Goals of this Talk

- Why transactions?
- Optimistic Concurrency Control
- Three Apache projects: Omid, Tephra, Trafodion
- How are they different?
Transactions in noSQL?

History
• SQL: RDBMS, EDW, …
• noSQL: MapReduce, HDFS, HBase, …
• n(ot)o(nly)SQL: Hive, Phoenix, …

Motivation:
• Data consistency under highly concurrent loads
• Partial outputs after failure
• Consistent view of data for long-running jobs
• (Near) real-time processing
Stream Processing

Diagram showing components of a stream processing system, including Queue, Flowlet, HBase Table, and other related elements.
Write Conflict!

Queue ... Flowlet ... HBase Table ... Flowlet ...
Transactions to the Rescue

- Atomicity of all writes involved
- Protection from concurrent update
ACID Properties

From good old SQL:

- **Atomic** - Entire transaction is committed as one
- **Consistent** - No partial state change due to failure
- **Isolated** - No dirty reads, transaction is only visible after commit
- **Durable** - Once committed, data is persisted reliably
What is HBase?
What is HBase?

Simplified:

• Distributed Key-Value Store
• Key = <row>.<family>.<column>.<timestamp>
• Partitioned into Regions (= continuous range of rows)

• Each Region Server hosts multiple regions
• Optional: Coprocessor in Region Server

• Durable writes
ACID Properties in HBase

• **Atomic**
  • At cell, row, and region level
  • Not across regions, tables or multiple calls

• **Consistent** - No built-in rollback mechanism

• **Isolated** - Timestamp filters provide some level of isolation

• **Durable** - Once committed, data is persisted reliably

How to implement full ACID?
Implementing Transactions

• Traditional approach (RDBMS): locking
  • May produce deadlocks
  • Causes idle wait
  • complex and expensive in a distributed env

• Optimistic Concurrency Control
  • lockless: allow concurrent writes to go forward
  • on commit, detect conflicts with other transactions
  • on conflict, roll back all changes and retry

• Snapshot Isolation
  • Similar to repeatable read
  • Take snapshot of all data at transaction start
  • Read isolation
Optimistic Concurrency Control

client1: start  x=10  fail/rollback

client2: start  read x  commit

must see the old value of x
Optimistic Concurrency Control

Client 1: start

x = 10

Incr x

Commit

Client 2: start

Incr x

Sees the old value of x = 10

Rollback

Commit

x = 11

time
Conflicting Transactions

tx:A

- tx:B
  - tx:C (A fails)
  - tx:D (A fails)
  - tx:F (F fails)

- tx:E (E fails)
  - tx:G

- tx:A

- ...

- time
Conflicting Transactions

• Two transactions have a conflict if
  • they write to the same cell
  • they overlap in time

• If two transactions conflict, the one that commits later rolls back

• Active change set = set of transactions $t$ such that:
  • $t$ is committed, and
  • there is at least one in-flight tx $t'$ that started before $t$’s commit time

• This change set is needed in order to perform conflict detection.
HBase Transactions in Apache

Apache Tephra (incubating)

Apache Omid (incubating)

Apache Trafodion (incubating)
In Common

• Optimistic Concurrency Control must:
  • maintain Transaction State:
    • what tx are in flight and committed?
    • what is the change set of each tx? (for conflict detection, rollback)
    • what transactions are invalid (failed to roll back due to crash etc.)
  • generate unique transaction IDs
  • coordinate the life cycle of a transaction
    • start, detect conflicts, commit, rollback

• All of { Omid, Tephra, Trafodion } implement this
  • but vary in how they do it
Apache Tephra

- Based on the original Omid paper:
  Daniel Gómez Ferro, Flavio Junqueira, Ivan Kelly, Benjamin Reed, Maysam Yabandeh: *Omid: Lock-free transactional support for distributed data stores*. ICDE 2014.

- **Transaction Manager:**
  - Issues unique, monotonic transaction IDs
  - Maintains the set of excluded (in-flight and invalid) transactions
  - Maintains change sets for active transactions
  - Performs conflict detection

- **Client:**
  - Uses transaction ID as timestamp for writes
  - Filters excluded transactions for isolation
  - Performs rollback
Transaction Lifecycle

- Transaction consists of:
  - transaction ID (unique timestamp)
  - exclude list (in-flight and invalid tx)

- Transactions that do complete
  - must still participate in conflict detection
  - disappear from transaction state when they do not overlap with in-flight tx

- Transactions that do not complete
  - time out (by transaction manager)
  - added to invalid list
Apache Tephra

Client A

HBase

Region Server

Region Server

Tx Manager

start()
id: 42, excludes = {...}

write: x=11

write: y=17

in-flight: ...

id: 42, excludes = {...}
Apache Tephra

**Client B**

- `start()`
- `id: 48, excludes = {…,42}`

**HBase**

- `read x`

**Region Server**

- `x:10`
- `x:11`
- `37`
- `42`

**Region Server**

- `y:17`
- `42`

**Tx Manager**

- `in-flight: …,42,48`
Apache Tephra

Client A

commit() conflict

Tx Manager

make visible

roll back

HBase

Region Server

x:10 37

Region Server

in-flight: ...
Apache Tephra

Client A

commit()

success

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Tx Manager

in-flight:

...,52

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}

HBase

read x

Region Server

Region Server

x:10 37
x:11 42

Client C

in-flight:

...,42

Client A

commit()

success

Tx Manager

in-flight:

...,52

In-flight:

...,42

Client C

start()

id: 52, excludes: {...}
Apache Tephra

HBase

Region Server

Coprocessor

Region

... Region

Client

Tx lifecycle rollback

lifecycle transitions

data operations

Tx state Tx id generation

Region Server

Coprocessor

Region

... Region

data operations

operations

HBase

Region Server

Coprocessor

Region

... Region
Apache Tephra

• HBase coprocessors
  • For efficient visibility filtering (on region-server side)
  • For eliminating invalid cells on flush and compaction

• Programming Abstraction
  • TransactionalHTable:
    • Implements HTable interface
    • Existing code is easy to port
  • TransactionContext:
    • Implements transaction lifecycle
Apache Tephra - Example

txTable = new TransactionAwareHTable(table);
txContext = new TransactionContext(txClient, txTable);
txContext.start();
try {
    // perform Hbase operations in txTable
    txTable.put(...);
    ...
} catch (Exception e) {
    // throws TransactionFailureException(e)
    txContext.abort(e);
}
// throws TransactionConflictException if so
txContext.finish();
Apache Tephra - Strengths

• Compatible with existing, non-tx data in HBase
• Programming model
  • Same API as HTable, keep existing client code
• Conflict detection granularity
  • Row, Column, Off
  • Special “long-running tx” for MapReduce and similar jobs
• HA and Fault Tolerance
  • Checkpoints and WAL for transaction state, Standby Tx Manager
• Replication compatible
  • Checkpoint to HBase, use HBase replication
• Secure, Multi-tenant
Apache Tephra - Not-So Strengths

- Exclude list can grow large over time
  - RPC, post-filtering overhead
  - Solution: Invalid tx pruning on compaction - complex!
- Single Transaction Manager
  - performs all lifecycle state transitions, including conflict detection
  - conflict detection requires lock on the transaction state
  - becomes a bottleneck
  - Solution: distributed Transaction Manager with consensus protocol
Apache Trafodion

• A complete distributed database (RDBMS)
  • transaction system is not available by itself
  • APIs: jdbc, SQL

• Inspired by original HBase TRX (transactional region server)
  • migrated transaction logic into coprocessors
  • coprocessors cache in-flight data in-memory
  • transaction state (change sets) in coprocessors
  • conflict detection with 2-phase commit

• Transaction Manager
  • orchestrates transaction lifecycle across involved region servers
  • multiple instances, but one per client
Apache Trafodion

Transaction Manager Log
Meta info/state about transaction (more than one region in a txn) HBase table(s)

Transaction Manager
- DTM Core
- Branch I/F
- Transaction Management JNI
- TransactionManager
- TrxRegionEndpoint coprocessor client

HBase RegionServer
- HRegion
- HRegion
- HRegion

HBase Write Ahead Log
Region level transaction info

Transaction Management JNI
- register(region, tx)

Transaction Manager Library
- begin/commit/abort(tx)

Resource Manager Library
- get/put/delete
- operations
- Transactional Table
- Transactional Scanner
- TrxRegionEndpoint coprocessor client

SQL Transaction initiating process
- SQL JNI

transactional aggregation(tx)

Java

C++

get/put/delete(tx)
open/perform/close scanner(tx)

Coprocessors implemented
TrxRegionEndpoint implements SQL transactions
TrxRegionObserver implements recovery process

Each region server has its own HLOG

HLOG

HRegion

© Copyright 2014 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice.
Apache Trafodion

Client A

Tx Manager

start()
id:42

region:

write
x=11
write
y=17

HBase

Region Server

Region Server

x:11

x:10

y:17
Apache Trafodion

Client B

start()
id: 48

x:10

HBase

read x

Region Server

x:10

x:11

Region Server

y:17

Tx Manager

in-flight:

…, 42, 48
Apache Trafodion

Client A

commit()

Tx Manager

in-flight:

…

HBase

Region Server

x:10

Region Server

1. conflicts?
2. roll back

33
Apache Trafodion

1. conflicts?
2. commit!

HBase

Region Server x:11

Region Server y:17
Apache Trafodion

HBase

Region Server
Coprocessor
Region
Region

Region Server
Coprocessor
Region
Region

Client 2
Tx 2 Manager
Client
Tx Manager

Tx lifecycle
transitions
region ids

Tx life cycle (commit)
Tx id generation

data
operations

In-flight data
Tx state
conflicts

2-phase
commit

Tx id generation

Client
Tx Manager

region ids

35
Apache Trafodion

• Scales well:
  • Conflict detection is distributed: no single bottleneck
  • Commit coordination by multiple transaction managers
  • Optimization: bypass 2-phase commit if single region

• Coprocessors cache in-flight data in Memory
  • Flushed to HBase only on commit
  • Committed read (not snapshot, not repeatable read)
  • Option: cause conflicts for reads, too

• HA and Fault Tolerance
  • WAL for all state
  • All services are redundant and take over for each other

• Replication: Only in paid (non-Apache) add-on
Apache Trafodion - Strengths

• Very good scalability
  • Scales almost linearly
  • Especially for very small transactions
• Familiar SQL/jdbc interface for RDB programmers
• Redundant and fault-tolerant
• Secure and multi-tenant:
  • Trafodion/SQL layer provides authn+authz
Apache Trafodion - Not-So Strengths

• Monolithic, not available as standalone transaction system
• Heavy load on coprocessors
  • memory and compute
• Large transactions (e.g., MapReduce) will cause Out-of-memory
  • no special support for long-running transactions
Apache Omid

• Evolution of Omid based on the Google Percolator paper:

• Idea: Move as much transaction state as possible into HBase
  • Shadow cells represent the state of a transaction
  • One shadow cell for every data cell written
  • Track committed transactions in an HBase table
  • Transaction Manager (TSO) has only 3 tasks
    • issue transaction IDs
    • conflict detection
    • write to commit table
Apache Omid

Transactional App

Omid Client

Start/Commit TXs

Transaction Status Oracle (TSO)

Get Start/Commit Timestamps

Timestamp Oracle

R/W data

HBase

App Table

Shadow Cells

Commit Table

Compactor

Guarantee SI

Keep track & Validate TXs

Commit data
Apache Omid

Client A

Tx Manager

write x=11

write: y=17

HBase

Region Server

Region Server

Commits

start()

id: 42

write

x=11

write:
y=17

x:10 37: commit.40
x:11 42: in-flight

y:17 42: in-flight

37: 40
Apache Omid

HBase

Region Server

- x:10 37: commit.40
- x:11 42: in-flight

Commits

- 37: 40

Region Server

- y:17 42: in-flight

Client B

start()

id: 48

Tx Manager

x:10

read x
Apache Omid

Client A
commit() conflict
roll back

HBase

Region Server
x:10 37: commit.40

Region Server

Region Server

Commits
37: 40

Tx Manager

Apache Omid

Client A
commit() conflict
roll back

HBase

Region Server
x:10 37: commit.40

Region Server

Region Server

Commits
37: 40

Tx Manager
Apache Omid

Client A

Client C

Region Server

Region Server

Commits

HBase

commit() success:50

mark as committed

read x

start() id: 52

x:11

y:17 42: commit.50

x:10 37: commit.40

x:11 42: commit.50

37: 40

42: 50
Apache Omid - Future

• Atomic commit with linking?
  • Eliminate need for commit table

HBase
Apache Omid

Client

Tx lifecycle
rollback
commit

start
commit

Tx Manager

Conflict detection
Tx id generation

data

operations
+ shadow cells

HBase

Region Server

Coprocessor

Region

...region

Region Server

Coprocessor

Region

...region

Tx state

commit

table

data

operations
+ shadow cells

Region

...region

Tx state

commit

table

Region

...region
Apache Omid - Strengths

• Transaction state is in the database
  • Shadow cells plus commit table
  • Scales with the size of the cluster
• Transaction Manager is lightweight
  • Generation of tx IDs delegated to timestamp oracle
  • Conflict detection
  • Writing to commit table
• Fault Tolerance:
  • After failure, fail all existing transactions attempting to commit
  • Self-correcting: Read clients can delete invalid cells
Apache Omid - Not So Strengths

• Storage intensive - shadow cells double the space
• I/O intensive - every cell requires two writes
  1. write data and shadow cell
  2. record commit in shadow cell
• Reads may also require two reads from HBase (commit table)
• Producer/Consumer: will often find the (uncommitted) shadow cell
  • Scans: high throughput sequential read disrupted by frequent lookups
• Security/Multi-tenancy:
  • All clients need access to commit table
  • Read clients need write access to repair invalid data
• Replication: Not implemented
## Summary

<table>
<thead>
<tr>
<th></th>
<th>Apache Tephra</th>
<th>Apache Trafodion</th>
<th>Apache Omid</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tx State</strong></td>
<td>Tx Manager</td>
<td>Distributed to region servers</td>
<td>Tx Manager (changes)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>HBase (shadows/commits)</td>
</tr>
<tr>
<td><strong>Conflict detection</strong></td>
<td>Tx Manager</td>
<td>Distributed to regions, 2-phase commit</td>
<td>Tx Manager</td>
</tr>
<tr>
<td><strong>ID generation</strong></td>
<td>Tx Manager</td>
<td>Distributed to multiple Tx Managers</td>
<td>Tx Manager</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>HTable</td>
<td>SQL</td>
<td>Custom</td>
</tr>
<tr>
<td><strong>Multi-tenant</strong></td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Strength</strong></td>
<td>Scans, Large Tx, API</td>
<td>Scalable, full SQL</td>
<td>Scale, throughput</td>
</tr>
<tr>
<td><strong>So so</strong></td>
<td>Scale, Throughput</td>
<td>API not Hbase, Large Tx</td>
<td>Scans, Producer/Consumer</td>
</tr>
</tbody>
</table>
Links

Join the community:

Apache Tephra (incubating)
http://tephra.apache.org/

Apache Omid (incubating)
http://omid.apache.org/

Apache Trafodion (incubating)
http://trafodion.apache.org/
Thank you

… for listening to my talk.

Credits:
- Sean Broeder, Narendra Goyal (Trafodion)
- Francisco Perez-Sorrosal (Omid)

Questions?