Apache Accumulo User Manual
Version 1.6

April 29, 2014

Contents

1 Introduction **2**

2 Accumulo Design **3**
- 2.1 Data Model . . . 3
- 2.2 Architecture . . . 3
- 2.3 Components . . . 4
  - 2.3.1 Tablet Server . . . 4
  - 2.3.2 Garbage Collector . . . 4
  - 2.3.3 Master . . . 4
  - 2.3.4 Tracer . . . 5
  - 2.3.5 Monitor . . . 5
  - 2.3.6 Client . . . 5
- 2.4 Data Management . . . 5
- 2.5 Tablet Service . . . 6
- 2.6 Compactions . . . 6
- 2.7 Splitting . . . 7
- 2.8 Fault-Tolerance . . . 7

3 Accumulo Shell **9**
- 3.1 Basic Administration . . . 9
- 3.2 Table Maintenance . . . 10
- 3.3 User Administration . . . 10

4 Writing Accumulo Clients **12**
- 4.1 Running Client Code . . . 12
- 4.2 Connecting . . . 13
- 4.3 Writing Data . . . 13
  - 4.3.1 BatchWriter . . . 13
  - 4.3.2 ConditionalWriter . . . 14

4.4 Reading Data . . . 14
4.4.1 Scanner . . . 14
4.4.2 Isolated Scanner . . . 15
4.4.3 BatchScanner . . . 15
4.5 Proxy . . . 16
4.5.1 Prequisites . . . 16
4.5.2 Configuration . . . 16
4.5.3 Running the Proxy Server . . . 17
4.5.4 Creating a Proxy Client . . . 17
4.5.5 Using a Proxy Client . . . 17

5 Development Clients **19**
5.1 Mock Accumulo . . . 19
5.2 Mini Accumulo Cluster . . . 20

6 Table Configuration **21**
6.1 Locality Groups . . . 21
6.1.1 Managing Locality Groups via the Shell . . . 21
6.1.2 Managing Locality Groups via the Client API . . . 22
6.2 Constraints . . . 22
6.3 Bloom Filters . . . 23
6.4 Iterators . . . 23
6.4.1 Setting Iterators via the Shell . . . 23
6.4.2 Setting Iterators Programmatically . . . 24
6.4.3 Versioning Iterators and Timestamps . . . 24
6.4.4 Filters . . . 25
6.4.5 Combiners . . . 26
6.5 Block Cache . . . 27
6.6 Compaction . . . 28
6.7 Pre-splitting tables . . . 29
6.8 Merging tablets . . . 30
6.9 Delete Range . . . 30
6.10 Cloning Tables . . . 31
6.11 Exporting Tables . . . 32

7 Table Design **33**
7.1 Basic Table . . . 33
7.2 RowID Design . . . 34
7.3 Lexicoders . . . 34
7.4 Indexing . . . 35

7.5 Entity-Attribute and Graph Tables 36
7.6 Document-Partitioned Indexing 38

8 High-Speed Ingest **40**
8.1 Pre-Splitting New Tables 40
8.2 Multiple Ingester Clients 41
8.3 Bulk Ingest 41
8.4 Logical Time for Bulk Ingest 42
8.5 MapReduce Ingest 42

9 Analytics **43**
9.1 MapReduce 43
9.1.1 Mapper and Reducer classes 43
9.1.2 AccumuloInputFormat options 44
9.1.3 AccumuloMultiTableInputFormat options 45
9.1.4 AccumuloOutputFormat options 45
9.2 Combiners 46
9.2.1 Feature Vectors 46
9.3 Statistical Modeling 47

10 Security **48**
10.1 Security Label Expressions 48
10.2 Security Label Expression Syntax 48
10.3 Authorization 49
10.4 User Authorizations 49
10.5 Pluggable Security 50
10.6 Secure Authorizations Handling 51
10.7 Query Services Layer 51

11 Administration **52**
11.1 Hardware 52
11.2 Network 52
11.3 Installation 53
11.4 Dependencies 53
11.5 Configuration 54
11.5.1 Edit conf/accumulo-env.sh 54
11.5.2 Native Map 54
11.5.3 Cluster Specification 55
11.5.4 Accumulo Settings 55
11.5.5 Deploy Configuration 56

11.6 Initialization 56
11.7 Running 56
11.7.1 Starting Accumulo 56
11.7.2 Stopping Accumulo 57
11.7.3 Adding a Node 57
11.7.4 Decomissioning a Node 57
11.8 Monitoring 58
11.9 Tracing 58
11.9.1 Tracers 58
11.9.2 Instrumenting a Client 58
11.9.3 Viewing Collected Traces 59
11.9.4 Tracing from the Shell 59
11.10Logging 60
11.11Recovery 60

12 Multi-Volume Installations **61**

13 Troubleshooting **63**
13.1 Logs 63
13.2 Monitor 63
13.3 HDFS 64
13.4 Zookeeper 65
13.4.1 Keeping the tablet server lock 66
13.5 Tools 67
13.6 System Metadata Tables 70
13.7 Simple System Recovery 72
13.8 Advanced System Recovery 73
13.8.1 HDFS Failure 73
13.8.2 ZooKeeper Failure 76
13.9 File Naming Conventions 77

A Configuration Management **79**
A.1 Configuration Overview 79
A.1.1 Zookeeper table properties 79
A.1.2 Zookeeper system properties 79
A.1.3 accumulo-site.xml 80
A.1.4 Default Values 80
A.2 Configuration in the Shell 80
A.3 Available Properties 81
A.3.1 rpc.* 81

A.3.2 instance.* . . . 83
A.3.3 general.* . . . 86
A.3.4 master.* . . . 88
A.3.5 tserver.* . . . 91
A.3.6 logger.* . . . 100
A.3.7 gc.* . . . 101
A.3.8 monitor.* . . . 102
A.3.9 trace.* . . . 103
A.3.10 trace.token.property.* . . . 105
A.3.11 table.* . . . 105
A.3.12 table.constraint.* . . . 112
A.3.13 table.iterator.* . . . 112
A.3.14 table.group.* . . . 113
A.3.15 table.majc.compaction.strategy.opts.* . . . 113
A.3.16 general.vfs.context.classpath.* . . . 113
A.4 Property Types . . . 113
A.4.1 duration . . . 113
A.4.2 date/time . . . 114
A.4.3 memory . . . 114
A.4.4 host list . . . 114
A.4.5 port . . . 114
A.4.6 count . . . 114
A.4.7 fraction/percentage . . . 114
A.4.8 path . . . 115
A.4.9 absolute path . . . 115
A.4.10 java class . . . 115
A.4.11 string . . . 115
A.4.12 boolean . . . 115
A.4.13 uri . . . 115

Chapter 1

Introduction

Apache Accumulo is a highly scalable structured store based on Google's BigTable. Accumulo is written in Java and operates over the Hadoop Distributed File System (HDFS), which is part of the popular Apache Hadoop project. Accumulo supports efficient storage and retrieval of structured data, including queries for ranges, and provides support for using Accumulo tables as input and output for MapReduce jobs.

Accumulo features automatic load-balancing and partitioning, data compression and fine-grained security labels.

Chapter 2

Accumulo Design

2.1 Data Model

Accumulo provides a richer data model than simple key-value stores, but is not a fully relational database. Data is represented as key-value pairs, where the key and value are comprised of the following elements:

Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		

All elements of the Key and the Value are represented as byte arrays except for Timestamp, which is a Long. Accumulo sorts keys by element and lexicographically in ascending order. Timestamps are sorted in descending order so that later versions of the same Key appear first in a sequential scan. Tables consist of a set of sorted key-value pairs.

2.2 Architecture

Accumulo is a distributed data storage and retrieval system and as such consists of several architectural components, some of which run on many individual servers. Much of the work Accumulo does involves maintaining certain properties of the data, such as organization, availability, and integrity, across many commodity-class machines.

2.3 Components

An instance of Accumulo includes many TabletServers, one Garbage Collector process, one Master server and many Clients.

2.3.1 Tablet Server

The TabletServer manages some subset of all the tablets (partitions of tables). This includes receiving writes from clients, persisting writes to a write-ahead log, sorting new key-value pairs in memory, periodically flushing sorted key-value pairs to new files in HDFS, and responding to reads from clients, forming a merge-sorted view of all keys and values from all the files it has created and the sorted in-memory store.

TabletServers also perform recovery of a tablet that was previously on a server that failed, reapplying any writes found in the write-ahead log to the tablet.

2.3.2 Garbage Collector

Accumulo processes will share files stored in HDFS. Periodically, the Garbage Collector will identify files that are no longer needed by any process, and delete them. Multiple garbage collectors can be run to provide hot-standby support. They will perform leader election among themselves to choose a single active instance.

2.3.3 Master

The Accumulo Master is responsible for detecting and responding to TabletServer failure. It tries to balance the load across TabletServer by assigning tablets carefully and instructing Tablet-Servers to unload tablets when necessary. The Master ensures all tablets are assigned to one TabletServer each, and handles table creation, alteration, and deletion requests from clients. The Master also coordinates startup, graceful shutdown and recovery of changes in write-ahead logs when Tablet servers fail.

Multiple masters may be run. The masters will choose among themselves a single master, and the others will become backups if the master should fail.

2.3.4 Tracer

The Accumulo Tracer process supports the distributed timing API provided by Accumulo. One to many of these processes can be run on a cluster which will write the timing information to a given Accumulo table for future reference. Seeing the section on Tracing for more information on this support.

2.3.5 Monitor

The Accumulo Monitor is a web application that provides a wealth of information about the state of an instance. The Monitor shows graphs and tables which contain information about read/write rates, cache hit/miss rates, and Accumulo table information such as scan rate and active/queued compactions. Additionally, the Monitor should always be the first point of entry when attempting to debug an Accumulo problem as it will show high-level problems in addition to aggregated errors from all nodes in the cluster. See the section on Monitoring for more information.

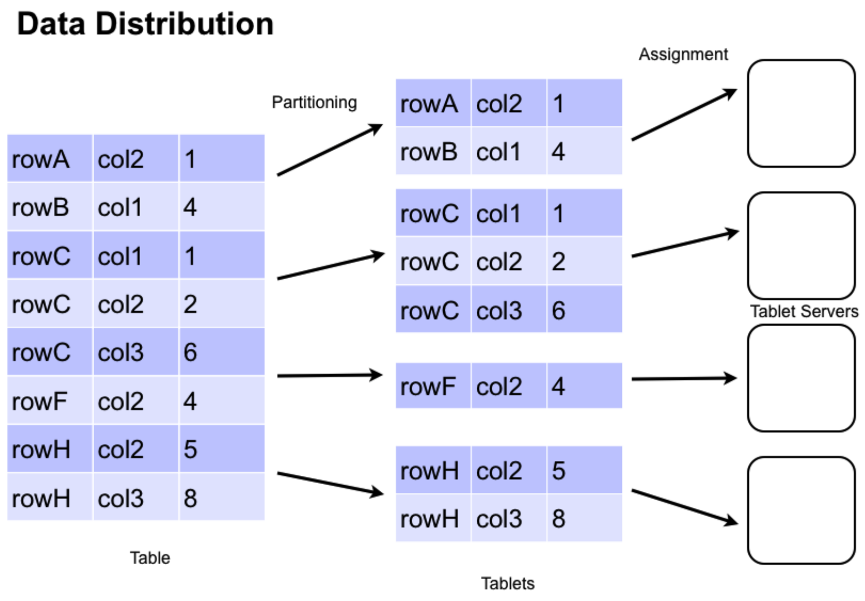
Multiple Monitors can be run to provide hot-standby support in the face of failure. Due to the forwarding of logs from remote hosts to the Monitor, only one Monitor process should be active at one time. Leader election will be performed internally to choose the active Monitor.

2.3.6 Client

Accumulo includes a client library that is linked to every application. The client library contains logic for finding servers managing a particular tablet, and communicating with TabletServers to write and retrieve key-value pairs.

2.4 Data Management

Accumulo stores data in tables, which are partitioned into tablets. Tablets are partitioned on row boundaries so that all of the columns and values for a particular row are found together within the same tablet. The Master assigns Tablets to one TabletServer at a time. This enables row-level transactions to take place without using distributed locking or some other complicated synchronization mechanism. As clients insert and query data, and as machines are added and removed from the cluster, the Master migrates tablets to ensure they remain available and that the ingest and query load is balanced across the cluster.



2.5 Tablet Service

When a write arrives at a TabletServer it is written to a Write-Ahead Log and then inserted into a sorted data structure in memory called a MemTable. When the MemTable reaches a certain size the TabletServer writes out the sorted key-value pairs to a file in HDFS called Indexed Sequential Access Method (ISAM) file. This process is called a minor compaction. A new MemTable is then created and the fact of the compaction is recorded in the Write-Ahead Log.

When a request to read data arrives at a TabletServer, the TabletServer does a binary search across the MemTable as well as the in-memory indexes associated with each ISAM file to find the relevant values. If clients are performing a scan, several key-value pairs are returned to the client in order from the MemTable and the set of ISAM files by performing a merge-sort as they are read.

2.6 Compactions

In order to manage the number of files per tablet, periodically the TabletServer performs Major Compactions of files within a tablet, in which some set of ISAM files are combined into one file. The previous files will eventually be removed by the Garbage Collector. This also provides an opportunity to permanently remove deleted key-value pairs by omitting key-value pairs suppressed by a delete entry when the new file is created.

2.7 Splitting

When a table is created it has one tablet. As the table grows its initial tablet eventually splits into two tablets. Its likely that one of these tablets will migrate to another tablet server. As the table continues to grow, its tablets will continue to split and be migrated. The decision to automatically split a tablet is based on the size of a tablets files. The size threshold at which a tablet splits is configurable per table. In addition to automatic splitting, a user can manually add split points to a table to create new tablets. Manually splitting a new table can parallelize reads and writes giving better initial performance without waiting for automatic splitting.

As data is deleted from a table, tablets may shrink. Over time this can lead to small or empty tablets. To deal with this, merging of tablets was introduced in Accumulo 1.4. This is discussed in more detail later.

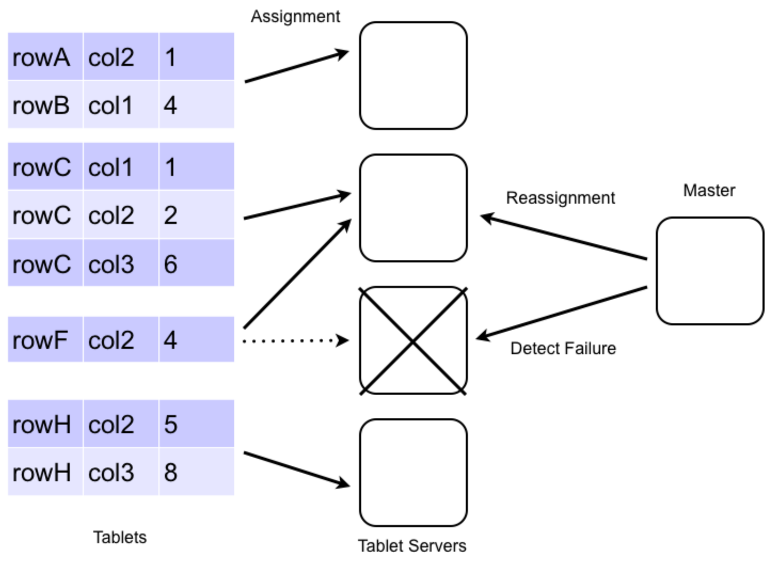
2.8 Fault-Tolerance

If a TabletServer fails, the Master detects it and automatically reassigns the tablets assigned from the failed server to other servers. Any key-value pairs that were in memory at the time the TabletServer fails are automatically reapplied from the Write-Ahead Log(WAL) to prevent any loss of data.

Tablet servers write their WALs directly to HDFS so the logs are available to all tablet servers for recovery. To make the recovery process efficient, the updates within a log are grouped by tablet. TabletServers can quickly apply the mutations from the sorted logs that are destined for the tablets they have now been assigned.

TabletServer failures are noted on the Master's monitor page, accessible via
http://master-address:50095/monitor.

Automatic Failure Handling



Chapter 3

Accumulo Shell

Accumulo provides a simple shell that can be used to examine the contents and configuration settings of tables, insert/update/delete values, and change configuration settings.

The shell can be started by the following command:

```
$ACCUMULO_HOME/bin/accumulo shell -u [username]
```

The shell will prompt for the corresponding password to the username specified and then display the following prompt:

```
Shell - Apache Accumulo Interactive Shell
-
- version 1.5
- instance name: myinstance
- instance id: 00000000-0000-0000-0000-000000000000
-
- type 'help' for a list of available commands
-
```

3.1 Basic Administration

The Accumulo shell can be used to create and delete tables, as well as to configure table and instance specific options.

```
root@myinstance> tables
accumulo.metadata
accumulo.root

root@myinstance> createtable mytable

root@myinstance mytable>

root@myinstance mytable> tables
accumulo.metadata
accumulo.root
mytable

root@myinstance mytable> createtable testtable

root@myinstance testtable>

root@myinstance testtable> deletetable testtable
deletetable { testtable } (yes|no)? yes
Table: [testtable] has been deleted.

root@myinstance>
```

The Shell can also be used to insert updates and scan tables. This is useful for inspecting tables.

```
root@myinstance mytable> scan

root@myinstance mytable> insert row1 colf colq value1
insert successful

root@myinstance mytable> scan
row1 colf:colq [] value1
```

The value in brackets "[]" would be the visibility labels. Since none were used, this is empty for this row. You can use the "-st" option to scan to see the timestamp for the cell, too.

3.2 Table Maintenance

The **compact** command instructs Accumulo to schedule a compaction of the table during which files are consolidated and deleted entries are removed.

```
root@myinstance mytable> compact -t mytable
07 16:13:53,201 [shell.Shell] INFO : Compaction of table mytable started for given range
```

The **flush** command instructs Accumulo to write all entries currently in memory for a given table to disk.

```
root@myinstance mytable> flush -t mytable
07 16:14:19,351 [shell.Shell] INFO : Flush of table mytable
initiated...
```

3.3 User Administration

The Shell can be used to add, remove, and grant privileges to users.

```
root@myinstance mytable> createuser bob
Enter new password for 'bob': *********
Please confirm new password for 'bob': *********

root@myinstance mytable> authenticate bob
Enter current password for 'bob': *********
Valid

root@myinstance mytable> grant System.CREATE_TABLE -s -u bob
```

```
root@myinstance mytable> user bob
Enter current password for 'bob': *********

bob@myinstance mytable> userpermissions
System permissions: System.CREATE_TABLE
Table permissions (accumulo.metadata): Table.READ
Table permissions (mytable): NONE

bob@myinstance mytable> createtable bobstable
bob@myinstance bobstable>

bob@myinstance bobstable> user root
Enter current password for 'root': *********

root@myinstance bobstable> revoke System.CREATE_TABLE -s -u bob
```

Chapter 4

Writing Accumulo Clients

4.1 Running Client Code

There are multiple ways to run Java code that uses Accumulo. Below is a list of the different ways to execute client code.

- using java executable
- using the accumulo script
- using the tool script

In order to run client code written to run against Accumulo, you will need to include the jars that Accumulo depends on in your classpath. Accumulo client code depends on Hadoop and Zookeeper. For Hadoop add the hadoop client jar, all of the jars in the Hadoop lib directory, and the conf directory to the classpath. For Zookeeper 3.3 you only need to add the Zookeeper jar, and not what is in the Zookeeper lib directory. You can run the following command on a configured Accumulo system to see what its using for its classpath.

```
$ACCUMULO_HOME/bin/accumulo classpath
```

Another option for running your code is to put a jar file in `$ACCUMULO_HOME/lib/ext`. After doing this you can use the accumulo script to execute your code. For example if you create a jar containing the class com.foo.Client and placed that in lib/ext, then you could use the command `$ACCUMULO_HOME/bin/accumulo com.foo.Client` to execute your code.

If you are writing map reduce job that access Accumulo, then you can use the bin/tool.sh script to run those jobs. See the map reduce example.

4.2 Connecting

All clients must first identify the Accumulo instance to which they will be communicating. Code to do this is as follows:

```
String instanceName = "myinstance";
String zooServers = "zooserver-one,zooserver-two"
Instance inst = new ZooKeeperInstance(instanceName, zooServers);

Connector conn = inst.getConnector("user", new PasswordToken("passwd"));
```

4.3 Writing Data

Data are written to Accumulo by creating Mutation objects that represent all the changes to the columns of a single row. The changes are made atomically in the TabletServer. Clients then add Mutations to a BatchWriter which submits them to the appropriate TabletServers.

Mutations can be created thus:

```
Text rowID = new Text("row1");
Text colFam = new Text("myColFam");
Text colQual = new Text("myColQual");
ColumnVisibility colVis = new ColumnVisibility("public");
long timestamp = System.currentTimeMillis();

Value value = new Value("myValue".getBytes());

Mutation mutation = new Mutation(rowID);
mutation.put(colFam, colQual, colVis, timestamp, value);
```

4.3.1 BatchWriter

The BatchWriter is highly optimized to send Mutations to multiple TabletServers and automatically batches Mutations destined for the same TabletServer to amortize network overhead. Care must be taken to avoid changing the contents of any Object passed to the BatchWriter since it keeps objects in memory while batching.

Mutations are added to a BatchWriter thus:

```
//BatchWriterConfig has reasonable defaults
BatchWriterConfig config = new BatchWriterConfig();
config.setMaxMemory(10000000L); // bytes available to batchwriter for buffering mutations

BatchWriter writer = conn.createBatchWriter("table", config)

writer.add(mutation);

writer.close();
```

An example of using the batch writer can be found at
accumulo/docs/examples/README.batch

4.3.2 ConditionalWriter

The ConditionalWriter enables efficient, atomic read-modify-write operations on rows. The ConditionalWriter writes special Mutations which have a list of per column conditions that must all be met before the mutation is applied. The conditions are checked in the tablet server while a row lock is held[1]. The conditions that can be checked for a column are equality and absence. For example a conditional mutation can require that column A is absent inorder to be applied. Iterators can be applied when checking conditions. Using iterators, many other operations besides equality and absence can be checked. For example, using an iterator that converts values less than 5 to 0 and everything else to 1, its possible to only apply a mutation when a column is less than 5.

In the case when a tablet server dies after a client sent a conditional mutation, its not known if the mutation was applied or not. When this happens the ConditionalWriter reports a status of UNKNOWN for the ConditionalMutation. In many cases this situation can be dealt with by simply reading the row again and possibly sending another conditional mutation. If this is not sufficient, then a higher level of abstraction can be built by storing transactional information within a row.

An example of using the batch writer can be found at
accumulo/docs/examples/README.reservations

4.4 Reading Data

Accumulo is optimized to quickly retrieve the value associated with a given key, and to efficiently return ranges of consecutive keys and their associated values.

4.4.1 Scanner

To retrieve data, Clients use a Scanner, which acts like an Iterator over keys and values. Scanners can be configured to start and stop at particular keys, and to return a subset of the columns available.

[1]Mutations written by the BatchWriter will not obtain a row lock.

```
// specify which visibilities we are allowed to see
Authorizations auths = new Authorizations("public");

Scanner scan =
    conn.createScanner("table", auths);

scan.setRange(new Range("harry","john"));
scan.fetchFamily("attributes");

for(Entry<Key,Value> entry : scan) {
    String row = entry.getKey().getRow();
    Value value = entry.getValue();
}
```

4.4.2 Isolated Scanner

Accumulo supports the ability to present an isolated view of rows when scanning. There are three possible ways that a row could change in Accumulo :

- a mutation applied to a table
- iterators executed as part of a minor or major compaction
- bulk import of new files

Isolation guarantees that either all or none of the changes made by these operations on a row are seen. Use the IsolatedScanner to obtain an isolated view of an Accumulo table. When using the regular scanner it is possible to see a non isolated view of a row. For example if a mutation modifies three columns, it is possible that you will only see two of those modifications. With the isolated scanner either all three of the changes are seen or none.

The IsolatedScanner buffers rows on the client side so a large row will not crash a tablet server. By default rows are buffered in memory, but the user can easily supply their own buffer if they wish to buffer to disk when rows are large.

For an example, look at the following
`examples/simple/src/main/java/org/apache/accumulo/examples/simple/isolation/InterferenceTest.java`

4.4.3 BatchScanner

For some types of access, it is more efficient to retrieve several ranges simultaneously. This arises when accessing a set of rows that are not consecutive whose IDs have been retrieved from a secondary index, for example.

The BatchScanner is configured similarly to the Scanner; it can be configured to retrieve a subset of the columns available, but rather than passing a single Range, BatchScanners accept a set of Ranges. It is important to note that the keys returned by a BatchScanner are not in sorted order since the keys streamed are from multiple TabletServers in parallel.

```
ArrayList<Range> ranges = new ArrayList<Range>();
// populate list of ranges ...

BatchScanner bscan =
    conn.createBatchScanner("table", auths, 10);

bscan.setRanges(ranges);
bscan.fetchFamily("attributes");
for(Entry<Key,Value> entry : scan)
    System.out.println(entry.getValue());
```

An example of the BatchScanner can be found at
accumulo/docs/examples/README.batch

4.5 Proxy

The proxy API allows the interaction with Accumulo with languages other than Java. A proxy server is provided in the codebase and a client can further be generated.

4.5.1 Prequisites

The proxy server can live on any node in which the basic client API would work. That means it must be able to communicate with the Master, ZooKeepers, NameNode, and the Data nodes. A proxy client only needs the ability to communicate with the proxy server.

4.5.2 Configuration

The configuration options for the proxy server live inside of a properties file. At the very least, you need to supply the following properties:

```
protocolFactory=org.apache.thrift.protocol.TCompactProtocol$Factory
tokenClass=org.apache.accumulo.core.client.security.tokens.PasswordToken
port=42424
instance=test
zookeepers=localhost:2181
```

You can find a sample configuration file in your distribution:

```
$ACCUMULO_HOME/proxy/proxy.properties.
```

This sample configuration file further demonstrates an abilty to back the proxy server by MockAccumulo or the MiniAccumuloCluster.

4.5.3 Running the Proxy Server

After the properties file holding the configuration is created, the proxy server can be started using the following command in the Accumulo distribution (assuming you your properties file is named config.properties):

```
$ACCUMULO_HOME/bin/accumulo proxy -p config.properties
```

4.5.4 Creating a Proxy Client

Aside from installing the Thrift compiler, you will also need the language-specific library for Thrift installed to generate client code in that language. Typically, your operating system's package manager will be able to automatically install these for you in an expected location such as /usr/lib/python/site-packages/thrift.

You can find the thrift file for generating the client:

```
$ACCUMULO_HOME/proxy/proxy.thrift.
```

After a client is generated, the port specified in the configuration properties above will be used to connect to the server.

4.5.5 Using a Proxy Client

The following examples have been written in Java and the method signatures may be slightly different depending on the language specified when generating client with the Thrift compiler. After initiating a connection to the Proxy (see Apache Thrift's documentation for examples of connecting to a Thrift service), the methods on the proxy client will be available. The first thing to do is log in:

```
Map password = new HashMap<String,String>();
password.put("password", "secret");
ByteBuffer token = client.login("root", password);
```

Once logged in, the token returned will be used for most subsequent calls to the client. Let's create a table, add some data, scan the table, and delete it.

First, create a table.

```
client.createTable(token, "myTable", true, TimeType.MILLIS);
```

Next, add some data:

```
// first, create a writer on the server
String writer = client.createWriter(token, "myTable", new WriterOptions());

// build column updates
Map<ByteBuffer, List<ColumnUpdate> cells> cellsToUpdate = //...
```

```
// send updates to the server
client.updateAndFlush(writer, "myTable", cellsToUpdate);

client.closeWriter(writer);
```

Scan for the data and batch the return of the results on the server:

```
String scanner = client.createScanner(token, "myTable", new ScanOptions());
ScanResult results = client.nextK(scanner, 100);

for(KeyValue keyValue : results.getResultsIterator()) {
  // do something with results
}

client.closeScanner(scanner);
```

Chapter 5

Development Clients

Normally, Accumulo consists of lots of moving parts. Even a stand-alone version of Accumulo requires Hadoop, Zookeeper, the Accumulo master, a tablet server, etc. If you want to write a unit test that uses Accumulo, you need a lot of infrastructure in place before your test can run.

5.1 Mock Accumulo

Mock Accumulo supplies mock implementations for much of the client API. It presently does not enforce users, logins, permissions, etc. It does support Iterators and Combiners. Note that MockAccumulo holds all data in memory, and will not retain any data or settings between runs.

While normal interaction with the Accumulo client looks like this:

```
Instance instance = new ZooKeeperInstance(...);
Connector conn = instance.getConnector(user, passwordToken);
```

To interact with the MockAccumulo, just replace the ZooKeeperInstance with MockInstance:

```
Instance instance = new MockInstance();
```

In fact, you can use the "fake" option to the Accumulo shell and interact with MockAccumulo:

```
$ ./bin/accumulo shell --fake -u root -p ''

Shell - Apache Accumulo Interactive Shell
-
- version: 1.6
- instance name: fake
- instance id: mock-instance-id
-
- type 'help' for a list of available commands
-
root@fake> createtable test
root@fake test> insert row1 cf cq value
```

```
root@fake test> insert row2 cf cq value2
root@fake test> insert row3 cf cq value3
root@fake test> scan
row1 cf:cq []    value
row2 cf:cq []    value2
row3 cf:cq []    value3
root@fake test> scan -b row2 -e row2
row2 cf:cq []    value2
root@fake test>
```

When testing Map Reduce jobs, you can also set the Mock Accumulo on the AccumuloInputFormat and AccumuloOutputFormat classes:

```
// ... set up job configuration
AccumuloInputFormat.setMockInstance(job, "mockInstance");
AccumuloOutputFormat.setMockInstance(job, "mockInstance");
```

5.2 Mini Accumulo Cluster

While the Mock Accumulo provides a lightweight implementation of the client API for unit testing, it is often necessary to write more realistic end-to-end integration tests that take advantage of the entire ecosystem. The Mini Accumulo Cluster makes this possible by configuring and starting Zookeeper, initializing Accumulo, and starting the Master as well as some Tablet Servers. It runs against the local filesystem instead of having to start up HDFS.

To start it up, you will need to supply an empty directory and a root password as arguments:

```
File tempDirectory = // JUnit and Guava supply mechanisms for creating temp directories
MiniAccumuloCluster accumulo = new MiniAccumuloCluster(tempDirectory, "password");
accumulo.start();
```

Once we have our mini cluster running, we will want to interact with the Accumulo client API:

```
Instance instance = new ZooKeeperInstance(accumulo.getInstanceName(), accumulo.getZooKeepers());
Connector conn = instance.getConnector("root", new PasswordToken("password"));
```

Upon completion of our development code, we will want to shutdown our MiniAccumuloCluster:

```
accumulo.stop()
// delete your temporary folder
```

Chapter 6

Table Configuration

Accumulo tables have a few options that can be configured to alter the default behavior of Accumulo as well as improve performance based on the data stored. These include locality groups, constraints, bloom filters, iterators, and block cache. For a complete list of available configuration options, see Appendix A.

6.1 Locality Groups

Accumulo supports storing sets of column families separately on disk to allow clients to efficiently scan over columns that are frequently used together and to avoid scanning over column families that are not requested. After a locality group is set, Scanner and BatchScanner operations will automatically take advantage of them whenever the fetchColumnFamilies() method is used.

By default, tables place all column families into the same "default" locality group. Additional locality groups can be configured anytime via the shell or programmatically as follows:

6.1.1 Managing Locality Groups via the Shell

```
usage: setgroups <group>=<col fam>{,<col fam>}{ <group>=<col fam>{,<col
fam>}} [-?] -t <table>

user@myinstance mytable> setgroups group_one=colf1,colf2 -t mytable

user@myinstance mytable> getgroups -t mytable
```

6.1.2 Managing Locality Groups via the Client API

```
Connector conn;

HashMap<String,Set<Text>> localityGroups = new HashMap<String, Set<Text>>();

HashSet<Text> metadataColumns = new HashSet<Text>();
metadataColumns.add(new Text("domain"));
metadataColumns.add(new Text("link"));

HashSet<Text> contentColumns = new HashSet<Text>();
contentColumns.add(new Text("body"));
contentColumns.add(new Text("images"));

localityGroups.put("metadata", metadataColumns);
localityGroups.put("content", contentColumns);

conn.tableOperations().setLocalityGroups("mytable", localityGroups);

// existing locality groups can be obtained as follows
Map<String, Set<Text>> groups =
    conn.tableOperations().getLocalityGroups("mytable");
```

The assignment of Column Families to Locality Groups can be changed anytime. The physical movement of column families into their new locality groups takes place via the periodic Major Compaction process that takes place continuously in the background. Major Compaction can also be scheduled to take place immediately through the shell:

```
user@myinstance mytable> compact -t mytable
```

6.2 Constraints

Accumulo supports constraints applied on mutations at insert time. This can be used to disallow certain inserts according to a user defined policy. Any mutation that fails to meet the requirements of the constraint is rejected and sent back to the client.

Constraints can be enabled by setting a table property as follows:

```
user@myinstance mytable> constraint -t mytable -a com.test.ExampleConstraint com.test.AnotherConstraint
user@myinstance mytable> constraint -l
com.test.ExampleConstraint=1
com.test.AnotherConstraint=2
```

Currently there are no general-purpose constraints provided with the Accumulo distribution. New constraints can be created by writing a Java class that implements the following interface:

```
org.apache.accumulo.core.constraints.Constraint
```

To deploy a new constraint, create a jar file containing the class implementing the new constraint and place it in the lib directory of the Accumulo installation. New constraint jars can be added to Accumulo and enabled without restarting but any change to an existing constraint class requires Accumulo to be restarted.

An example of constraints can be found in
`accumulo/docs/examples/README.constraints` with corresponding code under
`accumulo/examples/simple/src/main/java/accumulo/examples/simple/constraints` .

6.3 Bloom Filters

As mutations are applied to an Accumulo table, several files are created per tablet. If bloom filters are enabled, Accumulo will create and load a small data structure into memory to determine whether a file contains a given key before opening the file. This can speed up lookups considerably.

To enable bloom filters, enter the following command in the Shell:

```
user@myinstance> config -t mytable -s table.bloom.enabled=true
```

An extensive example of using Bloom Filters can be found at
`accumulo/docs/examples/README.bloom` .

6.4 Iterators

Iterators provide a modular mechanism for adding functionality to be executed by Tablet-Servers when scanning or compacting data. This allows users to efficiently summarize, filter, and aggregate data. In fact, the built-in features of cell-level security and column fetching are implemented using Iterators. Some useful Iterators are provided with Accumulo and can be found in the org.apache.accumulo.core.iterators.user package. In each case, any custom Iterators must be included in Accumulo's classpath, typically by including a jar in `$ACCUMULO_HOME/lib` or `$ACCUMULO_HOME/lib/ext`, although the VFS classloader allows for classpath manipulation using a variety of schemes including URLs and HDFS URIs.

6.4.1 Setting Iterators via the Shell

Iterators can be configured on a table at scan, minor compaction and/or major compaction scopes. If the Iterator implements the OptionDescriber interface, the setiter command can be used which will interactively prompt the user to provide values for the given necessary options.

```
usage: setiter [-?] -ageoff | -agg | -class <name> | -regex |
-reqvis | -vers   [-majc] [-minc] [-n <itername>] -p <pri>
[-scan] [-t <table>]

user@myinstance mytable> setiter -t mytable -scan -p 15 -n myiter -class com.company.MyIterator
```

The config command can always be used to manually configure iterators which is useful in cases where the Iterator does not implement the OptionDescriber interface.

```
config -t mytable -s table.iterator.scan.myiter=15,com.company.MyIterator
config -t mytable -s table.iterator.minc.myiter=15,com.company.MyIterator
config -t mytable -s table.iterator.majc.myiter=15,com.company.MyIterator
config -t mytable -s table.iterator.scan.myiter.opt.myoptionname=myoptionvalue
config -t mytable -s table.iterator.minc.myiter.opt.myoptionname=myoptionvalue
config -t mytable -s table.iterator.majc.myiter.opt.myoptionname=myoptionvalue
```

6.4.2 Setting Iterators Programmatically

```
scanner.addIterator(new IteratorSetting(
    15, // priority
    "myiter", // name this iterator
    "com.company.MyIterator" // class name
));
```

Some iterators take additional parameters from client code, as in the following example:

```
IteratorSetting iter = new IteratorSetting(...);
iter.addOption("myoptionname", "myoptionvalue");
scanner.addIterator(iter)
```

Tables support separate Iterator settings to be applied at scan time, upon minor compaction and upon major compaction. For most uses, tables will have identical iterator settings for all three to avoid inconsistent results.

6.4.3 Versioning Iterators and Timestamps

Accumulo provides the capability to manage versioned data through the use of timestamps within the Key. If a timestamp is not specified in the key created by the client then the system will set the timestamp to the current time. Two keys with identical rowIDs and columns but different timestamps are considered two versions of the same key. If two inserts are made into Accumulo with the same rowID, column, and timestamp, then the behavior is non-deterministic.

Timestamps are sorted in descending order, so the most recent data comes first. Accumulo can be configured to return the top k versions, or versions later than a given date. The default is to return the one most recent version.

The version policy can be changed by changing the VersioningIterator options for a table as follows:

```
user@myinstance mytable> config -t mytable -s table.iterator.scan.vers.opt.maxVersions=3

user@myinstance mytable> config -t mytable -s table.iterator.minc.vers.opt.maxVersions=3

user@myinstance mytable> config -t mytable -s table.iterator.majc.vers.opt.maxVersions=3
```

When a table is created, by default its configured to use the VersioningIterator and keep one version. A table can be created without the VersioningIterator with the -ndi option in the shell. Also the Java API has the following method

```
connector.tableOperations.create(String tableName, boolean limitVersion).
```

Logical Time

Accumulo 1.2 introduces the concept of logical time. This ensures that timestamps set by Accumulo always move forward. This helps avoid problems caused by TabletServers that have different time settings. The per tablet counter gives unique one up time stamps on a per mutation basis. When using time in milliseconds, if two things arrive within the same millisecond then both receive the same timestamp. When using time in milliseconds, Accumulo set times will still always move forward and never backwards.

A table can be configured to use logical timestamps at creation time as follows:

```
user@myinstance> createtable -tl logical
```

Deletes

Deletes are special keys in Accumulo that get sorted along will all the other data. When a delete key is inserted, Accumulo will not show anything that has a timestamp less than or equal to the delete key. During major compaction, any keys older than a delete key are omitted from the new file created, and the omitted keys are removed from disk as part of the regular garbage collection process.

6.4.4 Filters

When scanning over a set of key-value pairs it is possible to apply an arbitrary filtering policy through the use of a Filter. Filters are types of iterators that return only key-value pairs that satisfy the filter logic. Accumulo has a few built-in filters that can be configured on any table: AgeOff, ColumnAgeOff, Timestamp, NoVis, and RegEx. More can be added by writing a Java class that extends the
org.apache.accumulo.core.iterators.Filter class.

The AgeOff filter can be configured to remove data older than a certain date or a fixed amount of time from the present. The following example sets a table to delete everything inserted over 30 seconds ago:

```
user@myinstance> createtable filtertest
user@myinstance filtertest> setiter -t filtertest -scan -minc -majc -p 10 -n myfilter -ageoff
AgeOffFilter removes entries with timestamps more than <ttl> milliseconds old
----------> set org.apache.accumulo.core.iterators.user.AgeOffFilter parameter negate, default false
                keeps k/v that pass accept method, true rejects k/v that pass accept method:
----------> set org.apache.accumulo.core.iterators.user.AgeOffFilter parameter ttl, time to
                live (milliseconds): 3000
----------> set org.apache.accumulo.core.iterators.user.AgeOffFilter parameter currentTime, if set,
                use the given value as the absolute time in milliseconds as the current time of day:
user@myinstance filtertest>
user@myinstance filtertest> scan
user@myinstance filtertest> insert foo a b c
user@myinstance filtertest> scan
foo a:b [] c
user@myinstance filtertest> sleep 4
user@myinstance filtertest> scan
user@myinstance filtertest>
```

To see the iterator settings for a table, use:

```
user@example filtertest> config -t filtertest -f iterator
---------+---------------------------------------------+------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+------------------
table    | table.iterator.majc.myfilter .............. | 10,org.apache.accumulo.core.iterators.user.AgeOffFilter
table    | table.iterator.majc.myfilter.opt.ttl ...... | 3000
table    | table.iterator.majc.vers .................. | 20,org.apache.accumulo.core.iterators.VersioningIterator
table    | table.iterator.majc.vers.opt.maxVersions .. | 1
table    | table.iterator.minc.myfilter .............. | 10,org.apache.accumulo.core.iterators.user.AgeOffFilter
table    | table.iterator.minc.myfilter.opt.ttl ...... | 3000
table    | table.iterator.minc.vers .................. | 20,org.apache.accumulo.core.iterators.VersioningIterator
table    | table.iterator.minc.vers.opt.maxVersions .. | 1
table    | table.iterator.scan.myfilter .............. | 10,org.apache.accumulo.core.iterators.user.AgeOffFilter
table    | table.iterator.scan.myfilter.opt.ttl ...... | 3000
table    | table.iterator.scan.vers .................. | 20,org.apache.accumulo.core.iterators.VersioningIterator
table    | table.iterator.scan.vers.opt.maxVersions .. | 1
---------+---------------------------------------------+------------------
```

6.4.5 Combiners

Accumulo allows Combiners to be configured on tables and column families. When a Combiner is set it is applied across the values associated with any keys that share rowID, column family, and column qualifier. This is similar to the reduce step in MapReduce, which applied some function to all the values associated with a particular key.

For example, if a summing combiner were configured on a table and the following mutations were inserted:

```
Row     Family Qualifier Timestamp  Value
rowID1  colfA  colqA     20100101   1
rowID1  colfA  colqA     20100102   1
```

The table would reflect only one aggregate value:

```
rowID1  colfA  colqA     -          2
```

Combiners can be enabled for a table using the setiter command in the shell. Below is an example.

```
root@a14 perDayCounts> setiter -t perDayCounts -p 10 -scan -minc -majc -n daycount
                       -class org.apache.accumulo.core.iterators.user.SummingCombiner
TypedValueCombiner can interpret Values as a variety of number encodings
  (VLong, Long, or String) before combining
----------> set SummingCombiner parameter columns,
            <col fam>[:<col qual>]{,<col fam>[:<col qual>]} : day
----------> set SummingCombiner parameter type, <VARNUM|LONG|STRING>: STRING

root@a14 perDayCounts> insert foo day 20080101 1
root@a14 perDayCounts> insert foo day 20080101 1
root@a14 perDayCounts> insert foo day 20080103 1
root@a14 perDayCounts> insert bar day 20080101 1
root@a14 perDayCounts> insert bar day 20080101 1

root@a14 perDayCounts> scan
bar day:20080101 []    2
foo day:20080101 []    2
foo day:20080103 []    1
```

Accumulo includes some useful Combiners out of the box. To find these look in the `org.apache.accumulo.core.iterators.user` package.

Additional Combiners can be added by creating a Java class that extends `org.apache.accumulo.core.iterators.Combiner` and adding a jar containing that class to Accumulo's lib/ext directory.

An example of a Combiner can be found under

```
accumulo/examples/simple/src/main/java/org/apache/accumulo/examples/simple/combiner/StatsCombiner.java
```

6.5 Block Cache

In order to increase throughput of commonly accessed entries, Accumulo employs a block cache. This block cache buffers data in memory so that it doesn't have to be read off of disk. The RFile format that Accumulo prefers is a mix of index blocks and data blocks, where the index blocks are used to find the appropriate data blocks. Typical queries to Accumulo result in a binary search over several index blocks followed by a linear scan of one or more data blocks.

The block cache can be configured on a per-table basis, and all tablets hosted on a tablet server share a single resource pool. To configure the size of the tablet server's block cache, set the following properties:

```
tserver.cache.data.size: Specifies the size of the cache for file data blocks.
tserver.cache.index.size: Specifies the size of the cache for file indices.
```

To enable the block cache for your table, set the following properties:

```
table.cache.block.enable: Determines whether file (data) block cache is enabled.
table.cache.index.enable: Determines whether index cache is enabled.
```

The block cache can have a significant effect on alleviating hot spots, as well as reducing query latency. It is enabled by default for the metadata tables.

6.6 Compaction

As data is written to Accumulo it is buffered in memory. The data buffered in memory is eventually written to HDFS on a per tablet basis. Files can also be added to tablets directly by bulk import. In the background tablet servers run major compactions to merge multiple files into one. The tablet server has to decide which tablets to compact and which files within a tablet to compact. This decision is made using the compaction ratio, which is configurable on a per table basis. To configure this ratio modify the following property:

```
table.compaction.major.ratio
```

Increasing this ratio will result in more files per tablet and less compaction work. More files per tablet means more higher query latency. So adjusting this ratio is a trade off between ingest and query performance. The ratio defaults to 3.

The way the ratio works is that a set of files is compacted into one file if the sum of the sizes of the files in the set is larger than the ratio multiplied by the size of the largest file in the set. If this is not true for the set of all files in a tablet, the largest file is removed from consideration, and the remaining files are considered for compaction. This is repeated until a compaction is triggered or there are no files left to consider.

The number of background threads tablet servers use to run major compactions is configurable. To configure this modify the following property:

```
tserver.compaction.major.concurrent.max
```

Also, the number of threads tablet servers use for minor compactions is configurable. To configure this modify the following property:

```
tserver.compaction.minor.concurrent.max
```

The numbers of minor and major compactions running and queued is visible on the Accumulo monitor page. This allows you to see if compactions are backing up and adjustments to the above settings are needed. When adjusting the number of threads available for compactions, consider the number of cores and other tasks running on the nodes such as maps and reduces.

If major compactions are not keeping up, then the number of files per tablet will grow to a point such that query performance starts to suffer. One way to handle this situation is to increase the compaction ratio. For example, if the compaction ratio were set to 1, then every new file added to a tablet by minor compaction would immediately queue the tablet for major compaction. So if a tablet has a 200M file and minor compaction writes a 1M file, then the major compaction will attempt to merge the 200M and 1M file. If the tablet server has lots of tablets trying to do this sort of thing, then major compactions will back up and the number of files per tablet will start to grow, assuming data is being continuously written. Increasing the compaction ratio will alleviate backups by lowering the amount of major compaction work that needs to be done.

Another option to deal with the files per tablet growing too large is to adjust the following property:

```
table.file.max
```

When a tablet reaches this number of files and needs to flush its in-memory data to disk, it will choose to do a merging minor compaction. A merging minor compaction will merge the tablet's smallest file with the data in memory at minor compaction time. Therefore the number of files will not grow beyond this limit. This will make minor compactions take longer, which will cause ingest performance to decrease. This can cause ingest to slow down until major compactions have enough time to catch up. When adjusting this property, also consider adjusting the compaction ratio. Ideally, merging minor compactions never need to occur and major compactions will keep up. It is possible to configure the file max and compaction ratio such that only merging minor compactions occur and major compactions never occur. This should be avoided because doing only merging minor compactions causes $O(N^2)$ work to be done. The amount of work done by major compactions is $O(N * \log_R(N))$ where R is the compaction ratio.

Compactions can be initiated manually for a table. To initiate a minor compaction, use the flush command in the shell. To initiate a major compaction, use the compact command in the shell. The compact command will compact all tablets in a table to one file. Even tablets with one file are compacted. This is useful for the case where a major compaction filter is configured for a table. In 1.4 the ability to compact a range of a table was added. To use this feature specify start and stop rows for the compact command. This will only compact tablets that overlap the given row range.

6.7 Pre-splitting tables

Accumulo will balance and distribute tables across servers. Before a table gets large, it will be maintained as a single tablet on a single server. This limits the speed at which data can be added or queried to the speed of a single node. To improve performance when the a table is new, or small, you can add split points and generate new tablets.

In the shell:

```
root@myinstance> createtable newTable
root@myinstance> addsplits -t newTable g n t
```

This will create a new table with 4 tablets. The table will be split on the letters "g", "n", and "t" which will work nicely if the row data start with lower-case alphabetic characters. If your row data includes binary information or numeric information, or if the distribution of the row information is not flat, then you would pick different split points. Now ingest and query can proceed on 4 nodes which can improve performance.

6.8 Merging tablets

Over time, a table can get very large, so large that it has hundreds of thousands of split points. Once there are enough tablets to spread a table across the entire cluster, additional splits may not improve performance, and may create unnecessary bookkeeping. The distribution of data may change over time. For example, if row data contains date information, and data is continually added and removed to maintain a window of current information, tablets for older rows may be empty.

Accumulo supports tablet merging, which can be used to reduce the number of split points. The following command will merge all rows from "A" to "Z" into a single tablet:

```
root@myinstance> merge -t myTable -s A -e Z
```

If the result of a merge produces a tablet that is larger than the configured split size, the tablet may be split by the tablet server. Be sure to increase your tablet size prior to any merges if the goal is to have larger tablets:

```
root@myinstance> config -t myTable -s table.split.threshold=2G
```

In order to merge small tablets, you can ask Accumulo to merge sections of a table smaller than a given size.

```
root@myinstance> merge -t myTable -s 100M
```

By default, small tablets will not be merged into tablets that are already larger than the given size. This can leave isolated small tablets. To force small tablets to be merged into larger tablets use the "--force" option:

```
root@myinstance> merge -t myTable -s 100M --force
```

Merging away small tablets works on one section at a time. If your table contains many sections of small split points, or you are attempting to change the split size of the entire table, it will be faster to set the split point and merge the entire table:

```
root@myinstance> config -t myTable -s table.split.threshold=256M
root@myinstance> merge -t myTable
```

6.9 Delete Range

Consider an indexing scheme that uses date information in each row. For example "20110823-15:20:25.013" might be a row that specifies a date and time. In some cases, we might like to delete rows based on this date, say to remove all the data older than the current year. Accumulo supports a delete range operation which efficiently removes data between two rows. For example:

```
root@myinstance> deleterange -t myTable -s 2010 -e 2011
```

This will delete all rows starting with “2010” and it will stop at any row starting “2011”. You can delete any data prior to 2011 with:

```
root@myinstance> deleterange -t myTable -e 2011 --force
```

The shell will not allow you to delete an unbounded range (no start) unless you provide the “--force” option.

Range deletion is implemented using splits at the given start/end positions, and will affect the number of splits in the table.

6.10 Cloning Tables

A new table can be created that points to an existing table’s data. This is a very quick metadata operation, no data is actually copied. The cloned table and the source table can change independently after the clone operation. One use case for this feature is testing. For example to test a new filtering iterator, clone the table, add the filter to the clone, and force a major compaction. To perform a test on less data, clone a table and then use delete range to efficiently remove a lot of data from the clone. Another use case is generating a snapshot to guard against human error. To create a snapshot, clone a table and then disable write permissions on the clone.

The clone operation will point to the source table’s files. This is why the flush option is present and is enabled by default in the shell. If the flush option is not enabled, then any data the source table currently has in memory will not exist in the clone.

A cloned table copies the configuration of the source table. However the permissions of the source table are not copied to the clone. After a clone is created, only the user that created the clone can read and write to it.

In the following example we see that data inserted after the clone operation is not visible in the clone.

```
root@a14> createtable people
root@a14 people> insert 890435 name last Doe
root@a14 people> insert 890435 name first John
root@a14 people> clonetable people test
root@a14 people> insert 890436 name first Jane
root@a14 people> insert 890436 name last Doe
root@a14 people> scan
890435 name:first []    John
890435 name:last []    Doe
890436 name:first []    Jane
890436 name:last []    Doe
root@a14 people> table test
root@a14 test> scan
890435 name:first []    John
890435 name:last []    Doe
root@a14 test>
```

The du command in the shell shows how much space a table is using in HDFS. This command can also show how much overlapping space two cloned tables have in HDFS. In the example below du shows table ci is using 428M. Then ci is cloned to cic and du shows that both tables share 428M. After three entries are inserted into cic and its flushed, du shows the two tables still share 428M but cic has 226 bytes to itself. Finally, table cic is compacted and then du shows that each table uses 428M.

```
root@a14> du ci
             428,482,573 [ci]
root@a14> clonetable ci cic
root@a14> du ci cic
             428,482,573 [ci, cic]
root@a14> table cic
root@a14 cic> insert r1 cf1 cq1 v1
root@a14 cic> insert r1 cf1 cq2 v2
root@a14 cic> insert r1 cf1 cq3 v3
root@a14 cic> flush -t cic -w
27 15:00:13,908 [shell.Shell] INFO : Flush of table cic completed.
root@a14 cic> du ci cic
             428,482,573 [ci, cic]
                     226 [cic]
root@a14 cic> compact -t cic -w
27 15:00:35,871 [shell.Shell] INFO : Compacting table ...
27 15:03:03,303 [shell.Shell] INFO : Compaction of table cic completed for given range
root@a14 cic> du ci cic
             428,482,573 [ci]
             428,482,612 [cic]
root@a14 cic>
```

6.11 Exporting Tables

Accumulo supports exporting tables for the purpose of copying tables to another cluster. Exporting and importing tables preserves the tables configuration, splits, and logical time. Tables are exported and then copied via the hadoop distcp command. To export a table, it must be offline and stay offline while discp runs. The reason it needs to stay offline is to prevent files from being deleted. A table can be cloned and the clone taken offline inorder to avoid losing access to the table. See docs/examples/README.export for an example.

Chapter 7

Table Design

7.1 Basic Table

Since Accumulo tables are sorted by row ID, each table can be thought of as being indexed by the row ID. Lookups performed by row ID can be executed quickly, by doing a binary search, first across the tablets, and then within a tablet. Clients should choose a row ID carefully in order to support their desired application. A simple rule is to select a unique identifier as the row ID for each entity to be stored and assign all the other attributes to be tracked to be columns under this row ID. For example, if we have the following data in a comma-separated file:

```
    userid,age,address,account-balance
```

We might choose to store this data using the userid as the rowID, the column name in the column family, and a blank column qualifier:

```
Mutation m = new Mutation(userid);
final String column_qualifier = "";
m.put("age", column_qualifier, age);
m.put("address", column_qualifier, address);
m.put("balance", column_qualifier, account_balance);

writer.add(m);
```

We could then retrieve any of the columns for a specific userid by specifying the userid as the range of a scanner and fetching specific columns:

```
Range r = new Range(userid, userid); // single row
Scanner s = conn.createScanner("userdata", auths);
s.setRange(r);
s.fetchColumnFamily(new Text("age"));

for(Entry<Key,Value> entry : s)
    System.out.println(entry.getValue().toString());
```

7.2 RowID Design

Often it is necessary to transform the rowID in order to have rows ordered in a way that is optimal for anticipated access patterns. A good example of this is reversing the order of components of internet domain names in order to group rows of the same parent domain together:

```
com.google.code
com.google.labs
com.google.mail
com.yahoo.mail
com.yahoo.research
```

Some data may result in the creation of very large rows - rows with many columns. In this case the table designer may wish to split up these rows for better load balancing while keeping them sorted together for scanning purposes. This can be done by appending a random substring at the end of the row:

```
com.google.code_00
com.google.code_01
com.google.code_02
com.google.labs_00
com.google.mail_00
com.google.mail_01
```

It could also be done by adding a string representation of some period of time such as date to the week or month:

```
com.google.code_201003
com.google.code_201004
com.google.code_201005
com.google.labs_201003
com.google.mail_201003
com.google.mail_201004
```

Appending dates provides the additional capability of restricting a scan to a given date range.

7.3 Lexicoders

Since Keys in Accumulo are sorted lexicographically by default, it's often useful to encode common data types into a byte format in which their sort order corresponds to the sort order in their native form. An example of this is encoding dates and numerical data so that they can be better seeked or searched in ranges.

The lexicoders are a standard and extensible way of encoding Java types. Here's an example of a lexicoder that encodes a java Date object so that it sorts lexicographically:

```
// create new date lexicoder
DateLexicoder dateEncoder = new DateLexicoder();

// truncate time to hours
long epoch = System.currentTimeMillis();
```

```
Date hour = new Date(epoch - (epoch % 3600000));

// encode the rowId so that it is sorted lexicographically
Mutation mutation = new Mutation(dateEncoder.encode(hour));
mutation.put(new Text("colf"), new Text("colq"), new Value(new byte[]{}));
```

If we want to return the most recent date first, we can reverse the sort order with the reverse lexicoder:

```
// create new date lexicoder and reverse lexicoder
DateLexicoder dateEncoder = new DateLexicoder();
ReverseLexicoder reverseEncoder = new ReverseLexicoder(dateEncoder);

// truncate date to hours
long epoch = System.currentTimeMillis();
Date hour = new Date(epoch - (epoch % 3600000));

// encode the rowId so that it sorts in reverse lexicographic order
Mutation mutation = new Mutation(reverseEncoder.encode(hour));
mutation.put(new Text("colf"), new Text("colq"), new Value(new byte[]{}));
```

7.4 Indexing

In order to support lookups via more than one attribute of an entity, additional indexes can be built. However, because Accumulo tables can support any number of columns without specifying them beforehand, a single additional index will often suffice for supporting lookups of records in the main table. Here, the index has, as the rowID, the Value or Term from the main table, the column families are the same, and the column qualifier of the index table contains the rowID from the main table.

Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		
Term	Field Name	MainRowID			

Note: We store rowIDs in the column qualifier rather than the Value so that we can have more than one rowID associated with a particular term within the index. If we stored this in the Value we would only see one of the rows in which the value appears since Accumulo is configured by default to return the one most recent value associated with a key.

Lookups can then be done by scanning the Index Table first for occurrences of the desired values in the columns specified, which returns a list of row ID from the main table. These can then be used to retrieve each matching record, in their entirety, or a subset of their columns, from the Main Table.

To support efficient lookups of multiple rowIDs from the same table, the Accumulo client library provides a BatchScanner. Users specify a set of Ranges to the BatchScanner, which performs the lookups in multiple threads to multiple servers and returns an Iterator over all the rows retrieved. The rows returned are NOT in sorted order, as is the case with the basic Scanner interface.

```
// first we scan the index for IDs of rows matching our query

Text term = new Text("mySearchTerm");

HashSet<Range> matchingRows = new HashSet<Range>();

Scanner indexScanner = createScanner("index", auths);
indexScanner.setRange(new Range(term, term));

// we retrieve the matching rowIDs and create a set of ranges
for(Entry<Key,Value> entry : indexScanner)
    matchingRows.add(new Range(entry.getKey().getColumnQualifier()));

// now we pass the set of rowIDs to the batch scanner to retrieve them
BatchScanner bscan = conn.createBatchScanner("table", auths, 10);

bscan.setRanges(matchingRows);
bscan.fetchColumnFamily(new Text("attributes"));

for(Entry<Key,Value> entry : bscan)
    System.out.println(entry.getValue());
```

One advantage of the dynamic schema capabilities of Accumulo is that different fields may be indexed into the same physical table. However, it may be necessary to create different index tables if the terms must be formatted differently in order to maintain proper sort order. For example, real numbers must be formatted differently than their usual notation in order to be sorted correctly. In these cases, usually one index per unique data type will suffice.

7.5 Entity-Attribute and Graph Tables

Accumulo is ideal for storing entities and their attributes, especially of the attributes are sparse. It is often useful to join several datasets together on common entities within the same table. This can allow for the representation of graphs, including nodes, their attributes, and connections to other nodes.

Rather than storing individual events, Entity-Attribute or Graph tables store aggregate information about the entities involved in the events and the relationships between entities. This is often preferrable when single events aren't very useful and when a continuously updated summarization is desired.

The physical schema for an entity-attribute or graph table is as follows:

Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		
EntityID	Attribute Name	Attribute Value			Weight
EntityID	Edge Type	Related EntityID			Weight

For example, to keep track of employees, managers and products the following entity-attribute table could be used. Note that the weights are not always necessary and are set to 0 when not used.

RowID	**ColumnFamily**	**ColumnQualifier**	**Value**
$E001$	*name*	*bob*	0
$E001$	*department*	*sales*	0
$E001$	*hire_date*	20030102	0
$E001$	*units_sold*	$P001$	780
$E002$	*name*	*george*	0
$E002$	*department*	*sales*	0
$E002$	*manager_of*	$E001$	0
$E002$	*manager_of*	$E003$	0
$E003$	*name*	*harry*	0
$E003$	*department*	*accounts_recv*	0
$E003$	*hire_date*	20000405	0
$E003$	*units_sold*	$P002$	566
$E003$	*units_sold*	$P001$	232
$P001$	*product_name*	*nike_airs*	0
$P001$	*product_type*	*shoe*	0
$P001$	*in_stock*	*germany*	900
$P001$	*in_stock*	*brazil*	200
$P002$	*product_name*	*basic_jacket*	0
$P002$	*product_type*	*clothing*	0
$P002$	*in_stock*	*usa*	3454
$P002$	*in_stock*	*germany*	700

To allow efficient updating of edge weights, an aggregating iterator can be configured to add the value of all mutations applied with the same key. These types of tables can easily be created from raw events by simply extracting the entities, attributes, and relationships from individual events and inserting the keys into Accumulo each with a count of 1. The aggregating iterator will take care of maintaining the edge weights.

7.6 Document-Partitioned Indexing

Using a simple index as described above works well when looking for records that match one of a set of given criteria. When looking for records that match more than one criterion simultaneously, such as when looking for documents that contain all of the words 'the' and 'white' and 'house', there are several issues.

First is that the set of all records matching any one of the search terms must be sent to the client, which incurs a lot of network traffic. The second problem is that the client is responsible for performing set intersection on the sets of records returned to eliminate all but the records matching all search terms. The memory of the client may easily be overwhelmed during this operation.

For these reasons Accumulo includes support for a scheme known as sharded indexing, in which these set operations can be performed at the TabletServers and decisions about which records to include in the result set can be made without incurring network traffic.

This is accomplished via partitioning records into bins that each reside on at most one TabletServer, and then creating an index of terms per record within each bin as follows:

Key					Value
Row ID	Column			Timestamp	
	Family	Qualifier	Visibility		
BinID	Term	DocID			Weight

Documents or records are mapped into bins by a user-defined ingest application. By storing the BinID as the RowID we ensure that all the information for a particular bin is contained in a single tablet and hosted on a single TabletServer since Accumulo never splits rows across tablets. Storing the Terms as column families serves to enable fast lookups of all the documents within this bin that contain the given term.

Finally, we perform set intersection operations on the TabletServer via a special iterator called the Intersecting Iterator. Since documents are partitioned into many bins, a search of all documents must search every bin. We can use the BatchScanner to scan all bins in parallel. The Intersecting Iterator should be enabled on a BatchScanner within user query code as follows:

```
Text[] terms = {new Text("the"), new Text("white"), new Text("house")};

BatchScanner bs = conn.createBatchScanner(table, auths, 20);
```

```
IteratorSetting iter = new IteratorSetting(20, "ii", IntersectingIterator.class);
IntersectingIterator.setColumnFamilies(iter, terms);
bs.addScanIterator(iter);
bs.setRanges(Collections.singleton(new Range()));

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

This code effectively has the BatchScanner scan all tablets of a table, looking for documents that match all the given terms. Because all tablets are being scanned for every query, each query is more expensive than other Accumulo scans, which typically involve a small number of TabletServers. This reduces the number of concurrent queries supported and is subject to what is known as the 'straggler' problem in which every query runs as slow as the slowest server participating.

Of course, fast servers will return their results to the client which can display them to the user immediately while they wait for the rest of the results to arrive. If the results are unordered this is quite effective as the first results to arrive are as good as any others to the user.

Chapter 8

High-Speed Ingest

Accumulo is often used as part of a larger data processing and storage system. To maximize the performance of a parallel system involving Accumulo, the ingestion and query components should be designed to provide enough parallelism and concurrency to avoid creating bottlenecks for users and other systems writing to and reading from Accumulo. There are several ways to achieve high ingest performance.

8.1 Pre-Splitting New Tables

New tables consist of a single tablet by default. As mutations are applied, the table grows and splits into multiple tablets which are balanced by the Master across TabletServers. This implies that the aggregate ingest rate will be limited to fewer servers than are available within the cluster until the table has reached the point where there are tablets on every TabletServer.

Pre-splitting a table ensures that there are as many tablets as desired available before ingest begins to take advantage of all the parallelism possible with the cluster hardware. Tables can be split anytime by using the shell:

```
user@myinstance mytable> addsplits -sf /local_splitfile -t mytable
```

For the purposes of providing parallelism to ingest it is not necessary to create more tablets than there are physical machines within the cluster as the aggregate ingest rate is a function of the number of physical machines. Note that the aggregate ingest rate is still subject to the number of machines running ingest clients, and the distribution of rowIDs across the table. The aggregation ingest rate will be suboptimal if there are many inserts into a small number of rowIDs.

8.2 Multiple Ingester Clients

Accumulo is capable of scaling to very high rates of ingest, which is dependent upon not just the number of TabletServers in operation but also the number of ingest clients. This is because a single client, while capable of batching mutations and sending them to all TabletServers, is ultimately limited by the amount of data that can be processed on a single machine. The aggregate ingest rate will scale linearly with the number of clients up to the point at which either the aggregate I/O of TabletServers or total network bandwidth capacity is reached.

In operational settings where high rates of ingest are paramount, clusters are often configured to dedicate some number of machines solely to running Ingester Clients. The exact ratio of clients to TabletServers necessary for optimum ingestion rates will vary according to the distribution of resources per machine and by data type.

8.3 Bulk Ingest

Accumulo supports the ability to import files produced by an external process such as MapReduce into an existing table. In some cases it may be faster to load data this way rather than via ingesting through clients using BatchWriters. This allows a large number of machines to format data the way Accumulo expects. The new files can then simply be introduced to Accumulo via a shell command.

To configure MapReduce to format data in preparation for bulk loading, the job should be set to use a range partitioner instead of the default hash partitioner. The range partitioner uses the split points of the Accumulo table that will receive the data. The split points can be obtained from the shell and used by the MapReduce RangePartitioner. Note that this is only useful if the existing table is already split into multiple tablets.

```
user@myinstance mytable> getsplits
aa
ab
ac
...
zx
zy
zz
```

Run the MapReduce job, using the AccumuloFileOutputFormat to create the files to be introduced to Accumulo. Once this is complete, the files can be added to Accumulo via the shell:

```
user@myinstance mytable> importdirectory /files_dir /failures
```

Note that the paths referenced are directories within the same HDFS instance over which Accumulo is running. Accumulo places any files that failed to be added to the second directory specified.

A complete example of using Bulk Ingest can be found at
accumulo/docs/examples/README.bulkIngest

8.4 Logical Time for Bulk Ingest

Logical time is important for bulk imported data, for which the client code may be choosing a timestamp. At bulk import time, the user can choose to enable logical time for the set of files being imported. When its enabled, Accumulo uses a specialized system iterator to lazily set times in a bulk imported file. This mechanism guarantees that times set by unsynchronized multi-node applications (such as those running on MapReduce) will maintain some semblance of causal ordering. This mitigates the problem of the time being wrong on the system that created the file for bulk import. These times are not set when the file is imported, but whenever it is read by scans or compactions. At import, a time is obtained and always used by the specialized system iterator to set that time.

The timestamp assigned by Accumulo will be the same for every key in the file. This could cause problems if the file contains multiple keys that are identical except for the timestamp. In this case, the sort order of the keys will be undefined. This could occur if an insert and an update were in the same bulk import file.

8.5 MapReduce Ingest

It is possible to efficiently write many mutations to Accumulo in parallel via a MapReduce job. In this scenario the MapReduce is written to process data that lives in HDFS and write mutations to Accumulo using the AccumuloOutputFormat. See the MapReduce section under Analytics for details.

An example of using MapReduce can be found under
accumulo/docs/examples/README.mapred

Chapter 9

Analytics

Accumulo supports more advanced data processing than simply keeping keys sorted and performing efficient lookups. Analytics can be developed by using MapReduce and Iterators in conjunction with Accumulo tables.

9.1 MapReduce

Accumulo tables can be used as the source and destination of MapReduce jobs. To use an Accumulo table with a MapReduce job (specifically with the new Hadoop API as of version 0.20), configure the job parameters to use the AccumuloInputFormat and AccumuloOutputFormat. Accumulo specific parameters can be set via these two format classes to do the following:

- Authenticate and provide user credentials for the input
- Restrict the scan to a range of rows
- Restrict the input to a subset of available columns

9.1.1 Mapper and Reducer classes

To read from an Accumulo table create a Mapper with the following class parameterization and be sure to configure the AccumuloInputFormat.

```
class MyMapper extends Mapper<Key,Value,WritableComparable,Writable> {
    public void map(Key k, Value v, Context c) {
        // transform key and value data here
    }
}
```

To write to an Accumulo table, create a Reducer with the following class parameterization and be sure to configure the AccumuloOutputFormat. The key emitted from the Reducer identifies the table to which the mutation is sent. This allows a single Reducer to write to more than one table if desired. A default table can be configured using the AccumuloOutputFormat, in which case the output table name does not have to be passed to the Context object within the Reducer.

```
class MyReducer extends Reducer<WritableComparable, Writable, Text, Mutation> {
    public void reduce(WritableComparable key, Iterable<Text> values, Context c) {
        Mutation m;
        // create the mutation based on input key and value
        c.write(new Text("output-table"), m);
    }
}
```

The Text object passed as the output should contain the name of the table to which this mutation should be applied. The Text can be null in which case the mutation will be applied to the default table name specified in the AccumuloOutputFormat options.

9.1.2 AccumuloInputFormat options

```
Job job = new Job(getConf());
AccumuloInputFormat.setInputInfo(job,
        "user",
        "passwd".getBytes(),
        "table",
        new Authorizations());

AccumuloInputFormat.setZooKeeperInstance(job, "myinstance",
        "zooserver-one,zooserver-two");
```

Optional Settings:

To restrict Accumulo to a set of row ranges:

```
ArrayList<Range> ranges = new ArrayList<Range>();
// populate array list of row ranges ...
AccumuloInputFormat.setRanges(job, ranges);
```

To restrict Accumulo to a list of columns:

```
ArrayList<Pair<Text,Text>> columns = new ArrayList<Pair<Text,Text>>();
// populate list of columns
AccumuloInputFormat.fetchColumns(job, columns);
```

To use a regular expression to match row IDs:

```
IteratorSetting is = new IteratorSetting(30, RexExFilter.class);
RegExFilter.setRegexs(is, ".*suffix", null, null, null, true);
AccumuloInputFormat.addIterator(job, is);
```

9.1.3 AccumuloMultiTableInputFormat options

The AccumuloMultiTableInputFormat allows the scanning over multiple tables in a single MapReduce job. Separate ranges, columns, and iterators can be used for each table.

```
InputTableConfig tableOneConfig = new InputTableConfig();
InputTableConfig tableTwoConfig = new InputTableConfig();
```

To set the configuration objects on the job:

```
Map<String, InputTableConfig> configs = new HashMap<String,InputTableConfig>();
configs.put("table1", tableOneConfig);
configs.put("table2", tableTwoConfig);
AccumuloMultiTableInputFormat.setInputTableConfigs(job, configs);
```

Optional settings:

To restrict to a set of ranges:

```
ArrayList<Range> tableOneRanges = new ArrayList<Range>();
ArrayList<Range> tableTwoRanges = new ArrayList<Range>();
// populate array lists of row ranges for tables...
tableOneConfig.setRanges(tableOneRanges);
tableTwoConfig.setRanges(tableTwoRanges);
```

To restrict Accumulo to a list of columns:

```
ArrayList<Pair<Text,Text>> tableOneColumns = new ArrayList<Pair<Text,Text>>();
ArrayList<Pair<Text,Text>> tableTwoColumns = new ArrayList<Pair<Text,Text>>();
// populate lists of columns for each of the tables ...
tableOneConfig.fetchColumns(tableOneColumns);
tableTwoConfig.fetchColumns(tableTwoColumns);
```

To set scan iterators:

```
List<IteratorSetting> tableOneIterators = new ArrayList<IteratorSetting>();
List<IteratorSetting> tableTwoIterators = new ArrayList<IteratorSetting>();
// populate the lists of iterator settings for each of the tables ...
tableOneConfig.setIterators(tableOneIterators);
tableTwoConfig.setIterators(tableTwoIterators);
```

The name of the table can be retrieved from the input split:

```
class MyMapper extends Mapper<Key,Value,WritableComparable,Writable> {
    public void map(Key k, Value v, Context c) {
        RangeInputSplit split = (RangeInputSplit)c.getInputSplit();
        String tableName = split.getTableName();
        // do something with table name
    }
}
```

9.1.4 AccumuloOutputFormat options

```
boolean createTables = true;
String defaultTable = "mytable";

AccumuloOutputFormat.setOutputInfo(job,
        "user",
        "passwd".getBytes(),
        createTables,
        defaultTable);

AccumuloOutputFormat.setZooKeeperInstance(job, "myinstance",
        "zooserver-one,zooserver-two");
```

Optional Settings:

```
AccumuloOutputFormat.setMaxLatency(job, 300000); // milliseconds
AccumuloOutputFormat.setMaxMutationBufferSize(job, 50000000); // bytes
```

An example of using MapReduce with Accumulo can be found at
accumulo/docs/examples/README.mapred

9.2 Combiners

Many applications can benefit from the ability to aggregate values across common keys. This can be done via Combiner iterators and is similar to the Reduce step in MapReduce. This provides the ability to define online, incrementally updated analytics without the overhead or latency associated with batch-oriented MapReduce jobs.

All that is needed to aggregate values of a table is to identify the fields over which values will be grouped, insert mutations with those fields as the key, and configure the table with a combining iterator that supports the summarizing operation desired.

The only restriction on an combining iterator is that the combiner developer should not assume that all values for a given key have been seen, since new mutations can be inserted at anytime. This precludes using the total number of values in the aggregation such as when calculating an average, for example.

9.2.1 Feature Vectors

An interesting use of combining iterators within an Accumulo table is to store feature vectors for use in machine learning algorithms. For example, many algorithms such as k-means clustering, support vector machines, anomaly detection, etc. use the concept of a feature vector and the calculation of distance metrics to learn a particular model. The columns in an Accumulo table can be used to efficiently store sparse features and their weights to be incrementally updated via the use of an combining iterator.

9.3 Statistical Modeling

Statistical models that need to be updated by many machines in parallel could be similarly stored within an Accumulo table. For example, a MapReduce job that is iteratively updating a global statistical model could have each map or reduce worker reference the parts of the model to be read and updated through an embedded Accumulo client.

Using Accumulo this way enables efficient and fast lookups and updates of small pieces of information in a random access pattern, which is complementary to MapReduce's sequential access model.

Chapter 10

Security

Accumulo extends the BigTable data model to implement a security mechanism known as cell-level security. Every key-value pair has its own security label, stored under the column visibility element of the key, which is used to determine whether a given user meets the security requirements to read the value. This enables data of various security levels to be stored within the same row, and users of varying degrees of access to query the same table, while preserving data confidentiality.

10.1 Security Label Expressions

When mutations are applied, users can specify a security label for each value. This is done as the Mutation is created by passing a ColumnVisibility object to the put() method:

```
Text rowID = new Text("row1");
Text colFam = new Text("myColFam");
Text colQual = new Text("myColQual");
ColumnVisibility colVis = new ColumnVisibility("public");
long timestamp = System.currentTimeMillis();

Value value = new Value("myValue");

Mutation mutation = new Mutation(rowID);
mutation.put(colFam, colQual, colVis, timestamp, value);
```

10.2 Security Label Expression Syntax

Security labels consist of a set of user-defined tokens that are required to read the value the label is associated with. The set of tokens required can be specified using syntax that supports logical AND and OR combinations of tokens, as well as nesting groups of tokens together.

For example, suppose within our organization we want to label our data values with security labels defined in terms of user roles. We might have tokens such as:

```
admin
audit
system
```

These can be specified alone or combined using logical operators:

```
// Users must have admin privileges:
admin

// Users must have admin and audit privileges
admin&audit

// Users with either admin or audit privileges
admin|audit

// Users must have audit and one or both of admin or system
(admin|system)&audit
```

When both `|` and `&` operators are used, parentheses must be used to specify precedence of the operators.

10.3 Authorization

When clients attempt to read data from Accumulo, any security labels present are examined against the set of authorizations passed by the client code when the Scanner or BatchScanner are created. If the authorizations are determined to be insufficient to satisfy the security label, the value is suppressed from the set of results sent back to the client.

Authorizations are specified as a comma-separated list of tokens the user possesses:

```
// user possess both admin and system level access
Authorization auths = new Authorization("admin","system");

Scanner s = connector.createScanner("table", auths);
```

10.4 User Authorizations

Each Accumulo user has a set of associated security labels. To manipulate these in the shell while using the default authorizor, use the setuaths and getauths commands. These may also be modified for the default authorizor using the java security operations API.

When a user creates a scanner a set of Authorizations is passed. If the authorizations passed to the scanner are not a subset of the users authorizations, then an exception will be thrown.

To prevent users from writing data they can not read, add the visibility constraint to a table. Use the -evc option in the createtable shell command to enable this constraint. For existing tables use the following shell command to enable the visibility constraint. Ensure the constraint number does not conflict with any existing constraints.

```
config -t table -s table.constraint.1=org.apache.accumulo.core.security.VisibilityConstraint
```

Any user with the alter table permission can add or remove this constraint. This constraint is not applied to bulk imported data, if this a concern then disable the bulk import permission.

10.5 Pluggable Security

New in 1.5 of Accumulo is a pluggable security mechanism. It can be broken into three actions- authentication, authorization, and permission handling. By default all of these are handled in Zookeeper, which is how things were handled in Accumulo 1.4 and before. It is worth noting at this point, that it is a new feature in 1.5 and may be adjusted in future releases without the standard deprecation cycle.

Authentication simply handles the ability for a user to verify their integrity. A combination of principal and authentication token are used to verify a user is who they say they are. An authentication token should be constructed, either directly through it's constructor, but it is advised to use the init(Property) method to populate an authentication token. It is expected that a user knows what the appropriate token to use for their system is. The default token is PasswordToken.

Once a user is authenticated by the Authenticator, the user has access to the other actions within Accumulo. All actions in Accumulo are ACLed, and this ACL check is handled by the Permission Handler. This is what manages all of the permissions, which are divided in system and per table level. From there, if a user is doing an action which requires authorizations, the Authorizor is queried to determine what authorizations the user has.

This setup allows a variety of different mechanisms to be used for handling different aspects of Accumulo's security. A system like Kerberos can be used for authentication, then a system like LDAP could be used to determine if a user has a specific permission, and then it may default back to the default ZookeeperAuthorizor to determine what Authorizations a user is ultimately allowed to use. This is a pluggable system so custom components can be created depending on your need.

10.6 Secure Authorizations Handling

For applications serving many users, it is not expected that an Accumulo user will be created for each application user. In this case an Accumulo user with all authorizations needed by any of the applications users must be created. To service queries, the application should create a scanner with the application user's authorizations. These authorizations could be obtained from a trusted 3rd party.

Often production systems will integrate with Public-Key Infrastructure (PKI) and designate client code within the query layer to negotiate with PKI servers in order to authenticate users and retrieve their authorization tokens (credentials). This requires users to specify only the information necessary to authenticate themselves to the system. Once user identity is established, their credentials can be accessed by the client code and passed to Accumulo outside of the reach of the user.

10.7 Query Services Layer

Since the primary method of interaction with Accumulo is through the Java API, production environments often call for the implementation of a Query layer. This can be done using web services in containers such as Apache Tomcat, but is not a requirement. The Query Services Layer provides a mechanism for providing a platform on which user facing applications can be built. This allows the application designers to isolate potentially complex query logic, and enables a convenient point at which to perform essential security functions.

Several production environments choose to implement authentication at this layer, where users identifiers are used to retrieve their access credentials which are then cached within the query layer and presented to Accumulo through the Authorizations mechanism.

Typically, the query services layer sits between Accumulo and user workstations.

Chapter 11

Administration

11.1 Hardware

Because we are running essentially two or three systems simultaneously layered across the cluster: HDFS, Accumulo and MapReduce, it is typical for hardware to consist of 4 to 8 cores, and 8 to 32 GB RAM. This is so each running process can have at least one core and 2 - 4 GB each.

One core running HDFS can typically keep 2 to 4 disks busy, so each machine may typically have as little as 2 x 300GB disks and as much as 4 x 1TB or 2TB disks.

It is possible to do with less than this, such as with 1u servers with 2 cores and 4GB each, but in this case it is recommended to only run up to two processes per machine - i.e. DataNode and TabletServer or DataNode and MapReduce worker but not all three. The constraint here is having enough available heap space for all the processes on a machine.

11.2 Network

Accumulo communicates via remote procedure calls over TCP/IP for both passing data and control messages. In addition, Accumulo uses HDFS clients to communicate with HDFS. To achieve good ingest and query performance, sufficient network bandwidth must be available between any two machines.

In addition to needing access to ports associated with HDFS and ZooKeeper, Accumulo will use the following default ports. Please make sure that they are open, or change their value in conf/accumulo-site.xml.

Port	Description	Property Name
4445	Shutdown Port (Accumulo MiniCluster)	n/a
4560	Accumulo monitor (for centralized log display)	monitor.port.log4j
9997	Tablet Server	tserver.port.client
9999	Master Server	master.port.client
12234	Accumulo Tracer	trace.port.client
42424	Accumulo Proxy Server	n/a
50091	Accumulo GC	gc.port.client
50095	Accumulo HTTP monitor	monitor.port.client

Table 11.1: Accumulo default ports

In addition, the user can provide '0' and an ephemeral port will be chosen instead. This ephemeral port is likely to be unique and not already bound. Thus, configuring ports to use '0' instead of an explicit value, should, in most cases, work around any issues of running multiple distinct Accumulo instances (or any other process which tries to use the same default ports) on the same hardware.

11.3 Installation

Choose a directory for the Accumulo installation. This directory will be referenced by the environment variable `$ACCUMULO_HOME`. Run the following:

```
$ tar xzf accumulo-1.5.0-bin.tar.gz    # unpack to subdirectory
$ mv accumulo-1.5.0 $ACCUMULO_HOME # move to desired location
```

Repeat this step at each machine within the cluster. Usually all machines have the same `$ACCUMULO_HOME`.

11.4 Dependencies

Accumulo requires HDFS and ZooKeeper to be configured and running before starting. Password-less SSH should be configured between at least the Accumulo master and TabletServer machines. It is also a good idea to run Network Time Protocol (NTP) within the cluster to ensure nodes' clocks don't get too out of sync, which can cause problems with automatically timestamped data.

11.5 Configuration

Accumulo is configured by editing several Shell and XML files found in `$ACCUMULO_HOME/conf`. The structure closely resembles Hadoop's configuration files.

11.5.1 Edit conf/accumulo-env.sh

Accumulo needs to know where to find the software it depends on. Edit accumulo-env.sh and specify the following:

1. Enter the location of the installation directory of Accumulo for `$ACCUMULO_HOME`
2. Enter your system's Java home for `$JAVA_HOME`
3. Enter the location of Hadoop for `$HADOOP_PREFIX`
4. Choose a location for Accumulo logs and enter it for `$ACCUMULO_LOG_DIR`
5. Enter the location of ZooKeeper for `$ZOOKEEPER_HOME`

By default Accumulo TabletServers are set to use 1GB of memory. You may change this by altering the value of `$ACCUMULO_TSERVER_OPTS`. Note the syntax is that of the Java JVM command line options. This value should be less than the physical memory of the machines running TabletServers.

There are similar options for the master's memory usage and the garbage collector process. Reduce these if they exceed the physical RAM of your hardware and increase them, within the bounds of the physical RAM, if a process fails because of insufficient memory.

Note that you will be specifying the Java heap space in accumulo-env.sh. You should make sure that the total heap space used for the Accumulo tserver and the Hadoop DataNode and TaskTracker is less than the available memory on each slave node in the cluster. On large clusters, it is recommended that the Accumulo master, Hadoop NameNode, secondary NameNode, and Hadoop JobTracker all be run on separate machines to allow them to use more heap space. If you are running these on the same machine on a small cluster, likewise make sure their heap space settings fit within the available memory.

11.5.2 Native Map

The tablet server uses a data structure called a MemTable to store sorted key/value pairs in memory when they are first received from the client. When a minor compaction occurs, this data structure is written to HDFS. The MemTable will default to using memory in the JVM but a JNI version, called the native map, can be used to significantly speed up performance by utilizing the memory space of the native operating system. The native map also avoids the performance implications brought on by garbage collection in the JVM by causing it to pause much less frequently.

32-bit and 64-bit Linux versions of the native map ship with the Accumulo dist package. For other operating systems, the native map can be built from the codebase in two ways- from maven or from the Makefile.

1. Build from maven using the following command: `mvn clean package -Pnative.`
2. Build from the c++ source by running `make` in the `$ACCUMULO_HOME/server/src/main/c++` directory.

After building the native map from the source, you will find the artifact in `$ACCUMULO_HOME/lib/native.` Upon starting up, the tablet server will look in this directory for the map library. If the file is renamed or moved from its target directory, the tablet server may not be able to find it.

11.5.3 Cluster Specification

On the machine that will serve as the Accumulo master:

1. Write the IP address or domain name of the Accumulo Master to the `$ACCUMULO_HOME/conf/masters` file.
2. Write the IP addresses or domain name of the machines that will be TabletServers in `$ACCUMULO_HOME/conf/slaves`, one per line.

Note that if using domain names rather than IP addresses, DNS must be configured properly for all machines participating in the cluster. DNS can be a confusing source of errors.

11.5.4 Accumulo Settings

Specify appropriate values for the following settings in
`$ACCUMULO_HOME/conf/accumulo-site.xml` :

```
<property>
    <name>instance.zookeeper.host</name>
    <value>zooserver-one:2181,zooserver-two:2181</value>
    <description>list of zookeeper servers</description>
</property>
```

This enables Accumulo to find ZooKeeper. Accumulo uses ZooKeeper to coordinate settings between processes and helps finalize TabletServer failure.

```
<property>
    <name>instance.secret</name>
    <value>DEFAULT</value>
</property>
```

The instance needs a secret to enable secure communication between servers. Configure your secret and make sure that the `accumulo-site.xml` file is not readable to other users.

Some settings can be modified via the Accumulo shell and take effect immediately, but some settings require a process restart to take effect. See the configuration documentation (available in the docs directory of the tarball and in Appendix A) for details.

11.5.5 Deploy Configuration

Copy the masters, slaves, accumulo-env.sh, and if necessary, accumulo-site.xml from the `$ACCUMULO_HOME/conf/` directory on the master to all the machines specified in the slaves file.

11.6 Initialization

Accumulo must be initialized to create the structures it uses internally to locate data across the cluster. HDFS is required to be configured and running before Accumulo can be initialized.

Once HDFS is started, initialization can be performed by executing `$ACCUMULO_HOME/bin/accumulo init` . This script will prompt for a name for this instance of Accumulo. The instance name is used to identify a set of tables and instance-specific settings. The script will then write some information into HDFS so Accumulo can start properly.

The initialization script will prompt you to set a root password. Once Accumulo is initialized it can be started.

11.7 Running

11.7.1 Starting Accumulo

Make sure Hadoop is configured on all of the machines in the cluster, including access to a shared HDFS instance. Make sure HDFS and ZooKeeper are running. Make sure ZooKeeper is configured and running on at least one machine in the cluster. Start Accumulo using the `bin/start-all.sh` script.

To verify that Accumulo is running, check the Status page as described under Monitoring. In addition, the Shell can provide some information about the status of tables via reading the metadata tables.

11.7.2 Stopping Accumulo

To shutdown cleanly, run `bin/stop-all.sh` and the master will orchestrate the shutdown of all the tablet servers. Shutdown waits for all minor compactions to finish, so it may take some time for particular configurations.

11.7.3 Adding a Node

Update your `$ACCUMULO_HOME/conf/slaves` (or `$ACCUMULO_CONF_DIR/slaves`) file to account for the addition.

```
$ACCUMULO_HOME/bin/accumulo admin start <host(s)> {<host> ...}
```

Alternatively, you can ssh to each of the hosts you want to add and run:

```
$ACCUMULO\_HOME/bin/start-here.sh
```

Make sure the host in question has the new configuration, or else the tablet server won't start; at a minimum this needs to be on the host(s) being added, but in practice it's good to ensure consistent configuration across all nodes.

11.7.4 Decomissioning a Node

If you need to take a node out of operation, you can trigger a graceful shutdown of a tablet server. Accumulo will automatically rebalance the tablets across the available tablet servers.

```
$ACCUMULO_HOME/bin/accumulo admin stop <host(s)> {<host> ...}
```

Alternatively, you can ssh to each of the hosts you want to remove and run:

```
$ACCUMULO\_HOME/bin/stop-here.sh
```

Be sure to update your `$ACCUMUL_HOME/conf/slaves` (or `$ACCUMULO_CONF_DIR/slaves`) file to account for the removal of these hosts. Bear in mind that the monitor will not re-read the slaves file automatically, so it will report the decomissioned servers as down; it's recommended that you restart the monitor so that the node list is up to date.

11.8 Monitoring

The Accumulo Master provides an interface for monitoring the status and health of Accumulo components. This interface can be accessed by pointing a web browser to
`http://accumulomaster:50095/status`

11.9 Tracing

It can be difficult to determine why some operations are taking longer than expected. For example, you may be looking up items with very low latency, but sometimes the lookups take much longer. Determining the cause of the delay is difficult because the system is distributed, and the typical lookup is fast.

Accumulo has been instrumented to record the time that various operations take when tracing is turned on. The fact that tracing is enabled follows all the requests made on behalf of the user throughout the distributed infrastructure of accumulo, and across all threads of execution.

These time spans will be inserted into the `trace` table in Accumulo. You can browse recent traces from the Accumulo monitor page. You can also read the `trace` table directly like any other table.

The design of Accumulo's distributed tracing follows that of Google's Dapper.

11.9.1 Tracers

To collect traces, Accumulo needs at least one server listed in
`$ACCUMULO_HOME/conf/tracers`. The server collects traces from clients and writes them to the `trace` table. The Accumulo user that the tracer connects to Accumulo with can be configured with the following properties

```
trace.user
trace.token.property.password
```

11.9.2 Instrumenting a Client

Tracing can be used to measure a client operation, such as a scan, as the operation traverses the distributed system. To enable tracing for your application call

```
DistributedTrace.enable(instance, new ZooReader(instance), hostname, "myApplication");
```

Once tracing has been enabled, a client can wrap an operation in a trace.

```
Trace.on("Client Scan");
BatchScanner scanner = conn.createBatchScanner(...);
// Configure your scanner
for (Entry entry : scanner) {
}
Trace.off();
```

Additionally, the user can create additional Spans within a Trace.

```
Trace.on("Client Update");
...
Span readSpan = Trace.start("Read");
...
readSpan.stop();
...
Span writeSpan = Trace.start("Write");
...
writeSpan.stop();
Trace.off();
```

Like Dapper, Accumulo tracing supports user defined annotations to associate additional data with a Trace.

```
...
int numberOfEntriesRead = 0;
Span readSpan = Trace.start("Read");
// Do the read, update the counter
...
readSpan.data("Number of Entries Read", String.valueOf(numberOfEntriesRead));
```

Some client operations may have a high volume within your application. As such, you may wish to only sample a percentage of operations for tracing. As seen below, the CountSampler can be used to help enable tracing for 1-in-1000 operations

```
Sampler sampler = new CountSampler(1000);
...
if (sampler.next()) {
  Trace.on("Read");
}
...
Trace.offNoFlush();
```

It should be noted that it is safe to turn off tracing even if it isn't currently active. The Trace.offNoFlush() should be used if the user does not wish to have Trace.off() block while flushing trace data.

11.9.3 Viewing Collected Traces

To view collected traces, use the "Recent Traces" link on the Monitor UI. You can also programmatically access and print traces using the `TraceDump` class.

11.9.4 Tracing from the Shell

You can enable tracing for operations run from the shell by using the `trace on` and `trace off` commands.

```
root@test test> trace on
root@test test> scan
a b:c []    d
root@test test> trace off
Waiting for trace information
Waiting for trace information
Trace started at 2013/08/26 13:24:08.332
Time  Start  Service@Location       Name
 3628+0      shell@localhost shell:root
    8+1690     shell@localhost scan
    7+1691       shell@localhost scan:location
    6+1692         tserver@localhost startScan
    5+1692           tserver@localhost tablet read ahead 6
```

11.10 Logging

Accumulo processes each write to a set of log files. By default these are found under `$ACCUMULO/logs/`.

11.11 Recovery

In the event of TabletServer failure or error on shutting Accumulo down, some mutations may not have been minor compacted to HDFS properly. In this case, Accumulo will automatically reapply such mutations from the write-ahead log either when the tablets from the failed server are reassigned by the Master (in the case of a single TabletServer failure) or the next time Accumulo starts (in the event of failure during shutdown).

Recovery is performed by asking a tablet server to sort the logs so that tablets can easily find their missing updates. The sort status of each file is displayed on Accumulo monitor status page. Once the recovery is complete any tablets involved should return to an “online" state. Until then those tablets will be unavailable to clients.

The Accumulo client library is configured to retry failed mutations and in many cases clients will be able to continue processing after the recovery process without throwing an exception.

Chapter 12

Multi-Volume Installations

This is an advanced configuration setting for very large clusters under a lot of write pressure.

The HDFS NameNode holds all of the metadata about the files in HDFS. For fast performance, all of this information needs to be stored in memory. A single NameNode with 64G of memory can store the metadata for tens of millions of files.However, when scaling beyond a thousand nodes, an active Accumulo system can generate lots of updates to the file system, especially when data is being ingested. The large number of write transactions to the NameNode, and the speed of a single edit log, can become the limiting factor for large scale Accumulo installations.

You can see the effect of slow write transactions when the Accumulo Garbage Collector takes a long time (more than 5 minutes) to delete the files Accumulo no longer needs. If your Garbage Collector routinely runs in less than a minute, the NameNode is performing well.

However, if you do begin to experience slow-down and poor GC performance, Accumulo can be configured to use multiple NameNode servers. The configuration "instance.volumes" should be set to a comma-separated list, using full URI references to different NameNode servers:

```
<property>
  <name>instance.volumes</name>
  <value>hdfs://ns1:9001,hdfs://ns2:9001</value>
</property>
```

The introduction of multiple volume support in 1.6 changed the way Accumulo stores pointers to files. It now stores fully qualified URI references to files. Before 1.6, Accumulo stored paths that were relative to a table directory. After an upgrade these relative paths will still exist and are resolved using instance.dfs.dir, instance.dfs.uri, and Hadoop configuration in the same way they were before 1.6.

If the URI for a namenode changes (e.g. namenode was running on host1 and its moved to host2), then Accumulo will no longer function. Even if Hadoop and Accumulo configurations are changed, the fully qualified URIs stored in Accumulo will still contain the old URI. To handle this Accumulo has the following configuration property for replacing URI stored in its metadata. The example configuration below will replace ns1 with nsA and ns2 with nsB in Accumulo metadata. For this property to take affect, Accumulo will need to be restarted.

```
<property>
  <name>instance.volumes.replacements</name>
  <value>hdfs://ns1:9001 hdfs://nsA:9001, hdfs://ns2:9001 hdfs://nsB:9001</value>
</property>
```

Using viewfs or HA namenode, introduced in Hadoop 2, offers another option for managing the fully qualified URIs stored in Accumulo. Viewfs and HA namenode both introduce a level of indirection in the Hadoop configuration. For example assume viewfs:///nn1 maps to hdfs://nn1 in the Hadoop configuration. If viewfs://nn1 is used by Accumulo, then its easy to map viewfs://nn1 to hdfs://nnA by changing the Hadoop configuration w/o doing anything to Accumulo. A production system should probably use a HA namenode. Viewfs may be useful on a test system with a single non HA namenode.

You may also want to configure your cluster to use Federation, available in Hadoop 2.0, which allows DataNodes to respond to multiple NameNode servers, so you do not have to partition your DataNodes by NameNode.

Chapter 13

Troubleshooting

13.1 Logs

Q. The tablet server does not seem to be running!? What happened?

Accumulo is a distributed system. It is supposed to run on remote equipment, across hundreds of computers. Each program that runs on these remote computers writes down events as they occur, into a local file. By default, this is defined in `$ACCUMULO_HOME`/conf/accumule-env.sh as ACCUMULO_LOG_DIR.

A. Look in the `$ACCUMULO_LOG_DIR`/tserver*.log file. Specifically, check the end of the file.

Q. The tablet server did not start and the debug log does not exists! What happened?

When the individual programs are started, the stdout and stderr output of these programs are stored in ".out" and ".err" files in `$ACCUMULO_LOG_DIR`. Often, when there are missing configuration options, files or permissions, messages will be left in these files.

A. Probably a start-up problem. Look in `$ACCUMULO_LOG_DIR`/tserver*.err

13.2 Monitor

Q. Accumulo is not working, what's wrong?

There's a small web server that collects information about all the components that make up a running Accumulo instance. It will highlight unusual or unexpected conditions.

A. Point your browser to the monitor (typically the master host, on port 50095). Is anything red or yellow?

Q. My browser is reporting connection refused, and I cannot get to the monitor

The monitor program's output is also written to .err and .out files in the `$ACCUMULO_LOG_DIR`. Look for problems in this file if the `$ACCUMULO_LOG_DIR/monitor*.log` file does not exist.

A. The monitor program is probably not running. Check the log files for errors.

Q. My browser hangs trying to talk to the monitor.

Your browser needs to be able to reach the monitor program. Often large clusters are firewalled, or use a VPN for internal communications. You can use SSH to proxy your browser to the cluster, or consult with your system administrator to gain access to the server from your browser.

It is sometimes helpful to use a text-only browser to sanity-check the monitor while on the machine running the monitor:

```
$ links http://localhost:50095
```

A. Verify that you are not firewalled from the monitor if it is running on a remote host.

Q. The monitor responds, but there are no numbers for tservers and tables. The summary page says the master is down.

The monitor program gathers all the details about the master and the tablet servers through the master. It will be mostly blank if the master is down.

A. Check for a running master.

13.3 HDFS

Accumulo reads and writes to the Hadoop Distributed File System. Accumulo needs this file system available at all times for normal operations.

Q. Accumulo is having problems "getting a block blk_1234567890123." How do I fix it?

This troubleshooting guide does not cover HDFS, but in general, you want to make sure that all the datanodes are running and an fsck check finds the file system clean:

```
$ hadoop fsck /accumulo
```

You can use:

```
$ hadoop fsck /accumulo/path/to/corrupt/file -locations -blocks -files
```

to locate the block references of individual corrupt files and use those references to search the name node and individual data node logs to determine which servers those blocks have been assigned and then try to fix any underlying file system issues on those nodes.

On a larger cluster, you may need to increase the number of Xceivers

```
  <property>
    <name>dfs.datanode.max.xcievers</name>
    <value>4096</value>
  </property>
```

A. Verify HDFS is healthy, check the datanode logs.

13.4 Zookeeper

Q. `accumulo init` is hanging. It says something about talking to zookeeper.

Zookeeper is also a distributed service. You will need to ensure that it is up. You can run the zookeeper command line tool to connect to any one of the zookeeper servers:

```
  $ zkCli.sh -server zoohost
...
[zk: zoohost:2181(CONNECTED) 0]
```

It is important to see the word `CONNECTED`! If you only see `CONNECTING` you will need to diagnose zookeeper errors.

A. Check to make sure that zookeeper is up, and that `$ACCUMULO_HOME/conf/accumulo-site.xml` has been pointed to your zookeeper server(s).

Q. Zookeeper is running, but it does not say `CONNECTED`

Zookeeper processes talk to each other to elect a leader. All updates go through the leader and propagate to a majority of all the other nodes. If a majority of the nodes cannot be reached, zookeeper will not allow updates. Zookeeper also limits the number connections to a server from any other single host. By default, this limit can be as small as 10 and can be reached in some everything-on-one-machine test configurations.

You can check the election status and connection status of clients by asking the zookeeper nodes for their status. You connect to zookeeper and ask it with the four-letter "stat" command:

```
$ nc zoohost 2181
stat
Zookeeper version: 3.4.5-1392090, built on 09/30/2012 17:52 GMT
Clients:
 /127.0.0.1:58289[0](queued=0,recved=1,sent=0)
 /127.0.0.1:60231[1](queued=0,recved=53910,sent=53915)

Latency min/avg/max: 0/5/3008
Received: 1561459
Sent: 1561592
```

```
Connections: 2
Outstanding: 0
Zxid: 0x621a3b
Mode: standalone
Node count: 22524
$
```

A. Check zookeeper status, verify that it has a quorum, and has not exceeded maxClientCnxns.

Q. My tablet server crashed! The logs say that it lost it's zookeeper lock.

Tablet servers reserve a lock in zookeeper to maintain their ownership over the tablets that have been assigned to them. Part of their responsibility for keeping the lock is to send zookeeper a keep-alive message periodically. If the tablet server fails to send a message in a timely fashion, zookeeper will remove the lock and notify the tablet server. If the tablet server does not receive a message from zookeeper, it will assume its lock has been lost, too. If a tablet server loses its lock, it kills itself: everything assumes it is dead already.

A. Investigate why the tablet server did not send a timely message to zookeeper.

13.4.1 Keeping the tablet server lock

Q. My tablet server lost its lock. Why?

The primary reason a tablet server loses its lock is that it has been pushed into swap.

A large java program (like the tablet server) may have a large portion of its memory image unused. The operation system will favor pushing this allocated, but unused memory into swap so that the memory can be re-used as a disk buffer. When the java virtual machine decides to access this memory, the OS will begin flushing disk buffers to return that memory to the VM. This can cause the entire process to block long enough for the zookeeper lock to be lost.

A. Configure your system to reduce the kernel parameter "swappiness" from the default (60) to zero.

Q. My tablet server lost its lock, and I have already set swappiness to zero. Why?

Be careful not to over-subscribe memory. This can be easy to do if your accumulo processes run on the same nodes as hadoop's map-reduce framework. Remember to add up:

- size of the JVM for the tablet server
- size of the in-memory map, if using the native map implementation
- size of the JVM for the data node
- size of the JVM for the task tracker

- size of the JVM times the maximum number of mappers and reducers
- size of the kernel and any support processes

If a 16G node can run 2 mappers and 2 reducers, and each can be 2G, then there is only 8G for the data node, tserver, task tracker and OS.

A. Reduce the memory footprint of each component until it fits comfortably.

Q. My tablet server lost its lock, swappiness is zero, and my node has lots of unused memory!

The JVM memory garbage collector may fall behind and cause a "stop-the-world" garbage collection. On a large memory virtual machine, this collection can take a long time. This happens more frequently when the JVM is getting low on free memory. Check the logs of the tablet server. You will see lines like this:

```
2013-06-20 13:43:20,607 [tabletserver.TabletServer] DEBUG: gc ParNew=0.00(+0.00) secs
    ConcurrentMarkSweep=0.00(+0.00) secs freemem=1,868,325,952(+1,868,325,952) totalmem=2,040,135,680
```

When "freemem" becomes small relative to the amount of memory needed, the JVM will spend more time finding free memory than performing work. This can cause long delays in sending keep-alive messages to zookeeper.

A. Ensure the tablet server JVM is not running low on memory.

13.5 Tools

The accumulo script can be used to run classes from the command line. This section shows how a few of the utilities work, but there are many more.

There's a class that will examine an accumulo storage file and print out basic metadata.

```
$ ./bin/accumulo org.apache.accumulo.core.file.rfile.PrintInfo /accumulo/tables/1/default_tablet/A000000n.rf
2013-07-16 08:17:14,778 [util.NativeCodeLoader] INFO : Loaded the native-hadoop library
Locality group         : <DEFAULT>
        Start block          : 0
        Num   blocks         : 1
        Index level 0        : 62 bytes  1 blocks
        First key            : 288be9ab4052fe9e span:34078a86a723e5d3:3da450f02108ced5 [] 1373373521623 false
        Last key             : start:13fc375709e id:615f5ee2dd822d7a [] 1373373821660 false
        Num entries          : 466
        Column families      : [waitForCommits, start, md major compactor 1, md major compactor 2, md major compactor 3,
                                bringOnline, prep, md major compactor 4, md major compactor 5, md root major compactor 3,
                                minorCompaction, wal, compactFiles, md root major compactor 4, md root major compactor 1,
                                md root major compactor 2, compact, id, client:update, span, update, commit, write,
                                majorCompaction]

Meta block     : BCFile.index
      Raw size             : 4 bytes
      Compressed size      : 12 bytes
      Compression type     : gz

Meta block     : RFile.index
```

```
      Raw size            : 780 bytes
      Compressed size     : 344 bytes
      Compression type    : gz
```

When trying to diagnose problems related to key size, the PrintInfo tool can provide a histogram of the individual key sizes:

```
$ ./bin/accumulo org.apache.accumulo.core.file.rfile.PrintInfo --histogram /accumulo/tables/1/default_tablet/A000000n.rf
...
Up to size      count      %-age
         10 :        222  28.23%
        100 :        244  71.77%
       1000 :          0   0.00%
      10000 :          0   0.00%
     100000 :          0   0.00%
    1000000 :          0   0.00%
   10000000 :          0   0.00%
  100000000 :          0   0.00%
 1000000000 :          0   0.00%
10000000000 :          0   0.00%
```

Likewise, PrintInfo will dump the key-value pairs and show you the contents of the RFile:

```
$ ./bin/accumulo org.apache.accumulo.core.file.rfile.PrintInfo --dump /accumulo/tables/1/default_tablet/A000000n.rf
row columnFamily:columnQualifier [visibility] timestamp deleteFlag -> Value
...
```

Q. Accumulo is not showing me any data!

A. Do you have your auths set so that it matches your visibilities?

Q. What are my visibilities?

A. Use “PrintInfo” on a representative file to get some idea of the visibilities in the underlying data.

Note that the use of PrintInfo is an administrative tool and can only by used by someone who can access the underlying Accumulo data. It does not provide the normal access controls in Accumulo.

If you would like to backup, or otherwise examine the contents of Zookeeper, there are commands to dump and load to/from XML.

```
$ ./bin/accumulo org.apache.accumulo.server.util.DumpZookeeper --root /accumulo >dump.xml
$ ./bin/accumulo org.apache.accumulo.server.util.RestoreZookeeper --overwrite < dump.xml
```

Q. How can I get the information in the monitor page for my cluster monitoring system?

A. Use GetMasterStats:

```
$ ./bin/accumulo org.apache.accumulo.test.GetMasterStats | grep Load
 OS Load Average: 0.27
```

Q. The monitor page is showing an offline tablet. How can I find out which tablet it is?

A. Use FindOfflineTablets:

```
$ ./bin/accumulo org.apache.accumulo.server.util.FindOfflineTablets
2<<@(null,null,localhost:9997) is UNASSIGNED  #walogs:2
```

Here's what the output means:

1. `2«` This is the tablet from (-inf, +inf) for the table with id 2. "tables -l" in the shell will show table ids for tables.
2. @(null, null, localhost:9997) Location information. The format is `@(assigned, hosted, last)`. In this case, the tablet has not been assigned, is not hosted anywhere, and was once hosted on localhost.
3. #walogs:2 The number of write-ahead logs that this tablet requires for recovery.

An unassigned tablet with write-ahead logs is probably waiting for logs to be sorted for efficient recovery.

Q. How can I be sure that the metadata tables are up and consistent?

A. `CheckForMetadataProblems` will verify the start/end of every tablet matches, and the start and stop for the table is empty:

```
$ ./bin/accumulo org.apache.accumulo.server.util.CheckForMetadataProblems -u root --password
Enter the connection password:
All is well for table !0
All is well for table 1
```

Q. My hadoop cluster has lost a file due to a NameNode failure. How can I remove the file?

A. There's a utility that will check every file reference and ensure that the file exists in HDFS. Optionally, it will remove the reference:

```
$ ./bin/accumulo org.apache.accumulo.server.util.RemoveEntriesForMissingFiles -u root --password
Enter the connection password:
2013-07-16 13:10:57,293 [util.RemoveEntriesForMissingFiles] INFO : File /accumulo/tables/2/default_tablet/F0000005.rf
 is missing
2013-07-16 13:10:57,296 [util.RemoveEntriesForMissingFiles] INFO : 1 files of 3 missing
```

Q. I have many entries in zookeeper for old instances I no longer need. How can I remove them?

A. Use CleanZookeeper:

```
$ ./bin/accumulo org.apache.accumulo.server.util.CleanZookeeper
```

This command will not delete the instance pointed to by the local `conf/accumulo-site.xml` file.

Q. I need to decommission a node. How do I stop the tablet server on it?

A. Use the admin command:

```
$ ./bin/accumulo admin stop hostname:9997
2013-07-16 13:15:38,403 [util.Admin] INFO : Stopping server 12.34.56.78:9997
```

Q. I cannot login to a tablet server host, and the tablet server will not shut down. How can I kill the server?

A. Sometimes you can kill a "stuck" tablet server by deleting it's lock in zookeeper:

```
$ ./bin/accumulo org.apache.accumulo.server.util.TabletServerLocks --list
                  127.0.0.1:9997 TSERV_CLIENT=127.0.0.1:9997
$ ./bin/accumulo org.apache.accumulo.server.util.TabletServerLocks -delete 127.0.0.1:9997
$ ./bin/accumulo org.apache.accumulo.server.util.TabletServerLocks -list
                  127.0.0.1:9997             null
```

You can find the master and instance id for any accumulo instances using the same zookeeper instance:

```
$ ./bin/accumulo org.apache.accumulo.server.util.ListInstances
INFO : Using ZooKeepers localhost:2181

 Instance Name       | Instance ID                          | Master
---------------------+--------------------------------------+-------------------------------
              "test" | 6140b72e-edd8-4126-b2f5-e74a8bbe323b |                127.0.0.1:9999
```

13.6 System Metadata Tables

Accumulo tracks information about tables in metadata tables. The metadata for most tables is contained within the metadata table in the accumulo namespace, while metadata for that table is contained in the root table in the accumulo namespace. The root table is composed of a single tablet, which does not split, so it is also called the root tablet. Information about the root table, such as its location and write-ahead logs, are stored in ZooKeeper.

Let's create a table and put some data into it:

```
shell> createtable test
shell> tables -l
accumulo.metadata    =>        !0
accumulo.root        =>        +r
test                 =>         2
trace                =>         1
shell> insert a b c d
shell> flush -w
```

Now let's take a look at the metadata for this table:

```
shell> table accumulo.metadata
shell> scan -b 3; -e 3<
3< file:/default_tablet/F000009y.rf []    186,1
3< last:13fe86cd27101e5 []    127.0.0.1:9997
3< loc:13fe86cd27101e5 []    127.0.0.1:9997
3< log:127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995 []    127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995|6
3< srv:dir []    /default_tablet
3< srv:flush []    1
3< srv:lock []    tservers/127.0.0.1:9997/zlock-0000000001$13fe86cd27101e5
3< srv:time []    M1373998392323
3< ~tab:~pr []    \x00
```

Let's decode this little session:

1. `scan -b 3; -e 3<`
   Every tablet gets its own row. Every row starts with the table id followed by ";" or "<", and followed by the end row split point for that tablet.
2. `file:/default_tablet/F000009y.rf [] 186,1`
   File entry for this tablet. This tablet contains a single file reference. The file is "/accumulo/tables/3/default_tablet/F000009y.rf". It contains 1 key/value pair, and is 186 bytes long.
3. `last:13fe86cd27101e5 [] 127.0.0.1:9997`
   Last location for this tablet. It was last held on 127.0.0.1:9997, and the unique tablet server lock data was "13fe86cd27101e5". The default balancer will tend to put tablets back on their last location.
4. `loc:13fe86cd27101e5 [] 127.0.0.1:9997`
   The current location of this tablet.
5. `log:127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995 [] 127.0.  ...`
   This tablet has a reference to a single write-ahead log. This file can be found in /accumulo/wal/127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995. The value of this entry could refer to multiple files. This tablet's data is encoded as "6" within the log.
6. `srv:dir [] /default_tablet`
   Files written for this tablet will be placed into /accumulo/tables/3/default_tablet.
7. `srv:flush [] 1`
   Flush id. This table has successfully completed the flush with the id of "1".
8. `srv:lock [] tservers/127.0.0.1:9997/zlock-0000000001$13fe86cd27101e5`
   This is the lock information for the tablet holding the present lock. This information is checked against zookeeper whenever this is updated, which prevents a metadata update from a tablet server that no longer holds its lock.
9. `srv:time [] M1373998392323`
10. `~tab:~pr [] \x00`
    The end-row marker for the previous tablet (prev-row). The first byte indicates the presence of a prev-row. This tablet has the range (-inf, +inf), so it has no prev-row (or end row).

Besides these columns, you may see:

1. `rowId future:zooKeeperID location` Tablet has been assigned to a tablet, but not yet loaded.

2. `~del:filename` When a tablet server is done use a file, it will create a delete marker in the appropriate metadata table, unassociated with any tablet. The garbage collector will remove the marker, and the file, when no other reference to the file exists.
3. `~blip:txid` Bulk-Load In Progress marker
4. `rowId loaded:filename` A file has been bulk-loaded into this tablet, however the bulk load has not yet completed on other tablets, so this is marker prevents the file from being loaded multiple times.
5. `rowId !cloned` A marker that indicates that this tablet has been successfully cloned.
6. `rowId splitRatio:ratio` A marker that indicates a split is in progress, and the files are being split at the given ratio.
7. `rowId chopped` A marker that indicates that the files in the tablet do not contain keys outside the range of the tablet.
8. `rowId scan` A marker that prevents a file from being removed while there are still active scans using it.

13.7 Simple System Recovery

Q. One of my Accumulo processes died. How do I bring it back?

The easiest way to bring all services online for an Accumulo instance is to run the "start-all.sh" script.

```
$ bin/start-all.sh
```

This process will check the process listing, using "jps" on each host before attempting to restart a service on the given host. Typically, this check is sufficient except in the face of a hung/zombie process. For large clusters, it may be undesirable to ssh to every node in the cluster to ensure that all hosts are running the appropriate processes and "start-here.sh" may be of use.

```
$ ssh host_with_dead_process
$ bin/start-here.sh
```

"start-here.sh" should be invoked on the host which is missing a given process. Like start-all.sh, it will start all necessary processes that are not currently running, but only on the current host and not cluster-wide. Tools such as "pssh" or "pdsh" can be used to automate this process.

"start-server.sh" can also be used to start a process on a given host; however, it is not generally recommended for users to issue this directly as the "start-all.sh" and "start-here.sh" scripts provide the same functionality with more automation and are less prone to user error.

A. Use “start-all.sh“ or “start-here.sh“.

Q. My process died again. Should I restart it via “cron“ or tools like “supervisord“?

A. A repeatedly dying Accumulo process is a sign of a larger problem. Typically these problems are due to a misconfiguration of Accumulo or over-saturation of resources. Blind automation of any service restart inside of Accumulo is generally an undesirable situation as it is indicative of a problem that is being masked and ignored. Accumulo processes should be stable on the order of months and not require frequent restart.

13.8 Advanced System Recovery

13.8.1 HDFS Failure

Q. I had disasterous HDFS failure. After bringing everything back up, several tablets refuse to go online.

Data written to tablets is written into memory before being written into indexed files. In case the server is lost before the data is saved into a an indexed file, all data stored in memory is first written into a write-ahead log (WAL). When a tablet is re-assigned to a new tablet server, the write-ahead logs are read to recover any mutations that were in memory when the tablet was last hosted.

If a write-ahead log cannot be read, then the tablet is not re-assigned. All it takes is for one of the blocks in the write-ahead log to be missing. This is unlikely unless multiple data nodes in HDFS have been lost.

A. Get the WAL files online and healthy. Restore any data nodes that may be down.

Q. How do find out which tablets are offline?

A. Use “accumulo admin checkTablets”

```
  $ bin/accumulo admin checkTablets
```

Q. I lost three data nodes, and I’m missing blocks in a WAL. I don’t care about data loss, how can I get those tablets online?

See the discussion in section 13.6, which shows a typical metadata table listing. The entries with a column family of “log” are references to the WAL for that tablet. If you know what WAL is bad, you can find all the references with a grep in the shell:

```
shell> grep 0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995
3< log:127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995 []    127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995|6
```

A. You can remove the WAL references in the metadata table.

```
shell> grant -u root Table.WRITE -t accumulo.metadata
shell> delete 3< log 127.0.0.1+9997/0cb7ce52-ac46-4bf7-ae1d-acdcfaa97995
```

Note: the colon (":") is omitted when specifying the "row cf cq" for the delete command.

The master will automatically discover the tablet no longer has a bad WAL reference and will assign the tablet. You will need to remove the reference from all the tablets to get them online.

Q. The metadata (or root) table has references to a corrupt WAL.

This is a much more serious state, since losing updates to the metadata table will result in references to old files which may not exist, or lost references to new files, resulting in tablets that cannot be read, or large amounts of data loss.

The best hope is to restore the WAL by fixing HDFS data nodes and bringing the data back online. If this is not possible, the best approach is to re-create the instance and bulk import all files from the old instance into a new tables.

A complete set of instructions for doing this is outside the scope of this guide, but the basic approach is:

- Use "tables -l" in the shell to discover the table name to table id mapping
- Stop all accumulo processes on all nodes
- Move the accumulo directory in HDFS out of the way:

  ```
  $ hadoop fs -mv /accumulo /corrupt
  ```
- Re-initalize accumulo
- Recreate tables, users and permissions
- Import the directories under `/corrupt/tables/<id>` into the new instance

Q. One or more HDFS Files under /accumulo/tables are corrupt

Accumulo maintains multiple references into the tablet files in the METADATA table and within the tablet server hosting the file, this makes it difficult to reliably just remove those references.

The directory structure in HDFS for tables will follow the general structure:

```
/accumulo
/accumulo/tables/
/accumulo/tables/!0
/accumulo/tables/!0/default_tablet/A000001.rf
/accumulo/tables/!0/t-00001/A000002.rf
/accumulo/tables/1
/accumulo/tables/1/default_tablet/A000003.rf
```

```
/accumulo/tables/1/t-00001/A000004.rf
/accumulo/tables/1/t-00001/A000005.rf
/accumulo/tables/2/default_tablet/A000006.rf
/accumulo/tables/2/t-00001/A000007.rf
```

If files under /accumulo/tables are corrupt, the best course of action is to recover those files in hdsf see the section on HDFS. Once these recovery efforts have been exhausted, the next step depends on where the missing file(s) are located. Different actions are required when the bad files are in Accumulo data table files or if they are metadata table files.

Data File Corruption

When an Accumulo data file is corrupt, the most reliable way to restore Accumulo operations is to replace the missing file with an âĂIJemptyâĂİ file so that references to the file in the METADATA table and within the tablet server hosting the file can be resolved by Accumulo. An empty file can be created using the CreateEmpty utiity:

```
$accumulo org.apache.accumulo.core.file.rfile.CreateEmpty /path/to/empty/file/empty.rf
```

The process is to delete the corrupt file and then move the empty file into its place (The generated empty file can be copied and used multiple times if necessary and does not need to be regenerated each time)

```
$hadoop fs âĂŞrm /accumulo/tables/corrupt/file/thename.rf; \
hadoop fs -mv /path/to/empty/file/empty.rf /accumulo/tables/corrupt/file/thename.rf
```

Metadata File Corruption

If the corrupt files are metadata files, see 13.6 (under the path

```
/accumulo/tables/!0
```

) then you will need to rebuild the metadata table by initializing a new instance of Accumulo and then importing all of the existing data into the new instance. This is the same procedure as recovering from a zookeeper failure (see **??**, except that you will have the benefit of having the existing user and table authorizations that are maintained in zookeeper.

You can use the DumpZookeeper utility to save this information for reference before creating the new instance. You will not be able to use RestoreZookeeper because the table names and references are likely to be different between the original and the new instances, but it can serve as a reference.

A. If the files cannot be recovered, replace corrupt data files with a empty rfiles to allow references in the metadata table and in the tablet servers to be resolved. Rebuild the metadata table if the corrupt files are metadata files.

13.8.2 ZooKeeper Failure

Q. I lost my ZooKeeper quorum (hardware failure), but HDFS is still intact. How can I recover my Accumulo instance?

ZooKeeper, in addition to its lock-service capabilities, also serves to bootstrap an Accumulo instance from some location in HDFS. It contains the pointers to the root tablet in HDFS which is then used to load the Accumulo metadata tablets, which then loads all user tables. ZooKeeper also stores all namespace and table configuration, the user database, the mapping of table IDs to table names, and more across Accumulo restarts.

Presently, the only way to recover such an instance is to initialize a new instance and import all of the old data into the new instance. The easiest way to tackle this problem is to first recreate the mapping of table ID to table name and then recreate each of those tables in the new instance. Set any necessary configuration on the new tables and add some split points to the tables to close the gap between how many splits the old table had and no splits.

The directory structure in HDFS for tables will follow the general structure:

```
/accumulo
/accumulo/tables/
/accumulo/tables/1
/accumulo/tables/1/default_tablet/A000001.rf
/accumulo/tables/1/t-00001/A000002.rf
/accumulo/tables/1/t-00001/A000003.rf
/accumulo/tables/2/default_tablet/A000004.rf
/accumulo/tables/2/t-00001/A000005.rf
```

For each table, make a new directory that you can move (or copy if you have the HDFS space to do so) all of the rfiles for a given table into. For example, to process the table with an ID of "1", make a new directory, say "/new-table-1" and then copy all files from "/accumulo/tables/1/*/*.rf" into that directory. Additionally, make a directory, "/new-table-1-failures", for any failures during the import process. Then, issue the import command using the Accumulo shell into the new table, telling Accumulo to not re-set the timestamp:

```
user@instance new_table> importdirectory /new-table-1 /new-table-1-failures false
```

Any RFiles which were failed to be loaded will be placed in "/new-table-1-failures". Rfiles that were successfully imported will no longer exist in "/new-table-1". For failures, move them back to the import directory and retry the "importdirectory" command.

It is **extremely** important to note that this approach may introduce stale data back into the tables. For a few reasons, RFiles may exist in the table directory which are candidates for deletion but have not yet been deleted. Additionally, deleted data which was not compacted away, but still exists in write-ahead logs if the original instance was somehow recoverable, will be re-introduced in the new instance. Table splits and merges (which also include the deleteRows API call on TableOperations, are also vulnerable to this problem. This process should **not** be used if these are unacceptable risks. It is possible to try to re-create a view of the “accumulo.metadata“ table to prune out files that are candidates for deletion, but this is a difficult task that also may not be entirely accurate.

Likewise, it is also possible that data loss may occur from write-ahead log (WAL) files which existed on the old table but were not minor-compacted into an RFile. Again, it may be possible to reconstruct the state of these WAL files to replay data not yet in an RFile; however, this is a difficult task and is not implemented in any automated fashion.

A. The “importdirectory“ shell command can be used to import RFiles from the old instance into a newly created instance, but extreme care should go into the decision to do this as it may result in reintroduction of stale data or the omission of new data.

13.9 File Naming Conventions

Q. Why are files named like they are? Why do some start with “C” and others with “F”?

A. The file names give you a basic idea for the source of the file.

The base of the filename is a base-36 unique number. All filenames in accumulo are coordinated with a counter in zookeeper, so they are always unique, which is useful for debugging.

The leading letter gives you an idea of how the file was created:

- F - Flush: entries in memory were written to a file (Minor Compaction)
- M - Merging compaction: entries in memory were combined with the smallest file to create one new file
- C - Several files, but not all files, were combined to produce this file (Major Compaction)
- A - All files were compacted, delete entries were dropped
- I - Bulk import, complete, sorted index files. Always in a directory starting with "b-"

This simple file naming convention allows you to see the basic structure of the files from just their filenames, and reason about what should be happening to them next, just by scanning their entries in the metadata tables.

For example, if you see multiple files with “M” prefixes, the tablet is, or was, up against it’s maximum file limit, so it began merging memory updates with files to keep the file count reasonable. This slows down ingest performance, so knowing there are many files like this tells you that the system is struggling to keep up with ingest vs the compaction strategy which reduces the number of files.

Appendix A

Configuration Management

A.1 Configuration Overview

All accumulo properties have a default value in the source code. Properties can also be set in accumulo-site.xml and in zookeeper on per-table or system-wide basis. If properties are set in more than one location, accumulo will choose the property with the highest precedence. This order of precedence is described below (from highest to lowest):

A.1.1 Zookeeper table properties

Table properties are applied to the entire cluster when set in zookeeper using the accumulo API or shell. While table properties take precedent over system properties, both will override properties set in accumulo-site.xml

Table properties consist of all properties with the table.* prefix. Table properties are configured on a per-table basis using the following shell commmand:

```
config -t TABLE -s PROPERTY=VALUE
```

A.1.2 Zookeeper system properties

System properties are applied to the entire cluster when set in zookeeper using the accumulo API or shell. System properties consist of all properties with a 'yes' in the 'Zookeeper Mutable' column in the table below. They are set with the following shell command:

```
config -s PROPERTY=VALUE
```

If a table.* property is set using this method, the value will apply to all tables except those configured on per-table basis (which have higher precedence).

While most system properties take effect immediately, some require a restart of the process which is indicated in ‘Zookeeper Mutable’.

A.1.3 accumulo-site.xml

Accumulo processes (master, tserver, etc) read their local accumulo-site.xml on start up. Therefore, changes made to accumulo-site.xml must rsynced across the cluster and processes must be restarted to apply changes.

Certain properties (indicated by a ‘no’ in ‘Zookeeper Mutable’) cannot be set in zookeeper and only set in this file. The accumulo-site.xml also allows you to configure tablet servers with different settings.

A.1.4 Default Values

All properties have a default value in the source code. This value has the lowest precedence and is overriden if set in accumulo-site.xml or zookeeper.

While the default value is usually optimal, there are cases where a change can increase query and ingest performance.

A.2 Configuration in the Shell

The ‘config’ command in the shell allows you to view the current system configuration. You can also use the ‘-t’ option to view a table’s configuration as below:

```
$ ./bin/accumulo shell -u root
Enter current password for 'root'@'ac14': ******

Shell - Apache Accumulo Interactive Shell
-
- version: 1.5.0
- instance name: ac14
- instance id: 4f48fa03-f692-43ce-ae03-94c9ea8b7181
-
- type 'help' for a list of available commands
-
root@ac13> config -t foo
---------+---------------------------------------------+------------------------------------------------------
SCOPE    | NAME                                        | VALUE
---------+---------------------------------------------+------------------------------------------------------
default  | table.balancer ............................ | org.apache.accumulo.server.master.balancer.DefaultLoadBalancer
default  | table.bloom.enabled ...................... | false
default  | table.bloom.error.rate ................... | 0.5%
default  | table.bloom.hash.type .................... | murmur
default  | table.bloom.key.functor .................. | org.apache.accumulo.core.file.keyfunctor.RowFunctor
default  | table.bloom.load.threshold ............... | 1
default  | table.bloom.size ......................... | 1048576
default  | table.cache.block.enable ................. | false
default  | table.cache.index.enable ................. | false
default  | table.compaction.major.everything.at ..... | 19700101000000GMT
default  | table.compaction.major.everything.idle ... | 1h
default  | table.compaction.major.ratio ............. | 1.3
site     |    @override ............................. | 1.4
system   |    @override ............................. | 1.5
table    |    @override ............................. | 1.6
default  | table.compaction.minor.idle .............. | 5m
default  | table.compaction.minor.logs.threshold .... | 3
default  | table.failures.ignore .................... | false
```

A.3 Available Properties

A.3.1 rpc.*

Properties in this category related to the configuration of SSL keys for RPC. See also instance.ssl.enabled

rpc.javax.net.ssl.keyStore

Path of the keystore file for the servers' private SSL key

Type: PATH
ZooKeeper Mutable: no
Default Value: $ACCUMULO_CONF_DIR/ssl/keystore.jks

rpc.javax.net.ssl.keyStorePassword

Password used to encrypt the SSL private keystore. Leave blank to use the Accumulo instance secret

Type: STRING
ZooKeeper Mutable: no
Default Value:

rpc.javax.net.ssl.keyStoreType

Type of SSL keystore

Type: STRING
ZooKeeper Mutable: no
Default Value: jks

rpc.javax.net.ssl.trustStore

Path of the truststore file for the root cert

Type: PATH
ZooKeeper Mutable: no
Default Value: $ACCUMULO_CONF_DIR/ssl/truststore.jks

rpc.javax.net.ssl.trustStorePassword

Password used to encrypt the SSL truststore. Leave blank to use no password

Type: STRING
ZooKeeper Mutable: no
Default Value:

rpc.javax.net.ssl.trustStoreType

Type of SSL truststore

Type: STRING
ZooKeeper Mutable: no
Default Value: jks

rpc.useJsse

Use JSSE system properties to configure SSL rather than the rpc.javax.net.ssl.* Accumulo properties

Type: BOOLEAN
ZooKeeper Mutable: no
Default Value: false

A.3.2 instance.*

Properties in this category must be consistent throughout a cloud. This is enforced and servers won't be able to communicate if these differ.

instance.dfs.dir

~~*Deprecated.* HDFS directory in which accumulo instance will run. Do not change after accumulo is initialized.~~

~~*Type:* ABSOLUTEPATH~~
~~*ZooKeeper Mutable:* no~~
~~*Default Value:* /accumulo~~

instance.dfs.uri

~~*Deprecated.* A url accumulo should use to connect to DFS. If this is empty, accumulo will obtain this information from the hadoop configuration. This property will only be used when creating new files if instance.volumes is empty. After an upgrade to 1.6.0 Accumulo will start using absolute paths to reference files. Files created before a 1.6.0 upgrade are referenced via relative paths. Relative paths will always be resolved using this config (if empty using the hadoop config).~~

~~*Type:* URI~~
~~*ZooKeeper Mutable:* no~~
~~*Default Value:*~~

instance.rpc.ssl.clientAuth

Require clients to present certs signed by a trusted root

Type: BOOLEAN
ZooKeeper Mutable: no
Default Value: false

instance.rpc.ssl.enabled

Use SSL for socket connections from clients and among accumulo services

Type: BOOLEAN
ZooKeeper Mutable: no
Default Value: false

instance.secret

A secret unique to a given instance that all servers must know in order to communicate with one another. Change it before initialization. To change it later use ./bin/accumulo accumulo.server.util.ChangeSecret [oldpasswd] [newpasswd], and then update conf/accumulo-site.xml everywhere.

Type: STRING
ZooKeeper Mutable: no
Default Value: DEFAULT

instance.security.authenticator

The authenticator class that accumulo will use to determine if a user has privilege to perform an action

Type: CLASSNAME
ZooKeeper Mutable: no
Default Value: org.apache.accumulo.server.security.handler.ZKAuthenticator

instance.security.authorizor

The authorizor class that accumulo will use to determine what labels a user has privilege to see

Type: CLASSNAME
ZooKeeper Mutable: no
Default Value: org.apache.accumulo.server.security.handler.ZKAuthorizor

instance.security.permissionHandler

The permission handler class that accumulo will use to determine if a user has privilege to perform an action

Type: CLASSNAME
ZooKeeper Mutable: no
Default Value: org.apache.accumulo.server.security.handler.ZKPermHandler

instance.volumes

A comma seperated list of dfs uris to use. Files will be stored across these filesystems. If this is empty, then instance.dfs.uri will be used. After adding uris to this list, run 'accumulo init –add-volume' and then restart tservers. If entries are removed from this list then tservers will need to be restarted. After a uri is removed from the list Accumulo will not create new files in that location, however Accumulo can still reference files created at that location before the config change. To use a comma or other reserved characters in a URI use standard URI hex encoding. For example replace commas with %2C.

Type: STRING
ZooKeeper Mutable: no
Default Value:

instance.volumes.replacements

Since accumulo stores absolute URIs changing the location of a namenode could prevent Accumulo from starting. The property helps deal with that situation. Provide a comma seperated list of uri replacement pairs here if a namenode location changes. Each pair shold be separated with a space. For example, if hdfs://nn1 was repalced with hdfs://nnA and hdfs://nn2 was replaced with hdfs://nnB, then set this property to 'hdfs://nn1 hdfs://nnA,hdfs://nn2 hdfs://nnB'Replacements must be configured for use. To see which volumes are currently in use, run 'accumulo admin volumes -l'. To use a comma or other reserved characters in a URI use standard URI hex encoding. For example replace commas with %2C.

Type: STRING
ZooKeeper Mutable: no
Default Value:

instance.zookeeper.host

Comma separated list of zookeeper servers

Type: HOSTLIST
ZooKeeper Mutable: no

Default Value: localhost:2181

instance.zookeeper.timeout

Zookeeper session timeout; max value when represented as milliseconds should be no larger than 2147483647

Type: TIMEDURATION
ZooKeeper Mutable: no
Default Value: 30s

A.3.3 general.*

Properties in this category affect the behavior of accumulo overall, but do not have to be consistent throughout a cloud.

general.classpaths

A list of all of the places to look for a class. Order does matter, as it will look for the jar starting in the first location to the last. Please note, hadoop conf and hadoop lib directories NEED to be here, along with accumulo lib and zookeeper directory. Supports full regex on filename alone.

Type: STRING
ZooKeeper Mutable: no
Default Value: $ACCUMULO_CONF_DIR,
$ACCUMULO_HOME/lib/[^.].*.jar,
$ZOOKEEPER_HOME/zookeeper[^.].*.jar,
$HADOOP_CONF_DIR,
$HADOOP_PREFIX/[^.].*.jar,
$HADOOP_PREFIX/lib/[^.].*.jar,
$HADOOP_PREFIX/share/hadoop/common/.*.jar,
$HADOOP_PREFIX/share/hadoop/common/lib/.*.jar,
$HADOOP_PREFIX/share/hadoop/hdfs/.*.jar,
$HADOOP_PREFIX/share/hadoop/mapreduce/.*.jar,
/usr/lib/hadoop/[^.].*.jar,
/usr/lib/hadoop/lib/[^.].*.jar,
/usr/lib/hadoop-hdfs/[^.].*.jar,
/usr/lib/hadoop-mapreduce/[^.].*.jar,

/usr/lib/hadoop-yarn/[^.].*.jar,

general.dynamic.classpaths

A list of all of the places where changes in jars or classes will force a reload of the classloader.

Type: STRING
ZooKeeper Mutable: no
Default Value: $ACCUMULO_HOME/lib/ext/[^.].*.jar

general.kerberos.keytab

Path to the kerberos keytab to use. Leave blank if not using kerberoized hdfs

Type: PATH
ZooKeeper Mutable: no
Default Value:

general.kerberos.principal

Name of the kerberos principal to use. _HOST will automatically be replaced by the machines hostname in the hostname portion of the principal. Leave blank if not using kerberoized hdfs

Type: STRING
ZooKeeper Mutable: no
Default Value:

general.rpc.timeout

Time to wait on I/O for simple, short RPC calls

Type: TIMEDURATION
ZooKeeper Mutable: no
Default Value: 120s

general.server.message.size.max

The maximum size of a message that can be sent to a server.

Type: MEMORY
ZooKeeper Mutable: no
Default Value: 1G

general.vfs.cache.dir

Directory to use for the vfs cache. The cache will keep a soft reference to all of the classes loaded in the VM. This should be on local disk on each node with sufficient space. It defaults to ${java.io.tmpdir}/accumulo-vfs-cache-${user.name}

Type: ABSOLUTEPATH
ZooKeeper Mutable: no
Default Value: ${java.io.tmpdir}/accumulo-vfs-cache-${user.name}

general.vfs.classpaths

Configuration for a system level vfs classloader. Accumulo jar can be configured here and loaded out of HDFS.

Type: STRING
ZooKeeper Mutable: no
Default Value:

A.3.4 master.*

Properties in this category affect the behavior of the master server

master.bulk.retries

The number of attempts to bulk-load a file before giving up.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 3

master.bulk.threadpool.size

The number of threads to use when coordinating a bulk-import.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 5

master.bulk.timeout

The time to wait for a tablet server to process a bulk import request

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5m

master.fate.threadpool.size

The number of threads used to run FAult-Tolerant Executions. These are primarily table operations like merge.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 4

master.lease.recovery.interval

The amount of time to wait after requesting a WAL file to be recovered

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5s

master.port.client

The port used for handling client connections on the master

Type: PORT
ZooKeeper Mutable: yes but requires restart of the master
Default Value: 9999

master.recovery.delay

When a tablet server's lock is deleted, it takes time for it to completely quit. This delay gives it time before log recoveries begin.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 10s

master.recovery.max.age

Recovery files older than this age will be removed.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 60m

master.recovery.time.max

The maximum time to attempt recovery before giving up

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 30m

master.server.threadcheck.time

The time between adjustments of the server thread pool.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 1s

master.server.threads.minimum

The minimum number of threads to use to handle incoming requests.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 20

master.tablet.balancer

The balancer class that accumulo will use to make tablet assignment and migration decisions.

Type: CLASSNAME
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.server.master.balancer.TableLoadBalancer

master.walog.closer.implementation

A class that implements a mechansim to steal write access to a file

Type: CLASSNAME
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.server.master.recovery.HadoopLogCloser

A.3.5 tserver.*

Properties in this category affect the behavior of the tablet servers

tserver.archive.walogs

Keep copies of the WALOGs for debugging purposes

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

tserver.bloom.load.concurrent.max

The number of concurrent threads that will load bloom filters in the background. Setting this to zero will make bloom filters load in the foreground.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 4

tserver.bulk.assign.threads

The master delegates bulk file processing and assignment to tablet servers. After the bulk file has been processed, the tablet server will assign the file to the appropriate tablets on all servers. This property controls the number of threads used to communicate to the other servers.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 1

tserver.bulk.process.threads

The master will task a tablet server with pre-processing a bulk file prior to assigning it to the appropriate tablet servers. This configuration value controls the number of threads used to process the files.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 1

tserver.bulk.retry.max

The number of times the tablet server will attempt to assign a file to a tablet as it migrates and splits.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 5

tserver.bulk.timeout

The time to wait for a tablet server to process a bulk import request.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5m

tserver.cache.data.size

Specifies the size of the cache for file data blocks.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 128M

tserver.cache.index.size

Specifies the size of the cache for file indices.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 512M

tserver.client.timeout

Time to wait for clients to continue scans before closing a session.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 3s

tserver.compaction.major.concurrent.max

The maximum number of concurrent major compactions for a tablet server

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 3

tserver.compaction.major.delay

Time a tablet server will sleep between checking which tablets need compaction.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 30s

tserver.compaction.major.thread.files.open.max

Max number of files a major compaction thread can open at once.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 10

tserver.compaction.minor.concurrent.max

The maximum number of concurrent minor compactions for a tablet server

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 4

tserver.compaction.warn.time

When a compaction has not made progress for this time period, a warning will be logged

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 10m

tserver.default.blocksize

Specifies a default blocksize for the tserver caches

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1M

tserver.dir.memdump

A long running scan could possibly hold memory that has been minor compacted. To prevent this, the in memory map is dumped to a local file and the scan is switched to that local file. We can not switch to the minor compacted file because it may have been modified by iterators. The file dumped to the local dir is an exact copy of what was in memory.

Type: PATH
ZooKeeper Mutable: yes
Default Value: /tmp

tserver.files.open.idle

Tablet servers leave previously used files open for future queries. This setting determines how much time an unused file should be kept open until it is closed.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 1m

tserver.hold.time.max

The maximum time for a tablet server to be in the "memory full" state. If the tablet server cannot write out memory in this much time, it will assume there is some failure local to its node, and quit. A value of zero is equivalent to forever.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5m

tserver.memory.manager

An implementation of MemoryManger that accumulo will use.

Type: CLASSNAME
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.server.tabletserver.LargestFirstMemoryManager

tserver.memory.maps.max

Maximum amount of memory that can be used to buffer data written to a tablet server. There are two other properties that can effectively limit memory usage table.compaction.minor.logs.threshold and tserver.walog.max.size. Ensure that table.compaction.minor.logs.threshold * tserver.walog.max.size >= this property.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1G

tserver.memory.maps.native.enabled

An in-memory data store for accumulo implemented in c++ that increases the amount of data accumulo can hold in memory and avoids Java GC pauses.

Type: BOOLEAN
ZooKeeper Mutable: yes but requires restart of the tserver
Default Value: true

tserver.metadata.readahead.concurrent.max

The maximum number of concurrent metadata read ahead that will execute.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 8

tserver.migrations.concurrent.max

The maximum number of concurrent tablet migrations for a tablet server

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 1

tserver.monitor.fs

When enabled the tserver will monitor file systems and kill itself when one switches from rw to ro. This is usually and indication that Linux has detected a bad disk.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: true

tserver.mutation.queue.max

The amount of memory to use to store write-ahead-log mutations-per-session before flushing them. Since the buffer is per write session, consider the max number of concurrent writer when configuring. When using Hadoop 2, Accumulo will call hsync() on the WAL . For a small number of concurrent writers, increasing this buffer size decreases the frequncy of hsync calls. For a large number of concurrent writers a small buffers size is ok because of group commit.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1M

tserver.port.client

The port used for handling client connections on the tablet servers

Type: PORT
ZooKeeper Mutable: yes but requires restart of the tserver
Default Value: 9997

tserver.port.search

if the ports above are in use, search higher ports until one is available

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

tserver.readahead.concurrent.max

The maximum number of concurrent read ahead that will execute. This effectively limits the number of long running scans that can run concurrently per tserver.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 16

tserver.recovery.concurrent.max

The maximum number of threads to use to sort logs during recovery

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 2

tserver.scan.files.open.max

Maximum total files that all tablets in a tablet server can open for scans.

Type: COUNT
ZooKeeper Mutable: yes but requires restart of the tserver
Default Value: 100

tserver.server.message.size.max

The maximum size of a message that can be sent to a tablet server.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1G

tserver.server.threadcheck.time

The time between adjustments of the server thread pool.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 1s

tserver.server.threads.minimum

The minimum number of threads to use to handle incoming requests.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 20

tserver.session.idle.max

maximum idle time for a session

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 1m

tserver.sort.buffer.size

The amount of memory to use when sorting logs during recovery.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 200M

tserver.tablet.split.midpoint.files.max

To find a tablets split points, all index files are opened. This setting determines how many index files can be opened at once. When there are more index files than this setting multiple passes must be made, which is slower. However opening too many files at once can cause problems.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 30

tserver.wal.blocksize

The size of the HDFS blocks used to write to the Write-Ahead log. If zero, it will be 110% of tserver.walog.max.size (that is, try to use just one block)

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 0

tserver.wal.replication

The replication to use when writing the Write-Ahead log to HDFS. If zero, it will use the HDFS default replication setting.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 0

tserver.wal.sync

Use the SYNC_BLOCK create flag to sync WAL writes to disk. Prevents problems recovering from sudden system resets.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: true

tserver.walog.max.size

The maximum size for each write-ahead log. See comment for property tserver.memory.maps.max

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1G

tserver.workq.threads

The number of threads for the distributed work queue. These threads are used for copying failed bulk files.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 2

A.3.6 logger.*

Properties in this category affect the behavior of the write-ahead logger servers

logger.dir.walog

The property only needs to be set if upgrading from 1.4 which used to store write-ahead logs on the local filesystem. In 1.5 write-ahead logs are stored in DFS. When 1.5 is started for the first time it will copy any 1.4 write ahead logs into DFS. It is possible to specify a comma-separated list of directories.

Type: PATH
ZooKeeper Mutable: yes
Default Value: walogs

A.3.7 gc.*

Properties in this category affect the behavior of the accumulo garbage collector.

gc.cycle.delay

Time between garbage collection cycles. In each cycle, old files no longer in use are removed from the filesystem.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5m

gc.cycle.start

Time to wait before attempting to garbage collect any old files.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 30s

gc.port.client

The listening port for the garbage collector's monitor service

Type: PORT
ZooKeeper Mutable: yes but requires restart of the gc
Default Value: 50091

gc.threads.delete

The number of threads used to delete files

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 16

gc.trash.ignore

Do not use the Trash, even if it is configured

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

A.3.8 monitor.*

Properties in this category affect the behavior of the monitor web server.

monitor.banner.background

The background color of the banner text displayed on the monitor page.

Type: STRING
ZooKeeper Mutable: yes
Default Value: #304065

monitor.banner.color

The color of the banner text displayed on the monitor page.

Type: STRING
ZooKeeper Mutable: yes
Default Value: #c4c4c4

monitor.banner.text

The banner text displayed on the monitor page.

Type: STRING
ZooKeeper Mutable: yes
Default Value:

monitor.lock.check.interval

The amount of time to sleep between checking for the Montior ZooKeeper lock

Type: TIMEDURATION
ZooKeeper Mutable: no
Default Value: 5s

monitor.port.client

The listening port for the monitor's http service

Type: PORT
ZooKeeper Mutable: no
Default Value: 50095

monitor.port.log4j

The listening port for the monitor's log4j logging collection.

Type: PORT
ZooKeeper Mutable: no
Default Value: 4560

A.3.9 trace.*

Properties in this category affect the behavior of distributed tracing.

trace.password

The password for the user used to store distributed traces

Type: STRING
ZooKeeper Mutable: no
Default Value: secret

trace.port.client

The listening port for the trace server

Type: PORT
ZooKeeper Mutable: no
Default Value: 12234

trace.table

The name of the table to store distributed traces

Type: STRING
ZooKeeper Mutable: no
Default Value: trace

trace.token.type

An AuthenticationToken type supported by the authorizer

Type: CLASSNAME
ZooKeeper Mutable: no
Default Value: org.apache.accumulo.core.client.security.tokens.PasswordToken

trace.user

The name of the user to store distributed traces

Type: STRING
ZooKeeper Mutable: no
Default Value: root

A.3.10 trace.token.property.*

The prefix used to create a token for storing distributed traces. For each propetry required by trace.token.type, place this prefix in front of it.

A.3.11 table.*

Properties in this category affect tablet server treatment of tablets, but can be configured on a per-table basis. Setting these properties in the site file will override the default globally for all tables and not any specific table. However, both the default and the global setting can be overridden per table using the table operations API or in the shell, which sets the overridden value in zookeeper. Restarting accumulo tablet servers after setting these properties in the site file will cause the global setting to take effect. However, you must use the API or the shell to change properties in zookeeper that are set on a table.

table.balancer

This property can be set to allow the LoadBalanceByTable load balancer to change the called Load Balancer for this table

Type: STRING
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.server.master.balancer.DefaultLoadBalancer

table.bloom.enabled

Use bloom filters on this table.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

table.bloom.error.rate

Bloom filter error rate.

Type: FRACTION
ZooKeeper Mutable: yes
Default Value: 0.5%

table.bloom.hash.type

The bloom filter hash type

Type: STRING
ZooKeeper Mutable: yes
Default Value: murmur

table.bloom.key.functor

A function that can transform the key prior to insertion and check of bloom filter. org.apache.accumulo.core.file.keyfu
and org.apache.accumulo.core.file.keyfunctor.ColumnQualifierFunctor are allowable values. One can extend any of the above mentioned classes to perform specialized parsing of the key.

Type: CLASSNAME
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.core.file.keyfunctor.RowFunctor

table.bloom.load.threshold

This number of seeks that would actually use a bloom filter must occur before a file's bloom filter is loaded. Set this to zero to initiate loading of bloom filters when a file is opened.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 1

table.bloom.size

Bloom filter size, as number of keys.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 1048576

table.cache.block.enable

Determines whether file block cache is enabled.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

table.cache.index.enable

Determines whether index cache is enabled.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: true

table.classpath.context

Per table classpath context

Type: STRING
ZooKeeper Mutable: yes
Default Value:

table.compaction.major.everything.idle

After a tablet has been idle (no mutations) for this time period it may have all of its files compacted into one. There is no guarantee an idle tablet will be compacted. Compactions of idle tablets are only started when regular compactions are not running. Idle compactions only take place for tablets that have one or more files.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 1h

table.compaction.major.ratio

minimum ratio of total input size to maximum input file size for running a major compaction-When adjusting this property you may want to also adjust table.file.max. Want to avoid the situation where only merging minor compactions occur.

Type: FRACTION
ZooKeeper Mutable: yes
Default Value: 3

table.compaction.minor.idle

After a tablet has been idle (no mutations) for this time period it may have its in-memory map flushed to disk in a minor compaction. There is no guarantee an idle tablet will be compacted.

Type: TIMEDURATION
ZooKeeper Mutable: yes
Default Value: 5m

table.compaction.minor.logs.threshold

When there are more than this many write-ahead logs against a tablet, it will be minor compacted. See comment for property tserver.memory.maps.max

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 3

table.failures.ignore

If you want queries for your table to hang or fail when data is missing from the system, then set this to false. When this set to true missing data will be reported but queries will still run possibly returning a subset of the data.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: false

table.file.blocksize

Overrides the hadoop dfs.block.size setting so that files have better query performance. The maximum value for this is 2147483647

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 0B

table.file.compress.blocksize

Similar to the hadoop io.seqfile.compress.blocksize setting, so that files have better query performance. The maximum value for this is 2147483647. (This setting is the size threshold prior to compression, and applies even compression is disabled.)

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 100K

table.file.compress.blocksize.index

Determines how large index blocks can be in files that support multilevel indexes. The maximum value for this is 2147483647. (This setting is the size threshold prior to compression, and applies even compression is disabled.)

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 128K

table.file.compress.type

One of gz,lzo,none

Type: STRING
ZooKeeper Mutable: yes
Default Value: gz

table.file.max

Determines the max # of files each tablet in a table can have. When adjusting this property you may want to consider adjusting table.compaction.major.ratio also. Setting this property to 0 will make it default to tserver.scan.files.open.max-1, this will prevent a tablet from having more files than can be opened. Setting this property low may throttle ingest and increase query performance.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 15

table.file.replication

Determines how many replicas to keep of a tables' files in HDFS. When this value is LTE 0, HDFS defaults are used.

Type: COUNT
ZooKeeper Mutable: yes
Default Value: 0

table.file.type

Change the type of file a table writes

Type: STRING
ZooKeeper Mutable: yes
Default Value: rf

table.formatter

The Formatter class to apply on results in the shell

Type: STRING
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.core.util.format.DefaultFormatter

table.groups.enabled

A comma separated list of locality group names to enable for this table.

Type: STRING
ZooKeeper Mutable: yes
Default Value:

table.interepreter

The ScanInterpreter class to apply on scan arguments in the shell

Type: STRING
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.core.util.interpret.DefaultScanInterpreter

table.majc.compaction.strategy

A customizable major compaction strategy.

Type: CLASSNAME
ZooKeeper Mutable: yes
Default Value: org.apache.accumulo.tserver.compaction.DefaultCompactionStrategy

table.scan.max.memory

The maximum amount of memory that will be used to cache results of a client query/scan. Once this limit is reached, the buffered data is sent to the client.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 512K

table.security.scan.visibility.default

The security label that will be assumed at scan time if an entry does not have a visibility set.
Note: An empty security label is displayed as []. The scan results will show an empty visibility even if the visibility from this setting is applied to the entry.
CAUTION: If a particular key has an empty security label AND its table's default visibility is also empty, access will ALWAYS be granted for users with permission to that table. Additionally, if this field is changed, all existing data with an empty visibility label will be interpreted with the new label on the next scan.

Type: STRING
ZooKeeper Mutable: yes
Default Value:

table.split.threshold

When combined size of files exceeds this amount a tablet is split.

Type: MEMORY
ZooKeeper Mutable: yes
Default Value: 1G

table.walog.enabled

Use the write-ahead log to prevent the loss of data.

Type: BOOLEAN
ZooKeeper Mutable: yes
Default Value: true

A.3.12 table.constraint.*

Properties in this category are per-table properties that add constraints to a table. These properties start with the category prefix, followed by a number, and their values correspond to a fully qualified Java class that implements the Constraint interface.
For example:
table.constraint.1 = org.apache.accumulo.core.constraints.MyCustomConstraint
and:
table.constraint.2 = my.package.constraints.MySecondConstraint

A.3.13 table.iterator.*

Properties in this category specify iterators that are applied at various stages (scopes) of interaction with a table. These properties start with the category prefix, followed by a scope (minc, majc, scan, etc.), followed by a period, followed by a name, as in table.iterator.scan.vers, or table.iterator.scan.custom. The values for these properties are a number indicating the ordering in which it is applied, and a class name such as:
table.iterator.scan.vers = 10,org.apache.accumulo.core.iterators.VersioningIterator
These iterators can take options if additional properties are set that look like this property, but are suffixed with a period, followed by 'opt' followed by another period, and a property name.
For example, table.iterator.minc.vers.opt.maxVersions = 3

A.3.14 table.group.*

Properties in this category are per-table properties that define locality groups in a table. These properties start with the category prefix, followed by a name, followed by a period, and followed by a property for that group.
For example table.group.group1=x,y,z sets the column families for a group called group1. Once configured, group1 can be enabled by adding it to the list of groups in the table.groups.enabled property.
Additional group options may be specified for a named group by setting table.group.<name>.opt.<key>=<value>.

A.3.15 table.majc.compaction.strategy.opts.*

Properties in this category are used to configure the compaction strategy.

A.3.16 general.vfs.context.classpath.*

Properties in this category are define a classpath. These properties start with the category prefix, followed by a context name. The value is a comma seperated list of URIs. Supports full regex on filename alone. For example, general.vfs.context.classpath.cx1=hdfs://nn1:9902/mylibdir/*.jar. You can enable post delegation for a context, which will load classes from the context first instead of the parent first. Do this by setting general.vfs.context.classpath.<name>.delegation=post, where <name> is your context nameIf delegation is not specified, it defaults to loading from parent classloader first.

A.4 Property Types

A.4.1 duration

A non-negative integer optionally followed by a unit of time (whitespace disallowed), as in 30s.
If no unit of time is specified, seconds are assumed. Valid units are 'ms', 's', 'm', 'h' for milliseconds, seconds, minutes, and hours.
Examples of valid durations are '600', '30s', '45m', '30000ms', '3d', and '1h'.
Examples of invalid durations are '1w', '1h30m', '1s 200ms', 'ms', '', and 'a'.
Unless otherwise stated, the max value for the duration represented in milliseconds is 9223372036854775807

A.4.2 date/time

A date/time string in the format: YYYYMMDDhhmmssTTT where TTT is the 3 character time zone

A.4.3 memory

A positive integer optionally followed by a unit of memory (whitespace disallowed), as in 2G.
If no unit is specified, bytes are assumed. Valid units are 'B', 'K', 'M', 'G', for bytes, kilobytes, megabytes, and gigabytes.
Examples of valid memories are '1024', '20B', '100K', '1500M', '2G'.
Examples of invalid memories are '1M500K', '1M 2K', '1MB', '1.5G', '1,024K', '', and 'a'.
Unless otherwise stated, the max value for the memory represented in bytes is 9223372036854775807

A.4.4 host list

A comma-separated list of hostnames or ip addresses, with optional port numbers.
Examples of valid host lists are 'localhost:2000,www.example.com,10.10.1.1:500' and 'localhost'.
Examples of invalid host lists are '', ':1000', and 'localhost:80000'

A.4.5 port

An positive integer in the range 1024-65535, not already in use or specified elsewhere in the configuration

A.4.6 count

A non-negative integer in the range of 0-2147483647

A.4.7 fraction/percentage

A floating point number that represents either a fraction or, if suffixed with the '%' character, a percentage.
Examples of valid fractions/percentages are '10', '1000%', '0.05', '5%', '0.2%', '0.0005'.
Examples of invalid fractions/percentages are '', '10 percent', 'Hulk Hogan'

A.4.8 path

A string that represents a filesystem path, which can be either relative or absolute to some directory. The filesystem depends on the property. The following environment variables will be substituted: [ACCUMULO_HOME, ACCUMULO_CONF_DIR]

A.4.9 absolute path

An absolute filesystem path. The filesystem depends on the property. This is the same as path, but enforces that its root is explicitly specified.

A.4.10 java class

A fully qualified java class name representing a class on the classpath.
An example is 'java.lang.String', rather than 'String'

A.4.11 string

An arbitrary string of characters whose format is unspecified and interpreted based on the context of the property to which it applies.

A.4.12 boolean

Has a value of either 'true' or 'false'

A.4.13 uri

A valid URI