VoltDB The NewSQL database for high velocity applications

New Age Transactional Systems – Not Your Grandpa's OLTP

John Hugg

Sr. Software Engineer, VoltDB

jhugg@voltdb.com

strangeloop 2011

Transactional Data Profile

Old OLTP

- 100% human generated
- "Transactions per Minute"
- 100% "business" (i.e., structured) data
- Requires 100% accuracy, consistency

New OLTP

- Mix of human and system generated
  - + Human self-service means lots of web endpoints
  - + System-generated data is huge growth driver
- Hundreds of thousands of TPS
- Structured and semi-structured data
- Requires 100% accuracy, consistency

New Age Data Defined

- Velocity
  - Moves at very high rates (think sensor-driven systems)
  - Valuable in its temporal, volatile state
- Volume
  - Fast-moving data creates massive historical archives
  - Valuable for mining patterns, trends and relationships
- Variety
  - Structured (logs, business transactions)
  - Semi-structured and unstructured

3Vs

This talk is a lot about…

High

Velocity

Data

What’s a High Velocity Use-Case Look Like?

- Lots of independent things are happening at a very high frequency
- You want to update some state based on those events
- You want to query that state in real time, usually with pre-defined queries
- You ultimately want to record those events in a analytic store such as an OLAP RDBMS or Hadoop
- Often you’re on a budget

Casually referred to (by us) as:

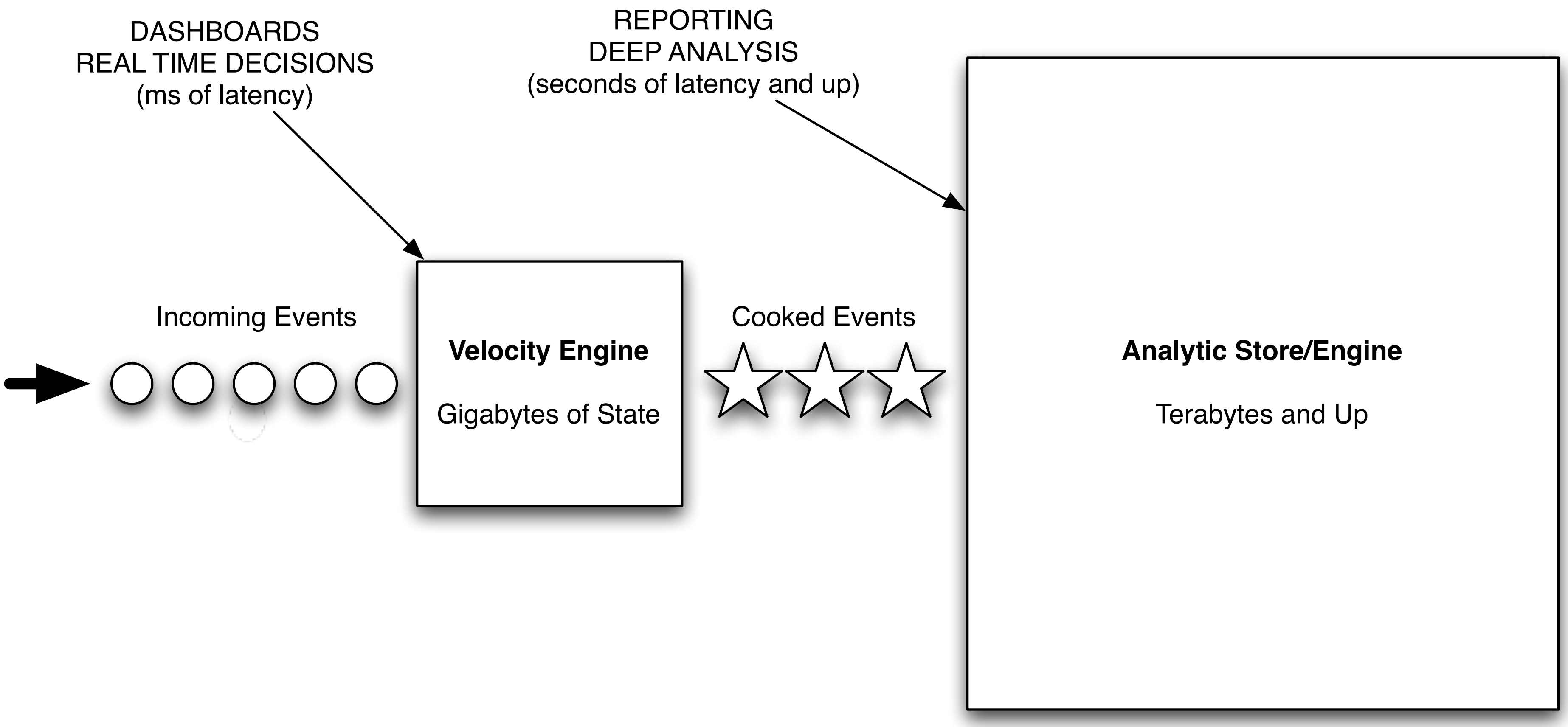
The Giant

Scoreboard in the Sky

High Velocity Data Use Cases

	Data Source	High-frequency operations	Lower-frequency operations
Financial trade monitoring	Capital markets	Write/index all trades, store tick data	Show consolidated risk across traders
Telco call data record management	Call initiation request	Real-time authorization	Fraud detection/analysis
Website analytics, fraud detection	Inbound HTTP requests	Visitor logging, analysis, alerting	Traffic pattern analytics
Online gaming micro transactions	Online game	Rank scores: •Defined intervals •Player "bests"	Leaderboard lookups
Digital ad exchange services	Real-time ad trading systems	Match form factor, placement criteria, bid/ask	Report ad performance from exhaust stream
Wireless location-based services	Mobile device location sensor	Location updates, QoS, transactions	Analytics on transactions

A picture is worth a thousand bullets…



High Velocity Data Challenges

You need to **ingest** in real-time

You need to **validate** in real-time

You need to **count** and **aggregate** in real-time

You need to **enrich** in real-time

You need to **analyze** and **respond** in real-time

High Velocity Data Management

- Ingest at very high speeds and rates
  - + Write in a transactionally consistent way
  - + Scale up and out as the rates increase
- Stay up in the face of failures
  - + Make handling failures and recovery as automatic as possible
- Support complex manipulations of state per event
  - + Service a range of real-time (or “near-time”) analytics
  - + Evolve easily to handle new analytic queries
- Integrate easily with high volume analytic datastores

So Let's Go Fast

Re-imagining the RDBMS

- Shared project
  + MIT / Yale / Brown / Vertica Systems
- Rethink the RDBMS for 21st century workloads
- Build screaming fast prototype
- Possibly commercialize (VoltDB)
- Research continues
  + http://hstore.cs.brown.edu/

The End of an Architectural Era
(It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos
MIT CSAIL
{stonebraker, madden, dna, stavros}@csail.mit.edu

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

ABSTRACT

1. INTRODUCTION

How Fast is Fast?

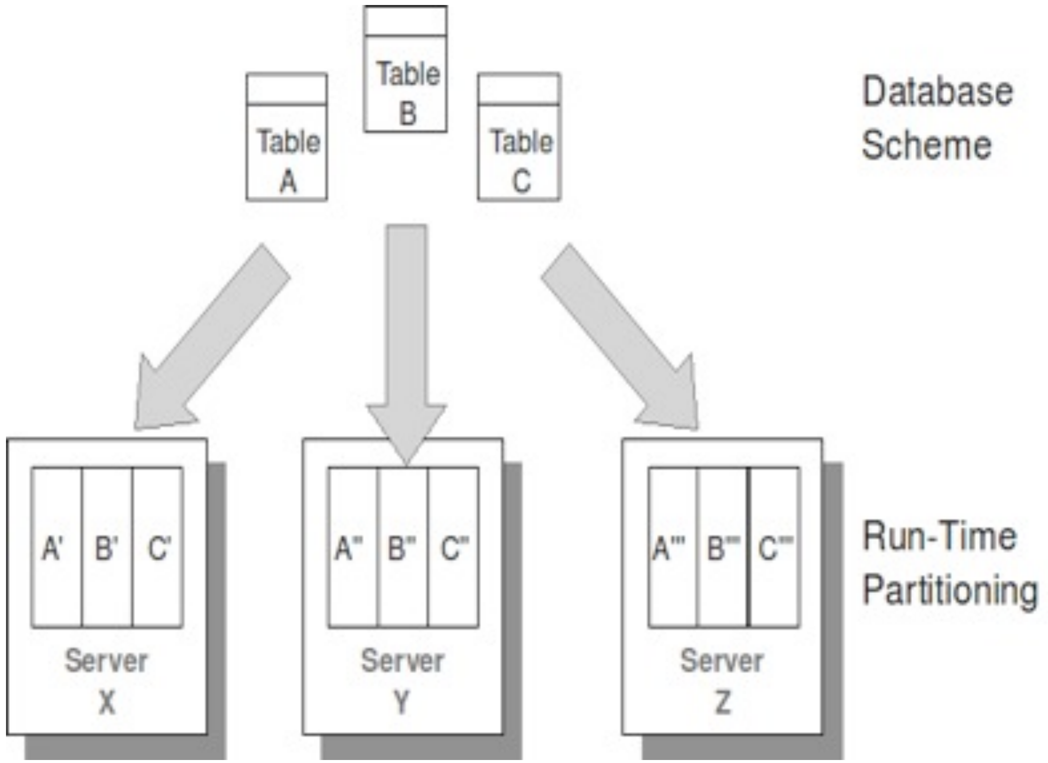
Per 8-core node:

- > 1 million SQL statements per second
- > 50,000 multi-statement procedures per second
- > 100,000 simpler procedures per second
- < 10ms average latency

Throughput and Scaling

- Scale to dozens of nodes
  - + At high node speeds, a dozen nodes is a LOT
  - + Most apps will use fewer than 10 nodes
- Process millions of events/transactions per second
- Minimize drag from HA and durability

How: Data



Split tables horizontally into **PARTITIONS**

Leverage Single Client Round-Trips

Transaction ==

Single invocation from a client:

+ One SQL Statement or a Stored Procedure Invocation

Two kinds of transactions – both ACID

Single-Partition

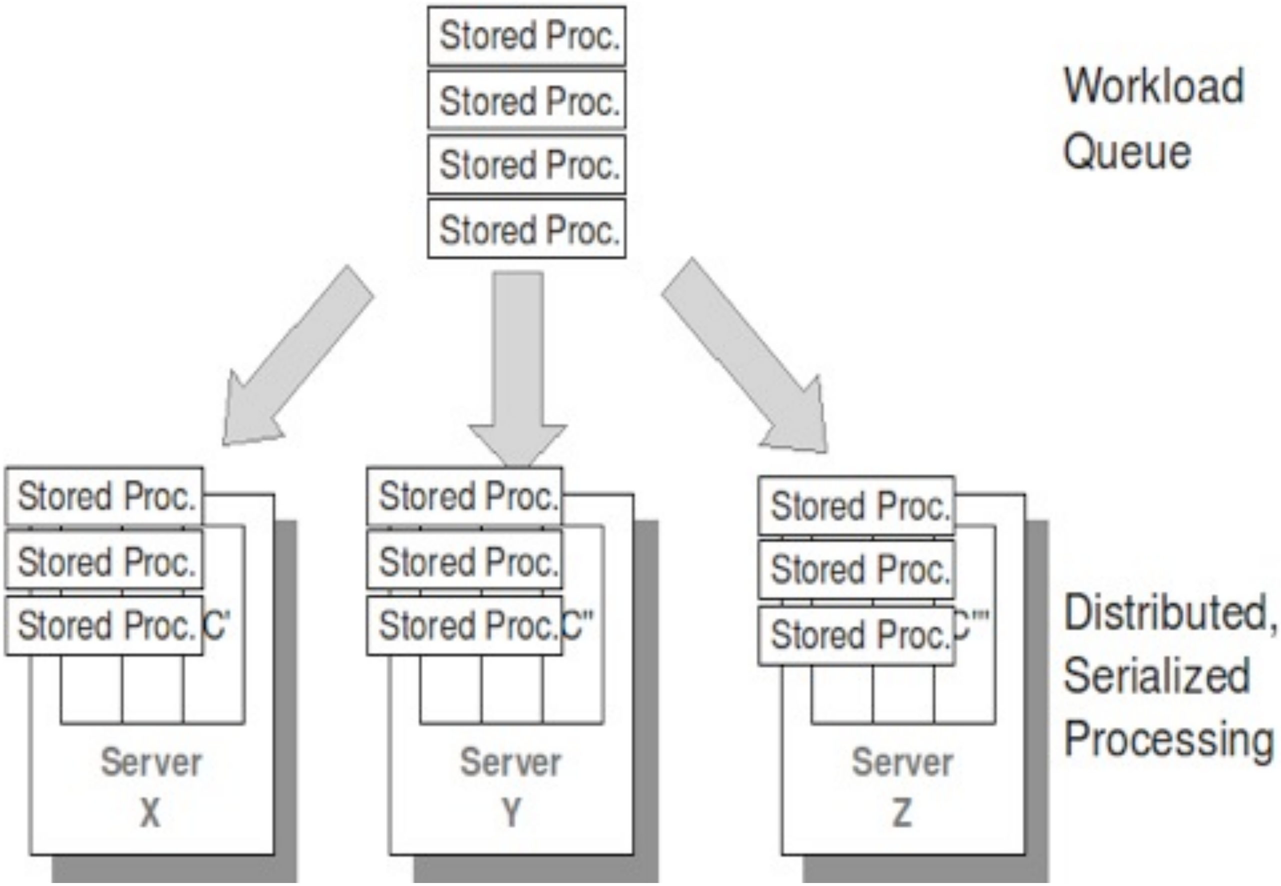
All reads/writes within a single partition

Multi-Partition

Operations on partitioned tables across partitions

Insert/update/delete on replicated data

How: Processing



Routed, **order** and **run** procedures at partitions

Running transactions

- Single-threaded executor at each partition
- No locking/dead-locks
- Transactions run to completion, serially, at each partition
- Single partition procedures run in microseconds
- However, single-threaded comes at a price
  + Other transactions wait for running transaction to complete
  + Don't do anything crazy in a procedure (request web page, send email)

Fast Because:

Eliminate stalls during transactions

Use Main Memory

No disk waits during a transaction

Transaction == One Invocation

No server <-> client network waits/chatter

Concurrency by scheduling, not by locking

Dashboard and Analytic Queries

- Multi-partition procedures get a consistent view of cluster-wide data
- Can run thousands per second depending on your configuration
- Materialized views amortize the cost of aggregation queries across event processing
- Many opportunities to optimize for common use cases

Example View

```
-- Agg. votes by contestant. Determine winner
create view votes_by_contestant (
   contestent_number, num_votes)
as select contestant_number, count(*)
from votes
group by contestant_number;
```

Don’t Go Down Easily

Obvious

- Store all data on more than one machine
- Don’t have a single point of failure
- Let app owner decide how many copies (i.e., make the trade-off between fault tolerance and cost)

Slightly Less Obvious

- Make the system easy to back up
    - \+ Make restore from backup trivial
- Don’t tell a client that work is safe UNTIL it’s confirmed in more than one place
- Actively check that replicated data is the same data
- Don’t allow a split-brain situation
- Make replacing a failed node trivial

For Some Reason, Less Obvious

- The less complexity required in user code, the better
- The less complexity in the system itself, the better

Consistency = Power

Counting is Hard: Voter Example

N contestants on some reality talent show

Vote over the phone and count votes per contestant

- Option 1: Accept that you're not going to be perfectly accurate, and that it might be difficult to bound how not accurate.
- Option 2: Buffer batches of votes for a while to amortize the cost of using external locks. The real-time count for a contestant is a count(*) on live buffers plus the roll up of prev. buffers.
- Option 3: Support isolated, atomic read-then-update for fault-tolerant counters (at very high throughput).

Counting is Hard: Voter Example

N contestants on some reality talent show

Vote over the phone and count votes per contestant

```
BEGIN PROCEDURE (PHONE, CONTESTANT)
    Atomically increment the votes for the contestant by 1
END PROCEDURE
```

Counting is Hard: Voter Example

N contestants on some reality talent show

Vote over the phone and count votes per contestant

Limit voters to X number of votes

BEGIN PROCEDURE (PHONE, CONTESTANT)

Check how many votes a phone number has.

If too many, return.

Otherwise atomically increment the votes for the phone number and for the contestant.

END PROCEDURE

Counting is Hard: Voter Example

- N contestants on some reality talent show.
- Vote over the phone and count votes per contestant.
- Limit voters to X number of votes.
- Allow voters who've signed up for the email list to vote twice as many times.
- Don't allow voters to vote twice in the same minute.
- Send voters an SMS text message after their 3rd vote. DON'T SEND TWO.
- Provide a live dashboard of contestant scores by geo.
- Record votes past the limits, but don't count them.

High Velocity Data Use Cases

	Data Source	High-frequency operations	Lower-frequency operations
Financial trade monitoring	Capital markets	Write/index all trades, store tick data	Show consolidated risk across traders
Telco call data record management	Call initiation request	Real-time authorization	Fraud detection/analysis
Website analytics, fraud detection	Inbound HTTP requests	Visitor logging, analysis, alerting	Traffic pattern analytics
Online gaming micro transactions	Online game	Rank scores: •Defined intervals •Player "bests"	Leaderboard lookups
Digital ad exchange services	Real-time ad trading systems	Match form factor, placement criteria, bid/ask	Report ad performance from exhaust stream
Wireless location-based services	Mobile device location sensor	Location updates, QoS, transactions	Analytics on transactions

Hooray For Consistency

- Consistency makes developing and iterating complex applications easier
  - + Sometimes applications are more complex than they initially seem

- Read-then-update == GOOD
- Update two things together == GOOD

Stuff you can get used to

CAP Theorem

In the face of imperfect networks (partitions), you can be always **consistent**, or always **available**, but not both.

CAP

- CAP is an proven/accepted theory based on a theoretical model.
- In *practice*, CAP says:
  Consistent $\Rightarrow$ $\exists$ failure modes that will bring you down.

- You didn't need fancy proofs to tell you that.
- But the point is, it's possible that by reducing consistency, you might avoid or mitigate one or more modes of failure.

CAP

- CAP is an proven/accepted theory based on a theoretical model.
- In practice, CAP says:
  Consistent $\Rightarrow$ $\exists$ failures that will bring you down.
- How likely?
  Depends.
- How much work to avoid?
  Depends.

CAP

- CAP is an proven/accepted theory based on a theoretical model.
- In practice, CAP says:
  Consistent $\Rightarrow$ $\exists$ failures that will bring you down.

- Should I spend my engineering resources on staying up in the face of those failures?
- Or should make choices to minimize them, have a plan in place to get back up fast when they happen, and spend my engineering resources on making my product better?

Network Partitions & Transactional RDBMSs

- A network partition splits one cluster into two, mutually isolated clusters.
- In the majority of cases, one of the two clusters will stop and the other will continue.
- In the small number of cases where both clusters must be stopped to ensure consistency, both are transactionally snapshotted to disk for easy recovery.
- Lots of user tradeoffs: EC2 vs. Dedicated, Redundancy Levels, etc..

Consistency/Transactions vs. Not (Summary)

- For systems at scale that get close to 100% uptime, the biggest commonality is tremendous effort, not the data engine.

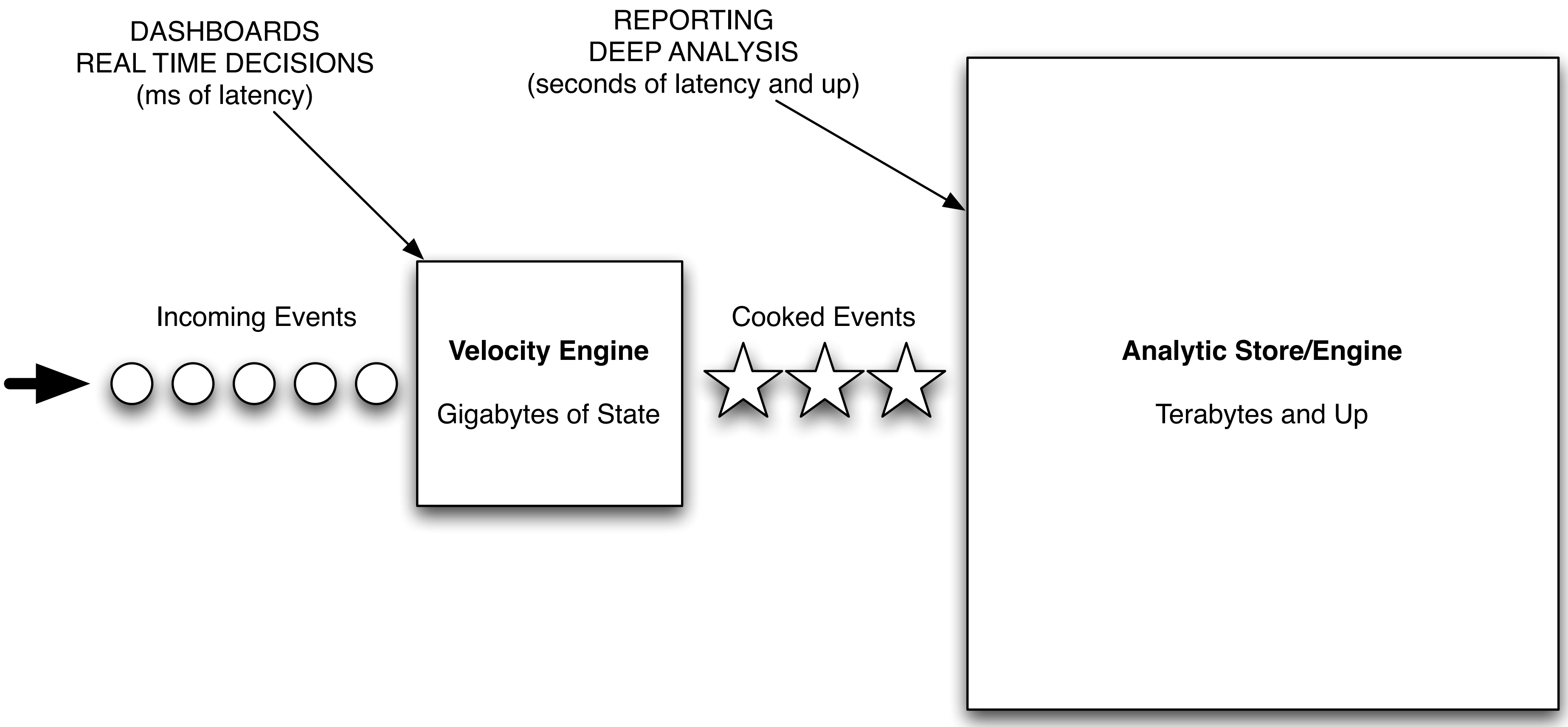
- Having your data engine prepared for lots of fun kinds of failure is a Good Thing™.

- Lots of systems out there make different real-world tradeoffs.

Pick a system that maximizes value, not ideology.

Data Lifecycle

Back to the Big Picture



What's Impossible

- If you're changing your state in the OLTP system at a rate that caused you to choose a super high performance OLTP system, then traditional ETL is broken.
- i.e. "Pull" is broken

- Data needs to be analyzed in place or pushed out.
- Both have strengths and weaknesses:
  + Multiple purpose-built systems are better at their purpose.
  + All-in-one systems are beautiful (like a unicorn)

Integration with Analytic Datastores

- Database should offer high performance, transactional export/migration (push)
- Export should allow a wide variety of common data enrichment operations
  + Normalize and de-normalize
  + De-duplicate
  + Aggregate
- Architecture should support loosely-coupled integrations
  + Impedance mismatches
  + Durability

For Example …

BEGIN TXN

Select from regular tables

MAKE COOL DECISIONS n’ STUFF™

Insert / Update regular tables

Insert into export stream

END TXN

For Example …

BEGIN TXN (phone number, contestant)

Look up the phone number of the voter

Update the contestant's score

Insert tuple into an export stream:

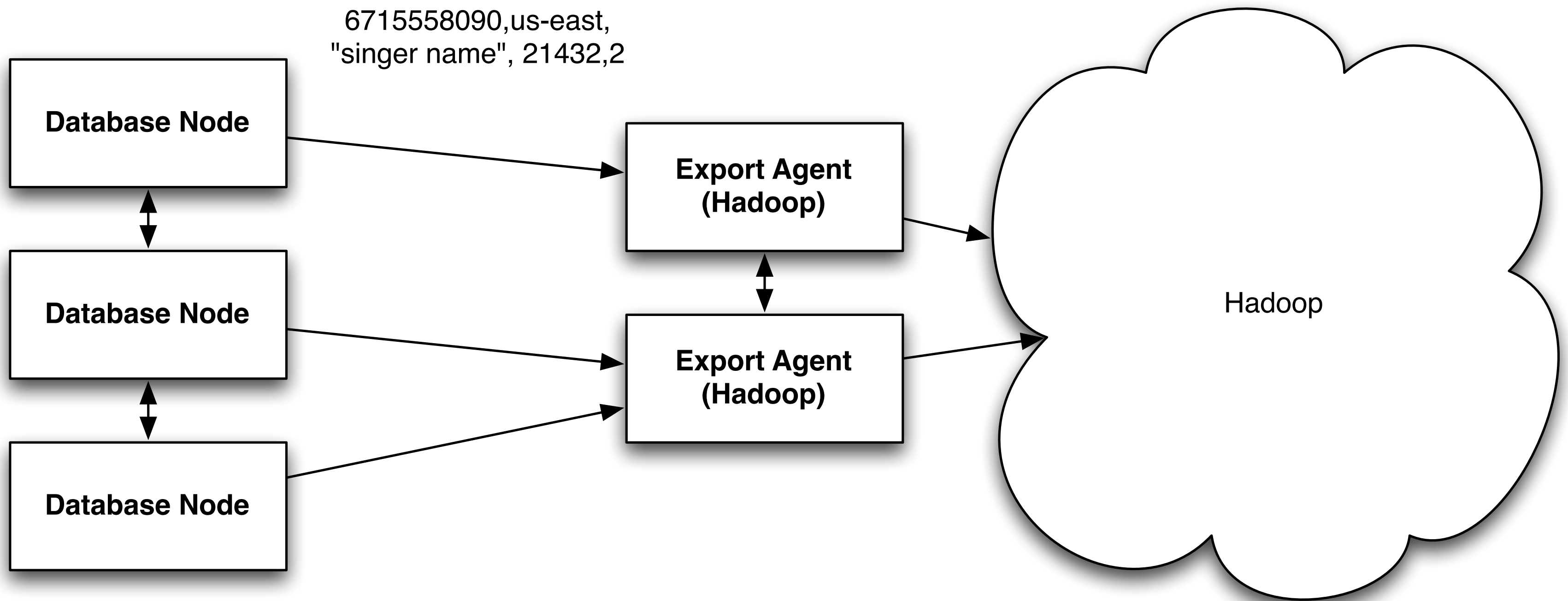
phone #, region, contestant, point-in-time contestant score, vote # for phone #

If this is an even thousandth vote for a contestant, also export:

contestant, point-in-time contestant score

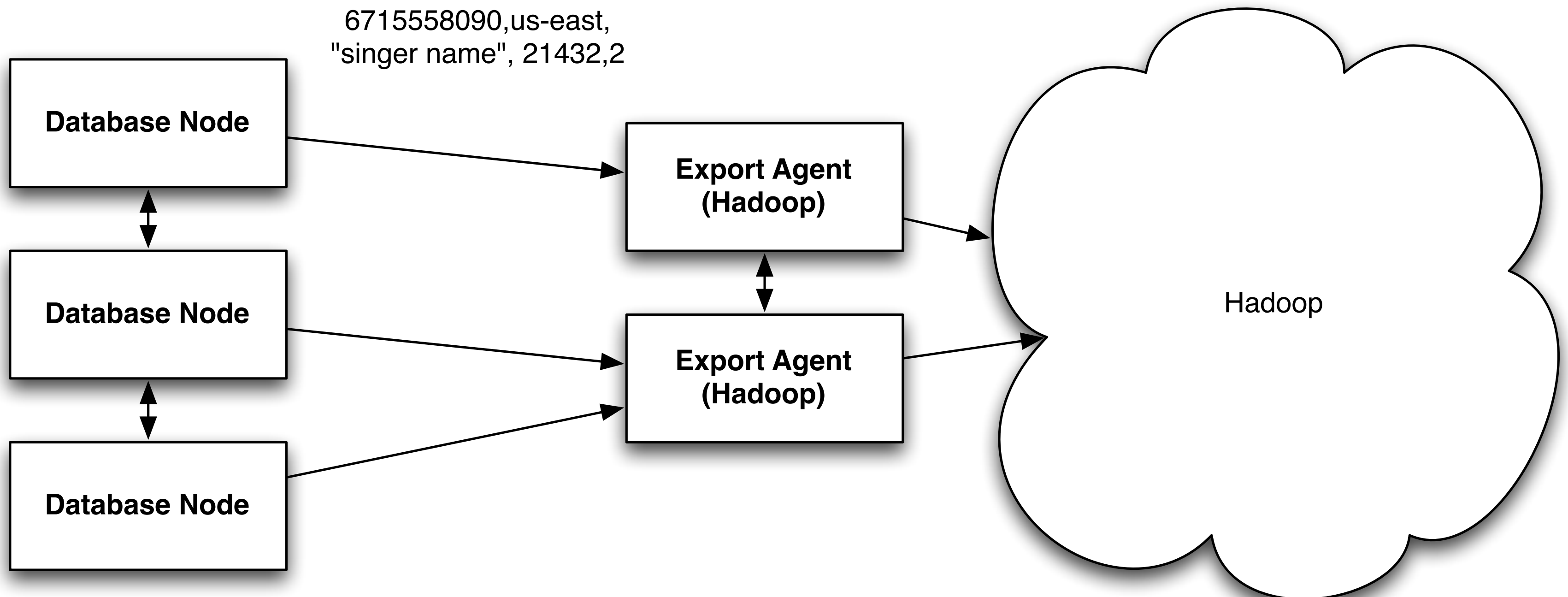
END TXN

Export Agent's Job



- Understand database cluster topology.
  + Respond to failures
  + De-dupe or otherwise enrich export data
- Understand target system. When is data "committed"?
- Control the flow and lifecycle of data between them.

Export Client's Job



- Loosely-coupled, asynchronous
- Data inserted into export tables is durable across failures.
- Minimize latency to export clients. Downstream varies…

Conclusions

- Sometimes it's Velocity – not Petabytes
- Data tiers will increasingly require more than one engine
  - High velocity, transactional engine for "fast" data
  - Analytic engine for "deep" data
- Value in real-time analysis of write intensive input
- Full consistency can make development at scale easier

Workshop: Tonight @ 5:30. Food. See flyer.

jhugg@voltdb.com http://voltdb.com @voltdb

Questions & Answers

Thanks!