Google Big Table Prototype	
Abhishek Modi	2011B4A7331P
Divyanshu Goyal	2011B4A7795P

PROBLEM DOMAIN

We aim to design a prototype database similar to Google's Big Table which is distributed, scalable and fault tolerant. Our design is based on the Google's BigTable paper.Google's Big Table is distributed, scalable and highly fault tolerant system which uses GFS(Google File System) for file management and Zookeeper for node management. We have also borrowed some ideas from Cassandra on consistent Hashing. We have named our system HBaseProto.

DATA MODEL

HBaseProto is a sparse,distributed, persistent sorted map which provides support for creating different tables, maintaining table schema, and fetching, deleting,updating data in the database. The map is indexed by a **key** and each value in the map is a string. In our design multi column data can be easily stored in the system by serializing the HashMap Object holding the data and writing it to a file. Column Families and Column name information is stored in a separate directory called Schemas which hold schema files for each table created on our prototype.

HBaseProto maintains data in lexicographic ordered by row key. The row range for a table is dynamically partitioned. Each row range is called a tablet. As a result, reads of short row ranges are efficient and typically require communication with only a small number of machines.

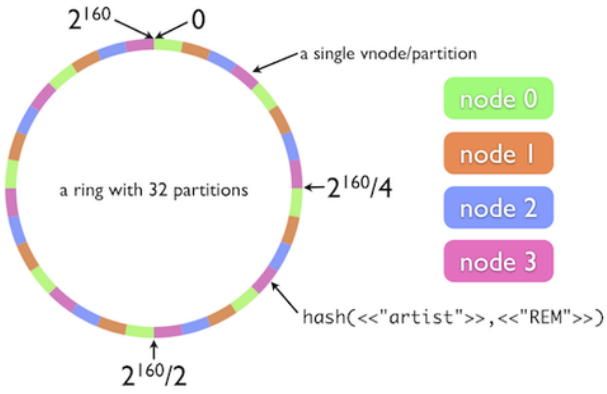
DESIGN

1. Data Distribution and Load Balancing

We have used Consistent Hashing which is used in distributed storage systems like Amazon Dynamo, memcached etc, for dividing the data between machines.

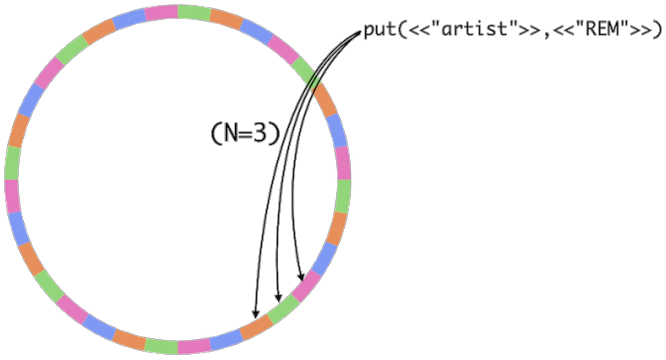
In consistent hashing, the servers, as well as the keys, are hashed, and looked up using this HashMap. The hash space is large, and is treated as if it wraps around to form a circle - hence forming a *hash ring*. The process of creating a hash for each server is equivalent to placing it at a point on the circumference of this circle. When a key needs to be looked up, it is hashed, which again corresponds to a point on the circle. In order to find its server, one then simply moves round the circle clockwise from this point until the next server is found. If no server is found from that point to end of the hash space, the first server is used - this is the "wrapping round" that makes the hash space circular.

This may leave some servers with a disproportionately large space before them, and this will result in greater load on the first server in the cluster and less on the remainder. To overcome this, we use 10 virtual nodes for each Tablet Server to distribute the load evenly.



2. Data Replication:

Every data entered in our prototype is replicated among two nodes. So that if one goes down other can be used. For data replication and maintaining map for the replicated data, the hash ring structure from the previous phase is used. For each data entry we replicate it to next two nodes next to its hash value in the ring so we can easily figure out the node to which data was assigned based on its hash value. So even if master goes and loses all of its mappings it can easily recover and figure out the node to which a particular key was assigned
(if assigned) based on the hash value.



3. Data Storage

Within a particular node, we store the data in a in-memory map. When the size of the map exceeds a threshold, we store the map in the filesystem on disk (known as tablets). We also maintain a mapping of the key ranges for each tablet stored in the disk, which helps in

efficient querying. While querying a row by key, we check these key ranges to narrow down our search space.

4. Tablet Server Performance

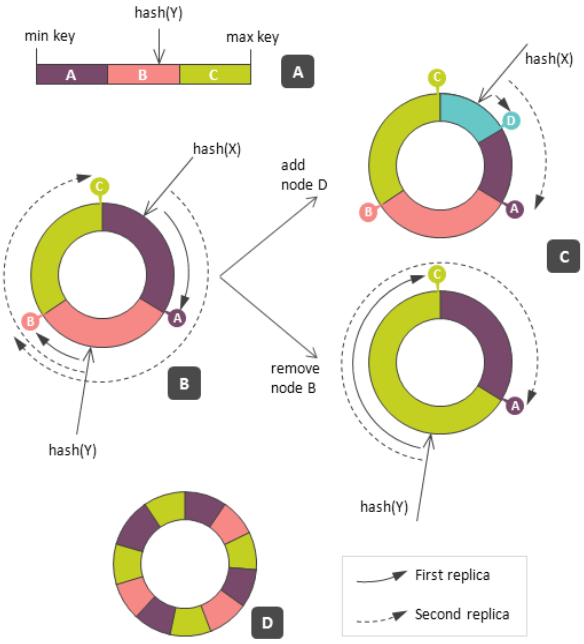
Tablet Servers use fork on request model to handle client request, we fork a separate thread for each new client. This ensures low latency from the tablet servers. Too many client requests may lead to spawning of huge number of threads which in turn may lead to degradation in performance. So as suggested optimization we can implement tablet servers as a preforked server model which creates a pool of threads initially to service client requests and don't spawn any new threads if number of clients becomes large which may lead to high latency for user response time but will ensure static system performance. Also since the rows are sorted by keys, lookup is fast and efficient.

4. Client

We have also implemented Client which can create, update or delete tables in HBaseProto for testing purpose. In the prototype, the client is contacting the server for its requests. But since we are using key hashes, once the client has the hash ring, it can contact the Tablet Servers directly. So the prototype can be easily extended to reduce the load on the server.

5. Fault Tolerance

We have implemented fault tolerance for both Tablet Servers and Master. For tablet servers we are replicating the data to the adjacent Tablet Server in the ring. The server maintains a persistent connection to each Tablet Server. If a Tablet Server goes down, the connection is terminated and a Connection Termination event handler is invoked. This event handler removes the Tablet Server from the hash ring. Now the future requests for this Tablet Servers keys would be served by the appropriate nodes in the hash ring.



The master maintains two copy of mappings for tables created in our prototype, one in main memory and other on disk. So if a master goes down and loses all its mappings in main memory, it can read table information from local disk. For Tablet Server key-range mapping, we use hash ring structure, since every tablet server node is hashed to multiple positions on the ring and key is also hashed to get that position, so master can easily figure out the Tablet Server to which a particular key has been mapped based on the hash value.

6. Architecture diagram