O'REILLY®

Early Release

RAW & UNEDITED

Accumulo

APPLICATION DEVELOPMENT, TABLE DESIGN, AND BEST PRACTICES

Michael Wall,
Aaron Cordova & Billie Rinaldi

Accumulo

Aaron Cordova, Billie Rinaldi, and Michael Wall

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

Accumulo
by Aaron Cordova, Billie Rinaldi, and Michael Wall

Revision History for the :

See *http://oreilly.com/catalog/errata.csp?isbn=* for release details.

Table of Contents

CHAPTER 1
Introduction

Apache Accumulo is a highly scalable, distributed, open source database modeled after Google's BigTable design.[1] Accumulo is built to store up to trillions of data elements and keeps them organized so that users can perform fast lookups. Accumulo supports flexible data schemas and scales horizontally across thousands of machines. Applications built on Accumulo are capable of serving a large number of users and can process many requests per second, making Accumulo an ideal choice for terabyte to petabyte-scale projects.

Recent Trends

Over the past few decades, several trends have driven the progress of data storage and processing systems in certain directions. The first of these is that more data is being produced, at faster rates than ever before. The rate of available data is increasing so fast that data produced in the last few years is greater than the data produced in all years previous.

The second is that the cost of storage has dropped dramatically. Hard drives now store multiple terabytes at roughly the same price as gigabyte drives did a decade ago.

Third is that disk throughput has improved more than disk seek times, for conventional hard drives. Solid state hard drives (SSDs) have the potential to radically alter this balance. However, the advantage of the sequential read performance due to high disk throughput of conventional hard drives over the random read performance due to seek times of SSDs is a large factor in the design of the systems we'll be discussing.

Finally, we've seen a shift from using one processor to multiple processors as increases in single-processor performance have slowed. This is reflected not only in a shift to

1. The BigTable paper can be found at *http://research.google.com/archive/bigtable-osdi06.pdf*

multi-threaded programs on a single server but also to programs distributed over multiple separate servers.

Distributed Applications

In order to effectively use increasing amounts of available data and in light of these recent trends, a few applications have emerged that automatically distribute data and processing over many separate commodity-class servers connected via a network, and that vastly prefer sequential disk operations over random disk seeks. Unlike some distributed systems these applications do not share memory or storage, an approach called a *Shared-Nothing Architecture*. They are designed to handle individual machine failures automatically with no interruption to operations.

Perhaps the most popular of these is *Apache Hadoop*,[2] which allows users to distribute their data over many commodity class machines and to run distributed processing jobs over the data in parallel. This allows data to be processed in a fraction of the time it would take on a single computer. Hadoop uses sequential I/O, opening and reading files from beginning to end during the course of a distributed processing job and writing output to new files in sequential chunks. A graphical representation of *Vertical* scaling versus *Horizontal* or *Shared-Nothing* scaling is shown in Figure 1-1.

Shared-Nothing Architectures

Some distributed applications are built to run on hardware platforms featuring many processors and large amounts of shared random access memory (RAM) and often connect to a Storage Area Network (SAN) via high speed interconnects such as fibre channel to access shared data storage.

In contrast, Shared-Nothing Architectures do not share RAM and do not connect to shared storage, but rather consist of many individual servers, each with its own processors, RAM, and hard drives. These systems are still connected to each other via a network such as gigabit ethernet. Often the individual servers are of the more inexpensive sort and often include cheaper individual components, such as SATA drives rather than SCSI drives.

Technologies such as RAID, which allow several hard drives within a server to be grouped together, are often not used but rather the application layer, such as the Hadoop Distributed Filesystem (HDFS), is relied upon to take care of data replication.

2. Apache Hadoop *http://hadoop.apache.org*

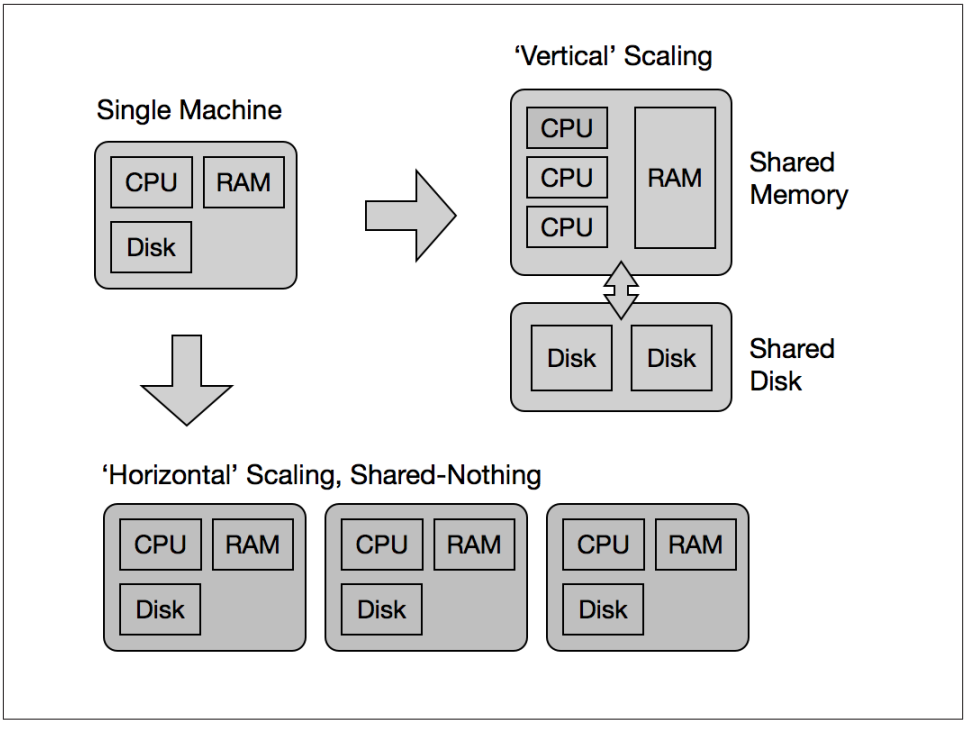

Figure 1-1. Scaling Strategies

Accumulo employs this distributed approach. But unlike Hadoop jobs, where the data is unorganized and each job processes most or all of the data, Accumulo is designed to store data in an organized fashion so users can quickly find the data they need or incrementally add to or update a data set. Accumulo does use disk seeks, but the goal is to perform, ideally, just one disk seek followed by a sequential read for the majority of user data requests. This enables applications to achieve fast, interactive response times even when the data sizes range in the petabytes.

Accumulo includes features that can be used to support a wide variety of scalable applications including storing structured or semi-structured sparse and dynamic data, building rich text search capabilities, indexing geospatial or multi-dimensional data, storing and processing large graphs, and maintaining continuously updated summaries over raw events using server-side programming mechanisms. Instances of Accumulo have been known to run on over a thousand servers, storing over a petabyte of data, and trillions key-value pairs.

Of particular interest is Accumulo's ability to combine data sets of different sensitivity levels into a single application that performs fine-grained filtering over the data and allows users of varying levels of access to query the same data sets and see only what

they are authorized to see. By bringing data together in this way, new questions can be asked across multiple data sets easily. Because the data is physically co-located and distributed along with processing power, sophisticated analysis is possible via server-side programming mechanisms or MapReduce jobs.

Accumulo belongs to a group of applications known as *NoSQL databases*. The term *NoSQL* refers to the fact that these databases support data access methods other than SQL (Structured Query Language), although the engineer who coined the term NoSQL, Carlo Strozzi, has expressed that it may be more appropriate to call these applications *Non-relational* databases.[3]

History

In 2003 Google published a paper describing the *Google File System* (GFS), a distributed file system for storing very large files across many commodity-class servers.[4] Google then published a paper in 2004 describing a simplified distributed programming model and associated fault-tolerant execution framework called *MapReduce*.[5]

In 2006 Google published a paper titled *BigTable: A Distributed Storage System for Structured Data*.[6] That same year a team from Yahoo! released an open source version called Apache Hadoop.

An open source implementation of Google's BigTable called HBase was started by a team at the company Powerset in the fall of 2007. HBase became a top-level Apache project in January 2008.

At the same time, a team[7] of computer scientists and mathematicians at the National Security Agency were evaluating the use of various big data technologies, including Apache Hadoop and HBase, in an effort to help solve the issues involved with storing and processing large amounts of data of different sensitivity levels.

After reviewing existing solutions and comparing the stated objectives of existing open source projects to the agency's goals, the team began a new implementation of Google's BigTable. The team was focused on performance, resilience, and access control of individual data elements. The intent was to follow the design as described in the paper closely in order to build on as much of the effort and experience of Google's engineers

3. "NoSQL Relational Database Management System: Home Page". Strozzi.it. 2 October 2007. Retrieved 29 March 2010.
4. *http://research.google.com/archive/gfs-sosp2003.pdf*
5. *http://research.google.com/archive/mapreduce-osdi04.pdf*
6. *http://research.google.com/archive/bigtable-osdi06.pdf*
7. Authors Billie Rinaldi and Aaron Cordova were part of this team

as possible. At the same time, the team extended the BigTable design with additional features that included a method for labeling each key-value pair with its own access information, called *Column Visibilities*, and a mechanism for performing additional server-side functionality, called *Iterators*.

In 2011 Accumulo became a public open source incubator project hosted by the Apache Software Foundation,[8] and in March 2012 Accumulo graduated to top-level project status.[9]

Data Model

At the most basic level, Accumulo stores key-value pairs on disk, keeping the keys sorted at all times. This allows a user to look up the value of a particular key or range of keys very quickly. Values are stored as byte arrays and Accumulo doesn't restrict the type or size of the values stored.

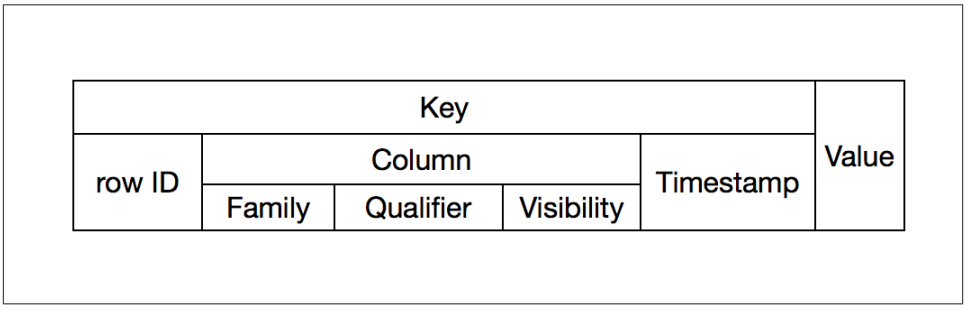

Figure 1-2. Accumulo Key Structure

Row ID
: The Row ID is used to group several key value pairs into a logical *row*. All the key-value pairs that have the same Row ID are considered to be a part of the same row. Row IDs are simply byte arrays. A logical row in Accumulo may contain more data than can fit in memory. Values for multiple columns within a logical row can be changed atomically.

Column Family
: The Column Family is used to group a set of key-value pairs together within a row. Column Families can be put into groups called Locality Groups that are stored together on disk. This allows the data for a Column Family to be read from disk without having to read data for other Column Families. Columnar databases that

8. *http://www.apache.org*
9. *http://accumulo.apache.org*

store data this way can make scanning for a subset of columns much more efficient than row-oriented storage.

Unlike BigTable and HBase, Accumulo Column Families need not be declared before being used, and Accumulo tables can have a very high number of Column Families if desired.

Column Qualifier
: Column Qualifiers are used to distinguish individual columns within a Column Family. Because keys in Accumulo point to exactly one value, Column Qualifiers can be used to create unique keys for storing multiple values in a Column Family.

Column Visibility
: Column Visibility is an element that was added to the original BigTable data model by the Accumulo developers to allow key-value pairs to be filtered based on a set of user access tokens. Column Visibilities must conform to a particular grammar and syntax that is described in Chapter 3 (Writing Applications).

Timestamp
: The timestamp functions as a kind of version number for keys. Two keys that are identical except for the timestamps can be considered to be two versions of the *same* key. Timestamps are usually numbers representing seconds since UNIX epoch and are sorted in descending order so that the newest version of a key appears first when keys are read in order.

  It is possible to specify the timestamp when inserting a new key into Accumulo, but this should only be done for advanced applications as timestamps are used in determining the ordering of insertions and deletions. If the timestamp is not specified by the client, the Tablet Server that receives the key-value pair will use the time at which the data arrived as the timestamp.

Value
: Values comprise the *value* part of the key-value pair. As such, they are not sorted. Values are byte arrays that can hold a wide variety of data. In practice, Values should be no bigger than a few 10s of megabytes in size, as several Values may need to be pulled into memory at once. Because the value lives at the intersection of a row and column in a spreadsheet, values are sometimes called *cells*.

Typically information that you are going to query or search on should not go in the value. These things are much more efficient to find somewhere in the key, again because the key is sorted. We will go over this more in Chapter 3 (Writing Applications).

Sometimes the term *cell-level security* is used when discussing Accumulo's ability to allow every value to have its own security label, which is stored in the Column Visibility element of the key. The term *cell-level* is used to contrast the granularity of Accumulo's security model with row-level or column-level security in which one can control access to all the data in a row or all the data in a column.

If you are coming from a relational database background, it might be confusing to think of a "row" in Accumulo as a set of key-value pairs. Looking at data retrieved in the Accumulo Shell, which we touch on first in "Shell" on page 11, a "row" will actually be many lines on the screen. Similarly, one "column" can be multiple lines with different Column Qualifiers. Table 1-2 may be a more familiar representation of the data, and you can see how it might translate into Accumulo in Table 1-1. In this example, a row, defined as a set of key-value pairs, is analogous to a record in a relational database. Everything with the same Row ID contains information about a given record.

Tables

Key-value pairs are grouped into tables. Users can apply some settings and storage at the table level to control the behavior and management of the data. The key-value pairs within tables are distributed automatically across multiple machines in a cluster.

More information about Tables will be given in Chapter 4 (Internals). For now, just think of a table as way to group Accumulo rows and get some benefits from the underlying architecture.

Sort Order

Keys are sorted first by the Row ID, and within a Row ID, by the column family and so on. Each element is a byte array sorted in ascending order lexicographically, except for the timestamp which is a number sorted in descending order.

The layout of a table can be thought of as simply a set of sorted key-value pairs.

Table 1-1. Table layout

Row ID	Column Family	Column Qualifier	Column Visibility	Timestamp	Value
bob jones	contact	address	billing	13234	123 any street
bob jones	contact	city	billing	13234	anytown
bob jones	contact	phone	billing	13234	555-1212
bob jones	purchases	sneakers	billing&inventory	13255	$60
fred smith	contact	address	billing	13222	444 main st.
fred smith	contact	city	billing	13222	othertown

Row ID	Column Family	Column Qualifier	Column Visibility	Timestamp	Value
fred smith	purchases	glasses	billing&inventory	13201	$30
fred smith	purchases	hat	billing&inventory	13267	$20

In Table 1-1, all the keys that share a row ID represent the data for one row. In this example, we've organized all the information about customers under their name. The names of the columns are found in the Column Family and Column Qualifier. Note that rows do not need to have the same set of columns and that Values can store a variety of data types.

Tables over a certain size (by default, 1 GB) are automatically partitioned into *tablets*, non-overlapping ranges of row IDs.

The data model labels portions of the key "row" and "column" partially because users are used to thinking of data in this way. If one were to lay out the data in a two-dimensional table by rows and columns some cells would be empty as in Table 1-2.

Note that not all rows have data for every column. This is what is meant by being able to store sparse data — Accumulo doesn't need to represent empty columns within a row, they simply do not appear in the list of key-value pairs.

A single row could have millions of columns and there are no constraints on the set of columns that exist within a table. This capability is very useful for applications that require storing high-dimensional data and is especially good for storing sparse high-dimensional data.

Table 1-2. A 2D Spreadsheet view of a sparse table

	contact	**contact**	**contact**	**purchases**	**purchases**	**purchases**
	address	**city**	**phone**	**sneakers**	**glasses**	**hat**
	billing	**billing**	**billing**	**billing& inventory**	**billing& inventory**	**billing& inventory**
bob jones	123 any street	anytown	555-1212	$60		
fred smith	444 main st.	othertown			$30	$20

Reading data

The Accumulo client library is used to locate keys and return their values or to scan across a range of key-value pairs.

Rather than providing a query language such as SQL, Accumulo provides a simple API and control over data layout, so that by designing tables carefully, many concurrent user requests can be satisfied very quickly with a minimal amount of work done at read time. As such, Accumulo's read API is simple and straightforward.

As you would expect from a key-value store, clients can provide a key and look up the associated value, if it exists. Instead of returning one value, however, clients can opt to

scan a range of key-value pairs beginning at a particular key. The performance difference between looking up and retrieving a single value versus scanning say a few hundred kilobytes of key-value pairs is fairly small, since the cost of reading that amount of data sequentially is dominated by the disk seek time.

This pattern allows clients to design rows such that the data required for a given request can be sorted nearby in the same table. Because rows may not all have the same columns, applications can be designed to take advantage of whatever data is available, potentially discovering new information in new columns along the way.

The ability to discover new information via scanning is valuable for applications that wish to combine information about similar subjects from difference sources, where each source may not contain the same information about each subject.

Furthermore, it is up to the application to interpret the columns and values retrieved. Some applications store simple strings or numbers, while others store JSON or serialized programmatic objects. Some applications store map tile images in values and assemble the tiles retrieved into a user-facing web interface, as Google Maps uses BigTable.

Introduction to the Read and Write API

Accumulo is written in Java and provides a Java client library. Clients in other languages can communicate with Accumulo via the provided Thrift Proxy. All clients use three basic classes to communicate with Accumulo.

BatchWriter
: All new inserts, updates, and deletes are packaged up into Mutations and given to a BatchWriter. A Mutation is a set of changes to a single row. Each BatchWriter sends Mutations to a single table. The BatchWriter knows how the table is split into tablets and which servers the tablets are assigned to. Using this information, the BatchWriter efficiently groups Mutations into batches to increase write throughput. The BatchWriter is multi-threaded and the tradeoff between latency and throughput can be tuned.

Scanner
: Key-Value pairs are read out of a table using a Scanner. A Scanner may start at the beginning of a table or at a particular key, and may stop at the end of the table or a given key. After seeking to the initial key, Scanners proceed to read out key-value pairs sequentially in key-order until reaching the end of the table or the specified ending key. Scanners can be configured to read only certain columns. Additional configuration for a Scanner can be made to apply specific Iterators, or specific options to Iterators, to alter the set of key-value pairs returned from a particular Scanner.

BatchScanner
: When multiple ranges of keys are to be read from a table, a BatchScanner can be used to read the key-value pairs for the ranges using multiple threads. The ranges are grouped by Tablet Server to maximize the efficiency of communication between threads and Tablet Servers. This can be useful for applications whose design requires many individual scans to answer a single question. In particular, tables designed for working with time series, secondary indices, and complex text search can all benefit from using BatchScanners.

Much more detail on developing applications for Accumulo is found in the Chapter 3 (Writing Applications) chapter.

Approach to Rows

Accumulo takes a slightly different approach to rows in the client API than HBase does. Accumulo's read API is designed to stream key-value pairs to the client rather than to package up all the key-value pairs for a row into a single unit before returning the data to the user.

This is often less convenient than working with data on a row-by-row basis, and applications that wish to work with entire rows can do additional configuration to assist with this. The upside is that rows in an Accumulo table may be very large and do not need to be able to fit in the memory of the Tablet Server or the client. Working with key-value pairs can come in handy when rows are being generated from underlying data and the number of columns per row may be unknown or may vary widely, as can happen when building secondary indices.

Exploiting Sort Order

The key to taking full advantage of Accumulo's design is to exploit the fact that Accumulo keeps keys sorted. This requires application designers to determine a way to order the data such that most user queries can be satisfied with one or a small number of scans consisting of a lookup into a table and fetching one or more sequential key-value pairs.

A single scan is able to perform this lookup and return one or even hundreds of key-value pairs often in less than a second even when tables contain trillions of key-value pairs. Applications that understand and use this property can achieve sub-second response times for most user requests without having to worry about performance degrading as the amount of data stored in the system increases dramatically.

This sometimes requires creative thinking in order to discover a key design that works for a particular application. A good example of this is the way Google describes the row ID of their WebCrawl table in the BigTable paper. In this table, the intent is to provide users with the ability to look up information about a given website, identified by the hostname. Since hostnames are hierarchical and since users may want to look at a specific hostname or all hostnames within a domain, Google chose to transform the host-

name to support these access patterns by reversing the order in which domain name components are stored under the row ID in Table 1-3.

Table 1-3. Google's WebCrawl row design

Row ID
com.google.analytics
com.google.mail
com.google.maps
com.microsoft
com.microsoft.bing
com.microsoft.developers
com.microsoft.search
com.microsoft.www
com.yahoo
com.yahoo.mail
com.yahoo.search
com.yahoo.www

Achieving optimal performance also depends on user requests being able to be satisfied by not having to filter out or ignore a large amount of key-value pairs as a part of the scan.

Because developers have such a high degree of control over how data is arranged there are a wide variety of options when designing tables. We cover these in depth in the Chapter 6 (Best Practices).

Shell

Accumulo has a Shell that can be used to create and delete tables, inspect and change configuration settings, and scan and edit tables. The Shell is useful for doing simple scans to verify tables are as they should be and for simple administration.

Following are a few quick examples of the types of things that can be done from the shell.

```
host:Software user$ accumulo/bin/accumulo shell -u root
Enter current password for 'root'@'acc': ******

Shell - Apache Accumulo Interactive Shell
-
...
- type 'help' for a list of available commands
-
```

Users can retrieve a list of tables and create new tables.

```
root@acc> tables
!METADATA
trace

root@acc> createtable test

root@acc> tables
!METADATA
test
trace
```

Authorizations granted to a user can be controlled via the shell.

```
root@acc test> getauths

root@acc test> setauths -s billing,inventory
root@acc test> getauths
billing,inventory
```

Small amounts of data can be inserted.

```
root@acc test> help insert
usage: insert <row> <colfamily> <colqualifier> <value> [-?] [-l <expression>] [-t <timestamp>]
description: inserts a record
  -?,--help                                 display this help
  -l,--authorization-label <expression>     formatted authorization label expression
  -t,--timestamp <timestamp>                timestamp to use for insert

root@acc test> insert -l billing "bob jones" contact phone 555-1212
root@acc test> insert -l billing "bob jones" contact address "123 anystreet"
root@acc test> insert -l billing "bob jones" contact city "anytown"
root@acc test> insert -l billing|inventory "bob jones" purchases sneakers "$60"

root@acc test> insert -l billing "fred smith" contact address "444 main street"
root@acc test> insert -l billing "fred smith" contact city "othertown"
root@acc test> insert -l billing|inventory "fred smith" purchases glasses "$30"
root@acc test> insert -l billing|inventory "fred smith" purchases hat "$20"
```

Users can scan tables in a variety of ways. To see an entire table, applying all authorizations available, simply type *scan*.

```
root@acc test> scan
bob jones contact:address [billing]    123 anystreet
bob jones contact:city [billing]    anytown
bob jones contact:phone [billing]    555-1212
bob jones purchases:sneakers [billing|inventory]    $60
fred smith contact:address [billing]    444 main street
fred smith contact:city [billing]    othertown
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $20
```

Scans can be restricted to a particular set of authorization tokens. In each case the user will only see the key-value pairs whose Column Visibility is logically satisfied by the set of supplied tokens.

```
root@acc test> scan -s billing
bob jones contact:address [billing]    123 anystreet
bob jones contact:city [billing]    anytown
bob jones contact:phone [billing]    555-1212
bob jones purchases:sneakers [billing|inventory]    $60
fred smith contact:address [billing]    444 main street
fred smith contact:city [billing]    othertown
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $20

root@acc test> scan -s inventory
bob jones purchases:sneakers [billing|inventory]    $60
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $20
```

Users can begin a scan at a particular row by specifying a start key.

```
root@acc test> scan -b "fred smith"
fred smith contact:address [billing]    444 main street
fred smith contact:city [billing]    othertown
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $20
```

Key-value pairs can be "updated" via inserting a new version of the value. By default Accumulo returns the latest value so new inserts appear to work like overwrites when a key already exists.

```
root@acc test> insert -l billing|inventory "fred smith" purchases hat "$25"

root@acc test> scan -b "fred smith"
fred smith contact:address [billing]    444 main street
fred smith contact:city [billing]    othertown
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $25
```

Users can scan for only certain column families within a row.

```
root@acc test> scan -b "fred smith" -c purchases
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $25
```

Users can scan for a fully qualified column by specifying the Column Family and Column Qualifier to see a single value.

```
root@acc test> scan -b "fred smith" -c purchases:glasses
fred smith purchases:glasses [billing|inventory]    $30
```

Scans can be performed over all rows but retrieving only certain column families. This is akin to the *SELECT* clause of SQL.

```
root@acc test> scan -c purchases
bob jones purchases:sneakers [billing|inventory]    $60
fred smith purchases:glasses [billing|inventory]    $30
fred smith purchases:hat [billing|inventory]    $25
```

Finally, tables can be deleted.

```
root@acc test> deletetable test
Table: [test] has been deleted.

root@acc> tables
!METADATA
trace
```

For a complete listing of Shell commands see Appendix A: Shell Commands.

Architecture Overview

The basic architecture of an Accumulo cluster is as follows. We cover the components in more depth the Chapter 4 (Internals) section.

Accumulo is a distributed application that depends on Apache Hadoop for storage and Apache Zookeeper for configuration.

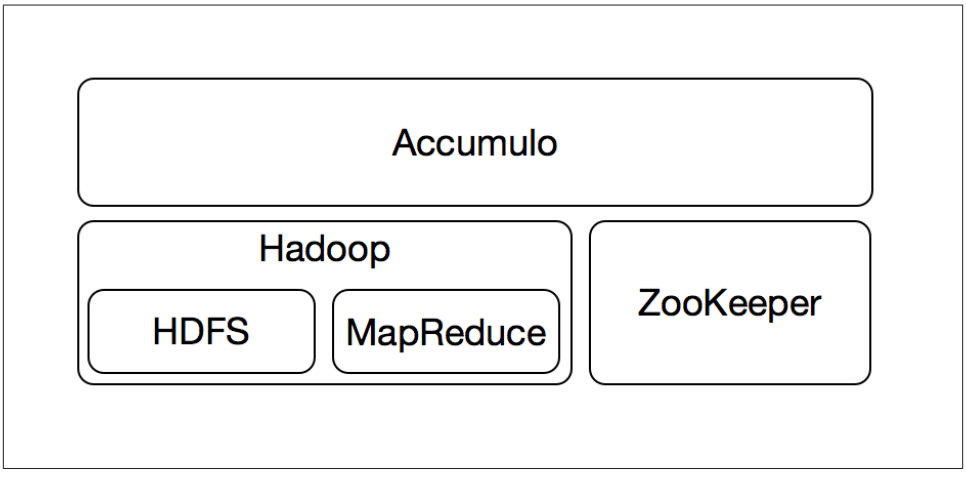

Figure 1-3. Accumulo Architecture

Since Accumulo is based on Google's BigTable it uses some of the same names for components as BigTable does, but there are some differences.

Apache Accumulo	BigTable	Apache HBase
Tablet	Tablet	Region
Tablet Server	Tablet Server	Region Server
Minor Compaction	Minor Compaction	Flush
Major Compaction	Merging Compaction	Minor Compaction
(Full) Major Compaction	Major Compaction	Major Compaction

Write-Ahead Log	Commit Log	Write-Ahead Log
HDFS	GFS	HDFS
Hadoop MapReduce	MapReduce	Hadoop MapReduce
MemTable	MemTable	MemStore
RFile	SSTable	HFile
Zookeeper	Chubby	Zookeeper

Zookeeper

Zookeeper is a highly available, highly consistent distributed application in which all the data is replicated on all the machines so that if one machine fails, clients reading from Zookeeper can quickly switch over to one of the remaining machines. In practice Zookeeper instances tend to consist of 3 or 5 machines. Using an even number of Zookeeper hosts would provide less resilience than using one fewer machine.

Accumulo stores persistent, essential information in Zookeeper. If the user deletes the Zookeeper data, Accumulo will no longer run.

Accumulo uses Zookeeper to store configuration and status information and to track changes in the cluster. Zookeeper is also used to help clients begin the process of locating the right servers for the data they seek.

Hadoop

In the same way Google's BigTable stores its data in a distributed file system called GFS, Accumulo stores its data in the Hadoop Distributed File System, or HDFS. Accumulo relies on HDFS to provide persistent storage, replication, and fault tolerance. Having a separate storage layer allows Accumulo to balance the responsibility for serving portions of tables independently of where they are stored, although data tends to be served from the same server on which it is stored.

Like Accumulo, HDFS is a distributed application, but one that allows users to view a collection of machines as a single scalable file system. HDFS files can be very large in size, up to terabytes per file. HDFS automatically breaks these files into blocks, by default 64 MB in size, and distributes these blocks across the cluster uniformly. In addition, each block is replicated on multiple machines. The default replication factor is three in order to avoid losing data when one machine or even an entire rack of servers becomes unavailable.

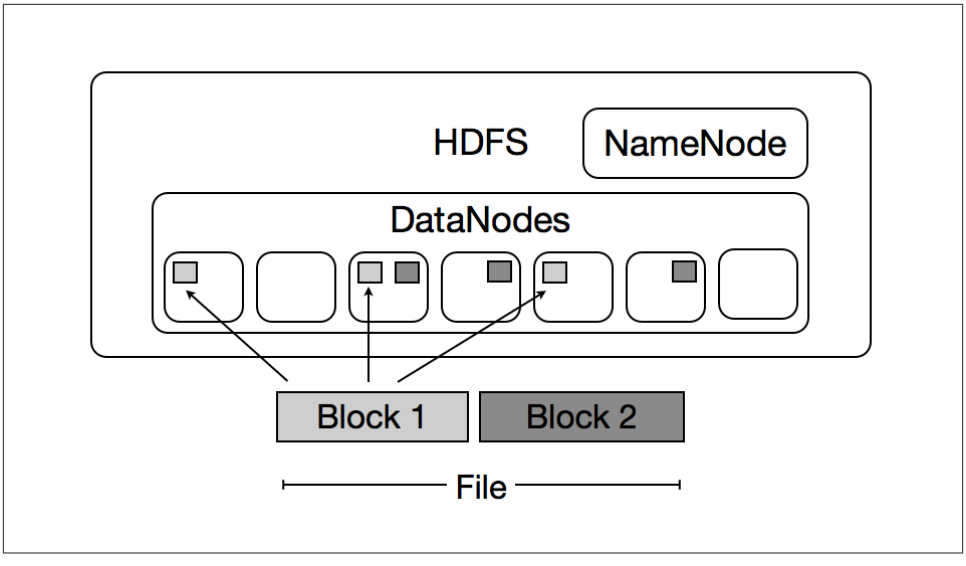

Figure 1-4. Hadoop Distributed File System

Accumulo

An Accumulo instance consists of several types of processes running on one to thousands of machines.

Analogous to HDFS files, Accumulo tables can be very large in size, up to 10s of trillions of key-value pairs. Accumulo automatically partitions these into tablets and assigns responsibility for hosting tablets uniformly across servers called *Tablet Servers*.

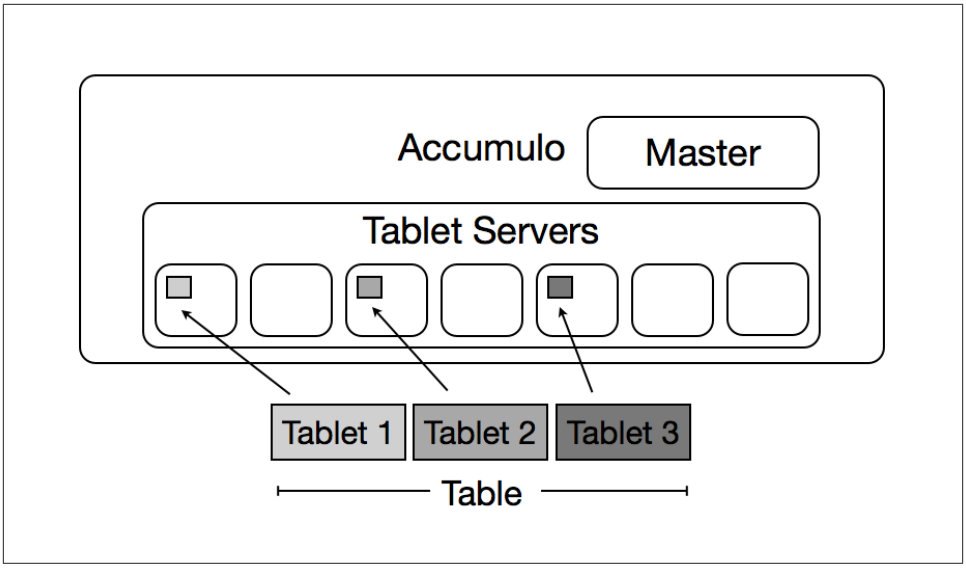

Figure 1-5. Accumulo

However, unlike HDFS block replicas, Accumulo tablets are assigned to exactly one Tablet Server at a time. This allows one server to manage all the reads and writes for a particular range of keys, allowing reads and write to be highly consistent, as no synchronization has to occur between Tablet Servers. When a client writes a piece of information to a row, clients reading that row immediately afterward will see the new information.

Tablet Servers
: Tablet Servers host a set of tablets and are responsible for all the writes and reads for those tablets. Clients connect directly to Tablet Servers to read and write data. Tablet Servers may host hundreds or even thousands of tablets, each consisting of about 1 GB of data or more. Tablet Servers store data written to these tablets in memory and in files in HDFS, and handle scanning data for clients, applying any additional filtering or processing the clients request.

Loggers
: Accumulo versions 1.4 and older use logger processes to record each new write in an unsorted Write-Ahead Log on disk that can be used to recover any data that was lost from the memory of a failed Tablet Server. In Accumulo 1.5, there are no longer dedicated logger processes. The Write-Ahead Logs are written directly to files in HDFS.

Master
: Every Accumulo cluster has one active Master process that is responsible for making sure all tablets are assigned to exactly one Tablet Server at all times and that tablets

are load-balanced across servers. The Master also helps with certain administrative operations such as startup, shutdown, and table and user creation and deletion.

Unlike the HDFS NameNode, Accumulo's Master may fail without causing interruption to Tablet Servers and clients. If a Tablet Server fails while the Master is down, some portion of the tablets will be unavailable until the Master is recovered or until a Master process can be started on a new machine.

It is possible to configure Accumulo to run multiple Master processes. Whichever process obtains a Master Zookeeper lock first will be the active Master, and the remaining processes will watch the lock so that one of them can take over if the active Master fails.

Garbage Collector
: The Garbage Collector process finds files that are no longer being used by any Tablets and deletes them from HDFS to reclaim space.

Monitor
: Accumulo ships with an informative monitor that reports cluster activity and logging information into one web interface. This monitor is useful for verifying that Accumulo is operating properly and for helping understand and troubleshoot cluster and application performance.

Client
: Accumulo provides a Java client library for use in applications. Many Accumulo clients can read and write data from an Accumulo instance simultaneously. Clients communicate directly with Tablet Servers to read and write data. Occasionally clients will communicate with Zookeeper and with the Accumulo Master for certain table operations, but no data is sent or received through Zookeeper or the Master.

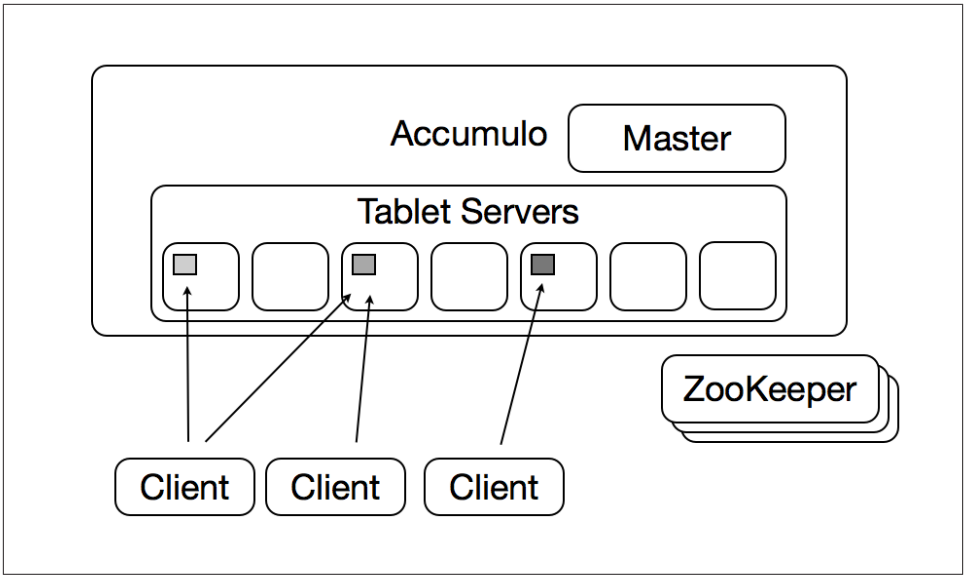

Figure 1-6. Accumulo Clients

Thrift Proxy
: As of version 1.5, Accumulo provides a Thrift Proxy that can be used to develop clients in languages other than those that run on the JVM. These other clients can connect to the Thrift Proxy which communicates with the Accumulo cluster and allows data to be read and written by these other clients.

A typical cluster

A typical Accumulo cluster consists of a few *control nodes* and a few to many *worker nodes*.

Control nodes include:

- 1, 3 or 5 machines running Zookeeper
- Ideally 2 machines running NameNode processes, one active, one for failover
- 1 to 2 machines running Accumulo Master, garbage collector, and/or monitor
- 1 optional machine running a Hadoop JobTracker process if MapReduce jobs are required

Each worker node typically includes:

- 1 HDFS DataNode process for storing data
- 1 Tablet Server process for serving queries and inserts
- 1 optional Hadoop TaskTracker for running MapReduce jobs

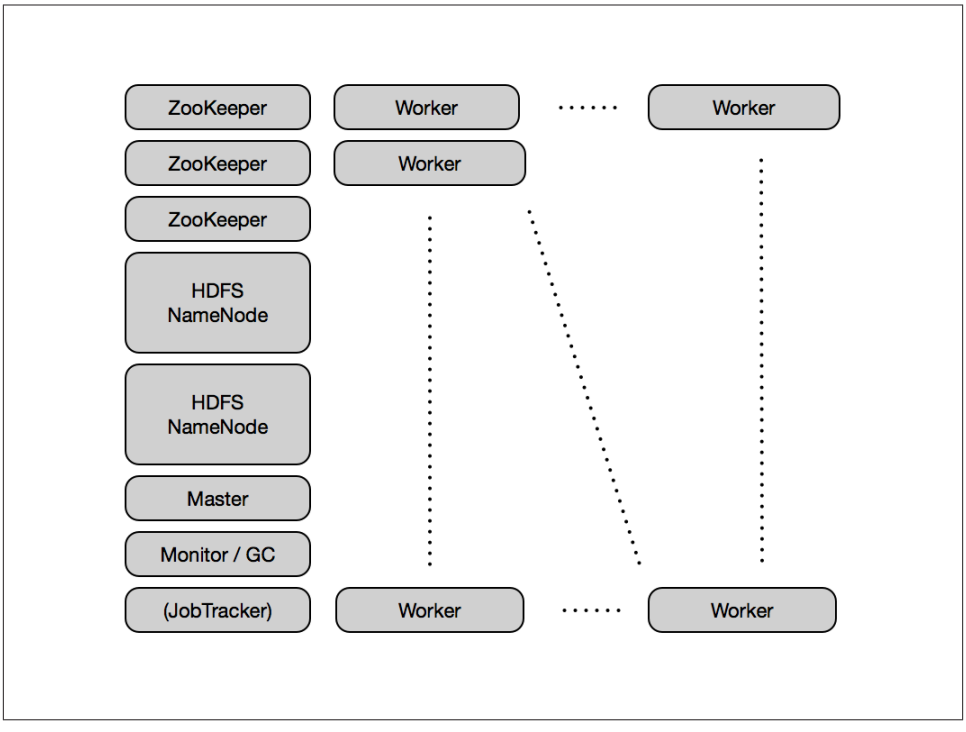

Figure 1-7. A Typical Cluster

Features

Automatic Data Partitioning

Accumulo tables can be very large, up to petabytes in size. To manage this, Accumulo automatically partitions tables into uniform pieces called *Tablets*. This process is called *splitting*. Users can tune the splitting process, but they don't have to worry about choosing a good key on which to partition because Accumulo automatically finds good split points.

High Consistency

Accumulo provides a highly consistent view of the data. Tablets are assigned to exactly one Tablet Server at a time. An update to a particular key's value (more accurately, a new version of the key's value) is immediately reflected in subsequent reads because those updates and reads go to the same server.

Other NoSQL systems allow writes for a particular key to happen on more than one server, and consistency is achieved via communication between these servers. Because this communication is not instantaneous, these systems are considered *eventually consistent*. One advantage of eventually consistent systems is that a single instance of the database can run over geographically disparate data centers and writes to some servers can continue even if those servers cannot communicate with all of the other servers participating in the cluster.

An Accumulo instance is designed to be deployed within a single data center and to provide a highly consistent view of the data. The advantage of high consistency is that application logic can be simplified.

Automatic Load Balancing

The Accumulo Master automatically balances the responsibility for serving tablets across Tablet Servers. When one Tablet Server has more tablets than another, the Master will instruct the overloaded Tablet Server to stop serving a tablet and for the underloaded Tablet Server to begin hosting that tablet.

Scalability

Accumulo is considered a *horizontally scalable* application, meaning that one can increase the capabilities of the system by adding more machines, rather than by replacing existing machines with bigger, more capable machines, which is termed *vertical scaling*. New machines joining an Accumulo cluster begin participating in the cluster very quickly as no data movement is required for these new machines to start hosting tablets and the reads and writes associated with them.

Accumulo is also *scalable* in that it works well at large scale, meaning, on clusters consisting of thousands of machines hosting petabytes of data.

A major benefit to building on Accumulo is that an application can be written and deployed on a small cluster when the amount of data and the number of concurrent writes and reads is low, and as data or read-write demand grows the Accumulo cluster can be expanded to handle more data and reads without having to rewrite the application.

Failure Tolerance and Automatic Recovery

Like Hadoop, Accumulo is designed to survive single server failures and even the failure of a single rack. If a single Accumulo Tablet Server fails, the Master notes this and reassigns its tablets to the remaining Tablet Servers. Accumulo clients automatically manage the failover from one Tablet Server to another. Application developers do not need to worry about retrying their operations simply because a machine fails.

In a large cluster these types of failures will be commonplace and Accumulo does a lot of work to minimize the burden on application developers as well as administrators so that a single instance running on thousands of machines is tractable.

Fine-Grained Security

One of Accumulo's unique features is the ability to label each key-value pair with its own visibility expression. This allows data of different sensitivity levels to be stored and indexed in the same physical tables, and for users of varying degrees of access to read these tables without seeing any data they are not authorized to see.

Accumulo extends the basic BigTable data model by adding an element called *Column Visibility*:

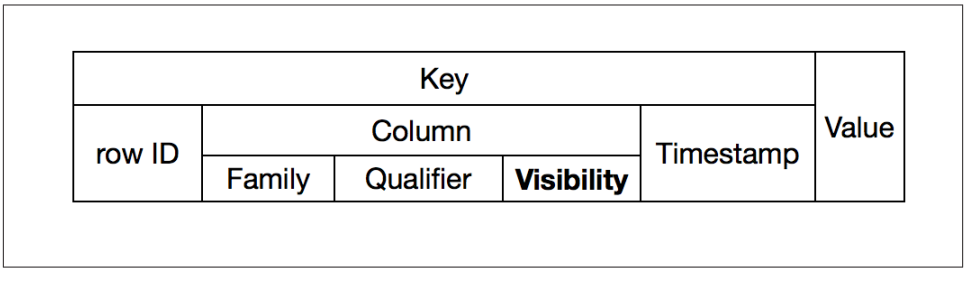

Figure 1-8. Column Visibilities

The Column Visibility can contain a security expression that consists of tokens and boolean operators, such as

```
audit|(finance&reporting)
```

Every time a user requests some data, Accumulo clients pass in the set of tokens that the user possesses and those tokens are applied to the security expressions. The key-value pairs whose security expressions are satisfied by the user's tokens are returned as part of the scan. Those that fail are never shown to the user.

This allows developers to build applications that can store and read data from multiple sources with varying sensitivity levels and control access with fine granularity.

Accumulo will provide the filtering functionality, but other considerations must be taken in order to build a secure application. Specifically the data must be labeled properly when written and clients must present the correct access tokens when querying. These issues are discussed in detail in Chapter 3 (Writing Applications) and Chapter 5 (Administration).

Support for Analysis

Storing and making large amounts of data searchable is really only part of the solution to the problem of taking full advantage of big data. Often data needs to be aggregated, summarized, or modeled in order to be fully understood and utilized. Accumulo provides a few mechanisms for performing analysis on data in tables.

Iterators

The first of these mechanisms are Iterators, which enable custom aggregation and summarization within Tablet Servers to allow users to maintain result sets efficiently and store the data at a higher level of abstraction. They are called Iterators because they iterate over key-value pairs and allow developers to alter the data before writing to disk or returning information to users.

There are various types of Iterators that range from filtering to simple sums to maintaining a set of statistics. These are covered in "Programming Tables" on page 85, Writing Applications: Programming Tables.

Developers have used Iterators to incrementally update edge weights in large graphs for applications such as social network analysis or computer network modeling. Others have used Iterators to build complex feature vectors from a variety of sources to represent entities such as website users. These feature vectors can be used in machine learning algorithms like clustering and classification to model underlying groups within the data or for predictive analysis.

MapReduce Integration

Beyond Iterators, Accumulo supports analysis via integration with the popular Hadoop MapReduce framework. Accumulo stores its data in Hadoop's Distributed File System (HDFS) and can be used as the source of data for a MapReduce job or as the destination of the output from a MapReduce job. MapReduce jobs can either read from Tablet Servers using the Accumulo client library, or from the underlying files in which Accumulo stores data via the use of specific MapReduce input and output formats.

In either case, Accumulo supports the type of data locality that MapReduce jobs require, allowing MapReduce workers to read data that is stored locally rather than having to read it all from remote machines over the network.

Data Life-Cycle Management

Accumulo provides a good degree of control over how data is managed in order to comply with space, legal, or policy requirements.

In addition, the timestamps that are part Accumulo's key structure can be used with Iterators to age data off according to a policy set by the administrator. This includes

aging off data older than a certain amount of time from now, or simply aging off data older than a specific date.

Timestamps can also be used to distinguish between two or more versions of otherwise identical keys. The built-in Versioning Iterator can be configured to allow any number of versions, or only a specific number of versions to be stored. In Google's original BigTable paper, they described using timestamps to distinguish between various versions of the web as they crawled and stored it from time to time.

By building this functionality into the database, work that otherwise must be done in a batch-oriented fashion involving a lot of reading and writing data back to the system can be performed incrementally and efficiently.

Compression

Accumulo compresses data by default using several methods. One is to apply a compression algorithm such as GZip or LZO to blocks of data stored on disk. The other is a technique called *relative-key encoding* in which the shared prefixes of a set of keys is stored only once, and the following keys only need express the unique changes to the initial key. Compressing data in this way can improve I/O as reading compressed data and doing decompression can be faster than reading uncompressed data and not doing decompression. Compression also helps offset the cost of the block replication that is performed by HDFS.

The BigTable paper describes two types of compression, one of which compresses long common strings across a large window, the other does compression over small windows of data. These types of custom compression are not implemented in Accumulo.

Accumulo and Other Data Management Systems

Application developers and systems engineers face a wide range of choices for managing their data today. Often the differences between these options are subtle and require a deep understanding of technologies' capabilities as well as the problem domain. To help in deciding when Accumulo may or may not be a good fit for a particular purpose we compare Accumulo to some other popular options.

Comparisons to Relational Databases

Relational databases, by far the most popular type of database in use today, have been around for several decades and serve a wide variety of uses. Understanding the relative strengths and weaknesses of these systems is useful for determining how and when to use those versus Accumulo.

SQL

One of the strengths of relational databases is that they implement a set of operations known as *relational algebra* codified in a popular language called *Structured Query Language* or SQL. SQL allows users to perform rich and complex operations at query time including set intersection, joins, aggregations, and sorting. Relational databases are heavily optimized to perform these operations at query time.

One challenge of using SQL is the performing this work at query time on a large amount of data. Relational Massively Parallel Processing (MPP) databases approach this by dividing the work to perform SQL operations among many servers. The approach taken by Accumulo is to encourage aggressive pre-computation where possible, often using far more storage to achieve the space-time tradeoff, in order to minimize the work done at query time and maintain fast lookups even when storing petabytes of data.

Space-Time Tradeoff and Cheap Space

In computer science, the space-time tradeoff refers to the fact that one can use more space to store the results of computation in order to reduce the time required to get answers to users. Conversely, one can save space by waiting until users ask and computing answers on the fly.

Over the past decade the cost of storage has dropped dramatically. As a result Accumulo applications tend to pre-compute as much as possible, often denormalizing into one table what would be stored as two or more tables in a relational database.

When applications are designed to support answering analytical questions about entities of interest, it is common to pre-compute the answer for all entities periodically, or to update the answers via iterators when new raw data is ingested, so that queries can consist of a simple, very fast lookup.

Transactions

In addition, many relational databases provide very strong guarantees around data updates, commonly termed ACID, for Atomicity, Consistency, Isolation, and Durability.

ACID

Atomicity
: Either all the changes in a transaction are made or none are made. No partial changes are committed.

Consistency
: The database is always in a consistent state. This means different things in different contexts. For databases in which some rows may refer to others, consistency means that a referenced row must exist.

Isolation
: Each transaction is made independent of other transactions. Changes appear the same whether done serially or concurrently.

Durability
: Changes are persistent and survive certain types of failure.

Accumulo guarantees these properties for a single mutation (by design, a set of changes for a single row) but does not provide support for atomic updates across multiple rows, nor does Accumulo maintain consistent references between rows.

In relational databases these properties are delivered via several mechanisms, one of which is a *transaction* which bundles a set of operations together into a logical unit. Transactions are important for supporting *operational workloads* such as maintaining information about inventory, keeping bank accounts in order, and tracking the current state of business operations. Transactions can contain changes to multiple values within a row, changes to values in two or more rows in the same table, or even updates to multiple rows in multiple tables. These types of workloads are considered *On-Line Transaction Processing* or OLTP.

Normalization

If you store multiple copies of the same data in different places, it can difficult to ensure a high degree of consistency. You might update the value in one place, but not the other. There, storing copies of the same values should be avoided. Values that don't have a one-to-one relationship to each other are often divided into separate tables that keep pointers between themselves. For example, a person typically only has one birth date, so you can store birth date in the same table as first name and other one-to-one data. But a person may have many nicknames or favorite songs. This type of one-to-many data is stored in a separate table. There is a well-defined process for doing this called *normalization*.

Another group of workloads is termed *On-Line Analytical Processing* or OLAP. Relational databases have been used to support these kinds of workloads as well. Often analysis takes the approach of looking at snapshots of operational data, or simply may bring together reference data that doesn't require updates, but rather efficient reads and aggregation capabilities.

Because OLAP workloads require fewer updates, tables are often pre-combined, or *denormalized*, to cut down on the operations that are carried out a query time. This is

another example of the space-time tradeoff, where an increase in storage space used reduces the time to get data in the format requested.

Accumulo and other NoSQL databases do not implement relational algebra. Accumulo provides ACID guarantees, but on a more limited basis. The only *transactions* allowed by Accumulo are inserts, deletes, or updates to multiple values within a single row. These transactions are atomic, consistent, isolated and durable. But a set of updates to multiple rows in the same table, or rows in different tables do not have these guarantees.

Accumulo is therefore used for massive operational workloads that can be performed via single-row updates, or for massive OLAP workloads.

Comparisons to other NoSQL Databases

Compared to other NoSQL databases, Accumulo has some features that make it especially dynamic and scalable, and a few features not found anywhere else.

Data Model

NoSQL is a somewhat nebulous term, and is applied to applications as varied as Berkley DB, memcached, BigTable & Accumulo, MongoDB, Neo4j, Amazon's Dynamo, etc.

Some of these applications have in common a key-value data model at a high level. Accumulo's data model consists of key-value pairs at the highest level, but because of the structure of the key it achieves some properties of conventional two-dimensional / flat-record tables, columnar and row-oriented databases, and a little bit of hierarchy in the data model via Column Families and Column Qualifiers.

Apache Accumulo, Apache Cassandra and Apache HBase share this basic BigTable data model (although Cassandra has an element called Super Columns).

Other NoSQL databases are considered to be document-oriented stores, such as MongoDB and CouchDB since they store JSON documents natively.

Neo4j is a graph-oriented database, whose data model consists of vertices and edges.

Choosing which data model is most appropriate for an application is probably the first and foremost factor one should consider when choosing a NoSQL technology. There is some flexibility in applying the data model since, for example, a key-value store can be made to store graph data and since a document-based data model is sort of a super set of the key-value model.

Key Ordering

To start, some NoSQL databases use hashing to distribute their keys to servers. This makes lookups simple for clients but can require some data to be moved when machines are added to or removed from the cluster. It can also make scanning across a sequential range of keys more difficult.

Because Accumulo maintains its own dynamic mapping of keys to servers it can handle machines joining or leaving the cluster very quickly with no data movement and no interruption to clients. In addition, the key space is partitioned dynamically and automatically so that the data is distributed evenly over the cluster.

High vs. Eventual Consistency

Some NoSQL databases are designed to run over geographically distributed data centers and allow data to be written in more than one place at once. This results in a property known as eventual consistency where a value read from the database may not be the most up to date version.

Accumulo is designed to run within a single data center and provides a highly consistent view of the data at all times. This means that users are guaranteed to always see the most up to date version of the data, which can simplify application development.

Access Control

Organizations are turning to Accumulo in order to satisfy stringent data access requirements and to comply with legal and corporate requirements and policies.

Most databases provide a level of access control over the data. Accumulo's Column Visibilities are often more fine-grained and can be used to implement a wide variety of access control scenarios.

Iterators

Accumulo's Iterators allow application developers to push some computation to the server side, which can result in a dramatic increase in performance depending on the operations performed. HBase provides a mechanism called *Co-Processors* which execute code and can be triggered at many places. Unlike Co-Processors, Iterators operate in only three places, are stackable, and are an integral part of the data processing pipeline as much of the Tablet Server's core behavior is implemented in built-in system Iterators.

Because Iterators can be used much like MapReduce map or combine functions, Iterators can help execute analytical functionality in a more streamlined and organized manner than batch-oriented MapReduce jobs. Developers looking to efficiently create and maintain result sets should consider Iterators as an option.

Use Cases Suited for Accumulo

Accumulo's design represents a very different set of objectives and technical features than data management systems such as file systems and relational databases. Knowing how these features work together is something that is new to many people. We present here a few applications in which Accumulo has been used that have leveraged Accumulo's strengths to great effect.

A new kind of flexible analytical warehouse

When attempting to build a system to analyze all the data in an organization by bringing together many disparate data sources, three problems can easily arise: a scalability problem, a problem managing sparse dynamic data, and security concerns.

Accumulo directly addresses all three of these with horizontal scalability, a rich key-value data model that supports efficiently storing sparse data and that facilitates discovery, and fine-grained access control. An analytical data warehouse built around Accumulo is still different than what one would build around a relational database. Analytical results would be aggressively pre-computed, potentially using MapReduce. Many types of data could be involved, including semi-structured JSON or XML, or features extracted from text or imagery.

Building the next Gmail

The original use case behind BigTable was for building websites that supports massive scale in two dimensions: number of simultaneous users and the amount of data managed. If your plan is to build the next Gmail, Accumulo would be a good starting point.

Massive Graph or Machine Learning problems

Features such as Iterators, MapReduce support, and a data model that supports storing dimensional sparse data make Accumulo a good candidate for creating, maintaining, storing, and processing extremely large graphs or large sets of feature vectors for machine learning applications.

MapReduce has been used in conjunction with Accumulo's scan capabilities to efficiently traverse graphs with trillions of edges, processing hundreds of millions edges per second.

Some machine learning techniques such as non-parametric algorithms are memory-based and require storing all the data rather than building a statistical model to represent the data. Accumulo is able to store large amounts of these data points and provides the basic data selection operations for supporting these algorithms efficiently.

In addition, for predictive applications that use models built from slowly changing historical data, Accumulo can be used to store and make historical data available for query and to support building models from this data via MapReduce. Accumulo's ability to manage large tables allows users to use arbitrarily complex predictive models to score all known entities and store their results for fast lookup rather than having to compute scores at query time.

Relieving relational databases

Because relational databases have performed so well over the past several decades, they have become the standard place for putting all the data and have had to support a wide

variety of data management problems. But as database expert Michael Stonebraker and others have argued,[10] trying to have one platform can result in challenges stemming from the difficulty of optimizing a single system for so many use cases.

Accumulo has been used to offload the burden of storing large amounts of raw data from relational databases, freeing them up for more specialized workloads such as performing complex runtime operations on selected subsets or summaries of the data.

Massive search applications

Google's BigTable has been used to power parts of their primary search application. Accumulo in particular has features such as automatic partitioning, batch scanning, and flexible Iterators that have been used to support complex text search applications.

Applications with a long history of versioned data

Wikipedia is an application with millions of articles edited by people around the world. Part of the challenge of these types of massive-scale collaborative applications is storing many versions of the data as users edit individual elements. Accumulo's data model allows several versions of data to be stored, and for users to retrieve versions in several ways. Accumulo's scalability makes having to store all versions of data for all time a more tractable proposition.

Versions

The first public open source version of Accumulo is 1.3.

Version 1.4 has been used in production for years on very large clusters.

As of this writing, the latest stable version of Accumulo is 1.5. We will focus this book on version 1.5, pointing out differences in other versions where appropriate. Version 1.5 includes the following new features and improvements over previous versions:

- Storing Write-Ahead Log in HDFS
- Group-commits to Write-Ahead Log
- Load JAR files from HDFS
- Support for Binary search within RFiles
- Support for running on Hadoop 2.0
- Debian packaging

10. "One Size Fits All": An Idea Whose Time Has Come and Gone. *http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.68.9136&rep=rep1&type=pdf*

- Support for running on Kerberos-enabled HDFS
- Utility for importing and exporting tables
- Thrift proxy server for allowing clients to be written in many languages
- Timeouts for BatchScanner and BatchWriter
- Ability to list ongoing compactions in the Shell
- Ability to cancel a table compaction

The complete list of Release Notes for the 1.5.0 release can be found at *http://svn.apache.org/repos/asf/accumulo/tags/1.5.0/CHANGES* or from the Accumulo project web page at *http://accumulo.apache.org*.

Additional Resources

Summary

As we've briefly described in this chapter, Accumulo is a highly scalable and flexible data store that is designed to make it easy to build powerful applications that harness the power of many servers and that can grow as the number of users or the amount data increases.

Accumulo builds on the work done by Google's engineers and delivers industrial strength data management capabilities via the open source model.

Before choosing any data store, application designers should carefully consider the features and tradeoffs involved. Accumulo provides a very different set of features than some conventional systems and each type of system should be employed as designed in order to realize its benefits.

CHAPTER 2
Quick Start

This chapter should get you up and running with Apache Accumulo. We will work through a couple of different install options and then work through a few examples. The accumulo-quickstart installation should be suitable for use with examples throughout the rest of this book.

The quickest method to get started with Accumulo is to use the *MiniAccumuloCluster*. It is a minimal version of Accumulo that starts Zookeeper and runs against the local filesystem instead of starting up HDFS. It provides a testing and experimentation environment that is close to that of a full-blown Accumulo installation, but without the initial configuration overhead. The MiniAccumuloCluster is great for writing automated tests and for experimenting with approaches. It will not scale to large data, but it is perfect for getting started.

Instamo Archetype

There is a contrib project called *Instamo Archetype* that uses the MiniAccumuloCluster. Some information on Accumulo contrib projects can be found at *http://accumulo.apache.org/contrib.html*. To use the most recently published archetype for Accumulo, follow these steps.

1. First, make sure you have Apache Maven installed. Download maven from *http://maven.apache.org/download.cgi*. Any version greater than 3 should work; we used version 3.0.4.
2. Change to the directory in which your new project will be located.
3. Use the published archetype to generate a new project.

```
mvn archetype:generate -DarchetypeCatalog=https://repository.apache.org/content/groups/public/
```

This will ask you questions and then create a project that uses the MiniAccumuloCluster in a new directory named *instamo*.

If the archetype version you are looking for has not been published, you can build it with the following steps.

1. You will also need to have *git* installed. The installation instructions vary depending on your operating system. See *http://git-scm.com/* for more information. We used version 1.8.1.1, but any recent version should do.
2. Clone the archetype repo.

```
git clone https://git-wip-us.apache.org/repos/asf/accumulo-instamo-archetype.git
```

3. Change to the *accumulo-instamo-archetype* directory and then checkout the branch or tag you want (remote branches can be listed with `git branch --list -r`). The following command would checkout the 1.5 branch:

```
git checkout -b 1.5-local origin/1.5
```

4. Next, you will need to use maven to install the archetype to your local repository so it can be used to generate a project.

```
mvn clean install
```

5. Change to the directory in which your new project will be located.
6. Use the archetype to create a project.

```
mvn archetype:generate -DarchetypeCatalog=local -DarchetypeGroupId=org.apache.accumulo -Darche
```

Generated project contents

The project generated by the Instamo Archetype includes a maven *pom.xml* file with populated dependencies. It also contains `MapReduceExample`, `ShellExample`, and `ExampleAccumuloUnitTest` classes that illustrate running different types of client code against a MiniAccumuloCluster.

Demo of the shell

We will be using an example project similar to the one generated by the Instamo Archetype with the addition of some tailored samples. First, you will need to clone the git repo.

```
git clone https://github.com/accumulobook/accumulo-quickstart
```

Now, let's begin with a look at how to use the shell. From the *accumulo-quickstart/examples* directory, run the following.

```
mvn clean compile exec:exec -Pshell
```

The shell profile that is selected with the *-Pshell* option runs the `main` method of the `ShellExample` class. This starts a Zookeeper process, an Accumulo Master process, two Accumulo TabletServer processes, and an Accumulo shell client. You will be presented with an interactive shell that looks something like this.

```
Shell - Apache Accumulo Interactive Shell
-
- version: 1.5.0
- instance name: miniInstance
- instance id: 38cfb323-faa7-4d19-9637-54eafc4c0e7d
-
- type 'help' for a list of available commands
-
root@miniInstance>
```

The default prompt on the shell shows the current Accumulo user, *root*, and the current Accumulo instance name, *miniInstance*.

If you shell is idle for too long, 60 minutes by default, you will receive an Authorization Timeout when you try to enter commands. You just need to enter the user's password again. As set in the *ShellExample.java* file, the password is *pass1234*.

The help command

Start by running the *help* command. This command will show all available commands. To see more information on a command, such as *delete*, run *help delete*. There is also more information on the shell commands listed in Appendix A: Shell Commands

The shell also supports history and tab completion.

Creating a table and inserting some data

Now that you know how to get help on shell commands, let's create a table and insert some data. Since Accumulo is a schema-less database, all you need is the table name. The schema will evolve as you insert data. So let's create a table a named *table1* by using the *createtable* command.

```
root@miniInstance> createtable table1
root@miniInstance table1>
```

Notice the prompt changed and now shows you the current table, *table1*.

The table is currently empty, so we need to insert some data. We will use the *insert* command. We introduced the Accumulo Data Model in ???, so let's insert the data from Table 1-1.

```
insert "bob jones" "contact" "address" "123 any street" -l "billing" -ts 13234
insert "bob jones" "contact" "city" "anytown" -l "billing" -ts 13234
insert "bob jones" "contact" "phone" "555-1212" -l "billing" -ts 13234
insert "bob jones" "purchases" "sneakers" "$60" -l "billing&inventory" -ts 13255
insert "fred smith" "contact" "address" "444 main st." -l  "billing" -ts 13222
insert "fred smith" "contact" "city" "othertown" -l "billing" -ts 13222
insert "fred smith" "purchases" "glasses" "$30" -l  "billing&inventory" -ts 13201
insert "fred smith" "purchases" "hat" "$20" -l "billing&inventory" -ts 13267
```

Generally it is best to let Accumulo manage the timestamps on its keys. Setting timestamps explicitly should be left to advanced use cases, since it can result in unexpected behavior if you are not very familiar with how the key's timestamp is used in Accumulo.

Scanning for data

Once you get data into Accumulo, you need to view it. We will use the *scan* command. But if you just type *scan*, no results will be returned. The data we entered included column visibilities with the *-l* switch, but the root user does not have authorizations to view those. A user in Accumulo can write data with any authorizations (unless it has been prohibited by configuring the `VisibilityConstraint` — see ???), but viewing records requires authorization. A user's current authorizations can be viewed with the *getauths* command. Assign the necessary authorizations with the following:

```
setauths -u root -s billing,inventory
```

You can now see all the records with a *scan*:

```
root@miniInstance table1> scan
bob jones contact:address [billing]    123 any street
bob jones contact:city [billing]    anytown
bob jones contact:phone [billing]    555-1212
bob jones purchases:sneakers [billing&inventory]    $60
fred smith contact:address [billing]    444 main st.
fred smith contact:city [billing]    othertown
fred smith purchases:glasses [billing&inventory]    $30
fred smith purchases:hat [billing&inventory]    $20
```

You may have noticed that the timestamps inserted in the records with the *-ts* switch are not displayed. Use the *-st* or *--show-timestamps* switch to see those.

```
root@miniInstance table1> scan -st
bob jones contact:address [billing] 13234    123 any street
bob jones contact:city [billing] 13234    anytown
bob jones contact:phone [billing] 13234    555-1212
```

```
bob jones purchases:sneakers [billing&inventory] 13255    $60
fred smith contact:address [billing] 13222    444 main st.
fred smith contact:city [billing] 13222    othertown
fred smith purchases:glasses [billing&inventory] 13201    $30
fred smith purchases:hat [billing&inventory] 13267    $20
```

If you want to view just the records for one row ID, use the *-r* switch. This will limit the results to one row ID.

```
root@miniInstance table1> scan -r "bob jones"
bob jones contact:address [billing]    123 any street
bob jones contact:city [billing]    anytown
bob jones contact:phone [billing]    555-1212
bob jones purchases:sneakers [billing&inventory]    $60
```

Using authorizations

By default, the shell *scan* command will use all of the current user's granted authorizations. Use the *-s* switch to limit the scan authorizations.

```
root@miniInstance table1> scan -s billing
bob jones contact:address [billing]    123 any street
bob jones contact:city [billing]    anytown
bob jones contact:phone [billing]    555-1212
fred smith contact:address [billing]    444 main st.
fred smith contact:city [billing]    othertown
```

We did not insert any records with only the *inventory* visibility, so you would not see any results using just that authorization.

Using a simple iterator

We have briefly discussed Accumulo's iterators. One built-in iterator is the `GrepIterator`,[1] which searches the Key and the Value for an exact string match. The shell's *grep* command sets up this iterator and uses it during the scan.

```
root@miniInstance table1> grep town
bob jones contact:city [billing]    anytown
fred smith contact:city [billing]    othertown
```

Demo of Java code

Now let's do the same things, but this time using Java code instead of shell commands. The Java code to perform these operations exists in the `JavaExample` class of accumulo-quickstart. It connects to any specified running Accumulo instance, and does not start up its own MiniAccumuloCluster.

1. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/iterators/user/GrepIterator.html*

For this exercise, leave the Accumulo shell example running, and we will connect to its MiniAccumuloCluster using Java code. Another typical use case would be to start up a MiniAccumuloCluster in a unit test. But for this exercise, leave the Accumulo shell running or restart it with the command

```
mvn compile exec:exec -Pshell
```

You will need to know the Zookeeper and instance information the `ShellExample` used to start up. It is output just before the shell starts up, something like

```
        ---- Initializing Accumulo Shell

    Starting the MiniAccumuloCluster in /var/folders/2y/n9lzqm2x10lfxqm9n40xvfvw0000gn/T/139506634
    Zookeeper is localhost:11272
    Instance is miniInstance
```

We will use this information to connect to the running Accumulo instance.

Creating a table and inserting some data

Staying in the *accumulo-quickstart/examples* directory, take a look at the *JavaExample.java* file in *src/main/java/com/accumulobook/quickstart*. Much of the code is for parsing arguments and selecting which command to run. The relevant code that uses the Accumulo client API will be highlighted here.

All commands that will be run here need to get a reference to the Accumulo `Connector`.[2] Using the instance name and the location of the running Zookeeper, the code to get this `Connector` is in the `getConnection` method of the `JavaExample` class. All the Commmand classes extend the `AbstractCommand` class, which provides a `setConnection` method that the `JavaExample` uses to set each Command's `Connector`. Here is what the code looks like for the `getConnection` method. The *instance* and *zookeepers* fields are set based on command line parameters when the code is executed.

```
public Connector getConnection() throws AccumuloException, AccumuloSecurityException {
    Instance i = new ZooKeeperInstance(instance, zookeepers);
    Connector conn = i.getConnector(user, new PasswordToken(password));
    return conn;
}
```

Using this `Connector` inside some Java code, let's create a table. For this part of the quickstart, the table will be called *table2*. From the terminal where you started the shell, you can run the *tables* command, and you should not see a *table2* table. Let's create that table now. In a new terminal window at the *accumulo-quickstart/examples* directory, run the following.

```
mvn clean compile exec:exec -Dtable.name=table2 -Dinstance.name=miniInstance -Dzookeeper.location=
```

2. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/Connector.html*

Be sure to replace the *instance.name* and *zookeeper.location* values with what was displayed when the `ShellExample` started. If this hangs for more than 15 seconds or so after outputting "Running create command", your Zookeeper location may be wrong. Once complete, you should be able to run *tables* from the shell in the other window and see *table2*.

The `Connector` has access to `TableOperations`, which is used to perform administrative operations on tables.[3] We use the `createTable` method to create the table. Here is the code example from the `CreateCommand` `run` method.

```
public void run() throws AccumuloException, AccumuloSecurityException, TableExistsException {
    System.out.println("Creating table " + table);
    if (connection.tableOperations().exists(table)) {
        throw new RuntimeException("Table " + table + " already exists");
    } else {
        connection.tableOperations().create(table);
        System.out.println("Table created");
    }
}
```

On line 3, we check to ensure the table does not already exist. This is not technically necessary, as a `TableExistsException` will be thrown, but this pattern of checking for existence before creating a table enables us to do something different if we wanted.

Now let's insert some data. As in the shell example, we will do this one row at time. Later, you will see how to batch up inserts, or mutations, and execute them together. Here is the first row. When cutting and pasting, remember to change the *instance.name* and *zookeeper.location*.

```
mvn clean compile exec:exec -Drow.id="bob jones" \
-Dcolumn.family=contact \
-Dcolumn.qualifier=address \
-Dtimestamp=13234 \
-Dauths=billing \
-Dvalue="123 any street" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert
```

The *-D* switches are simply setting system properties that get passed to the `JavaExample` as arguments.

3. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/admin/TableOperations.html*

From the shell, you can now change to the *table2* table by running *table table2*, and then you can run *scan*. If your user has the *billing* authorization, you should see the record. Should you not see any records, use *getauths* and *setauths* for the root user to ensure the *billing* authorization is present. We will look at how to handle authorizations with Java in the next section.

Again using the `Connector` object, here is the code that inserts data.

```
public void run() throws TableNotFoundException, MutationsRejectedException {
    System.out.println("Writing mutation for " + rowId);
    BatchWriter bw = connection.createBatchWriter(table, new BatchWriterConfig());
    Mutation m = new Mutation(new Text(rowId));
    m.put(new Text(cf), new Text(cq), new ColumnVisibility(auths), timestamp, new Value(val.getByt
    bw.addMutation(m);
    bw.close();
}
```

A `BatchWriter`[4] is created from the `Connector`. A `Mutation`[5] is constructed with the row ID for a new key-value pair. There are multiple `put` methods for `Mutation` with different signatures to define the rest of the key-value pair. The `Mutation` is added to the `BatchWriter`. You could also add multiple mutation objects to a batch writer, which is a more typical usage that saves the overhead of creating a new `BatchWriter` as well as batching together data to amortize network communication overhead. When the batch writer is closed or flushed, the added mutations are sent to Accumulo.

Here are commands you can copy and paste to insert the rest of the sample data. Each of these commands needs the instance name and Zookeeper location updated. There is also a batch script explained after these commands that may be easier to use.

```
mvn clean compile exec:exec -Drow.id="bob jones" \
-Dcolumn.family=contact \
-Dcolumn.qualifier=city \
-Dtimestamp=13234 \
-Dauths=billing \
-Dvalue="anytown" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert

mvn clean compile exec:exec -Drow.id="bob jones" \
-Dcolumn.family=contact \
-Dcolumn.qualifier=phone \
-Dtimestamp=13234 \
-Dauths=billing \
-Dvalue="555-1212" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert
```

4. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/BatchWriter.html*
5. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/data/Mutation.html*

```
mvn clean compile exec:exec -Drow.id="bob jones" \
-Dcolumn.family=purchases \
-Dcolumn.qualifier=sneakers \
-Dtimestamp=13255 \
-Dauths=billing\&inventory \
-Dvalue="\$60" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert

mvn clean compile exec:exec -Drow.id="fred smith" \
-Dcolumn.family=contact \
-Dcolumn.qualifier=address \
-Dtimestamp=13222 \
-Dauths=billing \
-Dvalue="444 main st." \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert

mvn clean compile exec:exec -Drow.id="fred smith" \
-Dcolumn.family=contact \
-Dcolumn.qualifier=city \
-Dtimestamp=13222 \
-Dauths=billing \
-Dvalue="othertown" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert

mvn clean compile exec:exec -Drow.id="fred smith" \
-Dcolumn.family=purchases \
-Dcolumn.qualifier=glasses \
-Dtimestamp=13201 \
-Dauths=billing\&inventory \
-Dvalue="\$30" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert

mvn clean compile exec:exec -Drow.id="fred smith" \
-Dcolumn.family=purchases \
-Dcolumn.qualifier=hat \
-Dtimestamp=13267 \
-Dauths=billing\&inventory \
-Dvalue="\$20" \
-Dtable.name=table2 \
-Dinstance.name=miniInstance -Dzookeeper.location=localhost:11272 -Pjava:insert
```

The batch script is located in accumulo-quickstart/examples/bin/insert-all.sh. It runs all these commands for you, but allows you to pass in the instance name, Zookeeper location and table name values once at the beginning. Run it as follows from the *accumulo-quickstart/examples* directory, replacing the parameters with the correct values for your running shell.

```
./bin/insert-all.sh miniInstance "localhost:11272" table2
```

You can run these mutations multiple times, but still end up with only 8 key-value pairs. The reason is that Accumulo defaults to keep only the most recent version of each key by configuring the `VersioningIterator` on every new table. Even if the same data is inserted multiple times, you will only see one version of each key-value pair.

If you left the shell running from the shell demo and used it for the this demo, table1 and table2 should now have exactly the same data.

Scanning for data

Now let's use some Java code to scan all the rows. Run the following, and all the rows will be printed.

```
mvn clean compile exec:exec -Dtable.name=table2 -Dinstance.name=miniInstance -Dzookeeper.location=
```

You will see output like the following:

```
Running scan command
Scanning table2
Scanning with all user auths
Scanning for all rows
Results ->
  bob jones contact:address [billing] 13234 false 123 any street
  bob jones contact:city [billing] 13234 false anytown
  bob jones contact:phone [billing] 13234 false 555-1212
  bob jones purchases:sneakers [billing&inventory] 13255 false $60
  fred smith contact:address [billing] 13222 false 444 main st.
  fred smith contact:city [billing] 13222 false othertown
  fred smith purchases:glasses [billing&inventory] 13201 false $30
  fred smith purchases:hat [billing&inventory] 13267 false $20
```

Here is the code that is run in the `ScanCommand`.

```
System.out.println("Scanning " + table);
Authorizations authorizations = null;
if ((null != auths) && (!auths.equals("SCAN_ALL"))) {
    System.out.println("Using scan auths " + auths);
    authorizations = new Authorizations(auths.split(","));
} else {
    System.out.println("Scanning with all user auths");
    authorizations = connection.securityOperations().getUserAuthorizations(user);
}
Scanner scanner = connection.createScanner(table, authorizations);
if ((null != row) && (!row.equals("SCAN_ALL"))) {
    System.out.println("Scanning for row " + row);
    scanner.setRange(new Range(row));
} else {
    System.out.println("Scanning for all rows");
}
System.out.println("Results ->");
```

```
for (Entry<Key,Value> entry : scanner) {
    System.out.println("  " + entry.getKey() + " " + entry.getValue());
}
```

On line 10, we create a `Scanner`[6] object from the `Connector`. This `Scanner` is what scans Accumulo and returns an `Iterable` of results. We iterate over those results on line 18 and print out the Accumulo `Key`[7] and `Value`[8]. The `toString` method of the `Key` outputs the key's timestamp by default. There is also a `toStringNoTime` method on `Key` if we wanted to model more closely the shell example. That is left as an exercise for the reader.

This scan does not limit the results; everything is returned just like running the *scan* command in the Accumulo shell. Let's see how we could limit the results to just the *bob jones* row. Run the following:

```
mvn clean compile exec:exec -Dtable.name=table2 -Drow="bob jones" -Dinstance.name=miniInstance -Dz
```

Here we provide *row="bob jones"*, which is passed along to the `run` method on the `ScanCommand`. Line 13 shows how to provide that information to the `Scanner`, using the `Range`[9] class. Ranges can also be defined in other ways to better limit the results. Review the API documentation for more information. We will also cover additional ways to construct key ranges throughout this book, in particular in ???.

This example uses the `createScanner` method on the `Connector`. This `Scanner` object runs in one thread and hits one range at a time. You can also use the `createBatchScanner` method, which returns a `BatchScanner` and will scan multiple ranges in parallel. When your data is spread out on many tablet servers, this `BatchScanner` can return results much faster. However, the `BatchScanner` does not guarantee any ordering of the results returned. Your code will have to handle that correctly.

Using authorizations

As we mentioned before, the shell *scan* command uses all of the user's authorizations by default. This is a convenience provided by the shell for interactively accessing the data. To make the Java scan example work the same way, we had to do a couple of things. First we set a default value of `SCAN_ALL` for the *auths* property used by `ScanCommand` in the *pom.xml* file. This allows us to not pass in *-Dauths=<something>*. If `ScanCommand` finds that the authorizations are set to the default `SCAN_ALL` value, it will look up the

6. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/Scanner.html*
7. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/data/Key.html*
8. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/data/Value.html*
9. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/data/Range.html*

user's entire set of authorizations and use those for the scan, as is done in the Accumulo shell. On line 8 of the `ScanCommand run` method, you see the `securityOperations` method of the `Connector` used to obtain a `SecurityOperations`[10] object. From this object, we use the `getUserAuthorizations` method to obtain an `Authorizations`[11] object that contains all the user's authorizations. Alternatively, if we explicitly pass in *auths* for `ScanCommand`, it will provide them to the string array constructor of `Authorizations` on line 5. In either case, the `Authorizations` object is used when creating a `Scanner` on line 10.

As we did in the shell demo, let's scan for just records with the *billing* authorization. Run the following:

```
mvn clean compile exec:exec -Dtable.name=table2 -Dauths=billing -Dinstance.name=miniInstance -Dzoo
```

You should get results like these:

```
Running scan command
Scanning table2
Using scan auths billing
Scanning for all rows
Results ->
  bob jones contact:address [billing] 13234 false 123 any street
  bob jones contact:city [billing] 13234 false anytown
  bob jones contact:phone [billing] 13234 false 555-1212
  fred smith contact:address [billing] 13222 false 444 main st.
  fred smith contact:city [billing] 13222 false othertown
```

Here, the results were limited to records that could be viewed using only the *billing* authorization.

Using a simple iterator

The last thing we want to show in Java code is how to set up an iterator. The shell example used the built-in `GrepIterator`. We will do the same here. To run the example, use the following, again replacing the instance name and Zookeeper location as appropriate.

```
mvn clean compile exec:exec -Dtable.name=table2 -Dterm=town -Dinstance.name=miniInstance -Dzookeep
```

Your results should look like this:

```
Running grep command
Grepping table2
Results ->
  bob jones contact:city [billing] 13234 false anytown
  fred smith contact:city [billing] 13222 false othertown
```

10. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/admin/SecurityOperations.html*

11. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/security/Authorizations.html*

The Java code is similar to the scan example, but instead of setting up a range, we add the `GrepIterator` to the `Scanner`. Here is the code:

```
System.out.println("Grepping " + table);
Authorizations authorizations = connection.securityOperations().getUserAuthorizations(user);
Scanner scanner = connection.createScanner(table, authorizations);
Map<String, String> grepProps = new HashMap<String, String>();
grepProps.put("term", term);
IteratorSetting is = new IteratorSetting(25, "sample-grep", GrepIterator.class.getName(), grepProp
scanner.addScanIterator(is);
System.out.println("Results ->");
for (Entry<Key,Value> entry : scanner) {
    System.out.println("  " + entry.getKey() + " " + entry.getValue());
}
```

The authorizations are set up on line 2 just as in the `SCAN_ALL` case for the `ScanCommand`. The scanner is created the same way also, shown on line 3. But on line 6, we construct an `IteratorSetting`[12] using a `GrepIterator` and map of properties that set the *term* to be the value we passed in, *town* in this case. The `25` in the constructor is the priority of this iterator. The string `"sample-grep"` is a unique name for the iterator that will be used as a key to group together the iterator's configuration information in Zookeeper (its priority, class, and options). Line 7 shows how the `IteratorSetting` is added to the `Scanner`. Looping over results is just like was shown in the `ScanCommand`.

Iterator names must be unique within a table and iterator scope. So must iterator priorities. More on iterator configuration can be found in ??? of Chapter 3, *Writing Applications*.

A more complete installation

While the Instamo Archetype using the MiniAccumuloCluster will get you started quickly, there are a lot of components that are not started such as the Monitor. The Hadoop installation is minimal, so you do not get a chance to learn any of those tools. The MiniAccumuloCluster is really more suitable for starting up Accumulo and using it for testing.

So if MiniAccumuloCluster is not what most developers use, what is the best way to get a development environment set up? Most developers use a full installation, either on one node or in small cluster with VMs or on a service like Amazon EC2. Typically that requires going through the process of installing Hadoop, Zookeeper and Accumulo

12. *http://accumulo.apache.org/1.5/apidocs/index.html?org/apache/accumulo/core/client/IteratorSetting.html*

individually. This can be a daunting task if you have no experience with any of these components.

Instead of providing a VM image for you use, we decided to facilitate setting up a one node install. The advantages of not using a pre-configured VM are better performance and more flexibility. A full installation, even on one node, will allow you to shutdown Accumulo and save the data. The rest of this book will assume you are using a full installation.

To get a more complete installation, we are going to use the quickinstall, which is the other top level directory in the accumulo-quickstart project. This project will download all the necessary components, install and configure them, and then start everything up for you. To use the quickinstall, you need to clone the repository if you haven't already.

```
git clone https://github.com/accumulobook/accumulo-quickstart
```

From the *accumulo-quickstart/quickinstall* directory, run the following.

```
./bin/install
```

This process takes about 20 minutes the first time, mostly to download the binaries for Hadoop, Zookeeper and Accumulo. Once these artifacts are downloaded and stored in your ivy2 cache, reinstalling will not take long at all.

If the install fails, the error messages should be helpful to resolve issues. Follow the instructions and rerun the install script after fixing whatever was wrong. If you have previously attempted to install Hadoop, make sure there are no `HADOOP_*` environment variables already set up in your environment. The quickinstall and other example commands below will not work otherwise.

Here is what the script does. First, all the dependencies and code are going to be downloaded and stored in your ivy2 cache, typically *~/.ivy2/cache* unless you set up something different. Next, these artifacts will be unzipped into the *accumulo-quickstart/quickinstall/install-home* directory. Configuration files are copied into the correct directories and modified for your environment. Then Hadoop is started, Zookeeper is started and Accumulo is started.

Let's look at the install-home directory when the installation is complete.

accumulo-1.5.0
: contains the Accumulo installation

hadoop-1.2.1
: contains the Hadoop installation

zookeeper-3.3.6
: contains the Zookeeper installation

hdfs
: is the directory where Hadoop stores data

mapred
: is the directory Hadoop uses for running MapReduce

zk-data
: is the directory where Zookeeper stores data

Also in the install-home is a bin directory with some helper scripts.

- cloud-helpers

  cloud-env
  : sets up the environment variables

  start-all
  : starts hadoop, zookeeper and accumulo

  stop-all
  : stops accumulo, hadoop and zookeeper

- Hadoop helpers

  hd-api
  : opens the hadoop javadocs in your browser

  hd-start-all
  : starts hadoop, calls HADOOP_PREFIX/bin/start-all.sh

  hd-stop-all
  : stops hadoop

- Zookeeper helpers

  zk-start
  : starts zookeeper

  zk-stop
  : stops zookeeper

- Accumulo helpers

  acc-api
  : opens the accumulo javadocs in your browser

  acc-start-all
  : starts accumulo, call ACCUMULO_HOME/bin/start-all.sh

  acc-stop-all
  : stops accumulo

After the install, everything should be running. It is a self contained environment and everything should be under the install-home directory. The benefit of this is that you can stop everything, remove that directory, and reinstall if needed.

You can verify everything is running with the command

```
jps -lm
```

This command should show you processes that include the following:

- Hadoop processes

```
org.apache.hadoop.mapred.TaskTracker
org.apache.hadoop.hdfs.server.namenode.SecondaryNameNode
org.apache.hadoop.mapred.JobTracker
org.apache.hadoop.hdfs.server.namenode.NameNode
org.apache.hadoop.hdfs.server.datanode.DataNode
```

- Zookeeper process

```
org.apache.zookeeper.server.quorum.QuorumPeerMain ./bin/../conf/zoo.cfg
```

- Accumulo processes

```
org.apache.accumulo.start.Main gc --address localhost
org.apache.accumulo.start.Main tserver --address localhost
org.apache.accumulo.start.Main tracer --address localhost
org.apache.accumulo.start.Main monitor --address localhost
org.apache.accumulo.start.Main master --address localhost
```

To start up an Accumulo shell, first run

```
source bin/cloud-env
```

This script will set up all the paths.

If things were not running, you could run *start-all* to start Hadoop, Zookeeper and then Accumulo after sourcing *cloud-env*. When you want to stop everything, run *stop-all*. Accumulo and Hadoop both include *start-all.sh* scripts, which can be confusing. The scripts provided with quickinstall start and stop all the processes you will need.

Now you need to run the main *accumulo* command, which is located in *install-home/accumulo-1.5.0/bin*. This is the main entry point for working with Accumulo from the command line. It will be on your path if you sourced the *cloud-env* script. Let's run the shell.

```
accumulo shell -u root -p secret
```

This will start the shell you saw in the shell example. Try out some of the commands, like *tables*. You should only see the `!METADATA` and the `trace` table, which are internal tables we will discuss more in Appendix C and ???, respectively.

Let's get set up to run the insert and scan commands with Java code again. We will handle the table creation and granting authorizations to the root user in the shell. Create a table named *table3*.

```
createtable table3
```

Now ensure the root user has *billing* and *inventory* authorizations.

```
setauths -u root -s "billing,inventory"
```

Once that is complete, we will use a script similar to the *./bin/insert-all.sh* script from the Java example. However, this script will use not use maven to execute the `JavaExample` class, instead it will run it directly. You will need to use maven to build a jar by executing the following from the *accumulo-quickstart/examples* directory.

```
mvn package
```

Source the *bin/cloud-env* script again and run the `JavaExample` using the *accumulo* command.

```
source ../quickinstall/install-home/bin/cloud-env
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar com.accumulobook.quickstart.JavaE
```

You should be presented with the default usage from the `JavaExample` class, explaining all the options.

Using the *accumulo* command along with the *-add* option is easiest and cleanest way to run Java programs with a classpath already set up for your Accumulo installation.

Now let's insert the first key-value pair.

```
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar \
  com.accumulobook.quickstart.JavaExample \
  -i accumulo -z localhost:2181 -u root -p secret \
  insert -r "bob jones" -t table3 -cq contact -cf address \
  -val "123 any street" -a billing -ts 13234
```

The other insert commands are in the file *./bin/insert-all2.sh*. You can either copy and paste the commands from this file into the terminal, including the variables that are set up, or you can just run the *insert-all2.sh* script.

```
./bin/insert-all2.sh
```

Once the data is inserted, let's use the shell to scan to make sure we can see all the data. We will use the *-e* switch to pass in a command for the shell to execute. This runs the command and exits out of the shell. For that reason, our scan command needs to use the *-t* switch to specify which table to scan. Using command line execution with the

accumulo shell is a useful technique, because you can then use all the regular Unix tools like *grep*, *sed* and *cut*.

```
accumulo shell -u root -p secret -e "scan -t table3 -st"
```

Hopefully you will see 8 key-value pairs. Pipe the previous command through *wc -l* if you don't want to count for yourself.

```
accumulo shell -u root -p secret -e "scan -t table3 -st" | wc -l
```

If you don't see all 8, revisit the previous commands and make sure the records have been inserted correctly and that your authorizations are configured properly. Now run the following to execute the same scan with the `JavaExample` command. You must be in the *accumulo-quickstart/examples directory* or modify the location of the jar to an absolute path.

```
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar \
  com.accumulobook.quickstart.JavaExample \
  -i accumulo -z localhost:2181 -u root -p secret scan -t table3
```

The result should be same as from scanning in the earlier example. Try limiting the results by row.

```
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar \
  com.accumulobook.quickstart.JavaExample \
  -i accumulo -z localhost:2181 -u root -p secret scan -t table3 \
  -r "bob jones"
```

Now try limiting the results by authorizations.

```
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar \
  com.accumulobook.quickstart.JavaExample \
  -i accumulo -z localhost:2181 -u root -p secret scan -t table3 \
  -a billing
```

For a last exercise, try running the grep example.

```
accumulo -add $PWD/target/quickstart-examples-0.0.1-SNAPSHOT.jar \
  com.accumulobook.quickstart.JavaExample \
  -i accumulo -z localhost:2181 -u root -p secret grep -t table3 \
  --term town
```

Other important resources

We have seen how to interact with Accumulo both in the shell and in code. Another important tool for interacting with Accumulo is the Monitor page. Assuming your quickinstall is still running, visit *http://localhost:50095*. We will not discuss the Monitor page in detail here, but feel free to click around and look at the different information it provides.

Another important tool is the log files. For the quickinstall, these will be located in *accumulo-quickstart/quickinstall/install-home/accumulo-1.5.0/logs*. The Accumulo pro-

cesses we listed with the *jps* command each has its own log. The logs are prefixed with gc, master, monitor, master or tserver. By default, Accumulo configures a log and debug.log for each process, with the latter logging everything at a log level of debug or higher.

One last example with a unit test

We talked about the MiniAccumuloCluster being really good for unit testing. There is an example named *Example.java* in the *accumulo-quickstart/examples/src/main/java/com/accumulobook/quickstart* directory. It is a bit of a contrived example, but it has no knowledge of the MiniAccumuloCluster. Instead it just knows about the instance name, Zookeeper location and root password, and uses those to connect to an Accumulo instance. The *ExampleTest.java* test in the *accumulo-quickstart/example/src/test/java/com/accumulobook/quickstart* directory starts up a MiniAccumuloCluster and then uses the instance name, Zookeeper location and root password from that to construct a new `Example`. Methods on this instance of `Example` are then tested against the MiniAccumuloCluster, as if it were a full Accumulo instance. You may have noticed a unit test was executed when you ran *mvn package* earlier. This ExampleTest.java was the test that ran. Feel free to run *mvn test* and study the output.

More info

Main apache page
: *http://accumulo.apache.org*

Official Documentation
: *http://accumulo.apache.org/1.5/accumulo_user_manual.html*

Javadocs
: *http://accumulo.apache.org/1.4/apidocs*

Downloads
: *http://accumulo.apache.org/downloads/*

Source code
: *http://git-wip-us.apache.org/repos/asf/accumulo.git*

Github mirror
: *https://github.com/apache/accumulo*

Mailing list info
: *http://accumulo.apache.org/mailing_list.html*

Issues/Jira
: *https://issues.apache.org/jira/browse/accumulo*

Build server
: *https://builds.apache.org/view/A-D/view/Accumulo/*

Latest Github projects
: *https://github.com/search?q=accumulo&type=Repositories&s=updated*

CHAPTER 3
Writing Applications

Accumulo applications are the primary method of getting data into and out of Accumulo tables. Applications can be written in Java using the provided client library or in other languages in conjunction with the Thrift proxy.

Applications tend to do one of two things - one, get data into Accumulo, applying any necessary transformations to existing data to map to the Accumulo data model, or two, perform scans against Accumulo tables to satisfy user requests. The first class of applications are sometimes called *ingest clients* and the latter, *query clients*.

Many Accumulo query clients are deployed as part of a web application, allowing users to perform interactive requests for information stored in Accumulo tables, although this is certainly not a requirement. Some clients provide access to information in Accumulo to other services.

Considerations

When designing an application on the Accumulo API, a few questions should be considered which will assist in determining how the ingest and query clients are written, how the data should be organized within one or more Accumulo tables, and ideally, what kind of performance can be expected?

These considerations are listed here not in order of importance; they are all equally important.

Application Logic

The first thing to consider when creating an application on Accumulo is simply what activities will the application carry out on behalf of the user? These can include but are not limited to:

- Does the application capture information provided by the user?
- Are there semantic rules governing relationships in the information managed?
- Does data need to be updated?

In particular, attention should be paid to the *access patterns* that the application requires. The term *access pattern* refers to how the user wants to access the data - for example, users may need to retrieve information about books based on the title, and at other times, by the author, and at other times, both. Knowing what information users know and how they will use that to find out information they don't know will help guide the design of tables and rule out designs that will not perform well.

Data

The second thing to consider is the data that is managed by Accumulo, including questions such as

- Does the data already exist or is it being created by the user via the application, or both?
- If some data already exists, in what format is it currently stored?
- Is combining two or more existing data sets required? If so, is the way they should be combined known beforehand?
- At what rate will data arrive?
- What sensitivities exist within the data?
- What groups of users will need access to which parts of the data?

Performance

The third consideration in application design is performance.

One fun aspect of working with large amounts of data is that computer science training and theory comes in very handy. Desktop computers are so powerful that sometimes application developers can get away with inefficient designs without affecting performance to the extent that users notice. When working with many terabytes of data, performance and efficient design once again becomes paramount.

Applications that interact with users in real time may need to respond to requests very quickly, often in under a second. Some usability experts have recommended time limits[1] that affect how users perceive an application:

1. *http://www.nngroup.com/articles/response-times-3-important-limits/*

> **0.1 second** is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
>
> **1.0 second** is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
>
> **10 seconds** is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.
>
> — Miller 1968
> *Card et al. 1991*

Some useful questions with regard to performance include:

- What performance is acceptable or expected of the application?
- What operations does the application need to perform and how much work is required to do it?
- Can some of the work to answer queries be performed at ingest time rather than query time (pre-computation)? How might this affect ingest performance?

The answers to these questions can help guide the application designer to determine how data should be processed and organized so that the required access patterns can be supported in a way that meets the performance requirements and the semantic rules of the application.

Understanding Accumulo Performance

Application designers must understand the capabilities of the hardware and subsystems on which their application must run in order to reason about performance and develop designs to meet performance requirements. As far as understanding hardware, Google BigTable author Jeffrey Dean has shared a list[2] of what he calls *Numbers Everyone Should Know* (Table 3-1). Of those, a few are of special interest to Accumulo application developers.

Table 3-1. Some Numbers Everyone Should Know

Main memory reference	0.0001 ms
Send 2K bytes over 1 Gbps network	0.020 ms
Read 1 MB sequentially from memory	0.25 ms

2. *http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/people/jeff/stanford-295-talk.pdf*

Round trip within same datacenter	0.5 ms
Disk seek	10 ms
Read 1 MB sequentially from network	10 ms
Read 1 MB sequentially from disk	30 ms

To help application designers understand how using Accumulo affects application performance it is useful to apply the information of how hardware performs to an understanding of how Accumulo operations use hardware.

Accumulo scans involve doing some lookups in the Accumulo client's cache, then communicating with a TabletServer to read out the data requested.

One important thing to note when modeling queries is that in our calculations above, the number and size of key-value pairs has a linear effect on performance of the time it takes to read data off of disk, sort it in memory, and transfer it over the network. But the total number of key-value pairs in the table has a much smaller effect. The bigger the table, the more servers it will take to store the data, but when querying, a single scan will usually involve only one Tablet Server and a small number files associated with the tablet containing the requested data, no matter how big the table gets.

The first step, finding the right Tablet Server for a scan, requires doing a binary search among tablet extents stored in memory. The time to do this search grows logarithmically with the number of tablets. This means that if finding the right tablet among 100 tablets takes 5 microseconds on average, finding the right tablet when there are 1000 tablets should only take an average of 10 microseconds, not 50.

Sometimes the blocks referenced as part of a scan will be cached in memory already, as Accumulo employs caching of blocks read from HDFS. However, Accumulo is designed to perform fast scans even when data is not cached and does so by minimizing disk seeks. This design is crucial to scaling to handle large amounts of data in a cost effective way, since disk is many times cheaper than memory. More information on how Accumulo works is found in Chapter 4 (Internals).

Of course, an application can itself reference memory, disk, other services over the network etc. Performing back-of-the-envelope calculations about how an application should perform is helpful in determining viable design alternatives.

Development Environment

To begin writing Java applications for Accumulo you'll need to obtain the Accumulo Java library. Information on developing applications in other languages is described in "Proxy Service" on page 91 (Proxy Service).

Obtaining the Client Library

The latest Accumulo Java client library can be obtained from the official site *http://accumulo.apache.org/downloads*. If using Maven to manage project dependencies, no special repositories need to be added to your settings.xml file.

Using Maven

To see if Maven is installed, type `mvn -version`. If you don't have Maven 3.0.4 or greater, download and install it from *http://maven.apache.org*. To add Accumulo as a dependency for your Maven project, add the following to the dependencies section of your pom.xml file:

```
<dependency>
  <groupId>org.apache.accumulo</groupId>
  <artifactId>accumulo-core</artifactId>
  <version>1.5.0</version>
</dependency>
```

Run `mvn clean package` to create a jar containing your code, or use the appropriate Maven goals for your project.

Using Maven with an IDE

Eclipse
: If you are using Eclipse, you may need to install a plugin for Maven support. See *http://maven.apache.org/eclipse-plugin.html* for information about plugins.

NetBeans
: Comes with Maven support (*http://wiki.netbeans.org/Maven*).

IntelliJ IDEA
: Comes with Maven support (*http://www.jetbrains.com/idea/features/build_tools.html*).

Configuring Classpath

Bundling up the Accumulo dependencies with your client jar is discouraged, because it can make debugging difficult later. A better way to handle dependencies is to configure your classpath properly. The accumulo-core, accumulo-trace, and zookeeper jars are ones that are likely to be needed on your classpath. Commons and log4j jars may also be necessary. If you are running a MapReduce job, you will need to pass some of the dependencies to the MapReduce child processes using the `-libjars` parameter. Accumulo comes with scripts that configure your classpath and libjars (if applicable) for standalone or MapReduce jobs. The usage for these scripts follows.

Standalone

```
bin/accumulo -add jarFile className args
```

MapReduce

```
bin/tool.sh jarFile className args
```

Basic Applications

Basic Accumulo applications often begin with a data set, and some things we want to do with that data. For the purposes of introducing readers to the Accumulo API, we're going to use the data from Wikipedia. We'll write an application to load this data and query it to allow users to explore the information contained within it in various ways.

The Wikipedia Data

Wikipedia is a collection of over 30 million articles in 287 languages, including 4.3 million in English, written by volunteers. Articles contain free form text and associated *meta data* including title, timestamps, contributor information, and references to other articles and sources.

A snapshot of the English Wikipedia articles can be downloaded from *http://en.wikipedia.org/wiki/Wikipedia:Database_download#English-language_Wikipedia*. Alternatively, a specific set of pages can be downloaded using the Special Export option: *http://en.wikipedia.org/wiki/Special:Export*.

The data is stored in the XML format. The body of the articles is in the MediaWiki markup format, developed specifically for Wikipedia.

An example of an article is as follows,

```
<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.8/" xmlns:xsi="http://www.w3.org/2001/XMLS
<siteinfo>
<sitename>Wikipedia</sitename>
<base>http://en.wikipedia.org/wiki/Main_Page</base>
<generator>MediaWiki 1.22wmf15</generator>
<case>first-letter</case>
<namespaces>
<namespace key="-2" case="first-letter">Media</namespace>
<namespace key="-1" case="first-letter">Special</namespace>
<namespace key="0" case="first-letter"/>
<namespace key="1" case="first-letter">Talk</namespace>
<namespace key="2" case="first-letter">User</namespace>
<namespace key="3" case="first-letter">User talk</namespace>
<namespace key="4" case="first-letter">Wikipedia</namespace>
<namespace key="5" case="first-letter">Wikipedia talk</namespace>
<namespace key="6" case="first-letter">File</namespace>
<namespace key="7" case="first-letter">File talk</namespace>
<namespace key="8" case="first-letter">MediaWiki</namespace>
<namespace key="9" case="first-letter">MediaWiki talk</namespace>
<namespace key="10" case="first-letter">Template</namespace>
```

```
<namespace key="11" case="first-letter">Template talk</namespace>
<namespace key="12" case="first-letter">Help</namespace>
<namespace key="13" case="first-letter">Help talk</namespace>
<namespace key="14" case="first-letter">Category</namespace>
<namespace key="15" case="first-letter">Category talk</namespace>
<namespace key="100" case="first-letter">Portal</namespace>
<namespace key="101" case="first-letter">Portal talk</namespace>
<namespace key="108" case="first-letter">Book</namespace>
<namespace key="109" case="first-letter">Book talk</namespace>
<namespace key="446" case="first-letter">Education Program</namespace>
<namespace key="447" case="first-letter">Education Program talk</namespace>
<namespace key="710" case="first-letter">TimedText</namespace>
<namespace key="711" case="first-letter">TimedText talk</namespace>
<namespace key="828" case="first-letter">Module</namespace>
<namespace key="829" case="first-letter">Module talk</namespace>
</namespaces>
</siteinfo>
<page>
<title>Apache Accumulo</title>
<ns>0</ns>
<id>34571412</id>
<revision>
<id>571876229</id>
<parentid>570686710</parentid>
<timestamp>2013-09-07T05:11:02Z</timestamp>
<contributor>
<username>CJGarner</username>
<id>17441534</id>
</contributor>
<minor/>
<comment>/* See also */ Added Column-oriented DBMS</comment>
<text xml:space="preserve" bytes="4536">
{{Infobox software | name = Apache Accumulo | logo = [[File:Accumulo logo.png]] | screenshot = | c
```

We'll parse these articles and write them to Accumulo. First we'll devise a way of mapping the data to the Accumulo data model into one or more tables. Next we'll ingest the data using parts of the Accumulo Java client API. Finally we'll write some code to allow users to query these tables in various ways.

Data Modeling

Instead of formulating questions as database queries, application designers should break questions into scan operations, and ideally as few of them as possible per user request. To start getting used to this way of thinking, it is a good idea to begin application design with a single table. Additional tables can be added to the application if it is determined that they are necessary to support additional access patterns.

When deciding how to represent data in Accumulo, developers face two challenges: one, creating a new *schema* that allows them to model the data they will be creating, or two,

mapping existing data conforming to an existing schema to the Accumulo data model in such a way that supports the access patterns required.

A Quick Overview of Data Modeling

Data modeling is a task that involves identifying the structure of data. It is often performed at many levels and in many ways. A *schema* is a description of a the structure in some data.

A particular schema can be represented in various ways when stored in different systems. For example, one could define an Address to be made up of a street number, a city name, a state, a zip code, and whether or not it has been verified. An Address such as this is composed of five named elements, and has a specific meaning to a particular application. For this reason, this schema could be considered a *semantic* or *conceptual* data model. Conceptual models can be represented in several formats, or *logical* models.

For example, one might represent or *map* a particular Address to the JSON format as follows:

```
{
  street: '123 any street',
  city: 'anytown',
  state: 'CA',
  zipCode: 90210,
  verified: true
}
```

Or as XML,

```
<records>
  <address>
    <street required=true>123 any street</street>
    <city>anytown</city>
    <state>CA</state>
    <zip code>90210</zip code>
    <verified/>
  </address>
</records>
```

In other places one might represent an address as a table, such as in a relational database.

Both JSON, XML, and two-dimensional tables can be thought of as logical methods of capturing a conceptual model in a particular way for a particular purpose. In fact, many applications make use of several logical data models to represent the same conceptual data in various places. For example, an application may retrieve some rows from a relational database representing a user's profile and deserialize them as a programmatic object in memory, and then convert the programmatic object to JSON before sending to a web browser.

Accumulo has its own logical model as well, consisting of multi-dimensional keys and simple values. We covered the elements of the data model in the introduction.

The final type of data model is a *physical model*. A physical model describes how data elements are stored or transmitted in some physical medium. An example of a physical model is a B-Tree file for relational databases or the RFile file format used by Accumulo.

In some systems there are more than three levels of abstraction. It just depends on the complexity of the system. Additional levels of abstraction can help in keeping each individual level simple and manageable.

The model is again included here as a convenient reminder during our modeling task.

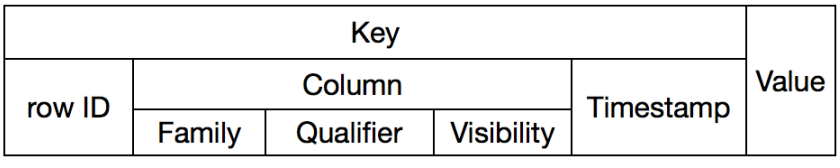

Figure 3-1. Accumulo Key Structure

Keep in mind that, as in any key-value system, if one knows the key, the associated value can be found very quickly. In Accumulo we should store the information that users are going to submit for a query in an element within the Accumulo key, and the information they want to retrieve in the value or in other elements of the key. To start, we'll create a table that allows users to specify an article title and retrieve the text or associated meta data. The way we'll map Wikipedia data to the Accumulo data model is as described in Table 3-2.

Table 3-2. One method for storing Wikipedia articles in Accumulo

Row	Column Family	Column Qualifier	Column Visibility	Value
page title	contents		*contents visibility*	*page contents*
page title	metadata	id	*id visibility*	*id*
page title	metadata	namespace	*namespace visibility*	*namespace*
page title	metadata	revision	*revision visibility*	*revision*
page title	metadata	timestamp	*timestamp visibility*	*timestamp*

The `WikipediaIngest` client will produce this table from a Wikipedia XML dump file.

```
Alternate Olympics contents: []     refimprove \x0AIn artistic gymnasti ...
Alternate Olympics metadata:id []    15713865
Alternate Olympics metadata:namespace []
Alternate Olympics metadata:revision []    7328338
Alternate Olympics metadata:timestamp []    2013-04-09T08:05:05Z
Ancient Olympic Games contents: []     pp-protected  for \x0A pp-move  ...
Ancient Olympic Games metadata:id []    19098431
Ancient Olympic Games metadata:namespace []
Ancient Olympic Games metadata:revision []    5207008
Ancient Olympic Games metadata:timestamp []    2013-08-19T10:52:07Z
Arena X-Glide contents: []    \x0AArena X-Glide is a swimsuit made ..
Arena X-Glide metadata:id []    23781846
Arena X-Glide metadata:namespace []
Arena X-Glide metadata:revision []    7328338
Arena X-Glide metadata:timestamp []    2012-09-24T15:35:39Z
...
```

Using the Accumulo API

Connecting to Accumulo

Both data and administrative actions are handled through an Accumulo *Connector*.

The Connector is obtained from an Accumulo Instance. An Accumulo instance is uniquely identified by a set of Zookeeper servers and an instance name. A single set of Zookeeper servers may manage multiple Accumulo instances, so the Accumulo instance is required.

```
String instanceName = "accumulo_instance";
String zooKeepers = "zoo_server_1:port,zoo_server_2:port";
Instance instance = new ZooKeeperInstance(instanceName, zooKeepers);
String principal = "user_name";
AuthenticationToken token = new PasswordToken("password");
Connector connector = instance.getConnector(principal, token);
```

The following sections illustrate how to use a Connector to read and write data. Connectors can also be used to perform administrative actions through the tableOperations(), securityOperations(), and instanceOperations() methods.

Managing Tables

Data in Accumulo is organized into Tables. Various table operations such as create, read, write, alter, and delete are controlled per user. More information on these permissions can be found in "Application Permissions" on page 147. Tables can be created using the TableOperations object.

```
TableOperations ops = connector.tableOperations();
ops.createTable('myTable');
```

The TableOperations object allows one to check whether a Table exists and to delete a Table as well.

```
if(ops.exists('myTable'))
  ops.delete('myTable');
```

Tables can also be created through the Accumulo shell.

```
> createtable myTable
```

Options that can be set on a table at creation time are whether to enable versioning and what timestamp type is used. Versioning is enabled by default, but can be disabled with an additional parameter to the `createTable` method:

```
ops.createTable('myTable', false);
```

or in the shell with the "no default iterators" flag:

```
> createtable myTable -ndi
```

The default time type is `TimeType.MILLIS` which uses the current system time in milliseconds since the epoch. The other possibility is `TimeType.LOGICAL` which uses a one-up counter. Logical time can be enabled through Java:

```
ops.createTable('myTable', true, TimeType.LOGICAL);
```

or in the shell:

```
> createtable myTable -tl
```

In our example code, we need to create a table to store Wikipedia articles. For this we'll use the following code:

```
TableOperations ops = connector.tableOperations();
if(!ops.exists("Articles")) {
  ops.createTable("Articles");
}
```

Insert

Writing data into Accumulo is accomplished by creating a Mutation object and adding it to a BatchWriter.

A mutation encapsulates a set of changes to a single row. The changes can be either *puts* or *deletes*. All the changes within a single mutation can be applied atomically, they either succeed or fail as a group, since a row is always fully contained within a single tablet which is always assigned to exactly one Tablet Server.

This makes it easy for applications to make concurrent updates to a row without worrying about Mutations being partially applied.

```
Mutation m = new Mutation("rowId");
m.put("columnFamily0", "columnQualifier0", "some value");
m.putDelete("columnFamily1", "columnQualifier1");
```

Strings vs Byte Arrays

In these examples we use String objects for elements of the Key. The advantage of using String objects is that they are human-readable and the sort order is relatively apparent. However, these don't have to be Strings. When supplied with Strings the Key class automatically converts them to UTF-8 encoded byte arrays.

For some applications, other objects can be serialized to byte arrays. When doing this, keep in mind that Accumulo will compare two byte arrays by comparing one byte at a time. If this doesn't result in the sort order desired, the serialization process can be manipulated to produce byte arrays that do sort properly.

Values of course are not sorted and also store byte arrays. This makes it possible to store binary or other data without worrying about having to escape certain characters.

When storing serialized objects other than Strings, it is possible to create a custom Formatter for displaying the objects in a human-readable way so that key-value pairs can be inspected in the Shell.

When calling put() on a Mutation, it isn't always necessary to use all the parts of the Key. One can leave parts of the Key "blank" by simply using an empty String.

For our Wikipedia data, we'll be creating one Mutation per article. Our parser will give us a Java object called WikipediaArticle containing the information for one article.

```
Mutation m = new Mutation(article.getTitle());
String wikitext = article.getText();
String plaintext = model.render(converter, wikitext)
    .replace("{{", " ")
    .replace("}}", " ");
m.put("contents", "", plaintext);
m.put("metadata", "namespace", article.getNamespace());
m.put("metadata", "timestamp", article.getTimeStamp());
m.put("metadata", "id", article.getId());
m.put("metadata", "revision", article.getRevisionId());
```

Committing Mutations

Mutations are sent to Accumulo in batches by adding them to a BatchWriter.

```
BatchWriter bw = connector.createBatchWriter("table_name", new BatchWriterConfig().setMaxMemory(MA
bw.addMutation(m);
bw.close();
```

Typical values for these parameters are

Max Memory
: 1000000 (1 million bytes or roughly 1 MB)

Max Latency
: 1000 (1 thousand milliseconds)

Max Write Threads
: 10

BatchWriters will efficiently group Mutations into batches depending on the way tablets are assigned to Tablet Servers in order to minimize the network overhead involved in sending Mutations to Tablet Servers. The size of batches is controlled via the Max Memory setting.

More memory means the BatchWriter can group more Mutations together before sending them over the network.

Max Latency determines how long a BatchWriter will wait before sending Mutations that do not comprise a full batch. This is so Mutations don't end up waiting a long time for a full batch to be created.

Most clients can experiment with these parameters to achieve the throughput and maximum latency they need.

The BatchWriter provides a **flush()** method that can be used to ensure all Mutations that haven't been sent are sent. This usually does not need to be called and can significantly degrade performance if used extensively as it will defeat the BatchWriter's attempts to amortize network overhead. It is provided to aid in synchronizing writes between two or more tables, for example.

BatchWriter.close() should be used to send any remaining Mutations and shutdown the write threads when a BatchWriter is no longer needed. Simply allowing Java to garbage collect the BatchWriter object may cause the final batches to be sent and threads to close, but explicitly calling close() is a better practice.

Handling Errors

The Accumulo client automatically handles errors related to the automatic failover from one TabletServer to another. This frees up the application designer to focus on the logic of the application rather than having to write the code to retry inserts.

Unlike some NoSQL databases, Accumulo assumes you do care to know that your Mutations have been successfully applied. If for some reason a Mutation fails, the Accumulo client will throw a MutationsRejectedException.

```
try {
  bw.addMutation(m);
  bw.close();
}
```

```
catch (MutationsRejectedException e) {
  for(KeyExtent extent : e.getAuthorizationFailures()) {
  }
  for(Entry<KeyExtent,Set<SecurityCode>> entry : e.getAuthorizationFailuresMap()) {
  }
}
```

When using BatchWriters, keep in mind that they create threads and should be closed via close() when no longer needed.

Thread Safety: The BatchWriter and BatchScanner create their own thread pools for efficiently communicating with multiple Tablet Servers simultaneously. If your client has more than one thread for other reasons keep in mind that Connector and BatchWriter objects are thread-safe, in that they can be shared and used by multiple threads without synchronization. But Scanners and BatchScanners are *not* thread-safe - each thread should obtain and use their own Scanners and BatchScanners or else use them in a synchronized manner so that only one thread is accessing them at a time.

Simple Lookups and Scanning

Reading data from Accumulo is accomplished with a Scanner. A Scanner returns data via an iterator over key-value pairs. By default a Scanner will return key-value pairs starting at the beginning of a table and eventually will return all key-value pairs.

```
Scanner scanner = connector.createScanner("table_name", new Authorizations());
for (Entry<Key,Value> entry : scanner) {
  Key k = entry.getKey();
  Value v = entry.getValue();
}
```

All the elements of the Key can be obtained from the Key object:

```
Key k = entry.getKey();
Text row = k.getRowID();
Text colFam = k.getColumnFamily();
Text colQual = k.getColumnQualifier();
ColumnVisibility colVis = k.getColumnVisibility();
Long ts = k.getTimestamp();
```

To specify a range of keys to scan over, use the setRange method of Scanner.

```
Scanner scanner = connector.createScanner("table_name", new Authorizations());
Range r = new Range(startKey, endKey);
scanner.setRange(r);
```

```
for (Entry<Key,Value> entry : scanner) {
}
```

In our Wikipedia example, we can retrieve all the info for a given article title

```
Scanner scanner = connector.createScanner("WikipediaArticles", new Authorizations());
scanner.setRange(new Range(articleTitle));
for(Entry<Key,Value> entry : scanner) {
  Key key = entry.getKey();
  String field;
  if(key.getColumnFamily().toString().equals("contents"))
    field = "contents";
  else
    field = key.getColumnQualifier().toString();
  String valueString = new String(entry.getValue().get());
  System.out.println(field + "\t" + valueString);
}
```

Crafting Ranges

Range has a variety of helpful constructors and utility methods to create a range covering all keys that match portions of a given key exactly.

To obtain all values in all columns for a specific row

```
Range.exact("row_0");
```

To get all values for a specific row and column family

```
Range.exact("row_0", "column_family_1");
```

To get the value for a specific row, column family, and column qualifier

```
Range.exact("row_0", "column_family_1", "column_qualifier_2");
```

The above will usually return only one value unless the table's versioning settings have been altered from the default or unless there happen to be more than one column visibility for this key.

To get a key with a specific column visibility

```
Range.exact("row_0", "column_family_1", "column_qualifier_2", "column_visibility_3");
```

To get the value for a fully specified key

```
Range.exact("row_0", "column_family_1", "column_qualifier_2", "column_visibility_3", 1234567890l);
```

Similarly, there are utility methods to create a range covering all keys that match a given prefix.

```
Range.prefix("row_prefix");
Range.prefix("row_0", "column_family_prefix");
Range.prefix("row_0", "column_family_1", "column_qualifier_prefix");
Range.prefix("row_0", "column_family_1", "column_qualifier_2", "column_visibility_prefix");
```

For example, if our WikipediaArticles table contains the following keys:

```
Whitaker
White
Whitehouse
Whitewash
Whiz
```

and we want to scan over all the keys that begin with the word white, sometimes signified by a wildcard in search systems as "white*", we can obtain the right Range via the following

```
Range whiteRange = Range.prefix("White");
```

Batch Scanning

Data can be retrieved for many Ranges simultaneously using a BatchScanner. Rather than a single Range, BatchScanners take a *set* of Ranges and communicate with many Tablet Servers in parallel threads to read all the data within the Ranges specified.

BatchScanners do not return data in sorted order because they retrieve data from many Tablet Servers at once.

When designing applications, keep in mind that the Scanner will always return key-value pairs in sorted order, but the BatchScanner will not.

A BatchScanner is obtained in a manner similar to that of a Scanner:

```
BatchScanner bscan = connector.createBatchScanner('myTable', auths, 10);
List<Range> ranges = new ArrayList<Range>();
ranges.add(new Range("white"));
ranges.add(new Range("cover"));
bscan.setRanges(ranges);
```

The last parameter designates the number of threads to use to communicate with Tablet Servers. Most clients will want to use more than one thread if there is more than one Tablet Server.

Results from BatchScanner are read the same way as from Scanner:

```
for(Entry<Key,Value> entry : bscan) {
  // access the elements of the entries
}
```

Batch-scanning comes in handy when looking up a set of record IDs retrieved from a secondary index or doing small joins between tables. See "Secondary Indexes" on page 70 on *Secondary Indexes* for examples of this.

Delete

A delete in Accumulo is a normal key with a delete flag set to true.

If a delete key is inserted, all keys with the same row and column as the delete key and the same or earlier timestamps than the delete key will be removed, along with their values. Deletes in Accumulo do not delete a specific key-value pair, rather they delete all earlier versions of the key.

The following deletes the column identified by the family "attributes" and qualifier "age" in the row identified by "bob".

```
Mutation m = new Mutation("bob");
m.putDelete("attributes", "age");
batchWriter.addMutation(m);
```

Delete mutations sort first when there are multiple mutations to be applied. What this means is that deleting a key and value pair and then inserting a different value with the same key will not work as expected. The end result of a compactation will be that both the original value and the newly inserted value will be deleted. To make this work as expected, insert a delete mutation, compact the table or a range on the table containing that key, then insert another mutation to recreate the key.

Updates and Versions

To update an existing key-value pair in Accumulo, simply insert a new key with the new value. By default, the new value will replace the old one, simply because it will be the value with the latest timestamp and Accumulo returns only the value with latest timestamp by default. You can configure Accumulo to keep multiple versions of a key-value pair by changing the maxVersions parameter of the VersioningIterator.

Using the shell:

```
user_name@instance_name> config -t table_name -s table.iterator.majc.vers.opt.maxVersions=2
user_name@instance_name> config -t table_name -s table.iterator.minc.vers.opt.maxVersions=2
user_name@instance_name> config -t table_name -s table.iterator.scan.vers.opt.maxVersions=2
```

or through Java:

```
connector.tableOperations().setProperty("table_name", "table.iterator.majc.vers.opt.maxVersions",
connector.tableOperations().setProperty("table_name", "table.iterator.minc.vers.opt.maxVersions",
connector.tableOperations().setProperty("table_name", "table.iterator.scan.vers.opt.maxVersions",
```

You can keep all versions by removing the VersioningIterator entirely.

```
user_name@instance_name> deleteiter -t table_name -n vers -all
```

Some applications may want to keep several versions on disk, return the latest by default, and allow clients to request more than the latest version whenever necessary.

To do this, the maxVersions option for *minc* and *majc* should be set to something greater than 1, say 10, and the *scan* option should be set to 1. Unless they specify otherwise, clients will see only the latest version for each value.

However, if they want to go back and view more versions, they can configure a Scanner to return more than one version thus:

```
Scanner scanner = connector.createScanner("myTable", auths);
IteratorSetting setting = new IteratorSetting(20, "vers", "org.apache.accumulo.core.iterators.user
VersioningIterator.setMaxVersions(setting, 10);
scanner.addScanIterator(setting);
```

Modifying Scanner Behavior

The behavior of a scan can be modified by configuring optional Iterators that apply only to the scan at hand.

Using IteratorSetting

Intermediate Applications

Applications that use a single-table and employ a single access pattern are the most fast, scalable and consistent. This type of application can work for a wide variety of applications. Beyond this there are techniques that applications use to provide a more general approach to accessing the data. In these applications however, the emphasis to minimize the work done at query time is still present, as this is the best way to ensure that applications scale as the amount of data and the number of concurrent users increases.

Secondary Indexes

Many applications can be built using a single table, and by accessing data in generally the same way each time, via the Row ID or a single range of Row IDs.

In our example application, we used the Wikipedia articles as the Row ID and so users can efficiently look up all the information for a given Article simply by doing a scan over the row containing all the key-value pairs associated with a given Article title.

But what if we wanted to find an article based on just a word contained in the text? Or to retrieve all articles with timestamps in a particular range? For these access patterns, building a *secondary index* can provide a solution.

A *secondary index* is simply another Accumulo table, but one that is designed to allow users to do the inverse of a normal table by scanning Values to obtain Row IDs. This is accomplished by copying the Values from the original table and storing them as the

Row IDs of a new table, and taking the Row IDs of the original table and storing them in the Column Qualifier of the new table.

Index Partitioned by Term

One way to build a secondary index is by storing terms to be queried in the RowID. For example, we can retrieve Wikipedia pages that contain a given word by building a table storing the words found in article text in the Accumulo RowID and the article title as the Column Qualifier.

Recall our Articles table, which used article titles as the RowID.

Table 3-3. Wikipedia article contents table

Row	Column Family	Column Qualifier	Column Visibility	Value
page title	contents		*contents visibility*	*page contents*
page title	metadata	id	*id visibility*	*id*
page title	metadata	namespace	*namespace visibility*	*namespace*
page title	metadata	revision	*revision visibility*	*revision*
page title	metadata	timestamp	*timestamp visibility*	*timestamp*

Now we create a secondary index that maps words appearing in Wikipedia pages to the page titles. An index organized by storing words or terms as the RowID is referred to as a *term-partitioned* index.

Table 3-4. Wikipedia index of contents

Row	Column Family	Column Qualifier	Column Visibility	Value
word	contents	*page title*	*page visibility*	

The entries for an index on a subset of Wikipedia articles are as follows:

```
white contents:Friendship Games []
white contents:Olympic Games []
white contents:Olympic Games ceremony []
whitfield, contents:Cotswold Olimpick Games []
whitsun, contents:Cotswold Olimpick Games []
whitsun. contents:Cotswold Olimpick Games []
who contents:Alternate Olympics []
who contents:Ancient Olympic Games []
who contents:Arena X-Glide []
```

Since we swap the positions of the RowIDs and values, an index like this is sometimes called an *inverted index*. However, note that we don't store the title / RowID from the Articles table in the value of the secondary index but rather we store titles in the Column Qualifier. This is because a term can appear in more than one article. A key in Accumulo can have exactly one value per row/column key, but a row can be associated with multiple

columns so we simply store the titles under the Column Qualifier and leave the Value blank.

This also works with the Article metadata fields. It is possible to store the index entries for all fields in the same table if we wish (Table 3-5). Often the concatenated Column Family and Qualifier from the original table is stored in the Column Family of the index so that a client can scan for a value in a particular column, if they so choose. By not specifying the Column Family a client can find the rows in which a given Value appeared in any Column, which can be a powerful technique.

Table 3-5. Wikipedia index of all fields

Row	Column Family	Column Qualifier	Column Visibility	Value
word	contents	*page title*	*page visibility*	
value	metadata:id	*page title*	*page visibility*	
value	metadata:namespace	*page title*	*page visibility*	
value	metadata:revision	*page title*	*page visibility*	
value	metadata:timestamp	*page title*	*page visibility*	

An example of an ingest client that writes data to the WikipediaArticles table and the WikipediaIndex table is as follows

```
Mutation m = new Mutation(article.getTitle());
String wikitext = article.getText();
String plaintext = model.render(converter, wikitext)
    .replace("{{", " ")
    .replace("}}", " ");
m.put("contents", "", plaintext);
m.put("metadata", "namespace", article.getNamespace());
m.put("metadata", "timestamp", article.getTimeStamp());
m.put("metadata", "id", article.getId());
m.put("metadata", "revision", article.getRevisionId());
try {
  writer.addMutation(m);
} catch (MutationsRejectedException e) {
  // ...
}
// write index entries
// tokenize article contents on whitespace and punctuation and set to lowercase
HashSet<String> tokens = Sets.newHashSet(plaintext.toLowerCase().split("\\s+"));
Value BLANK_VALUE = new Value("".getBytes());
for(String token : tokens) {
  Mutation indexMutation = new Mutation(token);
  indexMutation.put("contents", page.getTitle(), BLANK_VALUE);
  try {
    indexWriter.addMutation(indexMutation);
  } catch (MutationsRejectedException e) {
    e.printStackTrace();
```

```
    }
  }
```

Querying A Term-Partitioned Index

With this term-partitioned secondary index we can now look up articles by the value of any field, or by any word appearing in the article body. Once we know the titles of those articles, we can retrieve the information about the articles by doing lookups against the original WikipediaArticles table using a BatchScanner.

```
Authorizations auths = new Authorizations("contents");
Scanner scanner = conn.createScanner("WikipediaIndex", auths);
// look up all articles containing the word 'wrestling' in any field
scanner.setRange(new Range("wrestling"));
// store all article titles returned
HashSet<Range> matches = new HashSet<Range>();
for(Entry<Key,Value> entry : scanner) {
  matches.add(new Range(entry.getKey().getColumnQualifier().toString()));
}
// retrieve original articles
BatchScanner bscanner = conn.createBatchScanner("WikipediaArticles", auths, 10);
bscanner.setRanges(matches);
// fetch only the article contents
bscanner.fetchColumnFamily(new Text("contents"));
for(Entry<Key,Value> entry : bscanner) {
  System.out.println("Title:\t" + entry.getKey().getRow().toString() + "\n" +
            "Contents:\t" + entry.getValue().toString() + "\n\n");
}
```

Note here that we're only querying for a single term or a single range of terms at a time. If we needed to look up records that satisfy more than one criterion, say for example, all Wikipedia articles containing the word *baseball* with a metadata:timestamp newer than a year ago, we would need to do separate scans for each criterion and combine the article titles returned to get articles that match both criteria. To be specific, each scanner returns a set of titles matching the scan criterion applied, and the intersection of those sets of titles represent articles that match all criteria.

For multiple single-term lookups, such as all articles that contain both the word *baseball* and *record*, we can take advantage of the fact that the article titles, which are stored in the Column Qualifier, are sorted within a single Row and Column Family. This means that we can combine the titles returned from two single-term scans by simply comparing the titles as they are returned from each scan to find matches, rather than having to load all the titles in memory and perform set intersection using something like Java collections. This is important because we often can't predict how many records will match a given criterion.

Problem start to arise though when queries become more complex than this. These could be better addressed via a *document-partitioned* index, as described in following sections.

Consistency

Term-based secondary indices must be maintained along with the original table so that inconsistencies do not arise. Even though Accumulo does not provide multi-row transactions or cross-table transactions, this consistency can often be managed in the application.

One strategy for managing consistency between the original table and the secondary index table is to carefully order read and write operations. One can choose to wait until new rows are written to the original table first, and then write the corresponding entries to the secondary index. If for some reason a write to the original table fails, it can be retried before any index entries are written. This way clients aren't referred by the index to a row in the original table that doesn't exist.

Inversely, when deleting data from the original table, the index entries should be removed first, and then the row from the original table. This will prevent any clients from looking up data in the index that has not yet been written or which has been removed from the original table.

More complicated strategies may be required if an application involves concurrent updates to indexed data.

Index Partitioned by Document

A basic term-partitioned index is useful for retrieving all the data containing a particular word or having a specific value for a field. If we need to find all the data containing two different words, the client code would have to issue two scans to the basic index, bringing the document IDs for both back to the client side and intersecting the two lists. This can be inefficient if one or both of the terms appears in many documents, requiring many IDs to be retrieved. One solution to this problem is to build a *document-partitioned index*. In such an index, sets of documents are grouped together into partitions, and each partition is assigned an ID. The index is organized first by partition ID, then by word.

Document-Based Partitioning

Document-based partitioning is a concept employed by other systems. Google described a kind of document-based partitioning in a 2003 paper on the main indexes backing the famous Google search application, title "Web Search for a Planet: The Google Cluster Architecture"[3]

3. *http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.com/en/us/archive/googlecluster-ieee.pdf*

In this paper Google says that "the search is highly parallelizable by dividing the index into pieces (index shards), each having a randomly chosen subset of documents from the full index." and that "each query goes to one machine (or a subset of machines) assigned to each shard."

Similarly, some distributed MPP databases employ this type of strategy for distributed data across machines and executing queries across partitions.

Table 3-6. Document-partitioned table

Row	Column Family	Column Qualifier	Column Visibility	Value
partition ID	contents	*page title*	*page visibility*	*page contents*
partition ID	index	*word + page title*	*page visibility*	

The partition ID is the row portion of the key. The page contents are stored in one column of the row, and the index is stored in another column. To retrieve all the pages containing the words "wrestling" and "medal" in this partition, we can read over and merge the sorted lists of page titles obtained by scanning over the keys starting with *partition ID* : index : wrestling and starting with *partition ID* : index : medal.

This intersection can be accomplished on the server side with an appropriate Iterator. We discuss Iterators more in-depth in following sections of this chapter. An iterator that seeks to multiple starting points and intersects the results is generally called an *intersecting iterator*.

To use this method, a data set should be divided into an appropriate number of partitions so that the partitions are not too large or too small and there are enough of them that they are spread over the desired number of servers. Ideally you will have each partition fill an entire tablet, so its size should be somewhere between 256 MB and 10s of GB. For the Wikipedia data, we'll use 32 partitions.

Example code for building this table is as follows:

```
int NUM_PARTITIONS=32;
String wikitext = article.getText();
String plaintext = model.render(converter, wikitext);
plaintext = plaintext.replace("{{", " ").replace("}}", " ");
Mutation m = new Mutation(Integer.toString(article.getTitle().hashCode() % NUM_PARTITIONS));
m.put("contents" + '\0' + "enwiki", article.getTitle(), plaintext);
m.put("metadata", article.getTitle() + '\0' + "namespace", article.getNamespace());
m.put("metadata", article.getTitle() + '\0' + "timestamp", article.getTimeStamp());
m.put("metadata", article.getTitle() + '\0' + "id", article.getId());
m.put("metadata", article.getTitle() + '\0' + "revision", article.getRevisionId());
// tokenize article contents on whitespace and punctuation and set to lowercase
HashSet<String> tokens = Sets.newHashSet(plaintext.toLowerCase().split("\\s+"));
for (String token : tokens) {
  m.put("index", token + '\0' + "enwiki" + '\0' + article.getTitle() + '\0', BLANK_VALUE);
}
```

```
try {
  writer.addMutation(m);
} catch (MutationsRejectedException e) {
  throw new SAXException(e);
}
```

Unlike term-partitioned indexes, in a document-partitioned table Accumulo can make all the inserts for a given document atomically because they are all inserted into the same row. The tradeoff is that all partitions must be searched when querying for terms.

Key-value pairs in this table look as follows:

```
3 index:under\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:where\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:which\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:wikipedia,\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:wikiproject,\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:with\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 index:\xE2\x80\x94\x00enwiki\x00Category:WikiProject Olympics\x00 []
3 metadata:Category:Olympic records\x00id []    14329099
3 metadata:Category:Olympic records\x00namespace []    Category
3 metadata:Category:Olympic records\x00revision []    16019814
3 metadata:Category:Olympic records\x00timestamp []    2013-09-03T13:09:41Z
```

Querying a Document-Partitioned Index

When querying the data, we will use a BatchScanner along with an Intersecting Iterator to find relevant pages in each of the partitions.

The BatchScanner has a special mode in which it can be used to send queries to all tablets.

Code to query our document-partitioned index is as follows

```
Authorizations auths = new Authorizations("contents");
Scanner scanner = conn.createScanner("WikipediaPartitioned", auths);
// look up all articles containing the words 'wrestling' and 'medal'
IteratorSetting is = new IteratorSetting(50, IndexedDocIterator.class);
IndexedDocIterator.setColfs(is, "index", "contents");
IndexedDocIterator.setColumnFamilies(is, new Text[]{new Text("wrestling"), new Text("medal")});
scanner.addScanIterator(is);
for(Entry<Key,Value> entry : scanner) {
  System.out.println(entry.getKey().getColumnQualifier().toString());
}
```

Be aware than when using the document-partitioned index strategy with a BatchScanner, a single query is sent to all tablets involving all Tablet Servers in the query, whereas queries against term-partitioned indexes typically involve only a few machines. This reduces the number of concurrent queries that the cluster can support. By involving more machines, Accumulo can process more complex queries in a fairly bounded time frame. The document-partitioned indexing strategy described does minimize the network usage involved in these queries as well as round trips between the client and Tablet Servers by performing all intersection within the server memory and by storing the full record alongside the index entries for that record.

Indexing Data Types

Values in the original table can be just about anything. Accumulo will never interpret a Value and doesn't sort them. When building a secondary index, sort order of these items must be considered. In order to get values to sort properly they may need to be transformed.

For example, string representations of numbers, when sorted lexicographically as Accumulo does, do not end up in numerical order. These must be transformed in order to sort properly. One way to make lexicographic sorting match numeric sorting is to pad the numbers to a fixed width with zeros on their left.

```
0
1
11
2
```

versus

```
00
01
02
11
```

Padding requires you to know in advance the maximum possible number that appears in your data so that you can pad with enough zeros. Another solution for numbers is to convert them to a binary form. For example, long integers can be encoded in 8 bytes that will preserve numeric ordering when sorted lexicographically. This is true for positive or negative numbers, but if you have mixed positive and negative numbers you also have to flip the sign bit to get the negative numbers to sort before the positive ones.

Indexing IP Addresses

IP Addresses consist of four 8-bit numbers called octets, each of which ranges between 0 and 255, separated by a period. Because the String representation of an octet can be either one, two or three characters,

```
192.168.1.1
192.168.1.15
192.168.1.16
192.168.1.2
192.168.1.234
192.168.1.25
192.168.1.3
192.168.1.5
192.168.1.51
192.168.1.52
```

To avoid this situation, the octets can simply be zero-padded to sort IP Addresses properly.

```
192.168.001.001
192.168.001.002
192.168.001.003
192.168.001.005
192.168.001.015
192.168.001.016
192.168.001.025
192.168.001.051
192.168.001.052
192.168.001.234
```

Indexing Dates

```
20120101
20120102
20120201
20120211
20120301
```

Indexing Domain names

In the original BigTable paper the authors describe a method for storing Domain Names so that subdomains that share a common domain suffix sort together.

```
com.google.appengine
com.google.mail
com.google.www
com.msdn
com.msdn.developers
com.yahoo
com.yahoo.mail
com.yahoo.search
```

Joins

Although Accumulo doesn't provide SQL support, there are a few ways in which tables can be joined on the fly. This may not be tractable due to the size of tables, but in some cases it is feasible.

Joins and Sorts are expensive operations. Relational databases are optimized to perform this kind of work on tables at query time. Accumulo tables can be very large, so rather than performing joins and alternate sort orders at query time, one may consider trying to pre-compute these joined or sorted tables ahead of time, maintaining them as the original table is updated so users aren't waiting for results.

Of course not all types of joins and sorts can be anticipated. For truly ad-hoc joins, sorts, and aggregations, frameworks like Apache Hive or Cloudera's Impala can be used. There is work taking place to make Hive and Impala work with Accumulo.

The trade-off when using these tools is that, in order to make these operations fast enough to be used interactively, as many of the machines in cluster as possible are involved in the operation. This reduces the number of concurrent users that can be querying the system at any given time when compared to the number of users that can be performing scans against pre-computed Accumulo tables.

RowID to RowID

When two tables for some reason are to be joined based on the Row ID of each table, performing a join is a simple matter of scanning both tables once simultaneously and returning rows that appear in both.

The number of comparisons that needs to be done to accomplish this is on the order of the size of the two tables, O(a+b).

Value to RowID

When joining the values from one table, A, with the Row IDs of another table, B, it is best to scan the values of A and to look up whether B contains a Row ID for that value.

The number of comparisons required to perform this join is O(a * log2(b)) where a is the size of table A and b is the size of table B.

Value to Value

If two tables are to be joined on their values, it is more efficient to build an index of one of the tables and perform the join described above by scanning the values of one and doing lookups for each value in the other. This is because the number of comparisons required to join two tables' unsorted values is O(a*b) whereas the comparisons it takes to build an index of one is O(b*log2(b)) and building the index and doing the join, O(b*log2(b) + a*log2(b)) is less than O(a*b).

Security

There are a number of ways Accumulo controls access to data in its tables: *authentication*, *permissions*, *authorizations*, and *column visibilities*.

These can be thought of as applying at two levels - authentication and permissions at the higher application and tablet level, and authorizations and column visibilities at the lower data level.

Authentication

Some basic instance information such as instance ID, locations of master processes, and location of the root tablet can be retrieved from the `Instance` object itself. This information is available to anyone. To retrieve any additional information from Accumulo, you must *authenticate* as a particular user.

When you authenticate, you provide a user name and an `AuthenticationToken`. In return, you are given a `Connector` which represents a connection to Accumulo. The `Connector` can then be used to write data or read data from Accumulo, or to perform instance, security, or table operations.

The default `AuthenticationToken` is the `PasswordToken` that simply wraps a password for the user. `AuthenticationToken` can be extended to support other authentication methods such as LDAP.

Permissions

Once you are authenticated to Accumulo, the types of operations you are allowed to perform are governed by the permissions you are assigned. There are system permissions which are global and table permissions which are assigned per table. The user that creates a table is assigned all table permissions for that table. Users must be granted table permissions manually for tables they did not create. These are the permissions and what they allow users to do:

System

GRANT
: able to grant permissions to users

CREATE_TABLE
: able to create tables

DROP_TABLE
: able to remove tables

ALTER_TABLE
: able to configure and perform actions on tables

CREATE_USER
: able to create users

DROP_USER
: able to remove users

ALTER_USER
: able to change user authentication, permissions, or authorizations

SYSTEM
: able to perform administrative actions on tables or users

Table

READ
: able to scan a table

WRITE
: able to write to a table and perform some administrative actions for the table

BULK_IMPORT
: able to import files to a table

ALTER_TABLE
: able to configure and perform actions on a table

GRANT
: able to grant users permissions for a table

DROP_TABLE
: able to remove a table

Authorizations

Once a user is authenticated to Accumulo and is given permission to read a table, the user's `Authorizations` govern which key-value pairs can be retrieved. By default, if a user has write permission for a table, they may write keys that they do not have authorization to retrieve. This behavior can be changed by configuring a constraint on the table. With the `VisibilityConstraint`, users cannot write data they are not allowed to read.

```
connector.tableOperations().addConstraint(tableName, VisibilityConstraint.class.getName());
```

This can also be accomplished through the Accumulo shell. When the table is created, add an `-evc` flag to the `createtable` command. To add the constraint to an existing table, use the `constraint` command instead.

```
constraint -t tableName -a org.apache.accumulo.core.security.VisibilityConstraint
```

A user's Authorizations encapsulate a set of strings. These strings have no intrinsic meaning for Accumulo, but an application may assign its own meaning to them, such as groups or roles for its users.

To read data from Accumulo, a table name and a set of authorizations must be provided to either the `createScanner` or the `createBatchScanner` method of the `Connector`. Since the `Connector` is associated with a specific user, the authorizations provided when obtaining a `Scanner` or `BatchScanner` must be a subset of the authorizations assigned to that user. If they are not, an exception will be thrown.

Passing in a set of authorizations at scan time allows a user to act in different roles at different times. It also allows applications to manage their own users apart from Accumulo. You may choose to have one Accumulo user for an entire application and to let the application set the authorizations for each scan based on the current user of the application. Although this requires the application to take on the responsibility for managing accurate authorizations for their users, it also prevents users from having to interact with Accumulo or the underlying Hadoop system directly, allowing more strict control over access to your servers.

Column Visibilities

To determine which key-value pairs can be seen given a particular set of authorizations, each key has a *column visibility* portion. A column visibility consists of a boolean expression containing *tokens*, `&` (and), `|` (or), and parentheses, such as `(a&bc)|def`. To evaluate the boolean expression, each string must be interpreted as true or false. For a given set of authorizations, a string is interpreted as true if it is contained in the set of authorizations.

The visibility `(a&bc)|def` would evaluate to true for authorization sets containing the string "def" or containing both of the strings "a" and "bc". When the visibility evaluates to true for a given key and a set of authorizations, that key-value pair is returned to the user.

Tokens used in Column Visibilities can consist of letters, numbers, underscore, dash, colon, and as of Accumulo version 1.5, can contain periods and forward slashes.

Making Authorizations Work

Two things must happen in order for Authorizations to be effective in protecting access to data in Accumulo:

1. Properly apply column visibilities to data at ingest time

2. Apply the right authorizations at scan time

Often Accumulo applications will rely on specially vetted libraries for creating the proper column visibilities or else ingest clients are similarly reviewed and trusted.

For retrieving Authorizations, a separate service may be employed to manage the association of individual users to their set of Authorizations. This service is trusted by the application and the application itself is trusted to faithfully pass along the Authorizations retrieved from such a service to Accumulo.

A typical deployment may be like that of Figure 3-2.

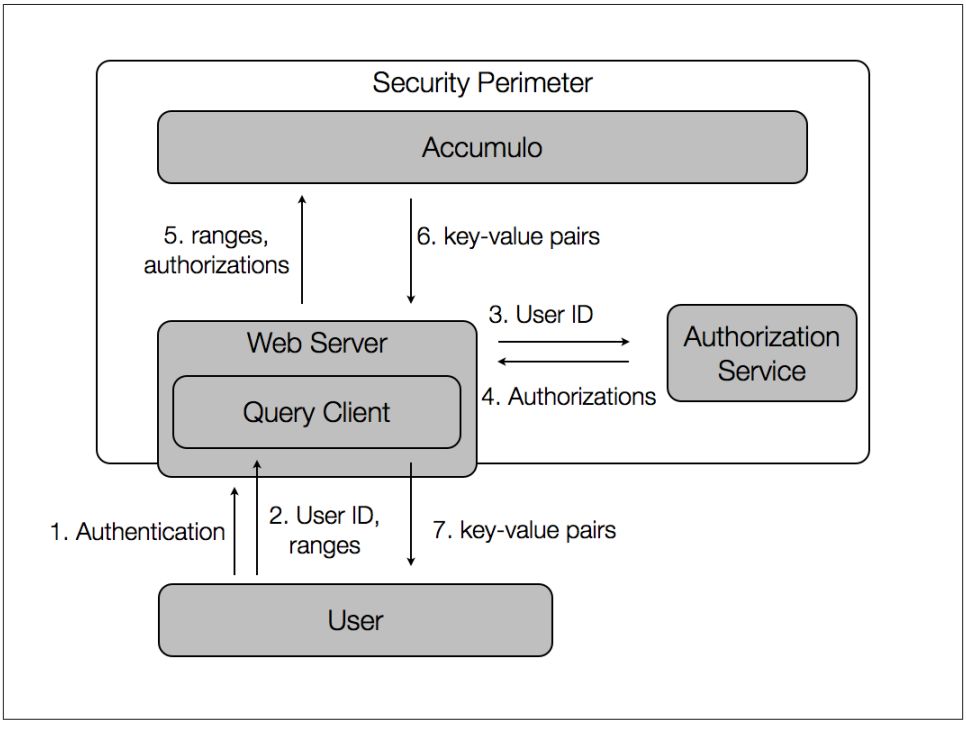

Figure 3-2. A Typical Accumulo Deployment

Other Security Considerations

In addition to Column Visibilities being properly applied at ingest time and the proper Authorizations retrieved and used in scans, there are some other things to consider when building a secure application on the Accumulo API:

- Direct access to Tablet Servers must be limited to trusted applications - this is because the application is trusted to present the proper Authorizations at scan time. A rogue client may be configured to pass in Authorizations the user does not have.
- Access to the underlying HDFS instance must not be allowed. Otherwise an HDFS client could open and read all the key-value pairs stored in Accumulo's RFiles without presenting the proper authorizations.
- Similarly, access should be disallowed to the underlying Linux filesystem on machines on which Tablet Server and HDFS DataNode processes run.
- Access to ZooKeeper should be restricted as Accumulo uses it to store configuration information about the cluster.

Network

The network that Accumulo uses to communicate between nodes and to HDFS and ZooKeeper should be protected against unauthorized access. Most Accumulo deployments do not use SSL between nodes, but rather use SSL between user browsers and trusted web applications.

Disk Encryption

Disks may be encrypted to prevent unauthorized reading of the data should a physical hard drive be stolen, but if those with physical access to the cluster are not trusted, then the operating system and memory of the machines participating the an Accumulo cluster would have to be similarly protected. When running Accumulo in multi-tenant environment such as a cloud infrastructure-as-a-service provider like Amazon's EC2 or Rackspace, consideration should be given to the security precautions implemented by the service provider.

For both situations, running in a cluster without trusting those with physical access or when running in the cloud, it may be most feasible to employ application-level encryption of values and to devise keys which are not sensitive. This is particularly challenging when it comes to building a secondary index.

If scans across ranges of terms in an index can be foregone, then using a strategy involving hashes of values as keys may still provide fast simple lookups. Ranges of terms could no longer be scanned because secure hashes of index terms would, by virtue of design of hash functions, no longer have any meaningful sort order. In this case, adjacent keys would have no relationship to each other.

Programming Tables

To customize a table's behavior, Accumulo provides a server-side programming framework called Iterators. Iterators are applied in succession, so that each iterator uses as its source data another iterator's output. Each iterator's output is sorted key-value pairs.

A series of iterators is applied in the three scopes in which a tablet server moves data:

1. Minor Compaction - when flushing the sorted in-memory map to a sorted file on disk
2. Major Compaction - when combining some number of sorted files into a single file
3. Scan - when reading all of its sorted in-memory and on disk structures to answer a scan query.

These scopes are labeled *minc*, *majc*, and *scan* respectively.

Iterators applied at minc and majc time permanently change the data stored in Accumulo. Scan-time iterators may be applied for all scans of a table, or on a per-scan basis.

To determine the order in which the iterators are applied, each iterator is assigned a priority. In each scope, the iterators are applied successively from the smallest priority to the largest.

Configuring Iterators

Iterators that implement the OptionDescriber interface can be configured through the shell using the *setiter* command. Iterators can also be configured through the Java API. Configuring iterators amounts to setting a number of string properties on a table. Table properties are stored in Zookeeper.

Iterators can also be configured manually by setting the appropriate properties, but this is not recommended because it is more error-prone. However, to change the parameters of an existing iterator, the easiest method is the change the string property through Accumulo's API or the shell.

Through the Shell

The *setiter* command requires a number of parameters. The iterator class can be specified with *-class <className>* where the class name is fully qualified. Some iterators have built-in flags, so you can use those as shortcuts instead of specifying -class with the entire class name. These are :

- the AgeOffFilter (-ageoff)
- the RegExFilter (-regex)
- the ReqVisFilter (-reqvis)

- and the VersioningIterator (-vers).

In addition to the class name, you must specify between one and three scopes, *-minc*, *-majc*, and / or *-scan*. You must also specify the iterator's priority, which controls the order in which the iterators are applied. Specifying a shorthand name for the iterator is optional with the *-n <iteratorName>* parameter. If you are in a table context (which can be attained with the command *table <tableName>*) the iterator will be applied to the current table. A different table can be specified with the *-t <tableName>* option.

For example, to configure an AgeOffFilter that ages off data older than one hour, do the following:

```
user@instance tablename> setiter -ageoff -scan -p 10
AgeOffFilter removes entries with timestamps more than <ttl> milliseconds old
----------> set AgeOffFilter parameter negate, default false keeps k/v that pass accept method, tr
----------> set AgeOffFilter parameter ttl, time to live (milliseconds): 3600000
----------> set AgeOffFilter parameter currentTime, if set, use the given value as the absolute ti
user@instance tablename>
```

Through the API

To configure an iterator through the Java API, create an IteratorSetting object.

```
IteratorSetting setting = new IteratorSetting(1, AgeOffFilter.class);
AgeOffFilter.setTTL(setting, 3600000);
connector.tableOperations().addIterator(setting);
```

By default, this adds the iterator to all scopes, but you can specify which scopes to use with an EnumSet passed to the constructor.

By Manually Configuring Properties

After you have configured an iterator on a table, type *config -t <tableName>* in the shell to see the actual properties. *config -t <tableName> -f iterator* reduces the list to the iterator-related properties. You could set these properties manually on the table using the shell or the Java API. To add the AgeOffFilter to the majc scope in addition to the scan scope, you would run the following commands:

```
config -t tableName -s table.iterator.majc.ageoff.opt.ttl=3600000
config -t tableName -s table.iterator.scan.ageoff=10,org.apache.accumulo.core.iterators.user.AgeOf
```

or in Java:

```
connector.tableOperations().setProperty("table.iterator.majc.ageoff.opt.ttl","3600000");
connector.tableOperations().setProperty("table.iterator.majc.ageoff","10,org.apache.accumulo.core.
```

It is a good idea to set the options first, in case the iterator happens to run between the time you set one property and the next.

VersioningIterator

There is one programmable iterator that is configured for all Accumulo tables by default: the VersioningIterator. Each Accumulo Key has a timestamp that is used for versioning. Let's say you insert a key with row *a*, column family *b*, and column qualifier *c*, with value *d*. If you insert the same key at a later time with value *e*, Accumulo considers the second key-value pair to be a more recent version of the first. By default, only the latest time-stamped version will be kept. You can configure the VersioningIterator to keep a different number of versions, or you can remove it to keep all versions.

Filters

Filters are iterators that simply decide whether or not to include existing key-value pairs. They do not alter the key-value pairs in any way.

Built-In Filters

Some useful filters are provided with Accumulo.

Timestamp-based Filters
: *AgeOffFilter*
  : Removes keys when their timestamps differ from the current time by more than a specified parameter (in milliseconds).

  ColumnAgeOffFilter
  : Stores a separate parameter for each column, to age off columns at different rates.

  TimestampFilter
  : Only keeps keys with timestamps earlier and / or later than given start and end parameters.

RegExFilter
: Returns key-value pairs that match a Java regular expression in a particular portion of the key or value (the row, column family, column qualifier, or value). Regular expressions can be provided for any subset of these four, and matches can be determined by ORing or ANDing together the results of each individual regular expression.

ReqVisFilter
: Removes keys with empty column visibilities.

Custom Filters

Custom filters can be written by extending org.apache.accumulo.core.iterators.Filter. Subclasses of Filter must implement an accept method that takes Key and Value as parameters and returns a boolean.

Combiners

Combiners are iterators that combine all the versions of a key-value pair into a single key-value pair, instead of keeping the most recent versions of a key-value pair like the VersioningIterator. Combiners work on sets of keys that only differ in their timestamp. If you want to combine an entire column of keys, you'll have to write a custom Iterator.

Built-In Combiners

Accumulo comes with a number of Combiners. These are in the org.apache.accumulo.core.iterators.user package. An additional example combiner is the StatsCombiner in the org.apache.accumulo.examples.simple.combiner package. Using this combiner is illustrated in Accumulo's README.combiner example.

LongCombiners
: An abstract Combiner that interprets Accumulo Values as Longs. It comes with three possible encoding types: `STRING`, that prints and parses the number as a string; `LONG`, that encodes the number in exactly 8 binary bytes; and `VARNUM`, that uses a variable-length binary encoding.

  MaxCombiner
  : Extends the LongCombiner, interpreting Values as Longs, and returns the maximum Long for each set of Values.

  MinCombiner
  : Extends the LongCombiner, interpreting Values as Longs, and returns the minimum Long for each set of Values.

  SummingCombiner
  : Extends the LongCombiner, interpreting Values as Longs, and returns the sum of the set of Values.

SummingArrayCombiner
: Interprets Values as an array of Longs, and returns an array of sums. If the arrays are not the same length, the shorter arrays are padded with zeros to equal the length of the largest array.

Custom Combiners

Custom combiners can be written by extending org.apache.accumulo.core.iterators.Combiner. Subclasses of Combiner must implement a reduce method that takes Key and Iterator<Value> as parameters and returns a Value. If the Values are always interpreted as a particular Java type, the TypedValueCombiner<V> can be used. This Combiner uses an Encoder to translate the type V to and from a byte array. Subclasses of TypedValueCombiner implement a typedReduce method that takes Key and Iterator<V> as parameters and returns a V. A LongCombiner is provided that interprets the values as Longs.

Other Built-In Iterators

LargeRowFilter
: Suppresses entire rows that have more than a configurable number of columns. It buffers the row in memory when determining whether it should be suppressed or not, so the specified number of columns should not be too large.

RowDeletingIterator
: Uses a special marker to indicate that an entire row should be deleted. The marker consists of a row ID, empty column family, qualifier and visibility, and a value of "DEL_ROW".

RowFilter
: An abstract Iterator that decides whether or not to include an entire row or not. Subclasses of RowFilter must implement an acceptRow method that takes as a parameter a SortedKeyValueIterator<Key,Value> (which will be limited to the row being decided upon) and returns a boolean. This allows you, for example, to fetch the contents of column A based on the contents of column B, or based on some feature of the row as a whole.

MapReduce

One advantage of Accumulo's integration with the Hadoop is that MapReduce jobs can be made to read input from Accumulo tables and also to write results to Accumulo tables.

Formats

Accumulo provide MapReduce input and output formats that read from Accumulo and write to Accumulo directly. There are input and output formats for both MapReduce APIs, org.apache.hadoop.mapred and org.apache.hadoop.mapreduce. Internally, each Mapper has a Scanner over a particular range, which provides key-value pairs to the map function. Each Reducer has a BatchWriter that sends data to Accumulo via Text (table name), Mutation pairs. Accumulo's InputFormatBase can be extended to provide arbitrary K,V to a Mapper, where K,V can be derived from any number of Key, Value pairs.

Bulk Import

In some cases, rather than writing data to Accumulo incrementally, an application will want to provide a bunch of new files to Accumulo all at once. A MapReduce output format, the AccumuloFileOutputFormat, is provided for creating a set of files for bulk import into Accumulo. Each Reducer will create a separate file, and it should be noted

that you must output data from your reduce method in sorted order. For example, a Reducer may take the following form:

```
public static class ReduceClass extends Reducer<Text,Text,Key,Value> {
  public void reduce(Text key, Iterable<Text> values, Context output) throws IOException, Interrup
    for (Text value : values) {
      // create outputKey and outputValue
      output.write(outputKey, outputValue);
    }
  }
}
```

If the for loop does not create output keys in sorted order, you can instead insert the Key, Value pairs into a TreeMap in the for loop, and then iterate over the TreeMap to do the output.writes at the end of the reduce method.

Using Ingesters and Combiners for Continuous MapReduce

The MapReduce programming model is designed for batch computation rather than incremental computation. For example, when calculating word counts over a set of ten thousand documents, a MapReduce job would read all the documents and calculate how many times each word appears. If we then add a single new documents to the corpus, we either must read in all the original ten thousand documents again along with the new document, or read all the previous word counts and add the counts from the one new document to the existing counts. Either option is a lot of work to add just one document. As a result, incrementally updating a result set such as this in an efficient way tends to be done by waiting until there are a good number of new documents before updating the result set, the cost of which is that the result set is not updated very often.

In contrast, Accumulo's combiners can be used to incrementally update a result set much more efficiently. In MapReduce, you can specify a combiner class that will be used to combine together intermediate output from the map phase before it is sent to the reduce phase. You can think of Accumulo's combiners as performing a similar function.

In the word count example, you map over documents and output "word,1" for each word in the document. A combiner sums up the word counts for each mapper, and sends those intermediate counts to a reducer, which tallies the final counts. In this simplest MapReduce use case, the same class is used for the reducer and the combiner. To perform word count in Accumulo, you configure a LongCombiner on the table, and insert entries with row "word" and value "1". After writing the data into Accumulo the computation is complete.

If the reduce performs an additional calculation, this can be performed on the Accumulo table on the fly at any point in the process of writing data into Accumulo. The final results of the computation would typically be performed by a scan-time iterator and are not persisted in the table. An example of a computation that might be performed on the fly is a running average.

Proxy Service

In addition to the standard Accumulo processes, there is an option to start up a proxy service. This service provides a Thrift API for interacting with Accumulo. Accumulo comes compiled with C++, Python, and Ruby clients for interacting with the Thrift API. Clients for other languages may be generated. The details of the API can be found at accumulo/proxy/thrift/proxy.thrift.

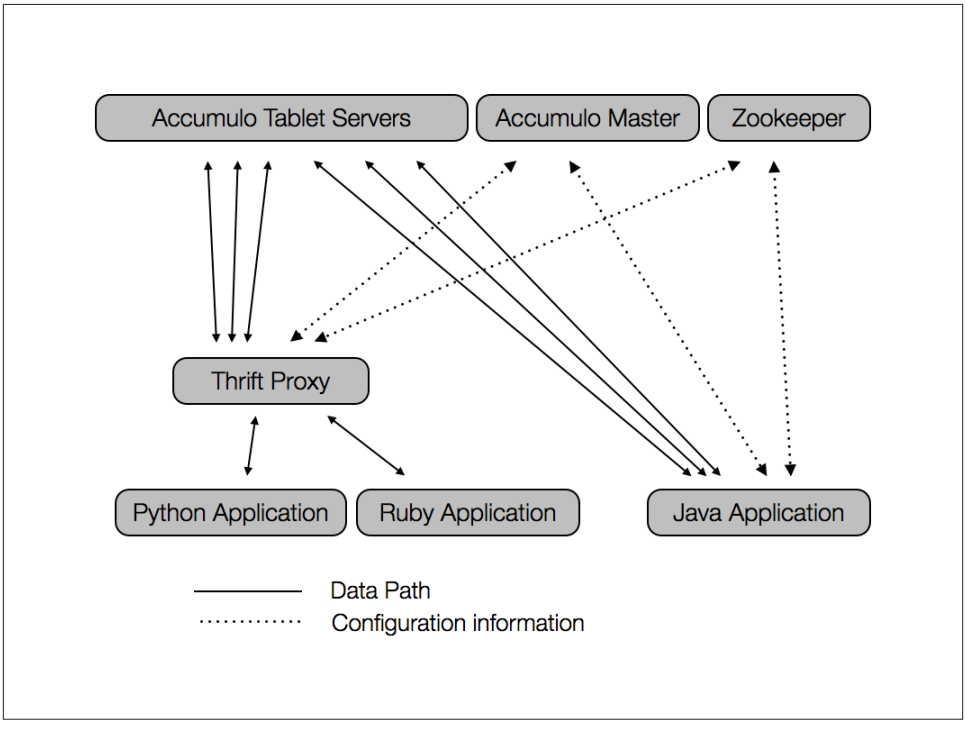

Figure 3-3. Applications with Accumulo Proxy

Starting a Proxy

You must pass a configuration file to the proxy process when starting it up. An example configuration file is included with accumulo at accumulo/proxy/proxy.properties. To start the proxy service, run `accumulo proxy -p /path/to/proxy.properties`.

Using a Proxy

You must install the Thrift libraries for your language of choice to use that language with the proxy. The syntax involved will then depend on the language. Language-specific

Thrift tutorials are available at *http://thrift.apache.org*. A sample of the Python and Ruby APIs follows.

Python	Ruby
socket = TSocket.TSocket(*localhost*, 42424)	socket = Thrift::Socket.new(*localhost*, 42424, 9001)
transport = TTransport.TFramedTransport(socket)	transport = Thrift::FramedTransport.new(socket)
protocol = TCompactProtocol.TCompactProtocol(transport)	protocol = Thrift::CompactProtocol.new(transport)
proxy = AccumuloProxy.Client(protocol)	proxy = AccumuloProxy::Client.new(protocol)
transport.open()	transport.open()
login = client.login(*root*, {*password*: *secret*})	login = proxy.login(*root*, {*password* ⇒ *secret*})
proxy.createTable(login, *tablename*, True, TimeType.MILLIS)	proxy.createTable(login, *tablename*, true, TimeType::MILLIS)

CHAPTER 4
Internals

This chapter describes the internal workings of Accumulo. While Accumulo may be used without a knowledge of its internals, developing an increased understanding of how Accumulo works will make it easier to understand and make decisions about how best to interact with Accumulo.

Tablet Server

Accumulo tables are split into contiguous ranges called *tablets*. Each tablet is assigned to a *tablet server* that is responsible for all reads and writes for the tablet. Each tablet server may be assigned hundreds or even a thousand tablets or more. If an Accumulo instance reaches well over one thousand tablets per server, it is time to start making adjustments: merging tablets, deleting old data, rethinking one's table design, or increasing the size of the cluster.

An Accumulo instance typically runs one tablet server per slave node (for example, each server that is running an HDFS DataNode). A tablet server registers with an Accumulo instance by obtaining a lock in Zookeeper. If a tablet server loses or is unable to monitor its lock, it will kill itself.

The following sections describe many of the operations tablet servers perform on their tablets.

The reading and writing for an individual tablet are governed by a log-structured merge-tree approach[1] as interpreted in the BigTable design. This technique is optimized for sequential writes and is most useful when much more data is written than is read.

When reading or writing new data, the Accumulo client library first locates which tablet should contain the data and which tablet server is hosting that tablet. This is accomplished by looking up the location of the *root tablet*, which contains information about the tablets of the *metadata table*. Once the desired metadata tablet and its location are read from the root tablet, the desired data tablet and its location are looked up in the metadata tablet. The client library caches tablet locations that it has found previously, so when data is looked up again it may be able to skip one or more of these steps.

Starting in Accumulo 1.6, the root tablet is considered to be in a separate *root table* that is never split. In earlier versions, the root tablet is considered to be part of the metadata table. This change does not affect how the client interacts with Accumulo, but it simplifies Accumulo's internal implementation.

Write path

Within a tablet, incoming writes are committed to an on-disk write-ahead log. In Accumulo 1.5.0 and later, the write-ahead log is stored directly in HDFS. The number of replicas required for the write-ahead log files can be configured. In Accumulo 1.4 a custom logging process writes to local disk instead. The write-ahead logs are manually replicated onto the disk of one remote server in addition to local disk.

Once the new data is confirmed to be on disk, it is inserted into a sorted in-memory structure. At this point, the data will available for reads.

When the resource manager determines that a tablet server's allocated memory is becoming full, it selects some tablets for *minor compaction*, which means flushing data from memory to disk. Successful minor compaction results in a new sorted HDFS file associated with the tablet.

Read path

Within a tablet, reads consist of constructing a merged view of sorted in-memory and on-disk structures. The on-disk structures are HDFS files associated with the tablet and do not include write-ahead log files, which are unsorted.

1. O'Neil, P., Cheng, E., Gawlick, D., AND O'Neil, E. The log-structured merge-tree (LSM-tree). Acta Inf. 33, 4 (1996), 351–385.

Resource Manager

Each tablet server has a resource manager that maintains thread pools for minor and major compaction, tablet splits, migration and assignment, and a read-ahead pipeline. The sizes of these thread pools are configurable via appropriate properties, with the exception of the split and assignment thread pools which each contain a single thread.

```
tserver.compaction.major.concurrent.max
tserver.compaction.minor.concurrent.max
tserver.migrations.concurrent.max
tserver.readahead.concurrent.max
tserver.metadata.readahead.concurrent.max
```

Minor Compaction

Minor compaction is the process of flushing data that is sorted in memory onto a sorted file on disk.

The resource manager includes a memory manager that periodically evaluates the current states of the server's assigned tablets and returns a list of tablets that should be minor compacted. The memory manager is pluggable with its class being configured by `tserver.memory.manager`.

The default memory manager is the `LargestFirstMemoryManager`. If the total memory usage of all the tablets on the tablet server exceeds a dynamic threshold, or if the time of a tablet's last write is far enough in the past, a tablet will be selected for minor compaction. The threshold is adjusted over time so that the highest memory usage before a minor compaction is between 80 and 90 percent of the maximum memory allowed for the tablet server, `tserver.memory.maps.max`. If too many minor compactions are already queued, it will not select additional tablets for minor compaction until the queue has decreased. The number of minor compaction tasks allowed in the queue is twice the number that can be executed concurrently, which is controlled by the global configuration property `tserver.compaction.minor.concurrent.max`. The per-table property `table.compaction.minor.idle` controls how long a tablet can be idle before becoming a candidate for minor compaction. The memory manager selects the tablet with the highest combination of memory usage and idle time (memory * 2^(minutes idle / 15)), or if memory is not too full, the idle tablet with the highest combination of these.

If a tablet has more than a specified number of write-ahead log files (`table.compaction.minor.logs.threshold`), it will be minor compacted outside of the memory management system. This is to reduce recovery time in the case of tablet server failure.

Major Compaction

When there are too many files for a tablet, read performance will suffer because each lookup must perform a merged read of all the sorted files and in-memory structures for

the tablet. For this reason, each tablet server has a thread pool devoted to merging together files within a tablet. This process of merging together files is called *major compaction*. If all of a tablet's files are merged into a single file, it is called a *full major compaction*.

A full major compaction for a table may be requested by the user through the shell or API. The compaction may be restricted to a range of tablets specified by start and end rows. A user-initiated compaction may also be canceled.

Major compactions are also initiated automatically by tablet servers. How often to evaluate whether files need to be merged is governed by the global property `tserver.compaction.major.delay`. When to merge files and which files are merged is governed by the per-table property `table.compaction.major.ratio`. To determine whether a tablet's files are in need of major compaction, the tablet server sorts the files by size. If the size of the largest file times the compaction ratio is less than or equal to the sum of the sizes of all the files, the set of files is merged into a single file. If this is not true, the largest file is removed from the set and the remaining files are evaluated. This is repeated until a set of files is selected for compaction or until there is only one file left in the set. This algorithm is chosen to reduce the number of times data is rewritten through major compaction.

There is a maximum for the number of files a single major compaction thread is allowed to open, `tserver.compaction.major.thread.files.open.max`. If a set of files is selected for major compaction that contains more files than this maximum, the compaction will merge the N smallest files where N is the number of files that are allowed.

Merging Minor Compaction

The major compaction algorithm can result in a large number of files as the tablet size grows, so there is another per-table property that provides a hard maximum on the number of files per tablet, `table.file.max`. When a tablet reaches this number of files, the tablet server will not create new files via minor compaction. Instead, the tablet server will choose the tablet's smallest file, and merge the data from this file and the in-memory structure into a new file. This process is called a *merging minor compaction*.

Consider adjustments made to the `table.file.max` property carefully. Making it low may increase read performance while decreasing write performance.

Splits

When a tablet's size reaches a configurable threshold, the tablet server will decide to split it into two tablets. The threshold is set in bytes via `table.split.threshold`. Conceptually, the server must create two new tablets, split the original tablet's data appro-

priately between the two new tablets, and update the metadata table with information about the two new tablets, removing information about the original tablet as needed.

To make splitting a light-weight metadata operation that does not require rewriting the original tablet's data, the names of a tablet's files are stored in the metadata table. This allows files to be associated with more than one tablet. When reading data from its files, a tablet restricts its reads to the range of keys in its own domain. When a tablet is split into two new tablets, both of the new tablets will use the files of the original tablet. Splitting takes priority over compaction so that a tablet that is growing very quickly can be split into as many tablets as needed before the new tablets start compacting their files, which would otherwise be an ingest bottleneck.

Fault tolerance during splitting is achieved by a multi-step process. Since multiple rows in the metadata table must be updated, it is not inherently an atomic operation. First the tablet is closed so that no new writes are accepted. Then three writes are made to the metadata table: the tablet is made smaller and is marked as splitting; a new tablet is added; and the original tablet's splitting marks are removed. The tablet server swaps the new tablets for the old tablet in its online tablet list, and the master is informed of the new tablets. If the tablet server goes down during the splitting process, a new tablet server will pick up the splitting process where it left off. The new server determines that splitting must be continued if a tablet it is assigned has splitting marks in the metadata table.

Splitting a table into enough tablets is essential to being able to take advantage of the parallelism of the system. By default, a table's tablets will be spread evenly across the tablet servers. For some applications it is better not to leave the split points to chance. Split points can be specified when a table is created, or added to an existing table.

Write-Ahead Logs

Write-ahead logs are used to guarantee data integrity in the presence of server or process failures. Since each tablet contains an in-memory map which stores recently written data, this data must be persisted to disk first to ensure that it isn't lost. A single write-ahead log file is only written to by a single tablet server, but log entries for all tablets assigned to that server may be intermingled in the same log file.

The logging mechanisms are different in different versions of Accumulo. In Accumulo 1.4 and earlier, there were custom logging processes. Each slave node would typically be configured to run one tablet server process and one logger process. When a tablet server received new data, it would log the data to the local logger process, if present, and to one remote logger process. It would send data to two remote loggers if a local logger was not present.

In Accumulo 1.5 and later, data is logged directly to a file in HDFS, so that separate logger processes are not needed. The tablet server waits until the data is replicated appropriately by HDFS, and then it proceeds with inserting the data into the in-memory map for the appropriate tablet. The HDFS replication of write-ahead log files is controlled with the `tserver.wal.replication` property. If set to zero, it will use the HDFS default replication. Setting this property to 2 will provide similar performance and data protection as Accumulo 1.4. Setting it to 3 will provide even better data protection, ensuring data is written to three different disks before it is committed to Accumulo. However, this will use more disk I/O resources when writing and may affect ingest performance. It is not recommended that the replication of write-ahead log files be set to 1, as a single server failure could result in data loss.

Recovery

Like regular data files, the write-ahead log files containing data for a given tablet are written to the metadata table row for that tablet. If a tablet server is assigned a tablet that has write-ahead log entries in the metadata table, the tablet server will conduct a log recovery before bringing the tablet online. Since the logs contain unsorted data for multiple tablets, the files are first sorted so that servers don't need to read through irrelevant data to recover the data for a single tablet. Sorting of log files is done by a work queue managed in zookeeper. The master submits files to this work queue when it reassigns tablets that have write-ahead log entries. Each tablet server monitors the work queue for new files that need to be sorted and maintains a thread pool for sorting tasks. The size of the thread pool is controlled with the `tserver.recovery.concurrent.max` property.

A single tablet server will win the race to grab a file from the work queue and begin sorting it. It reads a large chunk of the file into memory and writes it out sorted by entry type, tablet, and original order in the file. It repeats this process, creating a new file for each chunk that fits in memory, until the entire file is sorted. The size of the sorted chunks defaults to 200MB and can be adjusted by changing the `tserver.sort.buffer.size` property. A typical write-ahead log file size is 1GB, controlled by `tserver.walog.max.size`. Under normal circumstances a tablet will only have one or two write-ahead log files, so with the default settings there may be 5 to 10 sorted file chunks to read when recovering data.

In addition to logged data, the file contains minor compaction start and finish markers as well as specifying a concise tablet ID for each tablet referenced. The usual identifier for a tablet is its key extent (start row exclusive, end row inclusive), but a short ID is assigned in the write-ahead logs so that it can be specified in fewer bytes. Once the sorting is completed, the tablet server that will host the tablet begins recovering the data for that tablet. It conducts a merged read of all the sorted log chunks that may contain data for the tablet. First it finds the tablet ID, then it uses the ID to find the last minor compaction that succeeded for the tablet. This determines which log entries must be

replayed. Once the tablet server has replayed the log entries, it minor compacts the tablet. Currently, a tablet server only recovers one tablet at a time.

File formats

The file format used by Accumulo is the RFile, which stands for relative key file. It is based on the concept of an indexed sequential access map file, which is to say that there is a data portion that is sorted, and there is an index portion with a subset of the keys and corresponding location offsets pointing into the data portion. To locate a key in the file, the index is read into memory and the latest key that sorts before the desired key is located. The location offset retrieved is used to seek into the data portion of the file, and then keys are scanned sequentially to find the desired key.

The actual design has a number of optimizations over this basic model. First, the data portion of the file is compressed in blocks, controlled per table with the `table.file.compress.blocksize` property which defaults to 100KB. The size is prior to compression. The index then contains the key and location pointing to the beginning of every compressed block. A single index approach doesn't work well for very large files because as the file size grows, so must the size of the index. Since the index is read into memory when the file is opened, it will take longer to open larger files. This can be mitigated somewhat by increasing the compressed block size to decrease the size of the index, but then lookup times will also increase because the data blocks are scanned sequentially when looking up keys. To support very large files, Accumulo 1.4 introduces a multi-level index. The property `table.file.compress.blocksize.index`, defaulting to 128KB prior to compression, sets the maximum size of an index block. When writing a file, the index to the beginnings of the compressed data blocks is accumulated in memory. When the index reaches its maximum size, a *level 0* index block is written out, a *level 1* index is started with a pointer to the level 0 index block, and a new level 0 index block is started with pointers to data blocks. When all the data has been written out, any remaining index blocks are written to the file. When opening the file, only the highest level index must be read into memory. There is no limit to the number of index levels, but a two-level index is sufficient for files in the 10s of GB with the default settings. See Figure 4-1 for an illustration of the on-disk data layout for a two-level index.

Note that compressed blocks (~100KB) are not the same as HDFS blocks (128MB or more). There will be many compressed blocks per HDFS block.

Accumulo 1.5 features an additional optimization to reduce the time needed to sequentially scan a data block. Once a given data block has been accessed once, it is cached in memory. When it has been accessed twice, it begins building a dynamic, ephemeral

index by storing the key and pointer corresponding to the midpoint of the block. As the block continues to be used, when it is accessed 2^N times, N additional keys will be added to the ephemeral index.

Relative Key Encoding

Within a compressed block, Accumulo performs compression of identical portions and identical prefixes of portions of consecutive keys. A single byte is used to encode whether the row, column family, column qualifier, column visibility, and/or timestamp match those of the previous key. Of the remaining 3 bits in that byte, one is used to indicate whether any prefix compression is present for the key, another is used as the deletion flag for the key, and the last is unused. If prefix compression is present, an additional byte is used to indicate whether the row, column family, column qualifier, and/or column visibility have a common prefix as the previous key, and whether the timestamp is expressed as a difference from the previous timestamp. A common prefix must be at least two bytes. The remaining 3 bits of the prefix compression byte are unused.

If a portion of the key matches that portion of the previous key identically, as indicated in the first byte, no additional information needs to be written for that portion. If a portion of the key has a common prefix with that portion of the previous key, the length of the prefix is written followed by the remaining bytes for that portion of the key. In the case of the timestamp, there is no prefix length, but the difference from the previous key is written. If a portion of the key has no common prefix with the previous key, the entire portion is written.

Locality Groups

To enable greater intermingling of different types of data in a single Accumulo table, RFiles also support locality groups. This feature allows sets of column families to be grouped together on disk, which can result in better compression. It also allows applications to tune a table's disk layout to better suit its access patterns. For example, if two columns are always queried together, the columns could be put in a locality group. Alternatively, if one column contains very large data, such as image files, and another column contains much smaller data, such as text, these columns could be put in different locality groups to improve the lookup times when only retrieving text data.

By default, all columns are in a single group, the default locality group. Locality group mappings can be added or changed at any time. When new files are created through minor or major compaction, they will use the newest locality group configuration.

Figure 4-1 illustrates how the file layout is modified to make accessing data within a locality group efficient.

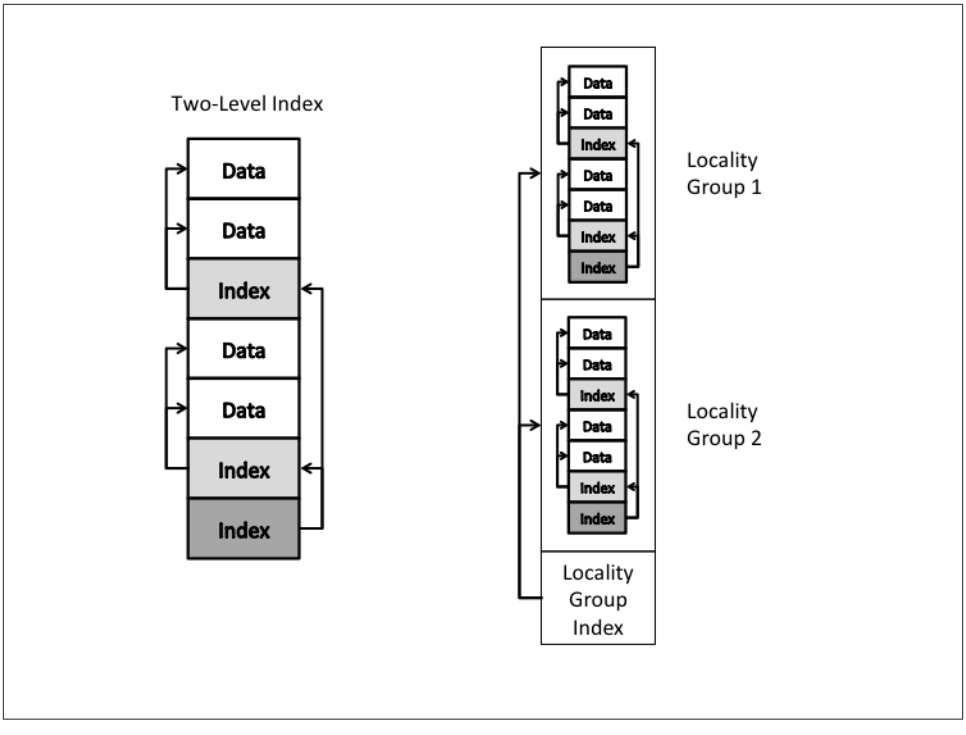

Figure 4-1. File layout with and without locality groups

Bloom Filters

Bloom filters may optionally be enabled per table. The properties governing the bloom filter configuration are of the form `table.bloom.*`.

If enabled, a bloom filter layer maintains a set of bloom filters for each file. Each key in a file is hashed with a number of hash functions, and the resulting hashes are represented as a bit vector. A bloom filter is the OR of the bit vectors for each key in a file.[2]

When looking up a key in the file, the bloom filter can be used to determine if the file may contain the key, or definitively does not contain the key. This check can be performed more efficiently than seeking for the key in the file, especially if the bloom filter has already been loaded into memory.

By default, the bloom filter layer hashes the row portion of the key, so it can be used to determine if a particular row appears in the file. However, it can be configured to hash the row and column family, or the row, column family, and column qualifier. This is

2. See *http://dl.acm.org/citation.cfm?id=362692&dl=ACM&coll=portal*, "Space/time trade-offs in hash coding with allowable errors," for general information about bloom filters.

controlled by setting `table.bloom.key.functor` to one of the three classes `org.apache.accumulo.core.file.keyfunctor.{RowFunctor,ColumnFamilyFunctor,ColumnQualifierFunctor}`.

Since seeks are specified for Accumulo in terms of a Range of keys, and not a specific key, the bloom filter layer will only provide improvements when the Range seeked only covers a single row in the case of the default `RowFunctor`, or a single row and column family in the case of the `ColumnFamilyFunctor`, or a single row, column family and column qualifier in the case of the `ColumnQualifierFunctor`.

Caching

Each tablet server process holds two `BlockCache` instances, one for data blocks and one for index blocks. These are caches in memory of individual compressed blocks of Accumulo RFiles. The size of each cache is specified in bytes with the `tserver.cache.data.size` and `tserver.cache.index.size` properties.

When the amount of data stored in a cache exceeds its specified maximum size, the cache will evict its least recently accessed blocks. The cache roughly reserves a third of its size for blocks that have been accessed a single time and two-thirds of its size for blocks that have been accessed more than once. Blocks never need to be invalidated in a cache, because Accumulo's RFiles are immutable.

The caches maintain counts including number of cache hits (block reads where the block was found in the cache) and total number of block reads, and Accumulo tracks this information and displays it on its monitor page.

Master

The master's main function is to monitor the status of tablet servers and tablets and to perform tablet assignment and load balancing as necessary.

To find tablets that need to be assigned, the master periodically scans the metadata table. Each tablet's state is determined based on the `current`, `future`, and `last` entries for that tablet in the metadata table. If a tablet is not in its desired state, for example if it is unassigned or is assigned to a dead tablet server, the master will assign the tablet by setting its `future` entry in the metadata table and by telling a selected tablet server that it should host the tablet. A load balancer is used to select a server for the tablet.

The master uses tablet server status information to balance the load of tablets across tablet servers. If there are active tablet servers that cannot be contacted, or there are unhosted tablets, or either the master or any tablet servers are in the process of shutting

down, the master will not perform load balancing. See "Load Balancer" on page 104 for more information on how tablets are balanced.

The status information collected from each tablet server includes the last contact time and some tablet server-wide information such as OS load, hold time (the amount of time the tablet server has been stuck waiting for minor compaction resources), number of lookups, cache hits, and write-ahead logs that are being sorted. Per-table information is also provided, including number of entries, entries in memory, number of tablets, number of online tables, ingest, query, and scan information, as well as number of minor and major compactions. If the master cannot obtain the status of a tablet server repeatedly, the master will request that the tablet server process halt.

The master is not a single point of failure because Accumulo can continue running without it. However, if the master is down for too long, the tablets may become unbalanced, and if tablet server processes go down while the master is down, their tablets will not be reassigned.

Multiple masters may be configured, and they will race to see which becomes the active master. The inactive masters will monitor the active master's zookeeper lock, waiting to obtain the lock in case the active master fails.

FATE

There are a number of administrative actions performed by the master on behalf of a user. These actions may involve multiple steps such as communicating with a tablet server, reading or writing data in Zookeeper, and reading or writing data to the metadata table. If the master fails without completing all the steps needed for a particular action, Accumulo and the client process could be left in an undesired state. For this reason, Accumulo introduced a fault-tolerant execution system, FATE, to ensure multi-step administrative operations are made atomically — either all the steps succeed, or the original state of the system is restored.

A FATE operation breaks down an administrative action into a set of repeatable, persisted operations, objects that implement the `Repo` interface. Each `Repo` must have the same end state when executed more than once, even if it has been partially executed previously. It also must be able to undo any changes it has made. On successful execution, a `Repo` returns the next `Repo` needed to continue the action. Before a `Repo` is executed, it is stored in Zookeeper along with a transaction ID associated with the FATE operation.

If the master goes down in the middle of performing a FATE operation, the next master that takes over will be able to continue the operation or roll it back based on the information recorded in Zookeeper. The actions currently managed as FATE operations include bulk import, compact range, cancel compaction, create table, clone table, delete table, import table, export table, rename table, and shutdown tablet server.

Load Balancer

The load balancer that the Accumulo master uses to assign tablets to tablet servers is controlled by the `master.tablet.balancer` property. The load balancer is responsible for finding assignments for tablets that are unassigned, at start time or any other time unassigned tablets are discovered (such as when a tablet server process goes down). It is also responsible for determining when the tablet load across tablet servers is out of balance, and determining which tablets should be moved from one server to another. This is called *migration* or reassignment. If the default `TableLoadBalancer` is used, a different balancer can be set for each table by changing the `table.balancer` property, which defaults to the `DefaultLoadBalancer`. Balancing each table independently is important, because otherwise a table's tablets might not be evenly distributed, even if each tablet server is hosting the same number of tablets.

The `DefaultLoadBalancer` attempts to assign a tablet to the last server that hosted it, if possible. If there is no last location, it will assign the tablet to a random server. When determining whether tablets need to be reassigned to keep the tablet servers evenly loaded, the `DefaultLoadBalancer` looks at the number of online tablets for each server. If there are tablet servers that have more and less than the average number of online tablets, this load balancer will move tablets from overloaded servers to underloaded servers. It picks a pair of tablet servers, starting with the most loaded and least loaded, and moves the smallest number of tablets necessary to bring one of the two servers to average load. When the `DefaultLoadBalancer` decides to move a tablet, it first decides which table the tablet should come from. If the higher loaded server has more tablets from any given table than the less loaded server, the balancer picks the most out of balance table. If none of the tables are out of balance, the balancer picks the busiest table (as defined by ingest rate plus query rate). Once a table is chosen, the balancer selects the most recently split tablet from that table. It repeats the tablet selection process until it selects the desired number of tablets for migration.

Garbage Collector

The garbage collector is a process that deletes files from HDFS when they are no longer used by Accumulo. This is a complex operation because a file may be used by more than one tablet. Early incarnations of the garbage collector compared the files in HDFS with the files listed in the metadata table. Prior to 1.6, the garbage collector can still be run in this way if Accumulo is not running and has been shut down cleanly. However, when Accumulo is running this method is not sufficient because files must be put in place before their metadata entries are inserted, which means the metadata table is expected to be slightly behind what exists on disk.

To address this issue, there is a section of the metadata table that records candidates for deletion. If a tablet doesn't need a file any more, it writes a deletion entry for that file to

the appropriate section of the metadata table. A tablet no longer needs a file if it has performed a compaction that rewrites that file's data into a new file. Major compaction, full major compaction, and merging minor compactions all result in files that may be deleted.

The garbage collector reads the deletion section of the metadata table to identify candidates for deletion. Then the garbage collector looks to see if the deletion candidates are still in use elsewhere, if they appear as file or scan entries for any tablets or if they are in a batch of files that are currently being bulk imported. After confirming the candidates that are no longer in use, the garbage collector removes those files from HDFS and removes the files' deletion entries from the metadata table.

Accumulo can run without the garbage collector. However, when the garbage collector is not running, Accumulo will not reclaim disk space used by files that are no longer needed. As with the master, multiple garbage collectors may be configured, and the inactive garbage collectors will monitor a zookeeper lock, waiting to obtain the lock in case the active garbage collector fails.

Monitor

The Accumulo monitor process provides a web service and UI for observing Accumulo's state. It connects to the master and the garbage collector processes via Thrift RPC.

The monitor is not essential for running Accumulo, but it is a useful tool for observing Accumulo's status and learning about any issues that Accumulo may be having. Currently Accumulo only works with a single monitor process.

Tracer

Accumulo includes a distributed tracing functionality based on the Google Dapper paper[3] that makes it possible to understand why some operations take longer than expected. This tracing functionality captures time measurements for different parts of the system across components and different threads.

One or more tracing processes are started to capture tracing information from Accumulo client and server processes using Apache Thrift RPC. The start-all script will launch one tracer process for each host defined the `ACCUMULO_CONF_DIR/tracers` file. Usually one tracer is sufficient to handle the tracing data for even a substantial Accumulo instance.

One trace is defined as a series of spans that have timing information for a specific operation. In addition to start and stop times, each span has a description and IDs for

3. *http://research.google.com/pubs/pub36356.html*

itself, its trace, and its parent span if one exists. Clients that want to use tracing must enable it for an application and then start and stop traces for individual operations (see ??? for an example of enabling tracing and retrieving the results). When tracing is started, or turned on, each span associated with a trace is sent to a tracer process chosen at random from the list of tracers that have registered in Zookeeper. Spans are received by each tracer asynchronously using Thrift RPC and inserted into the trace table in Accumulo. When the client operation being analyzed is complete, the client should turn tracing off, at which point any remaining spans from previous asynchronous calls will be inserted into the trace table. The complete trace information can then be retrieved from the trace table through the Accumulo shell or monitor page.

Accumulo server processes also use tracing to obtain and log information about their internal operations. Operations traced by Accumulo include every minor and major compaction, as well as one out of every 100 garbage collection rounds.

Client

Accumulo clients communicate with Accumulo server processes in a variety of ways. Information such as the instance ID, instance name, master locations, tablet server locations, and root tablet location are retrieved by the client directly from Zookeeper. Whenever the client retrieves information from Zookeeper, that information is cached in the `ZooCache`. If the same information is looked up again, the client will first check the cache. A Zookeeper `Watcher` is set on all Zookeeper data that is cached. If the data is changed in Zookeeper, it will be removed from the cache.

When the client obtains a connection to Accumulo, it reads the available tablet servers from Zookeeper and connects to a tablet server to authenticate. The `Connector` can then be used to retrieve various types of scanner or writer objects for reading from or writing to Accumulo. It can also be used to retrieve objects that can perform table operations, instance operations, and security operations.

Table Operations

Some table operations only require reading or writing table information in Zookeeper. Other operations require communicating with the master's Thrift interface or the Thrift interfaces of one or more tablet servers. Some operations use the client API to communicate with tablet servers and Zookeeper. A few operations also require accessing HDFS. Table 4-1 summarizes the types of communication necessary for each table operation.

Table 4-1. Table operations communication

Operations	Communications
listing tables checking to see if a table exists getting or setting configuration properties (including configuring iterators, constraints, and locality groups)	reading or writing information in Zookeeper
creating, deleting, cloning, renaming, or exporting a table flushing or compacting a table or a range of a table taking a table offline or online merging or deleting a range of a table	reading or writing information in Zookeeper, communicating with the master through a Thrift interface
adding a split point to a table getting the disk usage of a table testing class loading	reading or writing information in Zookeeper, communicating with one or more tablet servers through a Thrift interface
listing the split points for a table getting the maximum row of a table	communicating with Zookeeper and one or more tablet servers through the client API
bulk importing a directory importing a table	reading or writing information in Zookeeper, communicating with the master through a Thrift interface, accessing HDFS

Security Operations

All security operations involve reading or writing information in Zookeeper and communicating with a tablet server through its Thrift interface. These include create, authenticate, drop, or list users; change user password; get or change user authorizations; and get or change user permissions.

Instance Operations

Most instance operations involve communicating with Zookeeper and one or more tablet servers through a Thrift interface, with the exception of listing active tablet servers and setting or removing system configuration properties. The operations are detailed in Table 4-2.

Table 4-2. Instance operations communication

Instance Operations	Communications
getting a list of active tablet servers	reading or writing information in Zookeeper
setting or removing system configuration properties	reading or writing information in Zookeeper, communicating with the master through a Thrift interface
getting system or site configuration getting a list of active scans or compactions testing class loading pinging a tablet server	reading or writing information in Zookeeper, communicating with one or more tablet servers through a Thrift interface

Locating keys

When scanning a `Range` the tablet or tablets that overlap the `Range` must be located. When writing a `Mutation`, there will always be a single tablet containing the row ID of the `Mutation`, and that tablet must be located. The *location* of a tablet is the IP and port of the tablet server that hosts the tablet. Tablet locations are stored in the metadata table, which itself is split into multiple tablets. So, the metadata tablet containing information about the desired tablet must also be located. The locations of metadata tablets are stored in the root tablet. All the information about the metadata tablets is stored in a single tablet which is never split. The root tablet location is retrieved from Zookeeper, while all other tablet locations (metadata and not) are retrieved from the tablet servers hosting the root and metadata tablets. The client looks up tablet locations by conducting appropriate scans of the metadata table, but the tablet server's thrift API is used directly rather than going through the client scan API.

The location of the root tablet is cached in the `ZooCache`, while the locations of other tablets are cached separately. The separate tablet location cache is not invalidated when a tablet is moved to a different server, but if the client looks for a tablet by contacting a tablet server that is not hosting the tablet, the client will remove the location from the cache and retry.

Metadata table

The metadata table (along with the root tablet/table) is the authoritative source of information on the tablets and files for an Accumulo instance. The files for the metadata table are typically stored at a higher replication: the default is 5 replicas, rather than the default of 3 for other tables' files.

See Appendix C for details on the contents of the metadata table.

Uses of Zookeeper

Zookeeper is used heavily by Accumulo for determining liveness of processes, coordinating tasks, ensuring fault tolerance of administrative operations, and storing configuration that can be modified on the fly without restarting Accumulo.

See Appendix D for details on the data stored in Zookeeper.

CHAPTER 5
Administration

Accumulo is designed to run on a large number of servers and includes several features designed to make administrating and maintaining large clusters tractable for administrators.

In particular, dynamic load balancing and automatic recovery from common types of hardware failure help keep Accumulo healthy even in clusters of over a thousand machines, in which hardware failures are common.

Accumulo is a high-performance application and requires proper configuration and planning to operate effectively. In this chapter, we will start with considerations for planning a cluster, and discuss everything needed to setup and maintain a high-performing Accumulo instance.

Hardware Selection

Accumulo is designed to run on commodity-class servers. In general, using more expensive hardware will not dramatically improve Accumulo's performance or reliability and in some cases will work against Accumulo's availability features.

What do we mean by commodity class? Basically *commodity* here refers to servers that are widely available for a large number of uses - such as servers that can be used for serving web pages, handling email, etc. Using these general purpose machines has several advantages. First, when architecting the first MapReduce and BigTable clusters, Google calculated that they would get the most compute power per dollar using this hardware.

> Combining more than 15,000 commodity-class PCs with fault-tolerant software creates a solution that is more cost-effective than a comparable system built out of a smaller number of high-end servers.
>
> — Barroso et al.

Second, because these types of servers are so widely used, there are many vendors competing to sell them, which helps keep prices low and provides enough demand for hardware manufactures to keep improving performance.

Typical hardware for an Accumulo TabletServer is as follows:

CPU
: 2x 4 or 6 core CPUs

RAM
: 16-64 GB RAM

Disks
: 2-12x 1-3TB disks

Networking
: 1-2x 1Gigabit or 10Gigabit ethernet NICs

If these servers will also be hosting TaskTrackers for running MapReduce jobs, additional RAM and or CPU cores will come in handy.

Buying hardware with much more CPU cores and RAM, or *scaling up vertically*, may not result in higher performance as a single Tablet Server process is limited in some ways. Accumulo is designed to scale *horizontally*, meaning adding more servers rather than increasing the resources of each server.

Storage Devices

Unlike some databases, Accumulo is designed to keep most of the data managed on disk. As much data as will fit is cached into RAM as data is read from disk, but even reads that request data that is not cached in RAM are designed to be fast, as Accumulo minimizes disk seeks by keeping the data organized and reading fairly large chunks at a time.

Because Accumulo relies on HDFS to distribute and replicate blocks of data, it is recommended that storage consist primarily of inexpensive hard disk drives (HDDs), such as 1-3TB SATA 7200-15000 rpm drives, mounted separately (JBOD) rather than via RAID. HDFS essentially implements a RAID-1 data redundancy scheme, replicating entire disk blocks rather than using erasure coding, so employing RAID in addition to HDFS replication is unnecessary. The upsides of keeping full replicas are that there is no recovery time when a single hard drive is lost, and any of the replicas may be used for reading the data.

Storage Area Networks (SANs) are not well suited to providing storage for Accumulo as the scaling, independence, and failure characteristics are different than the shared-nothing unreliable hardware Accumulo and HDFS are designed for. SANs provide an

abstraction layer that defeats the attempts by HDFS to reason about data locality. If for some reason Accumulo must be run on a SAN it is likely that HDFS can be configured to keep only one replica of each block as the SAN will often provide its own replication.

Solid State Disks (SSDs) have presented an interesting development for databases in general as many databases require a high number of random access reads and writes. SSDs tend to provide a much higher number of random reads per second as there is no disk platter to rotate as with HDDs. However, as Accumulo is designed to reduce seeks by performing sequential disk accesses as much as possible, the advantages of SSDs over HDDs are not as pronounced with Accumulo as they would be with databases that perform a high number of seeks per user request. SSDs may work well in an environment where the ratio of reads is very high compared to the total stored data, such 25,000 random reads per second per terabyte of data.[1]

One interesting fact to keep in mind when considering databases to use with SSDs is that random writes can exacerbate an effect known as write amplification. Write amplification refers to the case when a single write from an application perspective may result in more physical writes as the SSD attempts to find or create an empty spot in which to write the data. Accumulo's write patterns, which are append-only and sequential, should result in a minimal level of write amplification.

Networking

Accumulo is a networked application and its storage layer, HDFS, is a networked file system. Clients connect directly to TabletServers to read and write data. TabletServers will try to read data from local disks when possible, avoiding reading data across the network, but will also often end up reading blocks of data from an HDFS DataNode over the network.

Having enough network bandwidth is fairly important to Accumulo. Even data read from disks local to a Tablet Server must still be transferred over the network to clients. For most clusters, servers with one or two 1 Gigabit ethernet cards are sufficient. If there is more than one NIC, they should be bonded in Linux to improve performance and availability. Currently Hadoop cannot utilize more than one NIC.[2] Many clusters' networks consist of a 10GB switch atop each rack of servers and a 10GB switch connecting a row of racks together.

Accumulo can be run on virtualized hardware, with a few caveats. HDFS makes some assumptions about the physical location of data in order to achieve good performance. If the virtual environment supports access to local disks then these assumptions can remain intact. If however the physical storage of data is abstracted away, onto remote

1. *http://www.fusionio.com/white-papers/fusion-io-and-sqrrl-make-accumulo-hypersonic/*
2. *https://issues.apache.org/jira/browse/HADOOP-8198*

media, then the efforts by HDFS to reduce network I/O will be pointless. This may or may not be a problem, depending on the total I/O available.

Another consideration is server responsiveness. Accumulo continually attempts to determine the liveness of servers. If server response time is highly variable due to unpredictable access to underlying physical resources, Accumulo's timeouts may need to be increased to avoid dropping servers that are alive but don't respond quickly enough. This increases the time that data may be unavailable before Accumulo recovers from a true server failure.

Finally there is the issue of independence and availability. The shared-nothing architecture Accumulo is built on relies on trying to reduce dependence between hardware in order to minimize the effect of an individual failure so that the overall system can continue functioning. In a virtual environment, a physical failure may affect more than one virtual server if those virtual servers happen to share any hardware, which may result in less availability than one might expect from separate machines. If these issues can be managed, Accumulo can be run successfully in a virtual environment.

Tips for Running in a Public Cloud Environment

In Amazon's Elastic Compute Cloud (EC2) environment for example, it is recommended that Tablet Server processes be run on instances with *ephemeral* storage, since that allows access to local disks. Some EC2 users recommend picking the largest instance type in a family, as supposedly this means that the virtual instance resources match the physical resources and that there will be only one virtual machine on a particular physical server, which may make access to the physical hardware less variable and services more responsive.

EBS volumes are recommended for storing the NameNode's data, preferably across several volumes in a RAID configuration, but not for primary HDFS storage.

Amazon EC2 uses Security Groups to restrict network access specific ports and hosts. See "Network Security" on page 147 for a list of ports that must be open.

Cluster Sizing

Several factors affect how much hardware is required for a particular use of Accumulo. To help in understanding this, we'll use a simple example. Let's imagine we are going to setup a Twitter clone that gets just as much traffic. Assume Twitter ingests 500,000,000 tweets a day[3] and each tweet is about 2,500 bytes on average.[4] That would about about 1.25TB of new data per day. Our incoming data might increase by 100% over the year

3. *https://blog.twitter.com/2013/new-tweets-per-second-record-and-how*
4. *http://codingrelic.geekhold.com/2011/05/dont-cross-tweetstreams.html*

so that by the end we're storing 2.5TB of new data per day. We need to store a year's worth of data online and available for queries on this system. Data older than 1 year old can be deleted as it is stored on another archival system (perhaps another HDFS cluster).

So we expect to have about 685 terabytes of data over the course of the next year.

Storage

Amount of original data

To start, knowing how much data you need to store can help determine the size of your cluster. Consider the amount of initial data that needs to be stored, and the rate of new data coming in. Accumulo uses Gzip compression by default. It also compresses sets of keys using a technique called *relative key encoding* (see ??? in the Internals chapter). These techniques can often result in a 3-4 : 1 compression ratio. This is convenient since the Hadoop Distributed File System replicates data by a factor of three by default. So even though HDFS will increase storage requirements by a factor of three, compression brings the amount of storage required for raw data closer to a 1:1 ratio of raw data ingested to data stored on disk.

This means to store a terabyte of data in Accumulo you will need at least a terabyte of disk space, but usually not 3 terabytes. But keep in mind this is before building any secondary index tables, which will require additional space.

For our 685TB we know we'll need at least that much raw storage.

Types of user requests and indexes required

If your application is designed to do lookups only one way, then raw data can be stored in a single table. In this case your table will require about 1x the size of the raw data. If additional lookup methods are required, a secondary index table will need to be built.

Typically, a single index table will suffice for any combination of equality expressions ANDed together. Some users will want to query *ranges* of values in multiple fields simultaneously, which can require additional index tables. Depending on your query requirements, knowing the number and type of additional index tables can help you plan to have enough storage.

If users need to be able to query *all* fields, the uncompressed size of the terms in the index would equal that of the original data, and that doesn't include the size of the unique identifiers that the index would use to point to the original data (see ??? on Secondary Indexes). Accumulo's compression and relative-key encoding are very efficient, so the disk storage needed for a full index might not actually exceed the original data size. However, if you find your indexes are larger than you want, you may wish to index only a subset of the fields.

For our example, let's assume that our desired index size is half the size of the original data, adding 50% to our storage needs, bringing the total to 1028TB.

Compaction

As new data is ingested into files in HFDS, periodically Accumulo compacts multiple files into a single file to make opening and reading files simpler and faster. During the compaction process, Tablet Servers copy several files into one new file. This requires additional storage and I/O resources. Just like with a MapReduce cluster, an Accumulo cluster will need some free space in which to operate. Let's estimate that 20% more storage is required for this purpose.

This ups our total to 1234TB.

Ingest Rate

To ensure you have enough raw storage to hold the initial data and the data added each day, to store the data in additional secondary indexes, and to perform compactions, the question of how many servers you need depends on the amount of storage per machine. Modern servers can support 12 or even more disks. At 3TB per disk, this means a single server can store 36TB or more. If we buy 2TB hard drives and can fit 12 drives into each server, we'll eventually need about 52 servers for our 1234TB of processed data.

But maxing out the storage per server may not be adequate to support the ingestion rate of the data. Ingestion not only depends on having enough raw storage, but also enough compute capacity and I/O to sort and manage the data. Let's calculate how many servers are needed to support the ingest rate we require.

For key-value pairs that are about 1k bytes in size, a single Tablet Server on typical hardware can support ingesting 30-100 thousand key-value pairs per second or more, depending on the number of CPU cores and drives and on how Accumulo is configured (see "Tablet Server Tuning" on page 160 for information on tuning Accumulo for better ingest rates). You will also have to adjust the number of ingest processes to achieve the best throughput for your system; a single ingest client may not be able to push Accumulo's tablet servers to their highest possible ingestion rate.

As the size of the cluster increases, the per-server rate will drop somewhat, simply because clients are forced to split their batches over more and more servers, which increases network overhead. Informal testing indicates that one can increase the aggregate write rate of a cluster by about 85% when the cluster hardware is doubled.

Let's estimate how many servers will be needed to support our initial rate of 500 million tweets per day. Each tweet has about 30 fields[5], but let's imagine that about half of those are usually empty. Just to ingest the data we will be writing 15 * 500 million = 7.5 billion

5. *https://dev.twitter.com/docs/platform-objects/tweets*

key-value pairs per day. If we index 10 fields per tweet, that's another 5 billion key-value pairs.

So we'll need to be able to ingest 12.5 billion key-value pairs per day if we write 25 key-value pairs per tweet. That's an average of 144,676 key-value pairs per second.

So we'll need 2 to 5 servers to get started. If the data arrives non-uniformly throughout the day, and peaks at say 2pm at 2x the average, we'd need 10 machines to handle peak load. To handle a peak as high as 140k tweets per second[31], we might need as many as 116 servers.

If we want to store 1234TB by the end of the year, we'll need to add a new server every 7 days. If we only buy 10 servers to start with, we'll have to buy more in less than 3 months.

If we buy half the cluster today, that's 26 machines. We may want to do this as in 6 months hardware may be slightly improved and we can get more bang for our buck.

Age off strategy

We can use Accumulo's Age-Off Iterator to automatically remove key-value pairs that are over a year old. Accumulo's files are immutable so to to do this we need to make sure we compact the tables periodically to create new files in which the old data is absent, and that we garbage-collect the old files.

A table can be compacted via the **compact** shell command. This will cause all files to be processed and iterators, such as the age-off iterator, to be applied in the creation of new files.

Pre-Installation

Operating Systems

Accumulo is regularly run and tested on several versions of Linux:

- RedHat Enterprise Linux
- CentOS 6
- Ubuntu 12 and above
- Development platforms include Linux and Mac OS X

Kernel tweaks

Swappiness TabletServers should be given enough operational memory to avoid swapping. Swapping is bad because it can cause a TabletServer to have to wait while the kernel

retrieves from disk some page of memory that was swapped out. This delay can interfere with Accumulo's ability to determine the liveness of the TabletServers and to keep all the data online.

To help avoid swapping, it is recommended that the Linux *swappiness* setting be set to 0 to instruct the kernel to not be eager at all when it comes to swapping pages from memory out to disk.

To set swappiness to 0 temporarily, do the following:

```
echo 0 > /proc/sys/vm/swappiness
```

And to make the setting persist across system reboots, do

```
echo "vm.swappiness = 0" >> /etc/sysctl.conf
```

If it is undesirable to set swappiness to 0 on a system, ensure that it is set to a low value, no more than 10.

Number of open files Accumulo needs to be able to create enough threads, network sockets, and file descriptors to respond to user requests. All of these require resources from the kernel which are limited by the number of open files allowed.

To set this edit */etc/security/limits.conf* and add the lines

```
accumulo        nofile  soft    65536
accumulo        nofile  hard    65536
```

Native Libraries

Because Java garbage collection can cause pauses that make it difficult to determine the liveliness of a process, Accumulo employs its own memory management for newly-written entries. This requires using binary libraries compiled for the specific architecture on which Accumulo is deployed. Binary libraries for Linux x86-64 are provided in the distributed files. If deploying to another architecture, building these libraries may be required.

If the binary libraries are not available, Accumulo will fall back on a pure-Java implementation, but at the cost of decreased performance and stability.

User accounts

Many distributions of Hadoop configure a *mapred* and *hdfs* user. Accumulo can be configured to use its own *accumulo* account.

If Accumulo will be installed from RPM or debian packages, the package scripts will create the *accumulo* user account.

Linux File System

Accumulo stores data in HDFS, which in turn stores blocks of data in the Linux filesystem. Popular Linux filesystems include Ext3, Ext4 and XFS.

Accumulo 1.4 and earlier versions required the ability to write to a local directory to store Write-Ahead Logs. A directory must be created for this purpose and must be writeable by the accumulo user. It can increase performance to put write-ahead logs on separate disks than disks storing HDFS data.

Accumulo 1.5 and above stores these files in HDFS so no additional directories need to be created.

System services

Accumulo relies on several system services to operate properly.

DNS - Domain Name Service
: Hadoop requires that domain names of machines be resolvable from domain to IP address and from IP address to domain name.

NTP - Network Time Protocol
: When an Accumulo table is configured to use a `TimeType` of milliseconds (`MILLIS`), Accumulo's TabletServers rely on system time for timestamping Mutations that do not otherwise have a timestamp provided. However, the TabletServers ensure that the timestamps they assign never decrease for any given tablet. With many machines in a cluster, some machines are bound to have clocks that are off. Running NTP daemons can help keep clocks closer in sync and avoid situations where assigned timestamps jump forward as tablets are migrated from one server to the next. Another solution is to use logical time (`TimeType.LOGICAL`), which uses a one-up counter as the timestamp instead of system time.

SSH - Secure SHell
: Accumulo ships with scripts that use SSH to start and stop processes on all machines in a cluster from a single node. This is not required, however, if other means of starting and stopping processes is used, such as init.d scripts.

To keep your Accumulo data secure, you also need ensure these services are secure just like any other system build on top of Hadoop. This topic continues to be discussed on the internet. For the scope of this book, we cover a few important details in "Security" on page 146.

Software Dependencies

Accumulo depends on several software packages. First, Accumulo is written in Java. Versions 1.6 and 1.7 have been tested and are known to work. The Sun/Oracle JDK is used more often in production although OpenJDK is often used for development.

Hadoop

Accumulo 1.5 binaries are built against Hadoop 1.0.4, but will likely work with other versions with little to no modification. To build against a different version see *Building from source*

Depending on the version of HDFS that is installed, different HDFS settings need to be configured in order to ensure Accumulo can flush Write-Ahead Logs to HDFS safely. The append or sync directive should be set to *true*.

Table 5-1. Table

Hadoop Version	Setting	Default
0.20.205	dfs.support.append	must be configured
0.23.x	dfs.support.append	defaults to true
1.0.x	dfs.support.append	must be configured
1.1.x	dfs.durable.sync	defaults to true
2.0.0-2.0.4	dfs.support.append	defaults to true

In addition, setting **dfs.datanode.synconclose** to true will help avoid data loss.

Apache Zookeeper

Apache Zookeeper is a distributed directory service designed to keep information completely replicated and synchronized across a small number of machines. Hence it is a highly available system for keeping small amounts of data. Accumulo uses Zookeeper to store configuration information and to coordinate actions across the cluster. See Appendix D for additional information on Accumulo's use of Zookeeper.

Zookeeper version 3.3.0 or later should be used.

The only configuration option of Zookeeper's that is regularly changed is the number of connections per client machine. The default is 10. Changing this is a matter of writing the line

```
maxClientCnxns=100
```

to the zoo.cfg file.

Installation

After Hadoop and Zookeeper are installed, Accumulo can be installed. Accumulo provides pre-compiled RPM and deb packages.

Tarball Distribution Install

Untar the tarball to the desired location and proceed to "Configuration" on page 122.

RPM-based Install

Accumulo provides RPM files for installation on RedHat compatible Linux distributions, including CentOS.

To install Accumulo from RPMs, download the following files from a mirror listed at *http://www.apache.org/dyn/closer.cgi/accumulo/filename*

For example, if downloading Accumulo version 1.5.x, the files would be

```
accumulo-1.5.x-bin.rpm
accumulo-1.5.x-native.rpm
```

As root, run

```
rpm -i accumulo-1.5.x-bin.rpm
rpm -i accumulo-1.5.x-native.rpm
```

This will install Accumulo in */opt/accumulo/accumulo-1.5.x/*

Administrators may want to create a symbolic link in */etc/accumulo/conf* that points to */opt/accumulo/accumulo-1.X.X/conf*.

At this point Accumulo is installed but not configured. Proceed to the configuration section.

Debian package-based Install

Accumulo provides debian packages for Debian and Ubuntu Linux distributions. The following steps have been tested on Ubuntu 13.10 (Saucy Salamander).

The java6-runtime package must be installed prior to installing Accumulo deb files.

For example, for Accumulo version 1.5.x:

Download the following files from a mirror listed at *http://www.apache.org/dyn/closer.cgi/accumulo/filename*

```
accumulo-1.5.x-bin.deb
accumulo-1.5.x-native.deb
```

Run as root:

```
dpkg -i accumulo-1.5.x-bin.deb
```

This will install binaries in */usr/lib/accumulo/* and configuration files in */etc/accumulo/conf/*.

To install native Accumulo libraries, make sure the following packages are already installed:

```
java6-sdk
g++
g++-multilib
```

Run as root:

```
dpkg -i accumulo-1.5.x-native.deb
```

If installing version 1.5.0, change line 29 of */var/lib/dpkg/info/accumulo-native.postinst* to

```
cd /usr/lib/accumulo/server/src/main/c++
```

and run as root:

```
dpkg-reconfigure accumulo-native
```

At this point Accumulo is installed but not configured. Proceed to the configuration section.

Building from Source

Accumulo is distributed under the Apache open source license, and as such, the source code can be downloaded and modified to suit a particular need. Any modifications to the source code, or to use different options than those that were used to create the binary distributions, will require building Accumulo from source.

There are several tools that make this process easier. Specifically, the Java SDK and Maven build tool should be installed. Java JDK version 1.6 or 1.7 and Maven version 3.0.4 will work.

To build from source, first download the source packages from a mirror listed on the Apache Accumulo website *http://accumulo.apache.org/downloads/*

Source code is found in the file ending in *src.tar.gz*. Once downloaded it can be unpacked via

```
tar -xzf accumulo-1.5.x-src.tar.gz
```

This will create a directory containing all the source files.

To Build a Tarball Distribution

To compile the source into binaries, change into this directory via

```
cd accumulo-1.5.x
```

and type

```
mvn package -P assemble
```

This will build a distribution compiled against Hadoop 1.0.4. The distribution should run with any supported version of Hadoop without being recompiled. However, if desired it is possible to compile with a different version by supplying appropriate options. To build for a different version of Hadoop 1, for example Hadoop 1.1.0, use the option

```
mvn -Dhadoop.version=1.1.0 package -P assemble
```

Hadoop 2 has a new API and so Accumulo has a special profile for it. To build for Hadoop 2, use the profile option as well as specifying the version of Hadoop, for example

```
mvn -Dhadoop.profile=2.0 -Dhadoop.version=0.23.5 package -P assemble
```

Once the build process is complete, there will be a tarfile distribution in *accumulo-1.5.x/assemble/target/accumulo-1.5.x-bin.tar.gz* similar to the binary distributions available from the Accumulo web site.

This tarball can be copied into the appropriate location and the instructions for installing from a Tarball Distribution can be followed to complete installation. Most installations will want to use the native libraries as well, described in the next section.

Building Native Libraries

Native libraries are written in C++ and must be built for a specific architecture. The binary distributions come with native libraries pre-built for the GNU Linux x86-64 architecture. If building from scratch or simply for a different platform, the native libraries can be built as follows.

Before building the native libraries, install the appropriate build tools. These include *make* and *g++*. Make sure these are installed.

For CentOS, g++ is installed via the *gcc-c++* package

The Java development kit packages should also be installed. On CentOS this requires *java-1.7.x-openjdk-devel* or *java-1.6.x-openjdk-devel* where *x* is the latest minor version number.

JAVA_HOME may need to be set appropriately, for example

```
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk.x86_64/
```

After downloading the source code as outlined in the previous section type:

```
cd server/src/main/c++
make
```

The native libraries will be found in the Accumulo install directory under *lib/native/map*. Accumulo will attempt to use these if present. If they are not, messages will appear in the logs warning that the native libraries could not be found.

To Build Debian Packages

One can also build Debian packages from source.

After running **mvn package -P assemble**, run:

```
mvn install -P deb
```

to build Debian packages. These will be deposited under *assemble/target* directory in the Accumulo installation directory.

Configuration

Configuring Accumulo is a process similar to configuring Hadoop, involving editing several files and distributing them across all the machines participating in an Accumulo cluster. In addition, after initialization and startup there are quite a few settings that are stored in Zookeeper that can be changed to effect changes across the cluster without restarting.

File permissions

Accumulo stores primary data in HDFS, including its Write-Ahead Logs. In HDFS the user that runs Accumulo processes must have the ability to read and write to files and directories under the directory specified as `instance.dfs.dir` in accumulo-site.xml, which defaults to */accumulo*. The user must also be able to create this directory, which means writing to the HDFS root directory if the default directory is unchanged. It is recommended to create a directory that the *accumulo* user has permission to write to (e.g. */user/accumulo*), and to make the `instance.dfs.dir` a subdirectory of this directory (e.g. */user/accumulo/accumulo*). Alternatively, the *accumulo* user account could be added to the HDFS supergroup.

The HDFS supergroup is set by adding a property in */etc/hadoop/conf/hdfs-site.xml* such as:

```
<property>
  <name>dfs.permissions.supergroup</name>
  <value>hdfs</value>
</property>
```

For example, if HDFS is configured with *hdfs* as the supergroup, the accumulo account can be added thus:

```
sudo usermod -a -G hdfs accumulo
```

The Hadoop Namenode may need to be restarted after this command to ensure the new group membership is reflected.

Accumulo needs to be able to write logs. If using the accumulo user to start Accumulo processes, these directories should be writable by the accumulo user.

For a Debian-based system these logs are in the */var/log* and */var/lib* directories.

```
#sudo mkdir /var/log/accumulo
#sudo mkdir /var/lib/accumulo
sudo chown -R accumulo:accumulo /var/log/accumulo
sudo chown -R accumulo:accumulo /var/lib/accumulo
```

For an RPM-based install, these are in */opt/accumulo/accumulo-1.X.X/logs*

```
sudo mkdir /opt/accumulo/accumulo-1.X.X/logs
sudo chown -R accumulo:accumulo /opt/accumulo/accumulo-1.X.X/logs
```

Configuration Files

Accumulo ships with some examples based on various memory configurations. To start with these example files, copy the files into Accumulo's conf directory (such as */etc/accumulo/conf/*)

```
cd /etc/accumulo/conf
sudo cp -r examples/3GB/native-standalone/* .
```

Accumulo needs to know how to talk to HDFS and Zookeeper in order to startup. Two files, accumulo-env.sh and accumulo-site.xml control most of Accumulo's startup settings. These two files should be copied to each machine that will run Accumulo processes and should be kept in sync if anything changes.

Heterogeneous clusters made of machines with differing hardware can have configuration files that differ in their memory settings, for example. However, properties such as the instance secret and hostnames of zookeeper must be the same.

accumulo-env.sh

Set the following system variables in the accumulo-env.sh file to the appropriate values for your system. If using debian packages this file will be mostly configured already. The only setting you may need to change is the HADOOP_CONF_DIR if using Hadoop 2.0 or later.

JAVA_HOME
: Should be the directory that contains *bin/java*

HADOOP_HOME
: Should contain hadoop jars and a *lib/* directory

HADOOP_CONF_DIR
: Should contain files core-site.xml, hdfs-site.xml, etc.

For Hadoop 2.0, comment out the line

```
# test -z "$HADOOP_CONF_DIR"       && export HADOOP_CONF_DIR="$HADOOP_PREFIX/conf"
```

and uncomment

```
test -z "$HADOOP_CONF_DIR"     && export HADOOP_CONF_DIR="$HADOOP_PREFIX/etc/hadoop"
```

ZOOKEEPER_HOME
: Should contain a zookeeper.jar file

Memory Settings
: Setting for various hardware configs. Also see the Tablet Server Tuning section in the Best Practices Chapter, "Tablet Server Tuning" on page 160.

4GB RAM

Tablet Server
: -Xmx1g -Xms1g -Xss160k

Master
: -Xmx1g -Xms1g

Monitor
: -Xmx1g -Xms256m

GC
: -Xmx256m -Xms256m

Logger
: -Xmx1g -Xms256m (1.4 and below only)

General Options
: -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75"

Other processes
: -Xmx1g -Xms256m"

16GB RAM
: .

64GB RAM
: .

96GB RAM
: .

accumulo-site.xml

This XML file contains properties, each with a name and a value. Hadoop uses similar files for its configuration. This file tells Accumulo processes which ZooKeeper instance

to use for configuration information and the values of various settings to use when starting up. The file should only be readable by the Accumulo user when it contains the instance secret. A separate site file without sensitive information can be created for client use.

If using debian packages or RPMs this file should be mostly configured already.

instance.zookeeper.host
: Write the list of zookeeper servers, separated by commas, such as

```
zookeeper1.mycluster.com:2181,zookeeper2.mycluster.com,zookeeper3.mycluster.com
```

instance.secret
: This is a shared secret among processes in a single Accumulo instance. It prevents processes that do not know the secret from joining the instance. It should be changed to a unique value for each installation.

general.classpaths
: If using Hadoop 2.0 or later, make sure all the HDFS and Hadoop common jars are added to the property *general.classpaths*. Some systems place HDFS 2.0 jars in */usr/lib/hadoop-hdfs/* and MapReduce jars in */usr/lib/hadoop-0.20-mapreduce* or */usr/lib/hadoop-mapreduce*.

Also see the Tablet Server Tuning section in the Best Practices Chapter, "Tablet Server Tuning" on page 160, for other properties that could be configured in *accumulo-site.xml*.

Cluster Definition

Accumulo processes can be assigned to hostnames by listing them in various files in Accumulo's *conf/* directory. These processes can be started started via provided start and stop scripts, or via init.d scripts.

slaves

The *conf/slaves* file is used by the *start-all.sh* script to start Accumulo processes on worker nodes - namely TabletServer and, in the case of Accumulo 1.4 and earlier, Logger processes. The hostnames of all machines that should run TabletServer processes should be listed in this file.

In cases where a TabletServer will run on the same machines that will host master processes, the hostname of those machines should be listed here to..

masters

The *conf/masters* file contains a list of the machines that will run a Master process. There should be at least one machine but more than 2 or 3 is generally not necessary. If more than one machine is listed here, the Master processes will choose an active Master, and

the others will serve as failover Masters in the case that the active Master fails. Unlike TabletServers, only a few machines need to run Master processes.

gc

This file contains a list of the machines that will run Garbage Collector processes. Like the master, only one will be active at any given time and any machines beyond the first will only take over should the active Garbage Collector fail.

monitor

This file contains a list of the machines that will run Monitor processes. Like the master, only one will be active at any given time and any machines beyond the first will only take over should the active Monitor fail.

tracer

This file contains a list of the machines that will run Tracer processes. Like the master, only one will be active at any given time and any machines beyond the first will only take over should the active Tracer fail.

Setting up Automatic Failover

Essential to running a large cluster is Accumulo's ability to tolerate certain types of failure. When certain processes fail, their workload is automatically reassigned to remaining worker nodes or backup processes on other machines. In general, setting up automatic failover for Accumulo processes is simply a matter of running an instance of a process on more than one server.

Tablet Servers

The Master process ensures that any tablets that were being served by a failed machine are reassigned to remaining TabletServers, who perform any recovery necessary by reading from write-ahead logs. For more on this process see Accumulo Internals.

Masters

A Master process must be running in order for Tablet Server failover to happen. If no Master is running, most client operations can proceed but if any Tablet Server fails while the master is down some tablets will be unavailable until a Master process is started.

To avoid a situation in which tablets may become unavailable, multiple Masters processes can be run to ensure that at least one Master is running at all times. In order to do this, the `$ACCUMULO_HOME`*/conf/masters* file should contain the hostnames of the machines on which Master processes are run.

The Master processes use Zookeeper to coordinate electing an active Master and to elect a new active Master in the event that the active Master fails.

Garbage Collectors

The Garbage Collector process is not critical to client operations, but must run to ensure aged-off, deleted, and redundant data is actually removed from HDFS.

Initialization

Before starting Accumulo for the first time, an Accumulo instance must be initialized. This can be done on a machine that can connect to both Zookeeper and HDFS, via the command:

```
accumulo init
```

This command should be run under the user account under which later Accumulo processes will be run, such as the *accumulo* user.

Be sure to verify that the init script is using the correct values for Zookeeper servers and the Hadoop Filesystem:

```
INFO: Hadoop Filesystem is hdfs://[your-namenode]:8020
INFO: Zookeeper server is [your-zookeeper]:2181
```

If an error occurs such as "java.io.IOException: No FileSystem for scheme: hdfs" check that the path to the Hadoop HDFS jars are included in the **general.classpaths** setting in the accumulo-site.xml file.

This script will create an entry in Zookeeper that will be used to coordinate all configuration information for this Accumulo instance. In addition, the script will create a directory called */accumulo* in HDFS, in which all table data will be stored.

If the accumulo user cannot write to the root directory of HDFS, an error with be thrown. Ensure that the accumulo user can create the */accumulo* directory in HDFS by adding the *accumulo* account to the HDFS supergroup, as described above in the *File Permissions* section. Alternatively, configure Accumulo to use a different directory that the accumulo user has permission to write to by changing the `instance.dfs.dir` property in accumulo-site.xml.

After initialization, there will be only two tables in Accumulo - the Metadata table, and the trace table. See the Internals chapter for more information.

To re-initialize

If for some reason you want to re-initalize an Accumulo cluster, the */accumulo* directory (or whichever directory is specified as `instance.dfs.dir`) in HDFS must be moved or deleted first. Deleting */accumulo* will erase all data in any existing Accumulo tables. This can be done via the command

```
hadoop fs -rmr /accumulo
```

or to simply move the directory

```
hadoop fs -mv /accumulo /[new path]
```

After this is done, the *accumulo init* script may be run again. The script will prompt for an instance name. If the instance name has ever been used before the script will prompt to delete the existing entry from Zookeeper. Answering Y will remove any information previously associated with that instance name, at which point initialization will proceed normally.

Multiple Instances

An Accumulo instance is a logical grouping of processes into one cooperative application. It is possible for multiple Accumulo instances to share a Zookeeper cluster and an HDFS cluster. The instances must have unique instance names and they must be configured to use different directories in HDFS. Additionally, if processes that belong to two different Accumulo instances are located on the same server, they must be configured to use different TCP ports to communicate. The port properties to configure depend on the type of process, and include master.port.client, tserver.port.client, gc.port.client, monitor.port.client, monitor.port.log4j, and trace.port.client.

Running

Starting

Before starting Accumulo, HDFS and Zookeeper must be running.

Accumulo can either be managed from a single control node using scripts provided in the bin directory, or using init.d scripts.

After starting, a running instance can be verified using the monitoring methods described below.

Via start-all.sh script

The *start-all.sh* script will SSH into all the machines listed in *masters*, *slaves*, *gc*, *monitor*, and *tracers* and start the associated processes. Password-less SSH is required to do this without having to type passwords for each machine.

Via init.d scripts

To use Linux's init.d system to start and stop Accumulo processes, first create a symbolic link from the *etc_initd_accumulo* script in Accumulo's *bin/* directory to */etc/init.d/accumulo*.

The environment variable `ACCUMULO_HOME` may need to be set in the script.

Accumulo can then be started by running

```
sudo service accumulo start
```

Depending on the *masters*, *slaves*, etc. files, processes will be started based on the files in which a machine's hostname appears.

Accumulo processes can be started at boot time and stopped at shutdown by adding the *accumulo* script to one or more runlevels.

Stopping

Accumulo clusters can be stopped gracefully, flushing all in-memory entries before exiting so that the next startup can proceed without having to recover any data from Write-Ahead logs. The Accumulo master orchestrates this shutdown.

Via stop-all.sh script

If using the *start-all.sh* and *stop-all.sh* scripts, Accumulo can be shutdown via running the *stop-all.sh* script. This script will attempt to talk to the master to orchestrate the shutdown. If the master is not running, the *stop-all.sh* script will hang. Hitting Ctrl-C will prompt the user to hit Ctrl-C again to cancel shutdown or else the script will forcefully kill all TabletServers and other Accumulo processes. If Accumulo is stopped this way, the next time the system is started TabletServers must recover any writes that were in memory at shutdown time from the Write-Ahead logs.

Usually this is not necessary. If the master is down, bringing up a new master process, perhaps on another machine, before attempting to shutdown will help reduce the need for recover on startup.

The accumulo admin command can also be used to stop the cluster

```
accumulo stopAll
```

If a `ClassNotFoundException` occurs when using the *stop-all.sh* script, ensure that the MapReduce and HDFS jars are correctly specified in the *general.classpaths* property in the *accumulo-site.xml* file.

Via init.d scripts

To stop all Accumulo processes on a particular node, run

```
sudo service accumulo stop
```

Stopping individual processes

Individual processes can also be stopped gracefully to avoid recovery from Write-Ahead logs.

To stop an individual TabletServer

```
accumulo admin stop [hostname]
```

To stop the master

```
accumulo admin stopMaster
```

Starting After a Crash

If TabletServer processes crash or exit due to a temporary network partition, they can simply be restarted and the Accumulo Master will start assigning tablets to them.

If a cluster was shutdown without allowing TabletServers to flush, e.g. if processes were all killed with a *kill -9* or if power was lost to the cluster, the cluster can be restarted and the process of recovery will begin.

For each tablet on a TabletServer that was killed before it could flush the entries in memory to HDFS, the Accumulo Master will coordinate a recovery process (see "Recovery" on page 98 for details).

During the recovery process, TabletServers will be assigned a tablet and will attempt to replay the mutations written to the Write-Ahead Log to recreate the state that was in the memory of the machines that were killed. This process can take from a few seconds to a few minutes depending on the size of the Write-Ahead Logs. The Master will display the status of this process on the Monitor page.

Clients can begin to read and write to tablets that aren't involved in recoveries while recoveries are taking place. As soon as a tablet's recovery is complete clients can again read and write key-value pairs to it.

Maintenance

Now that Accumulo is installed configured and running, clients can connect, create tables, and read and write data.

The primary administrative concerns at this point are monitoring system usage and health, and adding or removing machines from the cluster as a result of failure, or response to changes in usage. Accumulo is designed to operate on large clusters, and most failures do not require administrative attention. Otherwise keeping up would quickly become infeasible. Accumulo automatically detects certain types of machine failures and automatically recovers from them, reassigning work to remaining healthy machines in some cases.

For many maintenance operations it is not necessary to stop an Accumulo instance. This allows clients to continue to operate as machines fail, are added, or removed.

Monitoring

Accumulo provides several methods of monitoring system health and usage. These include the Monitor Web Service, logging, and JMX metrics.

Monitor Web Service

Accumulo provides a Monitor process that gathers information from a running cluster and presents it in one place for convenience. The Monitor makes it relatively simple to determine system health and performance.

(screenshot)

Overview The default view in the Monitor presents an overview of activity in the cluster. Particularly useful are the various graphs on this page. Administrators and developers can quickly gain an idea of how well the cluster is operating, spot issues, and analyze application performance.

The main section of this page shows two tables, followed by 10 graphs, all in two columns. The two tables show information from the Accumulo master and about the Zookeeper cluster.

An attempt is made to draw attention to any known problems with the cluster. This includes things such as the master being down, unassigned tablets, and log warnings and errors.

On the left are links to all the other views, described below.

Master Server View The Master Server View provides information on the instance overall, as well as a list of tables.

A legend is provided of the meaning of the columns in the first table:

- # Online Tablet Servers - Number of tablet servers currently available
- # Total Tablet Servers - The total number of tablet servers configured
- Last GC - The last time files were cleaned-up from HDFS
- Entries - The total number of key-value pairs in Accumulo
- Ingest - The number of key-value pairs inserted, per second (note that deleted records are “inserted” and will make the ingest rate increase in the near-term)
- Entries Read - The total number of key-value pairs read on the server side (not all may be returned because of filtering)
- Entries Returned - The total number of key-value pairs returned as a result of scans
- Hold Time - The maximum amount of time that ingest has been held across all servers due to a lack of memory to store the records

- Load - The one-minute load average on the computer that runs the monitor web server

This view is where active recoveries from any TabletServer failures are displayed.

Tablet Servers View The Tablet Servers View shows a list of active TabletServers and activity per server.

Here, an administrator can quickly see if tablets are evenly distributed across servers, how many key-value pairs (entries) per second are being ingested, and how many entries are being read for each server. This view is particularly useful for determining whether the entire cluster is being utilized evenly, or whether there are *hotspots*, i.e. a few servers which are handling the majority of the load.

Note that this view does not show which tablets belong to which tables. By default, the Accumulo master only tries to keep the number of tablets per server the same.

This means that a single table's tablets may not be evenly distributed - some servers may have more of a table's tablets than others. But the overall number of tablets should be close to the same on each server.

This will fluctuate, however, as new entries are written and as tablets split and are migrated from server to server. To change this behavior, other load balancers can be configured. See the section on Load Balancing.

Clicking on the hostname of a TabletServer will show a list of tablets currently hosted by that server. Statistics about server activities such as compactions and splits are also shown.

Server Activity View This view is a graphical representation of various aspects of server activity across the cluster.

Each circle or square in the grid represents a TabletServer. Various dimensions of each server can be displayed as color or motion, according to the selections from the drop down menus across the top of the view.

The intent of this view is to provide a high density of information at a quick glance. For example, one can choose to monitor load to see which servers may be overloaded - including by work from other processes running along side TabletServers. Or one may choose to show queries to try to highlight any hotspots and reveal weaknesses in table design.

Information for an individual server can be shown by hovering the mouse cursor over a particular square or circle.

Garbage Collector View The Garbage Collector View shows recent activity performed by the Garbage Collector. Administrators will want to check this view to make sure the Garbage Collector is running. Without it, the cluster could run out of disk space as files

are combined in compaction operations, creating new files and making old files obsolete.

Tables View The Tables View shows activity on a per-table basis.

Of particular interest here is the number of tablets per table. The aggregate ingest rate and number of concurrent queries will increase as the number of servers hosting a table's tablets are increase, up until all servers have at least one of the table's tablets.

To see which servers are hosting a table's tablets, administrators can click on the name of the table. A list of servers and the number of tablets from this particular tablet is shown.

Recent Traces View This view shows information about recent traces, which are a sample of operations that are timed to indicate performance.

For more on tracing, see the section on Tracing below.

Documentation View This view shows links to various types of documentation, including

- User Manual
- Administration
- Combiners
- Constraints
- Bulk Ingest
- Configuration
- Isolation
- Java API
- Locality Groups
- Timestamps
- Metrics
- Distributed Tracing

Recent Logs View This view collects log messages at the warn and error level from across the cluster. This can be very useful as clusters get larger and going out to each individual server more cumbersome.

Logs are listed in ascending time order, so the latest messages appear at the bottom. Messages that have been read and acknowledged can be dismissed by clicking *clear messages*.

JMX Metrics

Logging

Tracing

In a distributed system, diagnosing application performance and errors can be difficult. This is because operations may span several machines as clients call remote procedure calls on servers and servers call other servers.

In Accumulo, clients talk primarily to TabletServers which in turn talk to HDFS Data Node processes, all of which may be on different physical servers.

Tracing is a way of following an operation as it moves from server to server in order to get a holistic view of the timing of each stage of the operation. Clients can enable tracing explicitly, and Accumulo also traces some of its internal operations. If an Accumulo Tracer process is running, trace information is collected and stored in a *trace* table in Accumulo. The Shell has a special formatter configured to display this information.

See ??? for more information on Tracing.

Tracing Client Operations

Developers, particularly during debugging, can enable tracing to track down issues and validate design decisions.

```
String table = AccumuloConfiguration.getSystemConfiguration().get(Property.TRACE_TABLE);
Scanner scanner = shellState.connector.createScanner(table, auths);
scanner.setRange(new Range(new Text(Long.toHexString(scanTrace.traceId()))));
TraceDump.printTrace(scanner, new Printer() {
void print(String line) {
        System.out.println(line);
}
});
```

Tracing in the Shell

```
root@accumulo> table test
root@accumulo test> trace on
root@accumulo test> insert g g g g
root@accumulo test> scan
a b:c []    d
e f:g []    h
g g:g []    g
row f:q []    v
root@accumulo test> trace off
Waiting for trace information
Trace started at 2013/12/12 04:33:52.602
Time  Start  Service@Location       Name
14108+0      shell@ubuntu shell:root
```

```
   28+10122    shell@ubuntu close
    2+10122      shell@ubuntu binMutations
   28+10127      shell@ubuntu org.apache.accumulo.core.client.impl.TabletServerBatchWriter$Mutatio
   22+10127      shell@ubuntu org.apache.accumulo.core.client.impl.TabletServerBatchWriter$Mutatio
   22+10127        shell@ubuntu sendMutations
    2+10135          shell@ubuntu client:update
   10+10138          tserver@localhost update
    9+10138            tserver@localhost wal
    6+10140              tserver@localhost update
    6+10140                tserver@localhost wal
    2+10142                  tserver@localhost update
    2+10142                    tserver@localhost wal
    1+10147            tserver@localhost commit
    1+11998    tserver@localhost getTableConfiguration
    1+12003    tserver@localhost getTableConfiguration
    6+12012    shell@ubuntu scan
    6+12012      shell@ubuntu scan:location
    4+12013        tserver@localhost startScan
    4+12013          tserver@localhost tablet read ahead 7
    1+12020    tserver@localhost listLocalUsers
root@accumulo test>
```

Load Balancing

Tablets should always be balanced evenly across tablet servers. Which load balancer to use can be controlled on a per-table basis by changing the `table.balancer` property. Information about configuring load balancers and how the load balancers work can be found in "Load Balancer" on page 104.

Changing Settings

Accumulo allows administrators to configure a large number of settings that govern the behavior of Accumulo processes.

Besides the configuration controlled by the files in the Accumulo *conf/* directory, there are additional settings stored in Zookeeper. These settings allow changes to be made that in some cases can be reflected immediately across the cluster without having to restart processes.

The shell can be used to view current settings.

```
accumulo@hostname:~$ accumulo shell -u root
Password: ******

Shell - Apache Accumulo Interactive Shell
-
- version: 1.5.0
- instance name: accumulo
- instance id: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
-
- type 'help' for a list of available commands
```

```
-
root@accumulo> config
---------+---------------------------------------------+-----------------------------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+-----------------------------------------
default  | gc.cycle.delay ............................ | 5m
default  | gc.cycle.start ............................ | 30s
...
default  | instance.dfs.uri .......................... |
default  | instance.security.authenticator ........... | org.apache.accumulo.server.security.handle
default  | instance.security.authorizor .............. | org.apache.accumulo.server.security.handle
default  | instance.security.permissionHandler ....... | org.apache.accumulo.server.security.handle
default  | instance.zookeeper.host ................... | localhost:2181
...
default  | master.bulk.threadpool.size ............... | 5
default  | master.bulk.timeout ....................... | 5m
default  | master.fate.threadpool.size ............... | 4
default  | master.lease.recovery.interval ............ | 5s
default  | master.port.client ........................ | 9999
```

To view a specific setting, use the -f flag

```
root@accumulo> config -f table.compaction.major.everything.idle
---------+---------------------------------------------+-----------------------------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+-----------------------------------------
default  | table.compaction.major.everything.idle ... | 1h
system   |    @override .............................. | 2h
---------+---------------------------------------------+-----------------------------------------
root@accumulo>
```

One can also view settings that match a given prefix

```
root@accumulo> config -f tserver.bulk
---------+---------------------------------------------+-----------------------------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+-----------------------------------------
default  | tserver.bulk.assign.threads ............... | 1
default  | tserver.bulk.process.threads .............. | 1
default  | tserver.bulk.retry.max .................... | 5
default  | tserver.bulk.timeout ...................... | 5m
---------+---------------------------------------------+-----------------------------------------
root@accumulo>
```

The **config** shell command can also be used to change configuration settings.

In some cases this change will be reflected across the cluster immediately.

```
root@accumulo> config -s table.compaction.major.everything.idle=2h
```

Cluster changes

Accumulo is designed to withstand regular occurrences of machine failure, so being able to add and remove machines from a cluster configuration can be done easily. Ma-

chines can be added or removed from a running Accumulo instance without causing interruption to clients.

Adding new worker nodes

A major advantage of the shared-nothing, horizontal scale-out architecture of Accumulo and Hadoop is that adding more hardware resources can increase the aggregate performance of a cluster in a near-linear fashion. Rather than sharply diminishing returns, clusters can be increased in size quite a few times and still realize a significant increase in performance.

When additional storage, increased ingest rate, or increased concurrent query performance is required, worker nodes can be added. Empirically, some clusters have been shown to yield an increase of 85% when a cluster is doubled in size.

To add a new worker machine to a cluster, simply install the Hadoop and Accumulo software (most workers run Hadoop DataNode and TabletServer processes), and copy the configuration files. Upon starting, the DataNode process will contact the HDFS NameNode and let it know that the new machine is available for storing new HDFS data blocks. When the TabletServer starts it will register itself in Zookeeper. The Accumulo Master will notice that it is available and has no Tablets assigned, and will begin the process of migrating responsibility for hosting several Tablets to the new machine.

If the start-all.sh and stop-all.sh scripts are being used to start and stop the cluster, the new machine should be added to the *conf/slaves* file on the machine from which the start/stop scripts are run.

Removing worker nodes

Removing a live worker node from the cluster can be done as simply as turning off the machine. However, this will cause the cluster to have to recover the entries that were in the memory of the machine from Write-Ahead Logs. It can take a while before all the entries are again available for query. In addition, if the worker node is running a DataNode process, the replicas on that node must be re-replicated.

To avoid the recovery process, TabletServer processes can be shutdown gracefully in which they flush in-memory entries before exiting. This allows the machines that take over responsibility for its Tablets to begin hosting them immediately.

The accumulo admin command can be used to shutdown a specific TabletServer.

```
accumulo admin [hostname]
```

If using the start-all.sh and stop-all.sh scripts to control the cluster, the machine's hostname should be removed from the *conf/slaves* file.

If removing several machines at once, care must be taken to avoid taking all of the replicas of any particular HDFS data block offline at once. To avoid this HDFS provides

the ability to decommission a DataNode, which will cause HDFS to replicate all the blocks hosted by decommissioning nodes elsewhere. When this process is complete the nodes may be turned off.

Decommissioning HDFS DataNodes can be done by adding their hostnames to the *conf/excludes.xml* file and running

```
hadoop dfsadmin -refreshNodes
```

This will start the decommissioning process. The HDFS monitor page at *http://namenode:50070* can be used to monitor the process for completion. When the number of decommissioning nodes is 0 and the number of decommissioned nodes reaches the correct number, the DataNode processes can be stopped.

Adding new control nodes

Normally control nodes do not need to be added to the cluster - they don't participate in most client operations such as reading or writing data. From time to time a control node may fail and may need to be restored or replaced.

Removing control nodes

Control node processes do not maintain any persistent state and can simply be killed to remove them from the cluster. The hostnames should also be removed from masters, monitor, gc, and tracers files to avoid trying to start processes on removed machines the next time start scripts are run.

Failure Recovery

With clusters of up to thousands of relatively unreliable commodity-class machines, hardware failures are commonplace. Many types of failure, however, are automatically managed and clients and administrators do not need to take any action. The failure of a single Accumulo server, including the Master, will not cause any data to become unavailable, or even any interruption in client service.

If the Accumulo Master is down, clients can continue to communicate with TabletServers. However, if a TabletServer fails while there are no operational Masters, the tablets that were being hosted by the failed TabletServer will not be reassigned and hence will be unavailable for writes or reads by clients until the Master is started again.

It's important to know what types of failure Accumulo does not tolerate:

All Namenodes failing simultaneously Running a single Namenode used to be a big risk to Hadoop and Accumulo clusters since its failure meant some or all the mappings of filenames to data blocks in HDFS were lost or at least temporarily unavailable. Now HDFS can be configured to automatically failover from one Namenode to a hot standby that is kept in sync with the active Namenode.

Care must be taken to ensure the Namenodes don't share a common resource which, in the event of failure, would cause both Namenodes to go offline. There are limits to this, of course, as both Namenodes could lose power, being in the same datacenter. Accumulo is not designed to run over multiple geographically distributed datacenters.

All Zookeeper servers failing simultaneously Many of the same considerations for high-availability multiple Namenode apply to Zookeeper servers. At least one needs to be operating in order for Accumulo to function.

Power loss to the data center Accumulo is designed to run within a single data center, with low-latency networking between nodes. If power is lost to the data center, none of the machines in the Accumulo cluster will be operational.

Loss of all replicas of an HDFS data block HDFS replicates data blocks so that the loss of any one block will not cause an interruption in service. HDFS clients will simply find another remaining replica. If all replicas of a given block are unavailable, then Accumulo operations will fail.

To avoid the scenario in which a single hardware failure causes all replicas to become unavailable, HDFS provides the ability to specify the number of replicas, and also allows the specification of which machines live on which rack. The assumption here is that all machines within the same rack may share a common power supply or network switch, and so all replicas should not all be stored on machines in that rack, but rather, at least one should be stored on a machine in another rack. This capability is known as *rack-awareness*.

Network Partitions Accumulo does not "tolerate" network partitions in the sense that some other NoSQL databases do. In the event of a network partition, in which messages are lost between nodes in a cluster, some TabletServers will find themselves on a side of the partition that can continue to talk to the Zookeeper cluster, and others will not. Rather than allowing clients that can talk to TabletServers that are disconnected from Zookeeper to continue writing and reading data, Accumulo TabletServers will exit upon discovering that they can no longer communicate with Zookeeper.

Because all Accumulo TabletServers and often all clients are in the same data center, it's often the case that load balancers in front of clients can redirect requests to clients that can still talk to Zookeeper during the network partition.

Table Operations

Changing online status

A table can be brought *offline* - a state in which the table is no longer available for queries or writes and also no longer uses system resources.

All tables are online upon creation. The shell command **offline** can be used to take a table offline.

```
root@accumulo> offline mytable
```

The monitor's tables view will show the online/offline status of each table.

Taking tables offline can allow a cluster that is heavy on storage to store more tables than administrators would want to be available at any given time. This could be useful if tables need to be kept around for archival purposes, and can be brought online in the event that they need to be queried. Offline tables will not be affected by compactions.

To bring a table back online for queries and writes, use the **online** command:

```
root@accumulo> online mytable
```

Cloning

Accumulo tables can be *cloned* at very low cost in terms of system resources. This allows applications to operate on a version of a table without making permanent changes. As soon as a table is cloned, its clone can be written to and read from without affecting the original table.

Cloning works by copying just the configuration information for an original table to a new table. This configuration information is stored in Accumulo's METADATA table. Cloning tables is fast because Accumulo's files are all immutable, meaning they can be shared between several logical tables for reads. If new entries are written to a cloned table, they will be written to a separate set of files in HDFS.

Cloned and offline tables can be considered a consistent *snapshot* of a table as it existed at the time it was cloned. Tables that are offline can be exported for the purpose of backing up data or moving a table to another cluster.

To clone a table, use the **clonetable** command in the shell, specifying the name of the original table followed by the name of the table to create as a clone of the first. In this example we create a test table and clone it.

```
root@accumulo> createtable testtable
root@accumulo testtable> insert a b c d
root@accumulo testtable> scan
a b:c [] d
root@accumulo testtable> clonetable testtable testtableclone
root@accumulo testtable> table testtableclone
root@accumulo testtableclone> scan
a b:c [] d
```

In order to get a consistent view of a table before cloning, the **clonetable** command will first flush the original table. Flushing a table will ensure all key-value pairs that currently only live in the memory of TabletServers (and Write-Ahead Logs) are written to HDFS. Otherwise, these key-value pairs will not show up in the newly cloned table.

It is possible to clone a table without flushing first by using the -nf option when creating a clone. In this case, any entries that are still in the memory of TabletServers when the **clonetable** command is run will be excluded from the newly cloned table.

```
root@accumulo testtable> insert e f g h
root@accumulo testtable> scan
a b:c [] d
e f:g [] h
root@accumulo testtable> clonetable -nf testtable testtablenoflush
root@accumulo testtable> table testtablenoflush
root@accumulo testtablenoflush> scan
a b:c [] d
root@accumulo testtablenoflush>
```

To export a snapshot of a table elsewhere, for backup or other purposes, see the next section *Backup and Exporting Data*

Altering Cloned Table Properties

Table properties can be excluded or added to the cloned table before bringing it online. For example, if the original table has an age-off iterator configured that is designed to remove data older than a given date, we may want to exclude that iterator from our cloned table, if the intent of the cloned table is to serve as an archive of the original for some time beyond the age-off date.

A possible reason to add a property to the cloned table before it is brought online may include testing out an experimental Iterator or to simply to modify the behavior of the cloned table.

For example, to modify the cloned table to keep the latest 3 versions of each key-value pair:

```
root@accumulo testtable> clonetable testtable testtableclone -s table.iterator.majc.vers.opt.maxVe
root@accumulo testtable> config -t testtableclone -f table.iterator
---------+---------------------------------------------+---------------------------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+---------------------------------------
table    | table.iterator.majc.vers ................. | 20,org.apache.accumulo.core.iterators.use
table    | table.iterator.majc.vers.opt.maxVersions .. | 3
table    | table.iterator.minc.vers ................. | 20,org.apache.accumulo.core.iterators.use
table    | table.iterator.minc.vers.opt.maxVersions .. | 3
table    | table.iterator.scan.vers ................. | 20,org.apache.accumulo.core.iterators.use
table    | table.iterator.scan.vers.opt.maxVersions .. | 3
---------+---------------------------------------------+---------------------------------------
```

Cloning for MapReduce

One use case is to create a copy of a table whose files can be used as the input to a MapReduce job. Tables that are online may receive new inserts of data, and Accumulo may perform a compaction in which the set of files that comprise a table changes. Each

file Accumulo stores in HDFS is immutable once closed, but the set of files associated with a table at any given time can change. To avoid a situation in which a table's set of files changes, a table can be cloned, then taken offline.

```
root@accumulo> clonetable mytable mytablecopy
root@accumulo> tables
!METADATA
mytable
mytablecopy
trace
root@accumulo> offline mytablecopy
```

Once the table is offline, its set of files will not change and a MapReduce job can be run over them.

See the section MapReduce of the Writing Applications chapter for details on how to configure a MapReduce job to run over files of an offline table.

Backup and Exporting Data

Accumulo stores all its data in HDFS which features full-data replication to prevent data loss in the event that one or more machines fail. HDFS replication is often sufficient for maintaining data availability for applications, however there are a few remaining reasons to backup data to places other than the HDFS.

The most obvious is that an Accumulo instance is designed to run within a single data center. Data centers can suffer catastrophic failure, such as in the event of a widespread power failure or natural disaster. The ability to backup data to data centers in other geographic locations is important if data is to survive one of these catastrophic failures.

Another reason to create a backup may be in order to create a copy of the data to be used for purposes other than those of the original cluster. Servers are powerful enough these days to support mixed-workloads on a single cluster, and many Accumulo clusters serve double-duty, managing data for low-latency requests for applications as well as performing in-depth or historical bulk analysis via MapReduce. Nevertheless, the ability to copy data to another cluster can be useful for supporting a wide variety of workloads on the same data set.

Creating a backup of Accumulo data, like other systems, involves copying the data to be backed up to another storage medium and restoring that data at some future point to recover from a disaster, or simply loading into another Accumulo instance. To backup a table, administrators should first flush, clone, then offline the newly cloned table and leave it offline while the export is taking place. The export command will write information about a table's files to a directory in HDFS. This information can be used to copy table files to another cluster or other storage medium.

```
root@accumulo> clonetable mytable mytable_backup1
root@accumulo> offline mytable_backup1
root@accumulo> exporttable -t mytable_backup1 /table_backups/mytable
```

To use Hadoop's distcp (distributed copy) utility to move the files, reference the distcp.txt file thus:

```
$ hadoop distcp -f hdfs://namenode/backups/mytable/distcp.txt hdfs://othernamenode/tmp/mytable_bac
```

Hadoop Distributed Copy

Hadoop provides a mechanism for moving large amounts of data between HDFS instances, or other distributed file systems called *Distributed Copy* or *distcp* after the command name for short. Rather than pulling data out of HDFS to a single machine and then uploading it to another HDFS cluster, Distributed Copy sets up a MapReduce job that will stream files from where they are stored in HDFS DataNodes to DataNodes in another HDFS instance. This operation will attempt to use as much network bandwidth as is available and minimizes the chance for bottlenecks to occur while copying the data.

Of course HDFS can also be used to copy data within the same HDFS instance if that is required.

Importing an Exported Table

To import a table that has been exported via the **exporttable** command, Accumulo provides an **importtable** command. The importtable command tables two parameters - the name of the table to create into which the files will be imported, and the directory in HDFS where exported information is stored.

```
root@accumulo> importtable importedtable /tmp/mytable_backup1
```

If the accumulo user doesn't have permission to read the files in the import directory, this command will fail as it moves files from the import directory.

The imported table will have the same split points and configuration information as the table that was exported.

Bulk Loading Files from a MapReduce job

MapReduce jobs can be used to write data into files that Accumulo understands. (See the Writing Applications chapter for details on how this is done.) These files can then be *bulk loaded* into Accumulo without having to write each key-value pair to TabletServers. The advantage of bulk import is that users can take advantage of the efficiency of MapReduce to sort data the way Accumulo would if key-value pairs were written to its TabletServers one at a time, or one batch at a time as via the BatchWriter. Another advantage of using MapReduce to create files for bulk import is that a consistent set of files can be created without the chance of creating duplicates in the event that one or

more machines fail during the MapReduce job. See the Best Practices section on scenarios in which bulk importing files can help avoid creating duplicate rows in Accumulo tables.

To import files created from a MapReduce job, use the **importdirectory** command. This command will import files into the current table set in the shell, so administrators should first set the current table using the **table** command before running the **importdirectory** command.

importdirectory expects three parameters, the name of the HDFS directory that contains files to be imported, the name of a directory to which to store any files that fail to import cleanly, and finally whether to set the timestamp of imported key-value pairs.

Bulk Loading and Timestamps

An important consideration for bulk loading files created via MapReduce is whether to accept the timestamps of key-value pairs in the files as valid, or to set new timestamps for all imported key-value pairs based on the time the files are imported.

If there is a chance that one or more machines that participated in the MapReduce job used to create these files has a clock that is set at some time in the future, and if the MapReduce simply used system time to timestamp key-value pairs, then those key-value pairs will create problems when they are bulk imported. Specifically, if a key-value pair has a timestamp that is set in the future, then inserts and deletes that use the current time as the timestamp will appear to not have any effect, as the Versioning iterator uses timestamps to determine the order of operations. A key-value pair with a timestamp in the future will appear to be the latest mutation to a row, until time catches up with whatever future timestamp the key-value pair has.

If it is not reasonable to assume that the timestamps of key-value pairs created via MapReduce are valid, that is that there is a chance that some timestamps are set in the future, administrators should specify the last parameter of the **importdirectory** command as *true*.

Using Major Compaction to apply changes

Major compactions occur regularly in the background in order to consolidate multiple files and remove deleted data.

Other changes can be applied via compactions. For example, if the compression algorithm used for a table is changed, the existing files will need to be decompressed using the old algorithm and new files written using the new algorithm. This process can be carried out via compactions.

In this example we'll turn off compression for our test table. The **du** command shows how many bytes our table is using.

```
root@accumulo testtable> du
                    239 [testtable]
root@accumulo testtable> config -t testtable -s table.file.compress.type=none
root@accumulo testtable> config -t test -f table.file.compress.type
---------+--------------------------------------------+----------
SCOPE    | NAME                                       | VALUE
---------+--------------------------------------------+----------
default  | table.file.compress.type .................. | gz
table    |    @override .............................. | none
---------+--------------------------------------------+----------

root@accumulo testtable> du
                    239 [testtable]
```

Note that our table has not yet changed in size. Any new files created or merged in a merging compaction will not be compressed. To change the files already on disk we can schedule a compaction manually.

```
root@accumulo testtable> compact -t testtable
root@accumulo testtable> du
                    270 [testtable]
```

The -w option can be given to cause the shell wait for the compaction to complete before returning.

If the -t option is not specified, Accumulo will compact the current table in the shell.

If a compaction has been scheduled but has not yet begun and a user wishes to cancel it, the --cancel option can be specified to cancel the compaction

The --noflush option can be used to avoid flushing entries still in memory on TabletServers to disk before compacting.

Tables that are offline will not be affected by compactions.

Compacting Ranges

It is possible to compact only a portion of a table by specifying a range of rows to the **compact** command.

The -b or --begin-row option and -e or --end-row options will cause the compaction to only affect the key-value pairs between the begin and end rows, inclusively.

This can be used to alter sections of a table. For example, if a table's rows are based on time, we may wish to allow for more versions of each key-value pair for more recent data, and only keep one version of each key-value pair around for the oldest data.

To affect this change, we can temporarily change the VersioningIterator's options for our table, and schedule a compaction for a range comprising our oldest data. We can then restore the original configuration.

Accumulo 1.6 allows administrators to specify specific iterators and options to use for a particular compaction.

Security

Accumulo works to protect data from unauthorized access. Like any security measures the features Accumulo provides must be coordinated with other system security measures in order to achieve the intended protection.

There are three requirements for Accumulo to guarantee that no data is exposed in an unauthorized manner:

1) Data is properly labeled when inserted by Accumulo clients 2) Accumulo clients present the proper authorization tokens when reading data 3) Supporting systems listed in "System services" on page 117 and supporting software are secured.

Data Labels and Accumulo Clients

Accumulo will authenticate a user according to their credentials (such as a password), and authorize that user to read data according to the security labels present within that data and the authorizations granted to the user. All other means of accessing Accumulo table data must be restricted.

Support Software Security

Since Accumulo stored data in HDFS, access to these files must be restricted. This includes access to both the RFiles, which store long term data, and Accumulo's write-ahead logs, which store recently written data. Accumulo should be the only application allowed to access these files in HDFS.

Similarly, HDFS stores blocks of files in an underlying Linux file system. User who have access to blocks of HDFS data stored in the Linux filesystem would also bypass data-level protections. Access to the file directories on which HDFS data is stored should be limited to the HDFS daemon user.

Unnecessary services should be turned off.

The accumulo-site.xml file should not be readable except by the *accumulo* user, as it contains the *instance-secret* and the trace user's password. A separate conf directory with files readable by other users can be created for client use, with an accumulo-site.xml file that does not contain those two properties.

Network Security

IPTables or other firewall implementations can be used to help restrict access to TCP ports.

Accumulo uses the following port numbers by default. These should be reachable by Accumulo clients as well as by each other.

Table 5-2. Table Accumulo Network Ports

Setting Name	port number	Purpose
master.port.client	9999	The port used for handling client connections on the master
tserver.port.client	9997	The port used for handling client connections on the tablet servers
gc.port.client	50091	The listening port for the garbage collector's monitor service
monitor.port.client	50095	The listening port for the monitor's http service
monitor.port.log4j	4560	The listening port for the monitor's log4j logging collection.
trace.port.client	12234	The listening port for the trace server

Accumulo TabletServers must be able to communicate with HDFS DataNodes and the Namenode.

Only trusted client applications should be allowed to connect to Zookeeper, and Accumulo TabletServers.

Kerberized Hadoop

To use Accumulo with a kerberized HDFS instance, an Accumulo principal must be created.

```
kadmin.local -q "addprinc -randkey accumulo/[hostname]"
```

Principals can then be exported to a keytab file.

Application Permissions

Accumulo has the concept of a *User* permission, but more often these are associated with a particular application that may provide access to multiple users. Accumulo clients can do their own authentication of multiple users and also lookup any associated authorization tokens, which they then faithfully pass to Accumulo TabletServers when doing scans.

Before any user can read any data however, a *User* account must be created, authorization tokens assigned and access to tables granted. Administrators can work with application developers to determine the right level of access for a *User* account and how to determine the set of authorization tokens to grant to the account.

To create a *User* account in the shell, run the **createuser** command

```
root@accumulo> createuser myapp
Enter new password for 'myapp': *****
Please confirm new password for 'myapp': *****
```

To allow this user to read a particular table, run

```
root@accumulo> grant Table.READ -t mytable -u myapp
root@accumulo> System permissions:

Table permissions (!METADATA): Table.READ
Table permissions (mytable): Table.READ
```

To grant authorizations to a User, run

```
root@accumulo> setauths -u myapp -s myauth
root@accumulo> getauths -u myapp
myauth
```

To see what permissions a user has, run

```
root@accumulo> userpermissions -u myapp
```

Once this has been done an Accumulo client *myapp* can connect to Accumulo, passing in the password specified, and perform scans against the table *mytable* and pass in the authorization token *myauth*. If a client tries to read from another table, or tries to write to *mytable*, or tries to pass in a different authorization token, it will receive an Authorization exception.

A list of available permissions can be seen via the **systempermissions** and **tablepermissions** commands.

Troubleshooting

If Accumulo clients are experiencing issues - errors, timeouts, etc. there are several things that should be checked as part of the troubleshooting process.

Ensure Processes are Running

First, if any of the services on which Accumulo depends is not healthy, Accumulo will experience issues. Make sure HDFS is running and healthy. The HDFS monitor page at namenode host:50070 will show the status of HDFS. If any blocks are missing Accumulo will be unable to serve the data from the files those blocks belong to. If DataNode processes have crashed, it may be possible to restart them and for their blocks to become available again.

Zookeeper should also be running and healthy. Administrators can check this by telneting to a Zookeeper process at port 2181 by default and typing the word *ruok*, short for *are you ok?* The server should respond with *imok* (*I am ok*) and close the connection. If Zookeeper is down it should be restarted before attempting to start any Accumulo processes.

Finally, Accumulo processes should be checked to make sure they are running and operating properly. The Accumulo monitor page will try to highlight problems by literally highlighting issues in red. Having 0 running TabletServers, if any tablets are unassigned, if the Accumulo Master is unreachable the monitor page will show red boxes behind text.

Check Log Messages

The Accumulo monitor also gathers error log messages from TabletServers and displays them in one place for convenience. Checking for these can explain issues.

If the monitor is not showing any errors or if it is down, logs are still written to local files on each machine running Accumulo processes.

Some common issues and error messages are listed here:

WARN : Thread "shell" stuck on IO to 127.0.0.1:9999:9999

Understand Network Partitions

If for some reason a TabletServer is unable to reach Zookeeper, a condition known as a network partition, within a period of time it will lose its TabletServer lock. At this point the Accumulo Master will attempt to obtain the TabletServers lock. If successful, the TabletServer is no longer considered to be part of the cluster and the Master will reassign its Tablets to remaining healthy servers. The TabletServer that lost its lock will then exit to prevent clients from sending any more writes to it.

This procedure is designed to guarantee that each Tablet is hosted by only one TabletServer at a time. This also means that if Zookeeper or TabletServers are not responsive enough to network requests, TabletServers processes may terminate as they can't distinguish between arbitrarily delayed requests and a network partition. If TabletServers are exiting regularly due to a loss of a Zookeeper lock, they or Zookeeper may not have sufficient resources.

Causes of this can include swapping to disk if available memory is insufficient, Java garbage collector pauses when not using native libraries, or simply insufficient hardware for the application.

Inspecting RFiles

Invariably, during the development phase a situation can arise wherein key-value pairs are being labeled incorrectly. Accumulo is designed to take security labels very seriously. As such, it is not possible to simply turn off the Iterator responsible for examining security labels and filtering out key-value pairs whose label logic is not satisfied by the querying user's credentials - even if logged in as *root*.

What this means for key-value pairs that have incorrect labels is that they simply won't show up in any scan. A symptom that indicates that a table may contain entries with incorrect labels is if a scan over an entire table yields no results, when the monitor page indicates that there is, in fact, data in there, and when all the known granted labels are being used to scan the table (as is the default mode of scanning in the shell).

If this appears to be happening, a table can be configured to throw Exceptions if it is asked to store a key-value pair with a label that can't be satisfied with the writing user's credentials. This way, an incorrect security label shows up before it ever gets written to a table.

This constraint can be added in the shell thus:

```
root@accumulo> constraint -t tableName -a org.apache.accumulo.core.security.VisibilityConstraint
```

If for some reason this and other troubleshooting methods of fixing labels have failed, or other parts of key-value pairs need to be inspected, an administrator with access to read files from HDFS can inspect Accumulo's underlying RFiles to see what the key-value pairs actually are. Reading RFiles directly should not be done lightly, as there are no checks in place to ensure the user has the authorization to see all of the data stored in an Accumulo table. This is the exact reason that access to HDFS must be restricted.

Administrators can dump the contents of an RFile using the following procedure.

Determine the underlying table ID for the table containing suspected incorrect labels

This will allow us to locate the RFiles for the table of interest in HDFS. In the shell, type:

```
    root@accumulo> tables -l
    !METADATA       =>         !0
    baseball_stats  =>         17
wikipedia           =>         18
wikipedia_index     =>         15
    trace           =>          1
```

List the files in HDFS for one of the tablets in the table of interest In our example, we'll examine a file in the *default_tablet* of the table *wikipedia_index* with id *15*.

Exit the shell and run the *hadoop fs* command:

```
$ hadoop fs -ls /accumulo/tables/15/default_tablet

Found 7 items
-rw-r--r--   1 accumulo supergroup   24215993 2013-12-11 05:14 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup   18290804 2013-12-11 05:21 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup       4515 2013-12-11 09:01 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup     673682 2013-12-11 09:20 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup    1201112 2013-12-11 09:47 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup    5282634 2013-12-11 11:17 /accumulo/tables/15/default_tablet/
-rw-r--r--   1 accumulo supergroup    5631122 2013-12-11 11:24 /accumulo/tables/15/default_tablet/
```

To view simple details of the file, use the PrintInfo class with only the filename as an argument. This will show statistics from the file as well as the first and last key. These keys may show an example of one of the incorrect labels.

```
$ accumulo rfile-info /accumulo/tables/15/default_tablet/F0000hy3.rf

Locality group         : <DEFAULT>
        Start block          : 0
        Num   blocks         : 665
        Index level 0        : 83,095 bytes  1 blocks
        First key            : 0000313867566100000000205001 :fields [public] 0 false
        Last key             : 0000313867568500000000205001 :fields [prvate] 0 false
        Num entries          : 31,218
        Column families      : []

Meta block     : BCFile.index
      Raw size             : 4 bytes
      Compressed size      : 12 bytes
      Compression type     : gz

Meta block     : RFile.index
      Raw size             : 83,257 bytes
      Compressed size      : 12,992 bytes
      Compression type     : gz
```

In this case, the security token *prvate* probably represents a misspelling of the word *private*. If we scanned the table using the token *private* we would not see the keys with the label *prvate*.

To dump key-value pairs from the file, use the **-d** option:

```
$ accumulo rfile-info -d /accumulo/tables/15/default_tablet/F0000hy3.rf
```

The utility will print out the statistics as before, followed by string representations of the key-value pairs in this file.

CHAPTER 6
Best Practices

Accumulo provides developers with a high degree of control over data layout and options for managing data. For these reasons there is a wide range of choices when designing tables to support applications and selecting how and when to apply logic.

In this section we discuss common patterns that developers may want to consider when tackling design, performance, or maintenance issues.

Performance

Performance is a large topic and the previous chapters include information to help users reason about the performance impact of decisions where possible. In particular, an understanding of hardware performance and how Accumulo uses hardware is essential. The section ??? on Understanding Accumulo Performance in the Writing Applications chapter should be understood before reading this section. Here we discuss ways to pinpoint common performance issues and address them.

Measuring Performance

The Accumulo monitor is a convenient way to see metrics of cluster performance and diagnose problems. Knowing what a good theoretical estimate of cluster performance should be before starting can help verify the existence of a performance problem.

The monitor page is something you should get familiar with. Understanding it is important to not only measuring performance, but also monitoring day to day operation. The monitor page shows several items of interest to performance analyses:

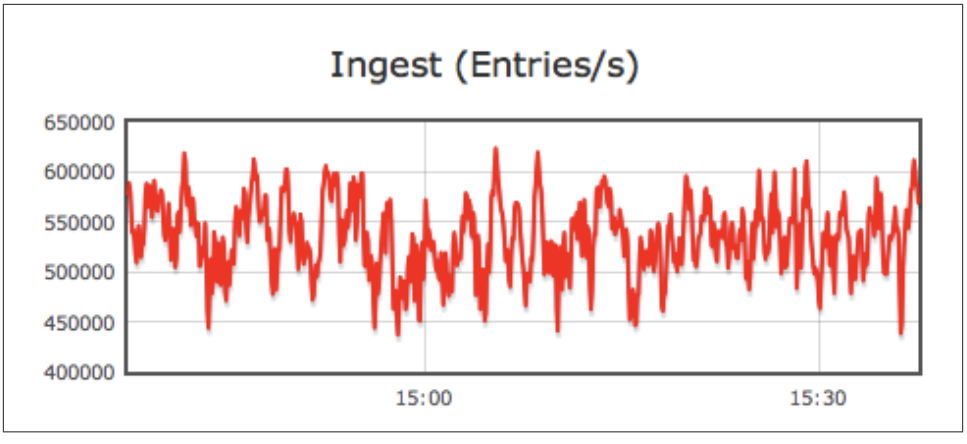

Figure 6-1. Aggregate ingest rate in key-value pairs per second

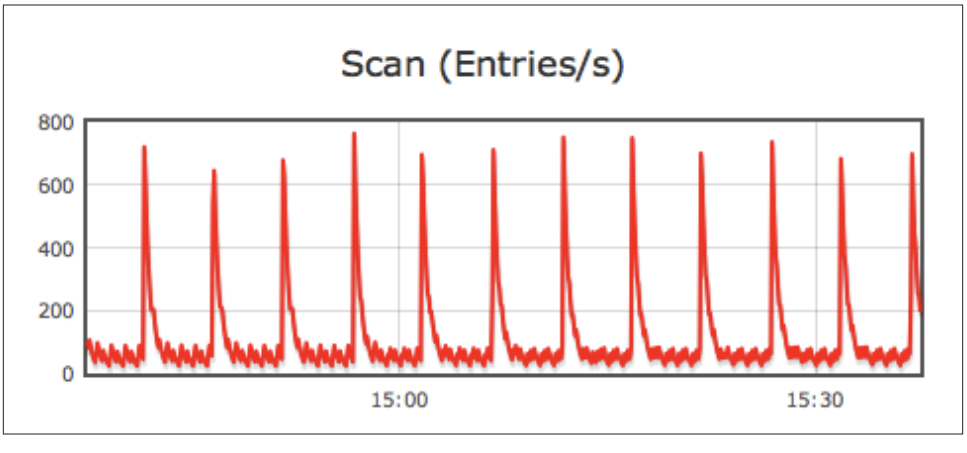

Figure 6-2. Aggregate scan rate in key-value pairs per second

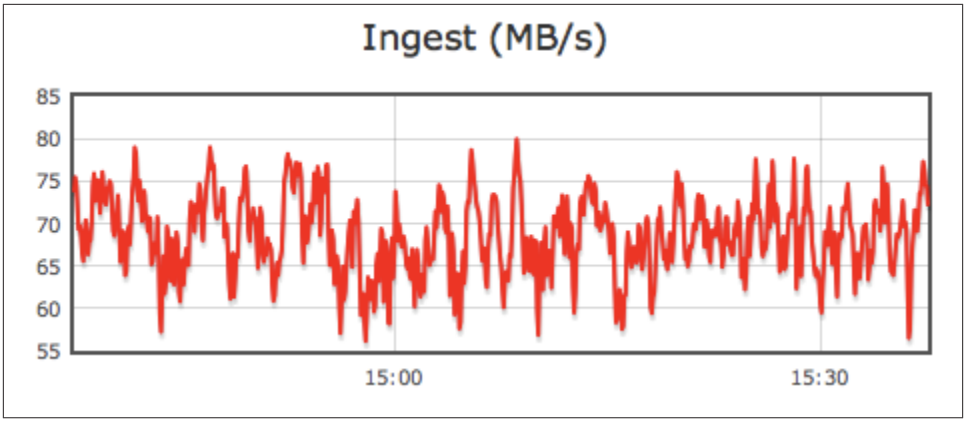

Figure 6-3. Aggregate ingest rate in megabytes of data per second

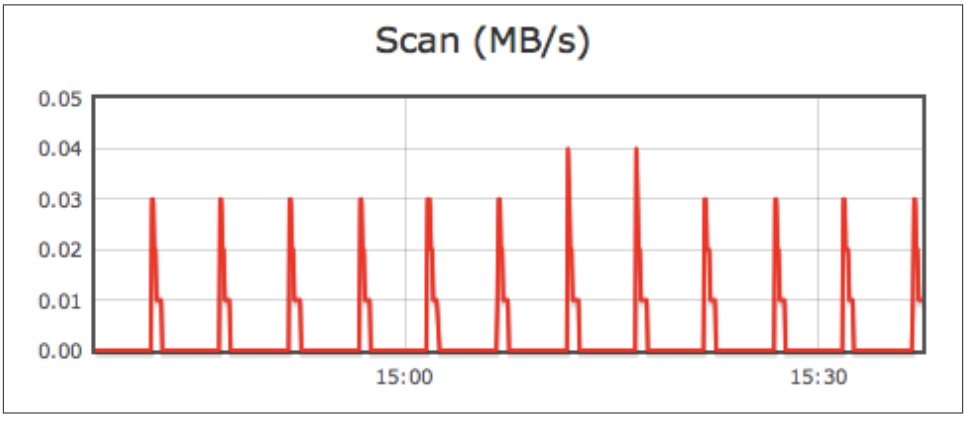

Figure 6-4. Aggregate scan rate in megabytes of data per second

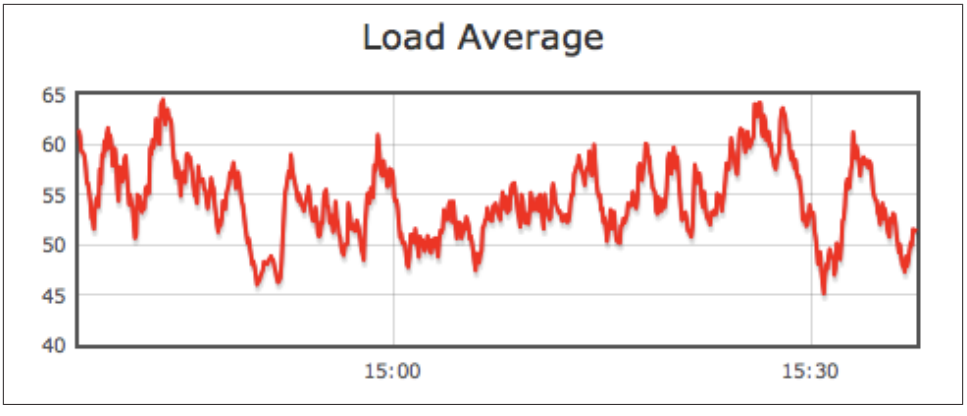

Figure 6-5. Aggregate CPU Load

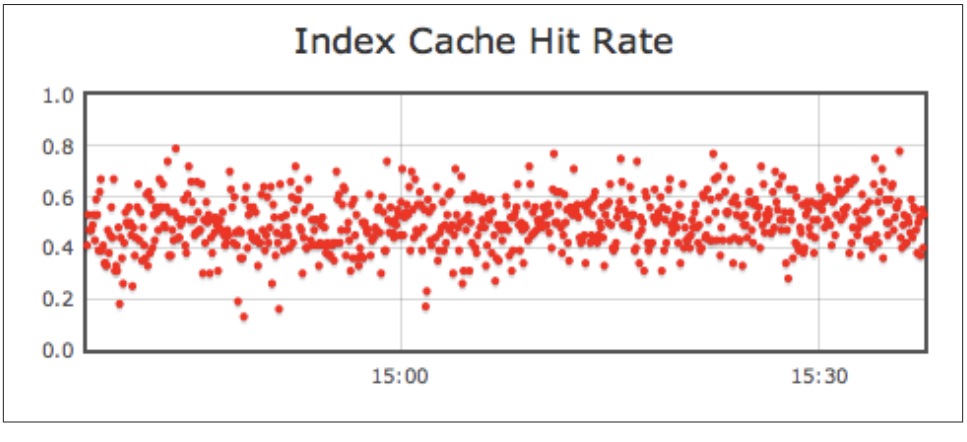

Figure 6-6. Number of index block cache misses (i.e. when Accumulo has to read an index block from HDFS)

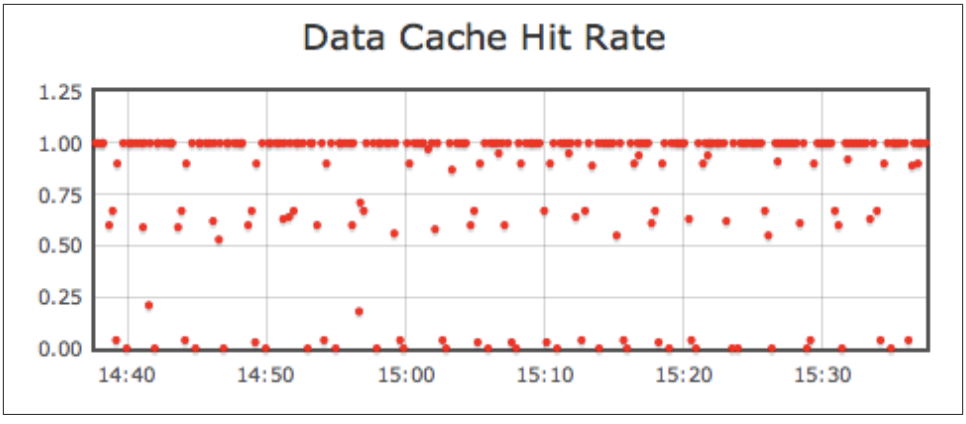

Figure 6-7. Number of data block cache misses (i.e. when Accumulo has to read a data block from HDFS)

In addition, users can measure the number of items that are being processed by Accumulo clients, which may be converting individual user requests into several key-value pairs. Knowing how to convert the number of user write or read requests into the number of key-value pairs or megabytes of data read and written will help verify that the cluster is performing optimally or that there is a problem.

For example, if every user request to write a record of application data involves writing the original record as one key-value pair to one table, and several key-value pairs for index entries of individual fields in the record, the ratio of application records to key-value pairs written may be 1 to 10 or more.

Getting familiar with these metrics will help in reasoning about performance and tuning decisions discussed in the next few sections.

Estimating write performance at scale

The best way to gather information on cluster performance starts with gathering empirical measures of performance using a single server. Accumulo is designed so that aggregate write and read performance scales with the number of machines participating in the cluster. The perfect theoretical limit is to scale linearly, meaning that by doubling the cluster size one gets double the aggregate performance, but as Amdahl's Law describes, because there are some overhead in operations that can't be parallelized, performance increases will be less than perfectly linear.

Informal testing shows that doubling the number of machines in an Accumulo cluster results in roughly and 85% increase in aggregate write performance. Several factors contribute to the efficiency of the performance increase seen when doubling the cluster, including network hardware and application design.

For the purposes of cluster planning using back-of-the-envelope calculations, a good practice is to prototype an application and measure the performance against a single server, and then against two servers, and look for the percentage increase in read and write performance. Users may have to write a significant amount of data in order to test the splitting and migration properties of tables before seeing an increase in aggregate write and read rates.

Based on these rates, users can estimate the number of machines required to reach a target number of user requests to read or write application data by multiplying 1 or 2 server aggregate rates by 1.85 until the target number of requests is reached.

For example, say we needed to be able to write a million key-value pairs per second, which would allow a theoretical MapReduce job writing to Accumulo to keep up with some reporting requirements.

Testing of an application prototype on some particular hardware reveals that the single server write rate is 120 thousand key-value pairs per second. Multiplying this by 1.85 we get an estimate of 222k pairs per second using two servers. We continue to multiply by 1.85 until we reach a million writes per second, doubling the number of servers in our cluster each time. At 4 servers we have a theoretical write rate of 760k key-value pairs per second. At 8 servers we have a rate of 1.4 million, so we need somewhere in between 4 and 8 servers.

A direct formula for estimating the number of servers required to reach a target write rate is as follows:

$$m = 2^{log_2(a/s)/0.7655}$$

```
m - estimated number of machines
a - target aggregate write rate in key-value pairs per second
s - measured single server performance in key-value pairs per second
```

On the other hand, if one wants to measure the total expected read or write rate of an existing set of servers, one can measure application performance against one or two servers and extrapolate to the size of the cluster to get the aggregate write rate.

For example, if we measure an application as being able to write 5 thousand user requests per second against one server (where each user request translates into several key-value pairs), and we have 20 servers, we can expect to see an aggregate write rate of about 71 thousand key-value pairs on 20 servers, or about 14x the single-server write rate.

To estimate the aggregate write performance of a cluster given the number of machines and single-server performance, the following formula can be used:

$$a = s \times 0.85^{log_2(m)} \times m$$

```
a - estimated aggregate write rate in key value pairs per second
s - measured single server write rate in key-value pairs per second
m - number of machines
```

Tuning

There are several places where performance bottlenecks may occur in an Accumulo application. Every application is bound by some resource of the system. Ideally that component is constrained only by something like available money or space.

In particular, it is important to make sure that Accumulo applications are taking advantage of all the hardware resources available and that each hardware component is being used efficiently.

Potential bottlenecks include:

- The number of clients writing or reading data - if there are not enough clients available to ship new data to or scan data from the cluster, tablet servers will be underutilized.
- The number of tablets available - if not enough tablets are available, some Tablet Servers will be idle, neither accepting writes nor serving queries. If performance is unsatisfactory for a given table and the number of tablets the table is split into is less that the number of available tablet servers, adding additional split points or turning down the split threshold temporarily will cause there to be enough tablets for every server to have one and participate in serving requests for the table.
- The number of operations per second that a single HDFS NameNode can support - at very large scales, the sheer number of file system operations can become the bottleneck of a cluster. A single HDFS namenode is limited to a few thousand operations per second since each operation is synced to disk.
- The throughput of the network - as clusters grow the network can become the bottleneck if it isn't scaled up along with the number of servers. In some cases upgrading from gigabit ethernet to 10 gigabit ethernet or upgrading the switches connecting racks is required to avoid the network becoming a bottleneck.
- The number and distribution of keys being read from or written to in tables (hotspots) - even if there are tablets on each server, it may be the case that all of the incoming keys end up going to a small number of tablets, if there are common keys that are appear frequently.
- The relative amount of CPU, RAM, and hard drives available per server - for example, servers may have lots of CPU but not enough disks or vice versa.
- The number of servers participating in the cluster - This is the ideal bottleneck. Performance of the application can be increased by adding more machines.

Mostly tuning will consist of balancing these resources relative to one another. For example, if we have 100 tablet servers available to host tablets but only one client process writing to the cluster, our aggregate performance will not increase as a result of adding

more tablet servers as the single client process is already limiting the max read and write performance.

Tablet Server Tuning

Tablet Servers are multi-threaded and will take advantage of multiple cores on a server. Additional tuning should be done according to the particular hardware on which Tablet Servers will run and what other processes are present.

External settings

There are a number of settings external to Accumulo that must be configured for Accumulo to work properly.

Number of open files
: Many file operations are performed whenever data is read from or written to Accumulo, and whenever Accumulo is cleaning up by performing a major compaction. Since each tablet server can be responsible for up to thousands of tablets, each of which may have multiple associated files, the default OS limit on number of open files is not sufficient. This limit should be increased to a large value such as 65536. Note that this is generally configured per user, so make sure it is increased for the user running Accumulo.

```
# vi /etc/security/limits.conf

...
accumulo       nofiles hard    65535
accumulo       nofiles soft    65535
```

Swappiness
: Accumulo requires its Tablet Servers to maintain active ZooKeeper locks to ensure their liveness. If these processes swap out memory, it is likely that they will lose their locks and exit. To keep this from happening, set swappiness to a low value as described in "Kernel tweaks" on page 115.

Number of ZooKeeper client connections
: All of Accumulo's processes must communicate with ZooKeeper frequently. There is a limit on the number of concurrent connections each server may make to ZooKeeper, so it is essential that this is increased by setting `maxClientCnxns=100` in ZooKeeper's *zoo.cfg* file.

```
# vi /etc/zookeeper/zoo.cfg

..
maxClientCnxns=250
```

Number of HDFS threads used to transfer data
: Depending on your version of HDFS, you may or may not need to increase the number of threads that can be used to transfer data. If the version you are using has

the `dfs.datanode.max.transfer.threads` property which defaults to 4096, you do not need to adjust it. If HDFS is still using `dfs.datanode.max.xcievers`, its value should be increased to 4096.

```
# vi /etc/hadoop/conf/hdfs-site.xml

<property>
  <name>dfs.datanode.max.xcievers</name>
  <value>4096</value>
</property>
```

HDFS durable sync
: For Accumulo 1.5 and later, HDFS durable sync must be enabled because Accumulo uses HDFS for its write-ahead log. Older versions of Hadoop may need `dfs.support.append` set to `true`, but newer versions default `dfs.durable.sync` to `true`, so that value merely needs to remained unchanged. In newer versions of Hadoop, the property `dfs.datanode.synconclose` should also be set to `true` to ensure that data in Accumulo RFiles is synced to disk when the files are closed.

In older versions of Hadoop:

```
# vi /etc/hadoop/conf/hdfs-site.xml

...
<property>
       <name>dfs.support.append</name>
       <value>true</value>
</property>
```

In newer versions:

```
# vi /etc/hadoop/conf/hdfs-site.xml

...
<property>
       <name>dfs.durable.sync</name>
       <value>true</value>
</property>
<property>
       <name>dfs.datanode.synconclose</name>
       <value>true</value>
</property>
```

Table Settings

For Accumulo to distribute the workload, you want your data spread out to multiple tablet servers. We have already discussed row design and will discuss it in more depth later in the chapter. Here are few other things to consider.

Tables with lots of little writes and lots of reads
: Any table that gets lots of little writes and lots of reads can slow down over time. These write are made in memory and to the write ahead logs. Reading from these

tables must combine the information in memory with the information written in disk, which may or may not be cached in a different memory location. Although it sounds counter intuitive, flushing this table will write the updates memory to disk and combine it with the information in the existing rfiles. As we have detailed, this is a minor compaction. The metadata table is an example of such a table. In Accumulo 1.6 and later, flushing the metadata is handled for you. For metadata prior to Accumulo 1.6 and any other table that falls into this category, you can improve performance with a scheduled flush, something like the follwing.

```
/path/to/accumulo/bin/accumulo shell -u username -p password -e 'flush -t tablename -w'
```

The -w switch causes the shell to wait and allows you redirect the output of the entire command to a file if desired.

Number of rfiles per table
: In between major compactions, the number of rfiles per range of data in a table will grow. You can improve performance by scheduling a compaction of the rfiles. For tables with more than a trivial amount of data, you will want to compact a range of data instead of the entire table. The major compaction will reduce that number of rfiles, which will improve performance in Accumulo and in HDFS.

Small tables need splits too
: If your application has a table with a small amount of data, make sure you split it up. Such tables could be filled with lookup information, or generated from another part of your application and so on. If these tables are on one tablet server, you are not taking advantage of Accumulo's distributed abilities and my be creating artificial hotspots. One way to split the table is the break the range up in 1 part for every node in your cluster that hosts tservers. This may not distribute it very well if data is not evenly distributed by row id. Another way is to compact this table and then look at the size of the current rfile. Then in the table config, add table.split.threshold roughly equal to the rfile size divided by the number tserver nodes. Run a compaction on the table again to make the splits. Be sure the set the table.split.threshold back to something reasonable to maintain those split points.

Memory Settings

TIP: Processes other than the Tablet Server only need sufficient memory to operate and don't benefit from increased memory. A potential exception is the Accumulo Garbage Collector. If you have a lot of tablets/files/servers, making the GC larger will keep collection efficient. Basically, if you can't keep the list of all files which are candidates for deletion in memory, it has to use multiple passes to reclaim files.

There are a few things to consider when configuring the amount of memory dedicated to various components of Tablet Servers.

In-memory map (for write performance)
: The in-memory map is where new writes are stored and sorted in memory, in addition to be written to disk in a write-ahead log, until they are flushed to disk in a minor compaction. In all but trivial testing systems, native in-memory maps should be built, if necessary, and enabled by leaving the `tserver.memory.maps.native.enabled` property set to `true`.

To ensure native-memory maps are being used, make sure the libraries exist in $ACCUMULO_HOME/lib/native. If not, they can be built by running the `make` command in $ACCUMULO_HOME/server/src/main/c++/.

At startup time Tablet Server logs in $ACCUMULO_HOME/logs/tserver*.log will show whether native maps are enabled:

```
...
[server.Accumulo] INFO : tserver.memory.maps.native.enabled = true
...
```

And whether or not they were not found:

```
...
[tabletserver.NativeMap] ERROR: Failed to load native map library ..
...
```

Or found:

```
...
[tabletserver.NativeMap]
...
```

The amount of memory that each Tablet Server reserves for in-memory maps is `tserver.memory.maps.max`, and increasing this property can improve write speed and decrease compactions. Note that the memory is divided among all tablets actively receiving writes on the same tablet server, so the property may need to be increased significantly to have a noticeable effect. Also, if the in-memory map size is increased, the number of write-ahead log files should also be adjusted. When a tablet has a given number of write-ahead logs, it will automatically be flushed, even if memory is not full. So, the number of write-ahead logs (`table.compaction.minor.logs.threshold`) times the size of each log (`tserver.walog.max.size`) should be at least as big as the amount of memory given for in-memory maps (`tserver.memory.maps.max`).

For example, if we are setting `tserver.memory.maps.max` to 12GB, and `tserver.walog.max.size` is set to 4GB, we would want to increase `table.compaction.minor.logs.threshold` to be greater than 3.

Increasing the size of the write-ahead logs via `tserver.walog.max.size` would make recovery take longer, so try not to adjust that property.

Cache memory (for read performance)

In general, more memory dedicated to caches will provide better query performance. However, Java Garbage Collection may then take longer to reclaim memory from objects no longer in use, decreasing Tablet Server responsiveness, which can interfere with ZooKeeper's attempts to determine server liveness. The cache size properties are `tserver.cache.index.size` and `tserver.cache.data.size` and they govern the amount of memory used to cache RFile index blocks and RFile data blocks, respectively. Ideally, the index cache size should be chosen so that all index blocks fit in memory. To estimate this size, the number of files per tablet and tablets per tablet server must be determined, as well as the average index size per file (see "Inspecting RFiles" on page 149 for information on inspecting individual RFiles). Any amount of data cache will be utilized, so its size can be made as large as makes sense for your application and your hardware.

Heap size

The Tablet Server heap size is an environment variable `ACCUMULO_TSERVER_OPTS` set in the *accumulo-env.sh* file. It should be chosen large enough to cover the total size given to caches, plus some overhead. If native maps are not enabled (which is not recommended), the heap size must also cover the size of the in-memory maps. Tablet Servers will error out if the memory allocation does not add up when native maps are turned off. The in-memory maps and caches are not the only things the Tablet Server stores in memory, but they are generally the values that are tuned larger when there is more memory available, and their values therefore have potential to be the largest contributors to a Tablet Server's memory usage.

tserver.mutation.queue.max

This is the number of bytes of write-ahead log data to store in memory before it is flushed to disk. Setting the value too low reduces Accumulo's throughput, but setting it too high can result in memory exhaustion if there are many concurrent writers. Values of `2M` or `4M` may be reasonable.[1]

Compaction settings

table.compaction.major.ratio

By default this is set to 3. If the total size of set of files over the largest file in the set is greater than this number, a Tablet Server will compact the files into one file. So by default the total size of a set of files has to be greater than three times the size of the biggest file in the set before the Tablet Server kicks of a major compaction. If this is not the case the Tablet Server considers the set of files with the largest file

1. See *https://issues.apache.org/jira/browse/ACCUMULO-1905* and *https://issues.apache.org/jira/browse/ACCUMULO-1950* for a more detailed analysis.

removed, and so on until a compaction is scheduled or until there are no more sets of files to look at.

Setting this ratio higher will cause Tablet Servers to do fewer compactions, leaving more IO resources for queries and inserts, but queries will require opening more files, increasing query latency. Setting this ratio to 1 will cause a major compaction every time there is more than one file for a tablet.

table.file.max
: To keep the number of files for a tablet from growing too large, a maximum can be specified. Setting this too small will result in inefficient use of I/O, because once the maximum is reached every flush from memory to disk involves rewriting an existing file in a merging minor compaction. Do not set the maximum to 1, or else the single, potentially very large file for the tablet would have to be rewritten for the tablet server to be able to be able to flush its in-memory map for the tablet.

tserver.compaction.major.concurrent.max and tserver.compaction.minor.concurrent.max
: These control the number of major and minor compactions that will run concurrently. If your system is not I/O bound, these could be cautiously increased to give more resources to compactions.

Read-ahead settings

tserver.readahead.concurrent.max tserver.metadata.readahead.concurrent.max

Balancing Inserts and Queries

Accumulo allows users to dedicate system resources to ingest or queries as necessary. By default, Accumulo does not throttle ingest in order to keep some resources available for query, so query performance can suffer if clients are using too much of Accumulo's resources for writing.

One method of throttling ingest and improving query performance is to decrease the maximum number of files Accumulo is allowed to create per tablet. This is controlled with the per-table setting `table.file.max` which defaults to 15. When the maximum number of files has been reached for a tablet, Accumulo will merge new data with data from one of the existing files instead of creating a new file for that tablet. The new, merged file will then replace the existing file. This process is called a *merging minor compaction*.

When merging minor compactions are occurring, the overall write rate of the cluster may begin to decrease, as a minor compaction to free up memory for new writes may be waiting on a merging minor compaction to complete. While this is happening, any new writes to the server cannot proceed and clients are told to wait. Increasing the `table.file.max` setting and/or decreasing the `table.compaction.major.ratio` set-

ting will ensure that enough background compactions occur so that minor compactions will not end up having to wait and block clients.

Running Alongside MapReduce Workers

It is common for Accumulo processes to be deployed on the same servers that are hosting MapReduce worker processes, such as Hadoop TaskTrackers. If this is the case it is important to make sure that there are enough available hardware resources for each process when all the processes are being utilized.

For example, the number of MapReduce task slots (simultaneous workers) should be multiplied by the amount of RAM allocated for each slot, and added to the RAM allocated to Accumulo Tablet Server and any other running processes. If this amount exceeds the available RAM, processes should each be allocated less RAM or the number of available MapReduce workers should be decreased to avoid a situation in which pages of RAM will be swapped to disk, which can cause delays that may be interpreted as a failed server and cause Tablet Servers to be terminated and excluded from the cluster.

The IO resources of servers will also be shared in this case, and running Accumulo compactions will affect MapReduce performance and vice versa. There is a limit to the number of files Accumulo will allow a tablet to have before forcing a major compaction, in order to keep the number of file resources per scan reasonable. If there is not enough IO on a server to support the number of compactions required to organize newly written data, compaction tasks will queue up on Tablet Servers. This can be seen in the Monitor on the Tablet Servers view

…

If compactions are queuing up, the resources dedicated to compactions may need to be increased, at the expense of resources dedicated to MapReduce. This may be an indication too that simply more hardware resources are needed.

Sharing ZooKeeper

Some other applications running on or close to the Accumulo cluster might also make use of ZooKeeper. In particular, Apache Kafka[2] is a popular distributed queue used to stream data to and from applications. Some Accumulo clients may make use of Kafka to stream data to Accumulo or from Accumulo to other applications.

Similarly Apache Stormfootnote[*https://storm.incubator.apache.org*], a popular streaming data processing framework, relies on ZooKeeper for configuration information as well.

2. *http://kafka.apache.org*

Be cognizant of the load placed on ZooKeeper by these other systemsfootnote[See ZooKeeper documentation on monitoring *http://zookeeper.apache.org/doc/r3.4.6/zookeeperAdmin.html#sc_monitoring*]. Although ZooKeeper itself is a distributed application, at any given time one machine of the quorum is serving as master, meaning that all writes go to it, although reads can go to other members of the quorum. This means that one server has to handle all writes, and replicate those writes synchronously to other members of the quorum, for the entire all applications using it on the cluster. Note that this means scalability of writes isn't improved by adding more machines, rather it makes each write *more expensive*, reducing the total write throughput of the ZooKeeper instance.

If ZooKeeper can't keep up with operations Accumulo will not function properly. There is really no reason multiple separate ZooKeeper instances can't be run on a cluster, each one consisting of 1, 3, or 5 nodes, as long as applications are configured to use different instances.

Scaling Vertically

Adding more memory and CPU to a single server will help a single tablet server process cope with more concurrent queries and writes. Modern servers can have up to 12 disks or more, which can increase the amount of CPU and RAM required to keep those disks busy.

Write-ahead logs can become a bottleneck for ingest since tablet servers each use one, albeit replicated, write-ahead log. If the ingest rate of a server is dominated by the time spent flushing mutations to the write-ahead log, adding more disks or CPU to each server will not increase the write rate. Adding more RAM and increasing the `tserver.mutation.queue.max` and `tserver.memory.maps.max` will improve performance up to a point. For this reason, it may be more cost effective to have more individual servers, each with less resources so that a greater portion of available disks are devoted to write-ahead logs. This is consistent with Accumulo's design to run well on relatively cheap servers.

Running multiple tablet servers per node is not recommended.

Cluster Tuning

In addition to having per-server properties properly tuned to take advantage of hardware efficiently, tuning should be performed across the entire cluster as well, in terms of balancing the number of clients versus Tablet Servers.

High Speed Aggregate Insert Rates

In order to maximize aggregate insert rates, applications should start by considering three numbers:

- The number of tablet servers
- The number of tablets
- The number of client processes writing to tablet servers

Before tablet servers can all participate in ingesting data, there have to be at least as many tablets in a table as tablet servers in order for the Accumulo Master to be able to assign at least one tablet to each server. See the section on Splitting Tables for details.

Once each server has at least one tablet from the table to which an application wants to write, there must also be enough client processes available to avoid artificially limiting the aggregate write rate. If the theoretical limit of the tablet servers in the Accumulo cluster is a million writes per second, and if client processes max out at 100 thousand writes per second we'll need at least ten client processes to reach our cluster maximum.

Once these are roughly balanced, then the next things to address are the load balancing strategy used by the Master, and whether we have hotspots in our table, as described in the next few sections.

Load Balancing

The Accumulo Master can use different load balancers to achieve a good distribution of tablets across tablet servers. By default, each table's tablets are balanced separately and are assigned evenly and randomly to tablet servers. However, some applications may require more fine-tuned control over their tablets. Some key design patterns require not only that a table's tablets are distributed evenly, but that specific subsets of tablets are also distributed evenly.

An example might be a table with a row key containing a date. Suppose an application built on top of this table frequently accesses dates falling within the same month at the same time. Using the default load balancer could end up assigning all the tablets for a month to a single server, limiting the insert and query capabilities across that group of tablets. Instead, this table could employ a custom load balancer to ensure that tablets falling within the same month are distributed evenly to tablet servers.

To write a custom load balancer, implement a class that extends `TabletBalancer`. Add a jar containing the new balancer to Accumulo's classpath on at least the Master nodes, and configure a table to use this balancer by setting the `table.balancer` property for the table.

```
public class CustomLoadBalancer extends TabletBalancer {

  @Override
  public void getAssignments(SortedMap<TServerInstance,TabletServerStatus> current, Map<KeyExtent,
    // populate the assignments map with new tablet server assignments based on the current assign
  }
```

```
  @Override
  public long balance(SortedMap<TServerInstance,TabletServerStatus> current, Set<KeyExtent> migrat
    // add new desired migrations to the migrationsOut list based on the current tablet server sta
    // return the amount of time to wait before balance is called again
  }

  @Override
  public void init(ServerConfiguration conf) {
    super.init(conf);
  }
}
```

Splitting Tables

Brand new tables in Accumulo start out as a single tablet. Accumulo automatically splits tablets when they reach a certain threshold knowns as the **table.split.threshold** setting in the tablet configuration.

Some applications might be bottlenecked by the number of tablets until there are enough tablets for every tablet server to host one or more. Users can choose to either wait until there is enough data ingested for their table to split automatically into the desired number of tablets, turn down the split threshold temporarily to cause automatic splits to happen sooner, or pre-split the table using a set of known split points.

Lowering the split threshold temporarily has the advantage of allowing Accumulo to still pick the split points uniformly, no matter what kind of distribution of keys exists within a table. This still assumes that the splits that will occur on the initial amount of data ingested are representative of the split points that would have been chosen after ingesting all the data. For example, if we are importing a list of users that has been sorted alphabetically, the initial split points will only occur within the first few letters of the alphabet and will not be representative of how the data would be split after the entire list is imported.

Users can try to obtain a representative sample of their data for the purposes of ingesting and allowing Accumulo to find good split points early.

To lower the split threshold for a table, users can configure the table in the shell thus:

```
config -t tableName -s table.split.threshold=1G
```

Rather than lowering the split threshold, users can submit a list of split points to Accumulo to use to create multiple tablets. The advantage of this is that the table will be distributed onto more servers before any data is ingested. The onus of picking good split points rests with the user.

To pre-split a table from a list of points, the split points should first be put into a text file, with one point per line, for example:

```
e
j
```

```
o
t
```

Adding these split points to a single-tablet table would result in 5 tablets: (-infinity, e], (e, j], (j, o], (o, t], and lastly (t, infinity). The special tablet with end point at infinity exists for every table and is called the *default tablet*.

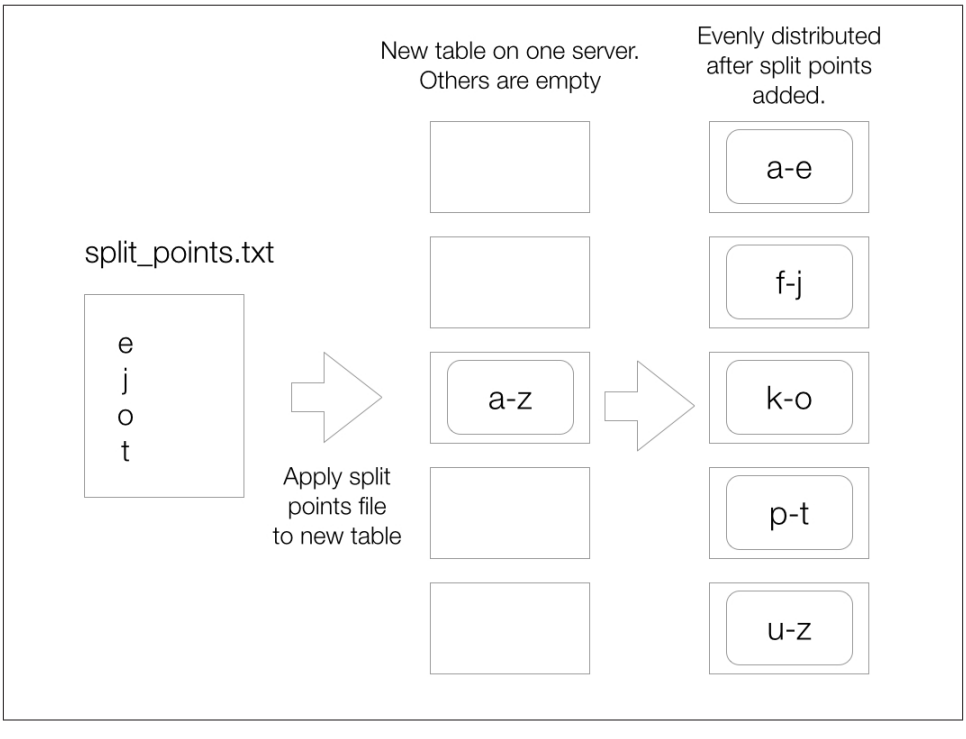

Figure 6-8. Adding Split Points

Then the split points file can be submitted to Accumulo for spitting via the shell:

```
user@accumulo> addsplits -t tableName -sf fileName
```

A file with split points can also be provided when first creating a table through the shell. To copy the split points from an existing table use:

```
user@accumulo> createtable myPreSplitTable --copy-splits existingTable
```

To use a file do the following:

```
user@accumulo> createtable myPreSplitTable --splits-file mySplitsFile.txt
```

A small set of specific split points can be added directly in the Shell:

```
user@accumulo> addsplits e j o t -t mytable
```

Splits can be added via the Java API as well.

```
ZooKeeperInstance inst = new ZooKeeperInstance(myInstance, zkServers);
Connector conn = inst.getConnector(principal, passwordToken);

SortedSet<Text> splitPoints = new TreeSet<>();
splitPoints.add(new Text("e"));
splitPoints.add(new Text("j"));
splitPoints.add(new Text("o"));
splitPoints.add(new Text("t"));

conn.tableOperations().addSplits("mytable", splitPoints);
```

If a set of split points are to be used to pre-split a tables from time to time, or to be distributed along with an application for use with multiple sources of data, care should be taken to ensure that the split points used on a table are representative of the distribution of the data to be loaded. The distribution of keys within a data set may change over time, and may not be the same from one data source to another.

Accumulo can also merge tablets, which is covered in "Merging Tablets" on page 223.

Physical Data Locality

In MapReduce jobs, the notion of physical data locality, in terms of the distance between data to be processed and the CPU and RAM elements in which is will be processed, is extremely important. Many types of MapReduce jobs are run on data that is so large that reading it from some storage medium over a network would limit the performance in an unacceptable way. The primary innovation of MapReduce versus many other types of data processing is that it sends the computation to the data, rather than moving the data to the computation.

This means that when key-value pairs are processed in a MapReduce job, the key-value pairs are read from a local disk by a copy of the Map or Reduce process that has been sent to the machine holding the data. This allows the overall job to be limited by the aggregate throughput of all the hard drives in the cluster, which is often much higher than the aggregate throughput rate of the network connecting machines in the cluster.

Data is stored in HDFS which automatically distributes it over many machines and handles replication and helping programs find a particular piece of data by exposing the IP address of physical machine on which it is stored.

For Accumulo, the concept of physical data locality is still important, but not paramount. Because Accumulo uses HDFS to store files, each time a Tablet Server flushes a new file to HDFS as part of the minor compaction process, one copy of that file is stored locally, on the machine hosting the Tablet Server process. Subsequent reads can then simply read from local disk rather than pulling data from a remote machine over a network.

Over time, as the Accumulo Master performs load balancing of tablets, some tablets may reference files for which there is no local copy, forcing reads to pull data from a

remote machine over the network. But eventually, the major compaction process will tend to create new files for each tablet, merging several old files into one new file, which will cause a local copy to be created. Good physical data locality is something Accumulo achieves eventually and asynchronously.

Accumulo has a utility for checking the level of data locality in a cluster. It can be run via the **accumulo** command thus:

```
accumulo org.apache.accumulo.server.util.LocalityCheck -u root -p secret

Server         %local  total blocks
 10.10.100.1 100.0        7
 10.10.100.2 100.0       10
 10.10.100.3 100.0       10
```

If for some reason a cluster has poor data locality, increasing the frequency of major compactions or scheduling a major compaction can cause files to be rewritten. To compact a table, use the compact command from the shell thus:

```
accumulo@cluster> compact myTable
```

Note that this will completely rewrite all files in a table. When stopping and restarting a cluster, Accumulo tries to reassign tablets to the same Tablet Servers that were previously hosting them before shutting down, so that physical locality is preserved.

Bulk Loading vs Streaming Ingest

An alternative to ingesting data via clients using BatchWriters to stream key-value pairs into Accumulo tables is to prepare key-value pairs into an Accumulo specific file format and simply add them to Accumulo tables.

To understand how Bulk Loading works, it's helpful to review how Accumulo the data of tables is stored in HDFS. An Accumulo table consists of a set of files in HDFS and metadata describing which files belong to which tablets and which key ranges each tablet spans. When ingesting via an Accumulo client, files called RFiles are created and stored in HDFS as part of the minor compaction process. After tablets are split they may share an RFile until a major compaction process writes out a separate set of RFiles for each tablet.

Eventually, Accumulo will end up having one or a small number of RFiles for each tablet and each RFile will only contain data for one tablet. This represents a kind of equilibrium state for Accumulo where no more compactions are necessary.

Users who want to get data into Accumulo at a very high rate of throughput can use MapReduce to create the RFiles such that they closely resemble the set of RFiles and tablets that Accumulo would create on it's own if the data were to be ingested via streaming clients.

Creating RFiles via MapReduce can be faster because a data set can be organized into the optimal set of RFiles in one MapReduce job rather than via several rounds of compaction on intermediate RFiles.

The downside of bulk loading is that none of the data is available for query until the entire data set is done being processed by MapReduce. It also requires that all the data to be loaded is staged in HDFS.

Bulk loading is an option for quickly loading in a large data set into Accumulo when it is possible to stage the data in HDFS and when the latency requirements are such that the data can be unavailable until the MapReduce job is complete.

Bulk Ingest to Avoid Duplicates

Another reason to use bulk import is to avoid writing duplicate entries into Accumulo tables when using a large number of clients to write data. The more clients involved in writing data, the higher the chance is that one may fail. If clients are simply writing data to Accumulo in response to individual requests, this may not be much of a problem. Applications can use conventional load balancers to find a live client and write their data.

However, in a scenario in which clients are writing information from a set of files for example, the loss of a client means that it is likely that only a portion of a file was ingested, and if another client is directed to re-ingest the file, there is a chance that it will create duplicate entries in the table.

One way to avoid this is to make the key-value pairs written for each piece of input data deterministic. That is to say, each input record is converted into the same set of key-value pairs no matter when or which client is ingesting the record. This may still result in the same key-value pair getting written more than once, but the Accumulo Versioning Iterator can be configured to ignore all but the latest version of a key-value pair, effectively eliminating duplicates.

Sometimes creating deterministic key-value pairs is not an option. For example, an application may want to create key-value pairs for an input record that use the timestamp of when the data was ingested as part of the rowID. This would allow data to be read from Accumulo roughly in the order in which it arrived. (For more discussion on storing data in time order see the section on Time Ordered Data)

In this case, reloading some input records from a partially processed input file would result in duplicate records with different rowIDs. Using MapReduce and bulk loading would avoid loading in any key-value pairs from a file that was partially processed when the machine processing it suffered a failure. This can also allow for loading some set of key-value pairs all together as an atomic unit, as each RFile is either completed and loaded or discarded so another worker can produce a complete file.

Running Large Scale Clusters

Accumulo is designed to run on clusters of up to thousands of machines. There are some things to consider when running at very large scale that may not be an issue on smaller clusters.

Networking

As a distributed application, Accumulo relies heavily on the network that connects servers to each other. Like Apache Hadoop, Accumulo does not require exotic networking hardware, and is designed to operate well on commodity class networking components such as 1GB and 10GB ethernet. Modern Hadoop configuration recommendations include considering 10GB ethernet for improved latency.

Limits

Accumulo is designed to run on thousands of servers. The largest clusters begin to be bottlenecked not by any component of Accumulo but by the underlying subsystems on which it runs. In particular, a single HDFS Namenode becomes a bottleneck in terms of the number of file operations that the entire cluster can perform over time. This limit can be observed by looking at the time that the Accumulo Garbage Collector takes to complete one pass. If the Garbage Collector is taking over 5 minutes to run, the Namenode is likely a bottleneck.

Accumulo 1.6 introduces the ability to run Accumulo over multiple Namenodes to overcome this limitation.

Using Multiple Namenodes

Accumulo version 1.6 and later can store files using multiple Namenodes. There are several options for doing this. One is to configure Accumulo to run over two or more separate HDFS instances, where each has a Namenode and a set of Datanodes and Datanodes each store data for only one Namenode. In this case, Datanodes are not shared between Namenodes and they operate without any knowledge of each other. Accumulo simply keeps track of which files live in which HDFS instance.

Accumulo does this by using full path names to all the files under management, including the hostname of the Namenode of the HDFS cluster in which each file lives. After informing Accumulo of the list of Namenodes to use in the configuration file as described below, there is no other configuration necessary. Accumulo will automatically distribute files across HDFS clusters evenly.

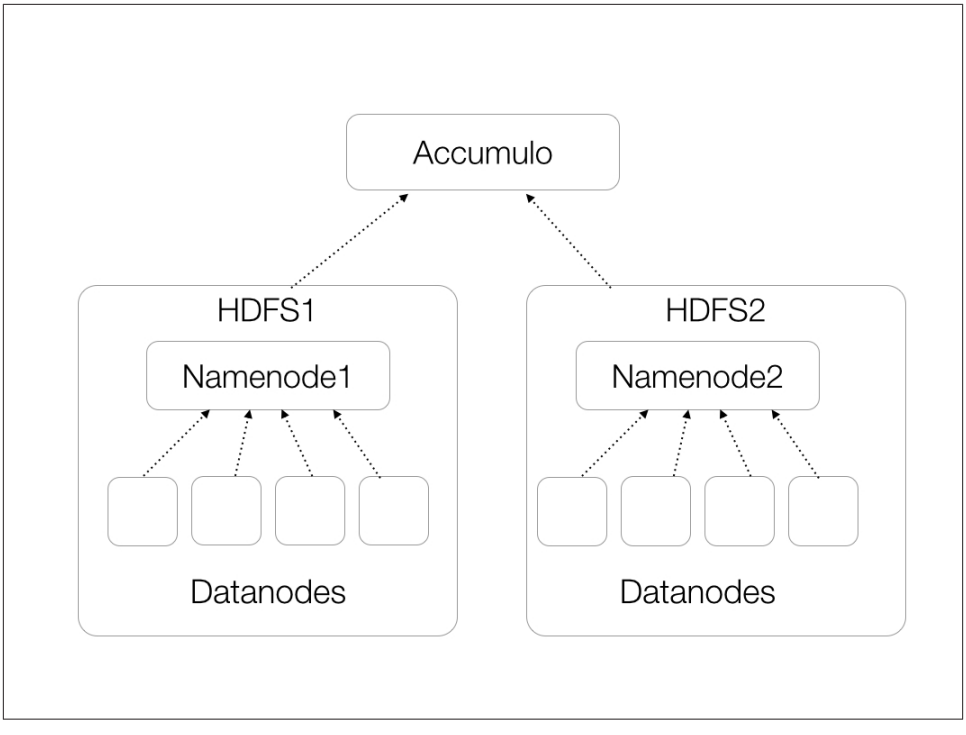

Figure 6-9. Accumulo on Multiple HDFS Clusters

Another option is to use Namenode Federation, in which a set of Datanodes are shared between two or more Namenodes. Federation can make it easier to keep the data in HDFS balanced as each Namenode can see the information on every Datanode and place data based on the load of all the Damenodes.

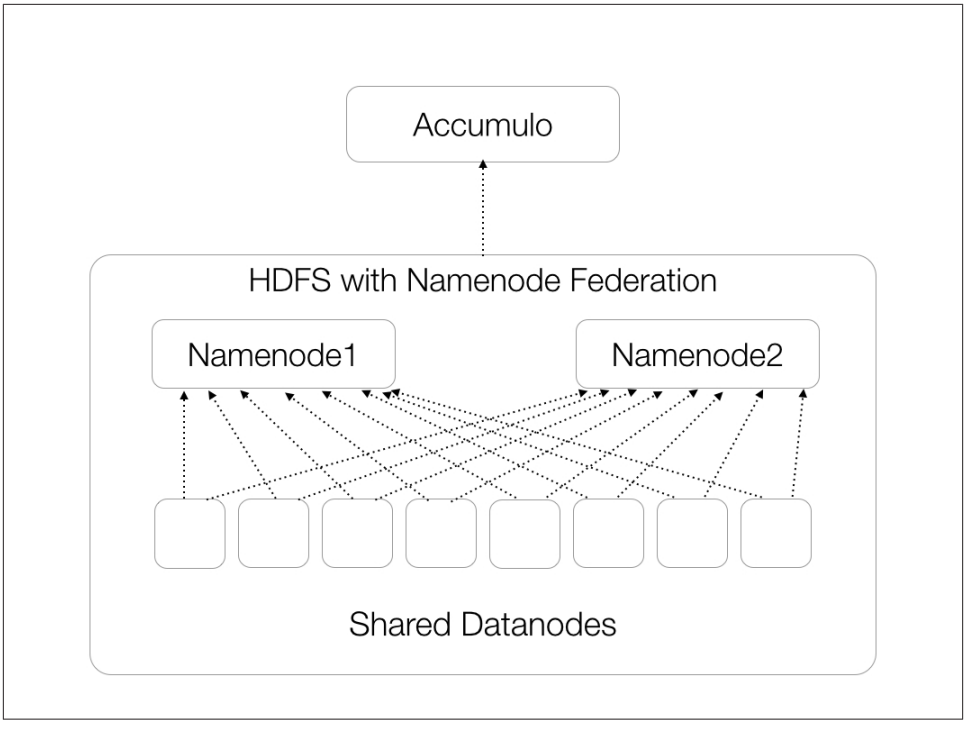

Figure 6-10. Accumulo on an HDFS Cluster Using Namenode Federation

In either of these cases, the configuration of Accumulo is the same. A list of Namenodes to use is specified in the accumulo-site.xml file under the `instance.volumes` property:

```
<property>
   <name>instance.volumes</name>
   <value>hdfs://namenode1:9001,hdfs://namenode:9001</value>
</property>
```

Accumulo instances that utilize multiple Namenodes are designed to scale to extremely large sizes, beyond a few thousand nodes to ten thousand or more. With modern hard drives each server could have up to 30 Terabytes raw, 10 Terabytes after replication, and a cluster of ten thousand servers could store up to 100 Petabytes. Accumulo provides a single unified view of all of this data and lookups remain fast because of the ordering of the keys in each table.

Handling Namenode Hostname Changes

Since Accumulo keeps track of the hostname of the Namenode when using multiple Namenodes, special care must be taken when moving a Namenode to a new hostname. If a Namenode is moved from *namenodeA* to *namenode1*, an additional configuration

property, `instance.volumes.replacements`, must be added and Accumulo must be restarted in order for Accumulo to be able to talk to the new Namenode.

```
<property>
   <name>instance.volumes.replacements</name>
   <value>hdfs://namenodeA:9001 hdfs://namenode1:9001</value>
</property>
```

If more than one Namenode changes hostnames this way, each pair of Namenode hostnames should be listed, with commas separating pairs of hostnames and spaces separating individual hostnames:

```
<property>
  <name>instance.volumes.replacements</name>
  <value>hdfs://namenodeA:9001 hdfs://namenode1:9001, hdfs://namenodeB:9001 hdfs://namenode2:9001
</property>
```

Metadata Table

Accumulo's Metadata Table is a special table designed to store the current location and other information about each tablet of every other table. As such it plays an important role in the operation of every Accumulo application.

By default, the Metadata table is configured to be scalable and to provide good performance for even large clusters. Understanding how the configuration of the Metadata table affects performance and scalability can help users fine tune their cluster.

In practice, large Accumulo clusters split the Metadata table fairly aggressively and host tablets of the Metadata table on more than half of the available Tablet Servers as the number of clients reading from the Metadata table requires that more resources than one server can provide are dedicated to serving these requests.

Config

The default configuration for the Metadata table is different from that of other tables. It is tuned for high performance and availability, and to take advantage of its relatively small size.

To improve query performance, the size of compressed file blocks is reduced from 100K to 32K, the data block cache is enabled so that the frequent reads by clients are serviced very quickly from data cached in memory, and the major compaction ratio is set to 1.

To increase availability and decrease the possibility of data loss, the file replication is increased to 5 (or the maximum replication defined in HDFS if that is less than 5, or the minimum replication defined in HDFS if that is greater than 5).

The split threshold is decreased from 1G to 64M because the Metadata table is so much smaller than data tables, and because we want the Metadata tablets spread onto multiple tablet servers for better read and write throughput.

There are two locality groups configured, so that columns that are accessed together frequently can be read more efficiently.

The Metadata table configuration generally does not need to be adjusted. If your Metadata table is heavily taxed early on, before it has gotten large enough for it to naturally split onto a desired number of tablet servers, you could lower the split threshold temporarily to obtain more Metadata tablets.

Scalability

The Metadata table's design is a not a limiting factor in the scalability of Accumulo. Going by the following simple calculation, the Metadata table can address more data than can be stored in HDFS.

> Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is sufficient to address 2^{34} tablets (or 2^{61} bytes in 128 MB tablets).
>
> — Chang et al.
> *Bigtable: A Distributed Storage System for Structured Data*

Tablet Sizing

Having fewer, larger tablets can reduce the overhead of managing a large scale cluster. This can be achieved by increasing the split threshold for splitting one tablet into two. Tablets that are 10s of GB in size are not unreasonable.

To increase the tablet split threshold, change the **table.split.threshold** in the shell

```
user@instance myTable> config -t myTable -s table.split.threshold=20GB
```

File Sizing

For the same reason larger tablet sizes can reduce overhead, it can be useful to increase the block size in HDFS to a value closer to the size of tablets. This reduces the amount of information the NameNode has to manage for each file, allowing the NameNode to manage more overall files.

To increase the block size for a table, set table.file.blocksize

```
user@instance> config -t mytable -s table.file.blocksize=1GB
```

Do not confuse **table.file.blocksize**, which controls the size of HDFS blocks for a given table with **tserver.default.blocksize**, which controls the size of blocks to cache in tablet server memory.

Using the root user

It is suggested that you do not use the *root* for anything other than table manipulation, like creating tables and granting privileges. Do not give the root user any security labels. By following this suggestion, you force developers and system admins to use the correct user to access data. This suggestion apply to any size cluster.

Restoring a cluster

While Accumulo is pretty resilent, there are times when servers go down unexpectedly and you need to restore a cluster. First thing is to run fsck on HDFS /accumulo folder to make sure you don't have missing or corrupt blocks. If there are problems in HDFS, run grep the Accumulo metadata table in the Accumulo shell to see if those rfiles are needed. For rfiles that are no longer referenced, you can delete those in HDFS. Assuming the rfiles are still referenced but are not part of the Accumulo metadata table, you will need to get those rfiles into a known good state. If corrupt rfiles are part of the metadata table, you will need rebuild the metadata by creating a new instance and importing all the data. Usually that known good state is from some time in the past, which will mean data is missing. To account for the missing data, you will need to replay all the changes since that known good state, unless you can figure out exactly what changes to replay. It is not recommended to modify the Accumulo metadata table to get to a good state. Here are some things you can do to help yourself with these step.

Replay data

Provide yourself the ability to replay incoming data. This may mean saving off the ingest source files for some period. It may also mean creating a change log of updates that can be replayed. Ensure you have timestamps for when this data was originally pushed to Accumulo.

Backup Namenode metadata

This is especially important for Hadoop 1 as the namenode is a single point of failure. Back up namenodes fsimage, edits, VERSION and fstime file so you can recover HDFS. Doing so will allow you get Accumulo into a good state from a ponit in the past and then you can do things like replay all updates since that point.

Backup table configuration, users and split points

If the Accumulo metadata table went away or got corrupted, you could bulk import the existing rfiles to recover. But to get your cluster to the same state, you would need to recreate the existing table configuration, table splits, and users. Some of this information is stored in zookeeper, but is tricky to pull it out. Some of this information is not saved anywhere outside the metadata table, which is why you should back it up yourself. The *config -t tablename* will show you the current table configuration. The *getsplits* command

in the Accumulo shell can be used to store the current splits. The *users* command in combination with the *getauths -u username* command will show you users and auths. The *systempermission*, *tablepermission* and *userpermission* will show permission information. Use these commands to dump text files with information.

Turn on HDFS trash

Turning on the HDFS makes a copy of every deleted file. Configuration is done by setting *fs.trash.interval* to a number of minutes greater than zero in core-site.xml, which is the default. The trash interval should based on how much your Accumulo changes and how much storage you have. For example, lots of HDFS storage and/or a higher rate of change in Accumulo would mean a longer trash interval.

Use a empty rfile

If you can't find a known good copy of an rfile, you can create a empty rfile that gets copied to that expected location. Accumulo 1.5.2 and later have a utility to create an empty rfile.

```
accumulo org.apache.accumulo.core.file.rfile.CreateEmpty /some/path/to/empty.rf
```

Prior to those versions, you can create a harmless rfile that will serve the same purpose

```
createtable foo
delete "" "" ""
flush -t foo
```

Now you can find the rfile that was created with something like the following and then copy or move it.

```
tables -l # look for id of the foo table, 22 for example
hadoop fs -ls /accumulo/tables/22/default_tablet
```

Always take Hadoop out of safemode manually

When Accumulo restarts, it will begin compacting and flushing write ahead logs. Additionally, any client will be able to write data, which could get flushed to rfiles. You can setup Hadoop to not come out of safemode automatically, which will prevent any changes from happening to rfiles. Setting *dfs.namenode.safemode.threshold-pct* to a value greater than 1 in the Hadoop hdfs-site.xml config file will require human intervention to take HDFS out of safemode.

Troubleshooting

Exception when scanning a table in the shell

When scanning a table in the shell results in an exception, it could be a bad formatter

```
config -t ?
```

will show what is being used.

```
formatter -r
```

will remove the formatter.

Graphs on the monitor are blocky

What we mean by "blocky"" is that the lines are completely horizontal for a period, there is a increase or decrease, then more horizontal lines, then increase or decrease and so on. This means that tservers are having delays in reporting information back to the monitor. Tservers report info back every 5 seconds. If data for 2 or more periods are late, the monitor uses the prior value. Usually this is an indication that one or more tservers are having trouble. You can look on the tablet server monitor page and sort by last update time to get an idea of which servers are having trouble. Another way to find the server is to start up a shell, run *debug on*, then *scan accumulo.metadata*. It will take some time, but you should see messages repeated with the IP address of problem servers. Once you find out which tserver or tservers are having problem, you can go there and look at system monitoring tools and the logs to diagnose better the cause.

Restarting the master

It is safe to stop and start the master if needed. Sometimes error message show tablets are not balancing, but there are no down tservers and no other indication. Or the master is having problems communication, but there is no other apparent cause. As scary as it may sound, Accumulo will gracefully handle stopping the master service and restarting.

Calculating a size of changes to a cloned table

Sometimes it is useful to see how much a table has changed since it was cloned. We talked about cloning in table operations and why it might be useful as a *snaphot* of the original table. But the original table is going to continue to change as data is inserted and deleted.

The *du* command in the Accumulo shell can be used to see the size in bytes of a table. When passed in multiple arguments, it also shows how much space is shared.

Here is an example run in the shell,

```
du table1Clone table1

 1,232,344 [table1]
51,212,424 [table1, table1Clone]
   723,232 [table1Clone]
```

This is showing that 51.2M are still shared between the tables, 723K have been removed from table1 and 1.2M have been added to table1 since the clone.

Unexpected or unexplainable query results

If you get unexpected results when running a query, be sure to consider all the iterators being applied. This includes both iterators configured on the table and iterators being applied programmatically to the scanner. Having a scan iterator at the same priority as a table iterator is allowed. This is so that table iterator options can be overridden at scan time by configuring an identical iterator at scan time with different options. But having different iterators at the same priority will cause unexpected behavior, because only one is applied and it is undeterministic which one. Two scans that appear exactly the same may use different iterators and return different results. Additionally consider the logic of lower numbered iterators that may remove or alter a record before it gets to iterator you are expecting. One way to help this situation could be to agree on a number, such as 40, that o determine

Slow queries

If you think queries are running slower than usual, the first thing to look for is hotspots as we have discussed. This is an indication that tablets or data are not balanced and Accumulo is not distributing the workload very well. Other things you might look at:

- Is diskspace filling up on tservers?
- Are there extra large log files? Be sure to check log files for other services running on the node as well, like datanode logs.
- Are scans in the shell slow too? This is indication of system problem instead of problems with your code.
- Contention on the tserver with other Accumulo process like garbage collection, compaction and even map reduce task.

Looking at Zookeeper

Sometimes it is useful to see what is stored in Zookeeper. Extreme caution should be used, as changing data in Zookeeper could cause serious issues for Accumulo. First, you need to know the instance id, which is displayed when you login via the shell. You can also find the instance id by looking in /accumulo/instance_id in hdfs or in the header of the monitory page. Use the zkCli.sh command included with Zookeeper and any of the zookeeper hosts defined in accumulo-site.xml under the instance.zookeeper.host property.

```
/path/to/zkCli.sh -server host:port
```

While in the command line client, use commands like

```
ls /accumulo/<replace with instance_id>
```

to see what Zookeeper has stored. The *get* command will display information about each entry.

Starting with Accumulo 1.4, some of the entries in Zookeeper are protected. Once the cli comes up, use a command like the following to authenticate.

```
addauth digest accumulo:SECRET
```

The SECRET passphrase should be replace with whatever is defined in the instance.secret property in the accumulo-site.xml

The listscans command is very useful

To find out what is happening currently in your cluster, the *listscans* command is very useful. It will show you information about scans running on every tserver. There is a lot of information though, so it is common to dump this information to a file

```
/path/to/accumulo/bin/accumulo shell -u username -p password -e 'listscan -np' > /path/to/dump/fil
```

As a developer, you can add information to what is displayed in the *listscans* command by setting options on any IteratorSettings you add to your scanner. This can be very useful for debugging long running queries.

Looking at user initiated compactions

When looking for long running processes on your cluster, sometime you may want to see what user initiated compactions are happening. These will show up as FATE operations and you can run the following to see what is currently in progress.

```
/path/to/accumulo/bin/accumulo org.apache.accumulo.server.fate.Admin print
```

To get a sense for these over time, you can loop this command like the following.

```
while true; do
  echo `date`;
  /path/to/accumulo/bin/accumulo org.apache.accumulo.server.fate.Admin print;
  sleep 60;
done
```

Note that these will not show system initiated compactions. For those, you will need to go to the tserver logs for the lines with "Starting MajC".

Read the Accumulo documentation

The Accumulo documentation has a section on troubleshooting. For Accumulo 1.6 you can view a copy at *http://accumulo.apache.org/1.6/accumulo_user_manual.html#_troubleshooting*. This documentation has more troubleshooting tips not covered in this book.

Table Designs

Over the last few years developers have created several table designs that have proven useful for organizing particular types of data for various purposes. Here we gather a few popular table designs. In order to serve large amounts of data to a large number of users, we'll want table designs that can satisfy most client requests with one or a small number of scans over exactly the data that users need.

Single Table Designs

Some applications require looking up data based on a few specific pieces of data most of the time. In these cases it is convenient to identify any hierarchies that may exist in the data and to build a single table that orders the data according to the hierarchy.

For example, if we are writing an application to store messages, such as email, we might have a hierarchy that consists of user accounts, identified by a unique email address, and within a user account we have folders, and each folder contains zero or more email messages.

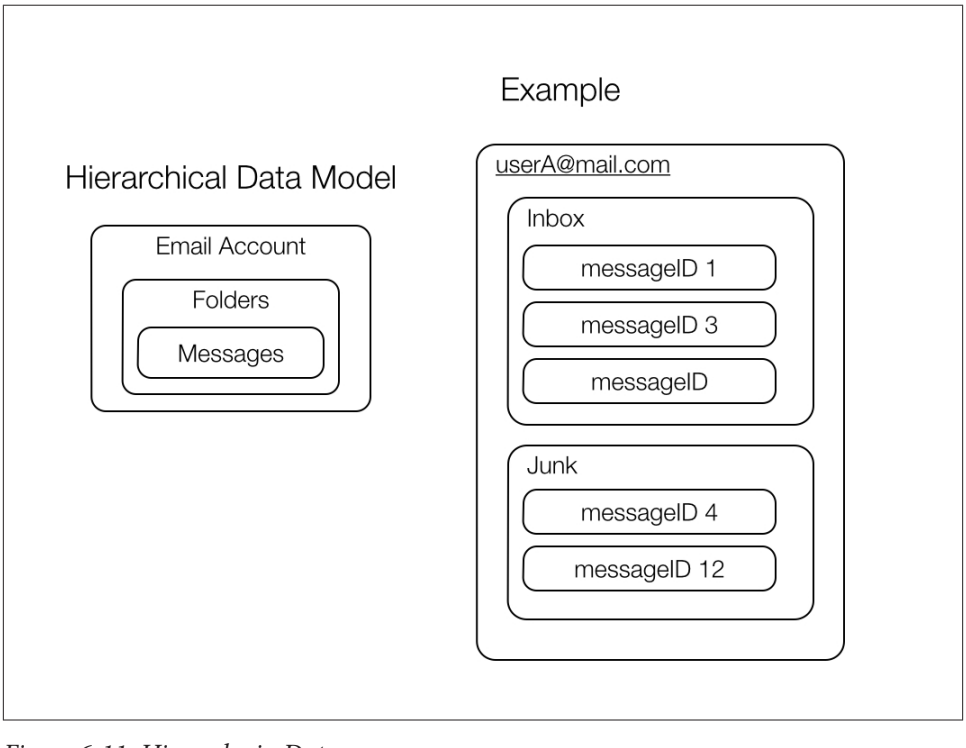

Figure 6-11. Hierarchy in Data

In addition to natural hierarchies in the data, we also need to consider access patterns. A common query will be to access a list of messages to or from a user within a particular folder, preferably in time order from most recent to oldest.

An example application method for fetching this data may look like the following:

```
listMessages(emailAddress, folder, offset, num)
```

Where **emailAddress** is the user's email address, **folder** indicates which mail folder to access, and **offset** and **num** together indicate which set of messages to fetch for the purposes of displaying email addresses in pages. The first page would have an **offset** of 0 and could have a **num** of 100 to show the first (latest) 100 email messages.

To support reading this data efficiently, we could to store all the messages that belong to a user under a row ID consisting of the user's email address, followed by the folder, and finally, the date and time which the email was created or arrived. We may also want to store a unique identifier for this email at the end, to distinguish two messages that may have arrived at the same time. Our row IDs then would look something like this:

```
alice@accumulomail.com_inbox_20110103051745_AFBBE
```

Where *alice@accumulomail.com* is the email address, followed by *inbox*, the folder name, followed by a zero-padded date and time representation that is designed to sort dates properly, followed by some hash of some part of the email or perhaps some ID that is delivered in the email header.

This works except that using the human-readable representation of the data would order our keys in ascending time order, rather than descending as most email applications do. To change this, we can transform the representation of the date in the rowID so that they sort in reverse time order. One way to do this is to subtract the date from a number larger than the largest date we expect to ever store. For example, the date element could be subtracted from the number 99999999999999. We can store the actual date in some value in this row.

We're using an underscore as the delimiter here. A different delimiter may be required depending on whether we ever need to parse the rowID and whether underscores are valid characters for the elements of the rowID.

We then need to determine how to store each part of the message. We may decide to break out the subject and body into different columns so that users can quickly get a list of messages showing the subject without having to read all of the bodies of those messages. Other times, a user will need to retrieve an entire message, including the body and subject. The application method to retrieve all the data for a single message may look like the following:

```
getMessage(emailAddress, folder, date, emailId)
```

So we can have one column family for small amounts of data like the subject, and another column family for the email bodies. This will allow us to store those two column families

in different locality groups, which means we can efficiently read one from disk without reading the other, and other times we can still read them both fairly efficiently.

Now a message in our table may look like this:

```
rowID                                                                                                co
alice@accumulomail.com_inbox_20110103051745_AFBBE       details                subject
alice@accumulomail.com_inbox_20110103051745_AFBBE       details                from
alice@accumulomail.com_inbox_20110103051745_AFBBE       content
```

This one table can now fulfill both types of requests. The implementation of listMessages() would involve creating a single scan such as:

```
public Iterator<Entry<Key,Value>> listMessages(
        String emailAddress,
        String folder,
        Authorizations auths) {

        Scanner scanner = inst.createScanner("emailMessagesTable", auths);

        // we only want to scan over the 'details' column family
        scanner.fetchColumnFamily("details");
        scanner.setRange(Range.prefix(emailAddress + "_" + folder));

        return scanner.iterator();
}
```

Similarly, the implementation of getMessage() would involve creating a single scan such as:

```
public Iterator<Entry<Key,Value>> listMessages(
        String emailAddress,
        String folder,
        String date,
        String emailID,
        Authorizations auths) {

        Scanner scanner = inst.createScanner("emailMessagesTable", auths);
        // we want all column families, and so we don't fetch a particular family
        scanner.setRange(Range.exact(emailAddress + "_" + folder + "_" + date + "_" + emai

        return scanner.iterator();
}
```

In this example, our table exploits natural hierarchies in the data and addresses the two most common access patterns for retrieving information for an application. There are any number of variations on this theme, but a design involving a single table is limited in the number of ways the data can be accessed. For example, this table would not support finding email messages that contain one or more search terms. For those access patterns, additional tables for secondary indexes are necessary.

Time Ordered Data

Reading and writing data in time-order is a common requirement. In the previous example, we ordered email messages in reverse time order within a particular folder belonging to a particular user account. Some applications want to access data primarily in time order, that is the first and most important element of the data is the time component. Examples include time series such as stock data, application logs, and series of events captured by sensors.

We could simply use a timestamp as the row ID of a table. Rows will be sorted in increasing time order and retrieving the data for one timestamp or a range of timestamps is straightforward.

But using a simple timestamp as the row ID of a table can be problematic when it comes to writing the data. This is because often new data arrives with timestamps that only ever increase. If we simply order our data this way, all new data will always be written to the end of the table, specifically to the last tablet which spans some timestamp we've already seen up to positive infinity.

Figure 6-12. Hotspot in time-ordered table

Because each tablet is hosted by exactly one Tablet Server this means that data will always be written to one server. This might be acceptable if one Tablet Server can cope with the rate of incoming data. If not, that last tablet on one Tablet Server will become a bottleneck in the overall ingest process.

To avoid this, some users have resorted to partitioning their table into several buckets and storing ideally uniform partitions of the data in each bucket. One way to do this is to prefix the timestamps with an integer. For example

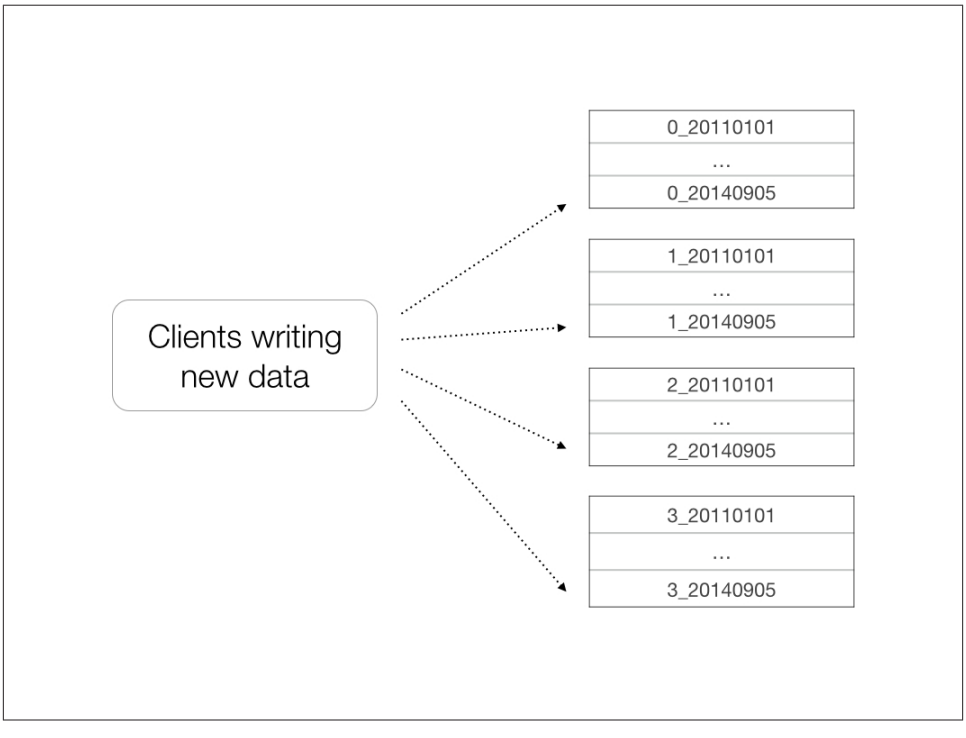

Figure 6-13. Alternative layout using buckets

Then new key-value pairs can be uniformly written to each of these buckets. This allows new data to be ingested in time order into several servers at once. The tradeoff is that each bucket must be queried in order to retrieve all the data for a particular time range. This is especially useful if the intent is to perform MapReduce jobs over time ranges of data.

Drawbacks include the fact that the number of buckets must be chosen ahead of time and that the table will be limited to being hosted by as many servers as there are buckets.

Graphs

Graphs are incredibly useful for representing data for a wide variety of problems. Graphs representing many real-world phenomena may present challenges for some other data storage systems as they can exhibit some difficult properties, depending on the representation. These include sparseness, in which most nodes connect to few other nodes, and skew in the distribution of node degrees, meaning that while most nodes connect to few other nodes, a small number of nodes connect to many other nodes, or even all of them.

Consider a graph that represents words and the documents they appear in. Such a graph could be considered an inverted index of a set of documents, if we were to store the graph by listing pairs of nodes that represent words in sorted order. This graph would exhibit the properties of sparseness and skew as a large number of words would appear in only a few documents and some words, such as *the*, *a*, and *or* would appear in all or almost all of the documents.

Similarly, in a social network, most people will have connections to a number of friends or colleagues likely numbering in the low hundreds, while some famous people will be followed or friended by many thousands or even millions of other users.

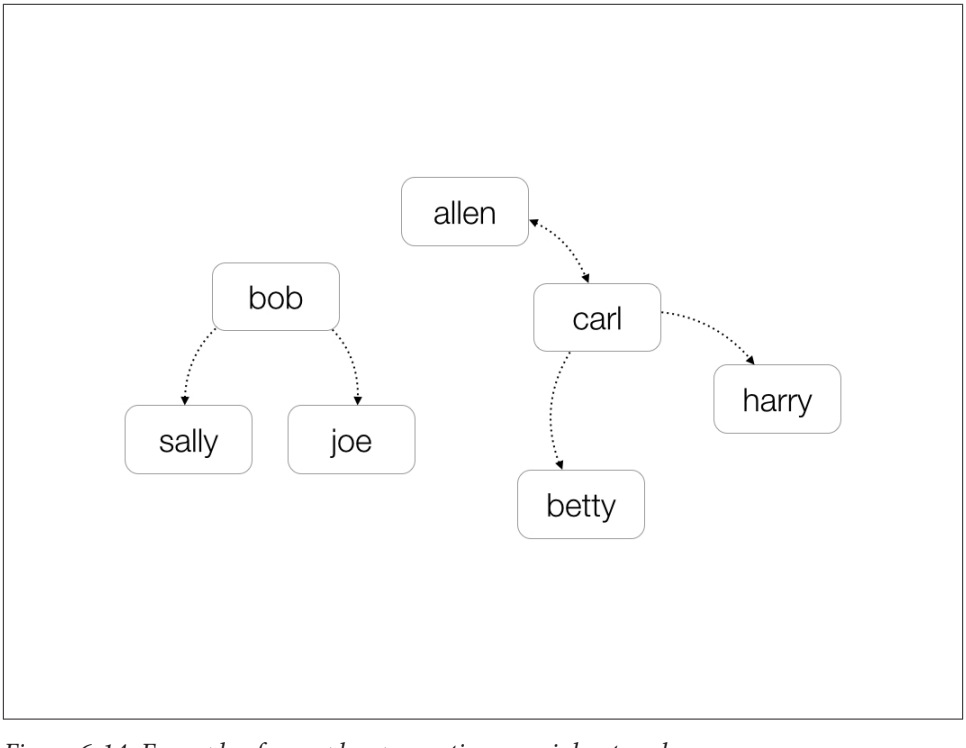

Figure 6-14. Example of a graph representing a social network

Of the two most common ways to represent graphs, as a matrix or as an edge-list, the edge-list representation is best suited to these types of graphs since no storage is required to represent the pairs of nodes that are not connected. Accumulo is especially well suited to handling an edge-list when pairs of nodes are mapped to rows and columns since Accumulo handles sparse rows efficiently, and because rows in Accumulo can be large and are not assumed to fit into the memory of a single machine.

An example of how we might store a graph in Accumulo is as follows:

```
row ID  column family   column qualifier        value
allen   friends         carl
bob     friends         joe
bob     friends         sally
carl    friends         allen
carl    friends         betty
carl    friends         harry
```

We can even store multiple types of edges by using the column family to describe the type of edge:

```
row ID  column family   column qualifier        value
allen   family          jared
allen   family          michael
allen   friends         carl
bob     friends         joe
bob     friends         sally
carl    friends         allen
carl    friends         betty
carl    family          susan
carl    friends         harry
```

This allows applications to efficiently lookup all of the nodes connected to a given node, and to select nodes connected via all or a subset of the types of edges present. Technically, a graph of this type with multiple types of edges is called a *multigraph*. It is possible in a multigraph to have two or more edges of different types between the same two nodes.

Some graphs feature directed edges, in which the relationship between two nodes flows in a particular direction. For example, one user of a social messaging app may follow the status updates of another user but not vice versa. In our table, the first node, represented by the row ID, can signify the originating node and the second node the destination node.

Graph edges may also be weighted in order to represent some feature of the edge such as strength, distance, cost, etc. Accumulo is especially well suited to maintaining large graphs whose edges, nodes, and weights are updated over time. This is because, in addition to the ability to add new columns at will, Accumulo's Iterators offer an extremely efficient way to aggregate contributions to edge weights.

We can choose to store the edge weight in the value of our table and we can configure our table to use an Iterator such as the built in SummingCombiner in order to add up

the weights for each unique row column pair. We may want to store the number of messages two applications users have sent to each other over time. Our table may then look like this:

```
row ID  column family   column qualifier        value
allen   family          jared                          2
allen   family          michael                        55
allen   friends         carl                           6
bob     friends         joe                                    17
bob     friends         sally                          1
carl    friends         allen                          3
carl    friends         betty                          20
carl    family          susan                          4
carl    friends         harry                          2
```

Whenever a user sends another user a message, we can update the graph simply by inserting a new mutation with a count of one:

```
user@accumulo friendGraph> insert allen family jared 1
```

Accumulo will automatically sum up the inserts for each row column pair and return the aggregated result whenever it is read

```
user@accumulo friendGraph> scan -b allen family:jared
allen family:jared [] 3
```

Because Accumulo applies Iterators and scan time and during compactions, *updates* to this graph table are as inexpensive as simple inserts. A large number of individual update/inserts can be sent to this table.

Tables like these have some interesting properties. One is that the number of key-value pairs will reflect the number of edges in the graph. The number of key-value pairs in this table then grows with the number of real-world relationships that exist, but not with the number of individual interactions that are aggregated to represent the weight of these relationships. Also, often graphs built to reflect real-world relationships have many fewer edges than they could have. A graph with N nodes can have up to N^2 edges but the number of edges in many real-world graphs is on the order of N, that is, simply some multiple of N.

So a summarized graph representation of a large number of events will end up having many fewer key-value pairs than there are raw events. In addition, the number of new nodes in real-world graphs often grows relatively slowly over time.

A table design that builds a graph from several different data sources and aggregates the interactions in relationships observed is a powerful method of combining data and allowing natural patterns to emerge. For example, imagine we had a table supporting an analytical application that combined sources of data such as interactions between customers, customers check-ins at store-locations, and products customers have rated. All the information for a particular customers would be organized under one row so

looking up everything that a company knows about that customer can be found quickly by doing a single scan.

Applications can be written to handle the appearance of new columns

Traversing Graph Tables

Tables as described above make it easy to do some types of graph traversal.

Retrieving all of a given node's directly connected neighbors (the one-hop neighbors) is particularly easy. It involves a simple scan of one row of the table and extraction of the connected neighbors by reading the column qualifiers.

Further traversing the graph and retrieving all the neighbors of the one-hop neighbors can be done using a BatchScanner. If we are careful to remove all the neighbors we've already visited from the set of neighbors we pass to the BatchScanner, we can continue to do this until we have visited the entire graph, in breadth-first order. We may run into some limitations around the amount of data we can keep in memory as we do this, but we may elect to store some of the state of our traversal back into the table in Accumulo.

Higher level traversals and algorithms can be built using sequences of scans and batch-scans. Some work has been done at the National Security Agency to process some very large graphs using MapReduce over Accumulo tables.[3]

There are a few other projects designed to process graphs that have been adapted for use with Accumulo, such as those described in the following sections.

Blueprints for Accumulo

Blueprints is part of the TinkerPop graph toolkit.[4] The blueprints-accumulo-graph project, written by Mike Leiberman and hosted on github [5] implements the Blueprints API on Accumulo.

This allows Blueprints-enabled applications to run on Accumulo. The implementation involves a table to store the graph that is similar to the design described above but includes a few more features as well as a separate optional table as an index on vertex and edge properties.

Titan

Titan is "a scalable graph database optimized for storing and querying graphs containing hundreds of billions of vertices and edges distributed across a multi-machine cluster.

3. NSA Big Graph Experiment *http://www.pdl.cmu.edu/SDI/2013/slides/big_graph_nsa_rd_2013_56002v1.pdf*
4. TinkerPop *http://www.tinkerpop.com/*
5. Blueprints on Accumulo *https://github.com/mikelieberman/blueprints-accumulo-graph*

Titan is a transactional database that can support thousands of concurrent users executing complex graph traversals."[6]

A few efforts have been started to begin allowing Accumulo to be used as the backend storage layer, but it is possible some changes in the Titan API may be required before this can be done efficiently.

The most mature effort can be found at *https://github.com/milindparikh/titan-accumulo*.

Semantic Triples

The semantic web and related natural language processing technologies have created significant interest in representing *triples* consisting of a subject, predicate, and object. Such triples may conform the RDF semantic web model.[7] Accumulo can be used to store and access triples efficiently, and even support higher level query languages such as Sparql.[8]

The general strategy is to store each Subject, Object, and Predicate three times, ordering the triples differently each time so that a scan can be performed to retrieve one, two, or three elements efficiently. One table stores elements in SPO order (Subject, Predicate, Object), another in OSP order, and the third in POS. This way any query for all three elements can be done on any table, a query for any two elements can be done on the table in which those to elements appear first, and a query for a single element can be done on the table in which the element appears first.

For example, to scan for the subject *Luke Skywalker* and the predicate *son of* we would scan the SPO table for all keys that sort after the row ID *Luke Skywalker_son of* and stopping before reaching the row ID *Luke Skywalker_son og*. This would return the keys

```
Luke Skywalker_son of_Darth Vader
Luke Skywalker_son of_Anakin Skywalker
Luke Skywalker_son of_Padme Amidala
```

Similarly, if we wanted to find out all statements in which *trained as a Jedi* is used as a predicate, we could scan the POS table starting at *trained as a Jedi* and stopping before *trained as a Jedj*, which would return

```
trained as a Jedi_Luke Skywalker_Yoda
trained as a Jedi_Obi Wan Kenobi_Qui Gon Jinn
trained as a Jedi_Anakin Skywalker_Obi Wan Kenobi
...
```

6. Titan *http://thinkaurelius.github.io/titan/*
7. *http://www.w3.org/RDF/*
8. Sparql Query Language for RDF - *http://www.w3.org/TR/rdf-sparql-query/*

To find all relationships shared by *Darth Vader* and *Obi Wan Kenobi* we could scan the OSP table from *Obi Wan Kenobi_Darth Vader* to *Obi Wan Kenobi_Darth Vades*.

```
Obi Wan Kenobi_Darth Vader_received Jedi training under
Obi Wan Kenobi_Darth Vader_killed
```

This strategy is described in the paper, "Rya: A Scalable RDF Triple Store for the Clouds"[9] These three tables can be used, along with the appropriate query logic, to satisfy Sparql queries on RDF triples.

This table design is a good example of how multiple orderings of elements can support a wide range of queries.

Spatial Data

In additional to numerical and textual data, Accumulo has been used to store, index, and retrieve spatial data consisting of geographical coordinates. Spatial data is challenging because each point is two dimensional and Accumulo's tables, as well as any data stored in memory or on disk, can only be sorted in one way. Naive implementations might choose to index latitude separately than longitude and use set operations to identify data that falls within a given box on the Earth, but this strategy often requires retrieving and then filtering out many more points than lie within the box.

There are several strategies for storing spatial data in Accumulo that try to avoid heavy reliance on filtering.

Geo-hashing and GeoMesa

GeoMesa is an open source project to support storing spatio-temporal data (a combination of geographical and time based information) in Accumulo. GeoMesa makes use of a technique called *geo-hashing*, which is described as "a binary string in which each character indicates alternating divisions of the global longitude-latitude rectangle"[10]

The GeoMesa project can be found at *http://geomesa.github.io/*. It is also designed to work with GeoServer, an open-source server for sharing geospatial data that can allow many popular mapping applications to connect to it, including Google Earth and ArcGIS.footnote[GeoServer *http://geoserver.org/about/*]

9. Rya: A Scalable RDF Triple Store for the Clouds - *http://sqrrl.com/media/Rya_CloudI20121.pdf*
10. SpatioTemporal Indexing *http://geomesa.github.io/assets/outreach/SpatioTemporalIndexing_IEEEcopyright.pdf*

Space-filling Curves

Space-filling curves are another technique besides geo-hashing for mapping higher dimensional data to a single dimension.[11] They work by imposing an order in which points in the higher dimensional space are visited. Listing these points in order is how these points are written to a one dimensional index.

Any ordering could be used but space-filling curves are designed to maintain good locality after the points are mapped to one dimension. This means the points that are close together in high dimensional space tend to be close together in the one dimensional list. The locality preserving property is useful for indexing data in Accumulo since the goal of our tables is to answer most user requests with one or a small number of scans.

There are several popular space-filling curves that have been used in Accumulo tables. The easiest to implement and understand is the Z-order curve, which visits points in higher dimensional space in a Z-like pattern.[12] Others include the Hilbert Curve [13] and Peano Curve [14] curves.

For two-dimensional geo-spatial data, we simply need to convert the points using the instructions for creating a space-filling curve and then write the transformed points to Accumulo.

For the Z-Order Curve, we may have the decimal-degree coordinates:

```
Lat: 033.11 Lon: 044.22
```

These are transformed into row ID:

```
0034341212
```

When scanning for the points that fall within a user-provided range, we similarly convert the starting and stopping points (i.e. the lower left hand point and upper right hand point) to points on the curve and scan the table to retrieve all the points that lie within that range.

For our Z-Order Curve example, we might query from (30.00, 40.00) to (35.00, 45.00). These query points are translated to X and Y We setup our scan go range from X to Y.

In this example we're using human readable Strings to store our z-order coordinates, but in practice a more compact representation can be used. Here is an example of code that implements the common pattern for scanning tables containing z-order coordinates:

11. Space-Filling Curves: *http://en.wikipedia.org/wiki/Space-filling_curve*
12. Z-Order Curve: *http://en.wikipedia.org/wiki/Z-order_(curve*)
13. Hilbert Curve: *http://en.wikipedia.org/wiki/Hilbert_curve*
14. Peano Curve: *http://en.wikipedia.org/wiki/Peano_curve*

```
Double startLat = 30.0;
Double stopLat = 35.0;
Double startLon = 40.0;
Double stopLon = 45.0;

// assuming we've stored our points as human-readable strings
String startZPoint = "0034000000";
String stopZPoint = "0034550000";

Range range = new Range(startZPoint, stopZPoint);
Scanner scanner = new Scanner("myGeoTable");
scanner.setRange(range);

ArrayList<Value> results = new ArrayList<>();

for(Entry<Key,Value> entry : scanner) {
       // unscramble the point returned
   String zPoint = entry.getKey().getRow().toString();
       StringBuilder latStr = new StringBuilder();
       StringBuilder lonStr = new StringBuilder();

       for(int i=0; i < zPoint.length(); i+=2) {
              latStr.append(zPoint.charAt(i));
              lonStr.append(zPoint.charAt(i+1));
       }

       Double lat = Double.parse(latStr.build());
       Double lon = Double.parse(lonStr.build());

       // filter points outside our box
       if(lat >= startLat && lat <= stopLat
          && lon >= startLon && lon <= stopLon) {

              results.add(entry.getValue());
       }
}
```

We are guaranteed to receive all points within the requested area, but we may get points that lie outside the requested range. These points can be filtered out, either by the client or by a filtering Iterator configured on the table.

The Z-order curve will sometimes return a large number of points outside the requested area, particularly around the middle of the indexed area. Other curves can perform better but at the cost of increased complexity of implementation.

Multi-dimensional Data

Some of the strategies for storing two dimensional spatial data in Accumulo can be generalized to store higher dimensional data. For example, in addition to latitude and longitude, an index can be built that stores altitude and other numerical data as well. Textual data can be combined with numerical data as well in these indexes.

For example, to store points consisting of latitude, longitude, and altitude, we may choose to zero pad our points and combine them using the Z-Order Curve:

```
Lat: 00033.11 Lon: 00044.22 Altitude (feet): 11555.00
```

Which would be combined in the Z-Curve to produce the point:

```
001001005345345120120
```

To query for a cube in this space we pick two three-dimensional corners of the cube on opposite sides, transform these points as before and perform a scan, filtering out any points that fall outside the cube.

Secondary Indexes

When storing records in an Accumulo table, we can store them in sorted order, but can only sort them one way. In a previous example we stored emails in order of email address of the recipient, then by the date, and finally by a unique email ID. In this case the *record ID* used is a concatenation of those three elements. If we want to lookup records based on other criteria, we must scan the entire table.

Secondary indexes are tables that allow users to quickly identify the record IDs containing values from any field. Those record IDs can then be used to retrieve the full record from the original table. We discuss two types of secondary indexes, a *term-partitioned* index and a *document-partitioned* index in the Writing Applications Chapter ???

The primary issue facing applications that wish to employ secondary indexing is consistency. Accumulo provides no support for multi-row or multi-table transactions so applications are responsible for ensuring the consistency of secondary indices.

Typically this can be handled in several ways, depending on the mutability of the original record table that secondary indexes reference. If there are no updates to records, then consistency between the original records and secondary indices can be achieved by first committing writes to the record table, and then committing index entries to secondary index tables. This way, records must exist in the record table before any application attempts to follow a reference from an index entry.

Similarly, any records to be removed from the record table should first have their corresponding index entries be removed from any secondary index tables, ensuring that at no time are there index entries that reference records that do not exist.

If there are updates to a record table, the situation becomes more complicated. In this situation, using consistent timestamps for a record and its index entries may help distinguish

D4M and Matlab

D4M stands for *Dynamic Distributed Dimensional Data Model* and was developed by Dr. Jeremy Kepner and others at MIT Lincoln Labs to take advantage of data stores like Accumulo.[15] It has been used for a wide variety of applications, and has been shown to have extremely good performance characteristics[16]. D4M is a technique for applying linear algebra to databases that simplifies the creation of new analytics using parallel computation tools such as pMatlab (Parallel Matlab Toolkit), MatlabMPI, and gridMatlab.

The Accumulo tables used by D4M involve storing associative arrays in a table and also storing its logical transpose so that rows and columns can both be searched efficiently.

The D4M schema as used with Accumulo consists of four tables [17]:

If we ingest records, each consisting of a unique ID and a set of fields and values, the D4M Schema can be described as follows:

Table 6-1. TedgeTxt

row ID	column	value
record ID	field	full text

The TEdgeTxt table is used to store any original text found in records. It is used to retrieve the full text of any records found searching other tables, or to simply retrieve records by ID if the ID known.

Table 6-2. Tedge

row ID	column	value
record ID	field|value	1

The Tedge table is used to store any metadata found in records. It can also store tokenized words from the text in the TedgeTxt table.

Table 6-3. TedgeT

row ID	column	value
field|value	record ID	1

15. *http://www.mit.edu/~kepner/D4M/*
16. "Achieving 100,000,000 database inserts per second using Accumulo and D4M" *http://arxiv.org/ftp/arxiv/papers/1406/1406.4923.pdf*
17. The D4M schema is fully described in the paper "D4M 2.0 Schema: A General Purpose High Performance Schema for the Accumulo Database" *http://www.mit.edu/~kepner/pubs/D4Mschema_HPEC2013_Paper.pdf*

The transpose of the Tedge table is stored in the TedgeT table and is used to lookup record IDs by the values of specific fields.

Table 6-4. TedgeDeg

row ID	column	value
field|value	"degree"	count

The TedgeDeg table stores a count of the number of times a field-value pair has been seen. It can be used to answer simple questions and to provide statistics for queries involving more than one field-value pair.

Linear algebra operations can then be performed by fetching arrays using either the row oriented or column oriented tables and by executing operations in parallel using one of the parallel Matlab frameworks mentioned.

The D4M software can be obtained at from *http://www.mit.edu/~kepner/D4M/gpl.html*. The zip files there include an 8-lecture course on how to use D4M.

D4M Example

We'll walk through an example.

First we'll startup an Accumulo instance running locally. For testing we'll just start up a MiniAccumuloCluster using the accumulo-quickstart project from ???.

After cloning the accumulo-quickstart git project, start up a shell via:

```
mvn clean compile exec:exec -Pshell
```

This will start a shell and a MiniAccumuloCluster that we can use just for the duration of this example. This program will generate a different ZooKeeper port each time it runs, so we'll take note of the ZooKeeper port that is printed out and use it in the following examples to connect to this instance:

```
...
 ---- Initializing Accumulo Shell

Starting the MiniAccumuloCluster in /var/folders/...
Zookeeper is localhost:25641
Instance is miniInstance

Shell - Apache Accumulo Interactive Shell
...
```

We'll leave the shell running in a terminal window while we run these other steps.

Adding D4M to Octave or Matlab

To load and analyze data we'll need GNU Octave or Matlab. Octave is an open source high-level computing language similar to Matlab. Octave can be obtained at *http://www.gnu.org/software/octave/download.html*

Make sure you have a version of Octave with the Java interface. If you see the error "error: javaObject: Octave was not compiled with Java interface"" follow the instructions at *http://wiki.octave.org/Java_package#How_to_install_the_java_package_in_Octave.3F*.

Next we'll download the D4M libraries.

```
$ wget http://www.mit.edu/~kepner/D4M/d4m_api_2.5.1.zip
$ wget http://www.mit.edu/~kepner/D4M/libext_2.5.1.zip
```

Unzip both and place the libext folder inside the d4m_api folder. This folder will be our D4M_HOME.

```
$ unzip d4m_api_2.5.1.zip
$ unzip libext_2.5.1.zip
$ mv libext d4m_api
```

We'll also need to create a file called classpath.txt in our home directory so Octave can find the Java classes D4M needs. Add one line for each jar file in d4m_api/lib and d4m_api/libext

classpath.txt.

```
/home/user/d4m_api/libext/accumulo-core-1.5.0.jar
/home/user/d4m_api/libext/commons-jci-core-1.0.jar
/home/user/d4m_api/libext/jtds-1.2.5.jar
/home/user/d4m_api/libext/accumulo-fate-1.5.0.jar
/home/user/d4m_api/libext/commons-jci-fam-1.0.jar
/home/user/d4m_api/libext/libthrift-0.9.1.jar
/home/user/d4m_api/libext/accumulo-server-1.5.0.jar
/home/user/d4m_api/libext/commons-lang-2.4.jar
/home/user/d4m_api/libext/log4j-1.2.15.jar
/home/user/d4m_api/libext/accumulo-trace-1.5.0.jar
/home/user/d4m_api/libext/commons-logging-1.0.4.jar
/home/user/d4m_api/libext/slf4j-api-1.6.1.jar
/home/user/d4m_api/libext/commons-collections-3.2.jar
/home/user/d4m_api/libext/hadoop-core-1.1.2.jar
/home/user/d4m_api/libext/slf4j-log4j12-1.6.1.jar
/home/user/d4m_api/libext/commons-configuration-1.5.jar
/home/user/d4m_api/libext/hadoop-tools-1.1.2.jar
/home/user/d4m_api/libext/zookeeper-3.3.5.jar
/home/user/d4m_api/libext/commons-io-1.4.jar
/home/user/d4m_api/libext/json.jar
/home/user/d4m_api/lib/D4M_API_JAVA_AC.jar
/home/user/d4m_api/lib/D4M_API_JAVA.jar
```

Loading Example Data

We'll use an example of ingesting data via Matlab scripts from *https://github.com/denine99/d4mBB/* Clone this git repo

```
$ git clone https://github.com/denine99/d4mBB/

$ cd d4mBB
```

Edit the d4m_Baseball_Demo.m script with the information for our MiniAccumulo instance. The top of the file should look like this:

```
%% Demonstrates D4M's capabilities on a small Baseball statistic data set by
% (1) Parsing data into a form ready for ingestion
% (2) Ingesting data into memory or an Accumulo Table
% (3) Querying data to answer several questions of interest

% User Parameters:
doDB = 1;   % Use an Accumulo Database instead of in-memory Associative Arrays
DB = DBserver('localhost:[your-zookeeper-port]','Accumulo','miniInstance','root','pass1234');
```

This shows us how to connect to Accumulo from Matlab or Octave.

Now we'll start up Matlab or Octave and add the D4M libraries to our path.

```
[user@hostname d4mBB]$ octave
GNU Octave, version 3.8.1
...
octave:1> addpath('../d4m_api/matlab_src')
```

Next we'll run the demo script which will load data from a CSV file into Accumulo tables and perform some commands to fetch the data.

This script loads data into memory to organize the information and then writes to Accumulo. If data does not fit in memory it can be loaded into memory and written to Accumulo in parts.

```
octave:2> d4m_Baseball_Demo
INGEST time (sec) = 0.731
INGEST time (sec) = 0.494
...
INGEST time (sec) = 0.147
INGEST time (sec) = 0.094
INGEST time (sec) = 0.064
Creating baseballMaster in localhost:12953 Accumulo
Creating baseballMasterT in localhost:12953 Accumulo
Creating baseballMasterDeg in localhost:12953 Accumulo
Creating baseballSalaries in localhost:12953 Accumulo
Creating baseballSalariesT in localhost:12953 Accumulo
Creating baseballSalariesDeg in localhost:12953 Accumulo
Creating baseballMaster in localhost:12953 Accumulo
```

```
Creating baseballMasterT in localhost:12953 Accumulo
Creating baseballMasterDeg in localhost:12953 Accumulo
Creating baseballSalaries in localhost:12953 Accumulo
Creating baseballSalariesT in localhost:12953 Accumulo
Creating baseballSalariesDeg in localhost:12953 Accumulo
...
```

This script will run a series of queries and display the results in a paginated screen. Hit q to exit that screen.

Let's take a look at what the script is doing. The D4M API uses the concept of a table and a table-pair. The script uses the following commands to create tables and table pairs in Accumulo

```
Tm = DB('baseballMaster', 'baseballMasterT');
Tmd = DB('baseballMasterDeg');
```

Next the demo script performs some queries. The first will "Find all stored information about a specific player: zobribe01 (Ben Zobrist)"

```
octave:3> Tm('playerID|zobribe01,',:)
(playerID|zobribe01,bats|B)     1
(playerID|zobribe01,birthCountry|USA)     1
(playerID|zobribe01,birthState|IL)     1
(playerID|zobribe01,birthYear|1981)     1
(playerID|zobribe01,height|75)     1
(playerID|zobribe01,nameFirst|Ben)     1
(playerID|zobribe01,nameLast|Zobrist)     1
(playerID|zobribe01,weight|200)     1
```

A few lines down we find another query to "Find how many players weigh < 200 lb. and bat with left hand or both hands" First the script finds out how many players meet each criterion separately.

```
octave:4> A = sum(str2num(Tmd('weight|000,:,weight|199,',:)),1);
octave:5> B = str2num(Tmd('bats|L,bats|B,',:));
octave:6> A
(1,degree)     13182
octave:7> B
(bats|B,degree)     1106
(bats|L,degree)     4629
```

Now the script will find those that meet both criteria. "A > B, so we will first query for all the rows of players that bat L or B" "Then, within those rows, we will find the players that weigh < 200 lb."

```
octave:8> A_LB = Tm(:,'bats|L,bats|B,');
octave:9> A_LB_all = Tm(Row(A_LB),:);
octave:10> A_LB_light = A_LB_all(:,'weight|000,:,weight|199,');
octave:11> NumStr(Row(A_LB_light))
ans =  4463
```

Help reference for the D4M Matlab scripts is available via:

```
>> help D4M
```

The D4M libraries ship with some built-in examples in D4M_HOME/examples/

Load Example Data Using Java

To load some example data using Java, we'll use the D4M_Schema project:

```
git clone https://github.com/medined/D4M_Schema.git
```

Open up the D4M_Schema/schema project in Netbeans or Eclipse. Edit the file in src/main/resources/d4m.properties to contain the following settings:

```
accumulo.instance.name=miniInstance
accumulo.zookeeper.ensemble=localhost:[your-zookeeper-port]
accumulo.user=root
accumulo.password=pass1234
```

Run the file com.codebits.example.d4m.TaxYear2007ToAccumulo.java.

This will create the D4M tables and load a CSV file containing tax information into those tables. This code does not use any libraries from the previous D4M example, it simply knows how to create tables of the structure that the D4M API understands.

These tables can be used to perform queries in Matlab as in the above examples and can also be seen in the MiniInstance shell:

```
root@miniInstance> tables
!METADATA
Tedge
TedgeDegree
TedgeMetadata
TedgeText
TedgeTranspose

root@miniInstance> table TedgeDegree

root@miniInstance TedgeDegree> scan
adjusted gross income (in thousands)|$-1 :degree []    1
adjusted gross income (in thousands)|$10007290 :degree []    1
adjusted gross income (in thousands)|$100192 :degree []    1
adjusted gross income (in thousands)|$1002480 :degree []    1
adjusted gross income (in thousands)|$10025 :degree []    1
adjusted gross income (in thousands)|$100273 :degree []    1
...
```

After running these examples, quitting the MiniAccumulo shell program will shutdown the Mini Accumulo instance and erase the data used.

These examples use a single machine to process data retrieved from Accumulo. It is possible to perform these operations in parallel using the Parallel Matlab Toolbox from *http://www.ll.mit.edu/mission/cybersec/softwaretools/pmatlab/pmatlab.html*. This will enable these analytics to scale along with the size of the data by using multiple machines.

Designs for Machine Learning

Large scale data storage and retrieval are two challenges that are directly addressed by Accumulo. But often simply storing raw data and delivering subsets to users as the results of queries can still be overwhelming, if even subsets consist of more results than users can understand.

Data mining, knowledge discovery, and machine learning techniques help address the problem of transforming an overwhelming amount of raw data into higher level representations that are more amenable for making decisions. Because Accumulo is often a part of decision processes based on big data, some techniques for doing machine learning in Accumulo have emerged.

Storing Feature Vectors

Some techniques for storing graphs we mentioned earlier can be used in machine learning methods. Machine learning often involves the construction of *feature vectors* which are used to describe entities of interest. For example, users of a video rental application may be the entities that are described by feature vectors consisting of all the movies they have ever watched or rated. A feature can be any measure that might be informative for determining a property of interest about an entity, such as whether a user might want to watch a new movie, or whether a particular location might be a good place to build a new store, etc. Features may be binary, either an entity has a feature or it doesn't, or features may be weighted with some real number. The construction of features is not always obvious and often involves the application of domain expertise to help come up with likely useful features.

An example feature vector

```
actionMoviesWatched: 28
dramasWatched: 3
...
raidersOfLostArkRating: 4
flightOfTheNavigatorRating: 3
...
```

A feature vector consists of all the features that apply to a particular entity. These vectors may be very large (i.e. high dimensional) and may vary widely from one entity to another. For this reason, storing feature vectors can be challenging for many traditional databases. Accumulo is particularly well-suited to storing a feature vector as set of columns in a row because of its support for dynamic columns, large rows, and sparse rows.

In addition, Accumulo Iterators can be used to efficiently increment feature weights as new raw observations arrive. For example each time a user watches a film we might add a new feature describing the fact that the user watched this movie, as well as incrementing the weight for the number of films the user has watched in the genre of this movie:

```
Movie:
  Casablanca

Features:
  watchedCasablanca: true
  dramasWatched: +1
```

Using an Iterator to update the feature *dramasWatched* allows us to increment the value without reading out the old value first.

We can store an entity's feature vector in a single row in an Accumulo table. The row ID is the name of the entity, the columns are named after the feature, and the value can be either ignored when the presence of the column is taken to mean that the entity has that feature, and absent columns are taken to mean the entity doesn't, or the value can be used to store the weight of the feature.

```
user123        watchedCasablanca       -
user123        dramasWatched           3
```

We might also elect to put different types of features into their own locality groups so that we can efficiently select different types of features in case we don't always want to use them all.

Once we've built a table to store feature vectors we might also want to create another table that stores the names of features in the row ID and the names of entities in columns so we can quickly look up all the entities that have a particular feature.

```
watchedCasablanca      user123
watchedCasablanca      user234
watchedGattaca         user123
```

This table can be considered an index on features.

A Machine Learning Example

These tables make it possible to perform a variety of machine learning tasks. For example, there is a classifier called the K Nearest Neighbors classifier that works by finding some number, k, of entities in the database that are most similar or nearest to an entity of interest based on a similarity metric that compares to feature vectors. These entities are the nearest neighbors. Once the k nearest neighbors are found, we can get an estimate for some property of the entity of interest by averaging the value of that property across the nearest neighbors. K nearest neighbors is an example of a non-parametric method, or instance-based learning since it requires storing information about every previous instance seen, rather than building a representative model. Accumulo is well suited to such a task because of its scalability.

For example, if we want to estimate the rating that a new user, user456, will give to the movie Casablanca, having not watched it yet, we can first find the seven or so entities in our database that are most similar to user456, based on comparing their feature

vectors to user456's feature vector, and average the rating that those users gave to the movie Casablanca to come up with our estimate.

We can carry this out by first fetching user456's feature vector using a single scan. Once we have these features, we can quickly find all the other users that share at least one feature in common with user456 by using a BatchScanner to retrieve the list of users associated with each feature from the table containing our features index. Some features may be very common and may not contribute much to the overall answer to our question, so some tuning may be required to remove features that are not very informative and that would cause us to end up retrieving a large number of potentially similar users.

Now we have a list of candidate nearest neighbors. We can fetch each user's feature vector again using a BatchScanner and use a similarity metric to calculate the similarity of each user's feature vector to user456's, keeping track of the most similar seven users as we go. A good similarity metric for our purposes might be _cosine similarity_[18] or perhaps _tanimoto distance_[19]

If the average rating that the top seven most similar users gave to Casablanca was 3.5 stars out of 4, we might elect then to recommend to user456 that she watch the movie Casablanca.

These scan operations are designed to be fast enough to return answers in time for some interactive applications. Depending on the data set, features will likely need to be tuned to minimize the number of candidates examined at query time. We could also decide to simply update this table throughout the day and at night pre-calculate a list of movies to recommend to each user using a MapReduce job, storing this list of recommendations in Accumulo for fast lookup as users visit our web application.

Similar operations can be used to cluster users into natural groups, again using similarity metrics to compare feature vectors. Other types classifiers can also be trained on these feature vectors to produce models appropriate for scoring new instances in real time to predict class or some value of interest.

In machine learning applications in which Accumulo is already a part of the architecture, it could be used to provide low-latency, high-throughput access to static global resources, even though some other more lightweight distributed key value stores may perform better.[20] In this case Accumulo could be used to store and update model probabilities, especially when the model is larger than fits comfortably in memory even in a distributed cache such as memcached. Accumulo may be more suitable than some other key-value stores as the use of Iterators could make incrementing model weights very efficient, as

18. Cosine Similarity: *http://en.wikipedia.org/wiki/Cosine_similarity*

19. Tanimoto Distance: *http://en.wikipedia.org/wiki/Jaccard_index*

20. Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework, Jimmy Lin et al. *http://hcil2.cs.umd.edu/trs/2009-01/2009-01.pdf*

long as the application is more interested in incrementing weights more often than reading weights. If there is any locality associated with weights, it could be exploited to write and scan weights in batches, resulting in better throughput than reading and writing each individually.

Approximating Relational and SQL Database Properties

Accumulo is a non-relational database, meaning, it doesn't provide built-in support for joins, cross-row or cross-table transactions, or referential integrity. Similarly, Accumulo does not implement Structured Query Language (SQL) operations over tables. However, Accumulo can be made to behave more like a relational or SQL database if desired.

Schema Constraints

One thing that some other databases do is to enforce a schema on data inserted. If a particular row doesn't conform to the specified schema of a table, the insert fails. In contrast, Accumulo is designed to support building tables with widely varying structure across rows. The set of columns and value types in a table are defined only by the actual data. This requires applications to handle missing columns and values that may be of any type or size. In practice the code used to insert data is therefore more closely coupled with the code used to query the data.

By requiring that inserts conform to a particular schema, applications may be less closely coupled and can be simplified. Accumulo can be made to apply Constraints to tables that are examined at insert time. Users can write a Constraint that requires each column name to be one of a specified set, and can also apply type and size limitations to Values stored. Mutations that fail these constraints will be rejected.

See the section on Constraints in the Writing Applications chapter ???.

SQL Operations

Although Accumulo doesn't support SQL, there are a few operations that are trivial to implement using the Accumulo API.

SELECT

Also known as *projection*, the SQL *SELECT* clause is used to select a set of columns from a table. By default, Scanners retrieve all available columns, even those the user may not know about. Scanners can be configured to retrieve only a particular set of columns using the **fetchColumn** and **fetchColumnFamily** methods.

WHERE

Also known as *selection*, the SQL *WHERE* clause is used to select a subset of rows from a table. Accumulo Scanners can be configured to scan over only a subset of rows by

specifying a range of row IDs. Similarly, applications can use secondary indexes to identify sets of row IDs that can be combined using set operations to identify a final set of row IDs that satisfy the logic of a WHERE clause. This set of row IDs can be passed to a BatchScanner to retrieve the full rows from a record table.

Projection and Selection can be combined by calling the **fetchColumn** or **fetchColumnFamily** methods on the BatchScanner used to retrieve the final set of original records.

Most relational databases generate statistics about indices and data in tables as the data is inserted. These statistics are used by query planners to optimize data retrieval. For example, if we were querying for records containing the value *book* in one field and the values $10 in another field, we could consult the statistics table to see which value appears less frequently. If the value *book* appears in only a small number of records and the value $10 appears in many, we can plan our query out by by first fetching those record IDs that contain the value *book*, and then filtering that subset to find records that match the second criterion.

There is no reason that similar statistics could not be stored in Accumulo. The SummingCombiner may come in handy for updating counts in a statistics table.

JOIN, GROUP BY, and ORDER BY

Because Accumulo is designed to answer user requests in subsecond times, and to scale to very large amounts of data, the operations that can be performed at query time are limited to those things that can be done very quickly, despite the size of the data. *SELECT* and *WHERE* operations are among those.

JOIN, *GROUP BY*, and *ORDER BY* are more complex and require significant resources to perform on large amounts of data. Recently, several new systems have emerged to aid in performing these operations at scale, namely MPP databases such as Vertica, Greenplum, Asterdata, etc. as well as Cloudera's Impala [21] and Facebook's Presto [22] which are based in part on Google's Dremel [23] and Tenzing [24] projects. These work by performing SQL operations in parallel quickly enough to be interactive.

Accumulo keeps data in tables sorted by keys at all times, but does not sort data on the fly. Users of Accumulo tend to materialize transformations of original data sets in several tables and perform simple lookups on these tables in sub-second times, rather than computing new transformations on the fly while users wait. In general, operations that require sorting and transforming all the data tend to be performed using MapReduce.

21. Cloudera Impala *http://www.cloudera.com/content/cloudera/en/products-and-services/cdh/impala.html*
22. Facebook Presto *http://prestodb.io/*
23. Google Dremel *http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36632.pdf*
24. Google Tenzing *http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/37200.pdf*

Accumulo doesn't provide built-in support for joins, but several strategies for performing joins are described in the Writing Applications chapter ???. *GROUP BY* and *ORDER BY* can be performed often in one MapReduce job.

In some cases, Accumulo can apply the functions typically used in a *GROUP BY* query using Iterators, provided that the data are already sorted in the desired fashion in a table. Accumulo supports applying iterators at scan time, allowing aggregated views of the data to be created on the fly as data are read from disk and sent to clients.

Designing Row IDs

Row IDs are the most powerful elements of the Accumulo data model as they determine the primary sort order of all the data. We've already covered many of the things that should be considered when designing effective row IDs. Here are a few more issues that may arise when designing rows and some methods for addressing them.

Avoiding hotspots

In many table designs, hotspots can arise as the result of uneven distribution of row IDs, often arising from skew in the source data. The worst case example discussed already was that of time-ordered row IDs, which we addressed by introducing a fixed number of bucket IDs as prefixes to spread newly written data across multiple parts of the table. Other examples include frequently appearing items in data sets, such as very frequent words appearing in textual data, highly populated areas in geo-spatial data, and temporal spikes in time series data. These can all cause an inordinate amount of data to be sent to one server, undermining the effectiveness of the distributed system.

Hotspots can involve simply one server being many times busier than all the others, but also can involve contention around individual rows and the creation of very large rows, due to Accumulo's control over concurrent access to each individual row.

In either case, the general approach is to alter the row IDs to either spread them over a larger portion of the table and therefore over a larger number of servers, or to simply break up highly contested rows into multiple rows to eliminate contention and overly large rows. In the case of many writes ending up going to one server, introducing some sort of prefix in front of the row ID can cause the writes to be sent to as many servers as there are unique prefixes. One example of this is the fixed buckets we already discussed.

An example of breaking up contentious rows is to append a suffix to the end of the row ID so that multiple writes to the same data end up going to different row IDs while keeping the rows next to each other, preserving scan order. One example is in indexing the word *the*, which appears more than any other word in English documents. Instead of simply indexing the word *the* one can attach a random suffix to the word *the* thus:

```
the_023012
the_034231
the_323133
the_812500
```

This way multiple writers can still index the word *the* and avoid contention since they are technically writing to different rows. In addition, the tablet containing the word *the* can now be split into multiple tablets hosted on multiple servers, which would not be possible if all instances of the word *the* were indexed into the same row.

Scanners only need to be modified to begin at the first random suffix and end at the last thus:

```
'the_' to 'the`'
```

Other strategies for avoiding hotspots in indexed data involve indexing pairs of words when one word is very frequent, so instead of indexing *the* we would index *the_car*, *the_house* etc. This has the advantage of making it easier to find records containing two words when one word is very frequent while preserving the ability to still retrieve all records containing just the frequent word.

Sometimes, very frequent items are not of interest to an application and can simply be omitted from the index. Lucene and other indexing libraries often employ *stop lists* which contain very frequent words that can be skipped when indexing individual words in a document.

Some users have used Accumulo Iterators to keep track of how many times a term appears in an index using a separate Column Family and cease to store additional terms after seeing a given amount of them. This strategy is useful because it doesn't require knowing the frequent terms before hand, as a stop list does. However, by itself it doesn't prevent clients from continuing to write frequent terms that will be ignored. An index like this could be scanned periodically (perhaps using a MapReduce job) to retrieve only the highly frequent terms for the purpose of creating a stop list that clients can use.

Designing Row IDs for Consistent Updates

Accumulo is designed to split tables into tablets on row boundaries. Tablet Servers will not split a row into two tablets, so each row is fully contained within one tablet. The Accumulo master ensures that exactly one Tablet Server will be responsible for each tablet, and therefore each row. This means that applications can make multiple changes to the data in one row simultaneously, or in database parlance, atomically, meaning that the server will never apply a portion of the changes. If something goes wrong while applying some changes, the mutation will simply fail, the row will revert to the last consistent state, and the client process can try it again.

Applications that need to make updates to several data elements simultaneously can try to use the row construct to gather the data that needs to be changed simultaneously

together under one row ID. An example is perhaps changing all of the elements of a customer address simultaneously so that the address is always valid and not some combination of an old and new address. Sometimes grouping data that needs to be changed under a common row ID is not possible.

An example is an application that needs to transfer amounts of money between accounts. This involves subtracting an amount from one account and adding it to the other account. Either both or none of these actions should succeed. If only one succeeds, money is either created or destroyed. The pair of accounts that needs to be modified is not known before hand and is impractical for use as a row ID.

It is possible to achieve this capability in a system based on BigTable as evidenced by Google's Percolator paper [25], in which they describe an application layer implemented over BigTable that provides distributed atomic updates, or *distributed transactions*.

Also see the section in the Introduction ??? discussing transactions.

Composite Row IDs

When constructing row IDs that consist of multiple elements, it is necessary to use delimiters in order to have rows sorted hierarchically, such that all of the rows that begin with one element are sorted before the first row of the next first place element appears.

For example, if we want to sort data by first name then secondarily by last name, we need a delimiter to ensure that if we read only first names we see them all in order regardless of what last names might follow each first name. Without a delimiter we can run into the following situation:

```
bobanderson
bobbyanderson
bobjones
```

In this case "bob anderson" should be followed by "bob jones", but because we are missing a delimiter, "bobby anderson" appears between them. Using a delimiter we get the desired sort order:

```
bob_anderson
bob_jones
bobby_anderson
```

Because the delimiter, in this case an underscore, sorts before the third *b* in "bobby", all firstnames with four or more letters are sorted after all the appearances of "bob".

An effective delimiter should be a character that sorts before any characters that are likely to appear in the elements of a row ID. the null character "\0" can be used if necessary.

25. *http://static.googleusercontent.com/media/research.google.com/en/us/pubs/archive/36726.pdf*

Composite row IDs may be human readable enough as-is. If not, custom Formatters can be written to make viewing them easier.

Key size

As mentioned in [writing_applications], Accumulo 1.6 has a Constraint added to new tables that the complete Key be under 1MB in size. Keep in mind the complete Key includes the row id, column family, column qualifier, visbility and timestamp. Keeping the Key under 1MB is a best practice for all version of Accumulo.

Typo Project

Typo is a side project developed by one of the primary Accumulo authors, Keith Turner. [26] From the Typo github repository:

"Typo is a simple serialization layer for Accumulo that makes it easy to read and write java objects directly to Accumulo fields. Typo serializes Java types in such a way that the lexicographic sort order corresponds to the object sort order. Typo is not an ORM layer, its purpose is to make it easy to read and write Java objects to the Accumulo key fields and value that sort correctly."

Typo allows users to easily define keys by declaring a list of elements and their types. Typo helps make sure these elements are sorted properly when converted to keys in an Accumulo table.

An example from the Typo project of creating a key consisting of a Long, String, Double, then String:

```
class MyTypo extends Typo<Long,String,Double,String> {
  public MyTypo() {
    super(new LongLexicoder(), new StringLexicoder(), new DoubleLexicoder(), new StringLexicoder(
  }
}
```

In this case the row ID is a Long, the Column Family a String, the Column Qualifier is a Double and the Value is a String. Typo ensures that the resulting keys are sorted in the same order in which the original Java objects would be sorted.

Typo also makes it easy to create custom Formatters.

Designing Values

Values in Accumulo are stored as byte arrays. As such they can store any type of data, but it is up to the application developer to decide how to serialize data to be stored.

26. Typo *https://github.com/keith-turner/typo*

Many applications store Java Strings or other common Java objects. There is no reason however, that more complicated values cannot be stored.

Some developers use custom serialization code to convert their objects to Values. Technologies such as Google's Protocol Buffers [27], Apache Thrift, or Apache Avro [28], have been used to generate code for serializing and deserializing complex structures to byte arrays for storage in Values. Kryo[29] is another good, Java-centric, technology for serializing Java objects extremely quickly.

Iterators can also be made to deserialize and operate on these objects.

Here we present an example using Apache Thrift. Thrift uses an Interface Description Language (IDL) to describe objects and services. The IDL files are then compiled by the Thrift compiler to generate code in whatever languages are desired for implementing servers and clients. The Thrift compiler will generate serialization and deserialization code in a variety of on-the-wire formats for any data structures declared in the IDL files and will generate Remote Procedure Calls (RPCs) for any services defined. Then it is up to the application developer to implement the logic behind the RPCs.

The Thrift compiler can be obtained at *https://thrift.apache.org/download*.

It is possible to implement the client in one language and the server in another. This is a primary advantage to using Thrift.

In our example, we won't create any Thrift services, but will simply use Thrift to define a data structure and generate code to serialize it for storage in an Accumulo table.

Thrift structs and services are written in the Interface Description Language and are stored in a simple text file. We'll design a struct in the Thrift IDL to store information about an order.

```
struct Order {
        1:i64 timestamp
        2:string product
        3:string sku
        4:float amount
        5:optional i32 discount
}
```

In our Order struct, we have 5 elements. The first four are required and the last is optional. The elements are numbered to support the ability to add and remove elements without breaking services that are built against older versions of these structs.

27. Google Protocol Buffers - *https://developers.google.com/protocol-buffers*
28. Apache Avro - *http://avro.apache.org*
29. *https://github.com/EsotericSoftware/kryo*

Next we'll use the Thrift compiler to generate Java classes to serialize and deserialize this struct.

```
laptop:~ cd thrift
laptop:thrift compiler/cpp/thrift -gen java order.thrift
```

This will create a directory called **gen-java** that will contain our Java classes. In our case just one, Order.java. The generated file for even this simple structure is fairly long so we won't include it here.

We can then use our newly generated class to serialize Java objects to byte arrays and back when writing to and reading from Accumulo tables.

```
package com.oreilly.accumulo.examples;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.Map.Entry;
import java.util.UUID;
import org.apache.accumulo.core.client.BatchWriter;
import org.apache.accumulo.core.client.Connector;
import org.apache.accumulo.core.client.MutationsRejectedException;
import org.apache.accumulo.core.client.Scanner;
import org.apache.accumulo.core.client.TableNotFoundException;
import org.apache.accumulo.core.data.Key;
import org.apache.accumulo.core.data.Mutation;
import org.apache.accumulo.core.data.Range;
import org.apache.accumulo.core.data.Value;
import org.apache.accumulo.core.security.Authorizations;
import org.apache.thrift.TException;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TMemoryBuffer;
import org.apache.thrift.transport.TMemoryInputTransport;
import org.apache.hadoop.io.Text;

public class OrderHandler {

  public void takeOrder(
      final long customerID,
      final String product,
      final Double amount,
      final int discount,
      final String sku,
      final BatchWriter writer) throws TException, MutationsRejectedException {

    // fill out the fields of the order object
    Order order = new Order();
    order.timestamp = new Date().getTime();
    order.product = product;
    order.sku = sku;
    order.amount = amount;
```

```
      if (discount > 0) {
        order.discount = discount;
      }

      // we use a TMemoryBuffer as our Thrift transport to write to
      // when serializing
      TMemoryBuffer buffer = new TMemoryBuffer(300);

      // we use the efficient TBinaryProtocol to store a compact
      // representation of this object.
      // other options include TCompactProtocol and TJSONProtocol
      TBinaryProtocol proto = new TBinaryProtocol(buffer);

      // this serialized our structure to the memory buffer
      order.write(proto);

      byte[] bytes = buffer.getArray();

      // we'll store this order under a row identified by the customer ID
      Mutation m = new Mutation(Long.toString(customerID));

      // we generate a UUID based on the bytes of the order to distinguish one order from another
      // in the list of orders for each customer
      m.put("orders", UUID.nameUUIDFromBytes(bytes).toString(), new Value(bytes));
      writer.addMutation(m);
    }

    ...
  }
```

When reading from this table we can use similar code to deserialize a list of Orders from Values found in the *orders* table.

```
  ...
  public class OrderHandler {

    ...

    public List<Order> getOrders(
      final long customerId,
      final Authorizations auths,
      final Connector connector) throws TableNotFoundException, TException {

      // instantiate a scanner to fetch this data from the table
      Scanner scanner = connector.createScanner("orders", auths);

      // create a range to restrict this scanner to read the given customer's info
      scanner.setRange(new Range(Long.toString(customerId)));
      scanner.fetchColumnFamily(new Text("orders"));

      ArrayList<Order> orders = new ArrayList<>();

      for(Entry<Key,Value> entry : scanner) {
```

```
        // use a TMemoryInputTransport to hold serialized bytes
        TMemoryInputTransport input = new TMemoryInputTransport(entry.getValue().get());

        // need to use the same protocol to deserialize as we did to serialize these objects
        TBinaryProtocol proto = new TBinaryProtocol(input);

        Order order = new Order();

        // deserialize the bytes in the protocol to populate fields in the Order object
        order.read(proto);
        orders.add(order);
      }

      return orders;
    }
  }
```

When using an object serialization framework a programmatic object is converted into a byte array and stored as a single Value in a table. This strategy is convenient in cases when the entire object is always retrieved.

When an application requires retrieving only a portion of an object the fields within an object could be mapped to one key-value pair each. The advantage of splitting up the fields of an object into separate key-value pairs is that individual fields can be retrieved without having to retrieve all the fields. Locality groups can be used to further isolate groups of fields that are read together from those that are not read. See the section in Chapter 3 on configuring Locality groups.

Storing Files and Large Values

Accumulo is designed to store structured and semi-structured data. It is not optimized to serve very large values, such as may arise when storing entire files in Accumulo. The practical limit for a Value size depends on available memory, as Accumulo loads several Values into memory at once when servicing client requests.

When storing larger values than what comfortably fits in memory, users typically do one of two things: store the files in HDFS or some other scalable filesystem or blob store such as Amazon's S3, or break up files into smaller sized chunks.

When storing files in an external filesystem or blob store, Accumulo only needs to store a pointer such as a URL to where the actual file can be retrieved from the external store. This has the advantage of allowing users to search and find files using Accumulo, and inherit all the benefits of security and indexing while not having to store that actual data in Accumulo, which frees up resources for just doing lookups.

If users are more interested in retrieving specific parts of files, breaking up files into chunks and storing them in Accumulo may work better, since Accumulo can then provide the chunk of the file requested by the user in one request rather than looking up

the file pointer in Accumulo and going to fetch it from an external system. Files broken up into chunks may still cause problems when retrieving many chunks at once, since they may still overwhelm available memory.

The Accumulo documentation includes an example for storing files as well as some discussion.[30]

Users have contributed some example techniques for doing this *https://github.com/calrissian/accumulo-recipes/blob/master/store/blob-store/*

All of this logic is managed in an application or service layer implemented above Accumulo.

Human Readable vs Binary Values and Formatters

In some cases it is convenient to store values in a format that is readable by humans. For example, debugging becomes easier, and viewing data in the Accumulo shell is possible.

There are cases in which values are stored in human readable form, such as UTF8 Strings, and are converted to binary for operations on the fly, then converted back to human readable values before writing them back to the table. One example is that of storing numbers as Strings in a table configured to sum the numerical values in those Strings. In this case, the Iterator that performs the summation is responsible for converting Strings into Long or Double objects before summing them together, and then converting them back into String objects before outputting them to be sent either to the user or to the disk for storage. The provided SummingCombiner can be configured to do this for Strings, or to simply treat values as Long objects.

Many applications can be made more efficient by using binary values. In this case, however, values are no longer easily read in the Shell. To make debugging and viewing binary values easier, users can create a custom Formatter by implementing org.apache.accumulo.core.util.format.Formatter. This will allow the Shell to display otherwise unreadable keys and values using some human readable representation.

```
package org.apache.accumulo.examples;
/**
 * this is an example formatter that only shows a deserialized value
 * and not the key
 */
public class ExampleFormatter implements Formatter {
       private Iterator<Entry<Key,Value>> iter;
       @Override
       public void initialize(Iterable<Map.Entry<Key, Value>> scanner, boolean includeTimestamps)
              iter = scanner.iterator();
```

30. File System Archive Example *https://accumulo.apache.org/1.5/examples/dirlist.html*

```
        }
        @Override
        public boolean hasNext() {
                return iter.hasNext();
        }
        @Override
        public String next() {
                Entry<Key,Value> n = iter.next();
                byte[] bytes = n.getValue().getBytes();
                // deserialize
                String s = myDeserializationFunction(bytes);
                return s;
        }
        @Override
        public void remove() {
        }
        private String myDeserializationFunction(byte[] bytes) {
                ...
        }
}
```

Formatters can be configured on a per-table basis by setting the table.formatter option. Customer formatters only need to be included on the classpath when running the Shell.

The Shell also makes it easy to configure formatters using the **formatter** command

To add a formatter

```
user@accumulo> table myTable
user@accumulo myTable> scan

user@accumulo> formatter -f org.apache.accumulo.examples.ExampleFormatter -t myTable
user@accumulo> scan
```

To remove a formatter

```
user@accumulo> formatter -r -t myTable
```

Designing Security Labels

Recall that the Accumulo data model allows a security label to be stored as part of each key. Security labels are stored in the part of the key called the Column Visibility.

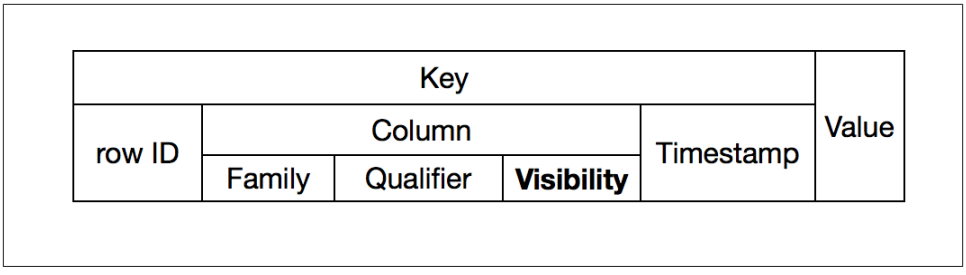

Figure 6-15. Accumulo Data Model

Accumulo's security labels are designed to be flexible to meet a variety of needs. However, this flexibility means that the way to define tokens and combine them into labels isn't always obvious.

There are several things to keep in mind when designing security labels. First is which attributes of the data define the sensitivity thereof.

- Is every record as sensitive as every other?
- Are some fields more sensitive than others?

Second is what requirements exist around accessing the data.

- Do users need training before being able to read particular data elements?
- Is access based on job role?
- How quickly do access control needs change?

Security labels are designed to not be changed often. In fact, it is impossible to actually change the security label stored in the column visibility portion of a key. Rather, users have to delete the old key and write a new key and value with a different column visibility.. The Versioning Iterator does not help us here since two keys that differ only by their column visibility are considered to be different keys by the Versioning Iterator.

Keys that differ only in Column Visibility

The fact that two keys that differ only in column visibility are considered two separate keys means that these keys can co-exist and won't be de-duplicated by the Versioning Iterator. This may be desirable if there are multiple representations of values at different sensitivity levels. A facetious example is that one's age with the column visibility *public* may be 29, when one's *private* age is really 34.

A better application of this concept is perhaps to use representations of values at different resolutions. For example, a person's full address may be more sensitive than just the

address limited to zip code. In practice, the zip code is a different field, and can be labeled separately.

But imagine perhaps a satellite image that may be very high definition and reveal potentially private details such as what is contained in an individual's backyard. We might choose to down-sample this image to a lower resolution to obscure sensitive details and make that representation of the image more widely accessible.

It is of course easy for application designers to choose to store these representations under different columns, but the option for multiple representations of a value that only differ in sensitivity should be understood.

A bigger issue in trying to change security labels is that there may be many billions or trillions of key value pairs, and if regular changes in security labels is required to support changes in access control, many new key value pairs must be written to suppress older versions of the data. For a non trivial amount of data, this is not tractable.

For this reason it is generally recommended to label data with attributes of the data or long standing use cases of the data, using tokens that describe attributes of the data or groups of users that are not likely to change frequently, if ever, and then to assign tokens to individual users in order to grant access. This mapping of users to tokens is always stored in some external system such as an LDAP server. As such, the user-token mappings can be changed rapidly without having to rewrite any data in Accumulo.

Coming up with good tokens

Tokens can really represent any attribute or class of the data or of users. A short example of a token based on the data may be useful.

In many industries some data needs to be stored that represents information that can be used to identify an individual. This kind of data is typically referred to as Personally Identifiable Information, or PII for short. There are guidelines and laws in the United States[31] and other countries around how PII is to be protected. Other fields related to this individual may be less sensitive if the fields containing PII are omitted. Often groups such as analysts and researchers need access to these other fields but not the PII so that they can find relationships and causes in activities and conditions.

Information such as a name, home address, date of birth are just a few of the types of fields that are deemed PII. It could be useful to label data in these fields with the fact that it is considered PII. We could simply define a token called **pii** and require that users possess this token in order to read it. The definition of information considered PII may

31. NIST Guide to Protecting PII *http://csrc.nist.gov/publications/nistpubs/800-122/sp800-122.pdf*

change, but it is not likely to change quickly. The set of users that are authorized to see PII data may change quickly so we keep this mapping in an external system.

Besides attributes of the data to create tokens like PII, a common pattern is to label data based on the general purpose of its existence. Some fields may exist only to express how data travels within the organization, which may be sensitive and is only useful for internal debugging or auditing. This data can be labeled as for internal use only or that it exists only for auditing. We can create tokens for each of these, perhaps **debug** and **audit**.

Finally, it is common to label data based on a well defined role in an organization that represents a group of people that need to work with it. The relationship of data to these groups is often slow changing, though the membership of individuals users in each group is often highly dynamic. Tokens that represent groups such as these may include such things as **administration**, **billing**, or **research** to denote the role that requires access to the data.

When a field has more than one characteristic we can combine these tokens using a boolean operators & or | representing that both tokens are required (logical AND) or that just one or both are required (logical OR), respectively.

Authorizations

It is hard to design ColumnVisiblities without thinking about Authorizations. We have discussed Authorizations in depth, and the best practices for ColumnVisibilities apply. One thing to note if you are upgrading to Accumulo 1.5 or later, the API for Authorizations has changed slightly. The toString(), method no longer calls the serialize() method. The serialize method now Base64 encodes the auths array. Be sure to test these changes thoroughly as you upgrade.

Data Life Cycle

Managing large amounts of data creates new challenges for long running systems as not only may data be large to begin with but over time as data is continually ingested the amount of data under management may get to be extremely large. Accumulo provides some features to help address data life cycle management even when dealing with very large scale data, by leveraging and coordinating resources across the cluster.

Versioning

It may be the case that the data stored in Accumulo is written once and never updated. If not, the first consideration in data life cycle management is how to deal with multiple versions of data. Applications may choose to store one, some, or all versions of data. It can be useful to be able to view previous versions of data to see how data changes over time, for debugging, or simply to make reverting an application to a previous state easier.

Accumulo handles versions by examining the timestamp element of keys that are otherwise identical. That is to say, when two or more keys have the same row ID, column family, column qualifier, and column visibility, the timestamp is the only thing left that can vary between keys. Keys that only vary in timestamp are considered to be different *versions*. Each version has a unique timestamp and potentially different Values. The built-in but optional VersioningIterator can be configured to keep one, some, or all versions.

When attempting to insert a key value pair where the key is identical to an existent key, including the timestamp, Accumulo will only keep one pair and the pair Accumulo chooses to keep is non deterministic. Accumulo is at heart a distributed map of keys to values and each key can have only one value.

The VersioningIterator is configured by default on new tables to key at most one version of a key value pair. This policy can be changed by setting the options

```
table.iterator.majc.vers.opt.maxVersions
table.iterator.minc.vers.opt.maxVersions
table.iterator.scan.vers.opt.maxVersions
```

which apply at major compaction, minor compaction, and scan time respectively.

Whatever the setting, the VersioningIterator will prefer more recent versions over older versions. This means the versions with the highest numbered timestamps.

Data Age-off

Many times data needs to be kept for a certain period of time after which it should be deleted or archived. Accumulo's timestamps and AgeOffIterator support automatically removing data with timestamps beyond a certain date.

Using Time-to-Live

```
setiter -ageoff -majc -minc -scan -n ageoff -p 30
-> set AgeOffFilter parameter negate, default false keeps k/v that pass accept method, true reject
-> set AgeOffFilter parameter ttl, time to live (milliseconds): 10000
-> set AgeOffFilter parameter currentTime, if set, use the given value as the absolute time in mil
```

Using an expiration date

Ensuring Deletes are Removed from Disk

The delete operation is implemented by inserting special delete markers that suppress earlier versions. As a result, deleted data, meaning key-value pairs suppressed by a delete marker, are not immediately removed from disk. Key-value pairs are only really re-

moved during the compaction processes, when data is transferred from memory to a file disk or when two or more files are combined into a merged file.

During the minor compaction process, when data in memory is flushed to a file on disk, any key-value pairs in memory that are suppressed by a delete marker are not written to disk, but the delete marker is. This eliminates some of the deleted data that was in memory. Delete markers are kept around in order to suppress any older versions of key-value pairs that may exist in other files.

During a major compaction when multiple files are combined, any existing versions of key-value pairs that are suppressed by a delete marker found in any of the files as part of the compaction are not written to the newly merged file. This eliminates more of the deleted data, but the delete marker is still written to the newly merged file.

It is only when a tablet server performs a full major compaction, merging all files for a particular tablet into one new file, that it is guaranteed that all deleted data is removed and the delete marker can also be omitted from the newly merged file.

Tablet servers do not typically merge all files for a tablet into one unless there is an idle period. The setting **table.compaction.major.everything.idle** controls how long a tablet server will wait after a table receives a mutation before compacting all of its files into one. But this is not guaranteed to happen. The tablet server may be busy with other tables.

In order to guarantee that all deleted data has been removed from disk at a particular time is to schedule a full major compaction via the Shell:

```
user@accumulo table> compact
```

Merging Tablets

A table's key distribution could change over time, which could even result in empty tablets due to data age off. Even when not using the age-off iterator, over time data may be deleted from tables during the normal course of an application. In this case a table may have empty tablets.

Tablet	Entries	I
[KqZmRN6CGZX5Eji7wwBBfg==]	23.78M	
[duYkPOon7RI52pDWBTbLMQ==]	0	
[9Qgn33cieSCTM8Hcy6AtqQ==]	14.22M	
[lRY/432kqWXN9XMJGOls4A==]	16.80M	
[xUd9e1um5++jeodj4h+ezw==]	0	
[NTt4+crE2Kz9zV6kfSWGLg==]	0	
[UpUnpwlYLpPu43b/5CcNsQ==]	0	
[pS5RGoAecnoVWrr4NQWJmA==]	0	
[/zGeSqJjUmIiMv01ShvARw==]	0	
[uWe7xX8cCnGaCfTtyOA1sA==]	42.47M	
[1M9lDvuTZkD8zygGOIlzxw==]	0	
[LuXDvyIRm7+WT5ei4PK6eQ==]	0	
[V1DWmAgemp4T6FVk1JW4tg==]	0	
[b49XcVCQ2iYyRTmI2aFQGw==]	53.38M	
[k9NvKinX5ItPBqpK3fh6MA==]	0	
[5pXbfe1/SvLWWgXypnl7Zg==]	0	
[1n8WJPMYh9cCVXbj+lhXfQ==]	0	
[MbgMiLt9zMTM9qi+oS3ePQ==]	0	
[ssEoDGtaxsvA63pRhpuBrQ==]	210.53K	
[62nmbgLl7AoQIMg5ufoQZw==]	0	

Figure 6-16. Empty tablets listed in the tablet server view of the Monitor

Empty tablets are not a big problem per se, but they are basically wasted overhead in the Metadata table, and they can also cause balancing to be a bit off, since the default load balancer only looks at tablets per server, and not necessarily key-value pairs per server.

Tablet merging can be used to reduce the number of tablets in the table. In a tablet merge, two or more tablets are combined to form one tablet.

The `merge` command in the Accumulo shell has two modes: merging a range of tablets into a single tablet, or merging consecutive tablets until a given minimum size is reached. To merge tablets to a given size in bytes:

```
user@accumulo> merge -s 100M -t mytable
```

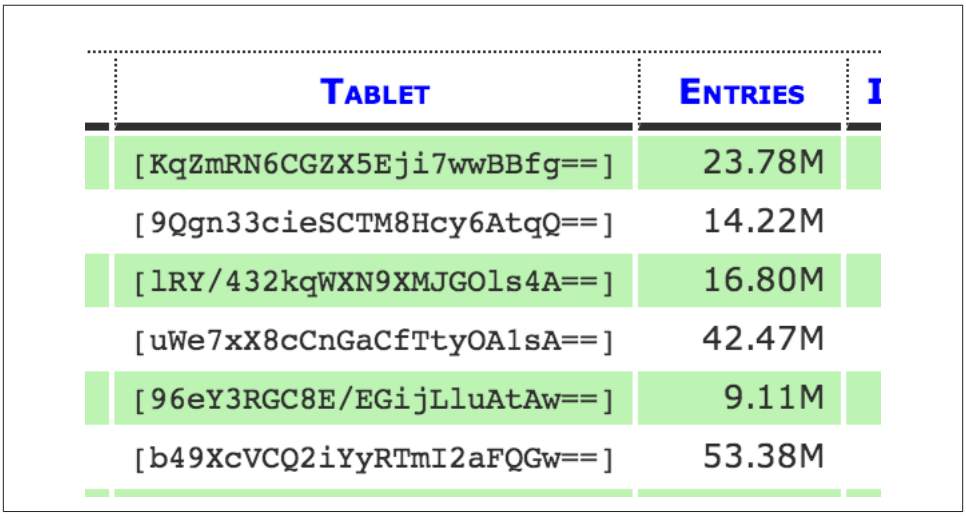

Figure 6-17. Empty tablets are eliminated after merging

To merge tables beginning at a given row and ending at a given row:

```
user@accumulo> merge -b j -e o -t mytable
```

This can also be done via the Java API, although only by specifying a range of tablets to merge:

```
Instance inst = new ZooKeeperInstance(myInstance, zkServers);
Connector conn = inst.getConnector(principal, passwordToken);
conn.tableOperations().merge("mytable", new Text("e"), new Text("o"));
```

Using Compaction

All of Accumulo's files are immutable, and data is only removed during the compaction process, when one or more files are combined, omitting any key-value pairs that fail to

pass the Versioning and AgeOff Iterator tests, and new files are written out. After compaction, the old files containing old key-value pairs that should be removed sit around in HDFS until the Garbage Collector removes them.

One thing to keep in mind is that some files may never be automatically compacted if there is no need to cut down on the number of files per tablet. To ensure that all extraneous versions and all aged-off data is removed administrators can periodically schedule compactions of all files in a table, ensuring that all files are compacted at least once. The **compact** shell command will do this.

```
user@accumulo> compact
```

After a compaction in which key value pairs are removed, the existing split points may no longer be appropriate. For example, the distribution of the data may have shifted and the table may have some tablets that are empty. The **merge** shell command can be used to eliminate empty tablets as described in the above section.

Garbage Collection

With both the VersioningIterator and AgeOffIterator, as well as individual deletes, the Garbage Collection process is ultimately responsible for getting rid of data permanently.

The Garbage Collector works by scanning the Metadata table to determine which files can be removed and issuing delete commands to the HDFS NameNode. The HDFS NameNode must service these calls and all other calls to create, rename, and open files, so Garbage Collection can impose a heavy burden on the NameNode if there are many files to collect.

Storing Dates in the Timestamp Element

Most applications allow Accumulo to assign a timestamp to a mutation when it arrives at a Tablet Server. However, it is possible for an application to specify the timestamp to be used for the key value pairs in a mutation.. Because it is called a *timestamp*, developers may be tempted to store any time related information in this element of the key. However, it is usually a mistake to use timestamps this way, as they are used for determining the order in which key value pairs are applied, including which key value pairs have been deleted, and which should be considered the most current by the VersioningIterator and AgeOffIterator.

Rather, timestamps from source data should most often be stored as values in their own column and are there ignored by TabletServers for the purpose of managing operations on the data, and can be used by applications. Also see the section on Reinsertion and Deletes ??? for ways to avoid issues with user-assigned timestamps.

Compacting a Range within a Table

At times it may be necessary to compact a specific range of a table, rather than the entire table. Tables that store chunks of data in time order may want to treat older sections of the table differently than newer sections, for example. A administrator could alter the major compaction settings temporarily, for example the set of iterators or compression algorithms applied, then issue a **compact** command over a specific range to change the files in that range accordingly. Settings may then be changed back for normal operations.

i. setting iterators for a specific compaction …

Applying Iterators Effectively

Accumulo's Iterators open up many opportunities for interesting use cases. Tasks that may otherwise involve reading out and modifying a lot of data can be handled much more efficiently by using iterators to apply logic asynchronously during compactions, or on the fly at scan time.

Good examples include maintaining a summary of all data seen, performing last minute filtering after ranges and column selection has been applied, or for doing set operations on a small partition of documents as in the IntersectingIterator.

However, Iterators are not a panacea for performing expensive operations that could be better handled by such things as secondary indexes. Performing a full table scan using an iterator to perform filtering is something that will take a long time and that therefore should only be done in one-off situations when secondary indexes are not available. In most cases if queries can't be answered via specifying a range of row IDs, and if such queries are likely to be performed more that once, building a secondary index on other fields is the right solution.

Changing iterators on a single table can be problematic as the logic of the new iterator may not work well with partially applied results from the previous iterator. It may be useful to issue the **compact** shell command on a table before changing iterators to eliminate partial results before switching iterators.

Safely Deploying Custom Iterators

Newly developed Iterators can be challenging to deploy. This is because one is effectively allowing some application logic to be hosted inside TabletServers where bugs can cause problems that affect data. One especially bad scenario is one in which a bug in an iterator causes data to be lost as it is compacted to disk.

To help avoid losing data, or experiencing other issues when deploying new custom iterators, it can be helpful to enable an iterator to be applied at scan-time only, at least at first.

This can be done via the shell as follows:

```
user@accumulo> setiter -class com.company.MyIterator -n myiterator -scan -t myTable -p 40
```

This allows administrators to view the effects of an iterator in a way that isn't permanent. Scanning an entire table with an iterator applied can help expose any bugs that may arise from running real data through the iterator.

If no issues are observed and iterator can be applied at minor and major compaction times as well, making the changes permanent on disk. In the shell this would be done thus:

```
user@accumulo> setiter -class com.company.MyIterator -n myiterator -minc -macj -t myTable -p 40
```

If issues are observed, an iterator can be disabled in the shell via the **deleteiter** command.

```
user@accumulo> deleteiter -n myiterator -minc -majc -scan -t myTable
```

In particular, shutting down a system with a faulty iterator configured to be applied at minor compaction time can cause especial havoc as Tablet Servers will perform minor compactions at shutdown and may prevent safe shutdown. It is preferable to disable any such iterators and attempt to correct the situation while the system is running.

Custom constraints can cause similar issues and can be similarly disabled if issues arise, but these are usually not as egregious as those with iterators.

Using Other Systems in Conjunction with Accumulo

There is a rich and quickly-evolving ecosystem around big data technologies. Many of these other software projects can be used with Accumulo as a part of a broader solution. In this section we touch on best practices for using Accumulo with some of these technologies.

Using Apache Kafka with Accumulo

Apache Kafka is a scalable, fast, distributed queue developed originally at LinkedIn-footnote[*http://kafka.apache.org*]. For this reason it is attractive as part of a larger data workflow as a way to connect different systems together. For example, a variety of applications can be made to publish their data to topics in Kafka and a different set of other systems can be configured to read data from the topics on the Kafka queue. The applications publishing and the applications reading don't have to be configured to talk to one another, just to talk to Kafka.

This was one powerful idea behind the push a few years ago for organizations to move to a Service Oriented Architecture including a centralized queue serving as a message bus for the entire organization. Because Kafka is distributed it is a good candidate for inclusion in a big data workflow.

Accumulo clients can read from Kafka topics and write the data read to Accumulo tables. This provides other applications with the capability to push data to Accumulo tables simply by publishing data to a Kafka topic. Accumulo clients can of course be configured to listen for data pushed from other applications directly, but using a queue allows multiple consumers to read the same data without configuring complicated pipelines.

Kafka provides some guarantees around the data consumed from its topics:

- Messages sent by a producer to a particular topic partition will be appended in the order they are sent. That is, if a message M1 is sent by the same producer as a message M2, and M1 is sent first, then M1 will have a lower offset than M2 and appear earlier in the log.
- A consumer instance sees messages in the order they are stored in the log.
- For a topic with replication factor N, we will tolerate up to N-1 server failures without losing any messages committed to the log.

One thing that is important for many applications is to process each message once, or sometimes at least once. Kafka can partition a topic and allow multiple consumers to be grouped within a common group id in order for the partitions to be consumed in parallel. Each message will be delivered to a consumer group only once. Within the consumer group, individual consumers can tell a broker that they have consumed a message from a particular partition of a topic is by updating an offset:

```
Our topic is divided into a set of totally ordered partitions, each of which is consumed by one co
```

When using Accumulo to store messages read from Kafka one can get closer to achieving this property of writing each message once in the presence of individual machine failures by synchronizing the updating of the Kafka consumer offsets with flushing batches successfully to Accumulo.

An example of code that acts as a Kafka consumer and Accumulo ingest client is as follows:

```
package com.oreilly.accumulo.integration;

import com.google.common.base.Function;
import com.google.common.collect.Iterables;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
```

```
import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
import kafka.message.MessageAndMetadata;

import org.apache.accumulo.core.client.BatchWriter;
import org.apache.accumulo.core.client.BatchWriterConfig;
import org.apache.accumulo.core.client.Connector;
import org.apache.accumulo.core.client.MutationsRejectedException;
import org.apache.accumulo.core.client.TableNotFoundException;
import org.apache.accumulo.core.data.Mutation;

public class KafkaIngestClient {
  private final ProblemMessageSaver saver;

  public interface ProblemMessageSaver {

    void save(final List<byte[]> messages);
  }

  private final KafkaStream<byte[], byte[]> stream;
  private final BatchWriter batchWriter;
  private final int batchFlushSize;
  private final Function<byte[], Mutation> messageConverter;
  private final ConsumerConnector consumerConnector;
  private final ArrayList<byte[]> messageBuffer;

  public KafkaIngestClient(
      final String zookeeper,
      final String consumerGroup,
      final String topic,
      final String table,
      final BatchWriterConfig bwc,
      final Connector conn,
      final int batchFlushSize,
      final Function<byte[], Mutation> messageConverter,
      final ProblemMessageSaver saver) throws TableNotFoundException {

    // create kafka consumer
    Properties props = new Properties();
    props.put("zookeeper.connect", zookeeper);
    props.put("auto.offset.reset", "smallest");
    props.put("autocommit.enable", "false");
    props.put("group.id", consumerGroup);


    ConsumerConfig consumerConfig = new ConsumerConfig(props);
    consumerConnector =
        kafka.consumer.Consumer.createJavaConsumerConnector(consumerConfig);

    Map<String, Integer> topicCountMap = new HashMap<>();
```

```
    topicCountMap.put(topic, new Integer(1));
    Map<String, List<KafkaStream<byte[],byte[]>>> consumerMap =
        consumerConnector.createMessageStreams(topicCountMap);

    stream =  consumerMap.get(topic).get(0);


    // create Accumulo batch writer
    batchWriter = conn.createBatchWriter(table, bwc);
    this.batchFlushSize = batchFlushSize;
    this.messageConverter = messageConverter;
    this.messageBuffer = new ArrayList<>();
    this.saver = saver;
  }

  public void run() {

    for(MessageAndMetadata<byte[], byte[]> mm : stream) {

      byte[] message = mm.message();
      messageBuffer.add(message);

      if(messageBuffer.size() >= batchFlushSize) {
        while (true) {
          try {
            batchWriter.addMutations(Iterables.transform(messageBuffer, messageConverter));
            batchWriter.flush();
            consumerConnector.commitOffsets();
            messageBuffer.clear();
            break;
          } catch (MutationsRejectedException ex) {

            // constraint violations and authorization failures
            // will not be solved simply by retrying
            if (ex.getConstraintViolationSummaries().size() > 0
                || ex.getAuthorizationFailuresMap().size() > 0) {

              // save off these messages for examination and continue
              saver.save(messageBuffer);
              consumerConnector.commitOffsets();
              messageBuffer.clear();
              break;
            }
            // else will retry until success
          }
        }
      }
    }
  }
}
```

In the event that a machine dies some of the messages read from the Kafka queue and batched in memory by the Accumulo client library will yet not have been written to Accumulo. Another machine starting up to take over consumption of the partition the failed machine was reading from (or perhaps an existing client allowed by Kafka to take over consumption of the partition from the failed machine) will start reading messages from the offset of the last message known to have been written to Accumulo successfully, so that no messages will fail to be written to Accumulo.

Since it is possible for the BatchWriter to flush message to Accumulo on its own in the background, this strategy provides *at-least-once* processing semantics, meaning each message will be processed once, or in the case of failure, perhaps more than once. If the mapping of Kafka message to key-value pairs written to Accumulo is deterministic, Accumulo's VersioningIterator can be configured to eliminate any duplicates by keeping only the latest version of a particular row and column within a key. In fact this is the default configuration for all tables in Accumulo.

Using Apache Storm with Accumulo

Apache Storm

```
Storm guarantees every tuple will be fully processed. One of Storm's core mechanisms is the abilit

Storm's basic abstractions provide an at-least-once processing guarantee, the same guarantee you g
```

Language Specific Clients

Accumulo provides an Apache Thrift proxy that enables clients to be written in any language that Thrift supports. For details on running the proxy see the section ???.

A few developers have created language specific client libraries to make it easier to use Accumulo in these languages. This list will likely become out of date very quickly, so we encourage you to search github at *https://github.com/search?o=desc&q=accumulo&ref=searchresults&s=stars&type=Repositories* and follow the Accumulo blog at *https://blogs.apache.org/accumulo/* The following is a list of some of these.

Python

Python is a popular scripting language. A project supporting language bindings for Python can be found at *https://github.com/accumulo/pyaccumulo*

Erlang

Erlang is a functional programming language that is popular for building distributed systems. *https://github.com/chaehb/erlaccumulo*

C++

C++ is a popular high performance object oriented language. *https://github.com/calrissian/accumulo-cpp*

Clojure

Clojure is a functional programming language that runs on the JVM. A project that uses Clojure on top of the Java API can be found at *https://github.com/charlessimpson/clojure-accumulo*

Scala

Scala is another JVM language that combines Object Oriented and Functional Programming. There are a couple of Scala wrappers on the top the Java API that can be evaluated for your use: - *https://github.com/mjwall/scala-accumulo* - *https://github.com/tetra-concepts-llc/accumulo-scala*

Integration with Analytical Tools

Many use cases call for processing data with additional analytical tools on a separate machine outside of Accumulo or on additional processes co-located with Accumulo Tablet Servers.

Pig

Pig is a high-level data processing language that compiles down to a series of MapReduce jobs that can be executed alongside Accumulo Tablet Servers. As of Pig 0.13 Accumulo can be used as Storage[32].

```
my_data = LOAD 'accumulo://table_name?connection_options' USING o.a.p.b.hadoop.accumulo.AccumuloS


STORE my_data INTO 'accumulo://table_name?connection_options' USING o.a.p.b.hadoop.accumulo.Accum
```

R

R is a popular analytical tool that implements a wide variety of statistical algorithms. Some work has gone into integrating R with Accumulo at *https://github.com/DataTacticsCorp/raccumulo*. This adapter makes it possible for R to function as an Accumulo client and to pull data into R for further analysis.

32. *http://pig.apache.org/docs/r0.13.0/func.html#AccumuloStorage*

OpenTSDB

OpenTSDB is a project for storing time series in scalable databases such as Accumulo. An adapter for OpenTSDB onto Accumulo can be found at *https://github.com/ericnewton/accumulo-opentsdb*

Summary

Accumulo has a wide range of features and options for organizing data to support applications. Within this range there is an incredible degree of flexibility. By far the most important decisions when it comes to achieving required performance involve how keys are defined in order to support access patterns efficiently. Accumulo developers have discovered several useful designs for various types of data and common tasks and there is a creative community that is discovering more all the time.

Once the foundation for fast access has been laid, it is well worth it to tune and tweak table and cluster configuration to achieve optimal performance.

APPENDIX A

Shell Commands

APPENDIX B

Configuration Options

APPENDIX C
Metadata Table

The metadata table contains a row for every tablet in Accumulo. Tablets are uniquely described by the ID of their table and the last row in the range assigned to the tablet, or *end row*. Table BC--1 describes the columns that may appear in a tablet's row in the metadata table and Table BC--2 shows some sample entries from a real metadata table.

In addition to tablet entries, there is a section of the metadata table that records file deletion entries. There is also a section for files that are in the process of being bulk imported into Accumulo, to assist the garbage collector in not deleting these files prematurely. More about file deletion can be found in "Garbage Collector" on page 104 on Garbage Collection.

Table BC--1. Metadata table description

Row	Column Family	Column Qualifier	Value
table id ; tablet end row	`file`	*regular data file name*	*size in bytes , number of keys*
table id ; tablet end row	`future`	*tablet server session id*	*tserver IP : port*
table id ; tablet end row	`last`	*tablet server session id*	*tserver IP : port*
table id ; tablet end row	`loc`	*tablet server session id*	*tserver IP : port*
table id ; tablet end row	`log`	*server / log file name*	*log set | table id*
table id ; tablet end row	`scan`	*file currently being scanned*	
table id ; tablet end row	`srv`	`compact`	*compaction id*
table id ; tablet end row	`srv`	`dir`	*tablet directory*
table id ; tablet end row	`srv`	`flush`	*flush id*
table id ; tablet end row	`srv`	`lock`	*zookeeper lock location*
table id ; tablet end row	`srv`	`time`	`M` or `L` *followed by latest time*
table id ; tablet end row	`~tab`	`~pr`	`0x01` *followed by previous tablet's end row*

Row ID

The row ID for a tablet contains the table ID and the tablet end row separated by a semicolon. For the last tablet in a table, there is no end row. The row for that tablet is the table ID followed by `<`.

Rows starting with `~del` are for deletion entries and rows starting with `~blip` are for files that are in the process of being bulk loaded. These entries also contain the name of the file marked for deletion or bulk loading.

There are also entries for problems with loading resources. If the problem involves the metadata table, the information about the problem is written directly to Zookeeper, but problems with other tablets are written to the metadata table. These entries have row ID beginning with `~err` and also containing the table name. The column family is either `FILE_READ`, `FILE_WRITE`, or `TABLET_LOAD`, indicating the type of problem, and the column qualifier is the resource name, which is either a file name or a tablet key extent (prev row and end row). The value contains additional information such as the time the problem occurred, the server, and the exception if available.

`file` column family

This column family contains information about a tablet's files. The column qualifier is the name of the file and the value contains information about the file, its size in bytes and number of keys. Under some conditions these values are estimates. For example, when a tablet is split, the two resulting tablets' file entries will each be assumed to contain about half the bytes and number of keys of the original tablet's files.

The first letter of the file name (the actual file name, not including its path) indicates what type of operation created the file.

F
: minor compaction

C
: major compaction

A
: full major compaction

M
: merging minor compaction

B
: bulk import

`scan` column family

This column family is used to ensure that files are not deleted while they are being scanned. The column qualifies is the name of a file currently being scanned. The garbage collector takes this information into account when determining which files are still in use and which can be safely deleted.

`future`, `last`, and `loc` column families

These column families contain information about where a tablet has been assigned. The `future` column contains the current assignment. The `loc` column contains the current assignment once the tablet has been successfully loaded by the assigned tablet server. The `last` column is the last assignment, used to try to reassign a tablet to the same server to improve data locality.

The column qualifier is the tablet server session ID and the value is the tablet server location, its IP and port. Each tablet server process has a unique session ID, so if the tablet server process is restarted on a machine Accumulo will be able to distinguish between tablets assigned to it before and after it was restarted.

`log` column family

This column family contains information about a tablet's write-ahead log files. The column qualifier is the server name and the log file name separated by a slash. The value is the log set and table ID separated by a pipe. In 1.5.0 and later, the log set is the same as the log file name.

`srv` column family

The `dir` column qualifier has the tablet's main directory as its value. The tablet may use files outside of this directory, but new files will be created in the directory.

The `compact` column qualifier has the most recent compaction ID as its value. The `flush` column qualifier has the most recent flush ID as its value. These IDs are used to determine whether requested flushes or compactions have successfully completed for all relevant tablets.

The `lock` column qualifier contains the Zookeeper lock location for a tablet server that is attempting to write to the metadata table. There is a constraint on the metadata table that only accepts writes from tablet servers with currently held Zookeeper locks.

The `time` column qualifier stores the timestamp of the most recently written data to a tablet. It is preceded by an `M` indicating that the timestamp is in milliseconds since the epoch or an `L` indicating that the timestamp is in logical time (essentially a one-up counter).

`~tab:~pr` column

This column contains the end row of the previous tablet, which helps Accumulo keep track of its metadata. The value is `0x01` followed by previous tablet's end row. For the first tablet in a table, there is no previous tablet, so the value is set to `0x00`.

Other columns

There are a few additional metadata entry types that are ephemeral, such as those written in the process of a tablet split operation. These include a `~tab:oldprevrow` and `~tab:splitRatio` for split operations; `chopped:chopped` for merge operations; `loaded`, for bulk import operations; and `!cloned`, for table clone operations.

Table BC--2. A sample of metadata table contents

Row	Column Family:Column Qualifier	Value
!!~del/!0/table_info/A0001c8l.rf :		
!0;!0<	srv:dir	/root_tablet
!0;!0<	~tab:~pr	\x00
!0;~	file:/table_info/A0001c8q.rf	965,28
!0;~	last:1409c5a89030283	127.0.0.1:9997
!0;~	loc:1409c5a89030283	127.0.0.1:9997
!0;~	srv:compact	10892
!0;~	srv:dir	/table_info
!0;~	srv:flush	10892
!0;~	srv:lock	tservers/127.0.0.1:9997/zlock-0000000000$1409c5a89030283
!0;~	srv:time	L3523
!0;~	~tab:~pr	\x01!0<
!0<	last:1409c5a89030283	127.0.0.1:9997
!0<	loc:1409c5a89030283	127.0.0.1:9997

Row	Column Family:Column Qualifier	Value
!0<	srv:compact	10892
!0<	srv:dir	/default_tablet
!0<	srv:flush	10892
!0<	srv:lock	tservers/127.0.0.1:9997/zlock-0000000000$1409c5a89030283
!0<	srv:time	L4504
!0<	~tab:~pr	\x01~
1<	file:/default_tablet/C0000dj7.rf	12617985,527400
1<	file:/default_tablet/C0000mcw.rf	18999363,790313
1<	file:/default_tablet/C00013vg.rf	25227499,1035563
1<	file:/default_tablet/C000191y.rf	7476173,305642
1<	file:/default_tablet/C0001alv.rf	2239839,91671
1<	file:/default_tablet/C0001boe.rf	1543104,63045
1<	file:/default_tablet/C0001bzk.rf	452015,18523
1<	file:/default_tablet/C0001c5h.rf	146692,5864
1<	file:/default_tablet/F0001c45.rf	95096,3852
1<	file:/default_tablet/F0001c6j.rf	43762,1750
1<	file:/default_tablet/F0001c7w.rf	55019,2206
1<	last:1409c5a89030283	127.0.0.1:9997
1<	loc:1409c5a89030283	127.0.0.1:9997
1<	log:127.0.0.1+9997/30d1970a-3db5-49fc-82d6-8adde36c9453	127.0.0.1+9997/30d1970a-3db5-49fc-82d6-8adde4
1<	srv:compact	0
1<	srv:dir	/default_tablet
1<	srv:flush	0
1<	srv:lock	tservers/127.0.0.1:9997/zlock-0000000000$1409c5a89030283
1<	srv:time	M1380306759481
1<	~tab:~pr	\x00
3<	file:/default_tablet/F0000005.rf	186,1
3<	last:1409c5a89030283	127.0.0.1:9997
3<	loc:1409c5a89030283	127.0.0.1:9997
3<	srv:dir	/default_tablet
3<	srv:flush	1
3<	srv:lock	tservers/127.0.0.1:9997/zlock-0000000000$1409c5a89030283

Row	Column Family:Column Qualifier	Value
3<	srv:time	M1377020908127
3<	~tab:~pr	\x00

APPENDIX D
Data Stored in Zookeeper

Under the `/accumulo` node in Zookeeper, there is a node for each instance of Accumulo keyed by instance ID. There is also an `instances` node that contains a node for each Accumulo instance name, with the data for each instance name being the instance ID currently associated with that name.

Under each instance ID node, there may exist the following nodes.

`masters`, `tservers`, `gc`, `monitor`, and `tracers` nodes

These nodes contain the locations of the various Accumulo processes.

The `masters/lock` node contains an ephemeral sequential master lock whose data is the master location. The `masters/goal_state` node contains the master's goal state (NORMAL, SAFE_MODE, or CLEAN_STOP).

The `tservers/`*`tablet_server_host:port`* nodes contain an ephemeral sequential lock for the specified tablet server. The data for the lock is `TSERV_CLIENT=`*`host:port`*.

The `gc/lock` node contains an ephemeral sequential lock for the garbage collector. The data for the lock is `GC_CLIENT=`*`IP:port`*.

The data for the `monitor` node is the location (IP:port) of the monitor server. There is also a child node `monitor/log4j_port`, whose data is the port of the monitor server used for collecting error logs from other Accumulo processes.

The data for the `tracers/trace-`*`ID`* nodes is the location (IP:port) for the tracer process. The ID is a one-up counter for the tracer processes.

problems/*problem_info* nodes

These nodes are created when a problem has occurred with a resource of the metadata table. Problems with non-metadata resources are stored in the metadata table.

The problem information encoded in the node name includes the table name, problem type (FILE_READ, FILE_WRITE, or TABLET_LOAD), and resource name (either a file name or tablet key extent). The data for the node is additional information about the problem, including the time the problem occurred, the server, and the exception if available.

root_tablet node

This node has children `lastlocation`, `location`, `future_location`, `dir`, and `walogs`. These nodes contain the information that is stored in the metadata table for other tablets: tablet server assignment information, HDFS directory, and write-ahead log files.

tables/*table_id* nodes

These nodes contain the following child nodes.

`state`
: data is the current state of the table (NEW, ONLINE, OFFLINE, or DELETING)

`conf/`*`table_property_name`*
: data is the value for the table property

`flush-id`
: data is the id of the last attempted flush

`compact-id`
: data is the id of the last attempted compaction followed by an encoding of the iterators used for the compaction

`compact-cancel-id`
: data is the id of the last canceled compaction

`name`
: data is the name of the table

config/*system_property_name* node

The data for each of these nodes is the value for the specified system property. These property values override what is configured in the `accumulo-site.xml` file

`users/username` nodes

Accumulo's user authentication and authorization mechanisms are pluggable. The default authentication implementation is a simple username / password system. The usernames are stored as Zookeeper nodes whose data is the hash of the user's password. The default authorization implementation stores the Accumulo users' maximum set of authorizations in a child node of the *username* node in Zookeeper. These authorizations are used along with column visibilities for each key to determine which key / value pairs can be seen by the user.

Other nodes

`hdfs_reservations`
: associates external HDFS directories with a FATE transaction ID when in the process of bulk importing files, importing tables or exporting tables.

`table_locks`
: contains table read and write locks associated with performing some types of table operations.

`next_file`
: used for creating unique file and directory names for the lifetime of an Accumulo instance, this stores the current maximum number of names that have been allocated.

`bulk_failed_copyq`
: contains tablet server work queue for copying files that failed to bulk import.

`recovery`
: contains tablet server work queue for sorting write-ahead log files that need to be recovered.

`dead/tservers`
: contains tablet servers that have been shut down.

`fate/transaction_id`
: contains state of in-progress FATE operations.

<mediaobject role="cover"> <!-- source in the cover -→ <imageobject role="front-large" remap="lrg"> <imagedata format="PNG" fileref="images/cover.png"/> </imageobject> </mediaobject> <author> <firstname>Aaron</firstname> <surname>Cordova</surname> <authorblurb>

```
    Aaron Cordova worked as a Computer Systems Researcher at the National Security Agency where he sta

      </authorblurb>
    </author>
    <author>
```

```
  <firstname>Billie</firstname>
  <surname>Rinaldi</surname>
  <authorblurb>
From 2008 to 2012, Billie Rinaldi was a leader of the National Security Agency computer science re
  </authorblurb>
</author>
<author>
  <firstname>Michael</firstname>
  <surname>Wall</surname>
  <authorblurb>
    Michael Wall has been using Apache Accumulo since September 2010 and has been involved in all
  </authorblurb>
</author>
```