Accumulo – Extensions to Google’s Bigtable Design

Adam Fuchs

National Security Agency
Computer and Information Sciences Research Group

March 29, 2012
Contents

1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Progress

1. **Design Drivers**

2. **Apache Accumulo**
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. **Design Patterns**

4. **Fin**
Analysis of big data is central to our customers’ requirements, in which the strongest drivers are:

- **Scalability**: The ability to do twice the work at only (about) twice the cost.
- **Adaptability**: The ability to rapidly evolve the analytical tools available in an operational environment, building upon and enhancing existing capabilities.

From these directives we can derive the following requirements:

- Simplicity in the overall architecture to encourage collaboration and ameliorate learning curve.
- Generic design patterns to store and organize data whose format we don’t control.
- Generic discovery analytics to retrieve and visualize generic data.
- Solutions for common sub-problems, such as multi-level security and enforcement of legal restrictions, built into the infrastructure.
... is a secondary concern, given:

- hundreds of evolving applications,
- hundreds of changing data sources,
- non-trivial data volumes,
- many complicated interactions.

Instead, we need a generic platform that is cheap, simple, scalable, secure, and adaptable, with pretty good performance.
Apache Accumulo

- First code written in Spring of 2008
- Open-sourced as an Apache Software Foundation incubator podling in September, 2011
- Graduated to Top-Level Project in March, 2012
- Mostly a clone of Bigtable, but includes several notable features:
  - Iterators: a framework for processing sorted streams of key/value entries
  - Cell-level Security: mandatory, attribute-based access control with key/value granularity
  - Fault-Tolerant Execution Framework (FATE)
  - A compaction scheduler with nice properties
1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Basic Data Type

An Accumulo Key is a 5-tuple, including:

- **Row**: controls *Atomicity*
- **Column Family**: controls *Locality*
- **Column Qualifier**: controls *Uniqueness*
- **Visibility**: controls *Access* (unique to Accumulo)
- **Timestamp**: controls *Versioning*

### Sample Entries

<table>
<thead>
<tr>
<th>Row</th>
<th>Col. Fam.</th>
<th>Col. Qual.</th>
<th>Visibility</th>
<th>Timestamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Food</td>
<td>(Public)</td>
<td>20090801</td>
<td>Sushi</td>
</tr>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Programming Language</td>
<td>(Private)</td>
<td>20090830</td>
<td>Java</td>
</tr>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Programming Language</td>
<td>(Private)</td>
<td>20070725</td>
<td>C++</td>
</tr>
<tr>
<td>Adam</td>
<td>Friends</td>
<td>Bob</td>
<td>(Public)</td>
<td>20110601</td>
<td></td>
</tr>
<tr>
<td>Adam</td>
<td>Friends</td>
<td>Joe</td>
<td>(Private)</td>
<td>20110601</td>
<td></td>
</tr>
</tbody>
</table>
Collections of key/value pairs form Tables

Tables are partitioned into Tablets

Metadata tablets hold info about other tablets, forming a three-level hierarchy

A Tablet is a unit of work for a Tablet Server
Distributed Processes
1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Quick and loose definitions:

**Table**: A map of keys to values with one global sort order among keys.

**Tablet**: A row range within a Table.

**Tablet Server**: The mechanism that hosts Tablets, providing the primary functionality of Bigtable or Accumulo.

Tablet servers have several primary functions:

1. Hosting RPCs (read, write, etc.)
2. Managing resources (RAM, CPU, File I/O, etc.)
3. Scheduling background tasks (compactions, caching, etc.)
4. Handling key/value pairs

Category 4 is almost entirely accomplished through the *Iterator framework*. 
An *Iterator* is an object that provides an ordered stream of entries (key/value pairs), and supports basic *selection* and *filtering* methods.

Core Iterators provide a basic view of a tablet's entries, implementing:

- File Reads
- Block Caching
- Merging
- Deletion
- Isolation
- Locality Groups
- Range Selection
- Column Selection
- Cell-level Security

Application-level Iterators modify table semantics to provide custom views, persisted or otherwise:

- Versioning
- Filtering
- Aggregation
- Partitioned Joins
An Accumulo Key is a 5-tuple, including:

- **Row**: controls *Atomity*
- **Column Family**: controls *Locality*
- **Column Qualifier**: controls *Uniqueness*
- **Visibility**: controls *Access* (unique to Accumulo)
- **Timestamp**: controls *Versioning*

### Sample Entries

<table>
<thead>
<tr>
<th>Row</th>
<th>Col. Fam.</th>
<th>Col. Qual.</th>
<th>Visibility</th>
<th>Timestamp</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Food</td>
<td>(Public)</td>
<td>20090801</td>
<td>Sushi</td>
</tr>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Programming Language</td>
<td>(Private)</td>
<td>20060830</td>
<td>Java</td>
</tr>
<tr>
<td>Adam</td>
<td>Favorites</td>
<td>Programming Language</td>
<td>(Private)</td>
<td>20070725</td>
<td>C++</td>
</tr>
<tr>
<td>Adam</td>
<td>Friends</td>
<td>Bob</td>
<td>(Public)</td>
<td>20110601</td>
<td></td>
</tr>
<tr>
<td>Adam</td>
<td>Friends</td>
<td>Joe</td>
<td>(Private)</td>
<td>20110601</td>
<td></td>
</tr>
</tbody>
</table>
Visibility Label Syntax and Semantics

**Document Labels**

- $Doc_1$: (Federation)
- $Doc_2$: (Klingon|Vulcan)
- $Doc_3$: (Federation&Human&Vulcan)
- $Doc_4$: (Federation&(Human|Vulcan))

**User Authorization Sets**

- $CptKirk$: \{Federation, Human\}
- $MrSpock$: \{Federation, Human, Vulcan\}

**Syntax**

- $WORD$ ⇒ [a-zA-Z0-9_]+
- $CLAUSE$ ⇒ AND
- $CLAUSE$ ⇒ OR
- $AND$ ⇒ AND & AND
- $AND$ ⇒ (CLAUSE)
- $AND$ ⇒ WORD
- $OR$ ⇒ OR | OR
- $OR$ ⇒ (CLAUSE)
- $OR$ ⇒ WORD

**Semantics**

\[
\frac{(T \Rightarrow \tau) \land (\tau \in A)}{(T, A) \models true} \quad \text{term}
\]

\[
\frac{(T \Rightarrow T_1 \land T_2) \land ((T_1, A) \models true) \land ((T_2, A) \models true)}{(T, A) \models true} \quad \text{and}
\]

\[
\frac{(T \Rightarrow T_1 | T_2) \land (((T_1, A) \models true) \lor ((T_2, A) \models true))}{(T, A) \models true} \quad \text{or}
\]

\[
\frac{(T \Rightarrow (T_1)) \land (T_1 \models true)}{(T, A) \models true} \quad \text{paren}
\]
Cell-Level Security Iterator

Security Iterator
authorized to
{Federation, Human}

Lower Level
Iterator Tree

System Iterators

$\text{Doc}_1$ (Federation)

$\text{Doc}_4$ (Federation & (Human | Vulcan))

$\text{Doc}_2$ (Klingon | Vulcan)

$\text{Doc}_3$ (Federation & Human & Vulcan)

$\text{Doc}_4$ (Federation & (Human | Vulcan))
Aggregation

Goals: Count the number of times a word appears in a dynamic corpus, and count the number of documents that contain a given word.

Sample Corpus

\[\text{Doc}_1\] : "foo and bar are common variable names"
\[\text{Doc}_2\] : "one cannot live on bar food alone"
\[\text{Doc}_3\] : "Mr.T pities the fool at the bar"
\[\text{Doc}_4\] : "someone should invent the kung foo bar"

Input Key/Value Pairs:

<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>alone</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>are</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>at</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>bar</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>cannot</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>common</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>foo</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>foo</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>food</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>fool</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>invent</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>kung</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>live</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>Mr.T</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>names</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
<tr>
<td>on</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>one</td>
<td>Doc$_2$</td>
<td>1</td>
</tr>
<tr>
<td>should</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>someone</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>pities</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>Doc$_3$</td>
<td>1</td>
</tr>
<tr>
<td>the</td>
<td>Doc$_4$</td>
<td>1</td>
</tr>
<tr>
<td>variable</td>
<td>Doc$_1$</td>
<td>1</td>
</tr>
</tbody>
</table>
Aggregators replace the “versioning” functionality of a table.

Any associative, commutative operations on the values for a given key can be encoded in an aggregator.

Aggregators can persist an aggregation of the entries written to the table.

Aggregators are significantly more efficient than a read-modify-write loop due to “lazy” aggregation.
We can compose multiple Iterators by streaming the results of one Iterator through another Iterator.

Partial aggregation for the persisted view keeps the table small.

Additional iterators and aggregators implement different discovery analytics at query time.
Accumulo vs. HBase Atomic Increment

- HBase performs a server-side *upsert* (read-modify-write), taking advantage of previous value being resident in write-cache
- Accumulo buffers inserts and aggregates lazily but consistently, taking advantage of merge-tree data streams
- Both methods implement the same atomic increment semantics
- Performance varies wildly...
Aggregator wins for write performance with many different keys

Upsert wins for read performance with a small number of keys

Can we use both approaches?
Multi-Term Query with Document Partitioning

Goal: Find all of the documents that contain the words “foo” and “bar”.

Partitioned Corpus

\[
\begin{align*}
\text{Doc}_1 &: \text{ "foo and bar are common variable names"} \\
\text{Doc}_2 &: \text{ "one cannot live on bar food alone"} \\
\text{Doc}_3 &: \text{ "Mr.T pities the fool at the bar"} \\
\text{Doc}_4 &: \text{ "someone should invent the kung foo bar"}
\end{align*}
\]

\{ \text{Partition}_1 \} \quad \text{\{ Partition}_2 \}
Document Partitioning

Divide and Conquer:

<table>
<thead>
<tr>
<th>Row</th>
<th>ColFam</th>
<th>ColQual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part1</td>
<td>alone</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>and</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>are</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>at</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>bar</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>bar</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>bar</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>cannot</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>common</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>foo</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>food</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>fool</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>live</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>Mr.T</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>names</td>
<td>Doc1</td>
</tr>
<tr>
<td>Part1</td>
<td>on</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>one</td>
<td>Doc2</td>
</tr>
<tr>
<td>Part1</td>
<td>pities</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>the</td>
<td>Doc3</td>
</tr>
<tr>
<td>Part1</td>
<td>variable</td>
<td>Doc1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Row</th>
<th>ColFam</th>
<th>ColQual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part2</td>
<td>bar</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>foo</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>invent</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>kung</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>should</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>someone</td>
<td>Doc4</td>
</tr>
<tr>
<td>Part2</td>
<td>the</td>
<td>Doc4</td>
</tr>
</tbody>
</table>
Partitioned Join Iterator

promote docs common to foo and bar

Part₁

Doc₁

keys for bar

keys for foo

Part₁

bar

bar cannot

Part₁

Part₁

Part₁

Part₁

Part₁

System iterators

Part₁

common

Part₁

food

Part₁

fool

Part₁

Part₁

Doc₁

Doc₁

Doc₁

Doc₁

Doc₁

Doc₁

Doc₂

Doc₂

Doc₂

Doc₃
Wikipedia Search Engine Experiment

Goals:
- Create a generic text indexing platform
- Support a complex query language (i.e. mappable from Lucene)
- Scale to multiple nodes
- Support low-latency updates
- Support automatic balancing and fail-over

Data
- Three languages of Wikipedia: EN, ES, DE
- 5.9 million articles
- 2.37 billion (word, document) tuples
- 11.8 GB (compressed)

Cluster
- 10 Nodes
- 30 TB disk (60x500GB drives)
- 120 cores
- 320 GB RAM
Wikipedia Search Results

- Tested on conjunctions of high-degree terms
- Retrieved entire contents of articles matching queries
- Paging possible for ultra-low latency response time

**Query Performance**

<table>
<thead>
<tr>
<th>Query</th>
<th>Samples (seconds)</th>
<th>Matches</th>
<th>Result Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>“old” and “man” and “sea”</td>
<td>4.07, 3.79, 3.65</td>
<td>22,956</td>
<td>3,830,102</td>
</tr>
<tr>
<td>“paris” and “in” and “the” and “spring”</td>
<td>3.06, 3.06, 2.78</td>
<td>10,755</td>
<td>1,757,293</td>
</tr>
<tr>
<td>“rubber” and “ducks” and “ernie”</td>
<td>0.08, 0.08, 0.10</td>
<td>6</td>
<td>808</td>
</tr>
<tr>
<td>“fast” and (“furious” or “furriest”)</td>
<td>1.34, 1.33, 1.30</td>
<td>2,973</td>
<td>493,800</td>
</tr>
<tr>
<td>“slashdot” and “grok”</td>
<td>0.06, 0.06, 0.06</td>
<td>14</td>
<td>2,371</td>
</tr>
<tr>
<td>“three” and “little” and “pigs”</td>
<td>0.92, 0.91, 0.90</td>
<td>2,742</td>
<td>481,531</td>
</tr>
</tbody>
</table>

**Documents per Term**

<table>
<thead>
<tr>
<th>Term</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>ducky</td>
<td>795</td>
</tr>
<tr>
<td>ernie</td>
<td>13,433</td>
</tr>
<tr>
<td>fast</td>
<td>166,813</td>
</tr>
<tr>
<td>furious</td>
<td>10,535</td>
</tr>
<tr>
<td>furriest</td>
<td>45</td>
</tr>
<tr>
<td>grok</td>
<td>1,168</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>in</td>
<td>1,884,638</td>
</tr>
<tr>
<td>little</td>
<td>320,748</td>
</tr>
<tr>
<td>man</td>
<td>548,238</td>
</tr>
<tr>
<td>old</td>
<td>720,795</td>
</tr>
<tr>
<td>paris</td>
<td>232,464</td>
</tr>
<tr>
<td>pigs</td>
<td>8,356</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Term</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>rubber</td>
<td>17,235</td>
</tr>
<tr>
<td>sea</td>
<td>247,231</td>
</tr>
<tr>
<td>slashdot</td>
<td>2,343</td>
</tr>
<tr>
<td>spring</td>
<td>125,605</td>
</tr>
<tr>
<td>the</td>
<td>3,509,498</td>
</tr>
<tr>
<td>three</td>
<td>718,810</td>
</tr>
</tbody>
</table>
Iterator Summary

Iterators provide a modular implementation of Tablet Server functionality, resulting in:

- Reduced complexity of Tablet Server code
- Increased unit testability
- Simple extensibility for specialized applications
1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
The Perils of Distributed Computing

Dealing with failures is hard!

- Operations like table creation are logically atomic, but consist of multiple operations on distributed systems.
- Resource locking (via mutex, semaphores, etc.) provides some sanity.
- Distributed systems have many complicated failure modes: clients, master, tablet servers, and dependent systems can all go offline periodically.

Who is responsible for unlocking locks when any component can fail?

How do we know it’s safe to unlock a lock?
Accumulo Testing Procedures

Testing Frameworks

- **Unit**: Verify correct functioning of each module separately
- **System**: Perform correctness and performance tests on a small running instance
- **Load/Scale**: Generate high loads at scale and measure performance and correctness
- **Random Walk**: Randomly, repeatedly, and concurrently execute a variety of test modules representative of user activity on an instance at scale
- **Simulation**: Evaluate the model to gauge expected performance

Other Considerations

- Scoping tests to include server-side code, client-side code, dependent processes, etc.
- Code coverage vs. path coverage
- Static vs. dynamic analysis
- Simulating failures of distributed components
- Strange failure modes (often hardware/physics-related)
Fault-Tolerant Executor

- If a process dies, previously submitted operations continue to execute on restart.
- FATE serializes every task in Zookeeper before execution.
- The Master process uses FATE to execute table operations and administrative actions.
- FATE eliminates the single point of failure.
Adampotence

- Idempotent: $f(f(x)) = f(x)$
- Adampotent: $f(f'(x)) = f(x)$, where $f'(x)$ denotes partial execution of $f(x)$
public interface Repo<T> extends Serializable {

    long isReady(long tid, T environment) throws Exception;

    Repo<T> call(long tid, T environment) throws Exception;

    void undo(long tid, T environment) throws Exception;

}

call() returns next op, null if done

call(), undo(), and isReady() must be idempotent

undo() should clean up any possible partial execution of isReady() or call()
FATE API

Client API

```
long startTransaction();
void seedTransaction(long tid, Repo op);
TStatus waitForCompletion(long tid);
Exception getException(long tid);
void delete(long tid);
```
FATE Execution State Model

Operation States
- New
  - In Progress
    - Succeeded
    - Failed

Executor States
- Reserve TID
  - Push Op
    - Execute Top Op
      - Save Exception
        - Mark Failure In Progress
      - Mark Success
        - Pop Op
          - Undo Top Op
            - Mark Failed
  - Mark Failed
CreateTable FATE Op

Steps for CreateTable Operation:

1. Reserve a Table ID
2. Set Table Permissions
3. Populate Configuration in Zookeeper
   - Reentrantly lock table
   - Relate table name to table ID
4. Create HDFS Directory
5. Populate Metadata Table Entries
6. Finish Create Table
   - Notify Master of new tablet(s)
   - Unlock table
FATE Admin Tool

<table>
<thead>
<tr>
<th>txid</th>
<th>status</th>
<th>op</th>
<th>locked</th>
<th>locking</th>
<th>top</th>
</tr>
</thead>
<tbody>
<tr>
<td>59c0403614dc0c39</td>
<td>IN_PROGRESS</td>
<td>RenameTable</td>
<td>[]</td>
<td>[W:cz]</td>
<td>RenameTable</td>
</tr>
<tr>
<td>37539f8d61548764</td>
<td>IN_PROGRESS</td>
<td>ChangeTableState</td>
<td>[]</td>
<td>[W:cz]</td>
<td>ChangeTableState</td>
</tr>
<tr>
<td>02f8323a3136e60d</td>
<td>IN_PROGRESS</td>
<td>TableRangeOp</td>
<td>[]</td>
<td>[W:cz]</td>
<td>TableRangeOp</td>
</tr>
<tr>
<td>044015732e97eeec1</td>
<td>IN_PROGRESS</td>
<td>CompactRange</td>
<td>[]</td>
<td>[R:cz]</td>
<td>CompactRange</td>
</tr>
<tr>
<td>6ce9dd63f9d51448</td>
<td>IN_PROGRESS</td>
<td>CompactRange</td>
<td>[]</td>
<td>[R:cz]</td>
<td>CompactRange</td>
</tr>
<tr>
<td>417cb9b60e44ecd9</td>
<td>IN_PROGRESS</td>
<td>TableRangeOp</td>
<td>[]</td>
<td>[W:cz]</td>
<td>TableRangeOp</td>
</tr>
<tr>
<td>5e7c5284a4677d6c</td>
<td>IN_PROGRESS</td>
<td>DeleteTable</td>
<td>[]</td>
<td>[W:cz]</td>
<td>DeleteTable</td>
</tr>
<tr>
<td>6633d3d841d66995</td>
<td>IN_PROGRESS</td>
<td>TableRangeOp</td>
<td>[W:cz]</td>
<td>[]</td>
<td>TableRangeOpWait</td>
</tr>
</tbody>
</table>

- Monitoring tool for FATE operations
- Supports debugging, such as with deadlocks
- Helps recovery from failed clients
FATE Summary

- FATE provides generic fault tolerance for administrative actions
- With FATE, we removed custom synchronization code for a dozen procedures
- Table-level locking is now low risk
- Improves testability
- Reduces complexity
- Increases modularity

FATE Operations
- BulkImport
- ChangeTableState
- CloneTable
- CompactRange
- CreateTable
- DeleteTable
- RenameTable
- TableRangeOp
- DisconnectLogger
- FlushTablets
- ShutdownTServer
- StopLogger
1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Major Compaction Efficiency

Major Compaction: Noun. The tablet operation that merges multiple files into one file.

- Overly aggressive major compaction results in $N^2$ write complexity
- Overly lazy major compaction results in disk thrashing during queries (or unavailable tablets)
- Tuning major compaction operations is a trade-off between ingest and query performance
Accumulo Major Compaction Algorithm

1. let \( r \geq 1.0 \) be some ratio
2. \( F \leftarrow \) all files referenced by a tablet
3. if \( F \) is empty then exit
4. \( f_0 \leftarrow \) biggest file in \( F \)
5. \( a \leftarrow \) aggregate size of files in \( F \)
6. if \( a > r|f_0| \) then compact all files in \( F \) and exit
7. otherwise, remove \( f_0 \) from \( F \) and go to step 3
Major Compaction Performance

![Graph showing Major Compaction Performance](image-url)
Progress

1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Our use of Accumulo fundamentally differs from how we use RDBMS technology. In particular, Accumulo supports:

- Wide, sparse rows
- Indexes that span multiple columns

To adapt Accumulo for use in our applications, we have formalized several design patterns for Accumulo (or any Bigtable clone) including:

- Information Retrieval Patterns and Discovery Analytics
- Graph Analysis Patterns
- Machine Learning Patterns
- ...
Event Table with Inverted Index

Table:  Event Table  Inverted Index

Row:  <UUID>  <Term>

Column Family:  <Type>  <Type> + <Field>

Column Qualifier:  <Field>  <UUID>

Value:  <Term>  <Digest of Event>
Document Partitioned Index

Table:

Indexed Event Table

Row:

Column Family:

“Event”

“Index”

“Geo”

<Partition ID>

Column Qualifier (2-Tuples):

<UUID>

<Field>

<UUID>

<Term>

<Morton Coordinate>

Value:

<Term>

Event and Index Records Co-Partitioned!
Multidimensional Index

Table: Geo Index

Row: <Morton Ordered Coordinate>

Column Family: <Event Type>

Column Qualifier: <UUID>

Value: <Digest of Event>

See also: http://en.wikipedia.org/wiki/Geohash
Graph Table

Table:

Row:

Column Family: "Node Info" "Out Edges" "In Edges"

Column Qualifier (Tuples):

Value:

<Field> <Node ID> <Node ID>

<Edge ID> <Edge Info> <Edge Info>
1. Design Drivers

2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction

3. Design Patterns

4. Fin
Check out Apache Accumulo (http://accumulo.apache.org/) for interesting implementations of:

- Merging Tablets
- Table Cloning: Hard link-style table copying
- Relative Key Encoded RFile file format
- Adaptive locality groups
- Isolation over scans of wide rows
- Bulk loading
- Logical time
- Client-side threading models for batch writes and scans
- Merging minor compactions
- Distributed write-ahead log