O'REILLY®

Accumulo

APPLICATION DEVELOPMENT, TABLE DESIGN, AND BEST PRACTICES

Aaron Cordova,
Billie Rinaldi & Michael Wall

O'REILLY®

Accumulo

Get up to speed on Apache Accumulo, the flexible, high-performance key-value store created by the US National Security Agency (NSA) and based on Google's Bigtable data storage system. Written by former NSA team members, this comprehensive tutorial and reference covers Accumulo architecture, application development, table design, and cell-level security.

With clear information on system administration, performance tuning, and best practices, this book is ideal for developers seeking to write Accumulo applications, administrators charged with installing and maintaining Accumulo, and other professionals interested in what Accumulo has to offer. You will find everything you need to use this system fully.

- Get a high-level introduction to Accumulo's architecture and data model
- Take a rapid tour through single- and multiple-node installations, data ingest, and query
- Learn how to write Accumulo applications for several use cases, based on examples
- Dive into Accumulo internals, including information not available in the documentation
- Get detailed information for installing, administering, tuning, and measuring performance
- Learn best practices based on successful implementations in the field

"If you need random access to large datasets you'd be wise to learn about Accumulo. And there's no better place to start than with this book."

—**Doug Cutting**
Founder of Hadoop

"Aaron Cordova, Billie Rinaldi, and Michael Wall have been leaders in the Accumulo community since its inception, and I can think of no one more qualified to write the definitive book on Accumulo."

—**Jeremy Kepner**
MIT Lincoln Laboratory

Aaron Cordova, a cofounder of Koverse Inc., started and led the Apache Accumulo project as a computer systems researcher at the US National Security Agency.

Billie Rinaldi, a senior technical staff member at Hortonworks, Inc., was a leader of the NSA computer science research team that implemented Accumulo.

Michael Wall, a graduate of the US Air Force Academy, served as a software engineer for the NSA and other government agencies. He develops a variety of applications with Accumulo.

DATA / DATABASES / SECURITY

US $49.99 CAN $57.99
ISBN: 978-1-449-37418-1

Twitter: @oreillymedia
facebook.com/oreilly

Accumulo: Application Development, Table Design, and Best Practices
by Aaron Cordova, Billie Rinaldi, and Michael Wall

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Marie Beaugureau
Production Editor: Matthew Hacker
Copyeditor: Kim Cofer
Proofreader: Eileen Cohen

Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrators: Aaron Cordova and Billie Rinaldi

July 2015: First Edition

Revision History for the First Edition
2015-06-30: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781449374181* for release details.

978-1-449-37418-1

[LSI]

Table of Contents

CHAPTER 1

Architecture and Data Model

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

Recent Trends

Over the past few decades, several trends have driven the progress of data storage and processing systems. The first is that more data is being produced, at faster rates than ever before. The rate of available data is increasing so fast that more data was produced in the past few years than in all previous years. In recent years a huge amount of data has been produced by people for human consumption, and this amount is dwarfed by the amount of data produced by machines. These systems and devices promise to generate an enormous amount of data in the coming years. Merely storing this data can be a challenge, let alone organizing and processing it.

The second trend is that the cost of storage has dropped dramatically. Hard drives now store multiple terabytes for roughly the same price as gigabyte drives stored gigabytes of data a decade ago. Although computer memory is also falling in price, making it possible for many applications to run with their working data sets entirely in memory, systems that store most data on disk still have a big cost advantage.

The third trend is that disk throughput has improved more than disk seek times, for conventional spinning-disk hard drives. Though solid-state drives (SSDs) have altered this balance somewhat, the advantage of the sequential read performance of

conventional hard drives versus random read performance is a large factor in the design of the systems we'll be discussing.

Finally, we've seen a shift from using one processor to multiple processors as increases in single-processor performance have slowed. This is reflected in a shift not only to multithreaded programs on a single server but also to programs distributed over multiple separate servers.

These trends have caused system and application developers to take a hard look at conventional designs and to consider alternatives. The question many are asking is: how should we build applications so we can take advantage of all this data, in light of current hardware trends, and in the most cost-effective way possible?

The Role of Databases

Conventional relational databases have served as the workhorse for persisting application data and as the processing engine for data analysis for many years. With the advent of the World Wide Web, web applications can be exposed to millions of concurrent users, creating the need for highly scalable data storage and retrieval technologies. Many applications begin with a single relational database as the storage engine and gradually reduce the number of features enabled on the database in order to get better performance and serve more requests per second. Eventually a single database is just not enough, and applications begin to resort to distributing data among several database instances in order to keep up with demand. All of the overhead for managing multiple databases and distributing data to them has to be handled by the application.

Similarly, databases have also played an important role in analytical applications. Often a relational database will be at the center of a data warehouse in which records from operational databases are combined and refactored to support queries that answer analytical questions. The field of Business Intelligence has grown up around the capabilities of data warehouses. As more and more data becomes available, the need for these analytical systems to scale becomes greater. Not only are organizations collecting and keeping more structured data from operational systems, but interest is also growing in other types of data that's less well-structured—such as application logs, social media data, and text documents. The ability to combine all of these data sets in one place in order to ask questions across them is a compelling use case that is driving innovation in scalable systems.

Accumulo is unlike some other new distributed databases in that it was developed with more of a focus on building analytical platforms, rather than simply as the scalable persistence layer for data generated via a web application. The flexibility of the data model and support for building indexes in Accumulo make analyzing data from a variety of sources easier. Accumulo also introduces fine-grained access control to

make it possible for organizations to confidently protect data of varying sensitivity levels in the same physical cluster.

Analysis and Column Storage

Many analytical databases take advantage of column-oriented storage rather than row-oriented storage, which is the primary storage for most databases.

Row-oriented storage is useful for operational applications that need to maintain some state across multiple fields or multiple rows. When updating multiple fields in a row, perhaps as part of a transaction, it is convenient to store all the fields that need to be updated simultaneously together on disk, read them off of disk together into memory in order to change values as part of a transaction, and write them back to disk together to maintain a consistent view of the data at all times.

In contrast, analytical applications often do not require any updates to data and are instead aggregating and summarizing the data. In many cases analytical questions are designed to calculate some statistic for one or a subset of the fields across all of the rows. It is inconvenient to store data in row-oriented format because it requires all the fields of one row to be read before any fields of the next row can be accessed. As a result, analytical storage engines often store data in column-oriented formats. This way, all of the data for a particular field across all rows can be found together on disk. This drastically reduces the time required to read data to answer these types of analytical questions. Because similarity is a property that compression relies on to reduce storage size, column-oriented storage also improves the opportunities for compression because the data values within a single field are often similar to one another.

Accumulo makes it possible to group sets of columns together on disk via a feature called *locality groups* so analytical applications can gain these advantages. As part of Accumulo's additional focus on analytical applications, its support for locality groups is more powerful than in some other distributed databases because the names of columns don't have to be declared beforehand, there is no penalty for a large number of different column names, and the columns can be mapped to locality groups in any way desired. We discuss locality groups in depth in "Column Families" on page 19.

Some relational databases have adopted a distributed approach to scaling to meet demand. In all distributed systems there are trade-offs. Distributed applications introduce new complexities and failure modes that might not have existed in one-server applications, so many distributed applications also ensure that the design and APIs offered are simple to make understanding the behavior of the entire system easier. In many ways new platforms like Accumulo represent stepping back to look at the problem and building a data store from the ground up to support these larger workloads and the concise set of features they require. The goal of Accumulo, being based on Google's Bigtable, is to provide a set of features that work well even as data sizes

grow into the tens of petabytes—even in the presence of the regular failures expected of cheaper, commodity-class hardware that is commonly used.

Distributed Applications

To effectively use increasing amounts of available data, a few application design patterns have emerged for automatically distributing data and processing over many separate commodity-class servers connected via a network, and that vastly prefer sequential disk operations over random disk seeks. Unlike some distributed systems, applications that implement these patterns do not share memory or storage, an approach called a *shared-nothing architecture*. These applications are designed to handle individual machine failures automatically with no interruption in operations.

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

Shared-Nothing Architectures

Some distributed applications are built to run on hardware platforms featuring many processors and large amounts of shared random-access memory (RAM), and often connect to a storage area network (SAN) via high-speed interconnects such as Fibre Channel to access shared data storage.

In contrast, shared-nothing architectures do not share RAM and do not connect to shared storage, but rather consist of many individual servers, each with its own processors, RAM, and hard drives. These systems are still connected to one another via a network such as Gigabit Ethernet. Often the individual servers are of the more inexpensive sort and often include cheaper individual components, such as Serial ATA (SATA) drives rather than Small Computer System Interface (SCSI) drives.

Technologies that increase the resilience of a single server, such as hardware Redundant Array of Independent Disks (RAID) cards, which allow several hard drives within a server to be grouped together for redundancy, are unnecessary in a shared-nothing architecture. These can be replaced with an application layer that tolerates the failure of entire servers, such as the Hadoop Distributed File System (HDFS).

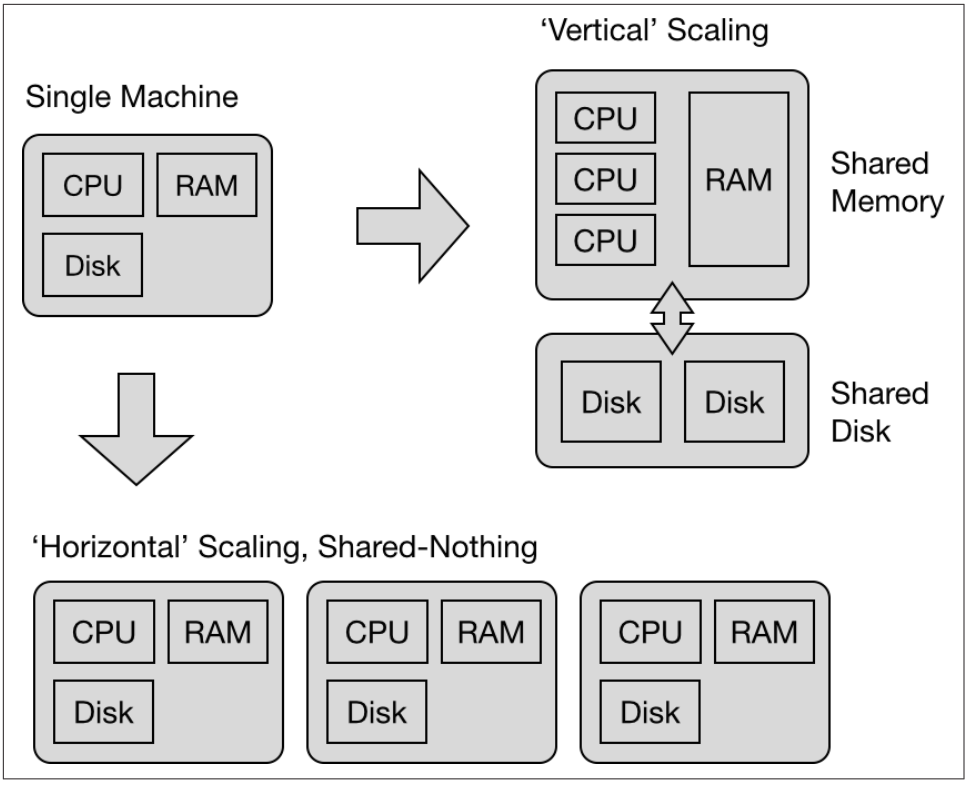

Figure 1-1. Scaling strategies

Accumulo employs this distributed approach by partitioning data across multiple servers and keeping track of which server has which partition. In some cases these data partitions are called *shards*, as in pieces of something that has been shattered. In Accumulo's case, data is stored in tables, and tables are partitioned into *tablets*. Each server hosts a number of tablets. These servers are called *tablet servers* (Figure 1-2).

Some other systems support this type of data partitioning and require that a particular field within the data be specified for the purpose of mapping a particular row to a partition. For example, a relational database may allow a table to be split into partitions based on the *Date* field. All of the rows that have a date value in January might be in one partition, and the rows with a date value in February in another. This structure is very sensitive to the distribution of values across rows. If many more rows have date values in February, that partition will be larger than the other partitions.

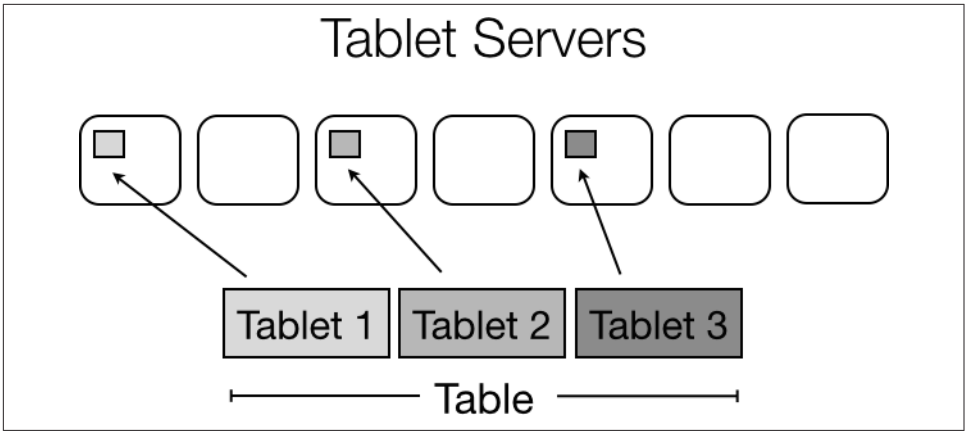

Figure 1-2. Tables are partitioned into tablets and distributed

In contrast, Accumulo does not require you to specify how to partition data. Instead, it automatically finds good points to use to split the data into tablets. As new data arrives, a particular single tablet may become larger than the others. When it reaches a configurable threshold, the tablet is split into two tablets. This way, tablets can be uniform in size without any intervention from administrators.

Partitions also have to be mapped to particular servers. If responsibility for storage is coupled with responsibility for processing requests for a particular tablet, movement of read and write processing for a tablet from one server to another also requires that the data be moved. This data movement can be expensive. So, rather than coupling responsibility for reads and writes with the storage of a tablet, Accumulo allows tablet servers to be responsible for tablets that are stored on another server, at least temporarily. Over time, tablet servers will create local copies of the data in background operations to avoid reads over the network in response to client requests.

The flexibility in assigning tablets to tablet servers allows Accumulo to be very responsive to handling individual hardware failures without requiring additional intervention from applications or administrators. This is crucial to running a large-scale cluster, because hardware failure becomes a common occurrence with hundreds or thousands of machines. Instances of Accumulo have been known to run on more than a thousand servers, hosting trillions of key-value pairs.[1]

Accumulo includes features that can be used to build a wide variety of scalable distributed applications, including storing structured or semistructured sparse and dynamic data, building rich text-search capabilities, indexing geospatial or multidi-

1 R. Sen, A. Farris, and P. Guerra, "Benchmarking Apache Accumulo BigData Distributed Table Store Using Its Continuous Test Suite." in IEEE International Congress on Big Data, 2013, pp. 334–341.

mensional data, storing and processing large graphs, and maintaining continuously updated summaries over raw events using server-side programming mechanisms.

Fast Random Access

Fast random access is important to many applications. *Random* access implies that even though the particular element of data that is sought is not known until the time of execution, the access time for any particular data element is roughly the same. This is in contrast to *sequential* access, in which the reads start at the beginning of a set of data and proceed to read more data until reaching the end. It's also important that that access time be fast enough to satisfy application requirements. Many web applications require that the data requested be accessible in less than one second.

There are several techniques for achieving good random-access performance. Two popular techniques are hashing and sorting. These techniques are used all the time in computer applications accessing data held in memory, but they conveniently also apply to data stored on disk, and even across multiple machines.

Unlike Hadoop jobs, where the data is often unorganized and where each job processes most or all of the data, Accumulo is designed to store data in an *organized* fashion so users can quickly find the data they need or incrementally add to or update a data set. Accumulo's role in life is to store key-value pairs, keeping the keys sorted at all times. This enables applications to achieve fast, interactive response times even when the data sizes range in the petabytes.

Accessing Sorted Versus Unsorted Data

Imagine a scenario in which you need to catch a flight, and your ticket shows your flight leaving from gate D5. Suppose that the gates are unordered; that is, gate A1 is right next to F3, which is right next to B2. If you are currently standing at gate B2, you would have no idea how close you are to D5, and no idea in which direction you should go to get closer to D5. The only strategy guaranteed to locate gate D5 is to begin visiting all the gates in the hope that you stumble across D5. This strategy is fine if you have hours and hours to spend searching. If you're in a hurry, chances are you will miss your flight. Not only is this too slow to be practical, but it is horribly inefficient. Every person trying to catch a flight will waste at least several hours and a lot of effort finding the right gate.

If the gates are sorted in a known order, such as alphabetical and numerical order so that gate A1 is physically next to gate A2 and the last A gate is next to the first B gate, finding a particular gate is much easier. You know that to find gate D5 you must skip all the A, B, and C gates, and that if you see E gates you've gone too far. Once you've found one of the D gates, say D8, you know that your gate is only three gates away. If

the next gate you see is D7 or D9, you now know whether to keep going or to turn around to get to D5.

This is the same way that computers use sorted data. A computer uses an algorithm known as a *binary search* to find a key-value pair in a list sorted by key (Figure 1-3). Binary search works by looking at the key in the middle of the list and comparing that to the key it wants to find. If the key in the middle of the list is *greater than* the key sought, the computer will then search the first half of the list. If the key in the middle of the list is *less than* the key sought, the computer will search the second half of the list.

Whichever half is chosen, the computer again picks the key in the middle and compares that to the key it's looking for, and based on this comparison it decides in which direction it must continue searching. This continues until the computer finds an exact match or determines that the key sought is not in the list.

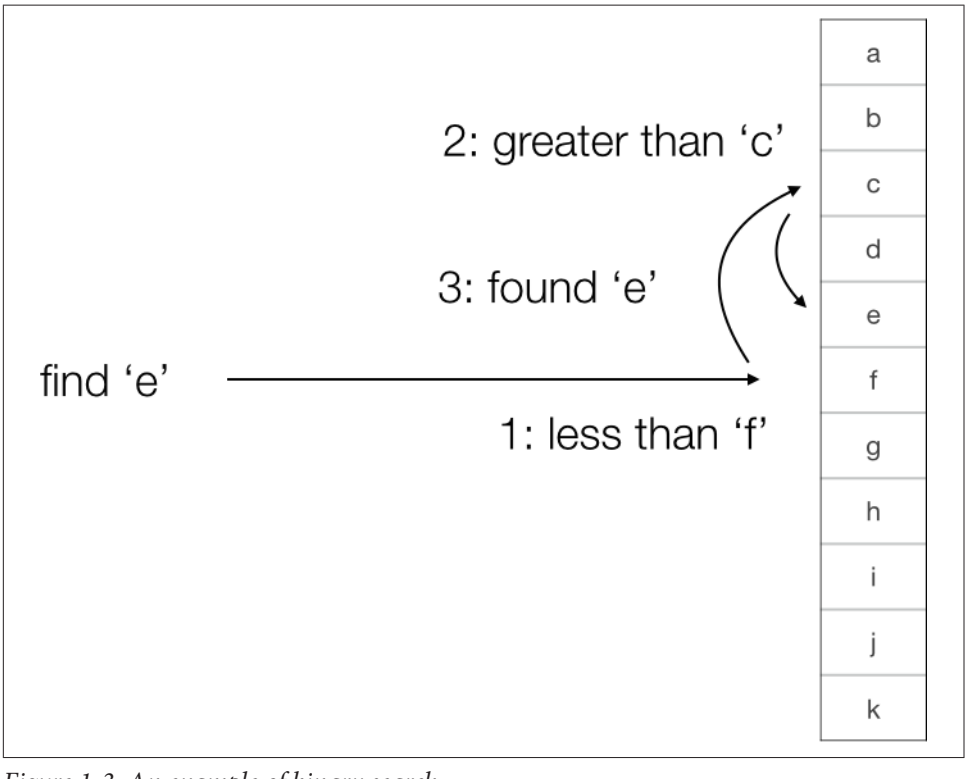

Figure 1-3. An example of binary search

This dramatically reduces the number of keys that must be examined and makes searching for a particular key faster. How much faster? If it takes 10 milliseconds to fetch and examine one key, finding a particular key in an unsorted list of a billion

keys will take an average of *57 days*, because the right key could be anywhere—best case it's the first one you look at; worst case it's the last.

If the list is sorted, it only takes an average of *300 milliseconds*. If the sorted list has not a billion key-value pairs, but a trillion, it takes *400 milliseconds*—only 30 percent longer for a 1000× increase in data!

Algorithms that have this kind of performance are said to exhibit *logarithmic* access time with respect to the number of data elements, as opposed to *linear* access time, because the access time is a function not of the number of data elements but of the logarithm of the number of elements.

Hashing Versus Sorting

Hashing is a popular technique for organizing data so that a given data element can be accessed quickly. If we are storing key-value pairs, where each key is associated with a single value, a hash function applied to the key can be used to determine where a key-value pair will be stored. Good hash functions map inputs to a range of output values uniformly. When storing key-value pairs, the key is passed as the input to the hash function (called *hashing the key*) and the output hash is used as the address of the key-value pair in the storage medium. For example, we might decide to store the key-value pair *favoriteColor→red* by first hashing the key, *favoriteColor*, which produces the value 1004, and so we store that key-value pair in the 1004th slot in memory.

Lookups designed to retrieve the value of a known key consist of hashing the key, noting the hash output value and jumping to the place referenced by that hash, and retrieving the value of the key-value pair. The hash can refer to a location in memory, on disk, or on a particular machine in a cluster. If we need to look up the value for the key *favoriteColor*, we simply hash it to obtain the address 1004 and go directly to the 1004th memory slot to retrieve the key-value pair (Figure 1-4).

In distributed systems hashing is sometimes used to distribute key-value pairs across machines in a cluster. Hashing has the advantage of not requiring the system to do anything special to keep the data uniformly spread out across machines. Lookups can consist of simply hashing the key to find the server on which a key-value pair is stored, and then hashing again to find the spot within the server that contains the key-value pair.

Because these lookups consist of just one step, hashing enables very fast random access to data. However, because the hash function is designed to spread keys out uniformly across the address space, any similarity among keys is lost. For, example if we wanted to be able to access the values for not just *favoriteColor* but *favoriteIceCream* and *favoriteMovie*, we would have to do three separate lookups because these key-value pairs would end up being assigned to different places by the hash function.

Accumulo does not rely on hashing for data distribution; it uses sorting instead.

Like hashing, sorting data can enable fast random access, but unlike hashing, sorting can preserve some of the relationships among keys. This way, we can quickly find one key that we want by doing a binary search, but also any closely related keys by reading a few additional keys that appear sequentially after the first key. Because disks can read sequential data much faster than accessing data randomly, the difference between finding and returning one key versus finding one key and scanning 1,000 of the keys that follow sequentially is minimal.

This property of sorted data allows application designers to exploit any relationships, sometimes called *locality*, in their data by creating keys that group related information together when sorted (Figure 1-5).

Maintaining asorted set of key-value pairs, especially when distributed across multiple machines, is more work than using hashing. Specifically, you have to maintain an additional mapping of which machine has which portion of the sorted set. In Accumulo, this mapping is called the *metadata table*, and Accumulo has a lot of functionality built in to handle the additional work of maintaining this information. We discuss the metadata table in depth in "Metadata Table" on page 379.

Figure 1-4. Hashing a key to an address

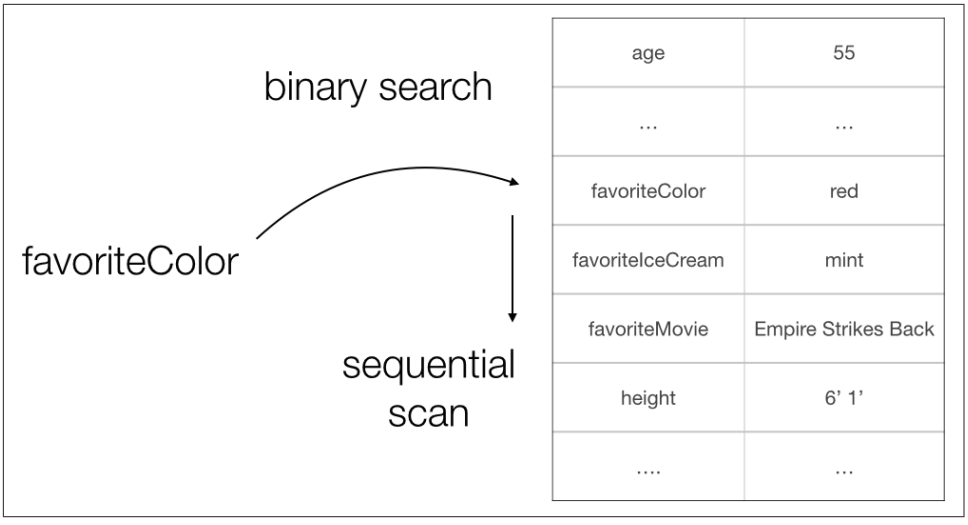

Figure 1-5. Accessing related keys in sorted data

Versions

The first public open source version of Accumulo is 1.3.

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

As of this writing, the latest stable version of Accumulo is 1.6. We will focus this book on version 1.6, pointing out differences in other versions where appropriate. Version 1.6 includes the following new features and improvements over previous versions:

- Multivolume support (running over multiple HDFS instances)
- Table namespaces
- Conditional mutations
- Partial encryption support
- Pluggable compaction strategies
- Lexicoders (tools for sorting tuples properly)
- Locality groups in memory
- Service IP addresses
- Support for ViewFS
- Maven plug-in
- Default key size constraint

You can find the complete Release Notes for the 1.6 release at the Apache Accumulo site.

History

Accumulo is one of several implementations based on Google's Bigtable. The others include Apache HBase, Hypertable, and Apache Cassandra.

Accumulo has been an open source project since 2011 and has since seen several releases. A brief history of the project is as follows:

2003
: Google publishes a paper describing the Google File System (GFS), a distributed filesystem for storing very large files across many commodity-class servers.

2004
: Google publishes a paper describing a simplified distributed programming model and associated fault-tolerant execution framework called MapReduce.

2006
: Google publishes a paper entitled "Bigtable: A Distributed Storage System for Structured Data". That same year a team from Yahoo! releases an open source version called Apache Hadoop.

Fall 2007
: An open source implementation of Google's Bigtable called HBase is started by a team at the company Powerset.

January 2008
: Hadoop becomes a top-level Apache project. HBase becomes a subproject.

  At the same time, a team of computer scientists and mathematicians at the US National Security Agency (NSA) are evaluating the use of various big data technologies, including Apache Hadoop and HBase, in an effort to help solve the issues involved with storing and processing large amounts of data of different sensitivity levels. Authors Billie Rinaldi and Aaron Cordova are part of this team.

July 2008
: Powerset is acquired by Microsoft.

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

The team extends the Bigtable design with additional features that includes a method for labeling each key-value pair with its own access information, called *column visibilities*, and a mechanism for performing additional server-side functionality, called *iterators*.

May 2009
: Version 1.0 of Accumulo is released, but it is not yet publicly available.

May 2010
: HBase becomes a top-level Apache project.

September 2011
: Accumulo becomes a public open source incubator project hosted by the Apache Software Foundation.

March 2012
: Accumulo graduates to top-level project status. First publicly available release is 1.3.5.

April 2012
: Version 1.4 is released.

May 2013
: Version 1.5 is released and includes a Thrift proxy, more control over compactions, and table import and export

May 2014
: Version 1.6 is released and extends the API to include conditional mutations and table namespaces.

Data Model

At the most basic level, Accumulo stores key-value pairs on disk (Figure 1-6), keeping the keys sorted at all times. This allows a user to look up the value of a particular key or range of keys very quickly. Values are stored as byte arrays, and Accumulo doesn't restrict the type or size of the values stored. The default constraint on the maximum size of the key is 1 MB.

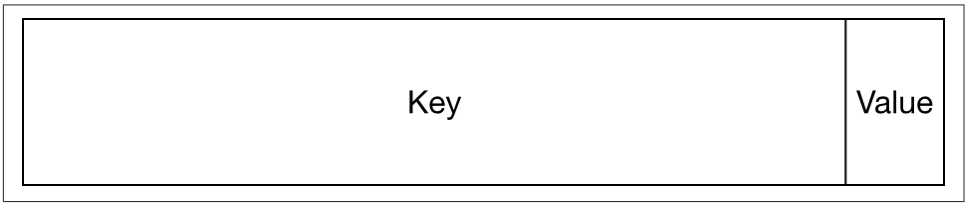

Figure 1-6. A simple key-value pair

Rather than simple keys as shown in Figure 1-6, Accumulo keys are made up of several components. Inside the key there are three main components: a row ID, a column, and a timestamp (Figure 1-7).

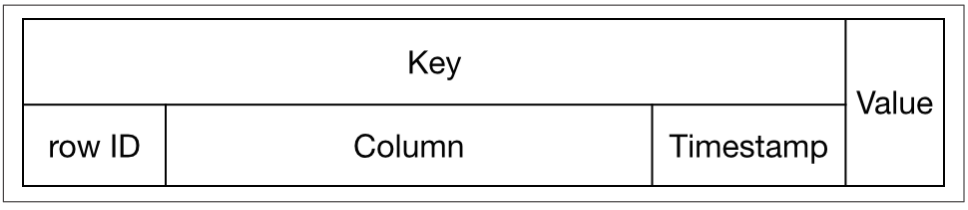

Figure 1-7. Main components of the key

Rows and Columns

The row ID and column components allow developers to model their data similarly to how one might store data in a relational database, or perhaps a spreadsheet. One major difference is that relational databases often have autogenerated row IDs and rely on secondary indexes for all data access, whereas the row IDs in Accumulo can contain data that is relevant to an application.

When sorting keys, Accumulo first sorts the data by row ID, then sorts keys with the same row ID by column, and finally sorts keys with the same row ID and column by timestamp. Row IDs and columns are sorted in ascending, *lexicographical* order—which means, roughly, alphabetical order—byte-by-byte.

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

The ability to modify rows *atomically* is an important feature for application designers to keep in mind when modeling their data. This means that Accumulo will commit the changes to a particular row all at once, or not at all in the case of a failure. This allows applications always to have a consistent view of the data in a row, and not to have to handle cases in which a change is partially applied. (We discuss atomicity more in "Transactions" on page 47.)

Columns allow a row to contain multiple elements, as in a relational database table. Each column is mapped to a value. But unlike in a relational database, you don't have to declare columns before storing data in them, and not every row has to have the same columns present. Further, the type of data stored under a particular column does not have to be the same across rows. Finally, columns do not have a specified maximum length in which values must fit. (Column *names*, being part of the key, are

by default limited, because the total key is constrained to be less than 1 MB. However, the values under these columns are not constrained in size by default.)

Accumulo tables can cope with missing or additional columns and changes in the underlying schema of the data because Accumulo does not make any assumptions about the schema. If rows imported every day for a month contain 10 columns and suddenly they now contain 11 columns, Accumulo will not reject a request to store the new rows; it will simply store them. Applications designed to read from the 10 known columns can continue to do so even with the new rows and simply ignore the additional column.

This departure from the relational model represents a trade-off. On the one hand, the flexibility makes storing data much easier. It is easier to store data that does not conform to a well-known schema, and it is also easier to store data whose structure changes over time.

However, whereas applications built on a relational database can rely on the database to ensure that values conform to specified types and lengths, applications built on Accumulo cannot assume that value types and lengths conform to any constraints, unless Accumulo is configured to apply specific constraints to the data. Application designers can decide whether to implement constraints to be applied by Accumulo at insert-time or whether to handle varying value types and lengths at read-time in the application.

For example, we may have a table that we use to store Wikipedia articles. The table contains some structured data, or *metadata*, about each article, along with the actual article text. Individual metadata elements may not be the same from one article to the next.

Notice that not all the rows in Figure 1-8 have data stored in every column, a property known as *sparseness*. In other systems, missing values must be indicated by storing a `NULL` value, which takes up space on disk. In Accumulo, the missing values simply do not appear in the list of key-value pairs. On disk, this data is laid out as a long series of sorted key-value pairs.

rowid	title	pageid	comment	text
Apache_Accumulo	Apache Accumulo	571876229	/* See also */ Added Column-oriented DBMS	Apache Accumulo is a sorted, distributed key/value store based on Google's BigTable. It is a system built on top of Apache Hadoop, Apache ZooKeeper, and Apache Thrift. …
Apache_Hadoop	Apache Hadoop	5919308		Apache Hadoop is an open source software framework for storage and large-scale processing of data-sets on clusters of commodity hardware. Hadoop is an Apache top-level project being built …
Apache_Thrift	Apache Thrift	10438451		Thrift is an interface definition language and binary communication protocol that is used to define and create services for numerous languages …
BigTable	BigTable	5919973		BigTable is a compressed, high performance, and proprietary data storage system built on Google File System, Chubby Lock Service, SSTable (log-structured storage like LevelDB) …

Figure 1-8. A table consisting of rows and columns

Note that there is no key-value pair in Figure 1-9 for the *comment* field for *Apache Thrift*, for example. Because Accumulo stores data this way, it can handle sparse data sets very efficiently. Writing a key-value pair that contains a column that doesn't appear in any other row is no different from Accumulo's perspective than storing any other key-value pair.

If you are coming from a relational database background, it might be confusing to think of a row in Accumulo as a set of key-value pairs. Looking at data retrieved in the Accumulo shell, which we touch on first in "Demo of the Shell" on page 60, a row will actually be many lines on the screen. Figure 1-8 may be a more familiar representation of the data, and you can see how it might translate into Accumulo in Figure 1-9. In this example, a row, defined as a set of key-value pairs, is analogous to a record in a relational database. Everything with the same row ID contains information about a given record.

rowid	column	value
Apache_Accumulo	comment	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	pageid	571876229
Apache_Accumulo	text	Apache Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	title	Apache Accumulo
Apache_Hadoop	pageid	5919308
Apache_Hadoop	text	Apache Hadoop is an open source software framework for storage …
Apache_Hadoop	title	Apache Hadoop
Apache_Thrift	pageid	10438451
Apache_Thrift	text	Thrift is an interface definition language and binary …
Apache_Thrift	title	Apache Thrift
…	…	…

Figure 1-9. Key-value pairs representing data for various rows and columns

Data Modification and Timestamps

Accumulo allows applications to update and delete existing information. These operations are essential to developing operational applications. Rather than modifying the data already written to disk, however, Accumulo handles modifications of this type via versioning.

The timestamp element of the key adds a new dimension to the well-known two-dimensional row-column model, and this allows data under a particular row-column pair to have more than one version (Figure 1-10). By default, Accumulo keeps only the newest version of a row-column pair, but it can be configured to store a specific number of versions, versions newer than a certain date, or all versions ever written.

rowid	title	pageid	comment	text
Apache_Accumulo	Apache Accumulo	571876229	/* See also */ Added Column-oriented DBMSany town	Accumulo is a sorted, distributed key/value store based on Google's B… Apache Accumulo is a sorted, distributed key/value store based on Google's BigTable. It is a system built on top of Apache Hadoop, Apache ZooKeeper, and …

Figure 1-10. A table consisting of rows and columns with multiple versions

The set of key-value pairs on disk appears as in Figure 1-11.

rowid	column	timestamp	value
Apache_Accumulo	comment	20111001	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	pageid	20111001	571876229
Apache_Accumulo	text	20120301	Apache Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	text	20111001	Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	title	20111001	Apache Accumulo
Apache_Hadoop	pageid	20060314	5919308
Apache_Hadoop	text	20060314	Apache Hadoop is an open source software framework for storage …
Apache_Hadoop	title	20060314	Apache Hadoop
Apache_Thrift	pageid	20070213	10438451
Apache_Thrift	title	20070213	Apache Thrift
…	…		…

Figure 1-11. Two key-value pairs that represent two versions of data for one row-column pair

Timestamps are stored as 64-bit integers using the Java `long` data type. They are sorted in descending order, unlike rows and columns, so that the newest versions of a row-column pair appear first when scanning down a table. In this way, Accumulo handles updates by simply storing new versions of key-value pairs. If only the newest version is retrieved, it appears as if the value has changed.

Timestamps that are assigned to key-value pairs by the tablet server use the number of milliseconds since midnight, January 1, 1970, also known as the Unix epoch.

Similarly, deletes are implemented using a special marker inserted in front of any existing versions. The appearance of a key-value pair with a delete marker is interpreted by Accumulo to mean "ignore all versions of this row-column pair older than this timestamp."

For example, if we wanted to remove the comment for the row identified by *Apache_Accumulo*, the Accumulo client library would insert a delete marker with the *Apache_Accumulo* row ID and the *comment* column, and that delete marker would be

assigned a timestamp representing the current time by the receiving tablet server. Subsequent reads of the *Apache_Accumulo* row would encounter the delete marker and know to skip any key-value pairs for that row and column appearing after the delete marker.

To add a *comment* field back into that row we would simply write a new key-value pair, which would get a newer timestamp than the delete marker, and so it would be returned by subsequent scans.

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

Applications that use time information typically store that time information as the value of a separate column rather than storing it in the timestamp portion of the key.

Advanced Data Model Components

Accumulo's data model includes additional components that help applications achieve better performance and data protection. These components are extensions to the basic concept of a column.

Columns are split into three components: a *column family*, a *column qualifier*, and a *column visibility*.

Most applications will start by simply assigning the names of fields to the column qualifier. Column families and column visibilities do not have to be populated. When developers have an idea for how data will be accessed, and for the sensitivity levels of various columns, these additional components can be used to help optimize and protect information.

Column Families

Often, applications find that they will access some columns together, and not other columns. Other times they need to access all of the columns within rows. This is especially prevalent in analytical applications.

When scanning for only a subset of the columns, it can be useful to change the way groups of columns are stored on disk so that frequently grouped columns are stored together, and so that columns containing large amounts of data that are not always scanned can be isolated.

For example, we might have some columns storing relatively small, structured data, and other columns storing larger values such as text or perhaps media such as imagery, audio, or video. In the Wikipedia table, the *text* column stores long text values. Sometimes our application may need to scan just the structured details about a user or multiple users and other times will need to scan the user details and the larger columns containing media content.

To cause related columns to be stored in consecutive key-value pairs in Accumulo, application designers can place these columns in the same *column family*. To apply this to our earlier example, we can choose to put the *text* and *comment* columns under a column family called *content* and the other columns under the *metadata* column family. If we retrieve the *metadata* column family, the tablet server can do less work to read just that one column family than if the individual metadata columns were scattered throughout each row, interleaved with content columns.

Unlike Bigtable and HBase, Accumulo column families need not be declared before being used, and Accumulo tables can have a very high number of column families if necessary.

Although grouping columns into families can make retrieving a single column family within one row more efficient, it can still be inefficient to read one column family across multiple rows, because we'll still have to scan over other column families before accessing the next row. For example, it would be inefficient if we always had to read the Wikipedia content off of disk when we are only interested in the user details.

To help avoid reading data unnecessarily from disk, application designers can choose to assign column families to a *locality group*. Locality groups are stored in separate contiguous chunks of data on disk so that an application that is only scanning over column families in one locality group doesn't need to read data from any other locality groups. This gives Accumulo more of a columnar-style storage that is amenable to many analytical access patterns.

Applying locality groups to our earlier example, we can choose to put the *content* column family in one locality group and the *metadata* column family in another locality group. Before we assigned column families to locality groups, a scan configured to read only the metadata columns would still end up reading the content columns off of disk (Figure 1-12), and tablet servers would filter them out, returning only the data requested.

rowid	column family	column qualifier	timestamp	value
Apache_Accumulo	content	comment	20111001	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	content	text	20120301	Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	metadata	pageid	20111001	571876229
Apache_Accumulo	metadata	title	20111001	Apache Accumulo
Apache_Hadoop	content	text	20060314	Apache Hadoop is an open source software framework for storage …
Apache_Hadoop	metadata	pageid	20060314	5919308
Apache_Hadoop	metadata	title	20060314	Apache Hadoop
Apache_Thrift	content	text	20070213	Thrift is an interface definition language and binary …
Apache_Thrift	metadata	pageid	20070213	10438451
Apache_Thrift	metadata	title	20070213	Apache Thrift
…	…	…	…	…

Figure 1-12. Reading over one column family still requires filtering out other column families

Once we assign the *content* column family to its own locality group, Accumulo will begin to store this textual content in a separate section on disk (Figure 1-13). Now when we read just the columns containing Wikipedia metadata, we don't have to read all of the text for each article off of disk.

Accumulo allows the assignment of column families to locality groups to change over time. New data written to Accumulo will always be written to disk according to the current assignment of column families to locality groups. Any data written prior to the change in assignment will need to be reprocessed before the benefit of the new locality groups is realized. Accumulo will reprocess data on disk automatically via a process called *compaction*, but compactions can also be forced as necessary. Using compactions to get previously written data to reflect changes in locality group assignments is described in "Locality Groups" on page 138.

Figure 1-13. Column families in different locality groups are stored together on disk

Column Visibility

Accumulo's focus on supporting analysis of data from several different sources has resulted in an additional component to the Bigtable data model called *column visibility*. The column visibility component is designed to logically isolate certain types of data based on sensitivity, by associating each value with a *security label expression*. This enables data to be protected from unauthorized access and for data sets of differing sensitivity to be stored in the same physical tables.

This feature is designed to reduce the amount of data movement that needs to occur when an organization decides that an application or an analytical process is allowed to look at two data sets. Imagine the case in which two data sets had to be stored in two physically separate systems for security reasons, called system A and system B. If one day an organization decides that it needs to join these data sets to answer an analytical question, the data from one system would have to be physically moved into the other system, say A into B, if there happens to be enough room. And the users of system B would have to be denied access to it while the data from system A resides there, if not all of them are also authorized to read data from system A. Or perhaps a third system will need to be stood up to handle the combination of this data, requiring that

new hardware be acquired, software installed, and the data from both system A and system B to be copied to the new system. That process could take months.

If the data is already all stored together physically, and protected with column visibilities, then granting access of a single analytical application to both data sets is trivial. While the analytical process is running, users authorized to read only one type of data or another can continue to submit queries against the system without ever seeing anything they aren't authorized to see.

In our example, we might decide that the data residing under the *comment* and *pageid* columns does not need to be exposed to applications that allow the public to read the article text and titles (Figure 1-14), and so we can decide to protect the data in these columns using the column visibility component of the key.

rowid	metadata		content	
	title	pageid	comment	text
Apache_Accumulo	Apache Accumulo	571876229	/* See also */ Added Column-oriented DBMS	Apache Accumulo is a sorted, distributed key/value store based on Google's BigTable. It is a system built on top of Apache Hadoop, Apache ZooKeeper, and Apache Thrift. …
Apache_Hadoop	Apache Hadoop	5919308		Apache Hadoop is an open source software framework for storage and large-scale processing of data-sets on clusters of commodity hardware. Hadoop is an Apache top-level project being built …
Apache_Thrift	Apache Thrift	10438451		Thrift is an interface definition language and binary communication protocol that is used to define and create services for numerous languages …
BigTable	BigTable	5919973		BigTable is a compressed, high performance, and proprietary data storage system built on Google File System, Chubby Lock Service, SSTable (log-structured storage like LevelDB) …

Figure 1-14. Two columns are deemed viewable by internal applications and users

The way we protect these values is by populating the column visibility components with *security label expressions*, sometimes called simply *security labels*. Security label expressions consist of one or more tokens combined by logical operators `&`, representing logical AND, and `|`, representing logical OR. Subexpressions can be grouped using parentheses.

In our simple example here, we are using just single-token expressions in our column visibility. On disk these key-value pairs now look like Figure 1-15.

rowid	column family	column qualifier	column visibility	timestamp	value
Apache_Accumulo	content	text	public	20120301	Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	metadata	comment	internal	20111001	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	metadata	pageid	internal	20111001	571876229
Apache_Accumulo	metadata	title	public	20111001	Apache Accumulo
Apache_Hadoop	content	text	public	20060314	Apache Hadoop is an open source software framework for storage ...
Apache_Hadoop	metadata	pageid	internal	20060314	5919308
Apache_Hadoop	metadata	title	public	20060314	Apache Hadoop
Apache_Thrift	content	text	public	20070213	Thrift is an interface definition language and binary ...
Apache_Thrift	metadata	pageid	internal	20070213	10438451
Apache_Thrift	metadata	title	public	20070213	Apache Thrift
...	...	...	...	...	...

Figure 1-15. Individual key-value pairs are labeled with column visibilities

Column visibilities are an extremely fine-grained form of access control. Sometimes the term *cell-level* is used when discussing Accumulo's ability to allow every value to have its own security label, which is stored in the column visibility element of the key. The term *cell-level* is used to contrast the granularity of Accumulo's security model with row-level or column-level security in which one can control access to all the data in a row or all the data in a column. It is not often the case that any one raw data set requires that each column of each row to have a different column visibility. Usually some combination of row-level or column-level access control will suffice, which column visibilities can support just as well.

But because a common application on Accumulo involves building secondary indexes, perhaps across several types of data of differing sensitivity levels, each key-value pair in an index will end up needing a specific column visibility based on the row and column from which it originated. Applications that use these types of indexes are very powerful because they allow different views of the data to be composed on the fly, according to the access level of the user performing the query.

For example, a user with only the *public* access token can scan this table and will only see the data with the *public* token in the column visibility portion of the key (Figure 1-16).

rowid	column family	column qualifier	column visibility	timestamp	value
Apache_Accumulo	content	text	public	20120301	Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	metadata	title	public	20111001	Apache Accumulo
Apache_Hadoop	content	text	public	20060314	Apache Hadoop is an open source software framework for storage ...
Apache_Hadoop	metadata	title	public	20060314	Apache Hadoop
Apache_Thrift	content	text	public	20070213	Thrift is an interface definition language and binary ...
Apache_Thrift	metadata	title	public	20070213	Apache Thrift
...	...	...	...	...	...

Figure 1-16. View of only public data in the table

A user with both the *public* and *internal* access tokens will see all of the data in the table when doing a scan (Figure 1-17).

rowid	column family	column qualifier	column visibility	timestamp	value
Apache_Accumulo	content	text	public	20120301	Accumulo is a sorted, distributed key/value store ..
Apache_Accumulo	metadata	comment	internal	20111001	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	metadata	pageid	internal	20111001	571876229
Apache_Accumulo	metadata	title	public	20111001	Apache Accumulo
Apache_Hadoop	content	text	public	20060314	Apache Hadoop is an open source software framework for storage ...
Apache_Hadoop	metadata	pageid	internal	20060314	5919308
Apache_Hadoop	metadata	title	public	20060314	Apache Hadoop
Apache_Thrift	content	text	public	20070213	Thrift is an interface definition language and binary ...
Apache_Thrift	metadata	pageid	internal	20070213	10438451
Apache_Thrift	metadata	title	public	20070213	Apache Thrift
...	...	...	...	...	...

Figure 1-17. View of all of the data in the table

A user or application with only the *internal* access token will only see the data with a column visibility containing the *internal* token (Figure 1-18).

rowid	column family	column qualifier	column visibility	timestamp	value
Apache_Accumulo	metadata	comment	internal	20111001	/* See also */ Added Column-oriented DBMS
Apache_Accumulo	metadata	pageid	internal	20111001	571876229
Apache_Hadoop	metadata	pageid	internal	20060314	5919308
Apache_Thrift	metadata	pageid	internal	20070213	10438451
...	...	...	...	...	...

Figure 1-18. View of only internal data in the table

Because column visibilities are used to filter data after specific rows and columns have been selected for a scan, table designers should be careful not to design an application that relies too heavily on filtering, because this will impact read performance.

The assignment of access tokens to applications, individual users, or groups of users is typically handled outside of Accumulo by a central user-management system, although access tokens can be restricted in conjunction with Accumulo or using only Accumulo if desired.

We discuss using column visibilities in designing applications in depth in "Column Visibilities" on page 184.

Full Data Model

Now that we've discussed all of the components of the Accumulo data model we can show the full model containing all components of the key, with the components of the column broken out (Figure 1-19).

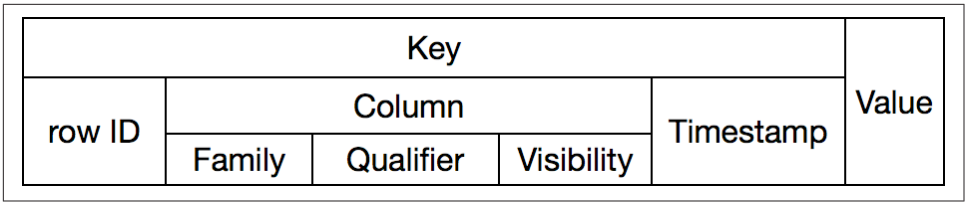

Figure 1-19. Accumulo key structure

Not all of the components must be used. At the very least, you can choose to use only the row ID and value portions of the key-value pair. In this case Accumulo will operate like a simple key-value store. Many applications start with rows and columns, and apply the use of additional components as designs are optimized.

Developers should consider carefully the components of the key that their application requires when designing tables.

Tables

When stored in Accumulo, key-value pairs are grouped into *tables*. You can apply some settings at the table level to control the behavior and management of the data. The key-value pairs within tables are partitioned into *tablets* and distributed automatically across multiple machines in a cluster.

Each table begins life as a single tablet, spanning all possible keys. Once data is written to a table and it reaches a certain size threshold, the tablet server hosting it finds a good point in the middle of the tablet and splits it into two tablets.

When a tablet server does this it always splits a tablet on a row boundary, guaranteeing that the data for each row is fully contained within one tablet and therefore resides on exactly one server. This is important to allowing consistent updates to be applied atomically to the data in an individual row.

For example, as our Wikipedia table grows, it will eventually be split along a row boundary into two tablets (Figure 1-20).

Figure 1-20. Splitting a tablet into tablets

Accumulo takes care of distributing responsibility for tablets evenly across tablet servers. A single tablet server can host several hundred tablets or more simultaneously.

We discuss the splitting process more in depth in "Splits" on page 365.

Introduction to the Client API

Accumulo provides application developers with a client library that is used to locate and communicate with tablet servers for writing data, and reading one or more key-value pairs.

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

As you would expect from a key-value store, clients can provide a key and look up the associated value, if it exists. Instead of returning one value, however, clients can opt to scan a range of key-value pairs beginning at a particular key. The performance differ-

ence between looking up and retrieving a single value versus scanning, say, a few hundred kilobytes of key-value pairs is fairly small, because the cost of reading that amount of data sequentially is dominated by the disk seek time.

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

The ability to discover new information via scanning is valuable for applications that want to combine information about similar subjects from different sources that may not contain the same information about each subject.

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

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

`BatchWriter`
: All new inserts, updates, and deletes are packaged up into `Mutation` objects and given to a `BatchWriter`. A `Mutation` object contains a set of changes to be applied to a single row. The batch writer knows how the table is split into tablets and which servers the tablets are assigned to. Using this information, the batch writer efficiently groups `Mutation` objects into batches to increase write throughput. Batch writers send batches of `Mutation` objects to various tablet servers. The batch writer is multithreaded, and the trade-off between latency and throughput can be tuned. See Figure 1-21.

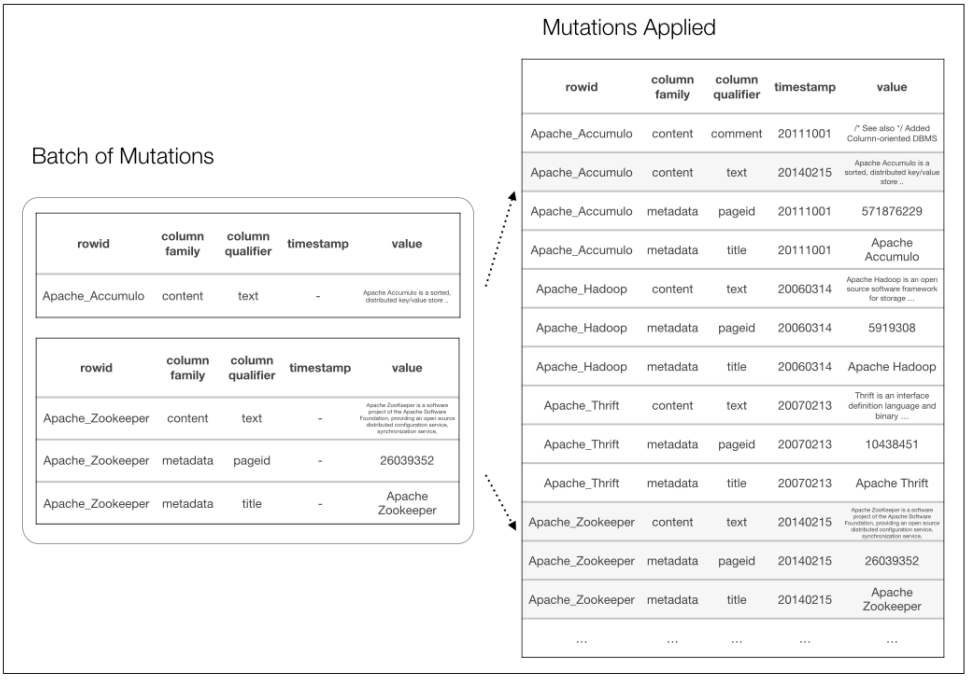

Figure 1-21. Writing mutations

`Scanner`

: Key-value pairs are read out of a table using a `Scanner` object. A scanner can start at the beginning of a table or at a particular key, and can stop at the end of the table or a given key. After seeking to the initial key, scanners proceed to read out key-value pairs sequentially in key order until reaching the end of the table or the specified ending key. Scanners can be configured to read only certain columns. Additional configuration for a scanner can be made to apply additional logic classes called *iterators*, and specific options to iterators, to alter the set of key-value pairs returned from a particular scanner. See Figure 1-22.

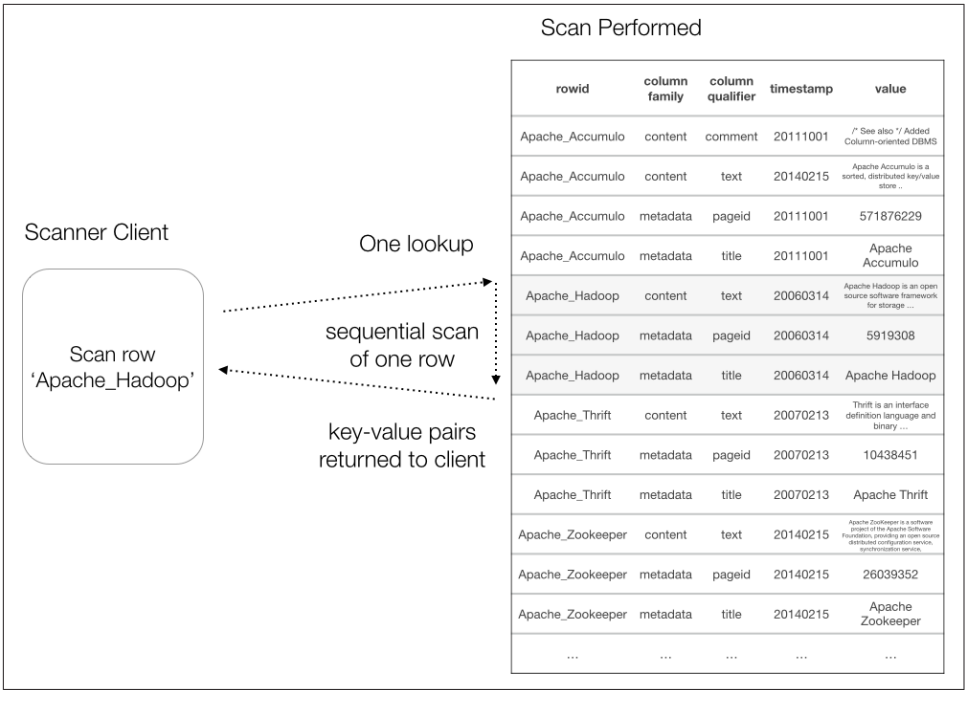

Figure 1-22. Scanning one row

`BatchScanner`

When multiple ranges of keys are to be read from a table, a `BatchScanner` can be used to read the key-value pairs for the ranges using multiple threads. The ranges are grouped by tablet server to maximize the efficiency of communication between threads and tablet servers. This can be useful for applications whose design requires many individual scans to answer a single question. In particular, tables designed for working with time series, secondary indexes, and complex text search can all benefit from using batch scanners. See Figure 1-23.

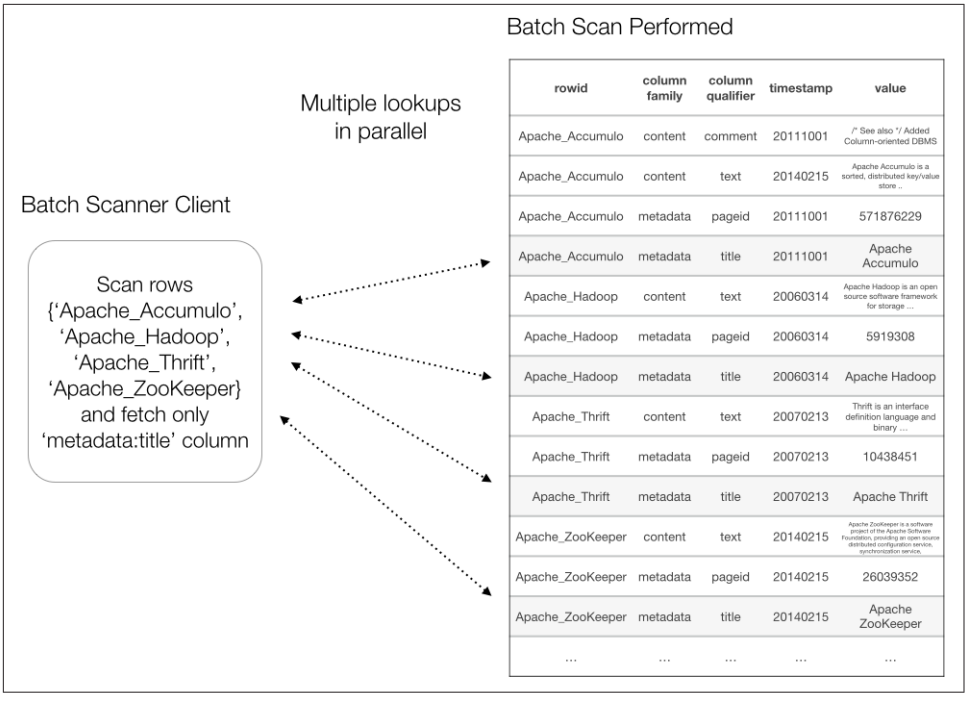

Figure 1-23. Scanning a batch of rows

More detail on developing applications using Accumulo's API is found in the chapters beginning with Chapter 3.

Approach to Rows

Accumulo takes a slightly different approach to rows in the client API than do some other implementations based on Bigtable, such as HBase. Accumulo's read API is designed to stream key-value pairs to the client rather than to package up all the key-value pairs for a row into a single unit before returning the data to the user.

This is often less convenient than working with data on a row-by-row basis, and applications that want to work with entire rows can do additional configuration to assist with this, as described in "Grouping by Rows" on page 110. The upside is that rows in an Accumulo table can be very large and do not need to fit in the memory of the tablet server or the client. Working with key-value pairs can come in handy when row IDs are coming from external data and the number of columns per row may be unknown or may vary widely, as can happen when building secondary indexes.

Exploiting Sort Order

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

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

This sometimes requires creative thinking in order to discover a key design that works for a particular application. A good example of this is the way Google describes the row ID of its WebCrawl table in the Bigtable paper. In this table, the intent is to provide users with the ability to look up information about a given website, identified by the hostname. Because hostnames are hierarchical and because users may want to look at a specific hostname or all hostnames within a domain, Google chose to transform the hostname to support these access patterns by reversing the order in which domain name components are stored under the row ID, as shown in Table 1-1.

Table 1-1. Google's WebCrawl row design

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

com.yahoo.search

com.yahoo.www

Achieving optimal performance also depends on the ability to satisfy user requests without having to filter out or ignore a large amount of key-value pairs as a part of the scan.

Because developers have such a high degree of control over how data is arranged, there are a wide variety of options for designing tables. We cover these in depth in Chapter 8.

Architecture Overview

Accumulo is a distributed application that depends on Apache Hadoop for storage and Apache ZooKeeper for configuration (Figure 1-24).

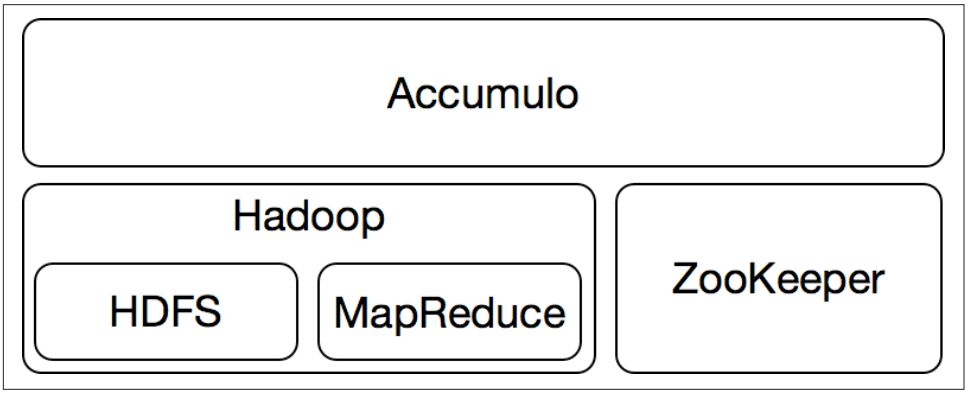

Figure 1-24. Accumulo architecture

Because Accumulo is based on Google's Bigtable, as HBase is, it uses some of the same names for components that Bigtable does, but there are some differences (Table 1-2).

Table 1-2. Accumulo and HBase Bigtable naming conventions

Apache Accumulo	Bigtable	Apache HBase
Tablet	Tablet	Region
Tablet Server	Tablet Server	Region Server
Minor Compaction	Minor Compaction	Flush

Major Compaction	Merging Compaction	Minor Compaction
(Full) Major Compaction	Major Compaction	Major Compaction
Write-Ahead Log	Commit Log	Write-Ahead Log
HDFS	GFS	HDFS
Hadoop MapReduce	MapReduce	Hadoop MapReduce
MemTable	MemTable	MemStore
RFile	SSTable	HFile
ZooKeeper	Chubby	ZooKeeper

ZooKeeper

ZooKeeper is a highly available, highly consistent, distributed application in which all data is replicated on all machines in a cluster so that if one machine fails, clients reading from ZooKeeper can quickly switch over to one of the remaining machines. ZooKeeper plays the role for Accumulo of a centralized directory and lock service that Google's Chubby provides for Bigtable. In addition, write replication is synchronous, which means clients wait until data is replicated and confirmed on all machines before considering a write successful. In practice, ZooKeeper instances tend to consist of three or five machines.

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

Hadoop

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

Like Accumulo, HDFS is a distributed application, but one that allows users to view a collection of machines as a single, scalable filesystem. HDFS files can be very large, up to terabytes per file. HDFS automatically breaks these files into blocks—by default 64 MB or 128 MB in size depending on the version of HDFS—and distributes these blocks across the cluster uniformly. In addition, each block is replicated on multiple

machines (Figure 1-25). The default replication factor is three in order to avoid losing data when one machine or even an entire rack of servers becomes unavailable. Usually, one replica is written to the local hard drive, another to another machine in the same rack, and a third to a machine in another rack. This way, even the loss of an entire rack won't cause data loss.

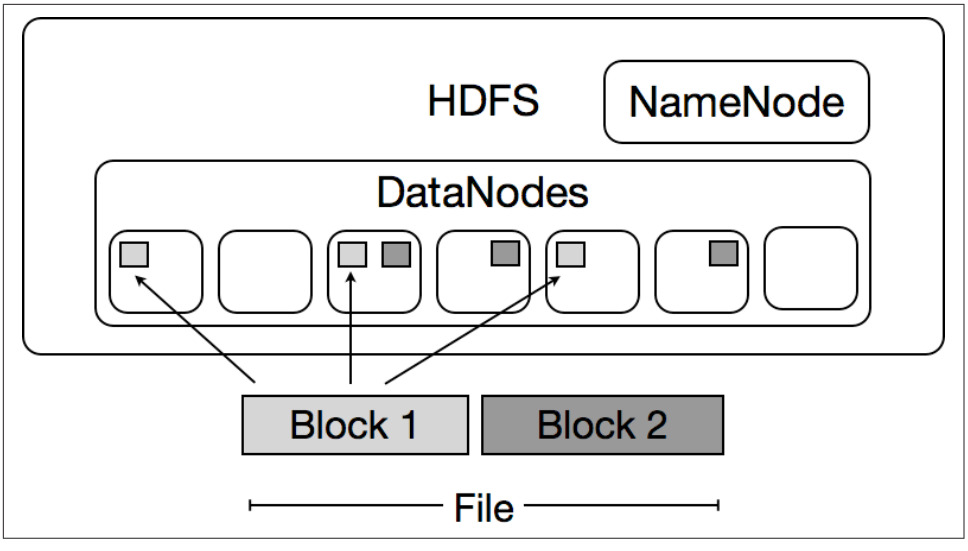

Figure 1-25. Hadoop Distributed File System

Accumulo

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

Analogous to HDFS files, Accumulo tables can be very large in size, up to tens of trillions of key-value pairs or more. Accumulo automatically partitions these into tablets and assigns responsibility for hosting tablets to servers called *tablet servers* (Figure 1-26).

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

Typically, a server will run one tablet server process and one HDFS DataNode process (Figure 1-27). This allows most tablets to have a local replica of the files they reference.

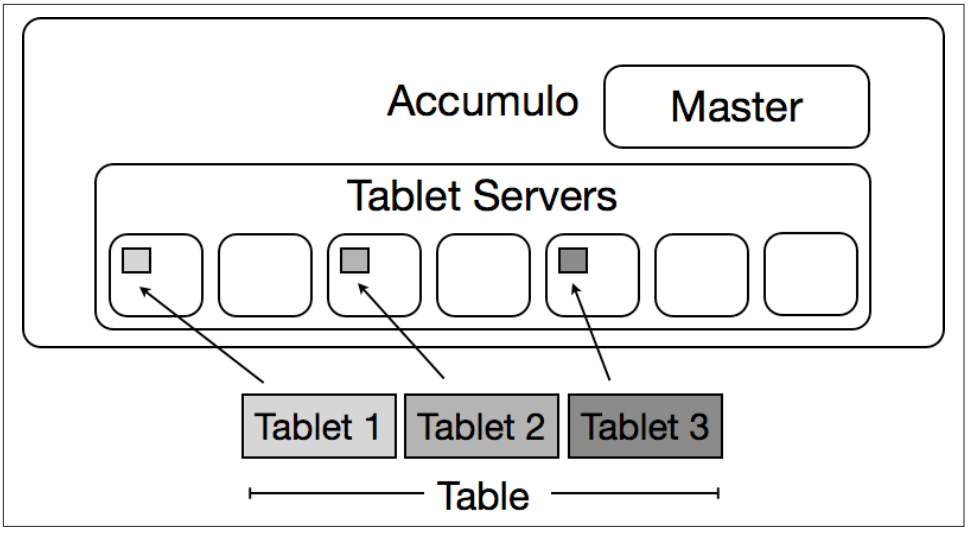

Figure 1-26. Accumulo

As a result, a tablet server can host a tablet whose file replicas are all located on other servers. This situation does not prevent the tablet's data from being read and is usually temporary, because any time a tablet server performs compaction of a tablet's files, it will by default create one local replica of each new file. Over time, a tablet tends to have one local replica for each file it references.

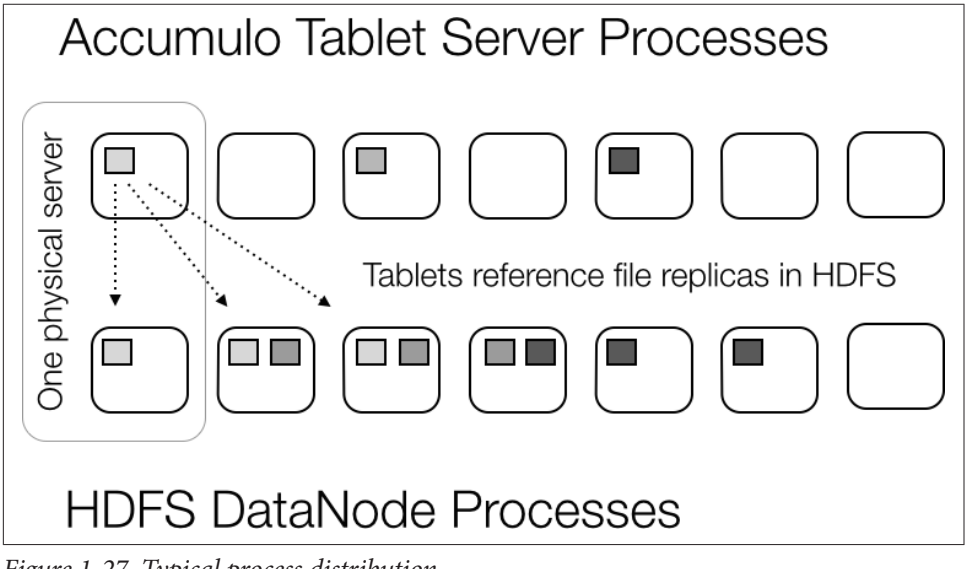

Figure 1-27. Typical process distribution

Tablet servers

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

Master

Every Accumulo cluster has one active *master* process that is responsible for making sure all tablets are assigned to exactly one tablet server at all times and that tablets are load-balanced across servers. The master also helps with certain administrative operations such as startup, shutdown, and table and user creation and deletion.

Accumulo's master can fail without causing interruption to tablet servers and clients. If a tablet server fails while the master is down, some portion of the tablets will be unavailable until a new master process is started on any machine. When the new master process starts, it will reassign any tablets that do not have a tablet server assignment.

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

Garbage collector

The garbage collector process finds files that are no longer being used by any tablets and deletes them from HDFS to reclaim disk space.

A cluster needs only one garbage collector process running at any given time.

Monitor

Accumulo ships with an informative monitor that reports cluster activity and logging information into one web interface (Figure 1-28). This monitor is useful for verifying that Accumulo is operating properly and for helping understand and troubleshoot cluster and application performance. "Monitor Web Service" on page 429 gives descriptions of the information displayed by the monitor.

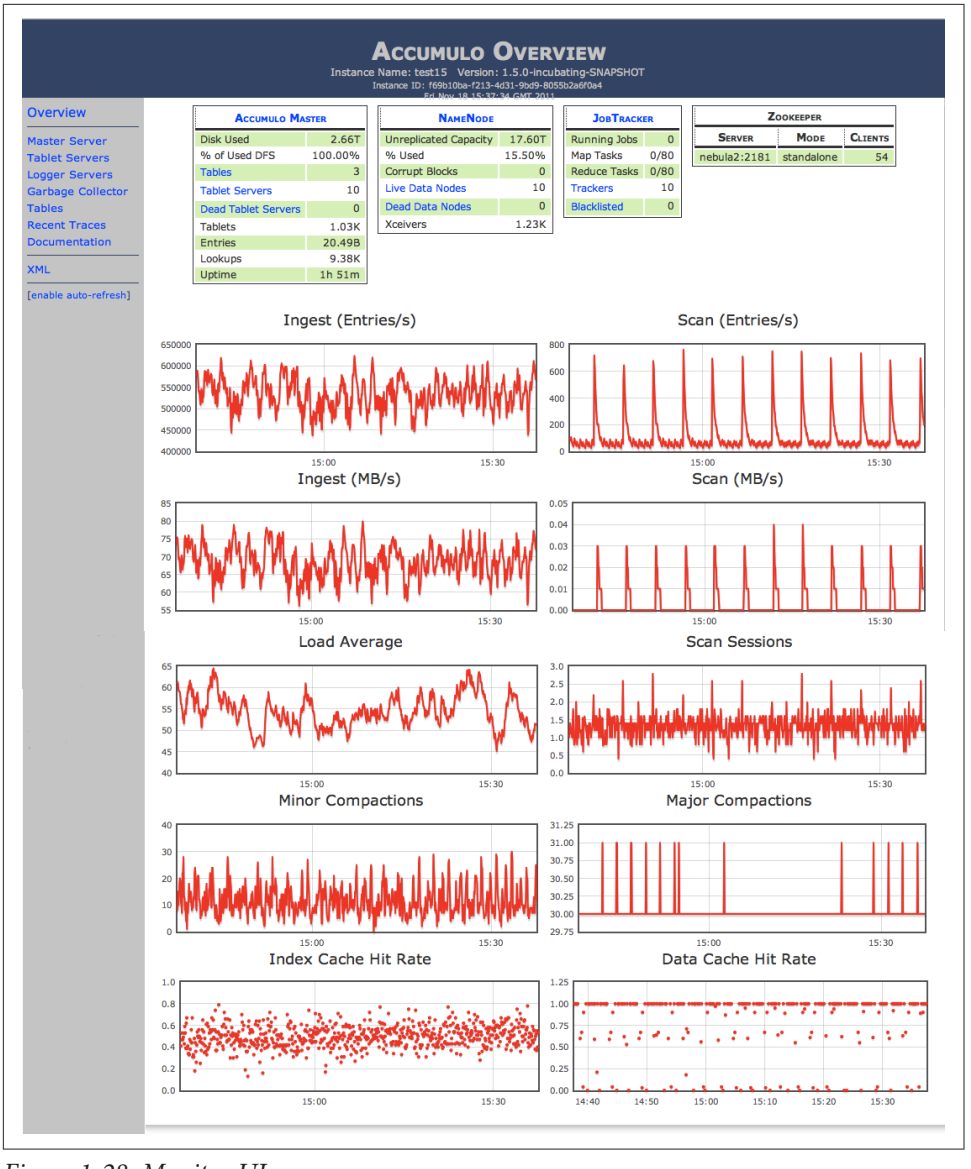

Figure 1-28. Monitor UI

Client

Accumulo provides a Java client library for use in applications. Many Accumulo clients can read and write data from an Accumulo instance simultaneously. Clients communicate directly with tablet servers to read and write data (Figure 1-29). Occasionally, clients will communicate with ZooKeeper and with the Accumulo master for

certain table operations, but no data is sent or received through ZooKeeper or the master.

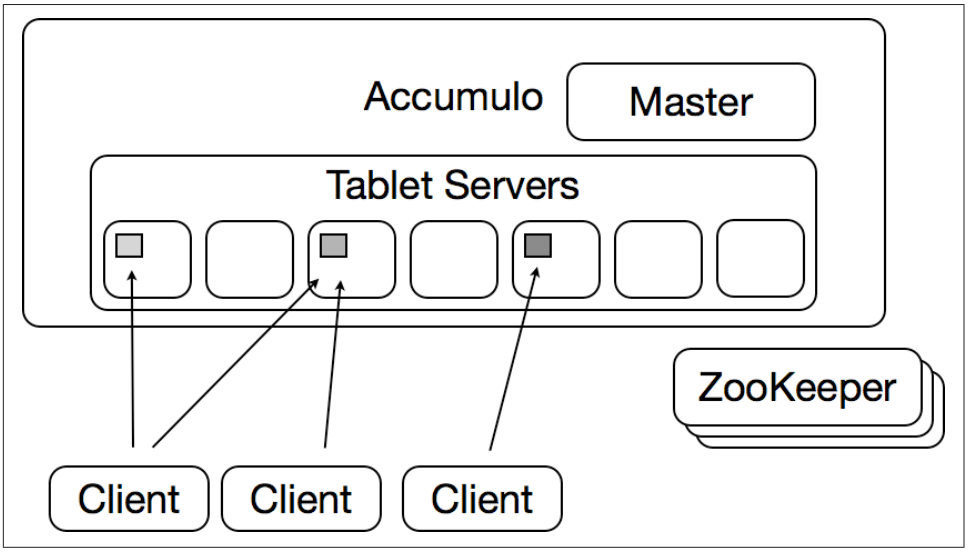

Figure 1-29. Accumulo clients

Thrift proxy

As of version 1.5, Accumulo provides an optional Thrift proxy that can be used to develop clients in languages other than those that run on the Java Virtual Machine (JVM). These other clients can connect to the Thrift proxy, which communicates with the Accumulo cluster and allows data to be read and written by these other clients.

Accumulo versions 1.4 and older use logger processes to record each new write in an unsorted write-ahead log on disk that can be used to recover any data that was lost from the memory of a failed tablet server. Accumulo 1.5 no longer has dedicated logger processes. The write-ahead logs are written directly to files in HDFS.

A Typical Cluster

A typical Accumulo cluster consists of a few *control nodes* and a few to many *worker nodes* (Figure 1-30).

Control nodes include:

- One, three, or five machines running ZooKeeper
- Ideally, two machines running HDFS NameNode processes, one active, one for failover
- One to two machines running Accumulo master, garbage collector, and/or monitor
- For Hadoop 1, an optional machine running a Hadoop job tracker process if MapReduce jobs are required
- For Hadoop 2, an optional machine running a YARN resource manager process if MapReduce jobs are required

Each worker node typically includes:

- One HDFS DataNode process for storing data
- One tablet server process for serving queries and inserts
- For Hadoop 1, an optional Hadoop task tracker for running MapReduce jobs
- For Hadoop 2, an optional YARN node manager for running MapReduce jobs

The logger process mentioned in Accumulo versions 1.4 and earlier would have typically run on each worker node.

In addition, applications require one to many processes using the Accumulo client library to write and read data.

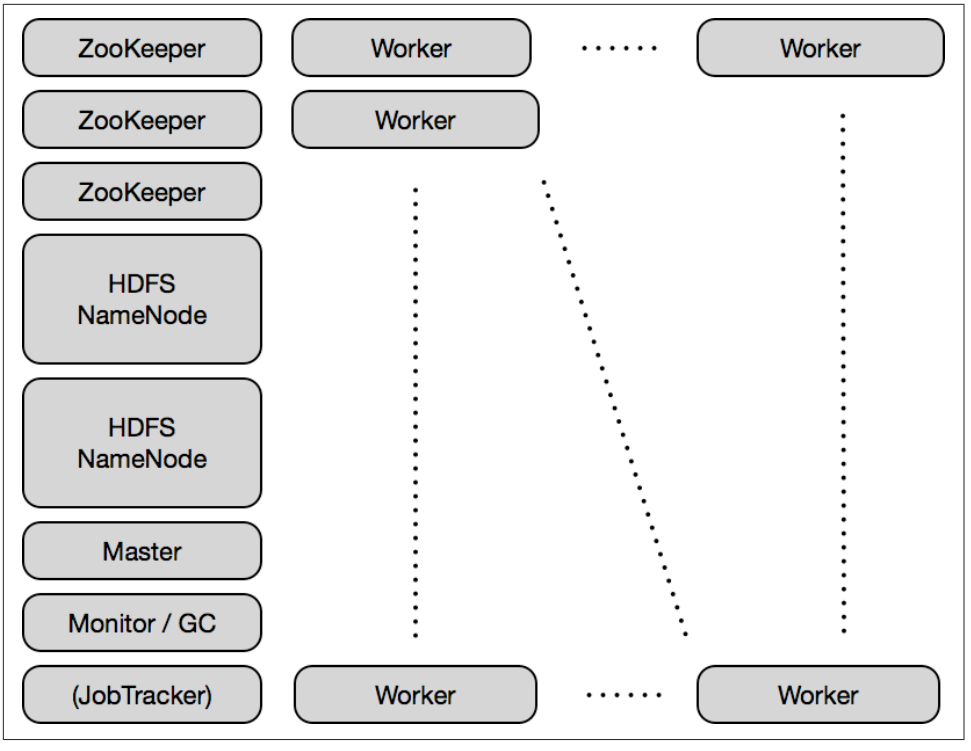

Figure 1-30. A typical cluster

Additional Features

In addition to the features already described, Accumulo provides more features to help you build scalable applications running on large clusters. Not all of these are unique to Accumulo, but the combination of these features is likely unique.

Automatic Data Partitioning

Accumulo tables can be very large, up to petabytes in size. You can tune the tablet-splitting process, but you don't have to worry about choosing a good key on which to partition because Accumulo automatically finds good split points.

High Consistency

Accumulo provides a highly consistent view of the data. Tablets are assigned to exactly one tablet server at a time. An update to a particular key's value is immediately reflected in subsequent reads because those updates and reads go to the same server.

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

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

Automatic Load Balancing

The Accumulo master automatically balances the responsibility for serving tablets across tablet servers. When one tablet server has more tablets than another, the master process will instruct the overloaded tablet server to stop serving a tablet and instruct the underloaded tablet server to begin hosting that tablet.

Massive Scalability

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

Accumulo can also work well at large scale, meaning on clusters consisting of thousands of machines hosting petabytes of data.

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

Many distributed systems today are built to scale from one server to many. Accumulo may be one of the most scalable data stores out there. As of version 1.6, Accumulo is capable of running across multiple instances of HDFS with different HDFS NameNodes. This means that Accumulo can be configured to support more update operations than can be accommodated by a single HDFS instance.

Failure Tolerance and Automatic Recovery

Like Hadoop, Accumulo is designed to survive single server failures and even the failure of a single rack. If a single Accumulo tablet server fails, the master process notes

this and reassigns its tablets to the remaining tablet servers. Accumulo clients automatically manage the failover from one tablet server to another. Application developers do not need to worry about retrying their operations simply because a machine fails.

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

Support for Analysis: Iterators

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

One of these mechanisms, Accumulo *iterators*, enable custom aggregation and summarization within tablet servers to allow you to maintain result sets efficiently and store the data at a higher level of abstraction. They are called iterators because they iterate over key-value pairs and allow developers to alter the data before writing to disk or returning information to users.

There are various types of iterators that range from filtering to simple sums to maintaining a set of statistics. These are covered in "Iterators" on page 209.

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

Support for Analysis: MapReduce Integration

Beyond iterators, Accumulo supports analysis via integration with the popular Hadoop MapReduce framework. Accumulo stores its data in HDFS and can be used as the source of data for a MapReduce job or as the destination of the output from a MapReduce job. MapReduce jobs can either read from tablet servers using the Accumulo client library, or from the underlying files in which Accumulo stores data via the use of specific MapReduce input and output formats.

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

We cover using MapReduce with Accumulo in depth in Chapter 7.

Data Lifecycle Management

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

In addition, the timestamps that are part Accumulo's key structure can be used with iterators to age data off according to a policy set by the administrator. This includes aging off data older than a certain amount of time from now, or simply aging off data older than a specific date.

Timestamps can also be used to distinguish among two or more versions of otherwise identical keys. The built-in `VersioningIterator` can be configured to allow any number of versions, or only a specific number of versions, to be stored. Google's original Bigtable paper describes using timestamps to distinguish among various versions of the Web as it was crawled and stored from time to time.

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

We cover age-off in depth in "Data Age-off" on page 450.

Compression

Accumulo compresses data by default using several methods. One is to apply a compression algorithm such as GZip or LZO to blocks of data stored on disk. The other is a technique called *relative-key encoding*, in which the shared prefixes of a set of keys are stored only once, and the following keys only need express the changes to the initial key.

Compressing data in this way can improve I/O, because reading compressed data and doing decompression can be faster than reading uncompressed data and not doing decompression. Compression also helps offset the cost of the block replication that is performed by HDFS.

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

Robust Timestamps

When Accumulo tablet servers are assigning timestamps to key-value pairs, Accumulo ensures that the timestamps are internally consistent. Accumulo only assigns new timestamps that are later than the most recent timestamp for a given tablet. In other words, timestamps assigned by a tablet server are guaranteed to increase.

This addresses the inevitable situation in which some servers in the cluster have clocks that are off and are applying timestamps from the future to keys. If these keys were transferred to another server, newly written data would be treated as older than existing data. It would be very confusing for users not to see the data they expect. It would be an even more critical problem in the Accumulo metadata that keeps track of tablets and their files. Entire data files could be lost if this problem were allowed to occur. Thus, Accumulo only assigns new timestamps that are later than the most recent timestamp for a given tablet.

It is also possible to use a one-up counter for timestamps by configuring a table with a time type of *logical* instead of the default time type of *milliseconds* since the UNIX epoch (Midnight UTC on January 1, 1970). In either case, tablet servers ensure that a newly written key-value pair is never stamped with a timestamp that precedes the most recent timestamp for the key's tablet. This does not, however, prevent arbitrary user-assigned timestamps from being written to a table.

Accumulo and Other Data Management Systems

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

Comparisons to Relational Databases

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

SQL

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

One challenge of using SQL is that of performing this work at query time on a large amount of data. Relational Massively Parallel Processing (MPP) databases approach this by dividing the work to perform SQL operations among many servers. The approach taken by Accumulo is to encourage aggressive precomputation where possible, often using far more storage to achieve the space-time trade-off, in order to

minimize the work done at query time and maintain fast lookups even when storing petabytes of data.

Space-Time Trade-off and Cheap Space

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

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

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

Transactions

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

ACID

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

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

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

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

In relational databases these properties are delivered via several mechanisms. One such mechanism is a *transaction*, which bundles a set of operations together into a

logical unit. Transactions are important for supporting *operational workloads* such as maintaining information about inventory, keeping bank accounts in order, and tracking the current state of business operations. Transactions can contain changes to multiple values within a row, changes to values in two or more rows in the same table, or even updates to multiple rows in multiple tables. These types of workloads are considered online transaction processing (OLTP).

Accumulo guarantees these ACID properties for a single mutation (a set of changes for a single row) but does not provide support for atomic updates across multiple rows. Nor does Accumulo maintain consistent references between rows. Row isolation for reads can be obtained by enabling the feature for a particular scanner (see "Isolated Row Views" on page 111).

Normalization

If you store multiple copies of the same data in different places, it can be difficult to ensure a high degree of consistency. You might update the value in one place but not the other. Therefore, storing copies of the same values should be avoided.

Values that don't have a one-to-one relationship to each other are often divided into separate tables that keep pointers between themselves. For example, a person typically only has one birth date, so you can store birth date in the same table as first name and other one-to-one data (Figure 1-31).

But a person may have many nicknames or favorite songs. This type of one-to-many data is stored in a separate table (Figure 1-32). There is a well-defined process, called *normalization*, for deciding which data elements to put into separate tables. There are several degrees to which normalization can be applied, but it typically involves breaking out data involved in one-to-many or many-to-many relationships into multiple tables and joining them at query time.

Another group of workloads is termed online analytical processing (OLAP). Relational databases have been used to support these kinds of workloads as well. Often analysis takes the approach of looking at snapshots of operational data, or simply may bring together reference data that doesn't require updates but requires efficient read and aggregation capabilities. Because these data snapshots are no longer updated, there is no opportunity for the data to become inconsistent, and the need for normalization is diminished.

Because OLAP workloads require fewer updates, tables are often precombined, or *denormalized*, to cut down on the operations that are carried out a query time (Figure 1-33). This is another example of the space-time trade-off, whereby an increase in storage space used reduces the time to get data in the format requested.

ID	Name	Date of Birth
1001	Bob Jones	1978-04-01
1002	Fred Smith	1965-11-02
1003	Wei Chang	1983-12-06
1004	David Garcia	1976-09-09

Figure 1-31. A table containing a one-to-one relationship

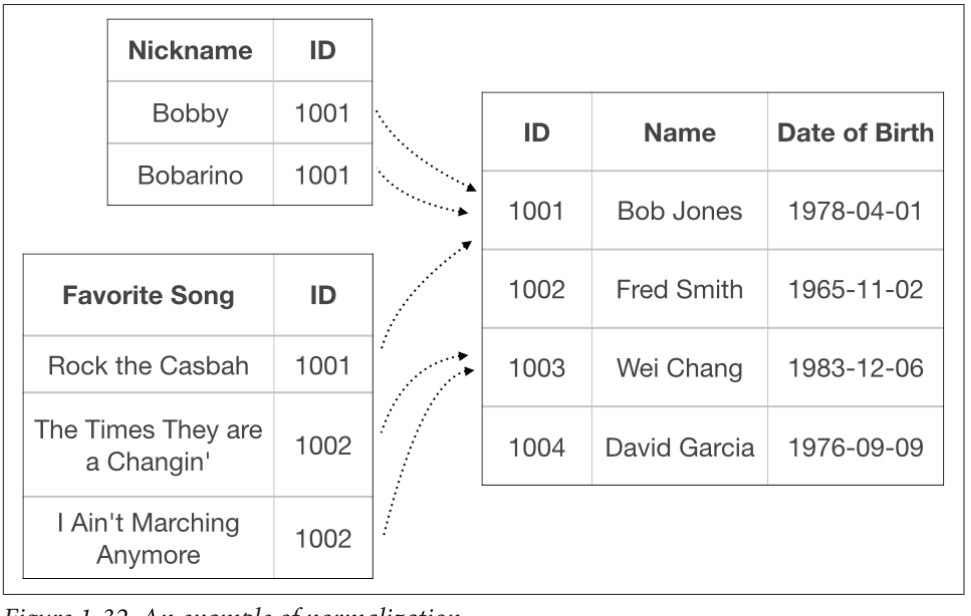

Figure 1-32. An example of normalization

ID	Name	Date of Birth	Nickname	Favorite Song
1001	Bob Jones	1978-04-01	Bobby	Rock the Casbah
1001	Bob Jones	1978-04-01	Bobarino	Rock the Casbah
1002	Fred Smith	1965-11-02		
1003	Wei Chang	1983-12-06		The Times They are a Changin'
1003	Wei Chang	1983-12-06		I Ain't Marching Anymore
1004	David Garcia	1976-09-09		

Figure 1-33. An example of denormalization

In the example in Figure 1-33 of denormalizing data for analysis, it is easy to see how you would want a system like Accumulo that is highly scalable, employs compression of redundant data, and handles sparse data well.

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

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

Comparisons to Other NoSQL Databases

Accumulo belongs to a group of applications known as *NoSQL databases*. The term *NoSQL* refers to the fact that these databases support data access methods other than SQL and is short for Not SQL or Not Only SQL—although the engineer who coined the term NoSQL, Carlo Strozzi, has expressed that it may be more appropriate to call these applications nonrelational databases.[2]

2 "NoSQL Relational Database Management System: Home Page." Strozzi.it. 2 October 2007. Retrieved 29 March 2010.

Rather than using SQL for creating queries to fetch data and perform aggregation, Accumulo provides a simplified API for writing and reading data. Departing from the relational model and SQL has two major implications: increased flexibility in how data is modeled and stored, and the fact that some operations that other databases perform at query time are instead applied when data is written. In other words, results are precomputed so that query-time operations can consist solely of simple, fast tasks.

Compared to other NoSQL databases, Accumulo has some features that make it especially dynamic and scalable.

Data model

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

Some folks have grouped distributed software systems into categories based on the data model supported. These categories can consist of the following:

Pure key-value

- Riak
- Dynamo
- memcached

Columnar (Bigtable)

- Bigtable
- Accumulo
- HBase
- Cassandra

Document

- MongoDB
- CouchDB

Graph

- Neo4j

Some of these applications have in common a key-value data model at a high level. Accumulo's data model consists of key-value pairs at the highest level, but because of the structure of the key it achieves some properties of conventional two-dimensional,

flat-record tables, columnar and row-oriented databases, and a little bit of hierarchy in the data model via column families and column qualifiers.

Apache Accumulo, Apache Cassandra, and Apache HBase share this basic Bigtable data model.

Other NoSQL data stores, such as MongoDB and CouchDB, are considered to be document-oriented stores because they store JavaScript Object Notation (JSON)–like documents natively.

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

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

Key ordering

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

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

Tight Hadoop integration

Many NoSQL databases have their own storage mechanism. Accumulo uses HDFS. This offers several advantages:

- Accumulo can use the output of MapReduce jobs without having to move large amounts of data. Accumulo can also serve as the source of input data for MapReduce jobs. This allows Hadoop clusters to be used for mixed workloads.
- Accumulo benefits from the significant work done by the Hadoop community to make HDFS resilient, scalable, and stable.
- Because Hadoop is becoming the de-facto standard for large data processing in many organizations, Accumulo reduces the cost of acquiring a scalable, low-latency query capability by building on existing investment in Hadoop.

High versus eventual consistency

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

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

When comparing NoSQL databases, you may want to consider which trade-offs have been made in the design. In particular, much attention has been paid to the CAP theorem, which states that in designing a distributed database, you can choose to provide at most two of the following properties: high Consistency, Availability, and Partition-tolerance (hence CAP). A good treatment of this concept is in "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services" by Seth Gilbert and Nancy Lynch.

See "Accumulo and the CAP Theorem" on page 379 for a discussion on the choices made in the Accumulo design with respect to the CAP Theorem.

Column visibility and access control

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

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

HBase in particular has implemented Accumulo's column visibilities—including the same types of security label expressions as Accumulo as well as a different mode of access involving attaching access-control lists (ACLs) to cells.

One important note is that HBase includes a `NOT` operator (!) that can make it impossible to allow users to view the data using a subset of all their tokens, because they could remove a token used as part of a `NOT` expression to protect data. See the Accumulo mailing list for the thread "'`NOT`' operator in visibility string."

For example, suppose there were multiple cells with the following labels:

```
kvpair1: private
kvpair2: ( private | admin ) & !probationary
kvpair3: admin
```

To query Accumulo's key-value pairs, the user must always provide a list of authorization tokens to use for the query. Accumulo's built-in `ColumnVisibilityFilter` determines whether a particular set of tokens is sufficient to view a particular key-value

pair. Each user has a maximum set of tokens he is allowed to use for queries. It is not uncommon for applications developed on Accumulo to allow users to issue queries with a subset of their allowed tokens in order to see data as it would be viewed at different visibility levels. For example, a user with both the *private* and *admin* tokens might choose to query the data with just the *private* token. This helps with publishing data to other groups of users that are granted different authorization tokens.

In the presence of the `NOT` operator, applications cannot allow users to view the data with any fewer than all of their tokens, because removing a token from a query would *increase* the number of key-value pairs visible to the user, amounting to an elevation of privilege. In the preceding example, imagine issuing a query with the *private* and *probationary* tokens versus a query with just the *private* token.

Another important note is that HBase does not consider the security label expression to be a part of the key portion of the data model, as Accumulo does. This implies a model in which a key-value pair at one visibility level can be overwritten with a different visibility level. In Accumulo's visibility model, you can store multiple values at different visibility levels for the same row and column, because the visibility is considered part of the key. It is not possible to overwrite one visibility with another less restrictive visibility.

HBase's implementation is also a bit different from Accumulo's in that it utilizes coprocessors since HBase doesn't have a construct like Accumulo iterators. There may be performance differences as a result.

MongoDB has recently added a feature called *redact* as part of its Aggregation Framework that can be used to filter out subdocuments based on a flexible set of expressions. It appears likely that Accumulo's filtering logic could also be implemented in this framework.

Iterators

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

Iterators are applied at scan time, when flushing memory to disk (minor compaction), and when combining files (major compaction).

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

Dynamic column families and locality groups

As mentioned in "Column Families" on page 19, Accumulo can have any number of column families, and column families can be assigned arbitrarily to locality groups. Accumulo does not require column families to be declared before they can be used. Accumulo stores key-value pairs together on disk according to how their column families are mapped to locality groups within a single file, rather than using separate files or directories to separate the data, which keeps file management overhead constant. Furthermore, changes can be made to how the data is stored on disk by reconfiguring locality groups on the fly, without changing how data is modeled in the Accumulo key.

In contrast, HBase requires column families to be declared beforehand, and each column family is stored in a separate directory in HDFS, which drastically limits the flexibility of column family usage. Because column families are mapped to HDFS directories in HBase, they must consist of printable characters, whereas in Accumulo they are arbitrary byte arrays. Because every column family is a separate storage directory in HBase, in practice it is recommended that tables have fewer than 10 column families total (see Lars George's *HBase: The Definitive Guide* [O'Reilly]). Each column family in HBase is effectively in its own locality group, and multiple families cannot be grouped together.

File handle resources are limited per server, and the overall number of files and directories in HDFS is limited by the capacity of the NameNode, so having the number of files be dependent on your specific data model rather than on the overall amount of data becomes a consequence that every application must consider. Accumulo application designers do not have to consider this problem because Accumulo does not have this limitation.

HBase requires that at least one column family be declared per table, and every key-value pair inserted must specify a column family, whereas Accumulo does not require the column family portion of the key to be filled out. It can be left blank, even if column qualifiers or other parts of the key are filled out.

Support for very large rows

Accumulo does not assume that rows must fit entirely in memory. Key-value pairs are streamed back to the client in batches, and it's possible for the client to fetch a portion of a row first and to stream the rest of the row in separate batches.

An example of an application design that may require arbitrarily large rows is in the use of tables to store secondary indexes for document search, where the row ID is used to store search terms that may be mapped to many document IDs stored in column qualifiers. The row corresponding to a common search term would be especially large, because that term is likely to appear in a large number of documents.

Parallelized BatchScanners

In addition to being able to scan over a single range of key-value pairs, Accumulo provides a `BatchScanner` in its client API that can be used to fetch rows from multiple places in a table simultaneously in multiple threads. This is also useful for applications performing queries using secondary indexes.

Namespaces

Accumulo tables can be assigned to a namespace, which enables them to be configured and managed as a group. This makes it easier to have multiple groups of people managing tables in the same cluster. See "Table Namespaces" on page 160 for details.

Use Cases Suited for Accumulo

Accumulo's design represents a set of objectives and technical features different from those in data management systems such as filesystems and relational databases. Application and system designers need to understand how these features work together. We present here a few applications that could leverage Accumulo's strengths.

A New Kind of Flexible Analytical Warehouse

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

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

Building the Next Gmail

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

Massive Graph or Machine-Learning Problems

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

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

Some machine-learning techniques, especially nonparametric algorithms such as k-nearest neighbors, are *memory-based* and require storing all the data rather than building a statistical model to represent the data. Keeping or "remembering" all the data points is what is meant by "memory-based," not that the data all lives in RAM. Accumulo is able to store large amounts of these data points and provides the basic data selection operations for supporting these algorithms efficiently. See "Machine Learning" on page 343 for more on this.

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

Relieving Relational Databases

Because relational databases have performed well over the past several decades, they have become the standard place for putting all data and have had to support a wide variety of data management problems. But as database expert Michael Stonebraker and others have argued, trying to have only one platform can result in challenges stemming from the difficulty of optimizing a single system for many use cases.

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

Massive Search Applications

Google has used Bigtable to power parts of its primary search application. Accumulo has features such as automatic partitioning, batch scanning, and flexible iterators that can be used to support complex and large-scale text search applications.

Applications with a Long History of Versioned Data

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