# CHAPTER 1

# INTRODUCTION

A Big-table is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products.

This describes the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and also describe the design and implementation of Bigtable.Over the last two and a half years Google designed, implemented, and deployed a distributed storage system for managing structured data called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability.

# CHAPTER 2

# DATA MODEL

A Bigtable is a sparse, distributed, persistent multi- dimensional sorted map. The map is indexed by a row,key,columnkey,timestamp.

(row:string, column:string, time:int64) → string

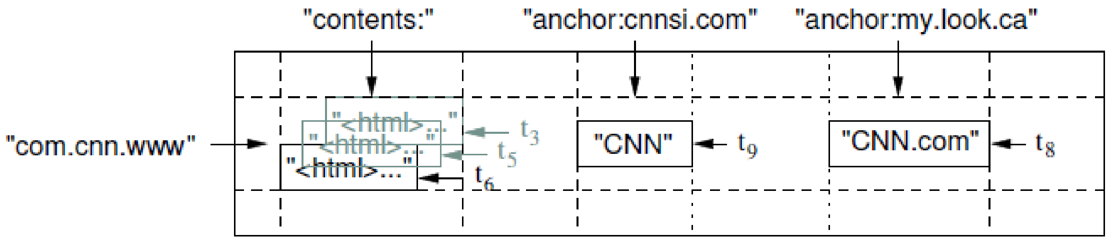


Figure 2.1: A slice of an example table that stores Web pages

The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps t3, t5, and t6.

We settled on this data model after examining a variety of potential uses of a Bigtable-like system. As one concrete example that drove some of our design decisions,suppose we want to keep a copy of a large collection of web pages and related information that could be used by many different projects; let us call this particular table the *Webtable*..

## 2.1 Rows

The row keys in a table are arbitrary strings (currently up to 64KB in size, although 10-100 bytes is a typical size for most of our users). Every read or write of data under a single row key is atomic (regardless of the number of different columns being read or written in the row), a design decision that makes it easier for clients to reason about the system's behavior in the presence of concurrent updates to the same row. Bigtable maintains data in lexicographic order by row key. The row range for a table is dynamically partitioned.

Each row range is called a *tablet*, which is the unit of distribution and load balancing. As a result, reads of short row ranges are effcient and typically require communication with only a small number of machines. Clients can exploit this property by selecting their row keys so that they get good locality for their data accesses. For example, in Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname components of the URLs. For example, we store data for maps.google.com/index.html under the key com.google.maps/index.html. Storing pages from the same domain near each other makes some host and domain analyses more efficent.

## 2.2 Column Familes

Column keys are grouped into sets called *column families*, which form the basic unit of access control. All data stored in a column family is usually of the same type (we compress data in the same column family together). A column family must be created before data can be stored under any column key in that family; after a family has been created, any column key within the family can be used. It is our intent that the number of distinct column families in a table be small (in the hundreds at most), and that families rarely change during operation. In contrast, a table may have an unbounded number of columns. A column key is named using the following syntax: *family*:*quali_er*. Column family names must be printable, but quali_ers may be arbitrary strings. An example column family for the Webtable is language, which stores the language in which a web page was written. We use only one column key in the language family, and it stores each web page's language ID. Another useful column family for this table is anchor; each column key in this family represents a single anchor,as shown in Figure 1. The quali_er is the name of the referring site; the cell contents is the link text. Access control and both disk and memory accounting are performed at the column-family level. In our Webtable example, these controls allow us to manage several different types of applications: some that add new base data, some that read the base data and create derived column families, and some that are only allowed to view existing data (and possibly not even to view all of the existing families for privacy reasons).

## 2.3 Timestamps

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Bigtable timestamps are 64-bit integers. They can be assigned by Bigtable, in which case they represent .real time. in microseconds, or be explicitly assigned by client.

Applications that need to avoid collisions must generate unique timestamps themselves. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read _rst. To make the management of versioned data less onerous, we support two per-column-family settings that tell Bigtable to garbage-collect cell versions automatically. The client can specify either that only the last *n* versions of a cell be kept, or that only new-enough versions be kept (e.g., only keep values that were

written in the last seven days).In our Web table example, we set the timestamps of the crawled pages stored in the contents: column to the times at which these page versions were actually crawled. The garbage-collection mechanism described above lets us keep only the most recent three versions of every page.*Mutations*, or changes, to a row take up extra storage space, because Cloud Bigtable stores mutations sequentially and compacts them only periodically. When Cloud Bigtable compacts a table, it removes values that are no longer needed. If you update the value in a cell, both the original value and the new value will be stored on disk for some amount of time until the data is compacted.Deletions also take up extra storage space, at least in the short term, because deletions are actually a specialized type of mutation. Until the table is compacted, a deletion uses extra storage rather than freeing up space.

# CHAPTER 3

# API

The Bigtable API provides functions for creating and deleting tables and column families. It also provides functions for changing cluster, table, and column family metadata, such as access control rights.Client applications can write or delete values in Bigtable, look up values from individual rows, or iterate over a subset of the data in a table. Figure 2 shows C++ code that uses a RowMutation abstraction to perform a series of updates. (Irrelevant details were elided to keep the example short.) The call to Apply performs an atomic mutation to the Webtable: it adds one anchor to www.cnn.com and deletes a different anchor. Figure 3 shows C++ code that uses a Scanner abstraction to iterate over all anchors in a particular row. Clients can iterate over multiple column families, and there are several mechanisms for limiting the rows,columns, and timestamps produced by a scan. For example,we could restrict the scan above to only produce anchors whose columns match the regular expression anchor.cnn.com, or to only produce anchors whose timestamps fall within ten days of the current time.

# CHAPTER 4

# BUILDING BLOCKS

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed Google File System (GFS) [17] to store log and data _les. A Bigtable cluster typically operates in a shared pool of machines that run a wide variety of other distributed applications, and Bigtable processes often share the same machines with processes from other applications. Bigtable depends on a cluster management system for scheduling jobs, managing resources on shared machines, dealing with machine failures, and monitoring machine status. The Google *SSTable* _le format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified.key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is con_gurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first  the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.

Bigtable relies on a highly-available and persistent distributed lock service called Chubby . A Chubby service consists of _ve active replicas, one of which is selected to be the master and actively serve requests. The service is live when a majority of the replicas are running and can communicate with each other. Chubby uses the Paxos algorithm .keep its replicas consistent in the face of failure. Chubby provides a namespace that consists of directories and small _les. Each directory or can be used as a lock, and reads and writes to . The Chubby client library provides consistent caching of Chubby _les. Each Chubby client maintains a *session* with a Chubby service. A client's session expires if it is unable to renew its session lease within the lease expiration time. When a client's session expires, it loses any locks and open handles. Chubby clients can also register callbacks on Chubby _les and directories for noti_cation of changes or session expiration.

Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data (see Section to discover tablet servers and tablet server deaths to store Bigtable schema information (the column family information for each table); and to store access control lists. If Chubby becomes unavailable for an extended period of time, Bigtable becomes unavailable. We recently measured this effect in 14 Bigtable clusters spanning 11 Chubby instances. The average percentage of Bigtable server hours during which some data stored in Bigtable was not available due to Chubby unavailability (caused by either Chubby outages or network issues) was 0.0047%. The percentage for the single cluster that was most affected by Chubby unavailability was 0.0326%.

# CHAPTER 5

# IMPLEMENTATION

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers. Tablet servers can be dynamically added (or removed) from a cluster to accommodate changes in workloads. The master is responsible for assigning tablets to tablet servers, detecting the addition and expiration of tablet servers, balancing tablet-server load, and garbage collection of in GFS. In addition, it handles schema changes such as table and column family creations. Each tsablet server manages a set of tablets (typically we have somewhere between ten to a thousand tablets per tablet server). The tablet server handles read and write requests to the tablets that it has loaded, and also splits tablets that have grown too large. As with many single-master distributed storage systems client data does not move through the master: clients communicate directly with tablet servers for reads and writes. Because Bigtable clients do not rely on the master for tablet location information, most clients never communicate with the master. As a result, the master is lightly loaded in practice. A Bigtable cluster stores a number of tables. Each table consists of a set of tablets, and each tablet contains all data associated with a row range. Initially, each table consists of just one tablet. As a table grows, it is automatically split into multiple tablets, each approximately 100-200 MB in size by default.

## 5.1 Tablet Location

We use a three-level hierarchy analogous to that of a B+- tree to store tablet location information (Figure 4).

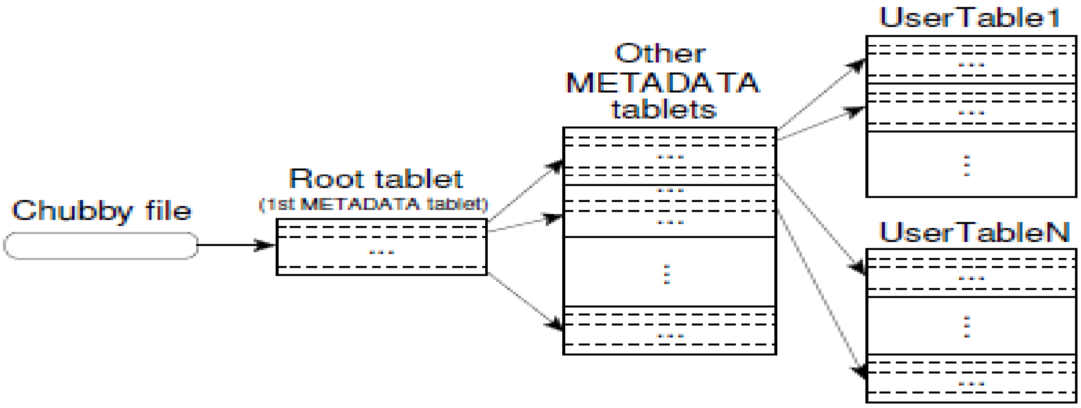


Figure 4: Tablet location hierarchy.

Figure 5.1: Tablet Location Hierarchy.

The  level is a _le stored in Chubby that contains the location of the *root tablet*. The *root tablet* contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The *root tablet* is just the  tablet in the METADATA table, but is treated specially.it is never split.to ensure that the tablet location hierarchy has no more than three levels.The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table. identi_er and its end row. Each METADATA row stores approximately 1KB of data in memory. With a modest limit of 128 MB METADATA tablets, our three-level location scheme is suf_cient to address 234 tablets (or 261 bytes in 128 MB tablets).

The client library caches tablet locations. If the clientdoes not know the location of a tablet, or if it discovers that cached location information is incorrect, then it recursively moves up the tablet location hierarchy. If the client's cache is empty, the location algorithm requires three network round-trips, including one read from Chubby. If the client's cache is stale, the location algorithm could take up to six round-trips, because stale cache entries are only discovered upon misses (assuming that METADATA tablets do not move very frequently). Although tablet locations are stored in memory, so no GFS accesses are required, we further reduce this cost in the common case by having the client library prefetch tablet locations: it reads the metadata for more than one tablet whenever it reads the METADATA table.

We also store secondary information in the METADATA table, including a log of all events pertaining to each tablet (such as when a server begins serving it). This information is helpful for debugging and performance analysis.

## 5.2 Table Assignment

Each tablet is assigned to one tablet server at a time. The master keeps track of the set of live tablet servers, and the current assignment of tablets to tablet servers, including which tablets are unassigned. When a tablet is unassigned, and a tablet server with sufficient room for the tablet is available, the master assigns the tablet by sending a tablet load request to the tablet server.

Bigtable uses Chubby to keep track of tablet servers. When a tablet server starts, it creates, and acquires an exclusive lock on, a uniquely-named  in a specific Chubby directory. The master monitors this directory (the *servers directory*) to discover tablet servers. A tablet server stops serving its tablets if it loses its exclusive lock: e.g., due to a network partition that caused the server to lose its Chubby session. (Chubby provides an effcient mechanism that allows a tablet server to check whether it still holds its lock without incurring network .) A tablet server will attempt to reacquire an exclusive lock on its  as long as the still exists. If the  no longer exists, then the tablet server will never be able to serve again, so it kills itself. Whenever a tablet server terminates (e.g., because the cluster management system is removing the tablet server's machine from the cluster), it attempts to release its lock so that the master will reassign its tablets more quickly.

The master is responsible for detecting when a tablet server is no longer serving its tablets, and for reassigning those tablets as soon as possible. To detect when a tablet server is no longer serving its tablets, the master periodically asks each tablet server for the status of its lock. If a tablet server reports that it has lost its lock, or if the master was unable to reach a server during its last several attempts, the master attempts to acquire an exclusive lock on the server's _le. If the master is able to acquire the lock, then Chubby is live and the tablet server is either dead or having trouble reaching Chubby, so the master ensures that the tablet server can never serve again by deleting its server _le. Once a server's  has been deleted, the master can move all the tablets that were previously assigned to that server into the set of unassigned tablets. To ensure that a Bigtable cluster is not vulnerable to networking issues between the master and Chubby, the master kills itself if its Chubby session expires. However, as described above, master failures do not change the assignment of tablets to tablet servers.

When a master is started by the cluster management system, it needs to discover the current tablet assignments before it can change them. The master executes the following steps at startup. (1) The master grabs a unique *master* lock in Chubby, which prevents concurrent master instantiations. (2) The master scans the servers directory in Chubby to  the live servers. (3) The master communicates with every live tablet server to discover what tablets are already assigned to each server. (4) The master scans the METADATA table to learn the set of tablets. Whenever this scan encounters a tablet that is not already assigned, the master adds the tablet to the set of unassigned tablets, which makes the tablet eligible for tablet assignment.

One complication is that the scan of the METADATA table cannot happen until the METADATA tablets have been assigned. Therefore, before starting this scan (step 4), the master adds the root tablet to the set of unassigned tablets if an assignment for the root tablet was not discovered during step 3. This addition ensures that the root tablet will  be assigned. Because the root tablet contains the names of all

METADATA tablets, the master knows about all of them after it has scanned the root tablet.

The set of existing tablets only changes when a table is created or deleted, two existing tablets are merged to form one larger tablet, or an existing tablet is split into two smaller tablets. The master is able to keep track of these changes because it initiates all but the last. Tablet splits are treated specially since they are initiated by a tablet server. The tablet server commits the split by recording information for the new tablet in the METADATA table. When the split has committed, it  the master. In case the split noti_cation is lost (either  because the tablet server or the master died), the master detects the new tablet when it asks a tablet server to load the tablet that has now split. The tablet server will notify the master of the split, because the tablet entry  in the METADATA table will specify only a portion of the tablet that the master asked it to load.

## 5.3 Tablet Serving

The persistent state of a tablet is stored in GFS, as illustrated in Figure 5. Updates are committed to a commit log that stores redo records. Of these updates, the recently committed ones are stored in memory in a sorted buffer called a *memtable*; the older updates are stored in a sequence of SSTables. To recover a tablet, a tablet server.

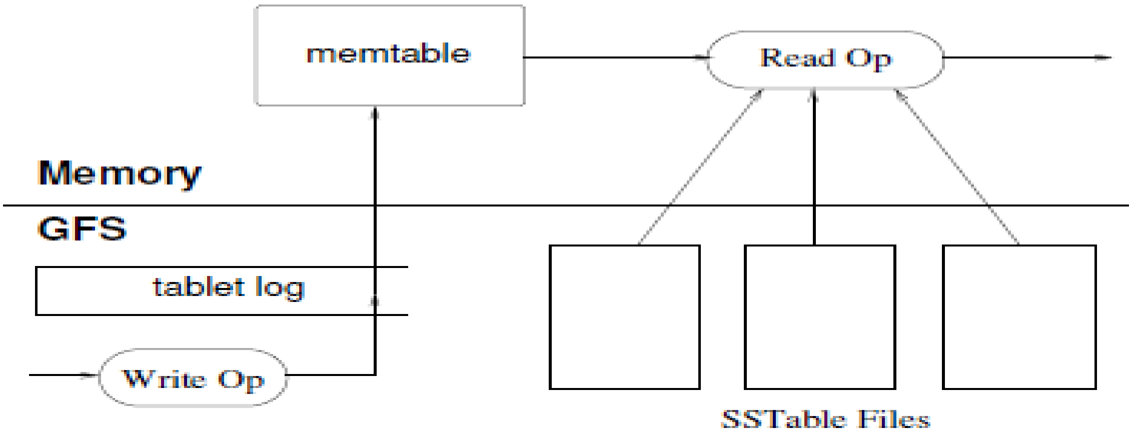


Figure 5: Tablet Representation

Figure 5.2: Tablet Representation

When a write operation arrives at a tablet server, the server checks that it is well-formed, and that the sender is authorized to perform the mutation. Authorization is performed by reading the list of permitted writers from a Chubby (which is almost always a hit in the Chubby client cache). A valid mutation is written to the commit log. Group commit is used to improve the throughput of lots of small mutations . After the write has been committed, its contents are inserted into the memtable.

When a read operation arrives at a tablet server, it is similarly checked for well-formedness and proper authorization. A valid read operation is executed on a merged view of the sequence of SSTables and the memtable. Since the SSTables and the memtable are lexicographically sorted data structures, the merged view can be formed efficiently. Incoming read and write operations can continue while tablets are split and merged.

## 5.4 Compaction

As write operations execute, the size of the memtable increases. When the memtable size reaches a threshold, the memtable is frozen, a new memtable is created, and the frozen memtable is converted to an SSTable and written to GFS. This *minor compaction* process has two goals: it shrinks the memory R6GFRTTFusage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

*compaction* process has two goals: it shrinks the memory usage of the tablet server, and it reduces the amount of data that has to be read from the commit log during recovery if this server dies. Incoming read and write operations can continue while compactions occur.

Every minor compaction creates a new SSTable. If this behavior continued unchecked, read operations might need to merge updates from an arbitrary number of SSTables. Instead, we bound the number of such by periodically executing a *merging compaction* in the background. A merging compaction reads the contents of a few SSTables and the memtable, and writes out a new SSTable. The input SSTables and memtable can be discarded as soon as the compaction has fnished.

A merging compaction that rewrites all SSTables into exactly one SSTable is called a *major compaction*. SSTables produced by non-major compactions can contain special deletion entries that suppress deleted data in older SSTables that are still live. A major compaction, on the other hand, produces an SSTable that contains no deletion information or deleted data. Bigtable cycles through all of its tablets and regularly applies major compactions to them. These major compactions allow Bigtable to reclaim resources used by deleted data, and also allow it to ensure that deleted data disappears from the system in a timely fashion, which is important for services that store sensitive data.

# CHAPTER 6

# REFINEMENTS

The implementation described in the previous section required a number of requirements to achieve the high performance, availability, and reliability required by our users. This section describes portions of the implementation in more detail in order to highlight these requirements.

## 6.1 Locality Groups

Clients can group multiple column families together into a *locality group*. A separate SSTable is generated for each locality group in each tablet. Segregating column families that are not typically accessed together into separate locality groups enables more effcient reads. For example, page metadata in Webtable (such as language and checksums) can be in one locality group, and the contents of the page can be in a different group: an ap- plication that wants to read the metadata does not need to read through all of the page contents. In addition, some useful tuning parameters can be specified on a per-locality group basis. For example, a locality group can be declared to be in-memory. SSTables for in-memory locality groups are loaded lazily into the memory of the tablet server. Once loaded, column families that belong to such locality groups can be read without accessing the disk. This feature is useful for small pieces of data that are accessed frequently: we use it internally for the location column family in the METADATA table.

## 6.2 Compression

Clients can control whether or not the SSTables for a locality group are compressed, and if so, which compression format is used. The user-specified compression format is applied to each SSTable block (whose size is controllable via a locality group specified tuning parameter).Although we lose some space by compressing each block separately, we benefit in that small portions of an SSTable can be read without decompressing the entire. Many clients use a two-pass custom compression scheme. The  pass uses Bentley and McIlroy's scheme which compresses long common strings across a large window. The second pass uses a fast compression algorithm that looks for repetitions in a small 16 KB window of the data. Both compression passes are very fast.they encode at 100.200 MB/s, and decode at 400.1000 MB/s on modern machines.

## 6.3 Caching for read performance

To improve read performance, tablet servers use two levels of caching. The Scan Cache is a higher-level cache that caches the key-value pairs returned by the SSTable interface to the tablet server code. The Block Cache is a lower-level cache that caches SSTables blocks that were read from GFS. The Scan Cache is most useful for applications that tend to read the same data repeatedly. The Block Cache is useful for applications that tend to read data that is close to the data they recently read (e.g., sequential reads, or random reads of different columns in the same locality group within a hot row).

## 6.4 Bloom Filters

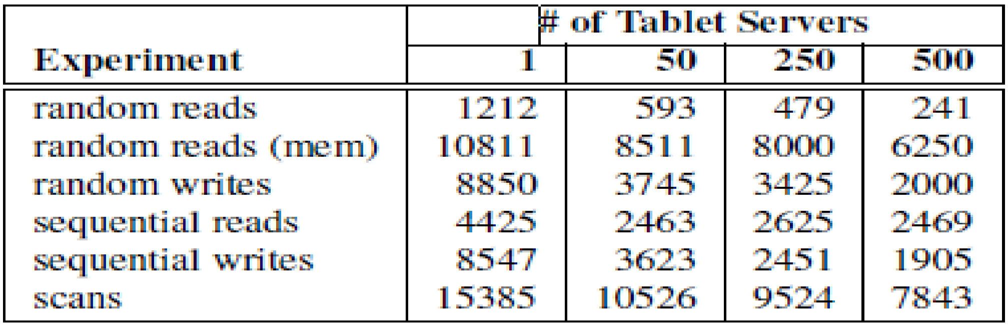
As described in Section 5.3, a read operation has to read from all SSTables that make up the state of a tablet. If these SSTables are not in memory, we may end updoing many disk accesses. We reduce the number of accesses by allowing clients to specify that Bloom filters  should be created for SSTables in a particular locality group. A Bloom filter allows us to ask whether an SSTable might contain any data for a specified row/column pair. For certain applications, a small amount of tablet server memory used for storing Bloom filters drastically reduces the number of disk seeks required for read operations. Our use of Bloom filters also implies that most lookups for non-existent rows or columns do not need to touch disk.

# CHAPTER 7

# PERFORMANCE EVALUATION

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. The tablet servers were con_gured to use 1GB of memory and to write to a GFS cell consisting of 1786 machines with two 400 GB IDE hard drives each. N client machines generated the Bigtable load used for these tests. (We used the same number of clients as tablet servers to ensure that clients were never a bottleneck.) Each machine had two dual-core Opteron 2 GHz chips, enough physical memory to hold the working set of all running processes, and a single gigabit Ethernet link. The machines were arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root. All of the machines were in the same hosting facility and therefore the round-trip time between any pair of machines was less than a millisecond

The tablet servers and master, test clients, and GFS servers all ran on the same set of machines. Every machine ran a GFS server. Some of the machines also ran either a tablet server, or a client process, or processes from other jobs that were using the pool at the same time as these experiments. R is the distinct number of Bigtable row keys involved in the test. R was chosen so that each benchmark read or wrote approximately 1 GB of data per tablet server. The *sequential write* benchmark used row keys withnames 0 to R 1. This space of row keys was partitioned into 10N equal-sized ranges.



| Experiment | # of Tablet Servers | | | |
|---|---|---|---|---|
| | 1 | 50 | 250 | 500 |
| random reads | 1212 | 593 | 479 | 241 |
| random reads (mem) | 10811 | 8511 | 8000 | 6250 |
| random writes | 8850 | 3745 | 3425 | 2000 |
| sequential reads | 4425 | 2463 | 2625 | 2469 |
| sequential writes | 8547 | 3623 | 2451 | 1905 |
| scans | 15385 | 10526 | 9524 | 7843 |

Table 7.1: The rate per tablet serve

.

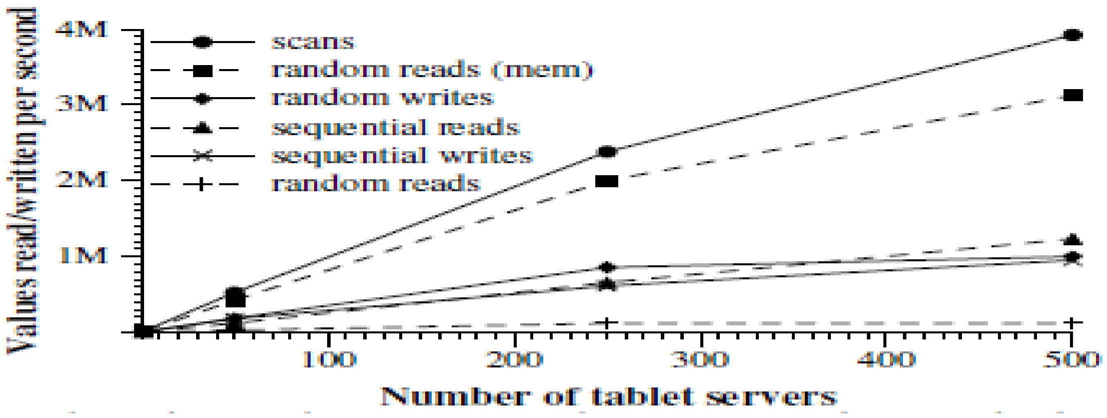


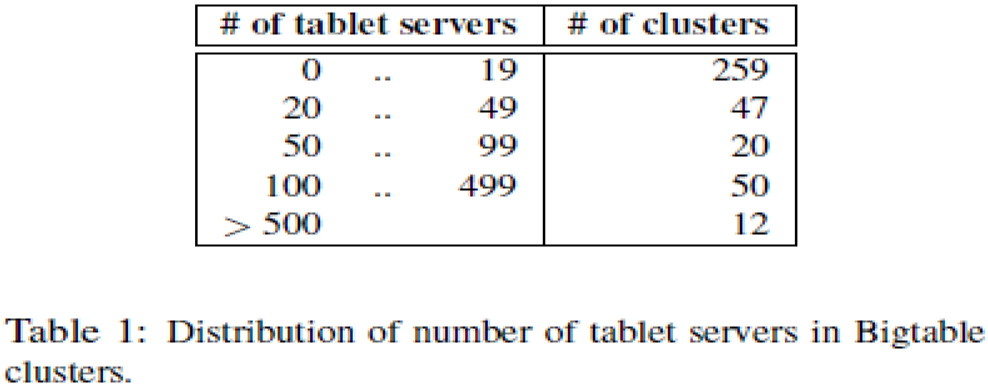
Figure 7.1: Number of 1000-byte values read/written per second.

Figure 6 shows two views on the performance of our benchmarks when reading and writing 1000-byte values to Bigtable. The table shows the number of operations per second per tablet server; the graph shows the aggregate number of operations per second.

This dynamic assignment helped mitigate the effects of performance variations caused by other processes running on the client machines. We wrote a single string under each row key. Each string was generated randomly and was therefore uncompressible.

## 7.1 Scaling

Aggregate throughput increases dramatically, by over a factor of a hundred, as we increase the number of tablet servers in the system from 1 to 500. For example, the performance of random reads from memory increases by almost a factor of 300 as the number of tablet server increases by a factor of 500. This behavior occurs because the bottleneck on performance for this benchmark is the individual tablet server CPU.



| # of tablet servers | # of clusters |
|---|---|
| 0 .. 19 | 259 |
| 20 .. 49 | 47 |
| 50 .. 99 | 20 |
| 100 .. 499 | 50 |
| > 500 | 12 |

Table 1: Distribution of number of tablet servers in Bigtable clusters.

Table 7.2: Distribution of number of tablet servers in Bigtable clusters.

However, performance does not increase linearly. For most benchmarks, there is a significant drop in per-server throughput when going from 1 to 50 tablet servers. This drop is caused by imbalance in load in multiple server configurations, often due to other processes contending for CPU and network. Our load balancing algorithm attempts to deal with this imbalance, but cannot do a perfect job for two main reasons: rebalancing is throttled to reduce the number of tablet movements (a tablet is unavailable for a short time, typically less than one second, when it is moved), and the load generated by our benchmarks shifts around as the benchmark progresses.

# CHAPTER 8

# SQL VS NOSQL

First, let’s take a look at one of the main features that separates these two systems: the way they structure data. A **relational database**—or, an SQL database, named for the language it’s written in, Structured Query Language (SQL)—is the more rigid, structured way of storing data, like a phone book. Developed by IBM in the 1970s, a relational database consists of two or more tables with columns and rows. Each row represents an entry, and each column sorts a very specific type of information, like a name, address, and phone number. The relationship between tables and field types is called a **schema**. In a relational database, the schema must be clearly defined before any information can be added.

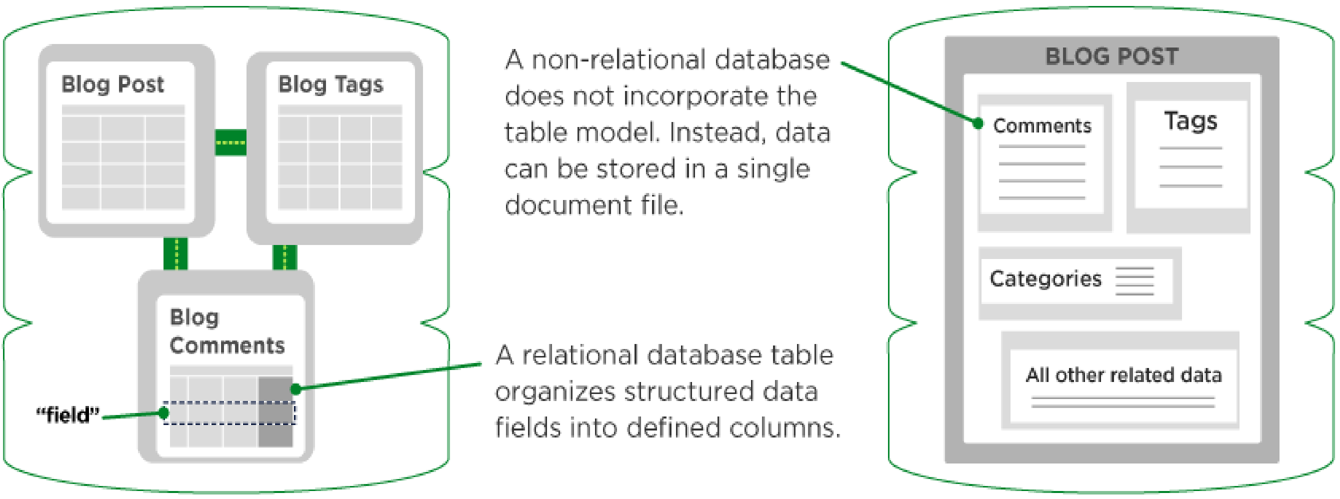
For a relational database to be effective, the data you’re storing in it has to be structured in a very organized way. A well-designed schema minimizes data redundancy and prevents tables from becoming out-of-sync, a critical feature for many businesses, especially those that record financial transactions. A poorly designed schema can result in organizational headaches due to its rigidity. For example, a column designed to store U.S. phone numbers might require 10 digits because that’s the standard for phone numbers in the U.S. This has the advantage of rejecting any invalid values (for example, if a number is missing an area code).

If your data requirements aren’t clear at the outset or if you’re dealing with massive amounts of unstructured data, you may not have the luxury of developing

the term NoSQL encompasses a broad range of alternatives to relational databases, what they have in common is that they allow you to treat data more flexibly.

RELATIONAL VS. NON-RELATIONAL DATABASES

RELATIONAL VS. NON-RELATIONAL DATABASES



A non-relational database does not incorporate the table model. Instead, data can be stored in a single document file.

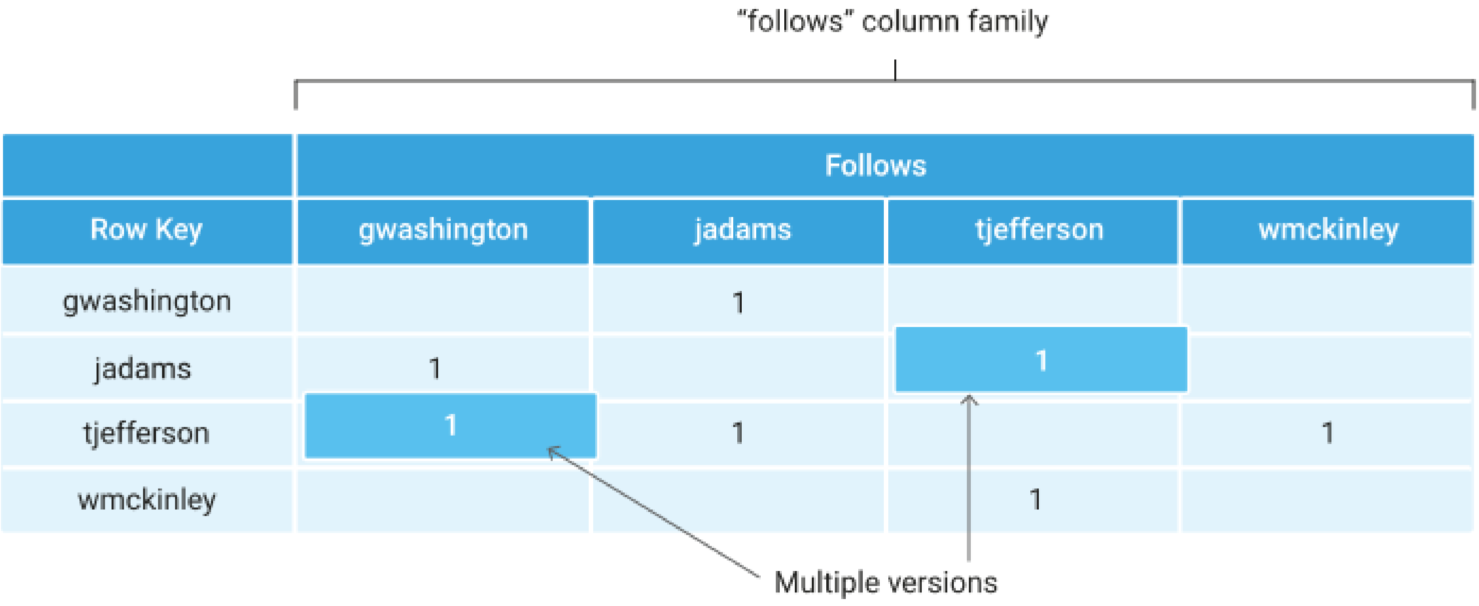
A relational database table organizes structured data fields into defined columns.

Figure 8.1: Relational Vs Non Relational databases

# CHAPTER 9

# BIGTABLE STORAGE MODEL

Bigtable stores data in massively scalable tables, each of which is a sorted key/value map. The table is composed of *rows*, each of which typically describes a single entity, and *columns*, which contain individual values for each row. Each row is indexed by a single *row key*, and columns that are related to one another are typically grouped together into a *column family*. Each column is identified by a combination of the column family and a *column qualifier*, which is a unique name within the column family.



“follows” column family
Multiple versions

| | Follows | | | |
|---|---|---|---|---|
| Row Key | gwashington | jadams | tjefferson | wmckinley |
| gwashington | | 1 | | |
| jadams | 1 | | 1 | |
| tjefferson | 1 | 1 | | 1 |
| wmckinley | | | 1 | |

Table 9.1 Bigtable Storage Model

The table contains one column family**,** the  family. This family contains multiple column qualifiers.Column qualifiers are used as data**.** This design choice takes advantage of the sparseness of Cloud Bigtable tables, and the fact that new column qualifiers can be added on the fly.The username is used as the row key**.**

# CHAPTER 10

# BIGTABLE ARCHITECTURE

The following diagram shows a simplified version of Bigtable's overall architecture:

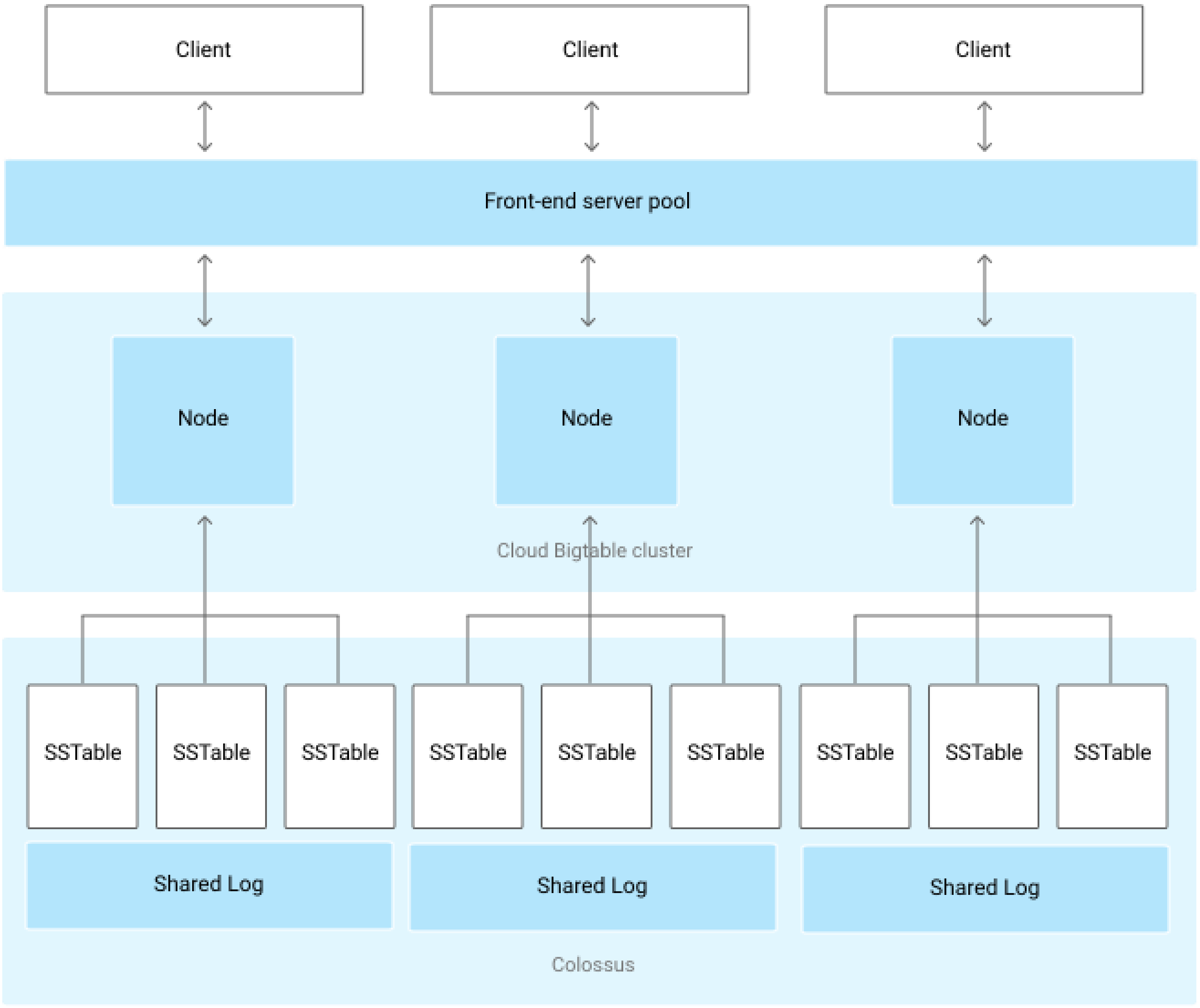


Figure 10.1: Bigtable Architecure

As the diagram illustrates, all client requests go through a front-end server before they are sent to a Cloud Bigtable node. The nodes are organized into a Cloud Bigtable cluster, which belongs to a Cloud Bigtable instance, a container for the cluster.

Each node in the cluster handles a subset of the requests to the cluster. By adding nodes to a cluster, you can increase the number of simultaneous requests that the cluster can handle, as well as the maximum throughput for the entire cluster.

A Bigtable table is sharded into blocks of contiguous rows, called *tablets*, to help balance the workload of queries. (Tablets are similar to HBase regions.) Tablets are stored on Colossus, Google's file system, in SSTable format. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Each tablet is associated with a specific Cloud Bigtable node. In addition to the SSTable files, all writes are stored in Colossus's shared log as soon as they are acknowledged by Cloud Bigtable, providing increased durability.

Importantly, data is never stored in Cloud Bigtable nodes themselves; each node has pointers to a set of tablets that are stored on Colossus. Rebalancing tablets from one node to another is very fast, because the actual data is not copied. Cloud Bigtable simply updates the pointers for each node.Recovery from the failure of a Cloud Bigtable node is very fast, because only metadata needs to be migrated to the replacement node.When a Cloud Bigtable node fails, no data is lost.

Bigtable is available as open source, which is a major advantage as it enriches the kind of comments and contributions it receives over time. Users are then assured a good degree of improvement and addition with an active developer base in the open source context. This also means that Bigtable would adhere to the required industry standards.For example, the HBase API, which is one of the most popularly used bases, is seamlessly supported and organizations that already use products like HBase would find it doubly simple to set up Bigtable for their data.

# CHAPTER 11

# REAL APPLICATIONS

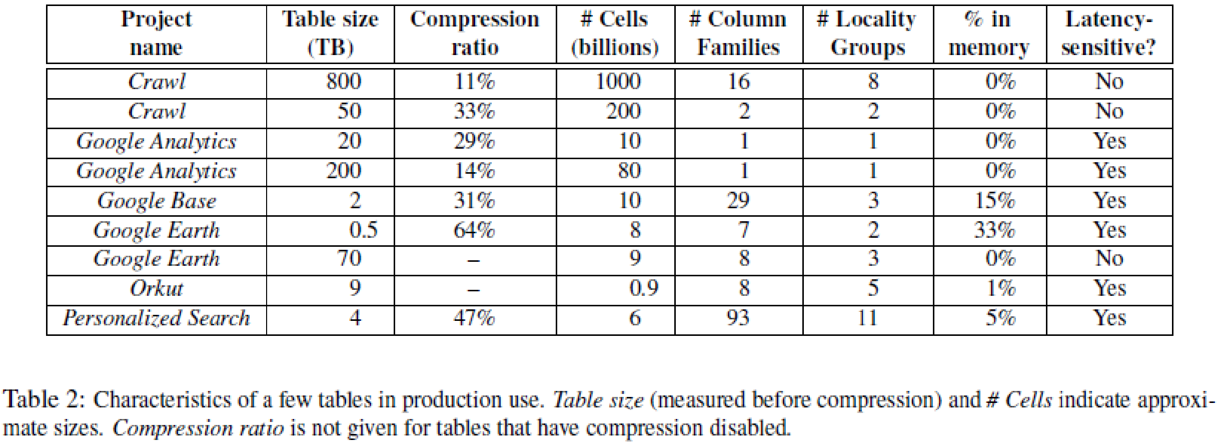
As of August 2012, there are 388 non-test Bigtable clusters running in various Google machine clusters, with a combined total of about 24,500 tablet servers. Table 1 shows a rough distribution of tablet servers per cluster. Many of these clusters are used for development purposes and therefore are idle for significant periods. One group of 14 busy clusters with 8069 total tablet servers saw an aggregate volume of more than 1.2 million requests per second, with incoming RPC traffic of about 741 MB/s and outgoing RPC traf_c of about 16 GB/s. Table 2 provides some data about a few of the tablescurrently in use. Some tables store data that is served to users, whereas others store data for batch processing the tables range widely in total size, average cell size.

## 11.1 Google Analytics

Google operates a collection of services that provide users with access to high-resolution satellite imagery of the world's surface, both through the web-based Google Maps interface (maps.google.com) and through the Google Earth (earth.google.com) custom client software. These products allow users to navigate across the world's surface: they can pan, view, and annotate satellite imagery at many different levels of resolution. This system uses one table to pre process data, and a different set of tables for serving client data. The pre processing pipeline uses one table to store raw imagery. During preprocessing, the imagery is cleaned and consolidated into serving data. This table contains approximately 70 terabytes of data and therefore is served from disk. The images are ef_ciently compressed already, so Bigtable compression is disabled.

Google Analytics (analytics.google.com) is a service that helps webmasters analyse traffic patterns at their web sites. It provides aggregate statistics, such as the number of unique visitors per day and the page views per URL per day, as well as site-tracking reports, such as the percentage of users that made a purchase, given that they earlier viewed a specific page. To enable the service, webmasters embed a small JavaScript program in their web pages. This program is invoked whenever a page is visited. It records various information about the request in Google Analytics, such as a user identifier and information about the page being fetched. Google Analytics summarizes this data and makes it available to webmasters. We brifiely describe two of the tables used by Google Analytics. The raw click table (.200 TB) maintains a row for each end-user session. The row name is a tuple containing the website's name and the time at which the session was created. This schema ensures that sessions that visit the same web site are contiguous, and that they are sorted chronologically. This table compresses to 14% of its original size.

## 11.2 Google Earth



| Project name | Table size (TB) | Compression ratio | # Cells (billions) | # Column Families | # Locality Groups | % in memory | Latency-sensitive? |
|---|---|---|---|---|---|---|---|
| *Crawl* | 800 | 11% | 1000 | 16 | 8 | 0% | No |
| *Crawl* | 50 | 33% | 200 | 2 | 2 | 0% | No |
| *Google Analytics* | 20 | 29% | 10 | 1 | 1 | 0% | Yes |
| *Google Analytics* | 200 | 14% | 80 | 1 | 1 | 0% | Yes |
| *Google Base* | 2 | 31% | 10 | 29 | 3 | 15% | Yes |
| *Google Earth* | 0.5 | 64% | 8 | 7 | 2 | 33% | Yes |
| *Google Earth* | 70 | – | 9 | 8 | 3 | 0% | No |
| *Orkut* | 9 | – | 0.9 | 8 | 5 | 1% | Yes |
| *Personalized Search* | 4 | 47% | 6 | 93 | 11 | 5% | Yes |

Table 2: Characteristics of a few tables in production use. *Table size* (measured before compression) and *# Cells* indicate approximate sizes. *Compression ratio* is not given for tables that have compression disabled.

Table 11.1 Charactristics of a few tables in production use.

Each row in the imagery table corresponds to a single geographic segment. Rows are named to ensure that adjacent geographic segments are stored near each other. The table contains a column family to keep track of the sources of data for each segment. This column family has a large number of columns: essentially one for each raw data image. Since each segment is only built from a few images, this column family is very sparse.

## 11.3 Personalized Search

Personalized Search stores each user's data in Bigtable. Each user has a unique userid and is assigned a row named by that userid. All user actions are stored in a table. A separate column family is reserved for each type of action (for example, there is a column family that stores all web queries). Each data element uses as its Bigtable timestamp the time at which the corresponding user action occurred. Personalized Search generates user  using a MapReduce over Bigtable. These user.

The Personalized Search data is replicated across several Bigtable clusters to increase availability and to reduce latency due to distance from clients. The Personalized
Search team originally built a client-side replication mechanism on top of Bigtable that ensured eventual consistency of all replicas. The current system now uses a replication subsystem that is built into the servers.Personalized Search (www.google.com/psearch) is an opt-in service that records user queries and clicks across a variety of Google properties such as web search, images, and news. Users can browse their search histories to revisit their old queries and clicks, and they can ask for personalized search results based on their historical Google usage patterns.

The design of the Personalized Search storage system allows other groups to add new per-user information in their own columns, and the system is now used by many other Google properties that need to store per-user configuration options and settings. Sharing a table amongst many groups resulted in an unusually large number of column families. To help support sharing, we added a simple quota mechanism to Bigtable to limit the storage consumption by any particular client in shared tables; this mechanism provides some isolation between the various product groups using this system for per-user information storage.

# CONCLUSION

We have described Bigtable, a distributed system for storing structured data at Google. Bigtable clusters have been in production used. and we spent roughly seven person-years on design and implementation before that date. As of August 2012, more than sixty projects are using Bigtable. Our users like the performance and high availability provided by the Bigtable implementation, and that they can scale the capacity of their clusters by simply adding more machines to the system as their resource demands change over time. Applications that use Bigtable have been observed to have benefitted from performance, high availability and scalability. The unusual interface to Bigtable compared to traditional databases, lack of general purpose transactions, etc have not been a hindrance given many google products successfully use Bigtable implementation. Google has had significant advantages building their own storage solution by being able to have full control and flexibility and by removing bottlenecks and inefficiencies as they arise.

# REFERENCES

1. ABADI, D. J., MADDEN, S. R., AND FERREIRA, M. C. Integrating compression and execution in columnoriented database systems. *Proc. of SIGMOD* (2006).

2. AILAMAKI, A., DEWITT, D. J., HILL, M. D., AND SKOUNAKIS,M. Weaving relations for cache performance. In *The VLDB Journal* (2001), pp. 169.180..

3. BANGA, G., DRUSCHEL, P., AND MOGUL, J. C. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd OSDI* (Feb.1999), pp. 45.58.

4. BARU, C. K., FECTEAU, G., GOYAL, A., HSIAO,H., JHINGRAN, A.,PADMANABHAN, S., COPELAND,G. P., AND WILSON, W. G. DB2 parallel edition. *IBM Systems Journal 34*, 2 (1995), 292.322.

5. BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D.,KARLIN, S., PETERSON, L., ROSCOE, T., SPALINK, T., AND WAWRZONIAK, M. Operating system support for planetary-scale network services. In *Proc. of the 1st NSDI* (Mar. 2004), pp. 253.266.

6. BENTLEY, J. L., AND MCILROY, M. D. Data compression using long common strings. In *Data Compression Conference* (1999), pp. 287.295.

7. BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors.
*CACM 13*, 7 (1970), 422.426.

8. BURROWS, M. The Chubby lock service for looselycoupled distributed systems.
In *Proc. of the 7th OSDI* (Nov. 2006).