Percolator

Large-Scale Incremental Processing using Distributed Transactions and Notifications

D. Peng & F. Dabek
Motivation

- Built to maintain the Google web search index
- Need to maintain a large repository, but cost of updates should be independent of size
- Must be able to maintain data invariants
- Must scale horizontally (number of machines)
- Must be easy for a programmer to reason about concurrency
Non-Goals

- Not required to be real-time (latency doesn’t matter)
- Availability secondary to scalability
EXISTING METHODS
MapReduce?

- Hides concurrency from the programmer
- Scales and performs very well

But...

- No incremental updates – all output is thrown away when the repository is updated
- Cost of update is proportional to size of the repository
Traditional DBMS?

• Provides incremental updates
• Concurrency is a non-issue for programmer
• Provides transactions and data constraints

But...

• Scalability (CPU, storage) usually limited to a few hosts
• Cannot handle larger data sets (tens of petabytes)
BigTable

• Distributed DBMS
• Very scalable in terms of throughput and storage using many machines
• Provides incremental updates
But...
• No transactions across rows – hard to maintain data invariants
• Provides few abstractions – essentially just structured storage
BigTable

• Dataset split into blocks
• Each block mapped to a BigTable ‘tablet’
• BigTable uses GFS for storage (high availability)
• Provides simple transactions and consistency guarantees for single rows
• Keeps revision history of individual cells
What is Percolator?

• Large-Scale system for incremental processing
• Built on top of a BigTable distributed database
• Each machine in cluster runs:
  – GFS Chunkserver
  – BigTable tablet server
  – Percolator worker
• Provides useful abstractions to the programmer:
  – ‘Observers’
  – Multi-row transactions
Overview

• Data is written to BigTable by external process
• Programmer chooses columns for Percolator to watch
• Changes to watched columns trigger small user programs – ‘observers’
• These observers can then update other parts of the dataset, triggering further observers
• Each observer can atomically update multiple fields (all or nothing)
Observers

- Piece of user code with conditions for when to run it
- A column is added for every watched column in a table, which is set when the a cell is updated
- Each Percolator worker continuously scans random subsets of the repository for changes
- Runs observer if flag is set, then resets flag
Transactions

- A single observer is likely to modify values across several rows/tables, how to guarantee consistency?
- Want to provide ‘Snapshot Isolation Semantics’ – easy to reason about
- Need a high-performance locking service
  – BigTable fits
- Transactions ensure that if two observers touch the same rows, only one can commit
Two-Phase Commit

- Mechanism for coordinating writes
- Do all writes in two stages – lock and commit
- Try to acquire lock on every row
- Iff all locks succeed, then commit changes
- Lock and write are implemented in one operation using BigTable *time dimension*
- Commit updates current data pointer
Two-Phase Commit (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>T</th>
<th>bal:data</th>
<th>bal:lock</th>
<th>bal:write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>£10</td>
</tr>
<tr>
<td>Joe</td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td>£2</td>
</tr>
</tbody>
</table>
## Two-Phase Commit (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>T</th>
<th>bal:data</th>
<th>bal:lock</th>
<th>bal:write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>7</td>
<td>£3</td>
<td>LOCK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Two-Phase Commit (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>T</th>
<th>bal:data</th>
<th>bal:lock</th>
<th>bal:write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>7</td>
<td>£3</td>
<td>LOCK</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>7</td>
<td>£9</td>
<td>LOCK @ Bob.bal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Two-Phase Commit (cont’d)

<table>
<thead>
<tr>
<th>Key</th>
<th>T</th>
<th>bal:data</th>
<th>bal:lock</th>
<th>bal:write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bob</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>£3</td>
<td></td>
<td>data @ 7</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>7</td>
<td>£9</td>
<td>LOCK @ Bob.bal</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Key</td>
<td>T</td>
<td>bal:data</td>
<td>bal:lock</td>
<td>bal:write</td>
</tr>
<tr>
<td>-----</td>
<td>---</td>
<td>---------</td>
<td>---------</td>
<td>----------</td>
</tr>
<tr>
<td>Bob</td>
<td>8</td>
<td></td>
<td></td>
<td>data @ 7</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>£3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Joe</td>
<td>8</td>
<td></td>
<td></td>
<td>data @ 7</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>£9</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td>data @ 5</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>£2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Transactions (cont’d)

• Each transaction has a start and commit timestamp requested from an oracle
• For writes:
  – if transaction sees another write after its start timestamp it aborts
  – if transaction sees another lock at any timestamp it aborts
  – otherwise, initiate Two-Phase commit
• For reads:
  – if locks prior to start timestamp, block
  – otherwise read the data value corresponding to the latest write record
• Synchronising on primary allows any client to clean any lock
EVALUATION
Evaluation Method

• Compared to MapReduce for web index
• Overhead of Percolator relative to BigTable
• Synthetic benchmarks (TPC-E)
• Resilience to failure
Percolator vs MapReduce

Remove duplicates from a billion document repository by clustering. Clustering is from a random value. Average cluster has 3.3 documents. Percolator processes fewer documents per unit time, but only processes updated documents.
Percolator vs BigTable

Worst case for Percolator - constant number of operations for commit.
Write incurs 3 RPC operations for lock handling.
Additional overhead comes from extra metadata.
Percolator vs DBMS

TPC-E is a widely recognized DBMS benchmark which simulates a brokerage firm with customers who perform trades, market search and account enquiries.

Current single-host record is 3183 TPS, Percolator is less efficient per host, but scales better.
Percolator vs Failures

Kill a third of the 15 BigTable tablet servers and allow them to restart. Performance drops by approximately 1/3.
Tradeoffs

• Blocking API calls
  – Makes programming easier
  – Individually, worker processes are limited to a single task
  – Achieve parallelism by running many workers

• Sacrifice performance for near-linear scalability

• Programmer must handle transactions

• Sacrifices availability for stronger consistency
Future Work

• Percolator overhead is 30x that of DBMS for a single host
  – What overhead is fundamental to distributed storage systems?
  – What is due to inefficiencies in the design?

• Percolator sacrifices availability (e.g. cross-datacentre replication) for consistency
  – Is this necessary?
FIN
Questions?