Distributed Transactions in Hadoop's HBase and Google's Bigtable

Hans De Sterck
Department of Applied Mathematics
University of Waterloo

Chen Zhang
David R. Cheriton School of Computer Science
University of Waterloo
this talk:

- how to deal with very large amounts of data in a distributed environment (‘cloud’)

- in particular, data consistency for very large sets of data items that get modified concurrently
example: Google search and ranking

- continuously ‘crawl’ all webpages, store them all on 10,000s of commodity machines (CPU+disk), petabytes of data
- every 3 days (*the old way...), build search index and ranking, involves inverting outlinks to inlinks, Pagerank calculation, etc., via MapReduce distributed processing system, on the same set of machines
- very large database-like storage system is used (Bigtable)
- build a $190billion internet empire on fast and useful search results
1. cloud computing
2. Google/Hadoop cloud frameworks
3. motivation: large-scale scientific data processing
4. Bigtable and HBase
5. transactions
6. our transactional system for HBase
7. protocol
8. advanced features
9. performance
10. comparison with Google’s percolator
11. conclusions
1. cloud computing

- cloud = set of resources for distributed computing and storage (networked) characterized by
  1. homogeneous environment (often through virtualization)
  2. dedicated (root) access (no batch queues, not shared, no reservations)
  3. scalable on demand (or large enough)

- cloud = ‘grid’, simplified such that it becomes more easily doable (e.g., it is hard to combine two (different) clouds!)
cloud computing

• cloud is
  1. homogeneous
  2. dedicated access
  3. scalable on demand

• we are interested in cloud for large-scale, serious computing (not: email and wordprocessing, ...) (but: cloud also useful for business computing, e-commerce, storage, ...)

• there are private (Google internal) and public (Amazon) clouds (it is hard to do hybrid clouds!)
2. Google/Hadoop cloud frameworks

- **cloud** (homogeneous, dedicated access, scalable on demand) used for large-scale computing/data processing needs ‘cloud framework’

- Google: first, and most successful (private) cloud (framework) for large-scale computing to date
  - Google File System
  - Bigtable
  - MapReduce
Google cloud framework

- first, and most successful (private) cloud (framework) for large-scale computing to date
  - Google File System:
    - fault-tolerant, scalable distributed file system
  - Bigtable
    - fault-tolerant, scalable sparse semi-structured data store
  - MapReduce
    - fault-tolerant, scalable parallel processing system

- used for search/ranking, maps, analytics, ...
Hadoop clones Google’s system

• Hadoop is an open-source clone of Google’s cloud computing framework for large-scale data processing
  ▪ Hadoop File System (HFS) (Google File System)
  ▪ HBase (Bigtable)
  ▪ MapReduce

• Hadoop is used by Yahoo, Facebook, Amazon, ... (and developed/controlled by them)
Hadoop usage

(from wiki.apache.org/hadoop/PoweredBy)

Yahoo!
- More than 100,000 CPUs in >36,000 computers running Hadoop
- Our biggest cluster: 4000 nodes (2*4cpu boxes w 4*1TB disk & 16GB RAM)
  - Used to support research for Ad Systems and Web Search
  - Also used to do scaling tests to support development of Hadoop on larger clusters
- Our Blog - Learn more about how we use Hadoop.
- >60% of Hadoop Jobs within Yahoo are Pig jobs.
Hadoop usage

Twitter
- We use Hadoop to store and process tweets, log files, and many other types of data generated across Twitter. We use Cloudera’s CDH2 distribution of Hadoop, and store all data as compressed LZO files.
- We use both Scala and Java to access Hadoop's MapReduce APIs.
- We use Pig heavily for both scheduled and ad-hoc jobs, due to its ability to accomplish a lot with few statements.
- We employ committers on Pig, Avro, Hive, and Cassandra, and contribute much of our internal Hadoop work to opensource (see hadoop-lzo)
- For more on our use of hadoop, see the following presentations: Hadoop and Pig at Twitter and Protocol Buffers and Hadoop at Twitter

LinkedIn
- We have multiple grids divided up based upon purpose. They are composed of the following types of hardware:
  - 100 Nehalem-based nodes, with 2x4 cores, 24GB RAM, 8x1TB storage using ZFS in a JBOD configuration on Solaris.
  - 120 Westmere-based nodes, with 2x4 cores, 24GB RAM, 6x2TB storage using ext4 in a JBOD configuration on CentOS 5.5
- We use Hadoop and Pig for discovering People You May Know and other fun facts.
Hadoop usage

Facebook
- We use Hadoop to store copies of internal log and dimension data sources and use it as a source for reporting/analytics and machine learning.
- Currently we have 2 major clusters:
  - A 1100-machine cluster with 8800 cores and about 12 PB raw storage.
  - A 300-machine cluster with 2400 cores and about 3 PB raw storage.
  - Each (commodity) node has 8 cores and 12 TB of storage.
- We are heavy users of both streaming as well as the Java apis. We have built a higher level data warehousing framework using these features called Hive (see the [http://hadoop.apache.org/hive/](http://hadoop.apache.org/hive/)). We have also developed a FUSE implementation over hdfs.

EBay
- 532 nodes cluster (8 * 532 cores, 5.3PB).
- Heavy usage of Java MapReduce, Pig, Hive, HBase
- Using it for Search optimization and Research.
3. motivation: large-scale scientific data processing

- my area of research is scientific computing (scientific simulation methods and applications)
- large-scale computing / data processing
- we have used ‘grid’ for distributed ‘task farming’ of bioinformatics problems (BLSC 2007)
motivation: large-scale scientific data processing

• use Hadoop/cloud for large-scale processing of biomedical images (HPCS 2009)

Case Study of Scientific Data Processing on a Cloud Using Hadoop

Chen Zhang¹, Hans De Sterck², Ashraf Aboulnaga¹, Haig Djambazian³, and Rob Sladek³

process 260GB of image data per day
motivation: large-scale scientific data processing

- workflow system using HBase (Cloudcom 2009)

- batch system using HBase (Cloudcom 2010)

problem: HBase does not have multi-row transactions
motivation: large-scale scientific data processing

cloud computing (processing) will take off for
- biomedical data (images, experiments)
- bioinformatics
- particle physics
- astronomy
- etc.
4. Bigtable and HBase

- relational DBMS (SQL) are not (currently) suitable for very large distributed data sets:
  - not parallel
  - not scalable
  - relational ‘sophistication’ not necessary for many applications, and these applications require efficiency for other aspects (scalable, fault-tolerant, throughput versus response time, ...)
... Google invents Bigtable

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

(OSDI 2006)
A Bigtable is a **sparse**, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, **column** key, and a timestamp; each value in the map is an uninterpreted array of bytes.

- sparse tables
- multiple data versions – timestamps
- scalable, fault-tolerant (tens of petabytes of data, tens of thousands of machines) (no SQL)
HBase

HBase is clone of Google’s Bigtable

<table>
<thead>
<tr>
<th></th>
<th>col1</th>
<th>col2</th>
<th>col3</th>
<th>col4</th>
<th>col5</th>
<th>col6</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>row2</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>row3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>x</td>
</tr>
</tbody>
</table>

- sparse tables, rows sorted by row keys
- multiple data versions – timestamps
- Random access performance on par with open source relational databases such as MySQL
- single global database-like table view (cloud scale)
5. transactions

- transaction = grouping of read and write operations
- $T_i$ has start time label $s_i$ and commit time label $c_i$ (all read and write operations happen after start time and before commit time)
transactions

- we need globally well-ordered time labels $s_i$ and $c_i$ (for our distributed system)
- $T_1$ and $T_2$ are concurrent if $[s_1,c_1]$ and $[s_2,c_2]$ overlap
- transaction $T_i$: either all write operations commit, or none (atomic)
snapshot isolation

define: (strong) snapshot isolation

1. $T_i$ reads from the last transaction committed before $s_i$ (that’s $T_i$’s snapshot)

2. concurrent transactions have disjoint write sets

(concurrent transactions are allowed for efficiency, but $T_1$ and $T_2$ cannot take the same $100 from a bank account)
snapshot isolation

1. $T_i$ reads from the last transaction committed before $s_i$

2. concurrent transactions have disjoint write sets

- $T_2$ does not see $T_1$’s writes
- $T_3$ sees $T_1$’s and $T_2$’s writes
- if $T_1$ and $T_2$ have overlapping write sets, at least one aborts
snapshot isolation

• desirables for implementation:
  ▪ first committer wins
  ▪ reads are not blocked

• implemented in mainstream DBMS (Oracle, ...), but scalable distributed transactions do not exist on the scale of clouds
6. our transactional system for HBase

- Grid 2010 conference, Brussels, October 2010
our transactional system for HBase

design principles:

- central mechanism for dispensing globally well-ordered time labels
- use HBase’s multiple data versions
- clients decide on whether they can commit (no centralized commit engine)
- two phases in commit
- store transactional meta-information in HBase tables
- use HBase as it is (bare-bones)
7. protocol

- transaction $T_i$ has
  - start label $s_i$ (ordered, not unique)
  - commit label $c_i$ (ordered with the $s_i$, unique)
  - write label $w_i$ (unique)
  - precommit label $p_i$ (ordered, unique)
## Protocol Summary

### User Table

<table>
<thead>
<tr>
<th>Row</th>
<th>colx</th>
<th>coly</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>La</td>
<td>m q</td>
</tr>
<tr>
<td>row2</td>
<td>Lb</td>
<td>n</td>
</tr>
</tbody>
</table>

### Counter Table

<table>
<thead>
<tr>
<th>Row</th>
<th>w-counter</th>
<th>p-counter</th>
<th>c-counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>row</td>
<td>1211</td>
<td>34</td>
<td>470</td>
</tr>
</tbody>
</table>

### Committed Table

<table>
<thead>
<tr>
<th>Commit Label</th>
<th>L_a</th>
<th>L_b</th>
<th>L_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>w_1</td>
<td>w_1</td>
<td></td>
</tr>
<tr>
<td>c_2</td>
<td>w_2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Precommit Queue Table (Queue up for Conflict Checking)

<table>
<thead>
<tr>
<th>Write Label</th>
<th>Precommit Label</th>
<th>L_a</th>
<th>L_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_1</td>
<td>p_1</td>
<td>w_1</td>
<td>w_1</td>
</tr>
<tr>
<td>w_2</td>
<td>p_2</td>
<td>w_2</td>
<td>w_2</td>
</tr>
</tbody>
</table>

### Commit Queue Table (Queue up for Committing)

<table>
<thead>
<tr>
<th>Write Label</th>
<th>Commit Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_1</td>
<td>c_1</td>
</tr>
<tr>
<td>w_2</td>
<td>c_2</td>
</tr>
</tbody>
</table>
protocol

- user data table I (user, many)

<table>
<thead>
<tr>
<th></th>
<th>colx</th>
<th>coly</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>m q</td>
<td></td>
</tr>
<tr>
<td>row2</td>
<td></td>
<td>n</td>
</tr>
</tbody>
</table>

- committed table (system, one)

<table>
<thead>
<tr>
<th>commit label</th>
<th>L_a</th>
<th>L_b</th>
<th>L_c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$w_1$</td>
<td>$w_1$</td>
<td></td>
</tr>
<tr>
<td>$c_2$</td>
<td>$w_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
transaction $T_i$

- at start, reads committed table, $s_i = c_{last} + 1$ (no wait)
- obtains $w_i$ from central $w$ counter
- reads $L_a$ by scanning $L_a$ column in committed table, reads from last $c_j$ with $c_j < s_i$
- writes preliminarily to user data table with HBase write timestamp $w_i$
- after reads/writes, queue up for conflict checking (get $p_i$ from central $p$ counter)
- after conflict checking, queue up for committing (get $c_i$ from central $c$ counter)
- commit by writing $c_i$ and writeset into committed table
central counters

- dispense unique, well-ordered $w_i$, $p_i$, $c_i$ labels
- use HBase’s built-in atomic IncrementColumnValue method on a fixed location in an additional system table (a separate counter for $w_i$, $p_i$ and $c_i$)
- take advantage of single global database-like table view
- c counter table (system, one)

<table>
<thead>
<tr>
<th>row</th>
<th>c-counter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>101</td>
</tr>
</tbody>
</table>
queue up for conflict checking

• precommit queue table (system, one)

<table>
<thead>
<tr>
<th>write label</th>
<th>precommit label</th>
<th>$L_a$</th>
<th>$L_b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>$p_1$</td>
<td>$w_1$</td>
<td>$w_1$</td>
</tr>
<tr>
<td>$w_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• how to make sure that the $T_i$ get processed in the order of the $p_i$ they get (‘first committer wins’): use a distributed queue mechanism

• $T_i$ puts $w_i$ in table, gets $p_i$ from $p$ counter, reads $\{w_j\}$ from table, then puts $p_i$ and writeset in table, waits until all in $\{w_j\}$ get a $p_j$ or disappear, wait for all $p_j < p_i$ (with write conflicts) to disappear, go on for conflict checking
conflict checking

- committed table

<table>
<thead>
<tr>
<th>commit label</th>
<th>$L_a$</th>
<th>$L_b$</th>
<th>$L_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$w_1$</td>
<td>$w_1$</td>
<td></td>
</tr>
<tr>
<td>$c_2$</td>
<td>$w_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $T_i$ checks conflicts in committed table: check for write conflicts with all transactions that have $s_i \leq c_j$, go on to commit if no conflicts, otherwise abort (remove $w_i$ from queue)
queue up for committing

• issue: make sure that committing transactions end up in committed table in the order they get their $c_i$ label (because $T_j$ gets its $s_j$ from committed table, and a gap in $c_i$ in the committed table that gets filled up later may lead to inconsistent snapshots)

<table>
<thead>
<tr>
<th>commit label</th>
<th>$L_a$</th>
<th>$L_b$</th>
<th>$L_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_2$</td>
<td></td>
<td>$w_2$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>commit label</th>
<th>$L_a$</th>
<th>$L_b$</th>
<th>$L_c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_1$</td>
<td>$w_1$</td>
<td></td>
<td>$w_1$</td>
</tr>
<tr>
<td>$c_2$</td>
<td></td>
<td>$w_2$</td>
<td></td>
</tr>
</tbody>
</table>
queue up for committing

- issue: make sure that committing transactions end up in committed table in the order they get their $c_i$ label: use a distributed queue mechanism!
- commit queue table (system, one)

<table>
<thead>
<tr>
<th>write label</th>
<th>commit label</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td>$w_2$</td>
<td></td>
</tr>
</tbody>
</table>

- $T_i$ puts $w_i$ in table, gets $c_i$ from counter, reads $\{w_j\}$ from table, then puts $c_i$ in table, waits until all in $\{w_j\}$ get a $c_j$ or disappear, goes on to commit: write $c_i$ and writeset in committed table, remove $w_i$ records from the two queues
### Protocol Summary

#### User Table

<table>
<thead>
<tr>
<th>Row</th>
<th>Write Label</th>
<th>Commit Label</th>
<th>Precommit Label</th>
<th>L_a</th>
<th>L_b</th>
</tr>
</thead>
<tbody>
<tr>
<td>row1</td>
<td>L_a</td>
<td>m q</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>row2</td>
<td>L_b</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Counter Table

<table>
<thead>
<tr>
<th>Write Label</th>
<th>Precommit Label</th>
<th>L_a</th>
<th>L_b</th>
<th>C-Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_1</td>
<td>p_1</td>
<td>w_1</td>
<td>w_1</td>
<td></td>
</tr>
<tr>
<td>w_2</td>
<td>p_2</td>
<td>w_2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Committed Table

<table>
<thead>
<tr>
<th>Write Label</th>
<th>Commit Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>w_1</td>
<td>c_1</td>
</tr>
<tr>
<td>w_2</td>
<td>c_2</td>
</tr>
</tbody>
</table>

(Strong) global snapshot isolation!
8. advanced features

version table (system, one)

<table>
<thead>
<tr>
<th>row</th>
<th>commit label</th>
</tr>
</thead>
<tbody>
<tr>
<td>L_a</td>
<td>c_1</td>
</tr>
<tr>
<td>L_b</td>
<td>c_2</td>
</tr>
</tbody>
</table>

- row L_a contains commit label of transaction that last updated location L_a
- lazily updated by reading transactions
- T_i that wants to read L_a, first checks version table: if c_1<s_i, scan [c_1+1,s_i-1] in committed table; if s_i<=c_1, scan [-inf,s_i-1]
advanced features

deal with straggling/failing processes
• add a timeout mechanism
• waiting processes can kill and remove straggling/failed processes from queues based on their own clock
• final commit does CheckAndPut on two rows (at once) in committed table

<table>
<thead>
<tr>
<th>commit label</th>
<th>L_a</th>
<th>L_b</th>
<th>w_1</th>
<th>w_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>c_1</td>
<td>w_1</td>
<td>w_1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c_2</td>
<td></td>
<td>w_2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>timeout</td>
<td></td>
<td></td>
<td>N</td>
<td>N</td>
</tr>
</tbody>
</table>

committed table
9. performance

Total Throughput vs Number of Clients

Total Throughput (# of incr operations per second)

Number of Clients
performance

Time to Traverse a Resultset vs Scan Row Range

Time to Traverse a Resultset (ms)

Scan Row Range

UNIVERSITY OF WATERLOO
performance
performance
10. comparison with Google’s percolator

• OSDI 2010, October 2010

Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

• goal: update Google’s search/rank index incrementally (‘fresher’ results, don’t wait 3 days)
• replace MapReduce by an incremental update system, but need concurrent changes to data
comparison with Google’s percolator

that MapReduce requires. To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository; we currently implement snapshot isolation semantics [5].

• snapshot isolation for Bigtable!
• percolator manages Google index/ranking since April 2010 (very very large dataset, tens of petabytes): it works and is very useful!
comparison with Google’s percolator

similarities with our HBase solution:

- central mechanism for dispensing globally well-ordered time labels
- use built-in multiple data versions
- clients decide on whether they can commit (no centralized commit engine)
- two phases in commit
- store transactional meta-information in tables
- clients remove straggling processes
comparison with Google’s percolator

differences with our HBase solution:

 percolator adds snapshot isolation metadata to user tables (more intrusive, but less centralized, no central system tables)

 percolator may block some reads

 percolator does not have strict first-committer-wins (may abort both concurrent T_i_s)

 different tradeoffs, different performance characteristics (percolator likely more scalable, throughput-friendly, less responsive in some cases)

(note: percolator cannot be implemented directly into HBase because HBase lacks row transactions)
11. conclusions

- we have described the first (global, strong) snapshot isolation mechanism for HBase
- independently and at the same time, Google has developed a snapshot isolation mechanism for Bigtable that uses design principles that are very similar to ours in many ways
- snapshot isolation is now available for distributed sparse data stores
- scalable distributed transactions do now exist on the scale of clouds