Lecture 6: BigTable, HBase, Hive

CSCI4180 (Fall 2013)

Patrick P. C. Lee

Outline

- Goal: understand how to manage big data in a systematic way
  - MapReduce: for computation
  - HDFS: for storage
  - Any system designed for data management, just like a database?
- This lecture introduces several systems for big-data management:
  - BigTable
  - HBase
  - Hive

Motivation

- Question: how do you provide real-time read/write random-access to big data?
- MapReduce is designed for batch processing rather than real-time access
- We need some database-like systems to access data efficiently
- Challenge: commercial databases are too sophisticated → they are not scalable

BigTable

- **BigTable** is a distributed storage system for managing structured data
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Built by Google, for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for all of Google's products

BigTable: Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data

Why Not Using Commercial DBs?

Not scalable

- Scale is too large for most commercial databases
- Even if it weren't, cost would be very high
  - Building internally means system can be applied across many projects for low incremental cost
- Low-level storage optimizations help performance significantly
  - Much harder to do when running on top of a database layer

Objectives

- Want asynchronous processes to be continuously updating different pieces of data
  - Want access to most current data at any time
- Need to support: Hadoop: Write once, read many
  - Very high read/write rates (millions of ops per second)
  - Efficient scans over all or interesting subsets of data
  - Efficient joins of large one-to-one and one-to-many datasets
- Often want to examine data changes over time
  - E.g. Contents of a web page over multiple crawls

BigTable: Design Goals

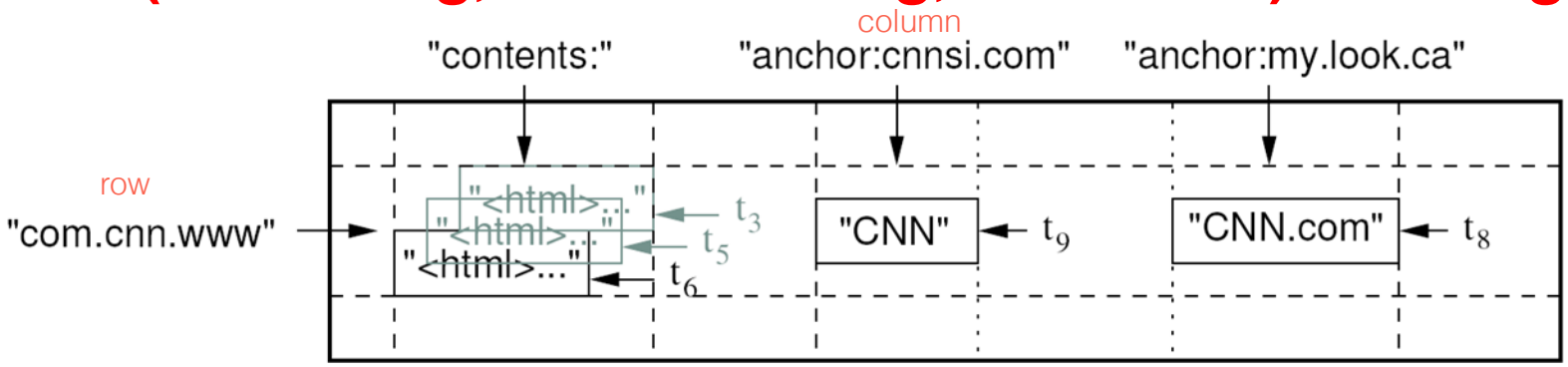
- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabyte of disk-based data
  - Millions of reads/writes per second, efficient scans
- Self-managing
  - Servers can be added/removed dynamically
  - Servers adjust to load imbalance

BigTable: In a Nutshell

- BigTable resembles a database
- BigTable does not support a full relational model
- Data is indexed using row and column names that can be arbitrary strings
- Clients can control locality of data via careful schema design

Data Model

- Map indexed by a row key, column key, and a timestamp
  - **(row:string, column:string, time:int64) → string**



- Webtable example:
  - Row key: reverse URL
  - Column keys: attributes of web page
    - Contents, and anchors that reference the page
    - E.g., CNN's homepage is referenced by cnnsi.com and my.look.ca, so the row has two columns for both anchors
  - Values: contents of web pages (under "contents:"), and other information. Each value is associated with a timestamp

Rows and Columns

- Rows
  - Each read or write of data under a single row key is atomic 
  - Maintained in sorted lexicographic order
    - for efficient row scans
  - Row ranges dynamically partitioned into tablets
- Columns
  - Grouped into column families
  - Column key = *family:qualifier*
  - Column families provide locality hints
    - E.g., anchor is a column family
  - Few column families, but unbounded number of columns
- Timestamp:
  - 64-bit integers → real time in microseconds

Building Blocks

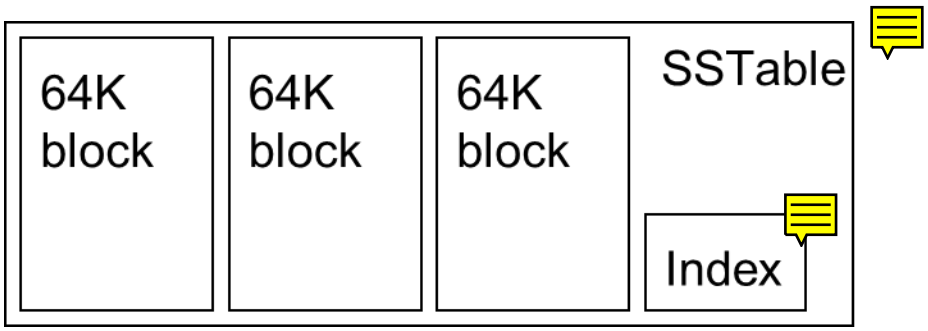
- GFS
  - Raw storage for storing log and data files
- Chubby
  - A distributed lock manager
  - Keeps replicas consistent in presence of failures
- SSTable
  - A file format for storing BigTable data

Chubby

- Basics
  - Distributed lock service
  - Five active replicas, one elected to be the master and actively serve requests
  - Use Paxos algorithm to keep replicas consistent in failure
  - Each directory or file is used as a lock, and reads and writes to a file are atomic
  - Each Chubby client maintains a session with a Chubby service. When a client's session expires, it loses any locks and open handles
- Use Chubby to (1) store bootstrap location, (2) discover tablet servers, (3) store BigTable schema information, (4) store access control lists

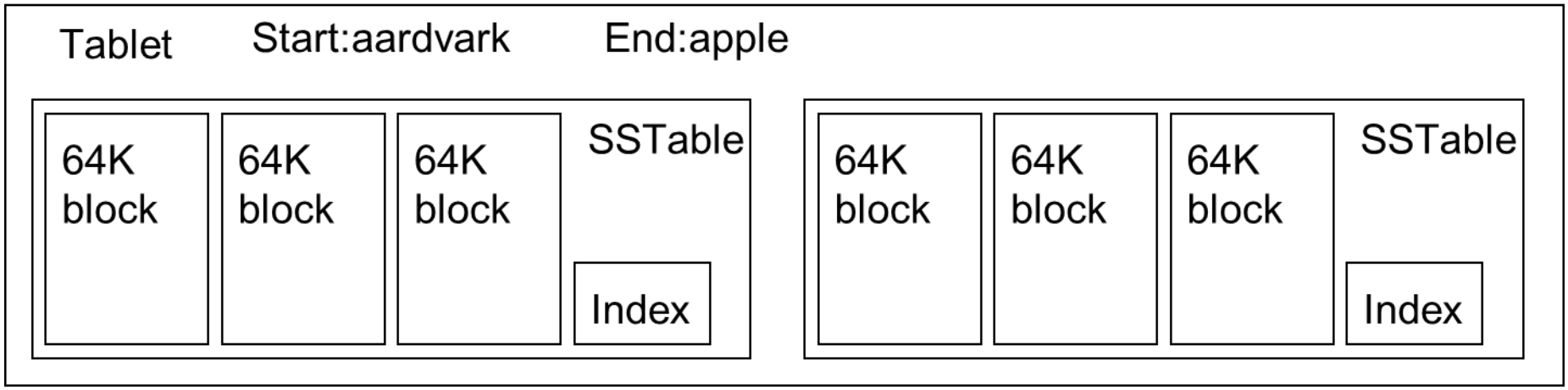
SSTable

- Persistent, ordered immutable map from keys to values
  - Stored in GFS
- Sequence of blocks on disk plus an index for block lookup
  - Can be completely mapped into memory
- Supported operations:
  - Look up value associated with key
  - Iterate key/value pairs within a key range



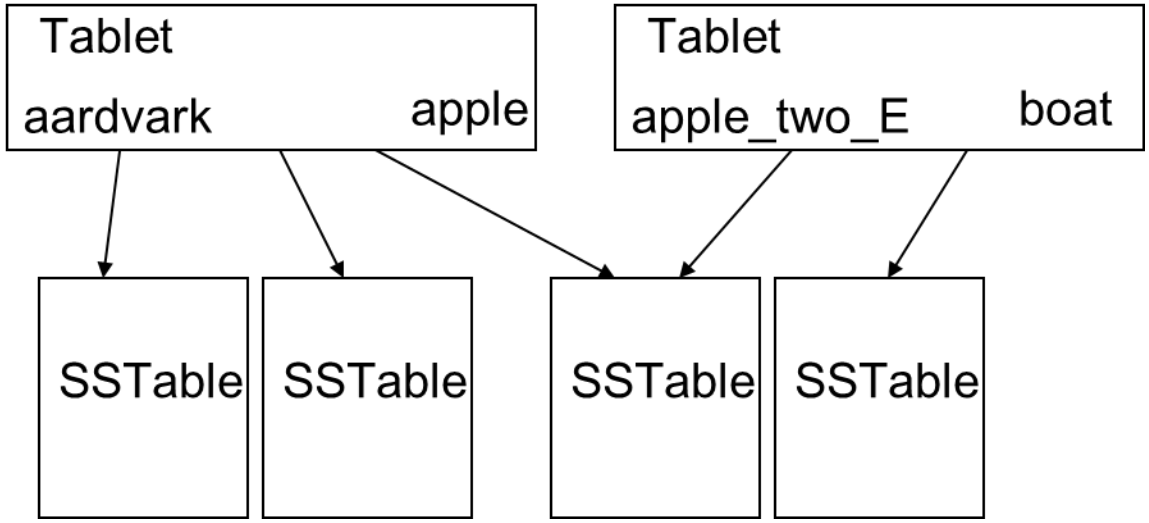
Tablet

- Dynamically partitioned range of rows
- Built from multiple SSTables
- Tablet is a unit of distribution and load balancing



Table

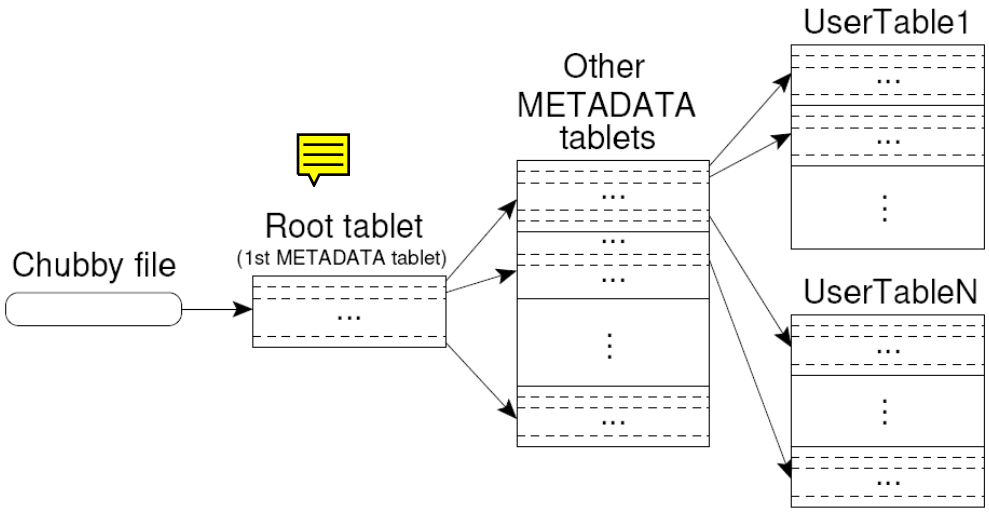
- Multiple tablets make up the table
- SSTables can be shared



BigTable Architecture

- Single-master, distributed storage architecture
- BigTable master:
  - Assigns tablets to tablet servers
  - Detects addition and expiration of tablet servers
  - Balances tablet server load
  - Handles garbage collection
  - Handles schema changes
- Tablet servers:
  - Each tablet server manages a set of tablets
    - Typically between ten to a thousand tablets
    - Each 100-200 MB by default
  - Handles read and write requests to the tablets
  - Splits tablets that have grown too large

Tablet Location

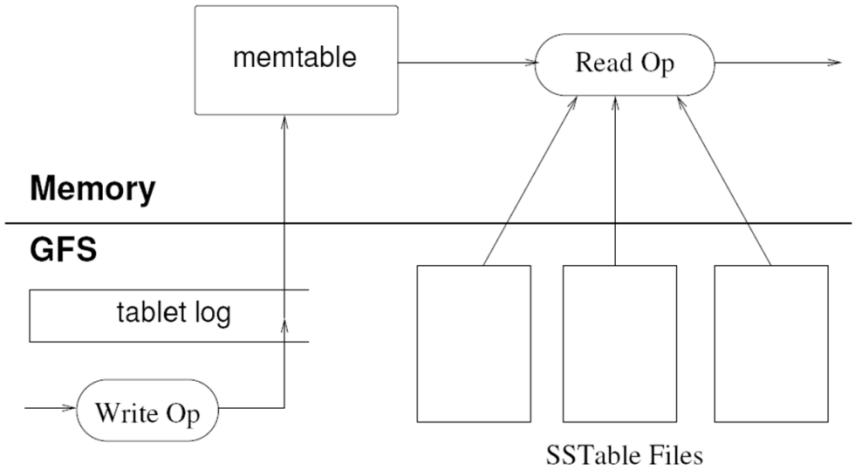


- B+ tree
  - 1st level: root tablet contains location of all tablets
  - 2nd level: metadata tablet contains locations of user tablets
  - 3rd level: user tablets

Tablet Assignment

- Master keeps track of:
  - Set of live tablet servers
  - Assignment of tablets to tablet servers
  - Unassigned tablets
- Each tablet is assigned to one tablet server at a time
  - Tablet server maintains an exclusive lock on a file in Chubby
  - Master monitors tablet servers and handles assignment
- Changes to tablet structure
  - Table creation/deletion (master initiated)
  - Tablet merging (master initiated)
  - Tablet splitting (tablet server initiated)

Tablet Serving



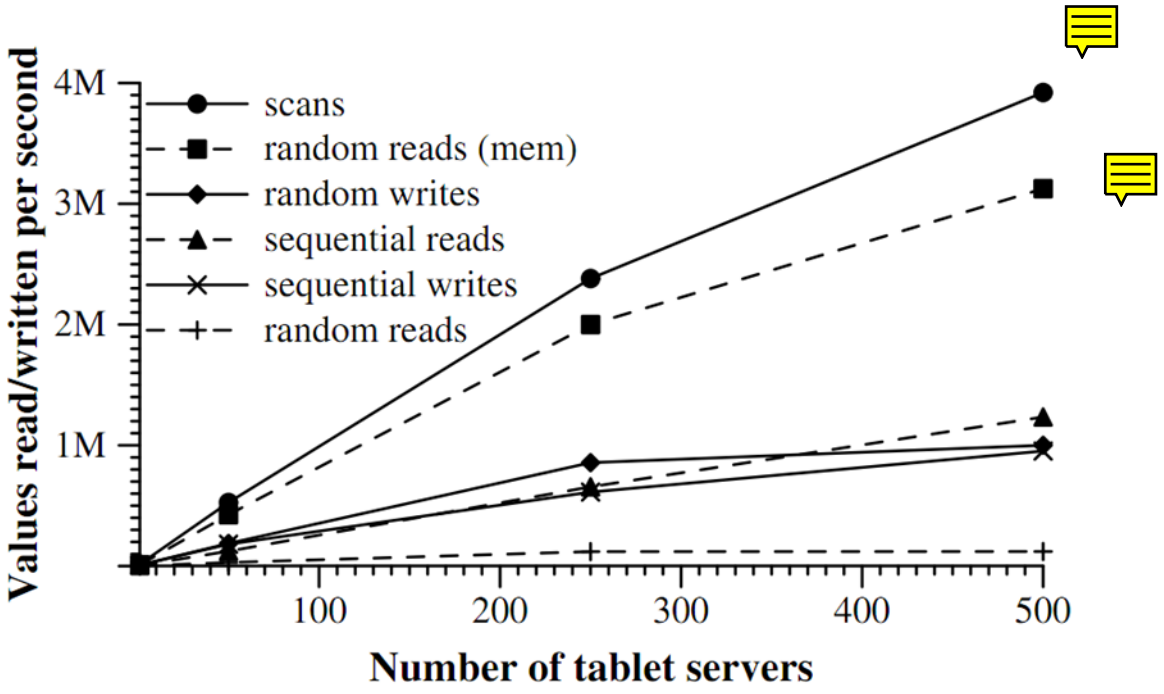
Tablet representation

- Write:
  - Updates committed to a commit log
  - Recently committed updates are stored in memtable
  - Older updates are stored in a sequence of SSTables
- Read:
  - form a merged view of SSTables and memtable
  - read <key-value> pair

Compaction

- Minor compaction
  - Converts the memtable into an SSTable
  - Reduces memory usage and log traffic on restart
- Merging compaction
  - Reads the contents of a few SSTables and the memtable, and writes out a new SSTable
  - Reduces number of SSTables
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data

Performance Evaluation



- Performance of random reads from memory increases by almost a factor of 300
  - Read from tablet's server memory rather than a GFS read
- Why not linear?

Load Imbalance

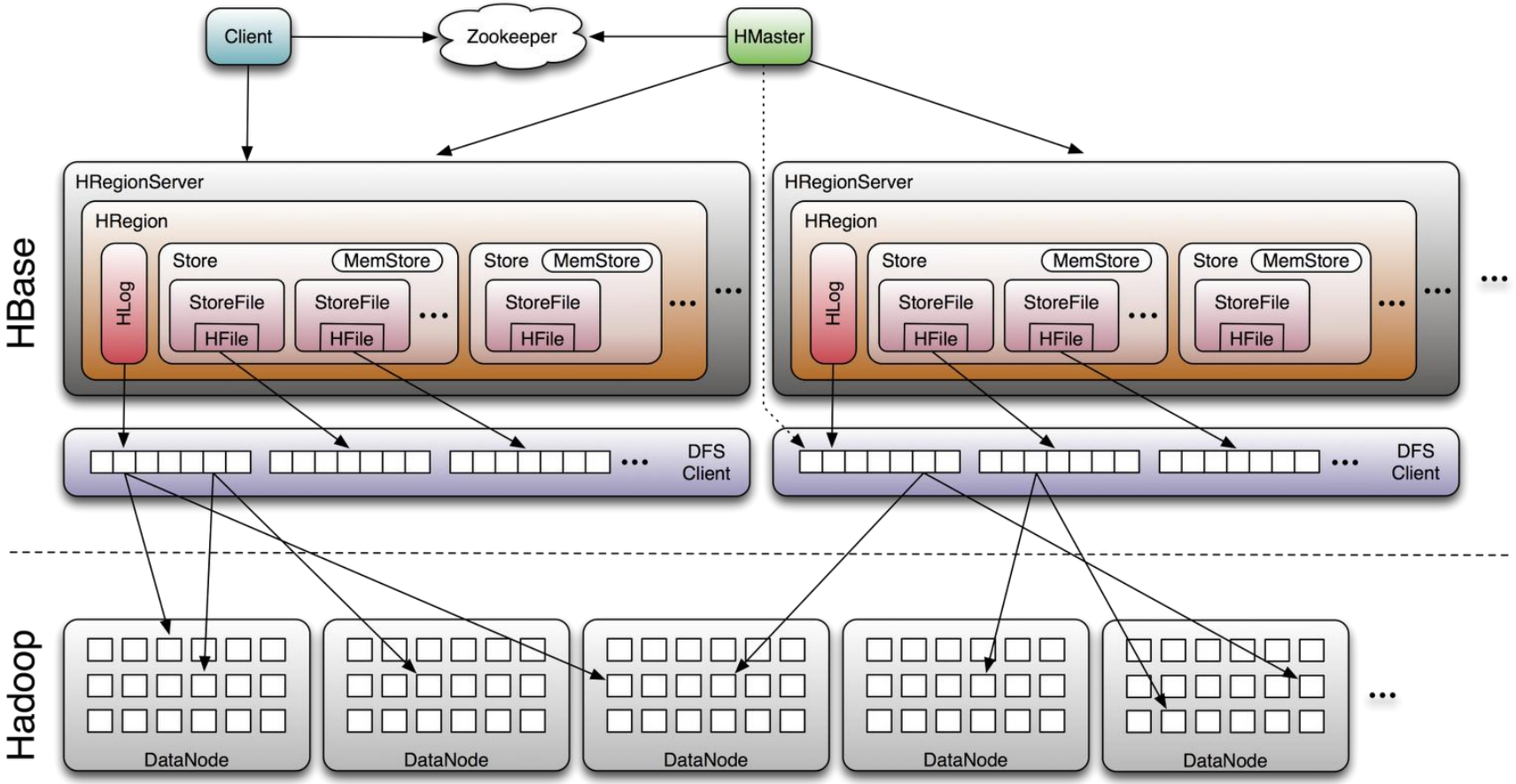
- Competitions with other processes
  - Network
  - CPU
- Rebalancing algorithm does not work perfectly
  - Why?


Summary

- BigTable applications
  - Data source and data sink for MapReduce
  - Google's web crawl
  - Google Earth
  - Google Analytics
- Lessons learned
  - Fault tolerance is hard
    - Example faults: clock skew, hung machines, Chubby bugs
  - Don't add functionality before understanding its use
    - Single-row transactions appear to be sufficient
  - Keep it simple

HBase

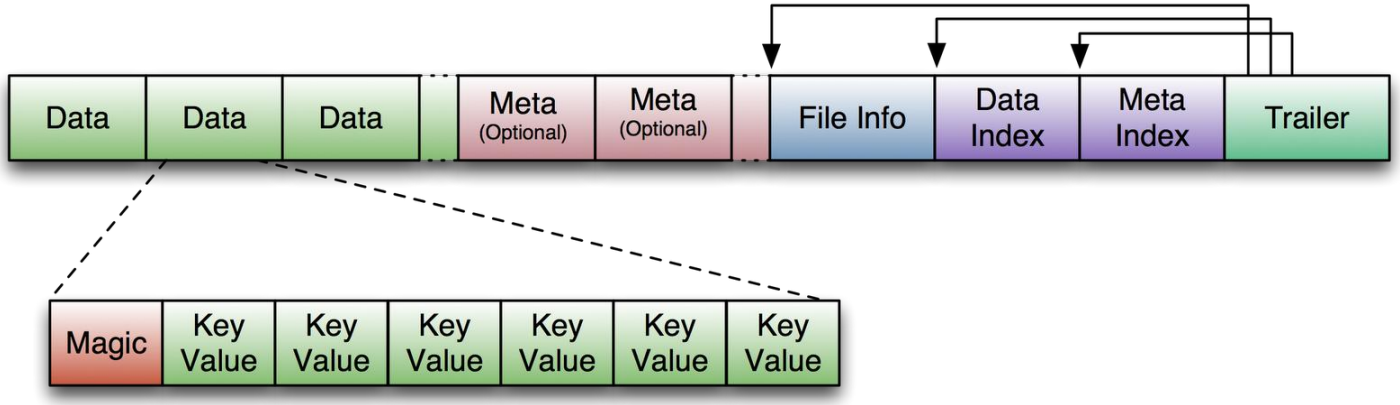
- Open-source clone of BigTable



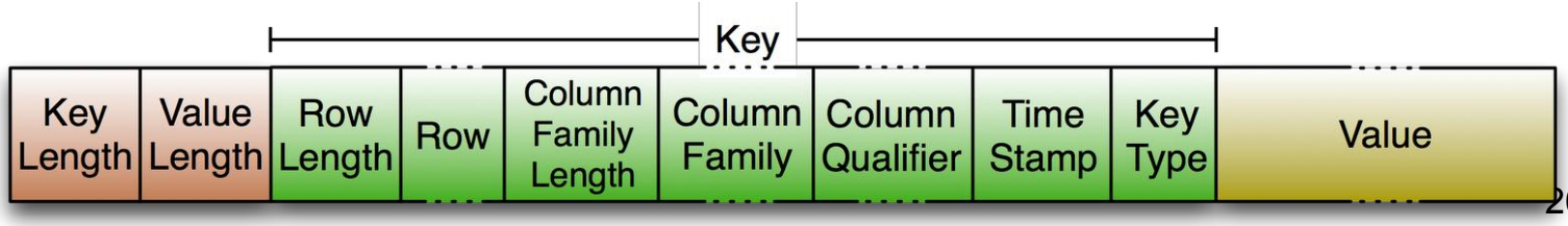
Reference: http://www.larsgeorge.com/2009/10/hbase-architecture-101-storage.html

HBase: Design

- HFile is the actual storage file, created to store HBase's data fast and efficiently
  - Mimics SSTable in BigTable



- Each key value is simply low-level byte array



HBase: Data Model

- HBase stores data in tables
  - Rows and columns
  - Row keys are identifiers, in the form of byte arrays
  - Row columns are grouped in column families
    - Each column family is associated with different qualifiers
  - Tables are partitioned in regions
- HBase uses Zookeeper for locking

HBase Implementation

- HBase master, multiple regionserver slaves
- Operations
  - -ROOT- table holds the list of .META. table regions
  - .META. table holds list of user-space regions

- Writes are appended to a commit log and then cached in memstore
- Reads consult memstore first

Installing HBase

- Install **hbase-0.90.5**
- Use **Java v1.6**

```
[hduser@localhost ~]$ start-hbase.sh
starting master, logging to /usr/local/hbase/logs/hbase-hduser-master-
localhost.localdomain.out

[hduser@localhost ~]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.90.5, r1212209, Fri Dec  9 05:40:36 UTC 2011

hbase(main):001:0>

[hduser@localhost conf]$ stop-hbase.sh
stopping hbase.......
```

HBase Operation Modes

- Standalone mode
  - Default mode
  - Does not use HDFS, but uses local filesystem instead
  - Runs all HBase daemons and a local Zookeeper all up in the same JVM
- Distributed mode
  - Uses HDFS
  - Can be divided into pseudo-distributed and fully distributed as in HDFS

HBase Configurations

```
<!-- hbase-site.xml -->
<configuration>
  <property>
    <name>hbase.rootdir</name>
    <value>file:///app/hbase</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
    <value>/app/hbase/tmp</value>
  </property>
  <property>
    <name>hbase.zookeeper.property.dataDir</name>
    <value>/app/hbase/tmp/zookeeper</value>
  </property>
</configuration>
```

- In distributed mode, replace it with HDFS namenode's address and port (hdfs://localhost:54310).
- Make sure to set hbase.cluster.distributed property to true

```
# hbase-env.sh
export JAVA_HOME=<your JAVA directory>
export HBASE_CLASSPATH=.   # current directory
```

Reference: http://hbase.apache.org/book/standalone_dist.html

Running HBase

- Compile a Java HBase program:

```
javac -classpath /usr/local/hbase/hbase-0.90.5.jar:/usr/local/hadoop/hadoop-core-0.20.203.0.jar Sample.java
```

- Run a Java HBase program:

```
$ hbase Sample <args>
```

HBase in Shell Mode

```
[hduser@localhost ~]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported
commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.90.5, r1212209, Fri Dec  9 05:40:36 UTC 2011

hbase(main):001:0> create 'test', 'data'
0 row(s) in 1.9300 seconds

hbase(main):002:0> list
TABLE
test
1 row(s) in 0.0250 seconds

hbase(main):003:0> put 'test', 'row1', 'data:1', 'value1'
0 row(s) in 0.1970 seconds

hbase(main):004:0> put 'test', 'row2', 'data:2', 'value2'
0 row(s) in 0.0320 seconds

hbase(main):005:0> put 'test', 'row3', 'data:3', 'value3'
0 row(s) in 0.0170 seconds
```

Create table with schema: (table name, column families)

List current table we have

insert rows

HBase in Shell Mode

```
hbase(main):006:0> scan 'test'
ROW           COLUMN+CELL
 row1         column=data:1, timestamp=1328863257474, value=value1
 row2         column=data:2, timestamp=1328863268165, value=value2
 row3         column=data:3, timestamp=1328863277646, value=value3
3 row(s) in 0.0640 seconds

hbase(main):007:0> disable 'test'
0 row(s) in 2.0540 seconds

hbase(main):008:0> drop 'test'
0 row(s) in 1.0890 seconds
```

Display all rows

Disable and drop the table

HBase Applications

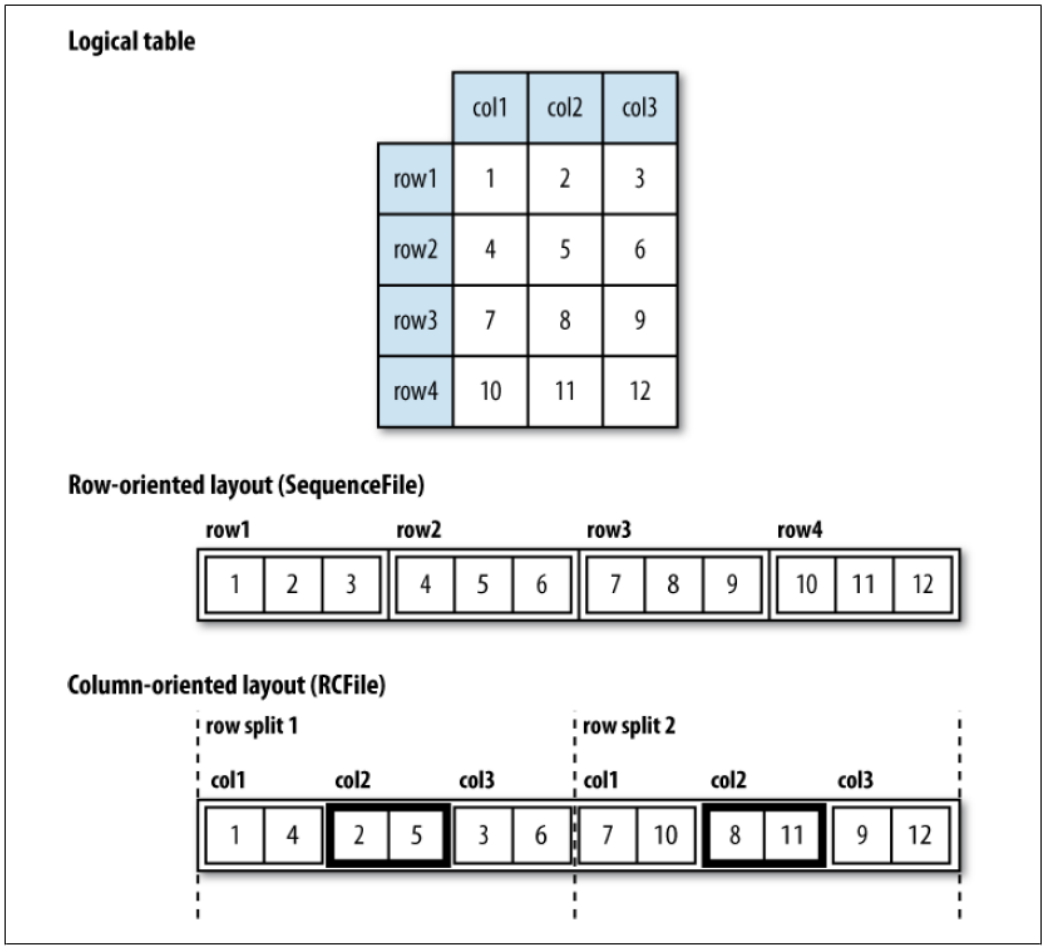
- HBase is used as a source and/or sink in MapReduce jobs
  - Splits are created on region boundaries so each map task handles a single region
- See demo programs:
  - MyUploader.java
    - Uploads people records to HBase
    - Uses MapReduce
  - MyClient.java
    - Queries people's name for a given ID
    - No MapReduce (why?)

HBase vs. RDBMS

- HBase is distributed, column-oriented
  - Column-oriented means data is stored by columns
  - Each column family has its own store in HDFS
  - Emphasis: scalability
- RDBMS is fixed-schema, row-oriented
  - Row-oriented means data is stored by rows
  - Emphasis: strong consistency, complex queries
  - Suitable for small- to medium-volume applications

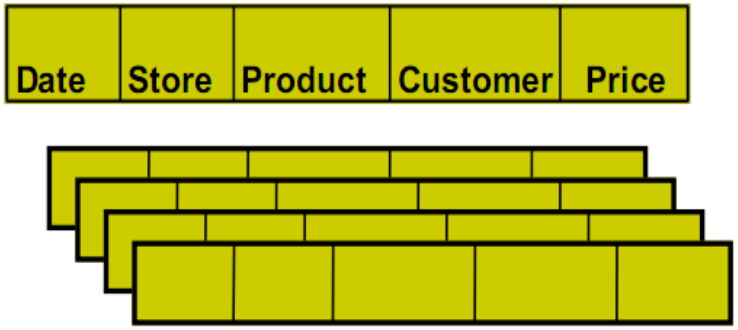
Row-Oriented vs. Column-Oriented

- Column-oriented layout permits columns that are not accessed in a query to be skipped.
- Consider a query that processes only column 2
- Row-oriented
  - Needs 4 reads (each for one row)
- Column-oriented
  - Needs 2 reads (each for one row split)



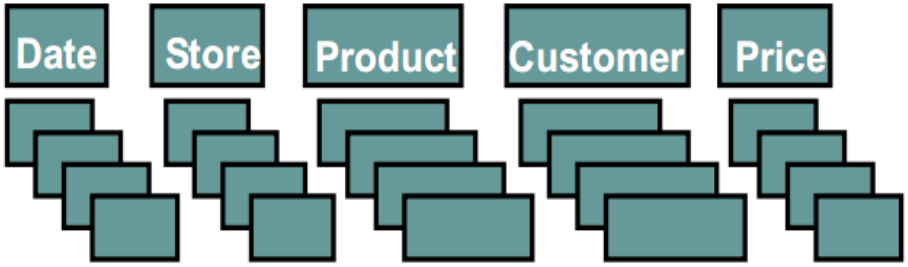
Row-Oriented vs. Column-Oriented

row-store



- + easy to add/modify a record
- - might read in unnecessary data

column-store



- + only need to read in relevant data
- - tuple writes require multiple accesses
- => *suitable for read-mostly, read-intensive, large data repositories*

Taken from Harizopoulos, Abadi, Boncz, Column-Oriented Database Systems, VLDB Tutorial 2009

Row-Oriented vs. Column-Oriented

- Row-oriented:
  - Easy to read a particular row of records
    - but may read unnecessary columns
- Column-oriented:
  - Good if reads only involve a few columns
    - e.g., Analyze the distribution of products being sold
    - but not good for reading particular rows
  - Efficient compression
    - imagine the column is year, which may be the same for many rows; the year column can be compressed
  - Efficient sorting and indexing on compressed data

HBase in Facebook

- Quick stats
  - 8B+ messages/day
  - Traffic to HBase
    - 75+ Billion R+W ops/day
    - At peak: 1.5M ops/sec
    - ~ 55% Read vs. 45% Write ops
    - Avg write op inserts ~16 records across multiple column families
- HBase
  - Small messages
  - Message metadata (thread/message indices)
  - Search index
- Haystack (our photo store)
  - Attachments
  - Large messages

http://people.apache.org/~nspiegelberg/HBase--FBMessagesHBasePoster.pdf

High-Level Languages

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
  - Not everyone wants to (or can) write Java code
- Solution: develop higher-level data processing languages
  - **Hive**: HQL is like SQL
  - **Pig**: Pig Latin is a bit like Perl

Reference: Lecture 12, http://www.umiacs.umd.edu/~jimmylin/cloud-2010-Spring/index.html

Hive and Pig

- Hive: data warehousing application in Hadoop
  - Query language is HQL (or HiveQL), variant of SQL
  - Tables stored on HDFS as flat files
  - Developed by Facebook, now open source
- Pig: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Developed by Yahoo!, now open source
  - Roughly 1/3 of all Yahoo! internal jobs

- Common idea:
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language “compiles down” to Hadoop jobs

Bonus

Hive Background

- Started at Facebook
- Data was collected by nightly cron jobs into Oracle DB
- "ETL" (extract, transform, load) via hand-coded python
- Grew from 10s of GBs (2006) to 1TB/day new data (2007), now 10x that (as of 2009).

Source: Cloudera, 2009.

Bonus

Hadoop as Enterprise Data WareHouse

- Scribe and MySQL data loaded into Hadoop HDFS
- Hadoop MapReduce jobs to process data
- Missing components:
  - Command-line interface for “end users”
  - Ad-hoc query support
    - … without writing full MapReduce jobs
  - Schema information

Bonus

Hive Applications

- Log processing
- Text mining
- Document indexing
- Customer-facing business intelligence (e.g., Google Analytics)
- Predictive modeling, hypothesis testing

Bonus

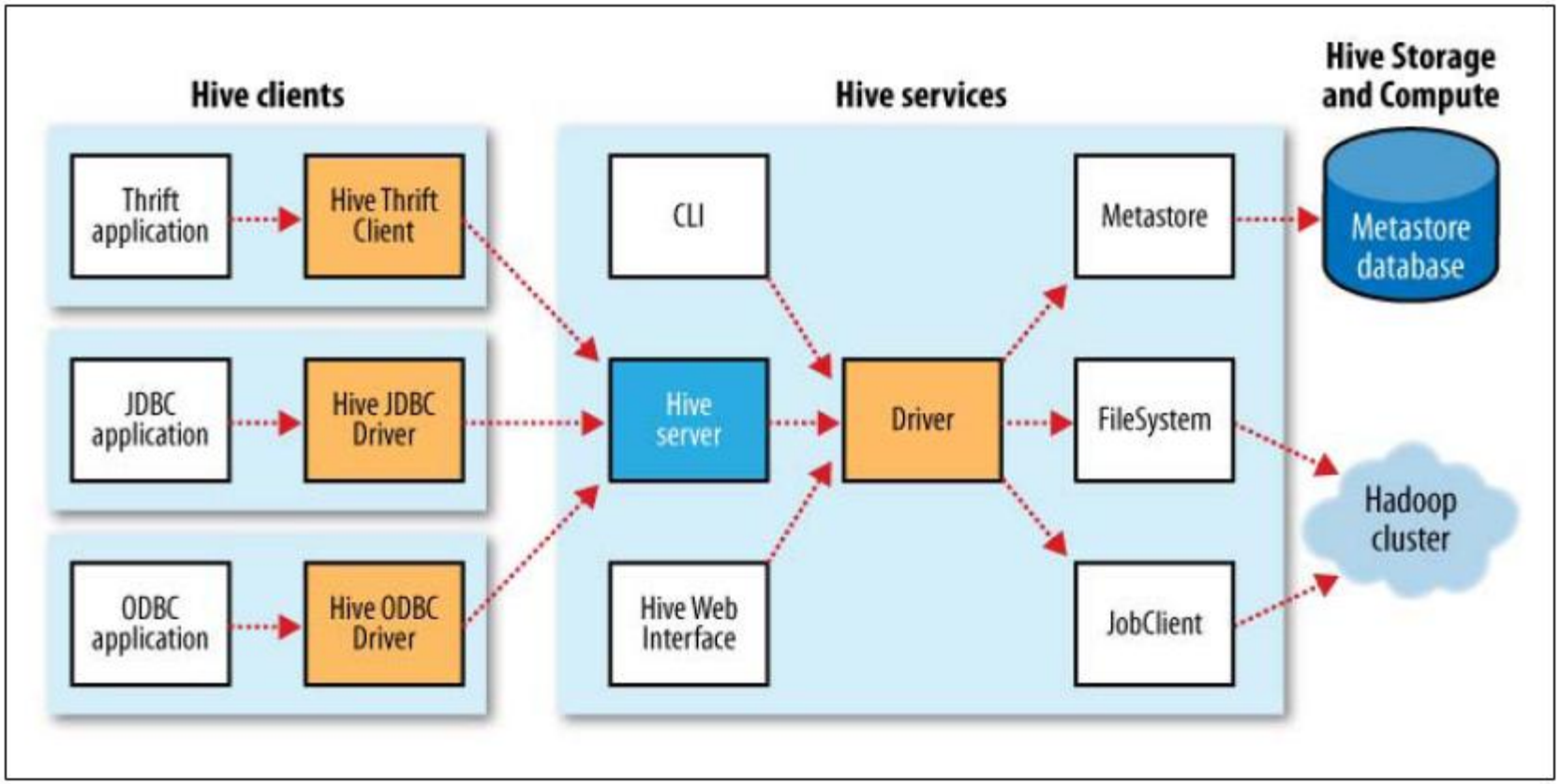
Hive Building Blocks

- Shell: allows interactive queries like
- MySQL shell connected to database
  - Also supports web and JDBC clients
- Driver: session handles, fetch, execute
- Compiler: parse, plan, optimize
- Execution engine: DAG of stages (M/R, HDFS, or metadata)
- Metastore: schema, location in HDFS, SerDe (storage format for the Serializer-Deserializer operation)

Bonus

Hive Architecture

- How to connect applications to the backend



Bonus

Hive: Data Model

- Tables
  - Typed columns (int, float, string, date, boolean)
  - Also, list: map (for JSON-like data)
- Partitions
  - e.g., to range-partition tables by date, so records of the same date will be stored in the same partition
- Buckets
  - Hash partitions within ranges (useful for sampling, join optimization)
  - Easy to try queries on a fraction of dataset

Bonus

Hive: Metastore

- Database: namespace containing a set of tables
- Holds table definitions (column types, physical layout)
- Holds partitioning information
- Can be stored in Derby, MySQL, and many other relational databases

Bonus

Hive: Physical Layout

- Warehouse directory in HDFS
  - E.g., /user/hive/warehouse
- Tables stored in subdirectories of warehouse
  - Partitions form subdirectories of tables
- Actual data stored in flat files
  - Control char-delimited text, or SequenceFiles
  - With custom SerDe, can use arbitrary format

Bonus

HBase/Hive

- Hive still uses HDFS for storage
  - Designed for batch processing
- HBase has a different storage characteristic to HDFS
  - Designed for real-time queries
- HBase integration with Hive remains in early stages of development
  - http://wiki.apache.org/hadoop/Hive/HBaseIntegration

Key-value Store

- Case study: Amazon Dynamo

Slides prepared by Paul Krzyzanowski
http://www.cs.rutgers.edu/~pxk/417/notes/content/25-dynamo-slides.pdf

Amazon Dynamo

- Not exposed as a web service
  - Used to power parts of Amazon Web Services (such as S3)
  - **Highly available, key-value storage system**
- In an infrastructure with millions of components, something is always failing!
  - Failure is the normal case
- A lot of services within Amazon only need primary-key access to data
  - Best seller lists, shopping carts, preferences, session management, sales rank, product catalog
  - No need for complex querying or management offered by an RDBMS
  - Full relational database is overkill: limits scale and availability
    - Still not easy to scale or load balance RDBMS on a large scale

Assumptions

- Two operations: get(key) and put(key, data)
  - Binary objects (data) identified by a unique key. Objects tend to be small (< 1MB)
- ACID gives poor availability
  - *Atomicity, Consistency, Isolation, Durability*
  - Use weaker consistency (C) for higher availability.
- Apps should be able to configure Dynamo for desired latency & throughput, balancing performance, cost, availability, durability guarantees.
- At least 99.9% of read/write operations must be performed within a few hundred milliseconds:
  - Avoid routing requests through multiple nodes

Design Decisions

- Incremental scalability
  - System should be able to grow by adding a storage host (node) at a time
- Symmetry
  - Every node has the same set of responsibilities
- Decentralization
  - Favor decentralized techniques over central coordinators
- Heterogeneity
  - Workload partitioning should be proportional to capabilities of servers

Compared to BigTable

- Dynamo targets apps that only need key/value access with a primary focus on high availability
  - key-value store versus column-store (column families and columns within them)
  - Bigtable: distributed DB built on GFS
  - Dynamo: distributed hash table
  - Updates are not rejected even during network partitions or server failures

Consistency and Availability

- Strong consistency & high availability cannot be achieved simultaneously
- Optimistic replication techniques – **eventually consistent** model
  - propagate changes to replicas in the background
  - can lead to conflicting changes that have to be detected & resolved
- When do you resolve conflicts?
  - During writes: traditional approach - reject write if cannot reach all (or majority) of replicas
  - During reads: Dynamo approach
    - Design for an "always writable" data store - highly available
    - read/write operations can continue even during network partitions
    - Rejecting customer updates won't be a good experience
      - A customer should always be able to add or remove items in a shopping cart

Consistency and Availability

- Who resolves conflicts?
  - Choices: the data store system or the application
  - Data store
    - application-unaware, so choices limited simple policy, such as "last write wins”
  - Application
    - app is aware of the meaning of the data
    - can do application-aware conflict resolution
    - e.g., merge shopping cart versions to get a unified shopping cart.
  - fall back on "last write wins" if app doesn't want to bother

Read & Writes

- Two operations:
- get(key) returns
  - 1. object or list of objects with conflicting versions
  - 2. context (resultant version)
- put(key, context, value)
  - stores replicas
  - the nodes that hold replicas are based on the key.
  - context: ignored by the application but includes version of object
  - key is hashed with MD5 to create a 128-bit identifier
    - used to determine the storage nodes that serve the key

Partitioning

- Break up database into chunks distributed over all nodes
  - Key to scalability
  - Example: Bigtable's tablets, Map-Reduce partitioning
- Relies on consistent hashing
  - Regular hashing: change in # slots requires all keys to be remapped
  - Consistent hashing:
    - K/n keys need to be remapped, K = # keys, n = # slots
- Logical ring of nodes
  - Each node assigned random value in the hash space: position in ring
  - Responsible for all hash values between its value and predecessor's value
  - Hash(key); then walk ring clockwise to find first node with position>hash
  - Adding/removing nodes affects only immediate neighbors

Replication

- Data replicated on N hosts (N is configurable)
  - Key is assigned a coordinator node (via the hashing)
  - Coordinator is in charge of replication
- Coordinator replicates keys at the N-1 clockwise successor nodes in the ring

Versioning

- Not all updates may arrive at all replicas
- Application-based reconciliation
  - Each modification of data is treated as a new version
- Vector clocks are used for versioning
  - Capture causality between different versions of the same object
  - Vector clock is a set of (node, counter) pairs
  - Returned as a context from a get() operation

Availability

- configurable values
  - N: number of replicas
  - R: minimum # of nodes that must participate in a successful read operation
  - W: minimum # of nodes that must participate in a successful write operation
  - Quorum-like system: R+W > N
    - What would happen if R + W <= N?
    - What's the impact of R and W?
- Common (N,R,W) in Dynamo: (3,2,2)