdb.com
@johnhugg
http://chat.voltdb.com slack