**Bigtable vs. Pnuts**

1. **Bigtable**

**▪What were the key goals of the paper?**

This paper describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format and has successfully provided a flexible, high-performance solution for many Google products, and describe the design and implementation of Bigtable.

**▪A brief summary of main ideas.**

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. 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.

**▪Was this a good paper? Did it achieve what it set out to do?**

Yes, this is an excellent paper which presented a whole design of Bigtable. It mainly focus on two key point. First, it focus on the design of Bigtable. This paper demonstrate the detail of data model, design and implement of Bigtable. Second, it shows the efficiency on test and real products. This paper describe the efficiency of Bigtable on some test scenario and some deployment of Bigtable on some Google products such as Google Analytics and Google Earth, and get some useful advises from its and discuss some lessons that learned in designing and supporting Bigtable.

**▪What would you do differently?**

**▪What challenges you see ahead in the area.**

Bigtable does not currently support general transactions across row keys, although it provides an interface for batching writes across row keys at the clients.

**▪Key features**

1. **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

Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

**1.1 Rows**

The row keys in a table are arbitrary strings. 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.

**1.2 Column Families**

Column keys are grouped into sets called column families, which form the basic unit of access control. 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:qualifier.

**1.3 Timestamps**

Each cell in a Bigtable can contain multiple versions of the same data; these versions are indexed by timestamp. Different versions of a cell are stored in decreasing timestamp order, so that the most recent versions can be read first. 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).

1. **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. Bigtable supports several other features that allow the user to manipulate data in more complex ways. First, Bigtable supports single-row transactions, which can be used to perform atomic read-modify-write sequences on data stored under a single row key. Second, Bigtable allows cells to be used as integer counters. Finally, Bigtable supports the execution of client-supplied scripts in the address spaces of the servers.

1. **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 files. 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 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.

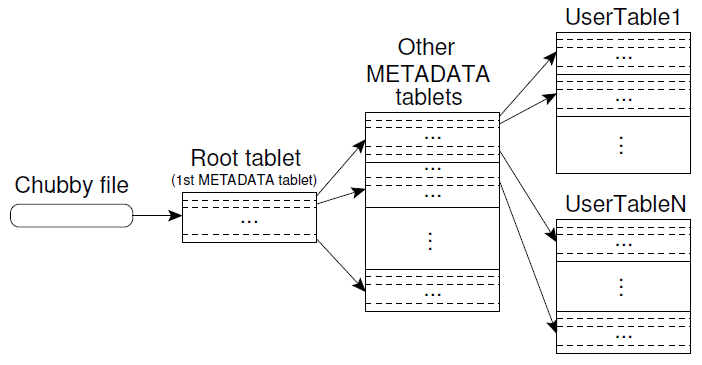
Bigtable relies on a highly-available and persistent distributed lock service called Chubby. Bigtable uses Chubby for a variety of tasks: to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data; to discover tablet servers and finalize tablet server deaths; to store Bigtable schema information (the column family information for each table); and to store access control lists.

1. **Implementation**

The Bigtable implementation has three major components: a library that is linked into every client, one master server, and many tablet servers.

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 [17, 21], 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.

**4.1 Tablet Location**



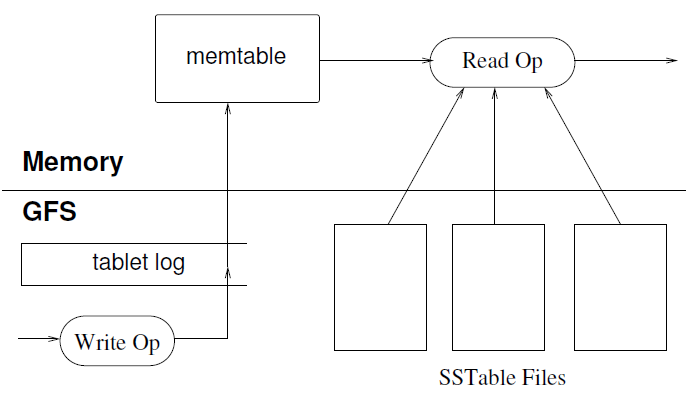
The first level is a file stored in Chubby that contains the location of the root tablet. The root tablet contains the location of all tablets in a special METADATA table. Each METADATA tablet contains the location of a set of user tablets. The root tablet is just the first tablet in the METADATA table, but is treated specially—it is never split–to ensure that the tablet location hierarchy has no more than three levels. 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.

**4.2 Tablet 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 file in a specific Chubby directory. The master monitors this directory (the servers’ directory) to discover tablet servers.

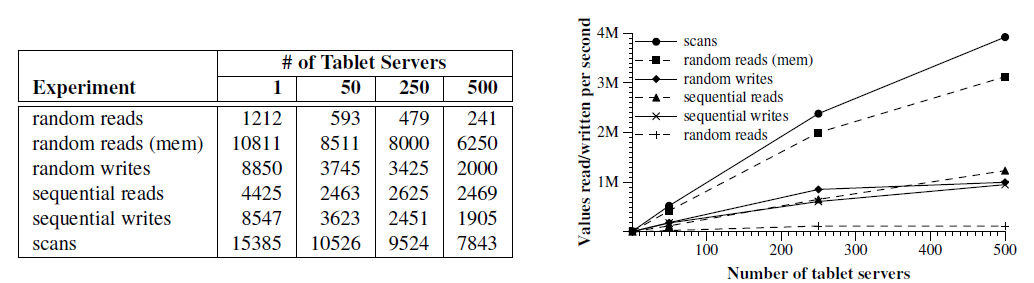
**4.3 Tablet Serving**

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.



1. **Performance Evaluation**

We set up a Bigtable cluster with N tablet servers to measure the performance and scalability of Bigtable as N is varied. N client machines generated the Bigtable load used for these tests.



1. **Pnuts**

**▪What were the key goals of the paper?**

This paper describe PNUTS, a massively parallel and geographically distributed database system for Yahoo!’s web applications. This paper describe the motivation for PNUTS and the design and implementation of its table storage and replication layers, and then present experimental results.

**▪A brief summary of main ideas.**

PNUTS provides data storage organized as hashed or ordered tables, low latency for large numbers of concurrent requests including updates and queries, and novel per-record consistency guarantees. It is a hosted, centrally managed, and geographically distributed service, and utilizes automated load-balancing and failover to reduce operational complexity. The foremost requirements of a web application are scalability, consistently good response time for geographically dispersed users, and high availability. At the same time, web applications can frequently tolerate relaxed consistency guarantees. In this paper, we present the design and functionality of PNUTS, as well as the key protocols and algorithms used to route queries and coordinate the growth of the massive data store.

**▪Was this a good paper? Did it achieve what it set out to do?**

Yes, this is an excellent paper which presented a whole design of Pnuts. It not only illustrates the design of Pnuts but also shows the deployment of Pnuts on some Yahoo! products such as User Database and Social Applications.

**▪What would you do differently?**

Recent interest in massively scalable databases has produced several systems, each optimized for a different point in the design space. Google’s BigTable [8] provides record oriented access to very large tables, but to our knowledge there have been no publications describing support for geographic replication, secondary indexes, materialized views, the ability to create multiple tables, and hash-organized tables. Amazon’s Dynamo [12] is a highly-available system that provides geographic replication via a gossip mechanism, but its eventual consistency model does not adequately support many applications, and it does not support ordered tables. Other large scale distributed storage systems include Amazon’s S3 and SimpleDB services, and Microsoft’s CloudDB initiative, but there is little information publicly available about the architecture of these systems.

**▪What challenges you see ahead in the area.**

**1 Indexes and Materialized Views**

In order to support efficient query processing, it is often critical to provide secondary indexes and materialized views. An index/view maintainer will listen to the stream of updates from message broker, and generate corresponding updates. For example, if a user moves from Wisconsin to California, and we have an index on location, the maintainer will delete the Wisconsin index entry for the user and insert a California index entry for the user. Further research is needed to examine the semantic implications of answering queries using possibly stale indexes and views.

**2 Bundled Updates**

Several customers have expressed a need for an extension of the consistency guarantees we provide. The extension, called bundled updates, provides atomic, no isolated updates to multiple records. That is, all updates in the bundle are guaranteed to eventually complete, but other transactions may see intermediate states resulting from a subset of the updates. For example, if Alice and Bob accept a bi-directed social network connection, we need to update both Alice’s and Bob’s records to point to the other user. Both updates need to complete (and the application writer would prefer not to check and retry to ensure this, as in the current system) but it is not critical to provide serializability; it is ok if Alice is temporarily a friend to Bob but not vice-versa. The challenges in implementing bundled updates are to ensure the timeline consistency guarantees described in Section 2.2 when the updates in the bundle are asynchronously and independently applied, and to provide a convenient mechanism for the client to determine when all updates in the bundle have completed.

**3 Batch-Query Processing**

Although PNUTS is optimized for web OLTP workloads, we believe that it can also serve as a data store for batch and bulk processing, such as that provided by MapReduce or Pig. This requires further investigation of how a scan-oriented bulk workload interacts with a seek-oriented serving workload. It may be necessary to separate PNUTS replicas into “batch” and “serving,” and optimize them separately for the different workloads. Also, parallel batch systems optimize their execution based on the current location of data, and therefore we may need to provide hooks for accessing tablets directly, bypassing routers.

**▪Key features**

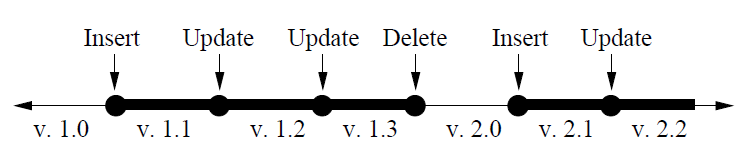
1. **Data Model**

PNUTS presents a simplified relational data model to the user. Data is organized into tables of records with attributes. PNUTS allows applications to declare tables to be hashed or ordered, supporting both workloads efficiently

1. **Consistency Model**

PNUTS provides a consistency model that is between the two extremes of general serializability and eventual consistency. Our model stems from our earlier observation that web applications typically manipulate one record at a time, while different records may have activity with different geographic locality. We provide per-record timeline consistency: all replicas of a given record apply all updates to the record in the same order. An example sequence of updates to a record is shown in this diagram:

This model is implemented as follows. One of the replicas is designated as the master, independently for each record, and all updates to that record are forwarded to the master. The master replica for a record is adaptively changed to suit the workload – the replica receiving the majority of write requests for a particular record becomes the master for that record. The record carries a sequence number that is incremented on every write. As shown in the diagram, the sequence number consists of the generation of the record (each new insert is a new generation) and the version of the record (each update of an existing record creates a new version). Note that we (currently) keep only one version of a record at each replica.



1. **API**

Using this per-record timeline consistency model, we support a whole range of API calls with varying levels of consistency guarantees.

**Read-any:** Returns a possibly stale version of the record.

**Read-critical (required version):** Returns a version of the record that is strictly newer than, or the same as the required version.

**Read-latest:** Returns the latest copy of the record that reflects all writes that have succeeded. Note that read-critical and read-latest may have a higher latency than read-any if the local copy is too stale and the system needs to locate a newer version at a remote replica.

**Write:** This call gives the same ACID guarantees as a transaction with a single write operation in it. This call is useful for blind writes, e.g., a user updating his status on his profile.

**Test-and-set-write(required version):** This call performs the requested write to the record if and only if the present version of the record is the same as required version

.

1. **SYSTEM ARCHITECTURE**

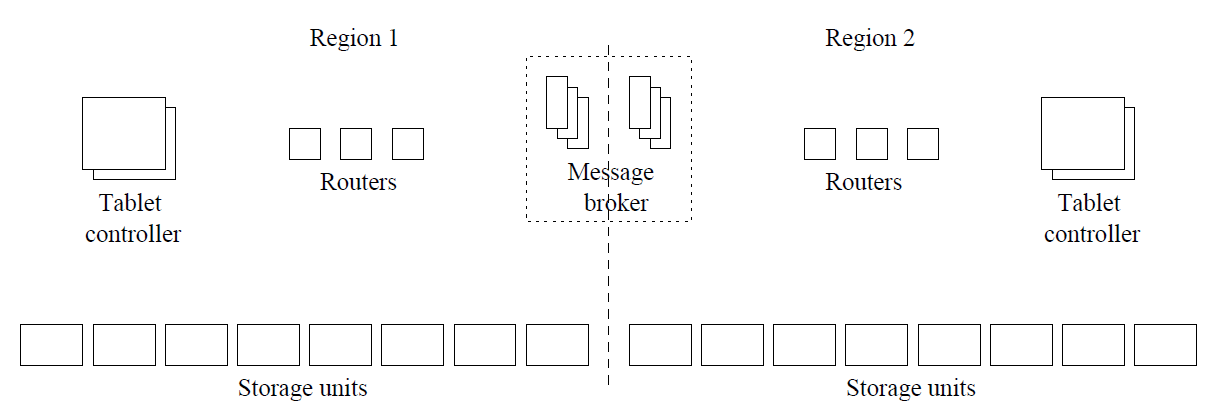


Figure 1 shows the system architecture of PNUTS. The system is divided into regions, where each region contains a full complement of system components and a complete copy of each table. Regions are typically, but not necessarily, geographically distributed. A key feature of PNUTS is the use of a pub/sub mechanism for both reliability and replication. Three components in Figure 1 are primarily responsible for managing and providing access to data tablets: the storage unit, the router, and the tablet controller. In order to determine which storage unit is responsible for a given record to be read or written by the client, we must first determine which tablet contains the record, and then determine which storage unit has that tablet. Both of these functions are carried out by the router.

Data tables are horizontally partitioned into groups of records called tablets. Tablets are scattered across many servers; each server might have hundreds or thousands of tablets, but each tablet is stored on a single server within a region.

1. **Replication and Consistency**

Our system uses asynchronous replication to ensure low latency updates. We use the Yahoo! message broker, a publish/subscribe system developed at Yahoo!, both as our replacement for a redo log and as our replication mechanism.

*10.1 Yahoo! Message Broker*

Yahoo! Messsage Broker (YMB) is a topic-based pub/sub system, which together with PNUTS, is part of Yahoo!’s Sherpa data services platform. Data updates are considered “committed” when they have been published to YMB. We are able to use YMB for replication and logging for two reasons. First, YMB takes multiple steps to ensure messages are not lost before they are applied to the database. Second, YMB is designed for wide-area replication: YMB clusters reside in different, geographically separated datacenters, and messages published to one YMB cluster will be relayed to other YMB clusters for delivery to local subscribers.

*3.2.3 Recovery*

Recovering from a failure involves copying lost tablets from another replica. Copying a tablet is a three step process. First, the tablet controller requests a copy from a particular remote replica (the “source tablet”). Second, a “checkpoint message” is published to YMB, to ensure that any in-flight updates at the time the copy is initiated are applied to the source tablet. Third, the source tablet is copied to the destination region.

1. **Hosted Database Service**

PNUTS is a hosted, centrally-managed database service shared by multiple applications. The system adapts by automatically shifting some load to the new servers. In all cases, adding more servers adds more of the bottleneck resource. When servers have a hard failure, it automatically recover by copying data (from a replica) to other live servers (new or existing), carrying out little or no recovery on the failed server itself.

This hosted model introduces several complications that must be dealt with. First, different applications have different workloads and requirements, even within our relatively narrow niche of web serving applications. Therefore, the system must support several different workload profiles, and be automatically or easily tunable to different profiles. For example, our mastership migration protocol adapts to the observed write patterns of different applications. Second, we need performance isolation so that one heavyweight application does not negatively impact the performance of other applications. In our current implementation, performance isolation is provided by assigning different applications to different sets of storage units within a region.

1. **Performance Evaluation**

. 