Bigtable: A Distributed Storage System for Structured Data

Bigtable is a distributed storage system developed by Google for managing structured data, designed to scale to petabytes of data across thousands of commodity servers. It's utilized by over sixty Google products for a variety of demanding workloads, from throughput-oriented batch processing to latency-sensitive data serving. While sharing commonalities with databases, Bigtable provides a simple data model allowing clients dynamic control over data layout and format, indexing data using arbitrary string-based row and column names. Clients can manage data locality through schema choices and control whether to serve data from memory or disk. This paper outlines Bigtable's data model, client API, underlying Google infrastructure, implementation details, performance metrics, and examples of use at Google, offering insights and lessons learned from its design and support.

Bigtable is structured as a sparse, multi-dimensional sorted map, indexed by row key, column key, and a timestamp. This design is chosen to cater to various use-cases such as managing a large collection of web pages, where URLs serve as row keys, different aspects of web pages as column names, and the timestamps of fetches as the index for page contents.

In Bigtable, the row keys are arbitrary strings with all data operations under a single row key being atomic. The data is maintained in lexicographic order by row key, and the range for a table is dynamically partitioned. Each of these row ranges, or 'tablets', is a unit of distribution and load balancing, optimizing short row range reads. To ensure efficient data access, clients select their row keys to achieve good data locality. For example, web pages in the same domain are grouped together by reversing the hostname components of the URLs.

Column keys are grouped into sets known as column families, forming the basic unit of access control. All data in a column family is typically of the same type. Column keys are named using the syntax of family:qualifier. Each cell in the system can contain multiple versions of the same data, indexed by timestamp. To manage this versioned data, Bigtable supports settings for automatic garbage collection of cell versions. Clients can specify to keep only the last 'n' versions or only recent versions of a cell.

The Bigtable API allows creation and deletion of tables and column families, as well as modification of cluster, table, and column family metadata, such as access control rights. It facilitates writing or deleting values, looking up values from individual rows, and iterating over a subset of the data in a table. Bigtable also supports single-row transactions, allowing atomic read-modify-write sequences on data under a single row key. Another feature includes the use of cells as integer counters. Additionally, Bigtable supports client-supplied scripts written in a language called Sawzall for data processing, providing functionalities like data transformation, arbitrary expression-based filtering, and summarization. Bigtable can also be integrated with MapReduce, a framework for running large-scale parallel computations, enabling a Bigtable to be used as both an input source and an output target for MapReduce jobs.

Bigtable is constructed on various Google infrastructure components, including the distributed Google File System (GFS) for storing log and data files and a cluster

management system for managing resources, scheduling jobs, handling machine failures, and monitoring machine status. The Google SSTable file format is used internally to store Bigtable data, which provides a persistent, ordered immutable map from keys to values. The SSTable contains a sequence of blocks, with an in-memory block index for location purposes, enabling efficient lookups and scans. Bigtable also relies on a distributed lock service called Chubby for tasks like ensuring only one active master exists, storing bootstrap locations, discovering tablet servers, storing Bigtable schema information, and maintaining access control lists. While Bigtable becomes unavailable if Chubby is out of service for a long duration, recent measurements show the impact of Chubby unavailability on Bigtable to be minimal.

The Bigtable implementation consists of three major components: a library linked into every client, one master server, and multiple tablet servers that can be dynamically added or removed to accommodate varying workloads. The master server assigns tablets to tablet servers, detects changes in tablet servers, balances load, handles garbage collection of files in GFS, and manages schema changes. Each tablet server manages a set of tablets, handling read and write requests and splitting tablets that become too large. Client data doesn't pass through the master, but is directly communicated with the tablet servers, resulting in a lightly loaded master. A Bigtable cluster stores several tables, with each table consisting of tablets that contain all data associated with a row range. Tables are automatically split into multiple tablets as they grow, maintaining a size of approximately 100-200 MB each by default.

Bigtable uses a three-level hierarchy, similar to a B+-tree, to store tablet location information. The first level is a file in Chubby that contains the location of the root tablet. This root tablet holds 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, which is the first tablet in the METADATA table, is treated specially and never split to ensure that the tablet location hierarchy remains no more than three levels. The METADATA table stores the location of a tablet under a row key that's an encoding of the tablet's table identifier and its end row. The client library caches tablet locations and employs an algorithm to navigate the location hierarchy in case of a miss or stale data. Additional secondary information such as a log of all events pertaining to each tablet is also stored in the METADATA table, aiding in debugging and performance analysis.

In Bigtable, 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 servers. When a tablet is unassigned, and there's a server with sufficient room available, the master assigns the tablet to that server. Tablet servers use Chubby, a distributed lock service, to maintain their identity and status. When a tablet server starts, it creates and acquires an exclusive lock on a uniquely-named file in a specific Chubby directory. If a server loses its lock, for example due to a network partition, it will stop serving its tablets. If it can't reacquire the lock because the file no longer exists, it will terminate itself. The master is responsible for detecting unresponsive tablet servers and reassigning their tablets.

The persistent state of a tablet is stored in Google File System (GFS). Updates are committed to a commit log that stores redo records. Recent updates are stored in memory in a sorted buffer called a memtable, while older updates are stored in a sequence of SSTables. To recover a tablet, a server reads its metadata from the METADATA table, which contains a list of SSTables that make up a tablet and a set of redo points. The server reads the indices of the SSTables into memory and reconstructs the memtable by applying all the updates that have committed since the redo points.

Authorization checks are performed on both write and read operations to ensure proper access control.

To bolster performance, availability, and reliability, several modifications were implemented in the system. A feature called 'locality groups' was introduced which allowed users to group multiple column families together, enhancing data reading efficiency. SSTables, serving as data storage units, were made immutable to facilitate read/write operations and mitigate concurrency issues. Alongside, compression techniques were applied to SSTables, saving significant storage space while maintaining fast encoding and decoding speeds.

Furthermore, two caching levels were incorporated to improve read performance - the Scan Cache, storing key-value pairs, and the Block Cache, retaining SSTable blocks read from GFS. Bloom filters were utilized to minimize disk access, consequently reducing the number of disk seeks required for reading operations. Moreover, a single commit log was maintained for each tablet server, resulting in notable performance gains but complicating recovery processes. However, the challenge of recovery was addressed by employing techniques that expedited tablet recovery and enabled swift tablet splitting.

The performance and scalability of Bigtable were tested using a cluster setup with N tablet servers, with N varying. Each tablet server was configured with 1 GB memory and wrote to a GFS cell consisting of 1786 machines, each having two 400GB IDE hard drives. The same number of client machines as tablet servers generated load for the tests, ensuring that clients were not a bottleneck. The machines were powered by two dual-core Opteron 2GHz chips, sufficient physical memory to handle all running processes, and a single gigabit Ethernet link. All machines were housed in the same hosting facility, ensuring a round-trip time between any pair of machines of less than a millisecond.

Several benchmarks, including sequential write, random write, sequential read, random read, and scans, were run on the setup. For random reads from memory, data per tablet server was reduced from 1GB to 100MB to ensure comfortable accommodation within the available memory. The performance was then analyzed based on the number of operations per second per tablet server, and aggregate operations per second. There were observed differences in the performance of random reads, sequential reads, random and sequential writes, with random reads being slower due to the transfer of a 64KB SSTable block from GFS for each 1000-byte value read. The performance scaled significantly with the increase of tablet servers, although not linearly due to load imbalance and shared network saturation.

Google uses Bigtable, its proprietary data storage system, in various services. One such application is Google Analytics, where Bigtable plays a critical role in analyzing web traffic. This service uses two primary tables: the raw click table and the summary table. The raw click table stores each end-user session, while the summary table uses the raw click data to generate aggregate statistics. By managing and interpreting this extensive data, Google Analytics provides insights into visitor behavior and site performance.

Another application of Bigtable is in Google Earth. This service employs Bigtable for managing raw imagery and serving client data. The preprocessing pipeline stores raw images in a table, where they're cleaned and consolidated into final serving data. Another table is used by the serving system to index data stored in the Google File System (GFS). This setup allows users to navigate and annotate high-resolution

satellite imagery, showcasing the power of Bigtable in managing complex data processing tasks.

Personalized Search, another Google service, also utilizes Bigtable. This service records user interactions across various Google properties and stores this information in Bigtable. Each user's data is neatly organized, with separate column families designated for different types of actions. The data, once processed, can be used to tailor search results based on individual users' historical patterns. This application showcases how Bigtable's versatility and adaptability can be harnessed to manage large, diverse datasets, and provide personalized experiences to users.