H-Store And VoltDB
One Database In Two Universes

Yiqun (Ethan) Zhang
December 3, 2018
Carnegie Mellon University
yzhang@voltdb.com

https://www.linkedin.com/in/yzhang1991
AGENDA

• History
AGENDA

• Architectural Overview
AGENDA

• How VoltDB diverged from H-Store
AGENDA

• New research followed H-Store
read-only
long-running
complex joins
exploratory queries
OLAP
compression

column-store

ID  Name  City  Country  Order_no
What if state fits in memory?

• A lot of data sets do fit in memory
• 100 MB per warehouse in TPC-C
• Even data for 1,000 such warehouses can still fit!
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order

Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order

Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008

- Buffer manager: 29.6%
- Latching: 10.2%
- Locking: 18.7%
- Logging: 21%
- Btree keys: 8.1%
- in-memory data storage: 70.4%
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order

Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
OLTP transactions are short-lived

• The heaviest TPC-C transaction:
  • reads/writes ~200 records;
  • can be finished in less than 1 millisecond;
  • CPU is not the bottleneck.
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008

- Buffer manager: 29.6%
- Latching: 10.2%
- Locking: 18.7%
- Logging: 21%
- Btree keys: 8.1%

- Run single-threaded: 33.4%
- In-memory data storage
Single-threaded problems

- Waiting on users leaves CPU idle.
- Single-threaded does not jive with the multicore world.
Transactions are repetitive

- Queries are known in advance;
- Control flows are settled in advance too.
- External transaction control can be converted into pre-compiled stored procedures with structured control code intermixed with parameterized SQL commands on the server.
Waiting on users external transaction control

• Don’t
• External transaction control and performance are not friends;
• Use server-side transactional logic;
• Move the logic to data, not the other way around;
Using ALL the cores

- Partitioning data is a requirement for scale-out.
- Single-threaded is desired for efficiency.
  Why not partition to the core instead of the node?
- Concurrency via scheduling, not shared memory.
CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008

- 21% logging
- 18.7% locking
- 10.2% latching
- 29.6% buffer manager
- 8.1% Btree keys
- 33.4% durability through replication
- Run single-threaded
- In-memory data storage
Where did we spend our time?

CPU Cycle Breakdown for Shore on TPC-C New Order
Source: Harizopoulos, Abadi, Madden and Stonebraker, “OLTP Under the Looking Glass”, SIGMOD 2008
What did we end up building?

- **In-memory** relational SQL database.
- No external transaction control – Stored Procedures
- **Single-threaded** engines run in parallel.
- **Partitioned** to the core.
- Concurrency via **Scheduling**, not shared memory.
- **Serializable** ACID.
- Durability through **Replication**
Architecture
Run in parallel

In-memory store

Single-threaded engine
### Node #1

<table>
<thead>
<tr>
<th>Gear</th>
<th>People</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Node #2

<table>
<thead>
<tr>
<th>Gear</th>
<th>People</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**PLAYER**

<table>
<thead>
<tr>
<th>PLAYER_ID</th>
<th>LAST_NAME</th>
<th>FIRST_NAME</th>
<th>CREDITS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Replicated table

Node #1

Node #2

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS
Node #1

Node #2

Read from a replicated table

Command Router

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

VOLTDB
Read from a replicated table

Command Router

Node #1

Node #2
Write to a replicated table

Node #1

Node #2

WRITE

Command Router

<table>
<thead>
<tr>
<th>PLAYER</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLAYER_ID</td>
</tr>
<tr>
<td>LAST_NAME</td>
</tr>
<tr>
<td>FIRST_NAME</td>
</tr>
<tr>
<td>CREDITS</td>
</tr>
</tbody>
</table>
Write to a replicated table

Node #1

Node #2

Command Router

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS
Write to a replicated table

Node #1

Node #2

Command Router

WRITE

28
Partitioned Table
### Single Partition Read

```
select * from PLAYER where PLAYER_ID = 687
```

#### Node #1
- **123, Brown, Joe, 100**
- **456, Silvers, Phil, 77**
- **234, Green, Peter, 41**
- **567, Brown, Mary, 68**

#### Node #2
- **345, White, Betty, 94**
- **687, Black, Mark, 55**
- **525, Snow, Ann, 73**
Node #1

- 123, Brown, Joe, 100
- 456, Silvers, Phil, 77
- 234, Green, Peter, 41
- 567, Brown, Mary, 68

Node #2

- 345, White, Betty, 94
- 687, Black, Mark, 55
- 525, Snow, Ann, 73

Single Partition Read

```sql
select * from PLAYER where PLAYER_ID = 687
```
**Node #1**

- **123, Brown, Joe, 100**
- **456, Silvers, Phil, 77**

**Node #2**

- **234, Green, Peter, 41**
- **567, Brown, Mary, 68**

**Single Partition Read**

```
select * from PLAYER where PLAYER_ID = 687
```

**Command Router**

- **PLAYER**
  - **PLAYER_ID**
  - **LAST_NAME**
  - **FIRST_NAME**
  - **CREDITS**

**READ**

- **345, White, Betty, 94**
- **687, Black, Mark, 55**
- **525, Snow, Ann, 73**
Node #1

123, Brown, Joe, 100
456, Silvers, Phil, 77
234, Green, Peter, 41
567, Brown, Mary, 68

Node #2

345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73

---

**Single Partition Write**

update PLAYER set CREDITS = 50 where PLAYER_ID = 123

---

Command Router

WRITE
update PLAYER set CREDITS = 50 where PLAYER_ID = 123
update PLAYER set CREDITS = 50 where PLAYER_ID = 123
Multi Partition Read

select * from PLAYER where CREDITS > 75

Command Router

Node #1

123, Brown, Joe, 50
456, Silvers, Phil, 77
234, Green, Peter, 41
567, Brown, Mary, 68

Node #2

345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73
Node #1

Node #2

Multi Partition Read

select * from PLAYER where CREDITS > 75
Multi Partition Read

```sql
select * from PLAYER where CREDITS > 75
```
update PLAYER set CREDITS = 0 where CREDITS < 60;
update PLAYER set CREDITS = 0 where CREDITS < 60;
Multi Partition Write

update PLAYER set CREDITS = 0 where CREDITS < 60;
Multi Partition Writes

• Need two-phase commit.
• Simple solution – block until the transaction finishes.
• Introduces network stall - **BAD**.
Single Partition case

Client → Coordinator → Partition 1

execute execute execute execute
Multi Partition case

Client

Coordinator

Partition 1

execute

execute

execute

execute

execute

execute

Partition 2

network stall

network stall

......

......
Multi Partition Case

Client

Coordinator

Partition 1

Partition 2

Early Acknowledgement

execute

execute

execute

network stall

......
Durability not guaranteed

select * from PLAYER where PLAYER_ID = 687
Node #1

- **123, Brown, Joe, 100**
- **456, Silvers, Phil, 77**

Node #2

- **234, Green, Peter, 41**
- **567, Brown, Mary, 68**

---

**Durability not guaranteed**

**Command Router**

**select * from PLAYER where PLAYER_ID = 687**

---

**Player Table**

- **PLAYER_ID**
- **LAST_NAME**
- **FIRST_NAME**
- **CREDITS**

**Command Router**

**SELECT**

- **PLAYERS**

**Command Router**

**SELECT**

- **PLAYERS**
Node #1

123, Brown, Joe, 100
456, Silvers, Phil, 77

Node #2

234, Green, Peter, 41
567, Brown, Mary, 68

select * from PLAYER where PLAYER_ID = 687

Durability not guaranteed

Command Router

345, White, Betty, 94
687, Black, Mark, 55
525, Snow, Ann, 73
Read from a replicated table

Command Router

Node #1

Node #2

38
Read from a replicated table

Node #1

Node #2

Command Router

38
Read from a replicated table

Command Router

Node #1

Node #2

READ

PLAYER

PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS
Node #1

Node #2

Read from a replicated table

Command Router

Replication!
Node #1

123, Brown, Joe, 50
456, Silvers, Phil, 77

Node #2

123, Brown, Joe, 50
456, Silvers, Phil, 77

Durability through replication

Command Router

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS

VOLTDB
Node #1

Partition leader

- 123, Brown, Joe, 50
- 456, Silvers, Phil, 77

Node #2

Partition leader

- 234, Green, Peter, 41
- 567, Brown, Mary, 68

Durability through replication

select * from PLAYER where PLAYER_ID = 234

Command Router
Node #1

Partition leader

123, Brown, Joe, 50
456, Silvers, Phil, 77

Node #2

Partition leader

123, Brown, Joe, 50
456, Silvers, Phil, 77

Durability through replication

select * from PLAYER where PLAYER_ID = 234

Command Router
Node #1

Partition leader

123, Brown, Joe, 50
456, Silvers, Phil, 77
234, Green, Peter, 41
567, Brown, Mary, 68

Node #2

![Error Icon]

select * from PLAYER where PLAYER_ID = 234

Durability through replication

Command Router

PLAYER
PLAYER_ID | LAST_NAME | FIRST_NAME | CREDITS
--------- | --------- | ---------- | -------

READ

!!!

39
Node #1

Durability through replication

select * from PLAYER where PLAYER_ID = 234

Node #2

READ

Command Router
Durability through replication

select * from PLAYER where PLAYER_ID = 234
ACTIVE VS. PASSIVE

Approach #1: Active-Active
→ A txn executes at each replica independently.
→ Need to check at the end whether the txn ends up with the same result at each replica.

Approach #2: Active-Passive
→ Each txn executes at a single location and propagates the changes to the replica.
→ Not the same as master-replica vs. multi-master
Active-Active Replication

Client → Coordinator

Coordinator → Partition 1

Partition 1 → Replica

Replica → Coordinator

execute

execute

execute

execute

network stall
Recall that for the Multi Partition case...
SP + Replication as bad as MP?

SP + Replication (K-safety) blocks K + 1 partitions
still has parallelism

MP blocks **ALL** partitions
**NO** parallelism
Determinism in Active-Active Replication

- Running the same transaction against several replicas.
- How do you ensure they end up with the same result?
Query Order

CREATE TABLE t (val INT);

CREATE TABLE t (val INT);
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
Query Order

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
UPDATE t SET val = val * 10;
Query Order

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
UPDATE t SET val = val * 10;

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
UPDATE t SET val = val * 10;
INSERT INTO t VALUES (2);

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
Tuple Order

CREATE TABLE t (val INT);

CREATE TABLE t (val INT);
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
DELETE FROM t LIMIT 1 ORDER BY val;

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
DELETE FROM t LIMIT 1;

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

CREATE TABLE t (val INT);
INSERT INTO t VALUES (1);
INSERT INTO t VALUES (2);
DELETE FROM t LIMIT 1;

<table>
<thead>
<tr>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
INSERT INTO t VALUES ( TODAY() );

2018/12/03

2018/12/03 23:59:59
Function Determinism

```
INSERT INTO t VALUES (TODAY());
```

- **2018/12/03**
- **2018/12/04 00:00:00**
Function Determinism

INSERT INTO t VALUES ('2018/12/03');
INSERT INTO t VALUES (TODAY());
H-Store

- Run single-threaded
- In-memory data storage
- Undo logging
- Durability through replication

8.1% BTree keys
21% logging
18.7% locking
10.2% latching
29.6% buffer manager
12.3%
What did we have to change? - except logos
#1 Disk-based durability

• No one had any interest whatsoever in in-memory-only OLTP.
Durability - Command Logging

- **Deterministic, Serializable** operations written to the command log on disk.
- Replay operations on the same starting state in the fixed order reproduces the same ending state.
- **Serializable Isolation**: a performance trick, rather than a performance compromise.
Why log the command?

- Bounded Size - throughput
- Latency
Write-Ahead Logging

Client

Coordinator

Partition 1

Disk

execute

log

Before Values (UNDO)
After Values (REDO)
Command Logging (Sync)

Only the operation is logged
Command Logging (Async)

Client

Coordinator

Partition 1

Disk

execute

log

log
Command Logging (Async)

Back Pressure mechanism to make sure the command log does not fall too far behind.
Checkpoint Snapshot

MVCC – “two version” concurrency control

User Data

Command Log Truncated by Snapshot

Tunable Frequency

Transactions

Command log
#2 Cross Datacenter Replication

- Durability
- Geographically Dispersed Datacenters
- Active-Passive and Active-Active
• Active-Active Geo Datacenter Replication
• Asynchronous Replication
• Conflict Detection
• Different Cluster Topologies
#3 Memory Fragmentation

- Long running clusters used more memory
- Memory usage doesn’t shrink after data deletion
Bucketing and Compaction

Tuple Storage

20% full
40% full
60% full
80% full

Index

Swap the node for deletion with something at the end of the allocated storage, fixing links up when needed.
#4 Shared Replicated Table

- Space efficiency
- Engine Complexity
Replicated table

A cluster configuration from a customer:
• 48 CPU cores (sites)
• 512 GB RAM
• 10Gbps ethernet
• 6 nodes
• k-safety = 1

A 100 MB replicated table takes
100 x 48 x 6 = 28,800 MB
SRT saved significant memory space

Node #1

Node #2

A 100 MB replicated table takes
100 x 48 x 6 = 28,800 MB
SRT saved significant memory space

A 100 MB replicated table takes
100 x 6 = \(600\) MB
Write to a shared replicated table

Node #1

Node #2

WRITE

Command Router
Write to a shared replicated table
Write to a shared replicated table

Node #1

Node #2

Command Router

WRITE

PLAYER
PLAYER_ID
LAST_NAME
FIRST_NAME
CREDITS
Latches in the execution engine

```java
latch.countDown();
if (isLowestSite()) {
    latch.await();
    doWrite();
}
```

**DEADLOCK**
- Current transaction cannot finish
- Next transaction cannot begin
Engine Memory Context Switch

Partitioned Table P join Replicated Table R:
#5 Materialized Views

• One of things that enables the streaming power in VoltDB.
SELECT c1, COUNT(*), SUM(c2+c3) FROM T WHERE ...

Without Materialized Views:

- NETWORKING
- TXN OVERHEAD
- ADD TUPLE IN MEM x 500K/s
- NETWORKING
- TXN OVERHEAD
- QUERY DASHBOARD x 1K/s

With Materialized Views:

- NETWORKING
- TXN OVERHEAD
- ADD TUPLE IN MEM
- UPDATE VIEW x 500K/s
- NETWORKING
- TXN OVERHEAD
- QUERY VIEW x 1K/s
#6 Importer/Exporters

• When you process transactions at extremely high velocity, the problem starts to look like stream processing a little bit.
Summary: AT HIGH VELOCITY

• Nobody wants black-box state. Real-time understanding has value.
• OLTP apps smell like stream processing apps.
• Processing and state management go well together.
• Adding features to a fast/stateful core is easier than reinventing wheels.
#7 More SQL

- User-Defined Functions
- Common Table Expressions
- Better planning via Calcite (In Progress)
- and more...
Things that were changed

• Disk-based Durability
• Cross Datacenter Replication
• Memory Fragmentation
• Shared Replicated Tables
• Materialized Views
• importers and Exporters
• More SQL
New Research Directions

• Stream Processing capabilities - S-Store
• Larger-than-memory data management
• Improve Multi Partition Transaction Performance
H-Store → S-Store: Stream Processing

• New constructs for streams:
  • **Window**: finite chunks of state over (possibly unbounded) streams.
  • **Trigger**: computations to be invoked for newly generated data.
  • **Workflow**: computation pipelines of dependent transactions.

• **Tuple TTL (Time-To-Live) – VoltDB 8.2**
Larger than memory data management

- More often than not, OLTP workloads have **hot** and **cold** portions of the database.

- General approach:
  - Identify cold tuples (online/offline)
  - Evict cold tuples to disk (when? track?)
  - Tuple retrieval (how? granularity?)
  - Tuple merge (when?)

- A lot of implementations:
  - H-Store, MemSQL, Hekaton (SQL Server In-Memory), etc.
Smarter Scheduling

- Use data-heavy node as coordinator
- reduces data movement

N-Partition instead of All-Partition

- Disable undo logging when possible (SP only)

Speculative concurrency control
- Execute other transactions speculatively while waiting for commit/abort.

- Use Markov model for transaction behavior forecast.
Smarter Partitioning

- Partition database to reduce the number of distributed transactions.
- Large-Neighborhood Search with sample workload trace.
- Skew-aware Cost Model
- Replicated secondary index
Elastic Partitioning: E-Store

- Two-tiered partitioning:
  - Individual hot tuples
  - Large blocks of colder tuples
- Tuple-level monitoring
- Tuple placement planning
- Online reconfiguration
Thank you