Google BigTable

Introduction

Bigtable 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. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

A Bigtable is really a sparse, distributed, persistent multidimensional sorted map. The map is listed in a row key, column key, along with a timestamp each value within the map is definitely an uninterpreted variety of bytes.

Google BigTable And NoSql

NoSQL is an umbrella term for all the databases that are different from 'the standard' SQL databases, such as MySQL, Microsoft SQL Server and PostgreSQL. These 'standard' SQL databases are all relational databases, feature the SQL query language and adhere to the ACID properties. These properties basically boil down to consistency. A NoSQL database is different because it doesn't support one or more of these key features of the so-called 'SQL databases':

* Consistency
* Relational data
* SQL language

Most of these features go hand in hand.

Consistency

Consistency is where most NoSQL databases differ from SQL databases. You can pull the plug from a SQL database and it will make sure your data is still consistent and uncorrupted. NoSQL databases tend to sacrifice this consistency for better scalability. Google's Bigtable also does this.

Relational data

SQL databases revolve around normalized, relational data. The database ensures that these relations stay valid and consistent, no matter what you throw at it.

NoSQL databases usually don't support relations, because they don't support the consistency to enforce these relations. Also, relational data is bad for performance when the data is distributed across several servers.

SQL language

The SQL language was designed especially for relational databases, the so-called 'SQL databases'. Since most NoSQL databases are very different from relational databases, they don't have the need for SQL. Also, some NoSQL databases have features that simply cannot be expressed in SQL, thus requiring a different query language.

The internal mechanics of Bigtable versus, say, MySQL are so dissimilar as to make comparison difficult, and the intended goals don't overlap much either. But you can think of Bigtable a bit like a single-table database. Imagine, for example, the difficulties you would run into if you tried to implement Google's entire web search system with a MySQL database -- Bigtable was built around solving those problems.

*Bigtable datasets can be queried from services like AppEngine using a language called GQL ("gee-kwal") which is a based on a subset of SQL.* Conspicuously missing from GQL is any sort of JOIN command. Because of the distributed nature of a Bigtable database, performing a join between two tables would be terribly inefficient. Instead, the programmer has to implement such logic in his application, or design his application so as to not need it.

NoSQL Emerged From a Need

*Data Storage:* The world's stored digital data is measured in exabytes. An exabyte is equal to one billion gigabytes (GB) of data.

According to Internet.com, the amount of stored data added in 2006 was 161 exabytes. Just 4 years later in 2010, the amount of data stored will be almost 1,000 ExaBytes which is an increase of over 500%. In other words, there is a lot of data being stored in the world and its just going to continue growing.

*Interconnected Data:* Data continues to become more connected. The creation of the web fostered in hyperlinks, blogs have pingbacks and every major social network system has tags that tie things together. Major systems are built to be interconnected.

*Complex Data Structure:* NoSQL can handle hierarchical nested data structures easily. To accomplish the same thing in SQL, you would need multiple relational tables with all kinds of keys. In addition, there is a relationship between performance and data complexity. Performance can degrade in a traditional RDBMS as we store the massive amounts of data required in social networking applications and the semantic web.

Googel BigTable Characteristics

BigTable is a distributed storage system that is structured as a large table: one that may be petabytes in size and distributed among tens of thousands of machines. It is designed for storing items such as billions of URLs, with many versions per page; over 100 TB of satellite image data; hundreds of millions of users; and performing thousands of queries a second. BigTable was developed at Google in has been in use since 2005 in dozens of Google services. An open source version, HBase, was created by the Apache project on top of the Hadoop core. Apache Cassandra, first developed at Facebook to power their search engine, is similar to BigTable with a tunable consistency model and no master (central server).

It is easy enough to picture a simple table. Let's look at a few characteristics of BigTable:

**Map**

A map is an associative array; a data structure that allows one to look up a value to a corresponding key quickly. BigTable is a collection of (key, value) pairs where the key identifies a row and the value is the set of columns.

**Persistant**

The data is stored peristantly on disk.

**Distributed**

BigTable's data is distributed among many independent machines. At Google, BigTable is built on top of GFS (Google File System). The Apache open source version of BigTable, HBase, is built on top of HDFS (Hadoop Distributed File System) or Amazon S3. The table is broken up among rows, with groups of adjacent rows managed by a server. *A row itself is never distributed.*

**Sparse**

The table is sparse, meaning that different rows in a table may use different columns, with many of the columns empty for a particular row.

**Sorted**

Most associative arrays are not sorted. A key is hashed to a position in a table. BigTable sorts its data by keys. This helps keep related data close together, usually on the same machine — assuming that one structures keys in such a way that sorting brings the data together. For example, if domain names are used as keys in a BigTable, it makes sense to store them in reverse order to ensure that related domains are close together. For example:

edu.rutgers.cs

edu.rutgers.nb

edu.rutgers.www

**Multidimensional**

A table is indexed by rows. Each row contains one or more named column families. Column families are defined when the table is first created. Within a column family, one may have one or more named columns. All data within a column family is usually of the same type. The implementation of BigTable usually compresses all the columns within a column family together. Columns within a column family can be created on the fly. Rows, column families and columns provide a three-level naming hierarchy in identifying data. For example:

edu.rutgers.cs" : { // row

"users" : { // column family

"watrous": "Donald", // column

"hedrick": "Charles", // column

"pxk" : "Paul" // column

}

"sysinfo" : { // another column family

"" : "SunOS 5.8" // column (null name)

}

}

To get data from BigTable, you need to provide a fully-qualified name in the form column-family:column. For example, users:pxk or sysinfo:. The latter shows an null column name.

**Time-based**

Time is another dimension in BigTable data. Every column family may keep multiple versions of column family data. If an application does not specify a timestamp, it will retrieve the latest version of the column family. Alternatively, it can specify a timestamp and get the latest version that is earlier than or equal to that timestamp.

Data Model

A Bigtable is a sparse, distributed, persistent multidimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

***(row:string, column:string, time:int64) -> string***

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. In Webtable, we would use URLs as row keys, various aspects of web pages as column names, and store the contents of the web pages in the contents: column under the timestamps when they were fetched, as illustrated in Figure 1

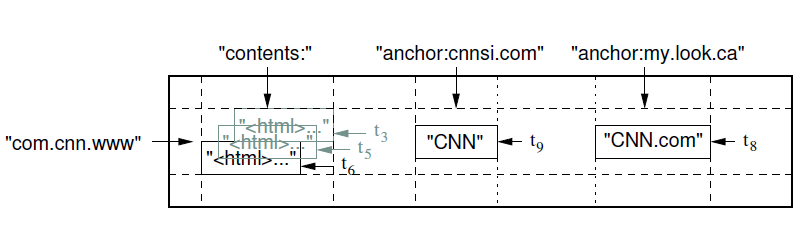
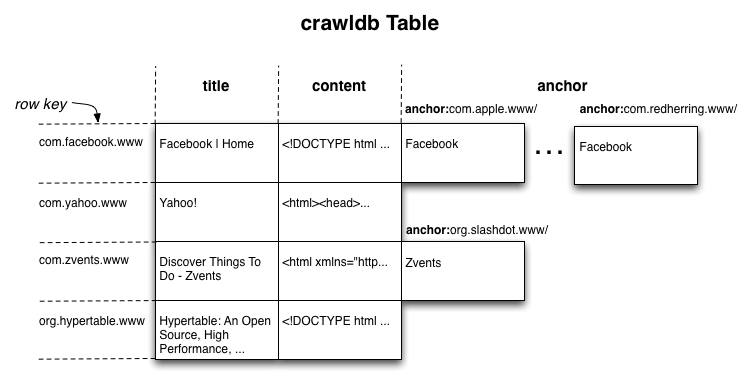
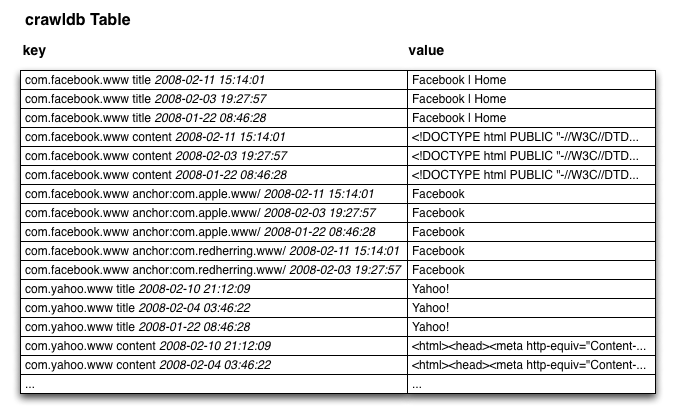


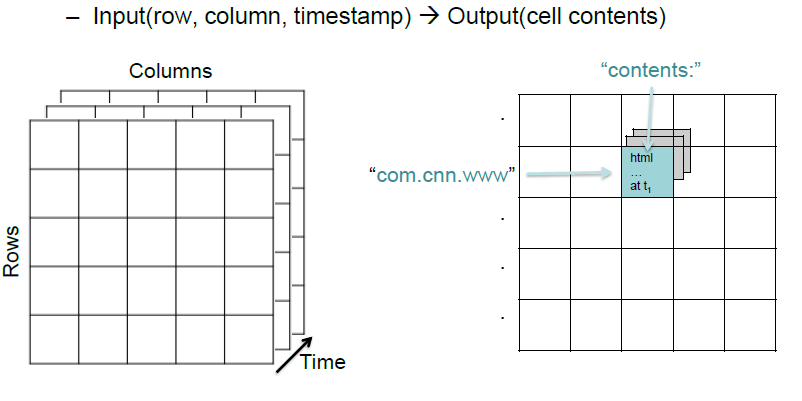
Figure 1





Key Value Pairs

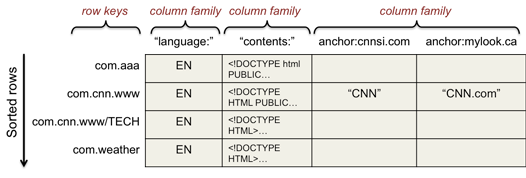
Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the 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.



***3 Dimensional Structure***

**Column Families**

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.



Let's look at a sample slice of a table that stores web pages (this example is from Google's paper on BigTable). The row key is the page URL. For example, "com.cnn.www". various attributes of the page are stored in column families. A contents column family contains page contents (there are no columns within this column family). A language column family contains the language identifier for the page. Finally, an anchor column family contains the text of various anchors from other web pages. The column name is the URL of the page making the reference. These three column families underscore a few points. A column may be a single short value, as seen in the language column family. This is our classic database view of columns. In BigTable, however, there is no type associated with the column. It is just a bunch of bytes. The data in a column family may also be large, as in the contents column family. The anchor column family illustrates the extra hierarchy created by having columns within a column family. It also illustrates the fact that columns can be created dynamically (one for each external anchor), unlike column families. Finally, it illustrates the sparse aspect of BigTable. In this example, the list of columns within the anchor column family will likely vary tremendously for each URL. In all, we may have a huge number (e.g., hundreds of thousands or millions) of columns but the column family for each row will have only a tiny fraction of them populated. While the number of column families will typically be small in a table (at most hundreds), the number of columns is unlimited.

**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 efficient 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 efficient.

**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. 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 first.

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).

**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.

// Open the table

Table \*T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor

RowMutation r1(T, "com.cnn.www");

r1.Set("anchor:www.c-span.org", "CNN");

r1.Delete("anchor:www.abc.com");

Operation op;

Apply(&op, &r1);

***Figure 2: Writing to Bigtable***.

Scanner scanner(T);

ScanStream \*stream;

stream = scanner.FetchColumnFamily("anchor");

stream->SetReturnAllVersions();

scanner.Lookup("com.cnn.www");

for (; !stream->Done(); stream->Next()) {

printf("%s %s %lld %s\n",

scanner.RowName(),

stream->ColumnName(),

stream->MicroTimestamp(),

stream->Value());

}

***Figure 3: Reading from Bigtable.***

Building Blocks

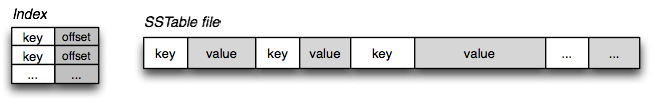
***Google File System (GFS)***

Bigtable is built on several other pieces of Google infrastructure. Bigtable uses the distributed ***Google File System (GFS)*** 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.

***SSTable***

The Google **SSTable** file 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 ). 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.

**SSTable**



***Chubby Lock Service***

Bigtable relies on a highly-available and persistent distributed lock service called Chubby. Chubby uses the Paxos algorithm. Chubby lock service, which is intended to provide coarse-grained locking as well as reliable (though low-volume) storage for a loosely-coupled distributed system. Chubby provides an interface much like a distributed ﬁle system with advisory locks, but the design emphasis is on availability and reliability, as opposed to high performance.

In BigTable, Chubby is used to:

* ensure there is only one active master
* store the bootstrap location of BigTable data
* discover tablet servers
* store BigTable schema information
* store access control lists

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 files in GFS. In addition, it handles schema changes such as table and column family creations.

Each tablet 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.

Implementation has Three Components:

* Library linked to each client
* Master server
* Tablet servers

*Library linked to each client*

* Client LibraryClient applications can lookup, read,and write to cells
* Single-row transactions (but not multi-)
* Sawzall scripts can read but not write

*Master Server*

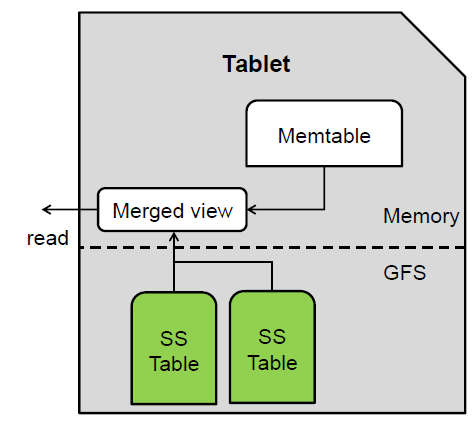
* Detects and adds new servers in the system
* Distributes data among servers
* Balances load
* Handles schema changes

*Tablet Servers*

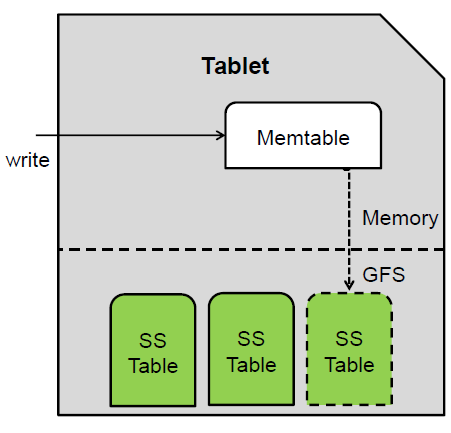
* Rows are grouped into tablets
* Each tablet contains a row range
* Tablets are distributed among the tablet servers

*Tablet Servers – read/write*

* Tablets exist partially in memory and on disk
* Memtable is a memory buffer for recent updates
* SSTable files in GFS are for old updates
* Servers check with Chubby to verify the authorization of the read or write
* Writes are first stored in memtable, and then later stored in SSTable files
* Reads are executed on the union of the memtable and SSTable files

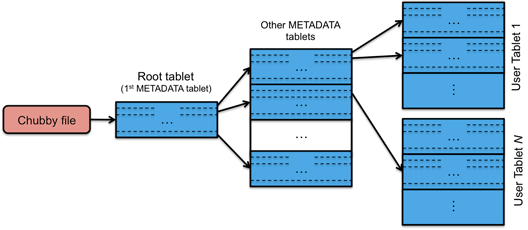


***Read Process and Write Process***



Tablet Location

We use a three-level hierarchy analogous to that of a B+ tree to store tablet location information. The first level is a file 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 first 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.

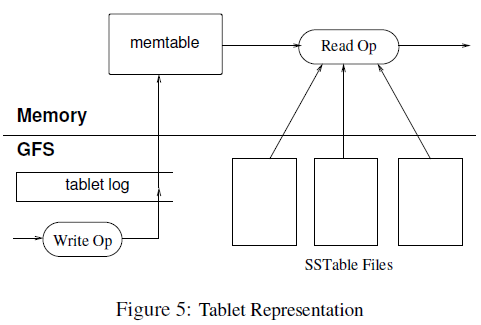


***BigTable indexing hierarchy***

The METADATA table stores the location of a tablet under a row key that is an encoding of the tablet's table identifier 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 sufficient to address 234 tablets (or 261 bytes in 128 MB tablets).

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.



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 file (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.

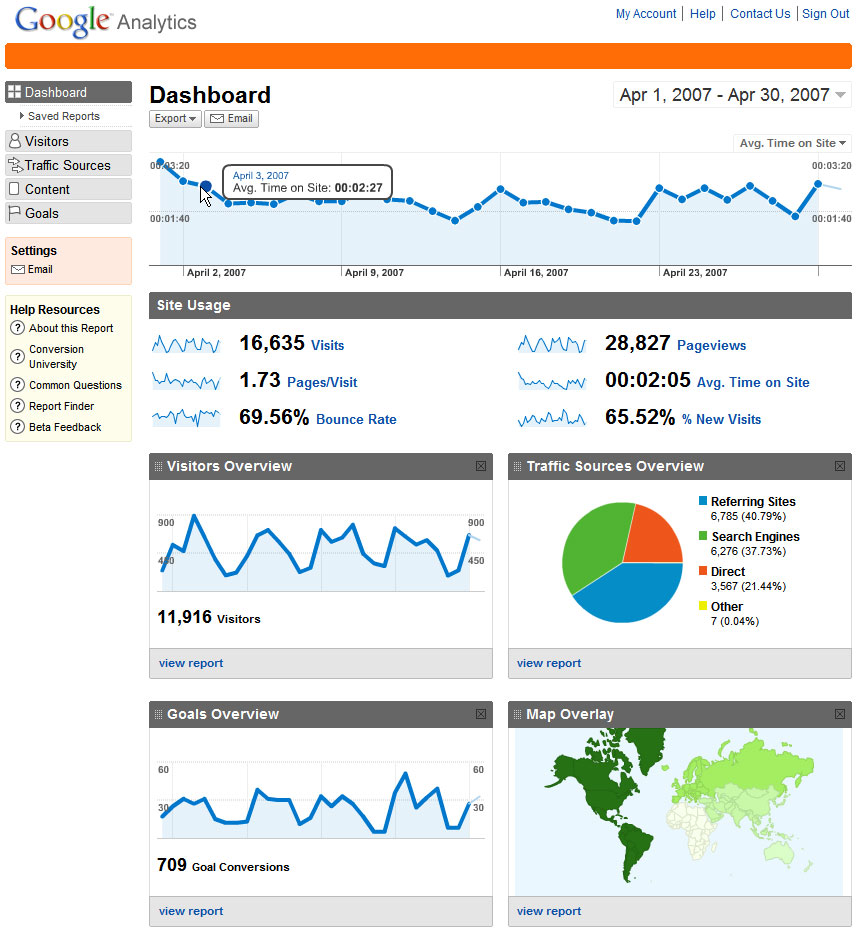
Real Applications

***Google Analytics (GA)***

Google Analytics (GA) is a service offered by Google that generates detailed statistics about a website's traffic and traffic sources and measures conversions and sales. The product is aimed at marketers as opposed to webmasters and technologists from which the industry of web analytics originally grew. It is the most widely used website statistics service.GA can track visitors from all referrers, including search engines and social networks, direct visits and referring sites. It also displays advertising, pay-per-click networks, e-mail marketing and digital collateral such as links within PDF documents.

***Google Earth***

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 preprocess data, and a different set of tables for serving client data.



Google Analytics (GA)

***Personalized Search***

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. 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 profiles using a MapReduce over Bigtable. These user profiles are used to personalize live search results.

Advantages

Big table was designed to maintain chronological queries and response time is far better for a query when compared to RDBMS. Conventional querying approaches like joins and normalization methodology used in RDBMS are not required here. Data compression is easier because of sparse rows.

Disadvantages

• It is not an open source database.

• Does not support final consistency.

• Capability of queries is limited.

• Inadequate access control.

• Requires adaptation to the Bigtable approach for application writing.

• Demands manual query programming as Structure query language is not supported by Bigtable.

• No support for ACID transactions as used in RDBMS.

**Conclusions**

* BigTable has a client library, master, and tablet servers
* Built on top of Chubby and GFS
* Scalable, distributed storage for highly structured data
* Built abstractions for building storage systems rather than building a particular storage system
* Built proof of concept file service
* Made many simplifying assumptions enabling simple designs
* Bigtable scales to petabytes of data across thousands of commodity Linux servers
* Developers can have a hard time adapting to different models
* Google's structured storage needs are satisfied

References

***Bigtable: A Distributed Storage System for Structured Data***

*Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach*

*Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber*

*ffay,jeff,sanjay,wilsonh,kerr,m3b,tushar,\_kes,gruberg@google.com*

**Google, Inc.**

*ABADI, D. J., MADDEN, S. R., AND FERREIRA,*

*M. C. Integrating compression and execution in columnoriented*

*database systems. Proc. of SIGMOD (2006).*

*BURROWS, M. The Chubby lock service for looselycoupled*

*distributed systems. In Proc. of the 7th OSDI*

*(Nov. 2006).*

*GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The*

*Google \_le system. In Proc. of the 19th ACM SOSP (Dec.*

*2003), pp. 29.43.*

*STONEBRAKER, M. The case for shared nothing.*

*Database Engineering Bulletin 9, 1 (Mar. 1986), 4.9.*

*STONEBRAKER, M., AOKI, P. M., DEVINE, R.,*

*LITWIN, W., AND OLSON, M. A. Mariposa: A new architecture*

*for distributed data. In Proc. of the Tenth ICDE*

*(1994), IEEE Computer Society, pp. 54.65.*

*SYBASE.COM. www.sybase.com/products/databaseservers/*

*sybaseiq. Product page.*