Big Data Technology
Incremental Processing using Distributed Transactions

Eshcar Hillel
Yahoo!

Ronny Lempel
Outbrain
Roadmap

- Previous classes
  - Stream processing

- This class
  - Sieve/Percolator – content management platform
  - Omid - Distributed transaction processing
  - Notifications mechanism
Search and Technology Evolution

Late 1990’s

Relatively static Web
Okay to have new content searchable within hours or days
Crawl the Web. Store content. Index in big batches.
Technology: Map-Reduce.

Late 2000’s

Super-dynamic Web, Mobile, Social
Content expected to become searchable within minutes
Ingest multiple content streams. Index instantly.
Search in the Dynamic World

- Crawl feed
- Mail feed
- Social feed

Content Processing

Indexing Pipeline

Real-Time Search Index

Serving

Queries
Zooming In on Indexing Pipelines

Process a stream of documents arriving uncoordinated
Produce a collection of artifacts (features) for each document
... required for fast retrieval and relevance ranking
Each document is processed independently
... processing might affect other documents (example?)
Traverse a series of topologies (dataflow)
... intermediate results stored for reliability
Sieve Architecture

Crawl, Docproc, Link Analysis, and Stream

Crawl schedule, Content, Links, Queue

Transaction and Processing

Apache Omid (Consistency), Apache HBase (storage), Scalable distributed Hadoop KV-store, Apache Storm (compute), Scalable distributed stream processing platform

CS236620 Big Data Technology
Percolator Architecture

Technologies

- **Stream processing**: series of observers each completes a task, and creates more work to downstream observers
  - Storm topologies

- **Data storage**: Key-Value Stores (NoSQL) are a natural fit
  - Bigtable (Google/proprietary)
  - HBase (Apache/open source)

- **End-to-end consistency**: transactional processing layer
  - Percolator (Google/proprietary)
  - Omid (Apache/open source)
Why Transactions?

- Multiple processing threads transform the repository concurrently.
- Transactions make it easier for the programmer to reason about the state of the repository:
  - Both in terms of fault tolerance (atomicity) and consistency (isolation).
- Better performance than ad-hoc (incorrect?) solutions.
Back to Web Indexing - Zooming In on Tasks

Document processing

- **Read** page content from the store
- Compute search index features
- **Update** computed features

De-duplicate content

- **Read** page content from the store
- Compute clustering
- **Update** primary duplicate for all pages in the cluster

Link processing

- **Read** outgoing links for a page
- **Update** reference for all linked-to pages
ACID Transactions

**Multiple** data accesses in a **single** logical operation

**Atomic**

“All or nothing” – no partial effect observable

**Consistent**

The DB transitions from one valid state to another

**Isolated**

Appear to execute in isolation

**Durable**

Committed data cannot disappear
Isolation Take I – Serializability

**Transactions appear to execute serially**
Each one logically happens at a single point in time

Popular implementation – **2-phase commit**
prepare phase - acquire locks, read set validation
commit phase – commit, update values, release locks
Critique and Remedy

Serializability is Deadlock/Livelock-prone
Serializability is Expensive

Up to 50% time wasted on locking (bad utilization)
Why pay for consistency in low-contention environments?

Remedy: Exploit Multi-Version Concurrency Control (MVCC)

Let concurrent operations work on multiple data versions
Execute optimistically, reconcile at the end

Fortunately, MVCC is implemented by modern KV-stores
Isolation Take II – Snapshot Isolation

**Two** synchronization points

Read snapshot (RS) – all reads consistent for RS
Write snapshot (WS) – all writes applied at WS (WS > RS)

```
begin  read(x)  write(y)  write(x)  read(z)  commit
```

Read point  Write point

Isolation guarantee: **no WW conflicts**

No external writes to T’s write set between T.RS and T.WS
→ Read-only transactions always succeed
Snapshot Isolation Example

<table>
<thead>
<tr>
<th>TxId</th>
<th>Temporal Overlap</th>
<th>Spatial Overlap (Write Set)</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td>K1, K3</td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td>K2, K4</td>
</tr>
<tr>
<td>T3</td>
<td></td>
<td>K3, K4</td>
</tr>
<tr>
<td>T4</td>
<td></td>
<td>K1, K2, K3, K4</td>
</tr>
</tbody>
</table>

TxId: Transaction ID
Temporal Overlap: Timeline of transactions
Spatial Overlap (Write Set): Write sets for each transaction.
Snapshot Isolation - Implementation

**Begin**

\[ RS \leftarrow \text{timestamp()} \] // RS greater than all committed txs

**Read**\( (X) \)

Read the latest version with timestamp \( \leq \) RS

**Write**

Tentatively mark write indication

**Commit**

Validate no conflicts; **abort** if validation fails

Writes take effect (atomically with validation)
Distributed Transaction Processing

- Many production systems
- E.g., Google’s Percolator and Spanner
  - Both implemented on top of BigTable
  - Both apply 2-phase commit
  - Percolator uses TS oracle for SI
  - Spanner supports serializability across data centers, uses atomic clocks
Omid –
Transactions for Key-Value Store

- **Apache incubator project**, incepted by Yahoo
- Transactional API over Hbase
  - Can be used directly or by SQL-over-HBase platforms (Phoenix)
- Scalable, Highly Available Service
  - 380K tps
- Database agnostic Architecture
  - Works with any MVCC-friendly KV-store
Omid Architecture
Running Example
Running Example

Client API

Transaction Manager

Data store (k1, v1, t1)

Data store

Data store (k2, v2, t1)

Commit table

Read(k', last committed t' < t1)

Get commit info(t')
Running Example

Transaction Manager

Client

API

API

Data store

(k1,v1,t1)

Data store

(k2,v2,t1)

Data store

Commit table

Commit(t1, {k1,k2})

Atomic commit(t1, t2)

(t2)
Running Example

Client

API

Transaction Manager

Commit(t1, {k1,k2})

t2

Write(k1,v1,t1,t2)

Write(k2,v2,t1,t2)

Remove (t1)

Data store (k1,v1,t1,t2)

Data store (k2,v2,t1,t2)

Commit table
High Availability

Primary-backup solution
Guarantee clients get the right answer
Clients write data only when their TSO is the leader
HA: Primary-Backup Transaction Manager

Client → Recovery state (ZK) → Transaction Manager

Backup
Primary

Data
Data
Data
Commit Table
Split Brain

Race Conditions Violate SI

Transaction Manager

Fence CT upon every write (slow!)

Backup

Primary

Client

Commit Table
High Availability – Key Ideas

Client detects inconsistencies, invalidate pending txs from previous epochs

Lease-based leader election
- Optimization: Check locally before and after write to the CT
- Zero overhead in non-fault scenarios
Scalability

The TSO and the CT scale independently

CT Scalability

Commit writes aggressively batched for throughput
Concurrent write to Hbase – horizontal scalability

TSO Scalability

Conflict detection the main bottleneck
Novel concurrent algorithm, scales vertically to 5M txn/sec
Sieve Notifications

- Stream processing workers scan data to identify changes
- Invoke corresponding observer (topology)
  - Perform txs
  - Store intermediate results
- Again, workers identify changes
  - ...triggers downstream observers
Notification and Acknowledgment Columns

Dirty column – notification
- set if observers must be triggered
- Very sparse columns

Randomized distributed scan over notification column
- in-memory columns

<table>
<thead>
<tr>
<th>Key</th>
<th>...</th>
<th>Val</th>
<th>n</th>
<th>ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>k184637</td>
<td></td>
<td>xxx</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>k298493</td>
<td></td>
<td>xxx</td>
<td>1</td>
<td>ts:yyy</td>
</tr>
<tr>
<td>k827164</td>
<td></td>
<td>xxx</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Bus Clumping

Randomized scanner tend to clump
- Reduces effective parallelism
- Creates contention in kv-store servers

Solution
- Obtain a lightweight lock per scanned row
- Upon failure, jump to a random point in table
Summary

- Transactions – important use case in stream processing
- Snapshot Isolation – scalable consistency model
- Omid – an open-source TP system for Hbase
- Battle-tested, Web-scale, HA
Further Reading

- Percolator (OSDI ‘10)
- Omid (FAST ‘17)
Next Class

- A/B testing