Accumulo – Extensions to Google's Bigtable Design

Adam Fuchs

National Security Agency
Computer and Information Sciences Research Group

March 29, 2012

Contents

Progress

Design Drivers

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.

Optimization

... 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.

Progress

1. Design Drivers
2. Apache Accumulo
   - Intro to Bigtable
   - Iterators
   - FATE
   - Major Compaction
3. Design Patterns
4. Fin

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

Progress

Basic Data Type

An Accumulo Key is a 5-tuple, including:

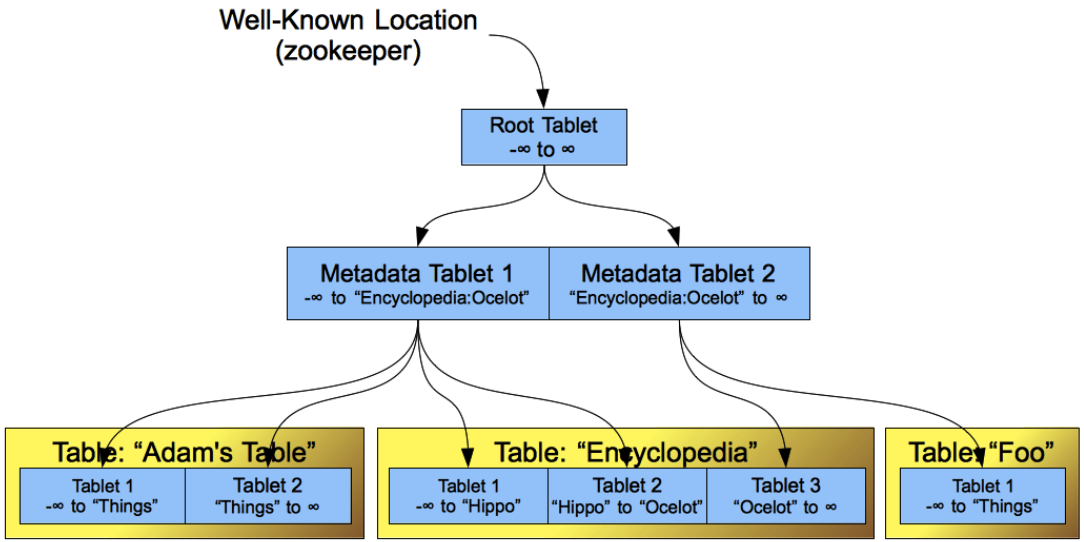
- **Row**: controls *Atomicity*
- **Column Family**: controls *Locality*
- **Column Qualifier**: controls *Uniqueness*
- **Visibility**: controls *Access* (unique to Accumulo)
- **Timestamp**: controls *Versioning*

Sample Entries

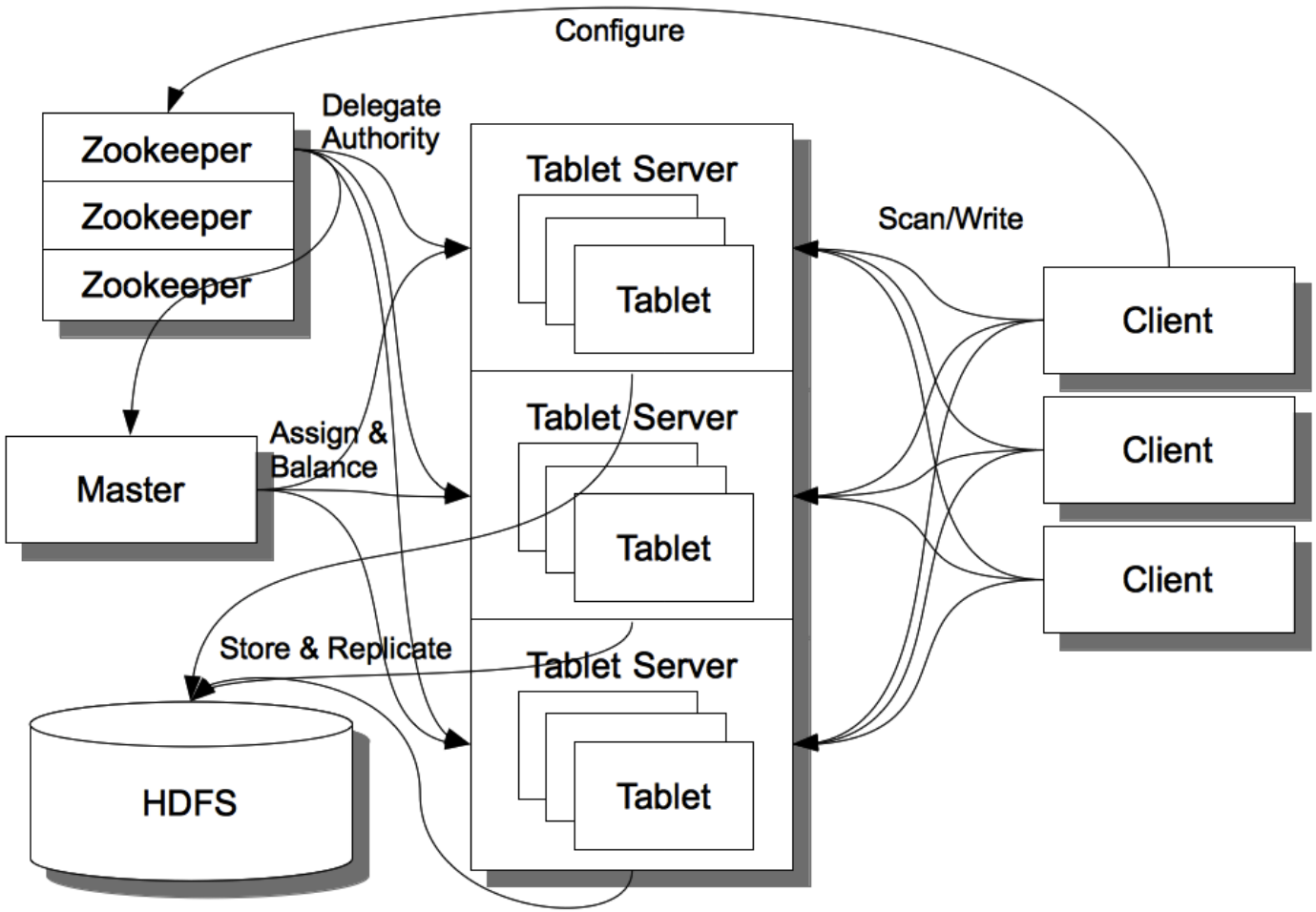
Row	: Col. Fam.	: Col. Qual.	: Visibility	: Timestamp	⇒ Value
Adam	: Favorites	: Food	: (Public)	: 20090801	⇒ Sushi
Adam	: Favorites	: Programming Language	: (Private)	: 20090830	⇒ Java
Adam	: Favorites	: Programming Language	: (Private)	: 20070725	⇒ C++
Adam	: Friends	: Bob	: (Public)	: 20110601	⇒
Adam	: Friends	: Joe	: (Private)	: 20110601	⇒

Tablets

- 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



Progress

Tablet Server Composition

Quick and loose definitions:

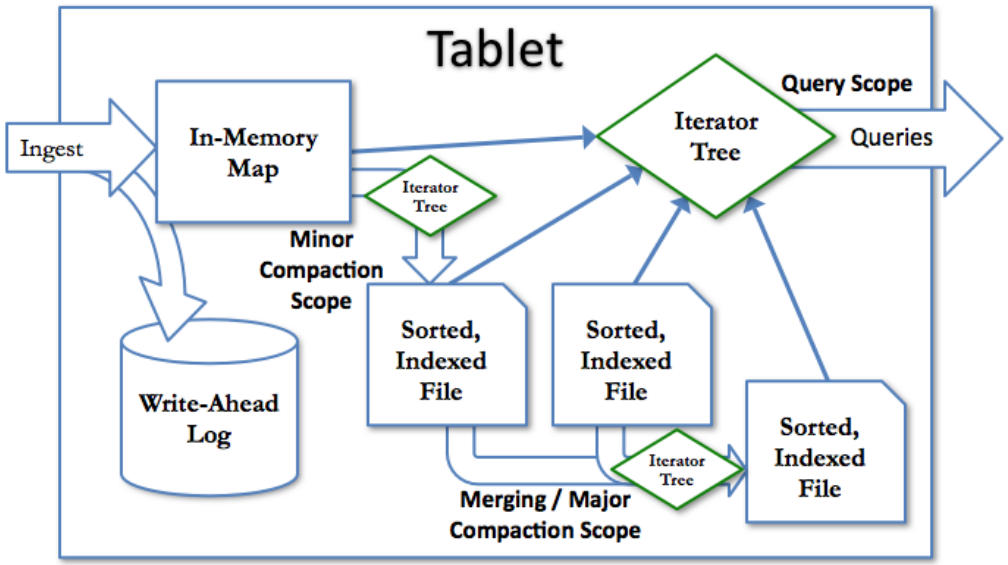
Table: A map of keys to values with one global sort order among keys.
Tablet: A row range within a Table.
Tablet Server: The mechanism that hosts Tablets, providing the primary functionality of Bigtable or Accumulo.

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*.

Tablet Server Data Flow

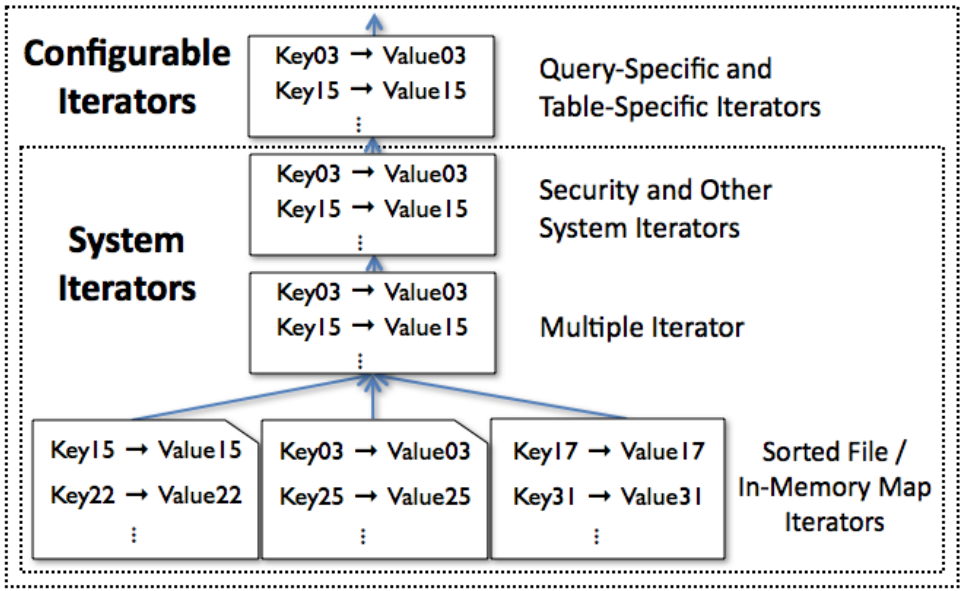


Iterator Uses

- File Reads
- Block Caching
- Merging
- Deletion
- Isolation
- Locality Groups
- Range Selection
- Column Selection
- Cell-level Security
- Versioning
- Filtering
- Aggregation
- Partitioned Joins

Iterators

- 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



Modified Key/Value Pair Definition

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

Row	: Col. Fam.	: Col. Qual.	: Visibility	: Timestamp	⇒ Value
Adam	: Favorites	: Food	: (Public)	: 20090801	⇒ Sushi
Adam	: Favorites	: Programming Language	: (Private)	: 20090830	⇒ Java
Adam	: Favorites	: Programming Language	: (Private)	: 20070725	⇒ C++
Adam	: Friends	: Bob	: (Public)	: 20110601	⇒
Adam	: Friends	: Joe	: (Private)	: 20110601	⇒

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	$\Rightarrow$	`[a-zA-Z0-9_]+`
CLAUSE	$\Rightarrow$	*AND*
	$\Rightarrow$	*OR*
AND	$\Rightarrow$	*AND* & *AND*
	$\Rightarrow$	(*CLAUSE*)
	$\Rightarrow$	*WORD*
OR	$\Rightarrow$	*OR* | *OR*
	$\Rightarrow$	(*CLAUSE*)
	$\Rightarrow$	*WORD*

Semantics

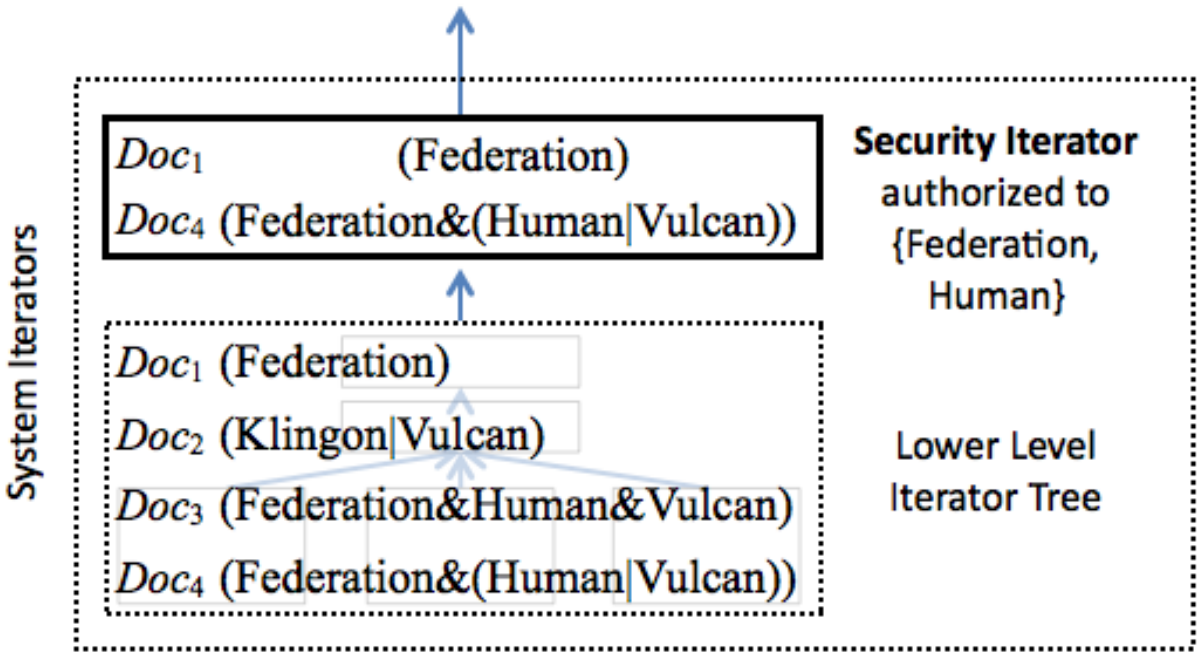
$$\frac{(\mathrm{T} \Rightarrow \tau) \wedge (\tau \in \mathrm{A})}{(\mathrm{T}, \mathrm{A}) \models \text{true}} \text{ term}$$

$$\frac{(\mathrm{T} \Rightarrow \mathrm{T}_1 \,\&\, \mathrm{T}_2) \wedge ((\mathrm{T}_1, \mathrm{A}) \models \text{true}) \wedge ((\mathrm{T}_2, \mathrm{A}) \models \text{true})}{(\mathrm{T}, \mathrm{A}) \models \text{true}} \text{ and}$$

$$\frac{(\mathrm{T} \Rightarrow \mathrm{T}_1 \mid \mathrm{T}_2) \wedge (((\mathrm{T}_1, \mathrm{A}) \models \text{true}) \vee ((\mathrm{T}_2, \mathrm{A}) \models \text{true}))}{(\mathrm{T}, \mathrm{A}) \models \text{true}} \text{ or}$$

$$\frac{(\mathrm{T} \Rightarrow (\mathrm{T1})) \wedge (\mathrm{T1} \models \text{true})}{(\mathrm{T}, \mathrm{A}) \models \text{true}} \text{ paren}$$

Cell-Level Security Iterator



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

Doc_1 : "foo and bar are common variable names"

Doc_2 : "one cannot live on bar food alone"

Doc_3 : "Mr.T pities the fool at the bar"

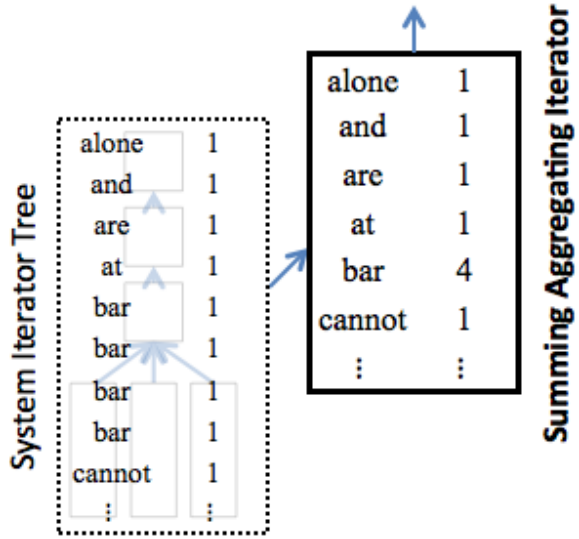
Doc_4 : "someone should invent the kung foo bar"

Input Key/Value Pairs:

Row	Column	Value
alone	Doc_2	1
and	Doc_1	1
are	Doc_1	1
at	Doc_3	1
bar	Doc_1	1
bar	Doc_2	1
bar	Doc_3	1
bar	Doc_4	1
cannot	Doc_2	1
common	Doc_1	1
foo	Doc_1	1
foo	Doc_4	1
food	Doc_2	1
fool	Doc_3	1
invent	Doc_4	1
kung	Doc_4	1
live	Doc_2	1
Mr.T	Doc_3	1
names	Doc_1	1
on	Doc_2	1
one	Doc_2	1
should	Doc_4	1
someone	Doc_4	1
pities	Doc_3	1
the	Doc_3	1
the	Doc_3	1
the	Doc_4	1
variable	Doc_1	1

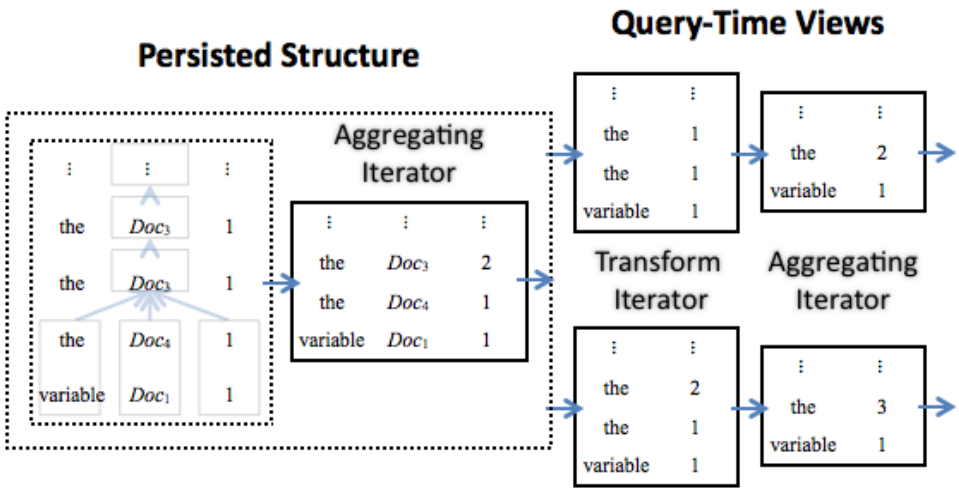
A Simple Aggregator

- 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



Composing Multiple Iterators

- 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

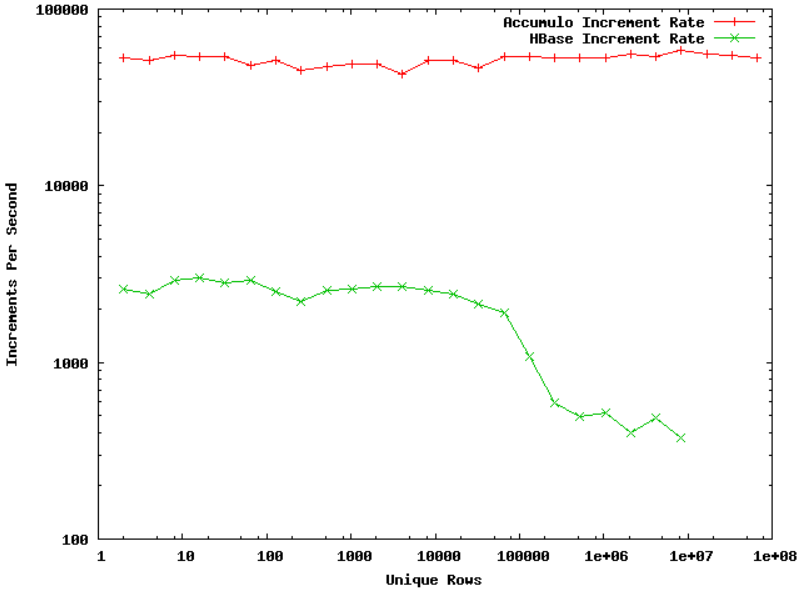


Accumulo vs. HBase Atomic Increment

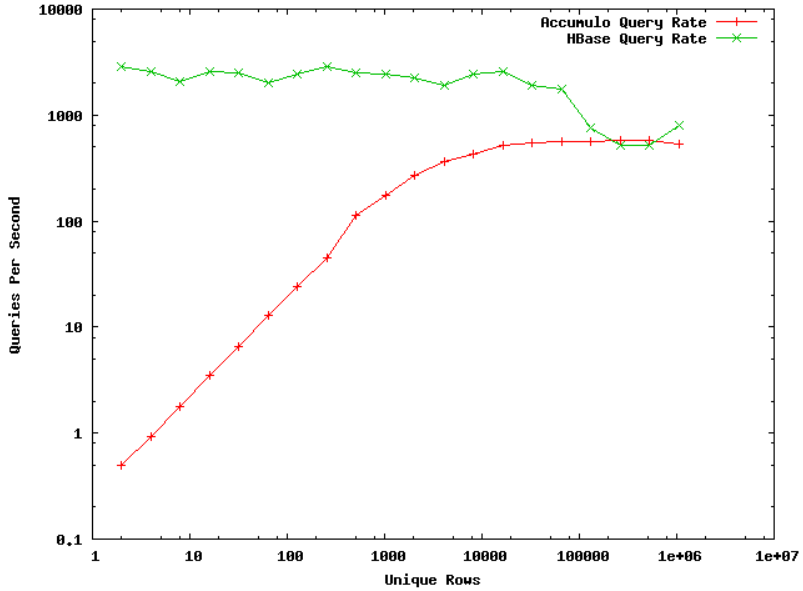
- HBase performs a server-side *upsert* (read-modify-write), taking advantage of previous value being resident in write-cache
- Accumulo buffers inserts and aggregates lazily but consistently, taking advantage of merge-tree data streams
- Both methods implement the same atomic increment semantics
- Performance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

Increment Performance Comparison

Write Performance



Read Performance



- 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

Doc_1 :	`"foo and bar are common variable names"`	$Partition_1$
Doc_2 :	`"one cannot live on bar food alone"`	
Doc_3 :	`"Mr.T pities the fool at the bar"`	
Doc_4 :	`"someone should invent the kung foo bar"`	$Partition_2$

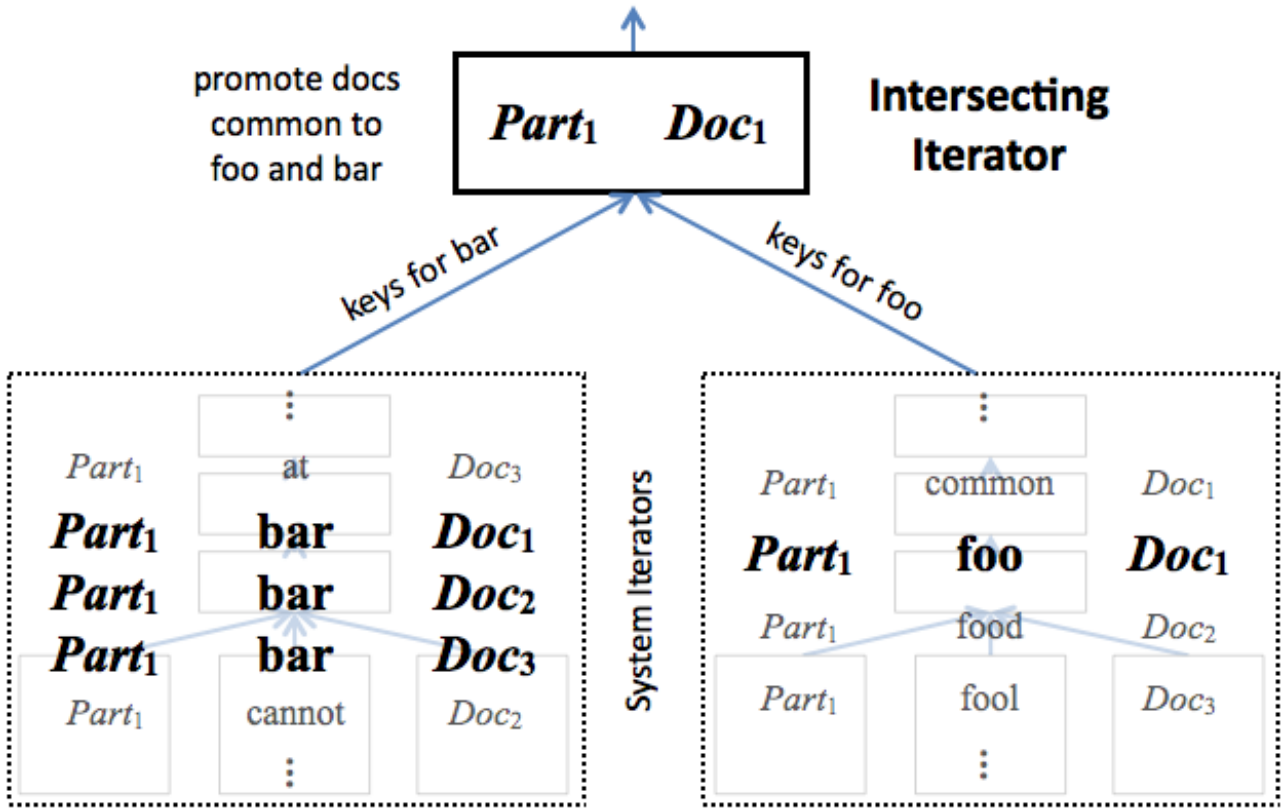
Document Partitioning

Divide and Conquer:

Row	ColFam	ColQual
$Part_1$	alone	Doc_2
$Part_1$	and	Doc_1
$Part_1$	are	Doc_1
$Part_1$	at	Doc_3
$Part_1$	bar	Doc_1
$Part_1$	bar	Doc_2
$Part_1$	bar	Doc_3
$Part_1$	cannot	Doc_2
$Part_1$	common	Doc_1
$Part_1$	foo	Doc_1
$Part_1$	food	Doc_2
$Part_1$	fool	Doc_3
$Part_1$	live	Doc_2
$Part_1$	Mr.T	Doc_3
$Part_1$	names	Doc_1
$Part_1$	on	Doc_2
$Part_1$	one	Doc_2
$Part_1$	pities	Doc_3
$Part_1$	the	Doc_3
$Part_1$	variable	Doc_1

Row	ColFam	ColQual
$Part_2$	bar	Doc_4
$Part_2$	foo	Doc_4
$Part_2$	invent	Doc_4
$Part_2$	kung	Doc_4
$Part_2$	should	Doc_4
$Part_2$	someone	Doc_4
$Part_2$	the	Doc_4

Partitioned Join Iterator



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

Query	Samples (seconds)					Matches	Result Size
"old" and "man" and "sea"	4.07	3.79	3.65	3.85	3.67	22,956	3,830,102
"paris" and "in" and "the" and "spring"	3.06	3.06	2.78	3.02	2.92	10,755	1,757,293
"rubber" and "ducky" and "ernie"	0.08	0.08	0.10	0.11	0.10	6	808
"fast" and ("furious" or "furriest")	1.34	1.33	1.30	1.31	1.31	2,973	493,800
"slashdot" and "grok"	0.06	0.06	0.06	0.06	0.06	14	2,371
"three" and "little" and "pigs"	0.92	0.91	0.90	1.08	0.88	2,742	481,531

Documents per Term

Term	Cardinality
ducky	795
ernie	13,433
fast	166,813
furious	10,535
furriest	45
grok	1,168

Term	Cardinality
in	1,884,638
little	320,748
man	548,238
old	720,795
paris	232,464
pigs	8,356

Term	Cardinality
rubber	17,235
sea	247,231
slashdot	2,343
spring	125,605
the	3,509,498
three	718,810

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

Progress

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 $f(x)$

REPO: Repeatable Persisted Operation

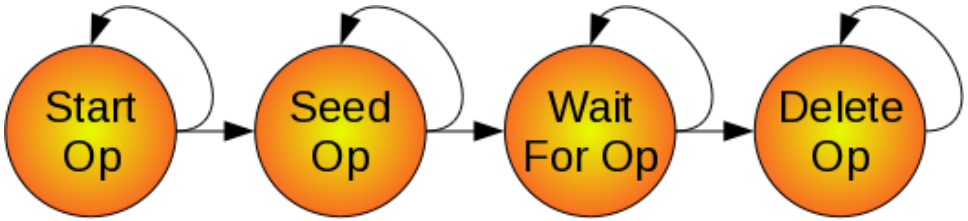

```
public interface Repo<T> extends Serializable {

  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 adampotent
- `undo()` should clean up any possible partial execution of `isReady()` or `call()`

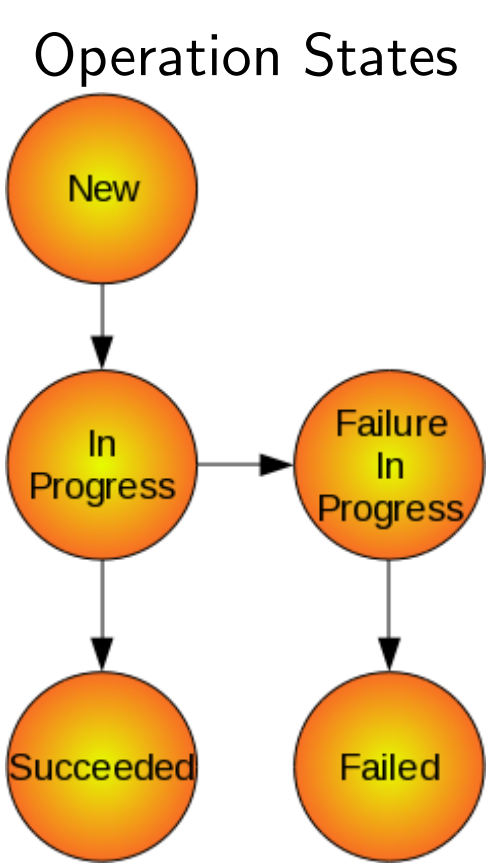
FATE API

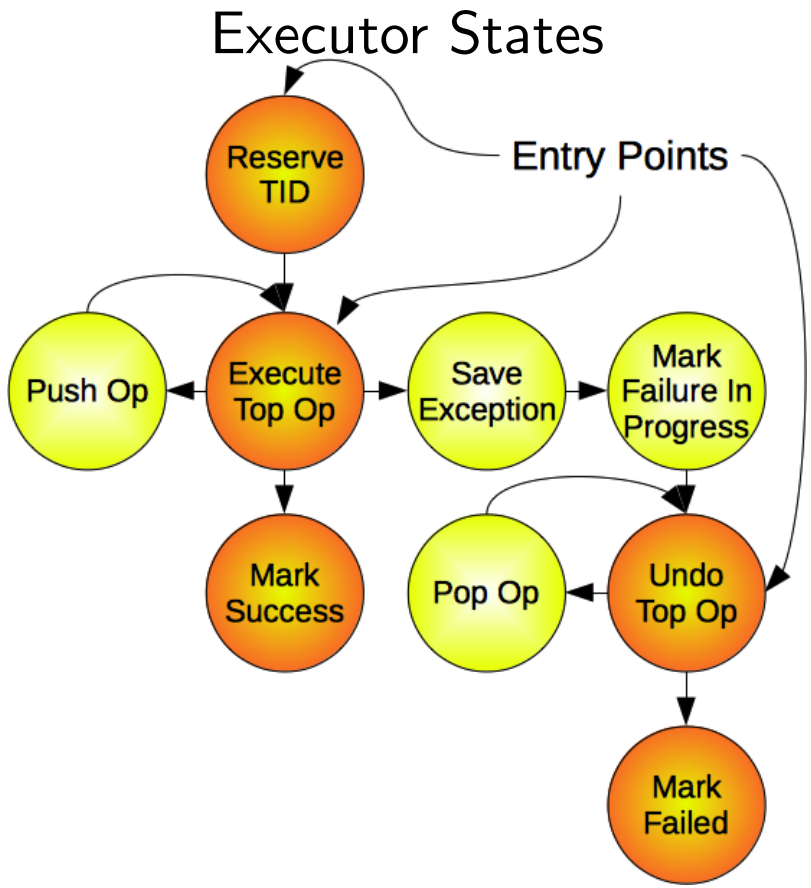
Client API

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



FATE Execution State Model





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


```
$ ./bin/accumulo org.apache.accumulo.server.fate.Admin print

txid: 59c0403614dc0c39 status: IN_PROGRESS op: RenameTable      locked: []     locking: [W:cz] top: RenameTable
txid: 37539f8d61548764 status: IN_PROGRESS op: ChangeTableState locked: []     locking: [W:cz] top: ChangeTableState
txid: 02f8323a3136e60d status: IN_PROGRESS op: TableRangeOp     locked: []     locking: [W:cz] top: TableRangeOp
txid: 044015732e97eec1 status: IN_PROGRESS op: CompactRange     locked: []     locking: [R:cz] top: CompactRange
txid: 6ce9dd63f9d51448 status: IN_PROGRESS op: CompactRange     locked: []     locking: [R:cz] top: CompactRange
txid: 417cb9b60e44ecd9 status: IN_PROGRESS op: TableRangeOp     locked: []     locking: [W:cz] top: TableRangeOp
txid: 5e7c5284a4677d6c status: IN_PROGRESS op: DeleteTable      locked: []     locking: [W:cz] top: DeleteTable
txid: 6633d3d841d66995 status: IN_PROGRESS op: TableRangeOp     locked: [W:cz] locking: []     top: TableRangeOpWait
```

- 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

Progress

Major Compaction Efficiency

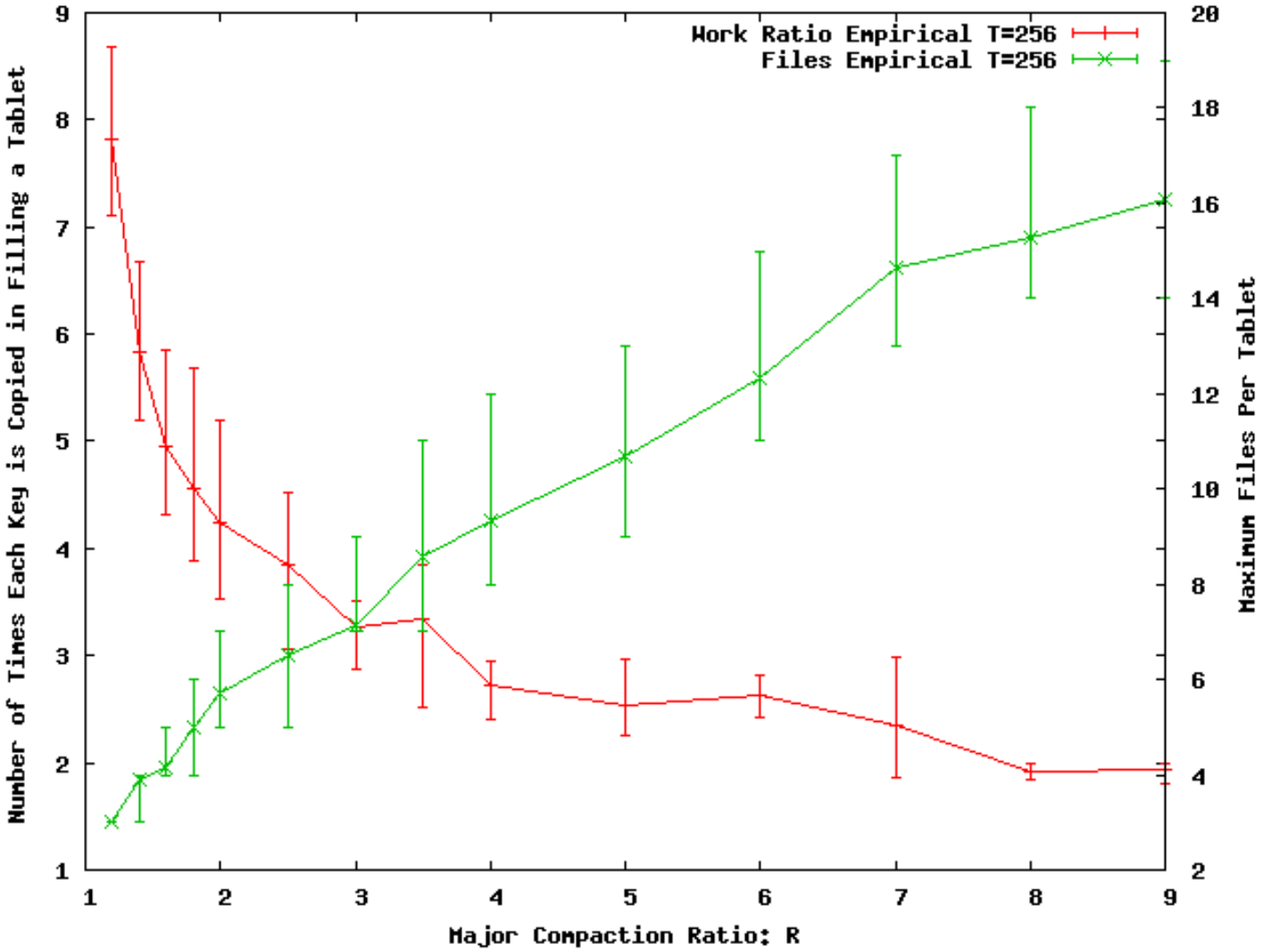
Major Compaction: Noun. The tablet operation that merges multiple files into one file.

- 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



Progress

Design Patterns

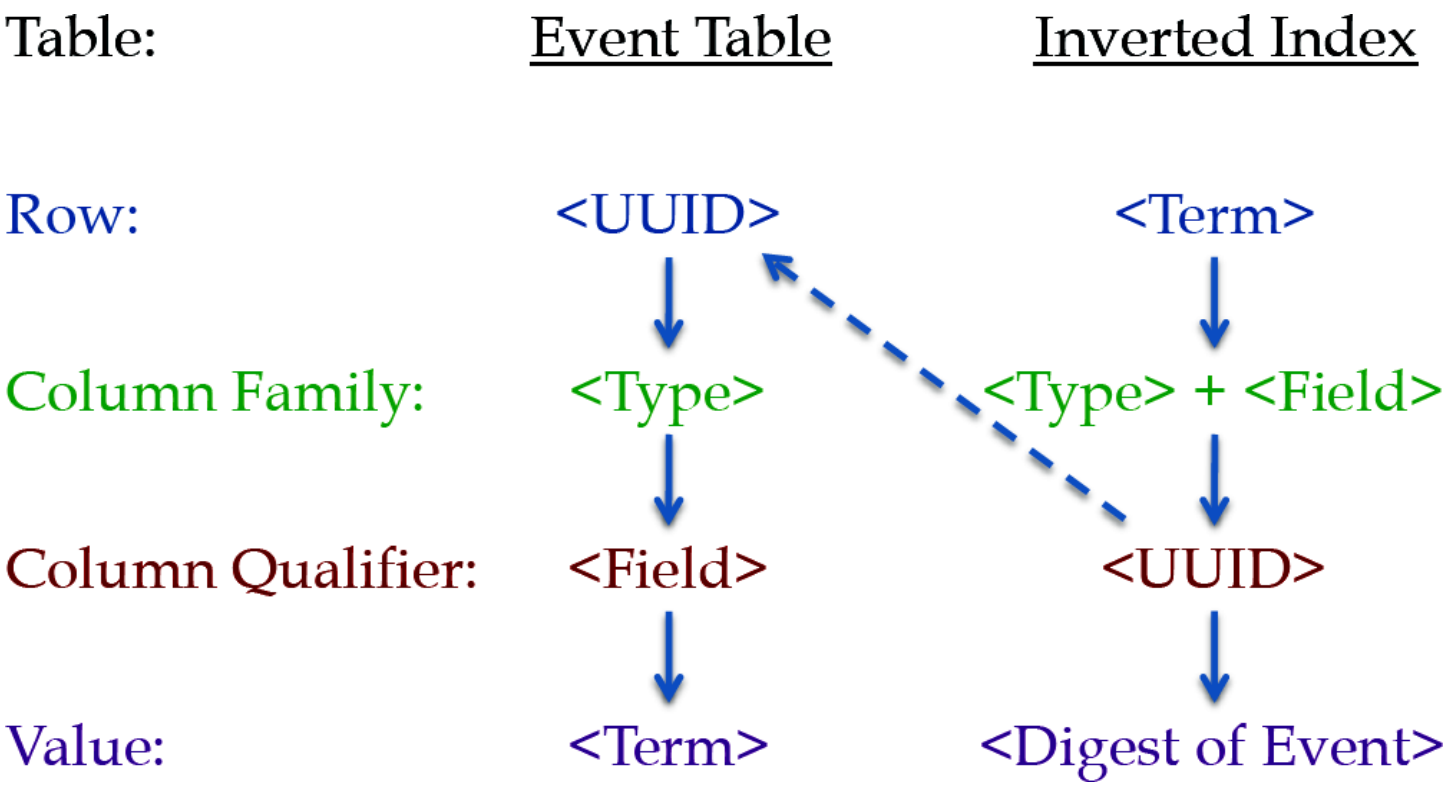
Our use of Accumulo fundamentally differs from how we use RDBMS technology. In particular, Accumulo supports:

- Wide, sparse rows
- Indexes that span multiple columns

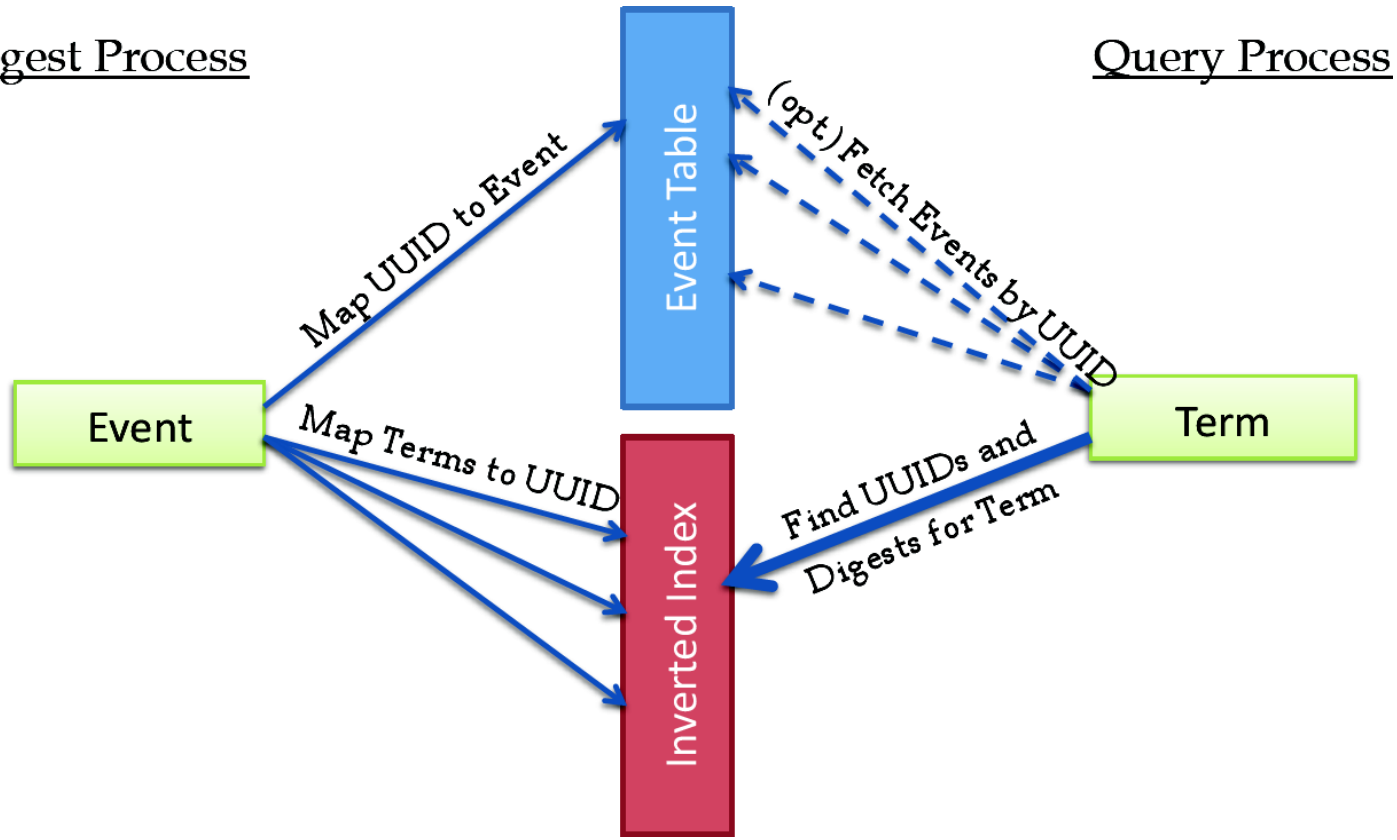
To adapt Accumulo for use in our applications, we have formalized several design patterns for Accumulo (or any Bigtable clone) including:

- Information Retrieval Patterns and Discovery Analytics
- Graph Analysis Patterns
- Machine Learning Patterns
- ...

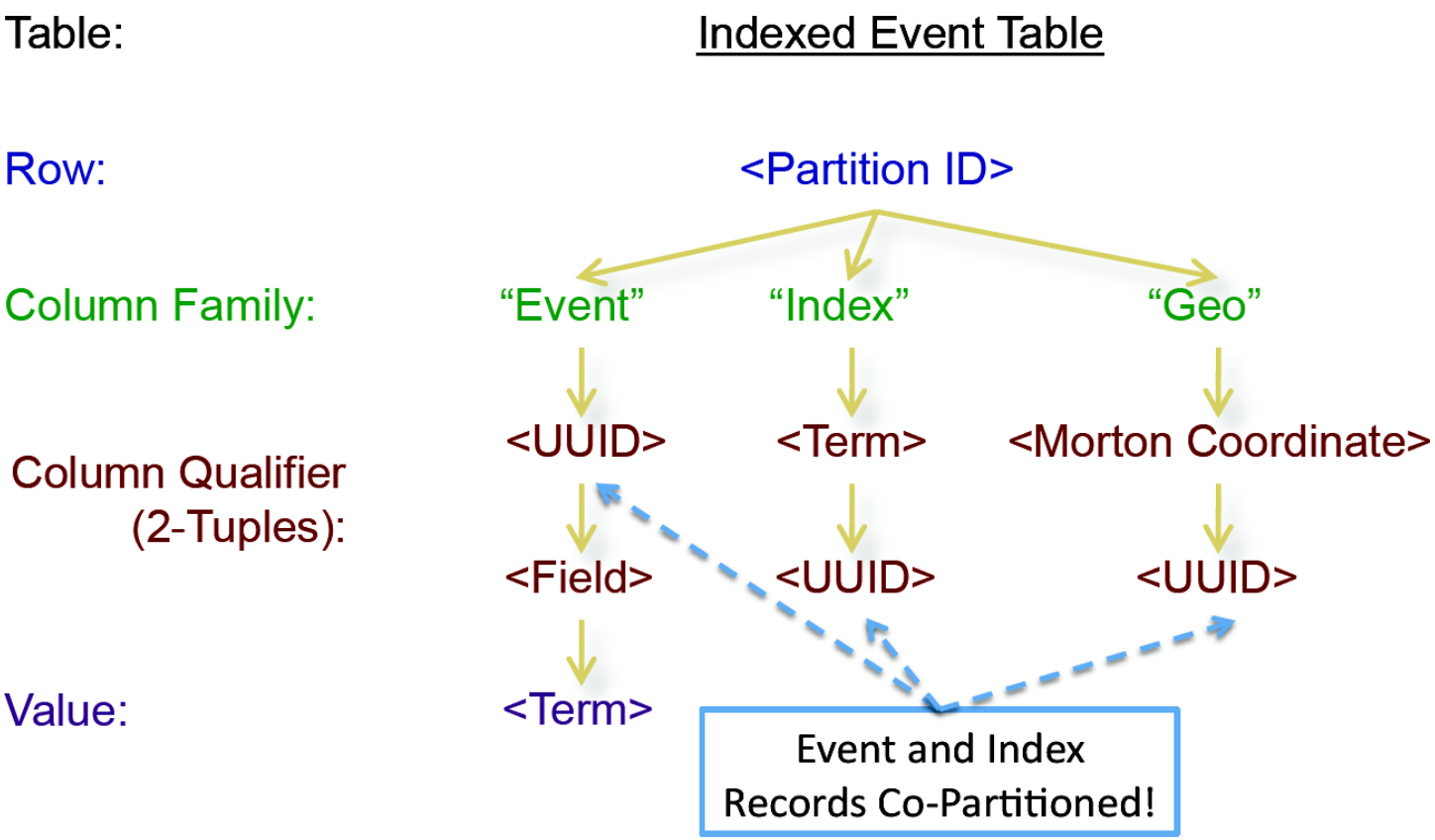
Event Table with Inverted Index



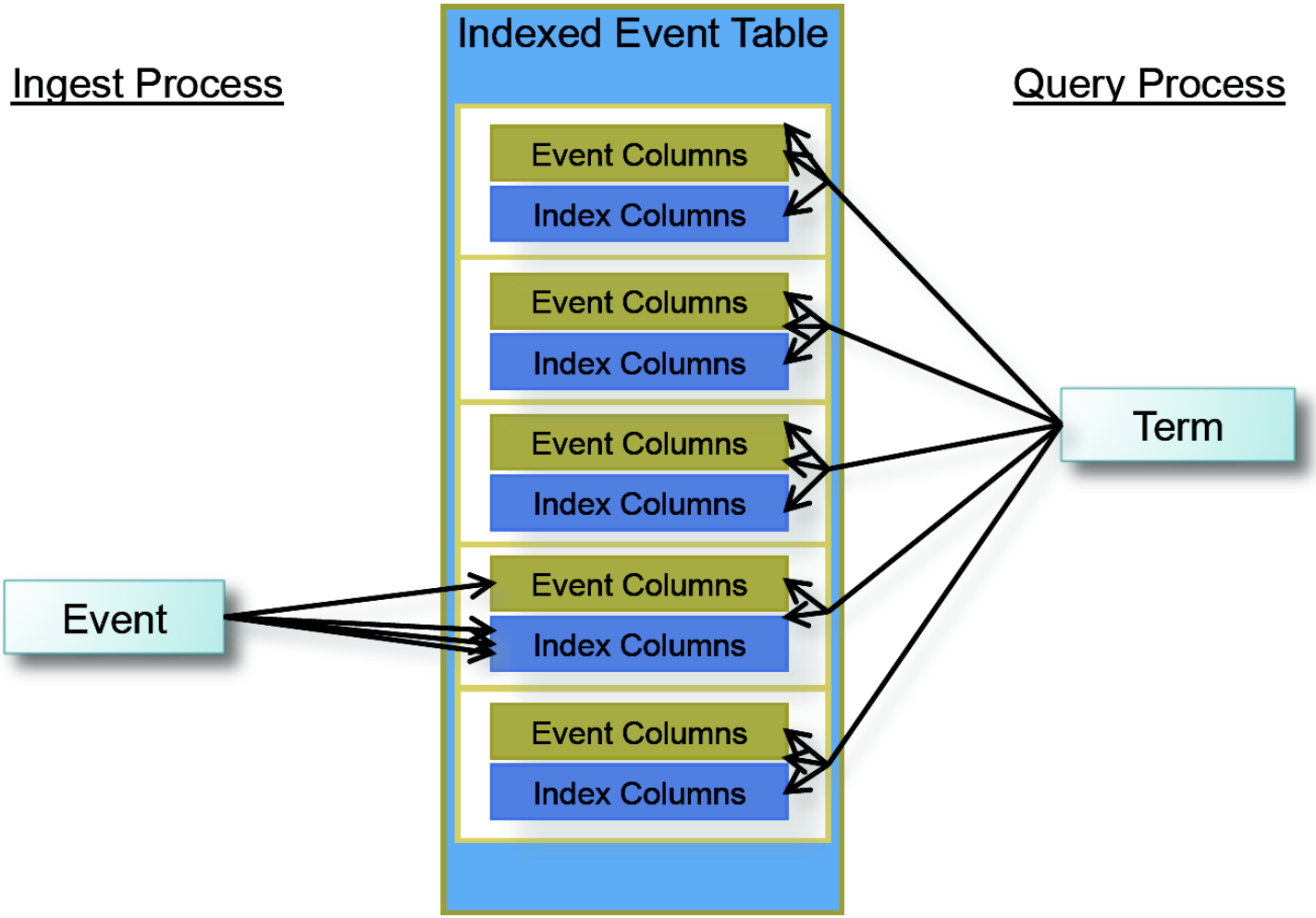
Inverted Index Flow



Document Partitioned Index

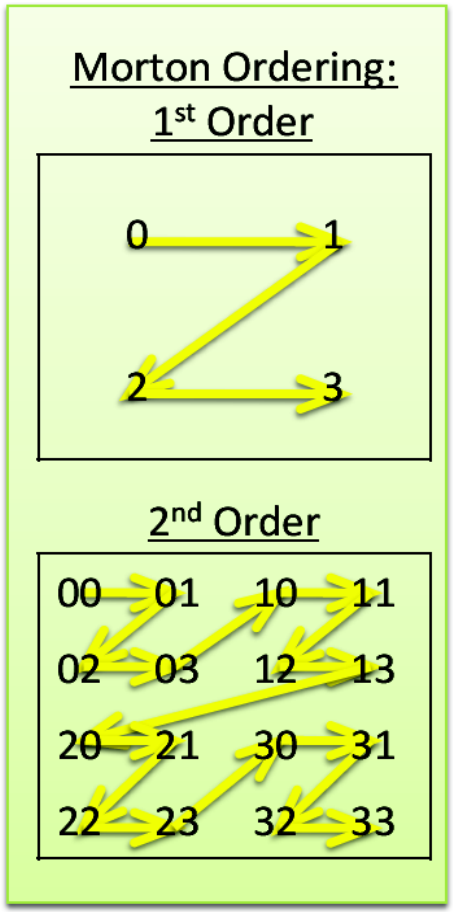


Document Partitioned Flow



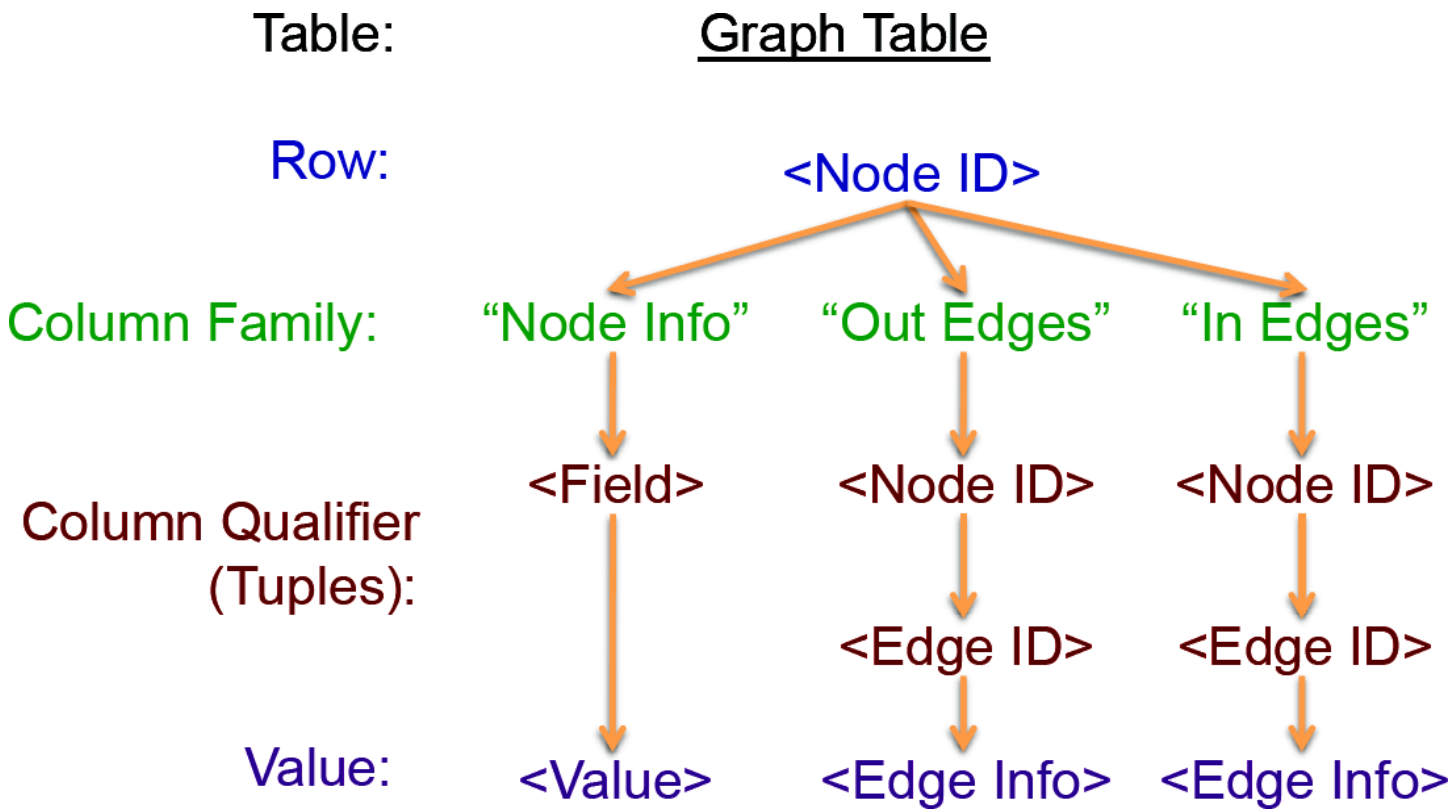
Multidimensional Index

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



Progress

Other Accumulo Features

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