

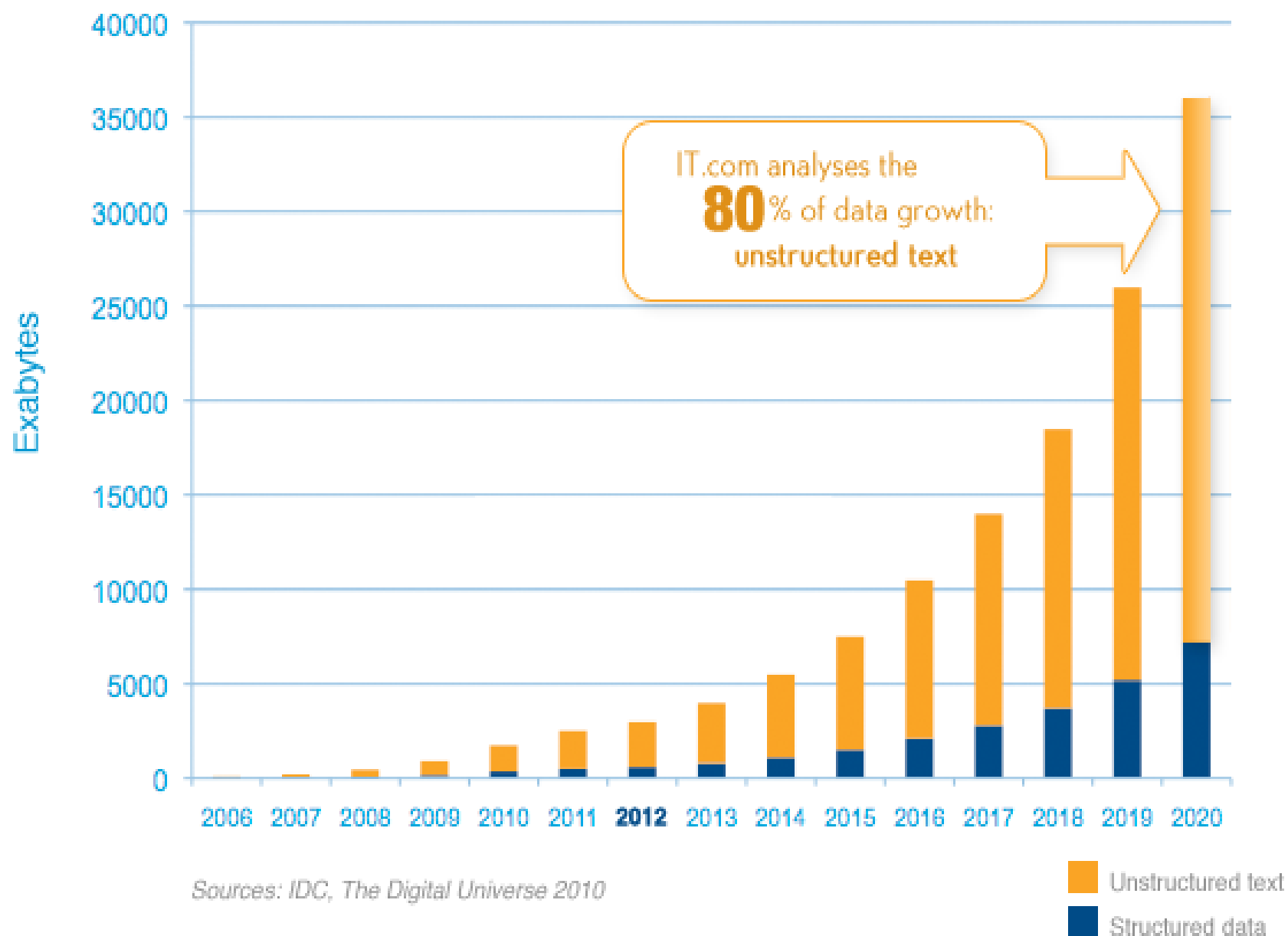
# ***Cloud Programming: Lecture8 – Storage: GFS & Big Table***

***National Tsing-Hua University  
2015, Spring Semester***



國立清華大學

# Worldwide Corporate Data Growth



# *Unstructured Data*

- Data can be of any type
  - Not necessarily following any format or sequence
  - Not follow any rules, so is not predictable
- Two Categories
  - Bitmap Objects
    - Inherently non-language based, such as image, video or audio files
  - Textual Objects
    - Based on a written or printed language, such as Microsoft Word documents, e-mails or Microsoft Excel spreadsheets

# Structured Data

- Data is organized in semantic chunks (entities)
- Entities in the same group have the same descriptions (attributes)
- Entities are grouped together (relations or classes)
- Descriptions for **ALL** entities in a group (**schema**)
  - The same defined format or type
  - A predefined length
  - The same ordering

| Professors     |
|----------------|
| ProfessorID    |
| ProfessorName  |
| ProfessorPhone |

| Students    |
|-------------|
| StudentID   |
| StudentName |
| AdvisorID   |



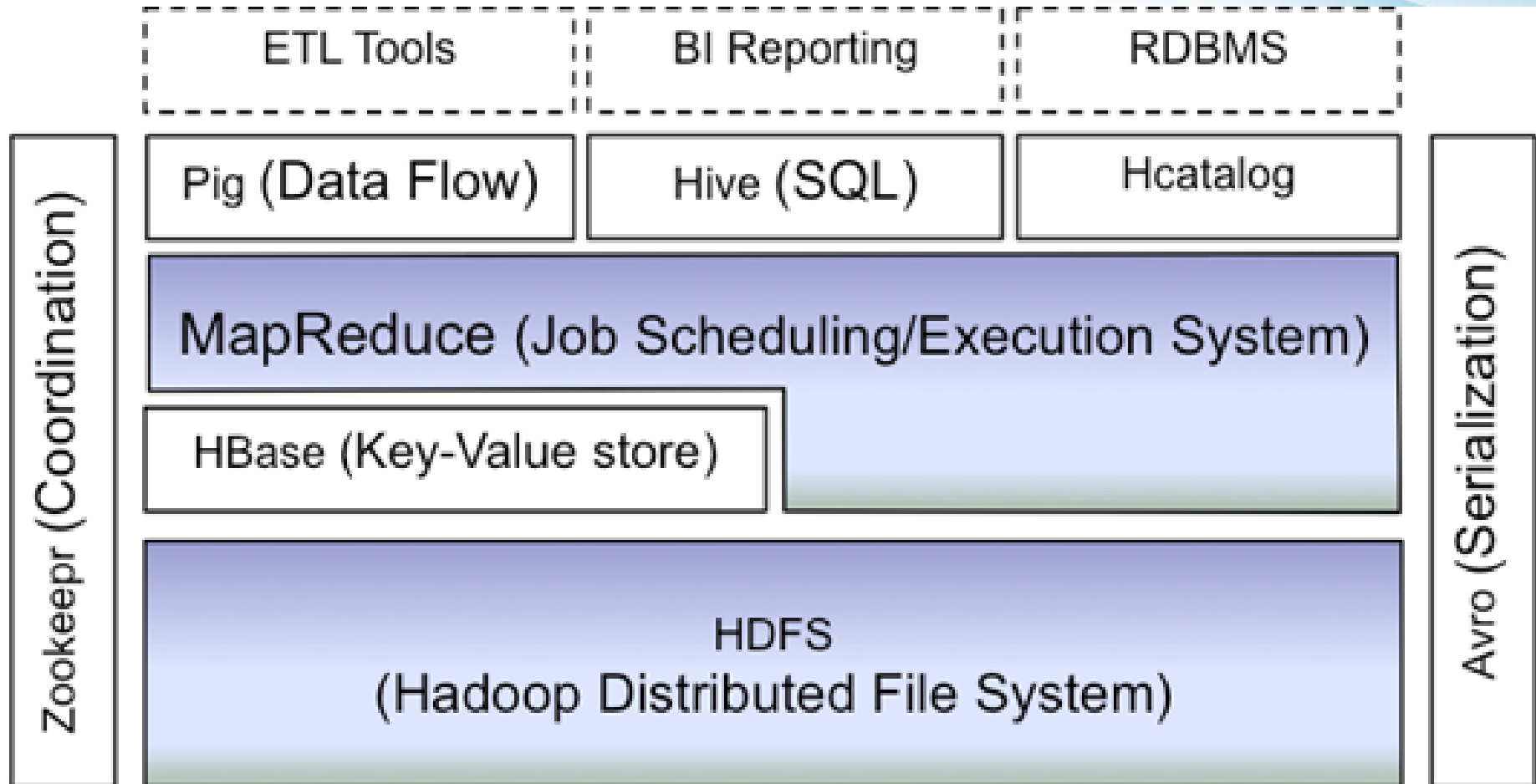
# *Semi-Structured Data*

- Organized in semantic entities
- Similar entities are grouped together
- Entities in same group may **not** have same attributes
  - Order of attributes not necessarily important
  - Not all attributes may be required
  - Size of same attributes in a group may different
  - Type of same attributes in a group may different

# *Example of Semi-Structured Data*

- Name: Computing Cloud
- Phone\_home: 035715131
- Name: TA Cloud
- Phone\_cell: 0938383838
- Email: cloudTA@gmail.com
- Name: Student Cloud
- Email: hiCloud@hotmail.com

# *Hadoop Data Processing Platform*



# *File System Overview*

- Physically, a file is a collection of disk blocks.
- Logically, a file is a unit of data on disks or other media.
- File system is a system that manages files
  - Maps file names and offsets to disk blocks
  - The set of valid paths form the “namespace” of the file system.
  - Manages file attributes, such as file size, date, types, owner, etc.
  - Manages volume properties, such free size etc.



# *FS Design Considerations*

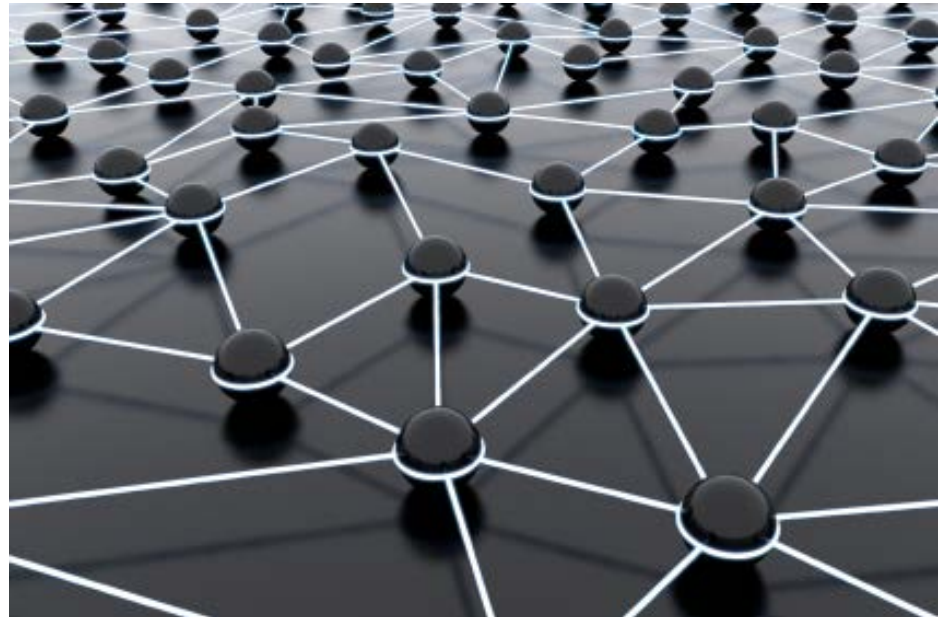
- Namespace: How to find a file?
  - Logical volume and physical mapping
- Consistency: What to do when more than one user reads/writes on the same file?
  - Caching and replication
- Security: Who can do what to a file?
  - Authentication/Access Control List (ACL)
- Reliability: How to insure data never lost?
  - Computer fails all the time due to various reasons
  - HW/SW failure, power outage, malicious users....

# *Local FS on Unix-like Systems*

- Namespace
  - root directory “/”, followed by directories and files.
- Consistency
  - “sequential consistency”, newly written data are immediately visible to open reads
- Security
  - Account security: password
  - File permission: uid/gid, R/W/X mode of files
- Reliability
  - fsck: command to check & repair a linux FS
  - Journaling: Change is logged before it is committed
  - Snapshot: Copy-on-write, any changes create a new file

# *Distributed File Systems (DFS)*

- Allows access to files from multiple hosts sharing via networks
  - May include facilities for transparent replication and fault tolerance
- Major problems
  - Concurrency
  - Dropped connections (reliability)



# *Why DFS?*

- Multiple users want to share files
- The data may be much larger than the storage space of a computer
- A user want to access his/her data from different machines at different geographic locations
- Maintenance
  - Backup
  - Management

# *Why File System Is Not Enough?*

1. What we need is not only data, but also the relations among them.
  - The relations of data are also data
  - Also need data to describe data (metadata)
2. Common data operations are easier to perform using DataBase Management System (DBMS)
  - Search: retrieve data from the database
  - Update: update existing data
  - Insertion: insert new data
  - Deletion: remove existing data

# ***DB & DBMS***

- DataBase (DB)
  - A collection of **data** stored in a **standardized format** designed to be shared by multiple users
- DataBase Management System (DBMS)
  - **Software** that defines and operates a database
  - Provides storage, query, security, backup and other facilities

# ***RDBMS***

- Database management systems for **relational DB**
- Support **SQL language**
- Enforce **ACID property** for transaction
- Almost all databases today is RDBMS & support SQL
  - Oracle, SQL Server, MySQL, DB2
- Advantages
  - Simplicity, Robustness, Flexibility, Performance
- However,
  - To offer all of these, relational databases have to be incredibly complex internally



# Relational Database

- First defined in 1970 by Edgar Codd, of IBM
- In a ***relational database*** (the most commonly used one), records are organized using tables
  - Columns for attributes; rows for records

| Name | StudentID | Major | Grade | B-day  | Gender | ... |
|------|-----------|-------|-------|--------|--------|-----|
| Tom  | 123456    | CS    | 2yr   | 1-1-11 | M      | ... |
| Mike | 789012    | EE    | 1yr   | 2-2-22 | F      | ... |
| ...  | ...       | ...   | ...   |        |        | ... |

- Primary key: one (or multiple) attributes that can be used to **uniquely** identify each row/entity in a table



# Relations: EM Model

- The relations among objects are described by the ER model (Entity-Relation model)



- Relations are also organized as tables

CourseTaking

| StudentID | CourseID | Grade | Status  |
|-----------|----------|-------|---------|
| 123456    | 990110   | -1    | Normal  |
| 234567    | 990221   | -1    | Dropped |

CourseTeaching

| TeacherID | CourseID |
|-----------|----------|
| 888999    | 990110   |
| 777666    | 990221   |

# SQL: *Structured Query Language*

- SQL: a special-purpose **programming language** designed for **managing data** in relational database
  - Data query: retrieve data
  - Data manipulation: add, update and delete data or tables
  - Data definition: define data schema
  - Data control: authorize data access permission
- 1974: first commercial languages for **relational DB**
- 1986: become a **standard** for ANSI & ISO

```
SELECT *  
FROM students  
WHERE ClassRank < 3  
ORDER BY StudentID;
```

| Students    |
|-------------|
| StudentID   |
| StudentName |
| ClassRank   |

# Update, Insertion, Deletion

- Suppose you want to add a course for next semester.

**INSERT INTO** CourseTaking **VALUES** ('123456', '990110');

- A **transaction** is more than just an insertion like that.

A sequence of operations must happen all together

- Before insertion

- The system needs to check if there is a schedule conflict
- Also, the capacity of the class, the pre-requirement, ...

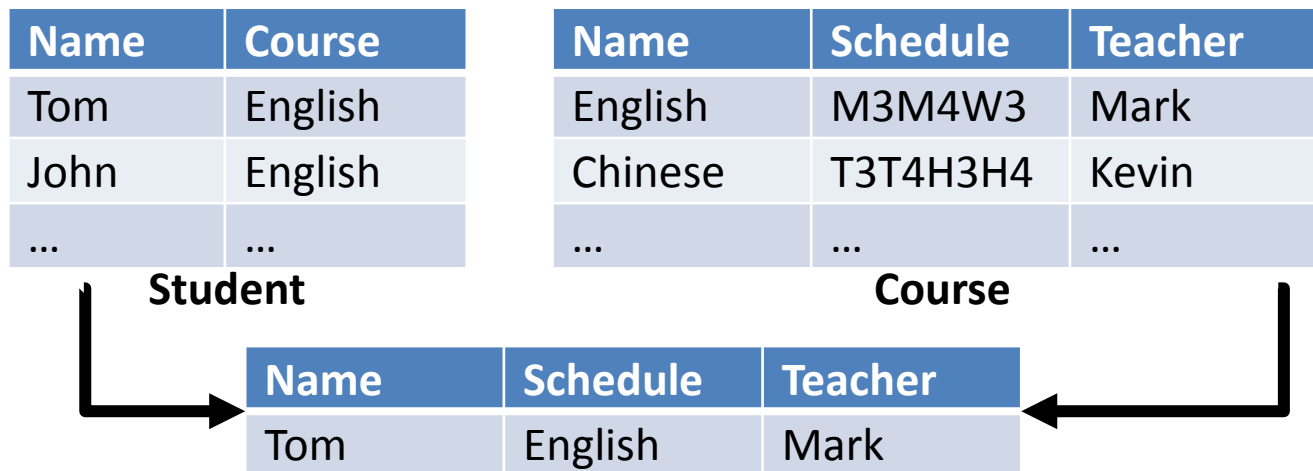
- After insertion

- Suppose there is an attribute in Course, called “NoStudent”, that records the total number of students taking this course.

**UPDATE** Course **SET** NoStudent=ns+1 **WHERE** CourseID='990110';

# Joint Query

- Select data from multiple tables needs join operation
  - Costly due to lookup & matching among tables



```
SELECT Student.Name, Student.Course, Course.Teacher
FROM Course, Student
WHERE Student.Name="Tom" AND
      Student.Course=Course.Name
```

Usually will build a “**view**” to speedup common queries

# *ACID Properties of Transactions*

- **A**tomicity: either all the operations of a transaction are executed or none of them are.
- **C**onsistency: the database is in a legal state before and after a transaction
- **I**solation: the effects of one transaction are isolated from other transactions.
- **D**urability the effects of successfully completed transactions endure subsequent failures.



# Database Integrity

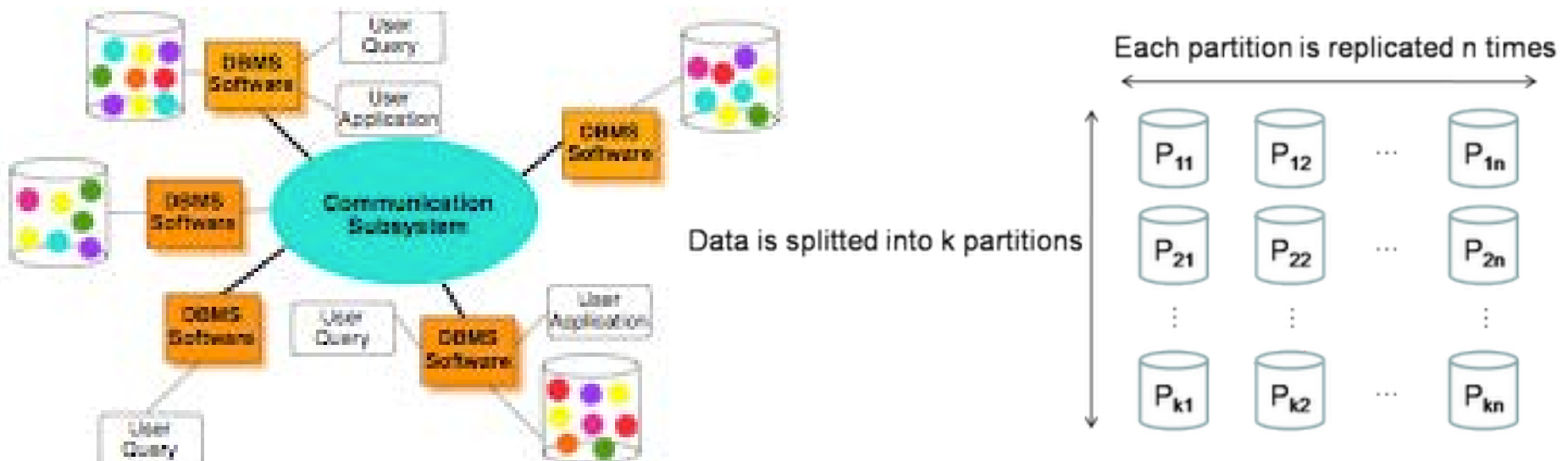
- RDBMS need to maintain database integrity
  - **Transaction log:** non-volatile record of each transaction's activities, built before the transaction is allowed to happen.
  - **Locking:** preventing others from accessing data being used by a transaction.
  - **Roll-back:** procedure to undo a failed, partially completed transaction.





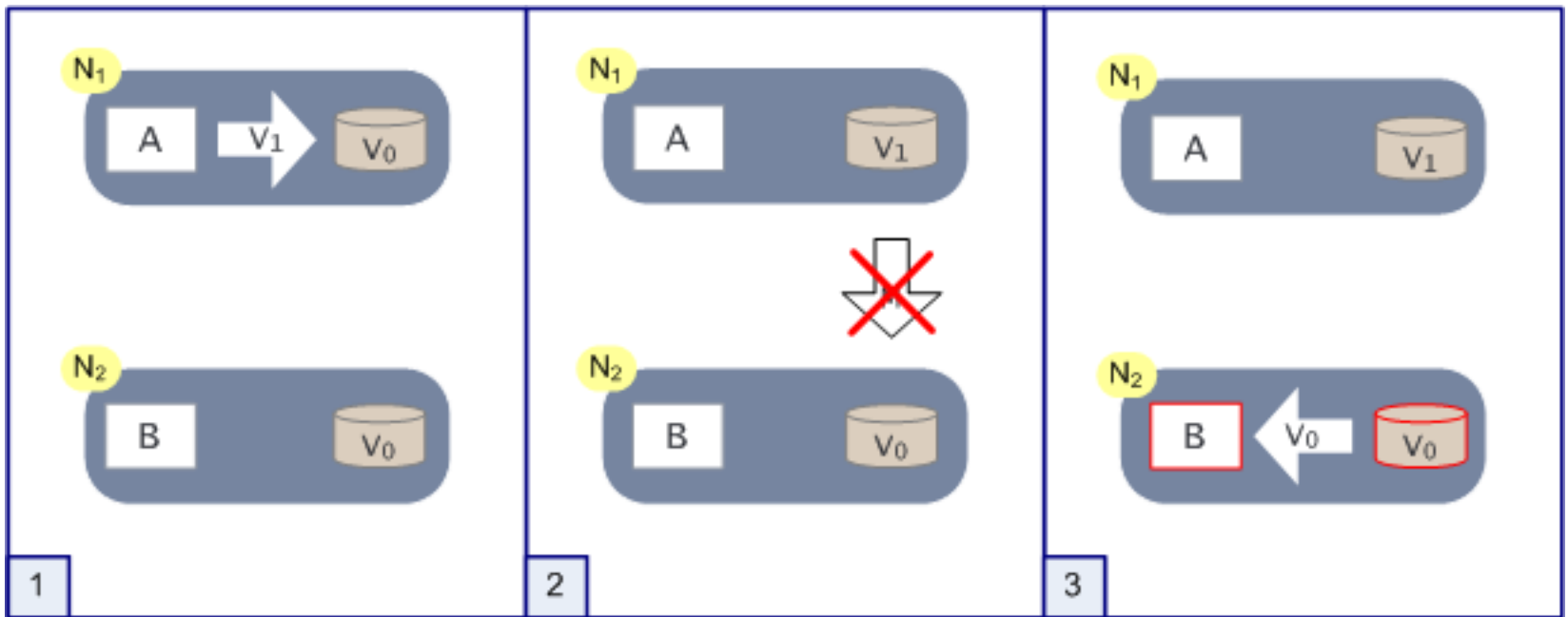
# Distributed RDBMS

- Distributed database management systems is a software for managing databases stored on multiple computers in a network.
- A natural way to scale up the DBMS



# Consistency Problem

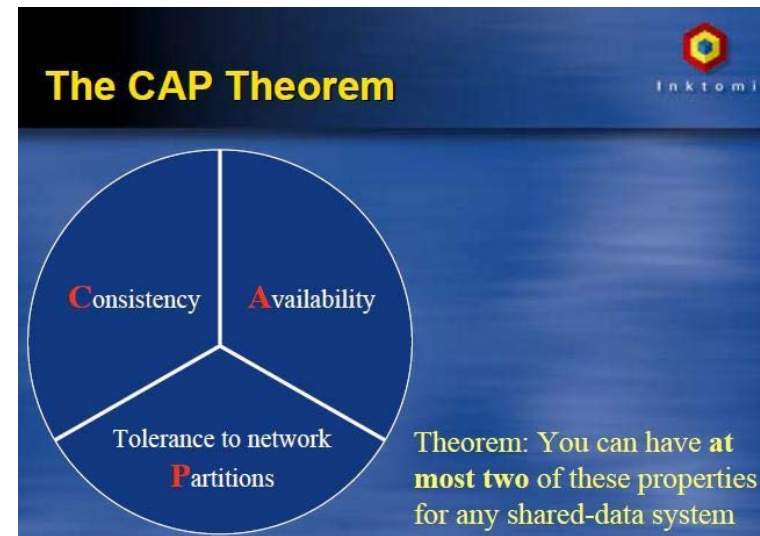
- Ex: A writes v1 to DB and B reads the data





# Brewer's CAP Theorem

- It is impossible for a distributed computer system to provide all three of
  - **Consistency**: all nodes see the same data at the same time
  - **Availability**: a guarantee that every request receives a response about whether it was successful or failed
  - **Partition tolerance**: the system continues to operate despite arbitrary message loss



# NOSQL

- **Not Only SQL**
  - A term used to designate database management systems differ from classic relational database management systems
- **Properties**
  - May not require fixed table **schemas**
  - Easy for **scale out**
  - Relax the data **consistency** requirement
  - Usually avoid **join** operations



# *Types of NoSQL*

- **Column:**
  - **BigTable** , **HBase**, Cassandra, AWS SimpleDB
- **Document:**
  - **MongoDB** , Apache CouchDB
- **Key-value:**
  - **Dynamo(S3)**, CouchDB, MemcacheDB
- **Graph:**
  - **Neo4J**, Allegro, InfiniteGraph, Virtuoso

# *Strength of NoSQL*

- **Scalability:**

- Efficient, scale-out architecture well-suited for big data. It's horizontal scaling meshes extremely well with the cloud.

- **Flexibility:**

- It is not locked into any one specific data mode or schema, so it can handle virtually any structure or format of data including unstructured data.

- **Cost-Effectiveness:**

- Its implementations are typically cheap, low-grade commodity devices. Its software is entirely open-source.

# *Weakness of NoSQL*

- **Performance and Scaling > Consistency**
  - Relax the data consistency requirement.
- **Standardization:**
  - No standard API or query language.
- **Maturity:**
  - There really aren't many reliable standards for NoSQL databases quite yet. Most products are from start-up companies with limited support to customers.
- **Ad-hoc query & analysis:**
  - Even a simple query requires significant programming expertise, and commonly used BI tools do not provide connectivity to NoSQL. (Pig & Hive are solving this problem.)

Motivations & Design Overview

System Interaction

Fault Tolerance

# ***GOOGLE FILE SYSTEM***

# *Assumptions/Motivation*

- Take failure as norm than exception
  - Inexpensive commodity components fail all the time
  - Fault-tolerance and auto-recovery need to be built into the system
- Modest number of large files
  - Expect a few million files, each 100 MB or larger
  - Multi-GB files are common and should be managed efficiently
- Co-design with the Google applications (Web-based Search)
  - Primary consist of large streaming reads & small random reads
  - Multiple clients concurrent append data instead of write
  - Seldom or No random write
- High sustained BW is more important than low latency
  - Place a premium on processing data in bulk at a high rate, while have stringent response time



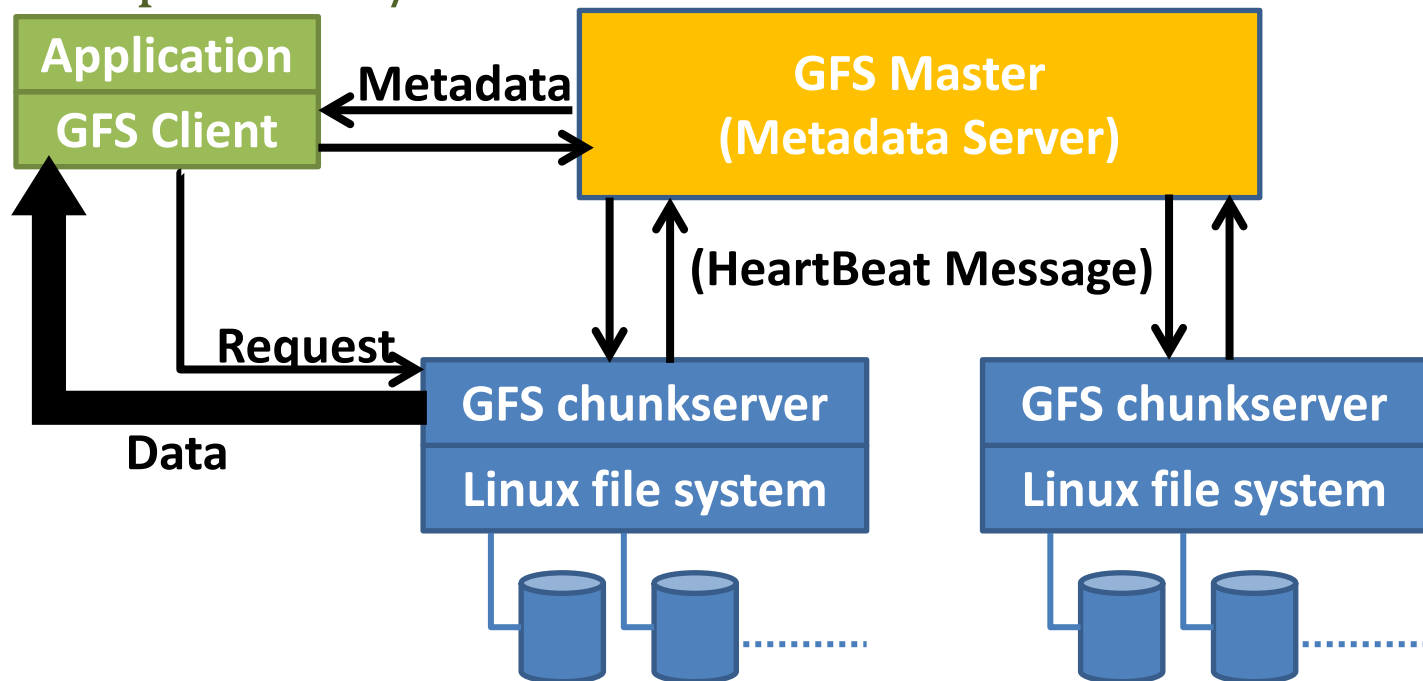
# Design Decisions

- Reliability through replication
  - Replication over both nodes & racks
- Single master to coordinate access and to keep metadata
  - Simple centralized management
- No data caching
  - Little benefit on client: large data sets / streaming reads
  - Eliminating cache coherence issues
- Relax consistency model
  - Trade consistency for performance
- Familiar interface, but customize the API
  - No POSIX: simplify the problem; focus on Google apps
  - Add *snapshot* and *record append* operations



# Architecture

- Components:
  - Master holds metadata
  - Chunkservers hold data
  - Client produces/consumes data

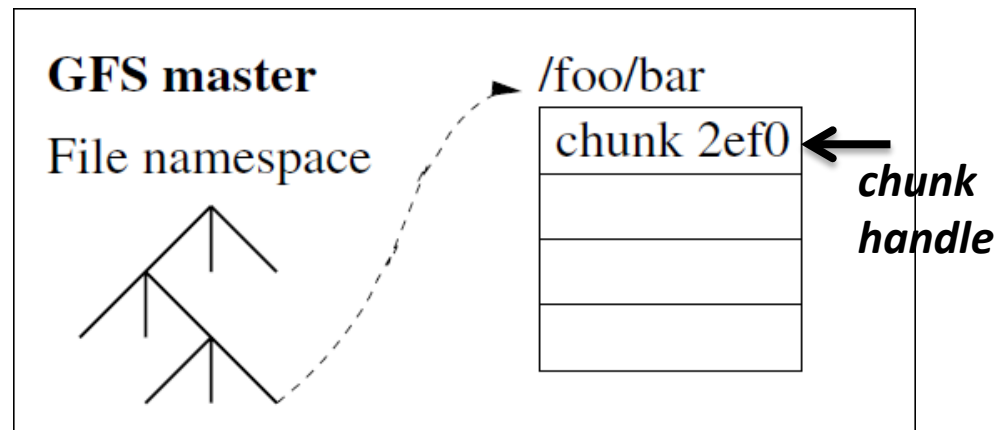


# *Single Master*

- The master have global knowledge of chunks
  - Easy to make decisions on placement, replication and recovery
- Drawback:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters (Replication of master)
  - Minimize master involvement
    - Separate data flow from control flow: master only involve for metadata not data
    - cache metadata at clients (not data itself)
    - large chunk size (64MB per req.)
    - lease management: master delegates authority to primary replicas in data mutations

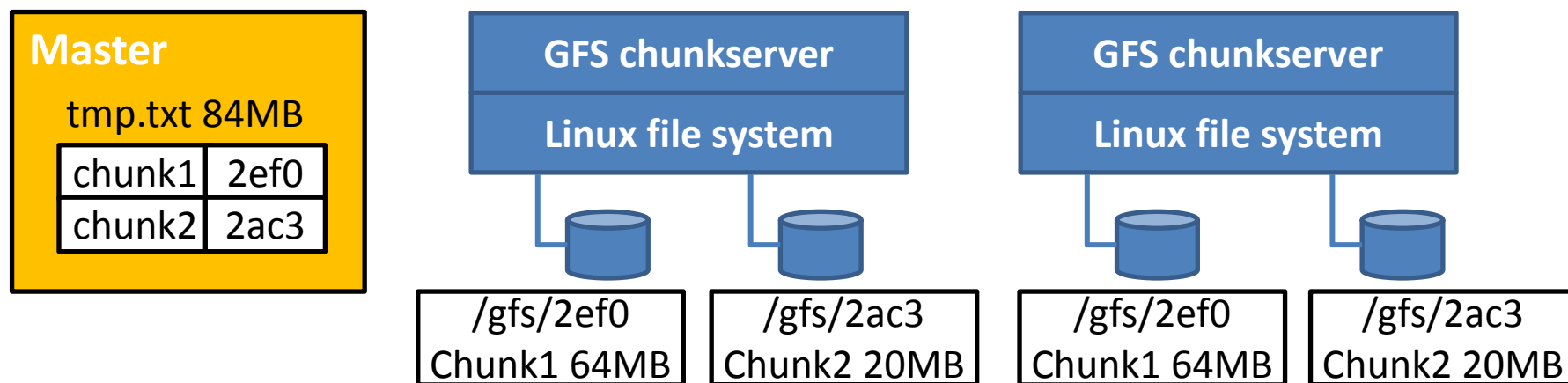
# Metadata

- Information:
  - Namespace tree
  - File to chunks mapping
    - A file is split into multiple chunks
    - Each chunk has a chunk handle Identified by an immutable and globally unique 64 bit
  - Chunk locations
    - Chunkserver IP, file path
  - Access control info
- Management:
  - Keep in **memory**
  - Use **checkpoint & log** to insure persistency



# Chunkserver - Data

- Files stored in *chunks* (c.f. “*blocks*” in disk file systems)
  - A chunk is a **Linux file** on local disk of a chunkserver
  - Unique chunk handles assigned by master at creation time
  - Read/write by (chunk handle, byte range)
  - Fixed large chunk size (i.e. 64MB)
  - Each chunk is replicated across 3+ chunkservers



# *Chunk Size*

- Typical chunk size is 64 MB
- A large chunk size offers important advantages when stream reading/writing
  - Less communication between client and master
  - Less memory space needed for metadata in master
  - Less network overhead between client and chunkserver (one TCP connection for larger amount of data)
- On the other hand, a large chunk size has its disadvantages
  - Hot spots
  - Fragmentation

# *Replica Placement*

- Traffic between racks is slower than within the same rack
- A replica is created for 3 reasons
  - Chunk creation: new chunks
  - Chunk re-replication: missing chunks
  - Chunk rebalancing: migrating chunks
- Master has a replica placement policy
  - Maximize data reliability and availability
  - Maximize network bandwidth utilization

Lease Management

Read/Write File

Consistency Model

Concurrent Write

Atomic Append

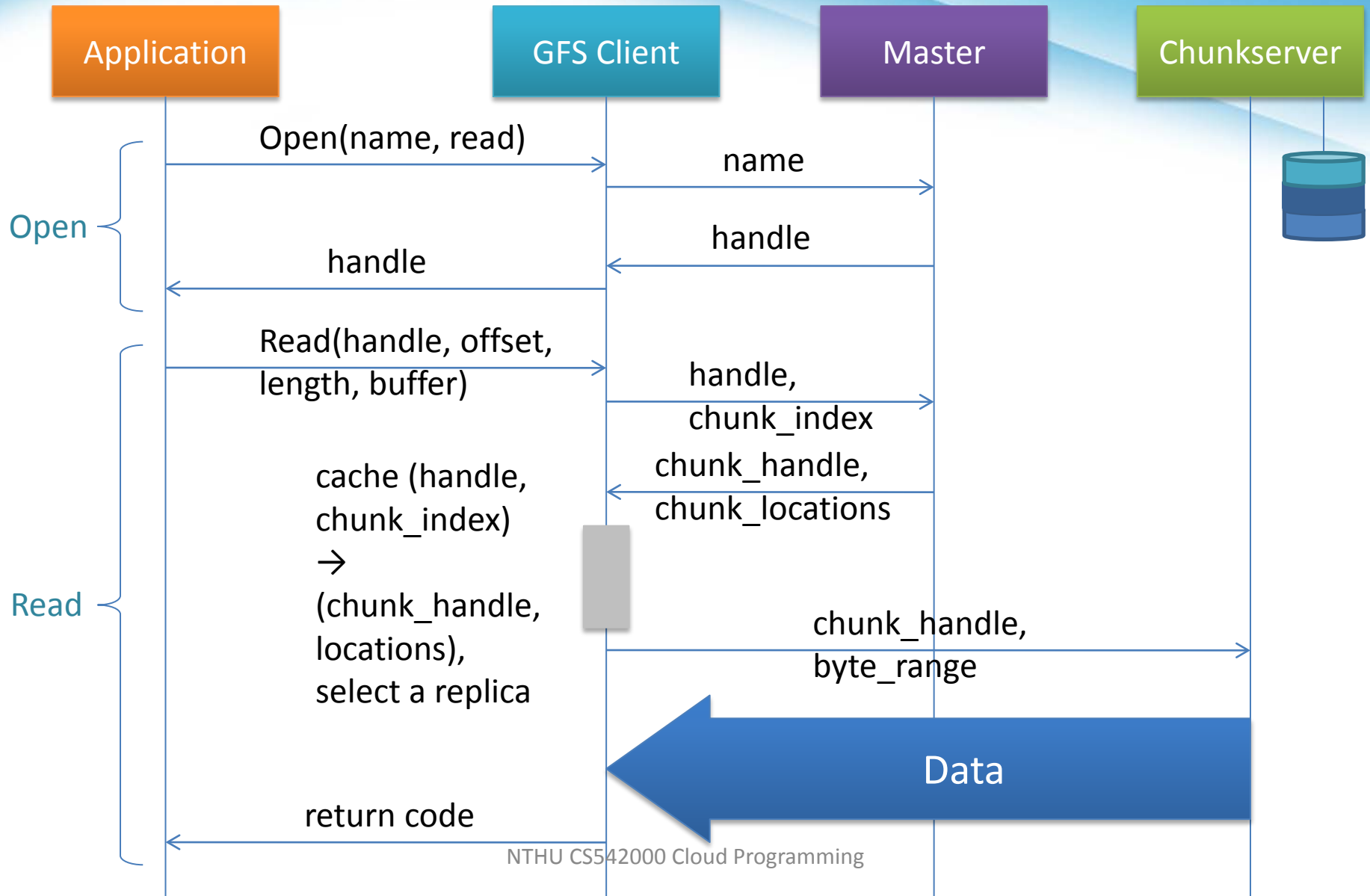
## ***GFS: SYSTEM INTERACTION***

# *Lease Management*

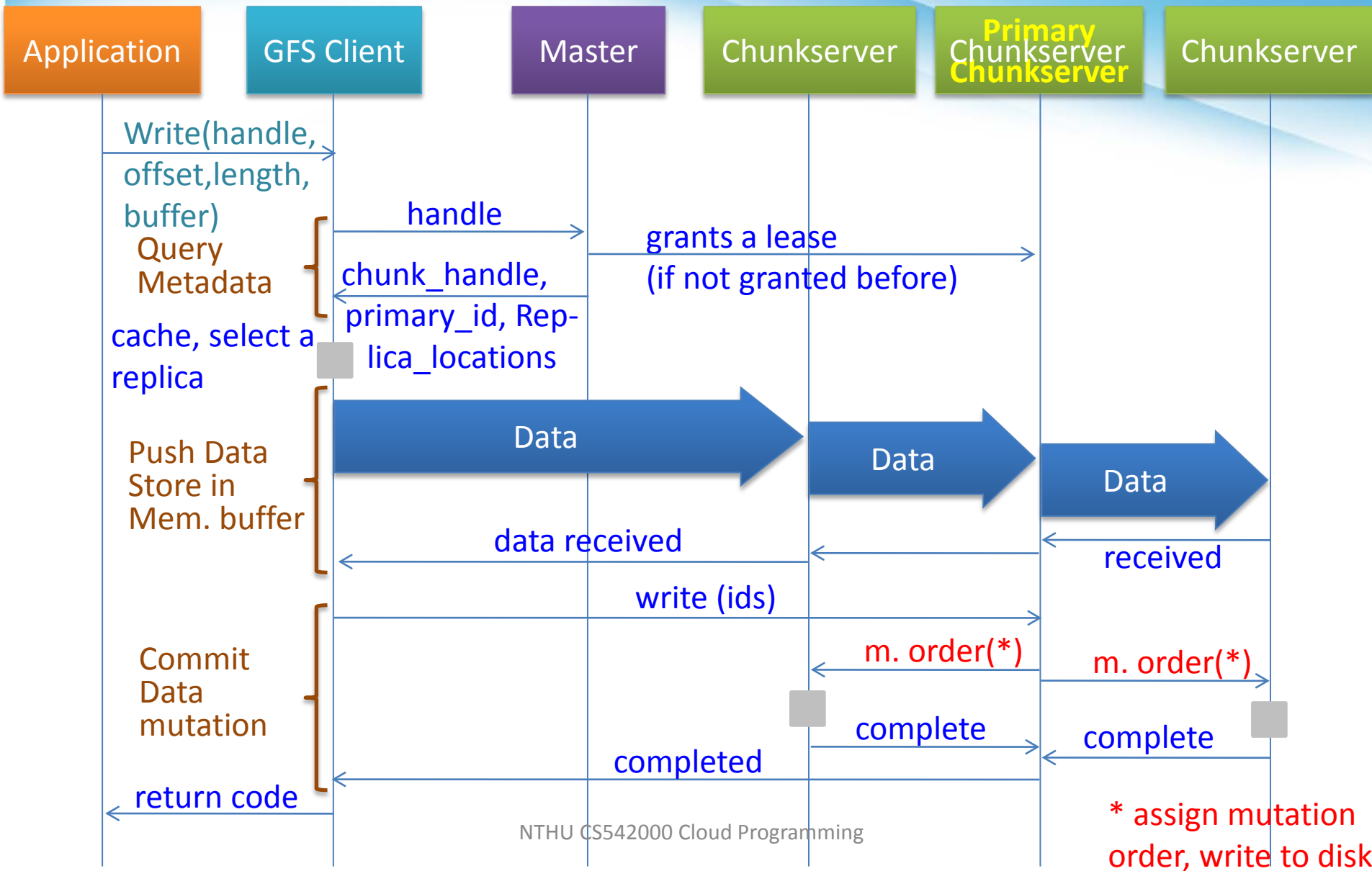
- Goal
  - Minimize master overhead by authorizing a datacenter to make decision during **data mutation** (i.e. write/append)
- One lease per chunk
  - master picks one replica as primary; gives it a “lease”
  - a lease = a lock that has an expiration time
  - primary defines a serial order of mutations, all replicas follow this order
- The primary still can
  - renew the lease before it expires
  - revoke a lease (e.g., for snapshot)
  - grant the lease to another replica if the current lease expires (primary crashed, etc)



# While reading a file



# While writing to a File



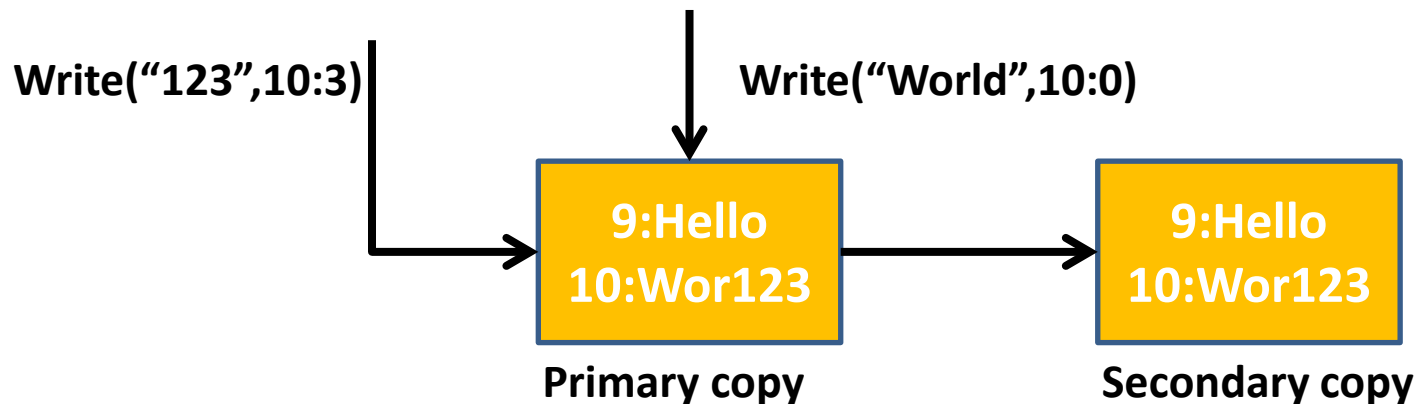
# Consistency Model

- GFS has a relaxed consistency model
- File **namespace** mutations are atomic and consistent
  - handled exclusively by the master
  - namespace lock guarantees atomicity and correctness
  - order defined by the operation logs
- File **region** mutations: complicated by replicas
  - “Consistent” = all replicas have the same data
  - “Defined” = consistent & reflects the mutation **entirely** (i.e. you see what you write)
  - A relaxed consistency model: not always consistent, not always defined, either

# Concurrent Write

- If two clients concurrently write to the same region of a file, **any** of the following may happen to the overlapping portion:
  - Eventually the overlapping region may contain data from exactly one of the two writes.
  - Eventually the overlapping region may contain a **mixture of data** from the two writes.

→ Consistent, not defined



# Atomic Record Appends

- GFS provides an atomic append operation called **“record append”**
- GFS guarantees that the data is appended to the file **atomically at least once**
  - GFS picks the offset, and **returns the offset to user**
  - GFS client **automatically reties** if any replica fails
- Used heavily by Google apps
  - serve as multiple-producer/single-consumer queues
  - merged results from many different clients

➔ Defined interspread with inconsistent



# Consistency Summary

- Applications are responsible for
  - self-validating, self-identifying records

|                      | Write                                     | Record Append  |
|----------------------|---|--|
| Serial success       | <i>defined</i>                            | <i>defined</i><br>interspersed with<br><i>inconsistent</i> |
| Concurrent successes | <i>consistent</i><br>but <i>undefined</i> |  |
| Failure              | <i>inconsistent</i>                       |  |

Data Integrity  
Garbage Collection  
Snapshot

## ***GFS: FAULT TOLERANCE***

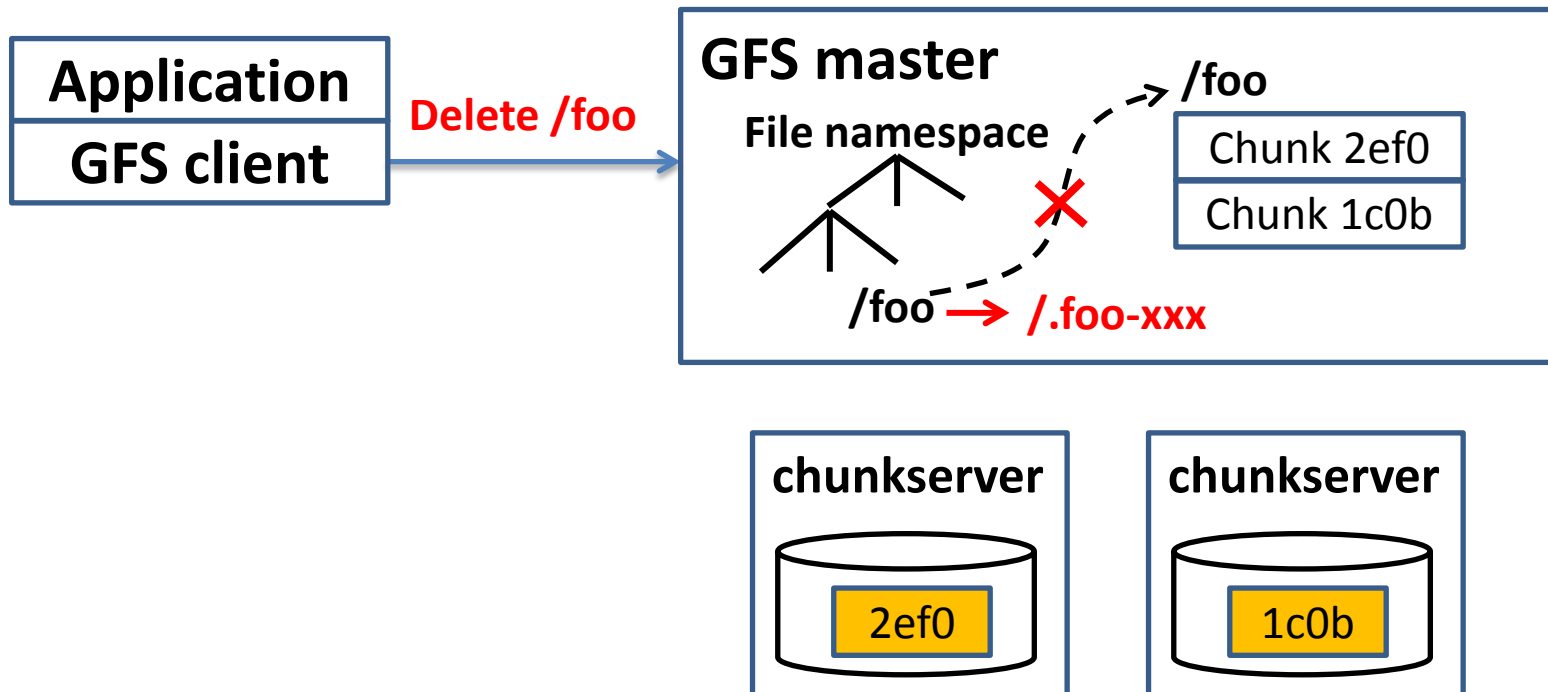
# *Data Integrity*

- Integrity:
  - Data can't be modified due to unexpected event, such as disk failure
- A responsibility of chunkservers, not master
  - GFS doesn't guarantee identical replica, independent verification is necessary
  - 32 bit checksum for every 64 KB block of data
  - chunkserver verifies checksum before every read
- If checksum mismatch occurs
  - Return error to client, report to master to delete the chunk
  - Client reads from another replica
  - Master makes another replica



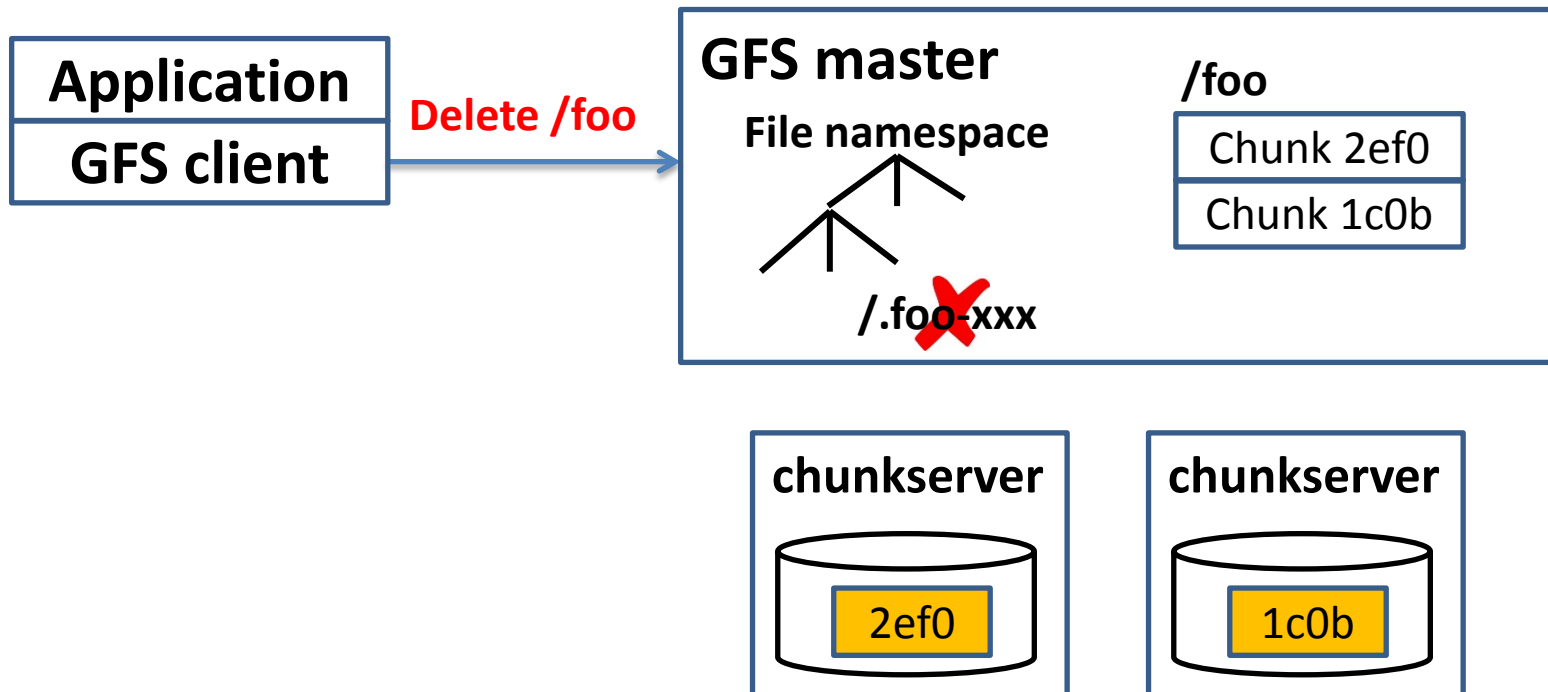
# Garbage Collection

- Chunks of deleted files are not reclaimed immediately
  1. renames the file to a hidden name with timestamp, and record in log



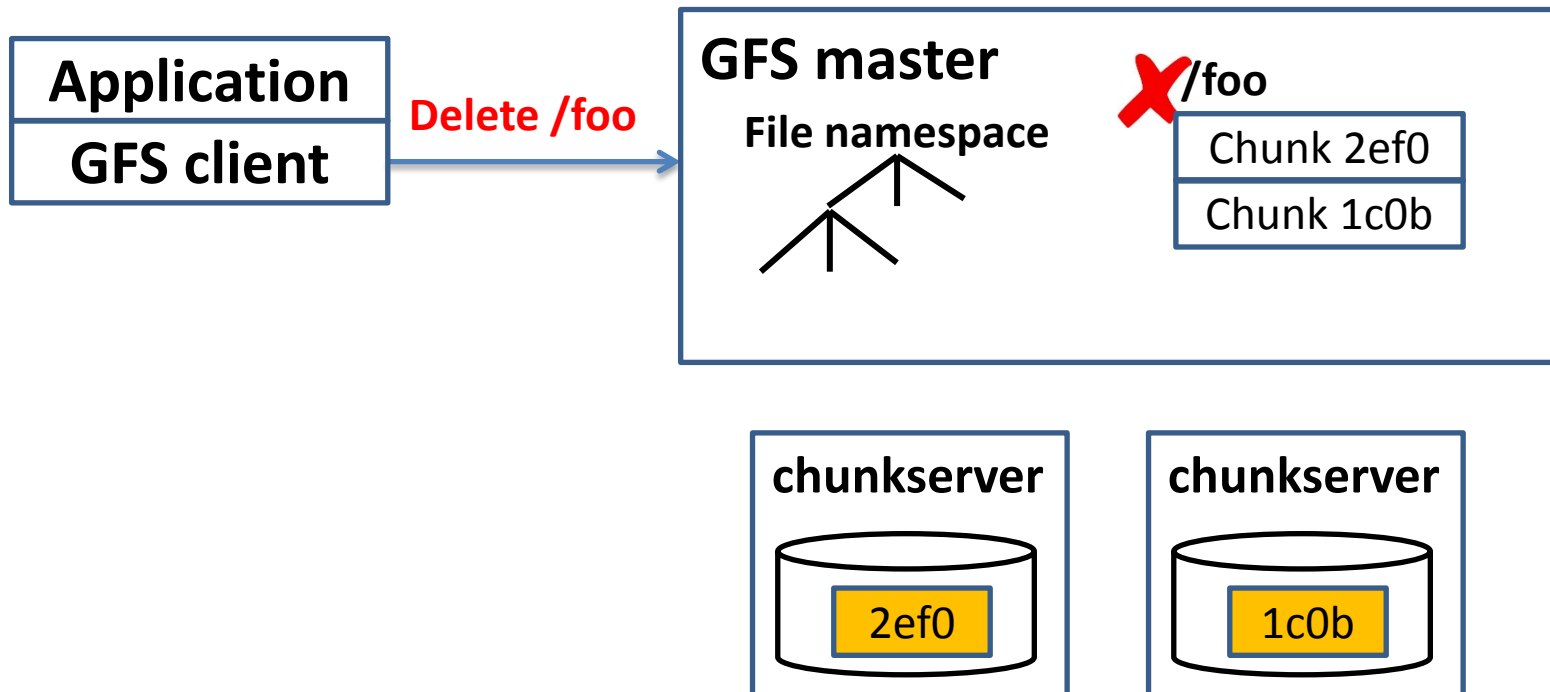
# Garbage Collection

- Chunks of deleted files are not reclaimed immediately
- 2. Master scans file namespace regularly to remove metadata of hidden files older than 3 days



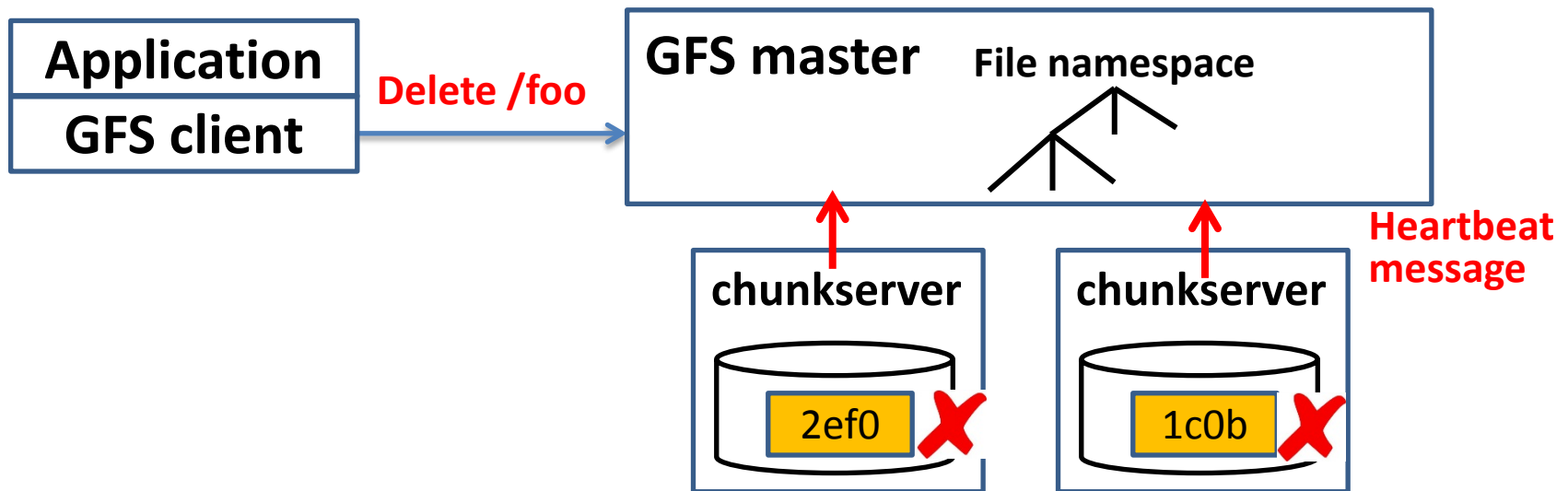
# Garbage Collection

- Chunks of deleted files are not reclaimed immediately
3. Master scans chunk table regularly to remove metadata of orphaned chunks



# Garbage Collection

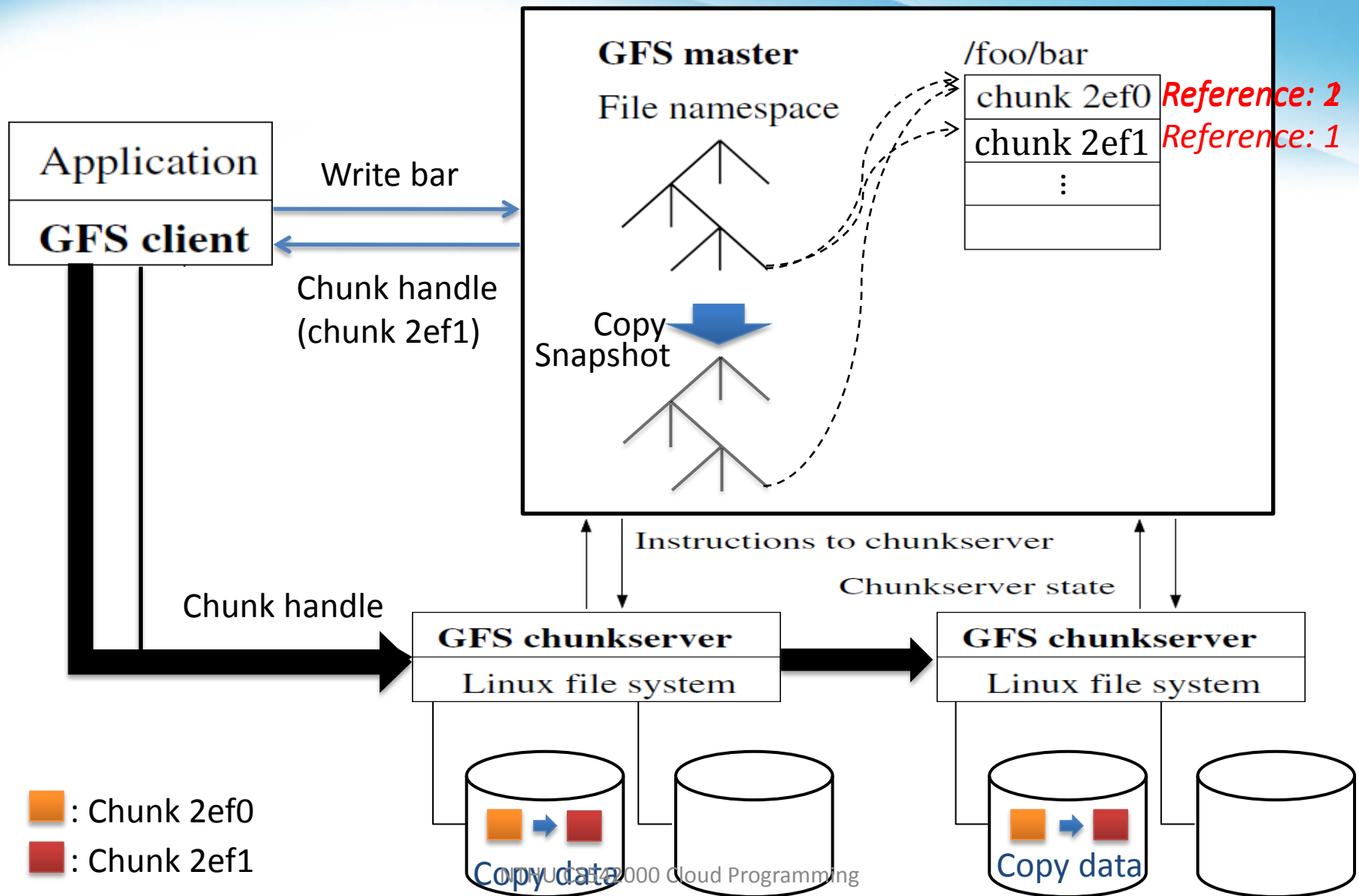
- Chunks of deleted files are not reclaimed immediately
- 4. Chunkserver sends master a list of chunk handles it has in regular HeartBeat message
  - Master replies the chunks not in namespace
  - Chunkserver is free to delete the chunks



# *Snapshot*

- Makes a copy of a file or a directory tree almost instantaneously for fault tolerance
  - minimize interruptions of ongoing mutations
  - copy-on-write with reference counts on chunks
- Steps:
  1. a client issues a snapshot request for source files
  2. master revokes all leases of affected chunks
  3. master logs the operation to disk
  4. master duplicates metadata of source files, pointing to the same chunks, increasing the reference count of the chunks
  5. New chunks are created on write request
  6. Perform write on new created chunk

# Snapshot



# *Summary of GFS*

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
  - designed to tolerate frequent component failures
  - uniform logical namespace
  - optimize for huge files that are mostly appended and read
  - relax and extend FS interface as required
  - relaxed consistency model
  - go for simple solutions (e.g., single master, garbage collection)

Motivation & Data Model

Storage

Operations

Conclusion

# ***BIG TABLE (NOSQL)***

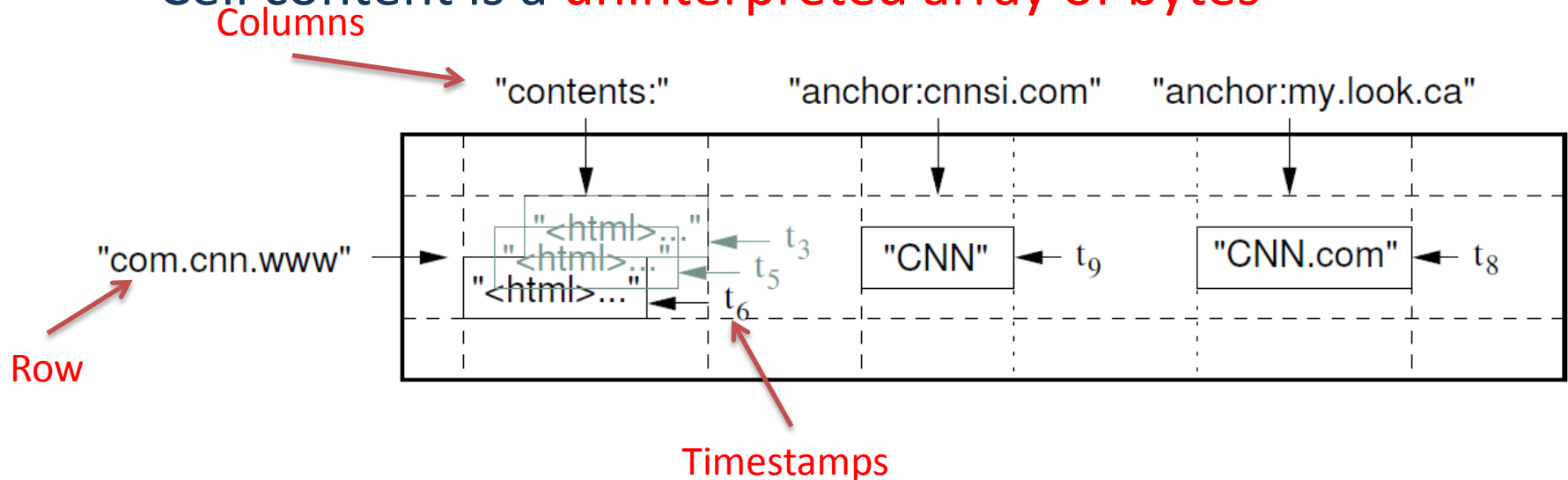


# *Motivation*

- Lots of (semi-)structured data at Google
  - Web: 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 of queries/sec
  - 100TB+ of satellite image data

# Data Model

- Semi-structured: multi-dimensional **sparse map (key-value pair)**
  - (row, column, timestamp) → cell contents
- Cell content is a **uninterpreted array of bytes**



- Good match for most of Google's applications

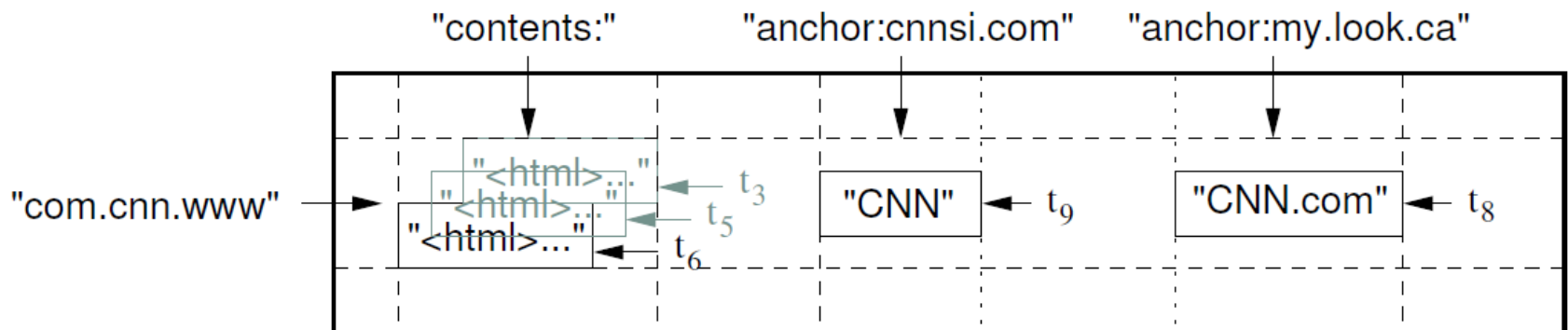
# Rows

- Name is an arbitrary string
  - Access to data in a row is **atomic**
  - Row creation is implicit upon storing data
- Rows ordered lexicographically by key
  - Rows are **dynamically** partitioned and **stored on multiple machines**
  - Clients can exploit this property to get good **data locality**
  - E.g.: reversing the hostname components of the URLs.
    - pages from the same domain are stored in a **single tablet server**

| Key                       | contents | ..... |
|---------------------------|----------|-------|
| maps.google.com/index     |          |       |
| calendar.google.com/index |          |       |
| gmail.google.com/index    |          |       |

# Columns

- Columns have two-level name structure
  - ***family:qualifier***
- Column family
  - All the data in a column family is stored into a single **GFS file** (i.e. data stores in **column major** instead of row major)
  - Basic unit of **access control**: readOnly, writeAble, etc
  - Has associated type information
- Qualifier gives **unbounded number of columns**
  - **But number of family has limit**



# *Timestamps*

- Used to store different **versions** of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
  - Sorted in decreasing order → latest version on top
- Lookup options
  - “Return most recent K values”
  - “Return all values in timestamp range (or all values)”
- Column families can be marked with attributes
  - “Only retain most recent K values in a cell”
  - “Keep values until they are older than K seconds”

Motivation & Data Model

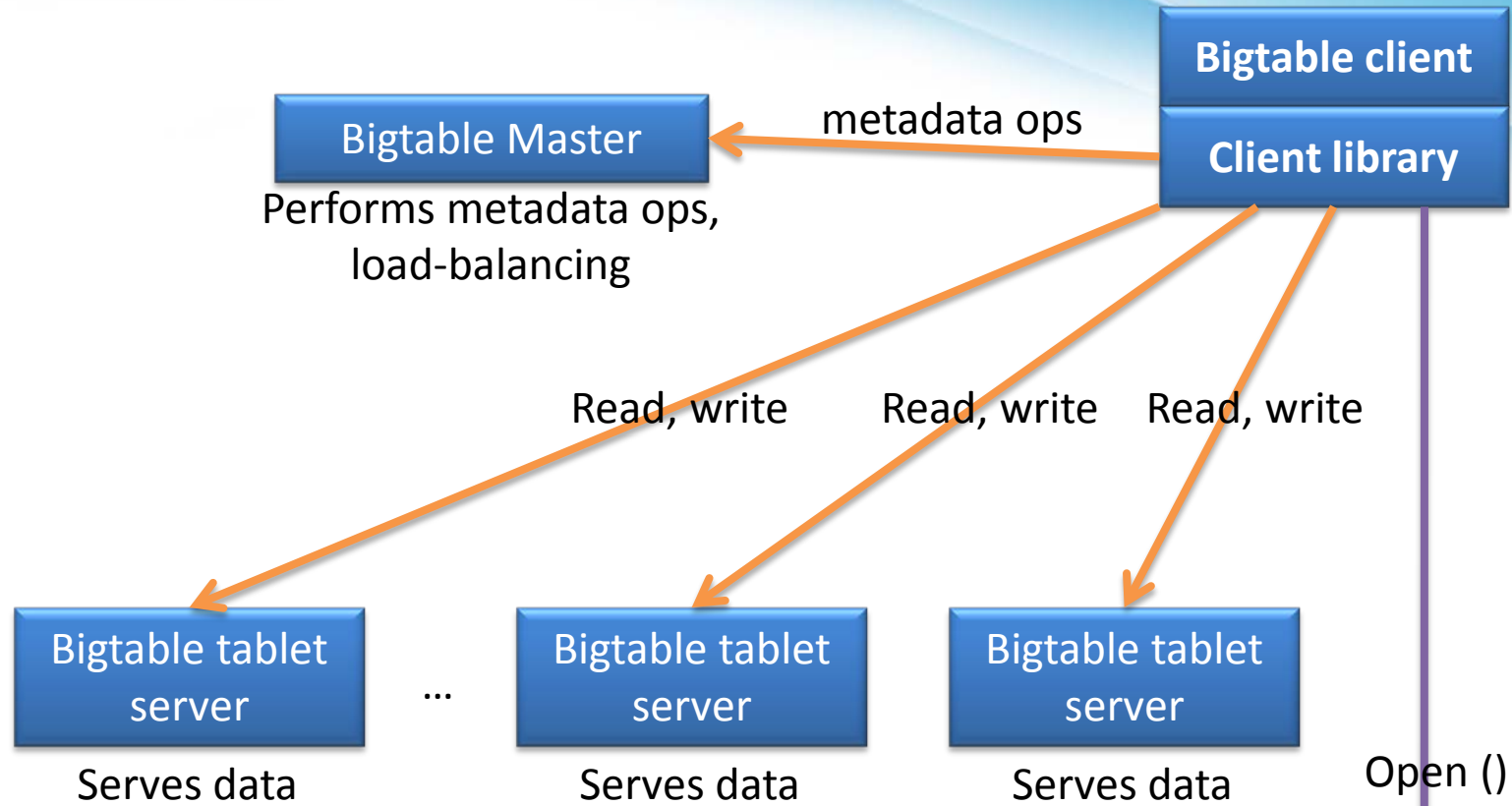
Storage

Operations

Conclusion

# ***BIG TABLE (NOSQL)***

# System Architecture



Cluster scheduling system

Handles failover,  
monitoring

Google File system (GFS)

Holds tablet data, logs  
NTHU CS542000 Cloud Programming

Lock service(Chubby)

Holds metadata, handles  
master election

# Tablets & Splitting

“language:”

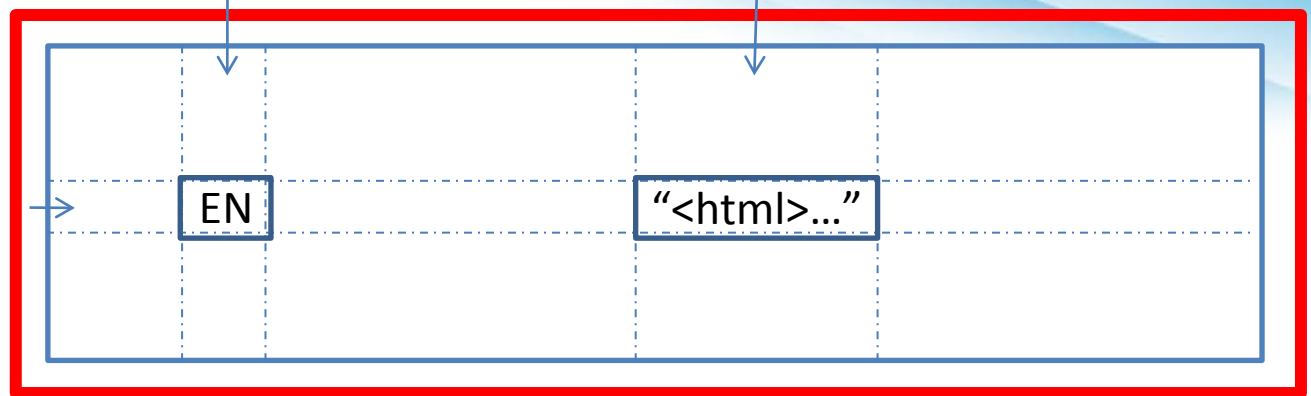
“contents:”

“aaa.com”

“cnn.com”

“cnn.com/sports.html”

Tablets



...

“website.com”

...

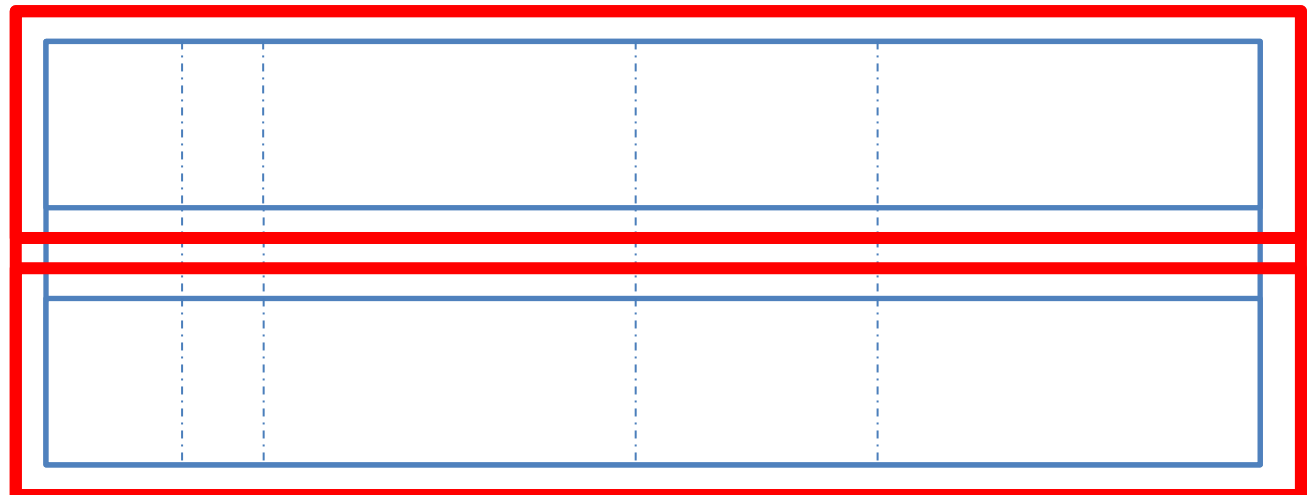
“yahoo.com/kids.html”

...

“yahoo.com/kids.html\0”

...

“zuppa.com/menu.html”





# Tablets

- Large tables broken into *tablets* at row boundaries
  - Tablet holds contiguous range of rows
  - Aim for ~100MB to 200MB of data per tablet
- Dynamic fragmentation of rows
  - Distributed over tablet servers: each responsible for ~100 tablets
  - Unit of load balancing: Migrate tablets away from overloaded machine
  - Tablets split and merge
    - automatically based on size and load
    - or manually

# How Tablet is Stored?

| Row Key       | Time Stamp | Column "contents:" | Column "anchor:"    |           | Column "mime:" |
|---------------|------------|--------------------|---------------------|-----------|----------------|
| "com.cnn.www" | t9         |                    | "anchor:cnnsi.com"  | "CNN"     |                |
|               | t8         |                    | "anchor:my.look.ca" | "CNN.com" |                |
|               | t6         | "<html>..."        |                     |           | "text/html"    |
|               | t5         | "<html>..."        |                     |           |                |
|               | t3         | "<html>..."        |                     |           |                |

## Conceptual View



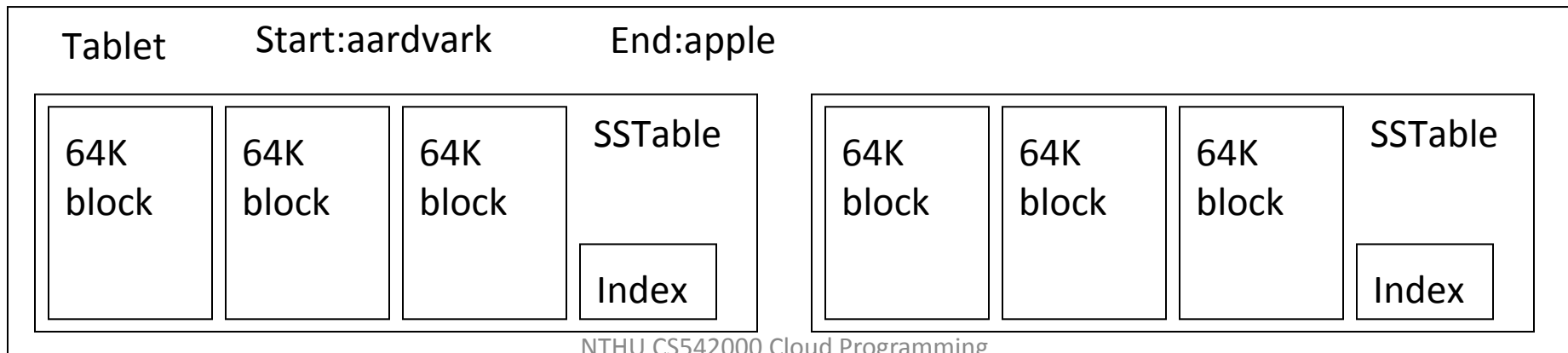
| Row Key       | Time Stamp | Column "contents:" | Row Key       | Time Stamp | Column "mime:" |
|---------------|------------|--------------------|---------------|------------|----------------|
| "com.cnn.www" | t6         | "<html>..."        | "com.cnn.www" | t6         | "text/html"    |
|               | t5         | "<html>..."        |               |            |                |
|               | t3         | "<html>..."        |               |            |                |

## Physical Storage View

| Row Key       | Time Stamp | Column "anchor:"    |           |
|---------------|------------|---------------------|-----------|
| "com.cnn.www" | t9         | "anchor:cnnsi.com"  | "CNN"     |
|               | t8         | "anchor:my.look.ca" | "CNN.com" |

# SSTable

- Tablet is built out of SSTables (GFS File)
  - Persistent, **ordered, immutable** <key, value> pair
  - Contains a sequence of blocks
    - Operations: lookup(key), iterate(key\_range)
  - A block index store the block location of each keys
    - Index loaded into memory → One disk seek per block read
    - The whole SSTable can also be mapped into memory



Motivation

Data Model

Implementation

**Operation**

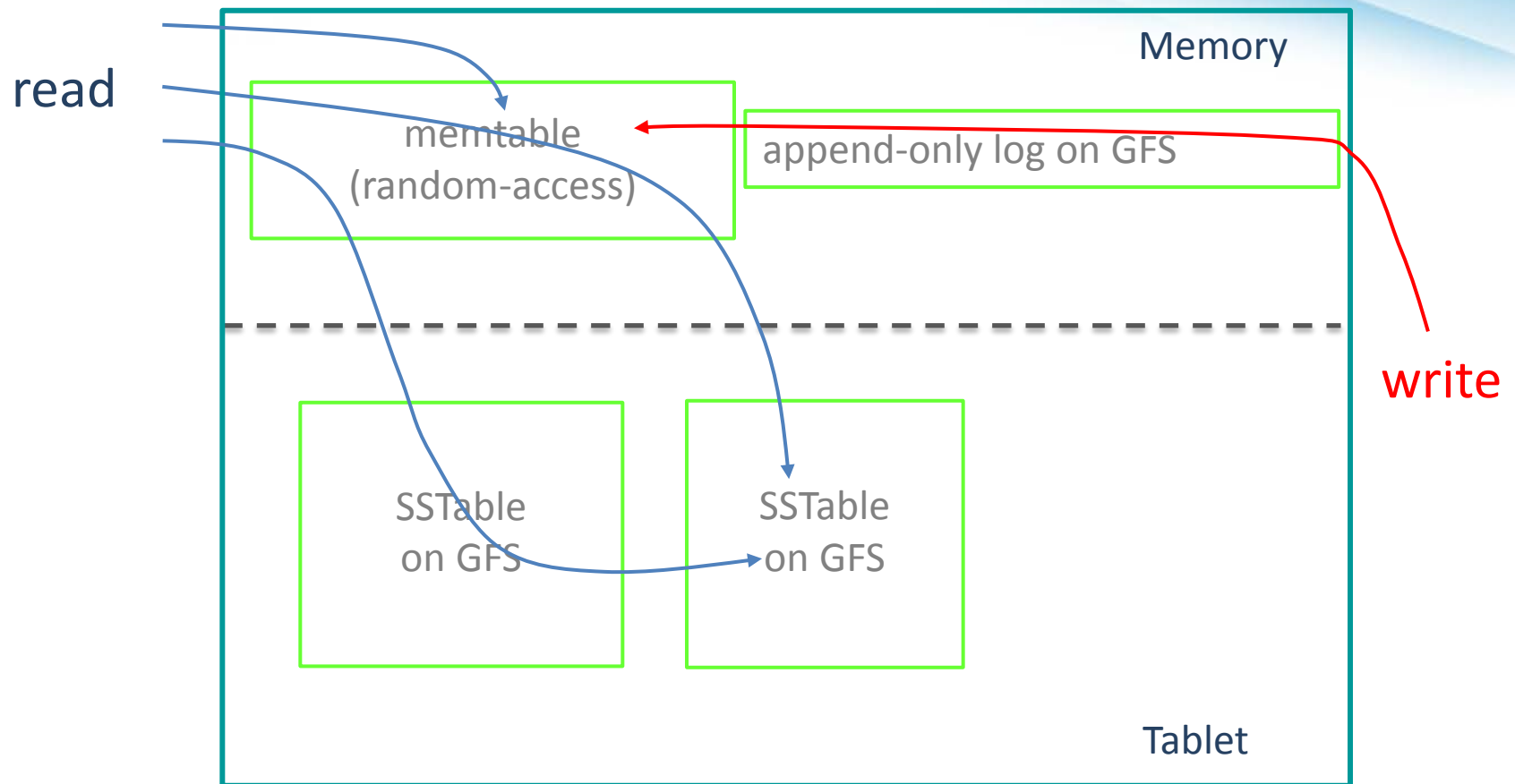
Conclusion

# ***BIG TABLE (NOSQL)***

# *Sequential Write*

- BigTable is highly optimized for **write**
  - **Appending a transaction entry to a log file**
    - the disk write is **sequential** with no disk seek
  - Write the data into an in-memory **Memtable**
    - When memtable fills up, it is frozen and **converted to an SSTable written to GFS at once**
  - When machine crashes
    - Reconstruct Memtable by **replaying the updates in the log file.**

# Tablet Serving



SSTable: Immutable on-disk ordered map from string->string  
string keys: *<row, column, timestamp> triples*

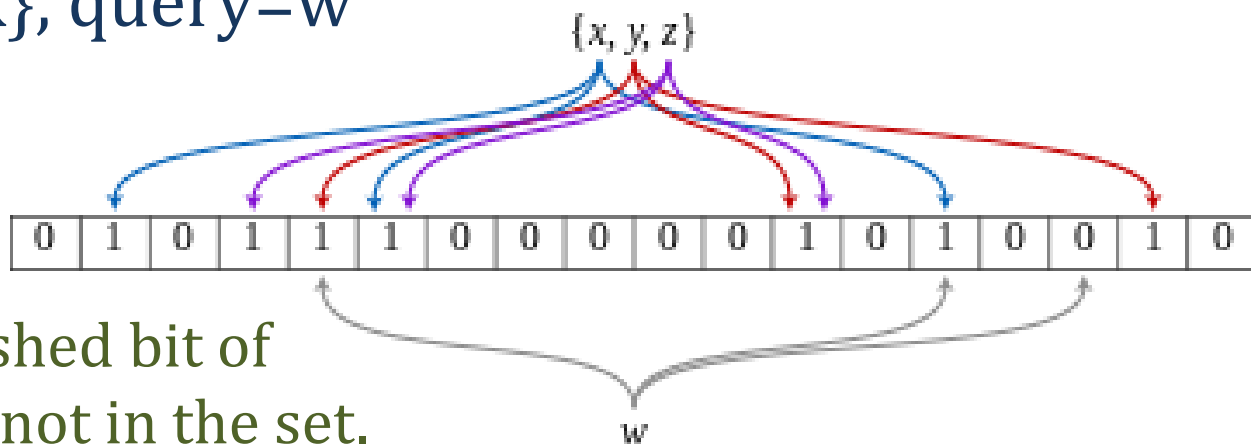
# Merged Read

- "Merged Read" means multiple places to lookup data when a read request is arrived.
  - It first looks at the Memtable by the row key of request. If not, it will look at the on-disk SSTables
  - Multiple SSTables are created for a tablet due to **write**
- It is inefficient for read when there are too many SSTables scattering around.
  - To speed up the detection, SSTable has a companion **Bloom filter** such that it can rapidly detect the absence of the row-key.
  - The system periodically merge the SSTables.

# Bloom Filter

- Conceived by Burton Howard Bloom in 1970
- A probabilistic data structure that is used to test whether an element is a member of a set.
  - **False positives** are possible, but not **false negatives**.
- Ex: a set= $\{x, y, z\}$ , query= $w$

- Elements are hashed to different bits

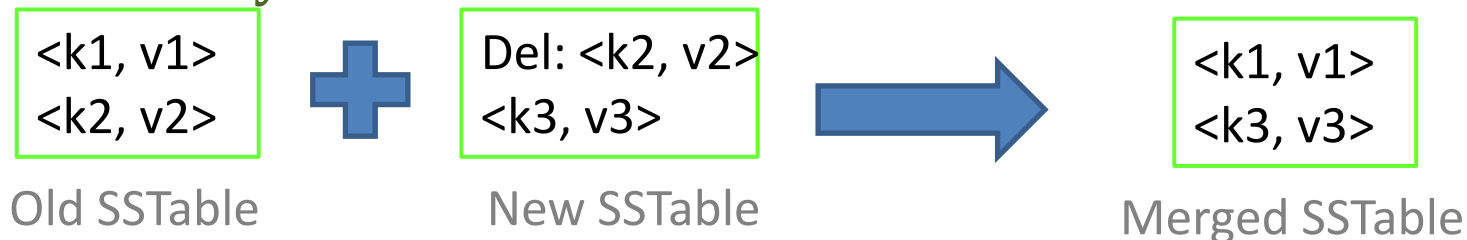


- One of the hashed bit of  $w$  is 0, so it is not in the set.

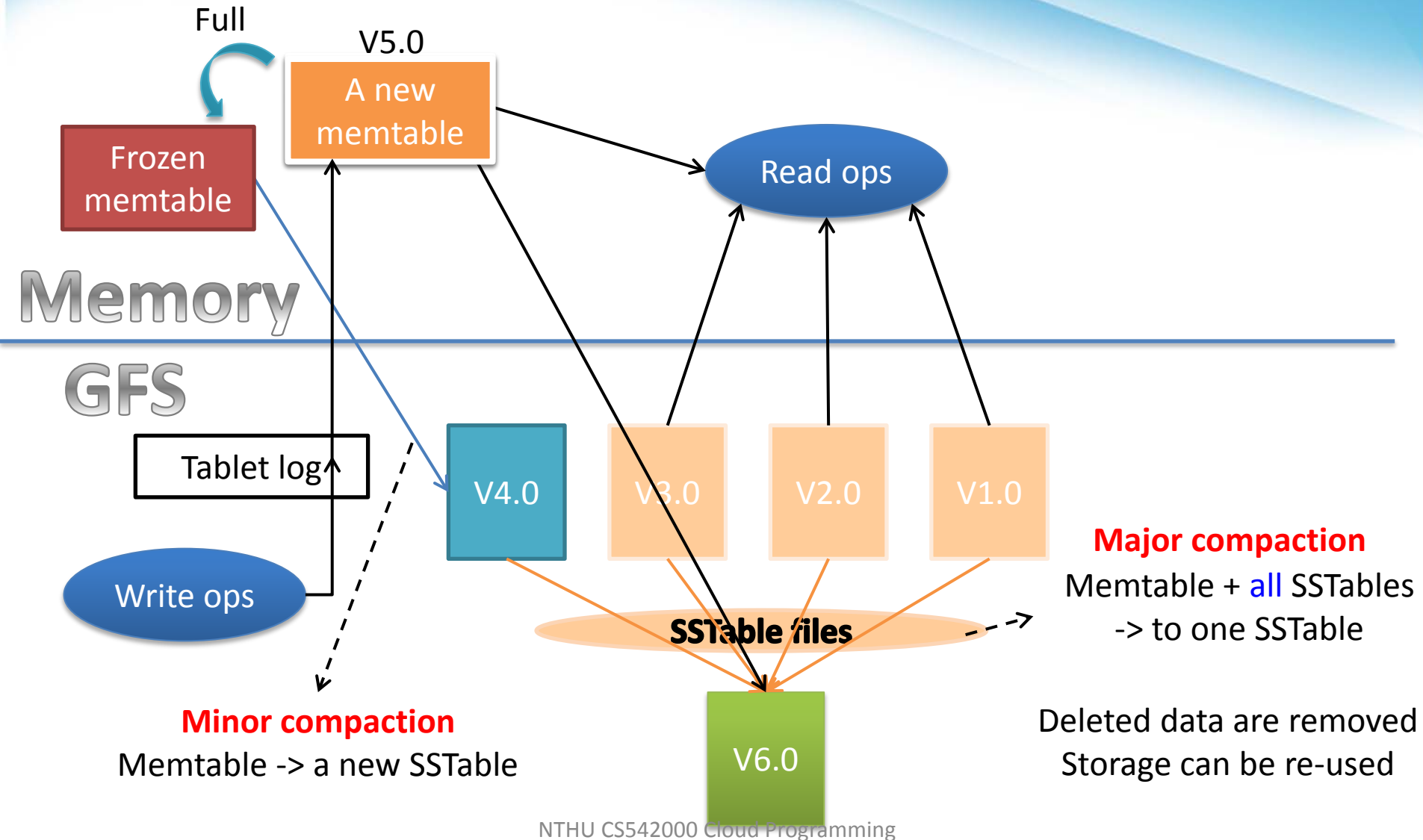


# Periodic Data Compaction

- SSTables are merged periodically
  - Each SSTable is individually **sorted by key**
  - A simple "merge sort" is sufficient to merge multiple SSTable into one
- Deleted entries/data are removed
  - Reclaim resources used by deleted data
  - Ensure delete data disappears from the system in a timely fashion



# Compactions Example



Motivation

Data Model

Implementation

Operation

Conclusion

# ***BIG TABLE (NOSQL)***

# *Summary of Bigtable*

- Data model applicable to broad range of clients
  - Actively deployed in many of Google's services
- System provides high performance storage system on a large scale
  - Self-managing
  - Thousands of servers
  - Millions of ops/second
  - Multiple GB/s reading/**writing**

# *Differences from RDBMS*

- Transaction protection is only guaranteed within a single row, not multiple rows
- Inconsistency
  - While you are reading a row, other people may have modified the same row and update it before you. Your view is not current anymore but your later update can easily wipe off other people's change.
- Limited SQL query supported due to effectiveness
  - Such as Insert & Join
- Lack of surrounding tools

# *Differences from RDBMS (cont')*

- No indexes
  - There is no index from the column value to its containing rowid.
  - If users require index in Bigtable, they need to build their own index at the application level.
  - If users build artificial index at the application level, they need to maintain the integrity of index when the base data is inserted, modified or deleted.

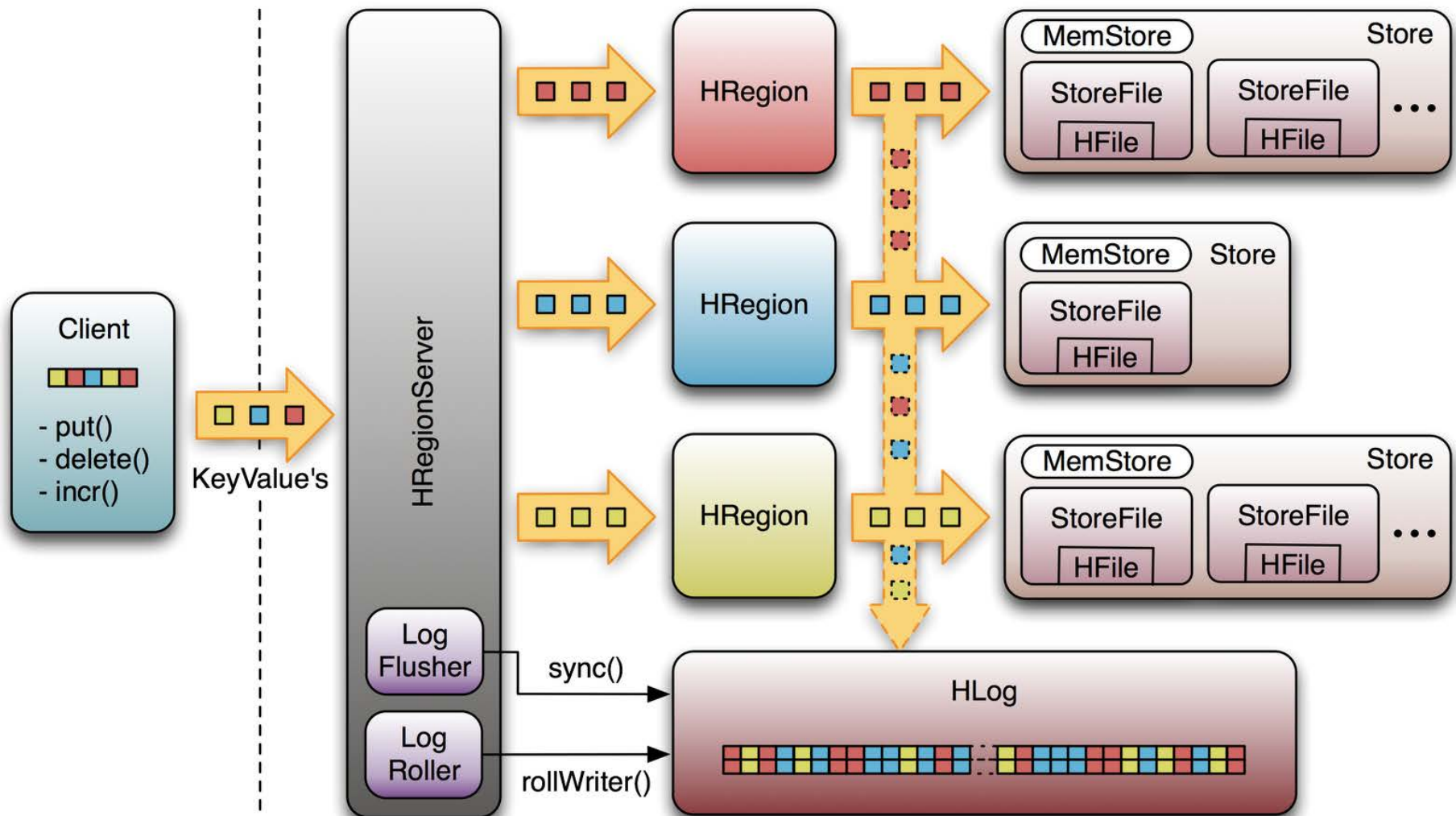
- Based on the BigTable, HBase uses the Hadoop Filesystem (HDFS) as its data storage engine.
- HBase doesn't need to worry about data replication, data consistency and resiliency because HDFS has handled it already.
  - It is also constrained by the characteristics of HDFS, which is not optimized for random read access.
  - There will be an extra network latency between the DB server to the File server (which is the data node of Hadoop).

# *BigTable v.s. HBase*

| BigTable           | HBase         |
|--------------------|---------------|
| Master             | Master        |
| Tablet server      | Region server |
| Google file system | HDFS          |
| SSTable            | HFile         |
| Chubby             | Zookeeper     |
| Memtable           | Memcached     |



# How does Hbase work?



# References

- Nancy Lynch and Seth Gilbert, “Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”, *ACM SIGACT News*, Volume 33 Issue 2 (2002), pg. 51-59.
- S. GHEMAWAT, H. GOBIOFF, and S.-T. LEUNG, “The Google file system,” In *Proc. of the 19th ACM SOSP (Dec. 2003)*
- Chang, F., et al. “Bigtable: A distributed storage system for structured data.” In *OSDI* (2006).
- Hbase.
  - <http://hbase.apache.org/>
- NCHC Cloud Computing Research Group.
  - <http://trac.nchc.org.tw/cloud>
- NTU course- Cloud Computing and Mobile Platforms.
  - [http://ntucsiecloud98.appspot.com/course\\_information](http://ntucsiecloud98.appspot.com/course_information)
- Wiki.
  - [http://en.wikipedia.org/wiki/Database#Database\\_management\\_systems](http://en.wikipedia.org/wiki/Database#Database_management_systems)