# Database - P31

HDFS (Hadoop Distributed File System), 即 Hadoop 分布式文件系统
separate from the Linux File System (Linux FS), with similar tree structure
can be accessed from Hadoop or Spark program
if not specified, HDFS is default when [redacted] on system with HDFS
[redacted] explicitly specify the file type:
HDFS : hdfs:///u/home/username/filename
Linux : file:///u/home/username/filename
file should be specified by complete absolute path name (完整的绝对路径)

HDFS 文件[redacted]有两类地址
hdfs:///u/home/$USER, 与 Linux 系统中的 home 文件夹一致
use hdfs://$PWD as working HDFS directory corresponding to Linux directory
hdfs:///user/$USER, 用于[redacted] Spark 的临时文件
default HDFS directory if no absolute pate specified

HDFF 文件/文件夹的所有与许可 (file/directory ownership and permission)
与 Linux FS 的定义类似.
r: read permission (可读)
w: write permission (可写)
x: execute permission (可执行), 对于 HDFS directory 为 traverse, 遍历)

命令行 hadoop [redacted] fs [redacted] -<operation> 用于处理 HDFS
-<operation> 为执行的指定命令

hadoop fs -help <operation> : help information for the specified operation
hadoop fs -cp <src> <dest> : copy file
hadoop fs -get <src> <dest> : copy from HDFS to local
hadoop fs -put <src> <dest> : copy from local to HDFS
hadoop fs -ls <path> : list files
hadoop fs -mv <src> <dest> : move file within HDFS
hadoop fs -rm <path> : remove file
hadoop fs -chmod <perm> <path> : change file/directory permission

# Database - P32

GFS (google file system)

motivation
- component failures are common : data center built from commodity parts
  - large number of machine mean the probability of failures is non-negligible
- huge file : multi-gigabyte to terabyte sized files are common
- most changes appends : random writes uncommon
  - file reay times after writen, usually sequentially

property
- constant monitoring (持续监控)
- error detection (错误检测)
- fault tolerance (差错容忍)
- automatic recovery (自动恢复)

- customize feature for common usage
  - caching block not useful for sequential access
  - relaxed consistency model (宽松的一致性模型)
  - atomic append ()原子性附加标签)

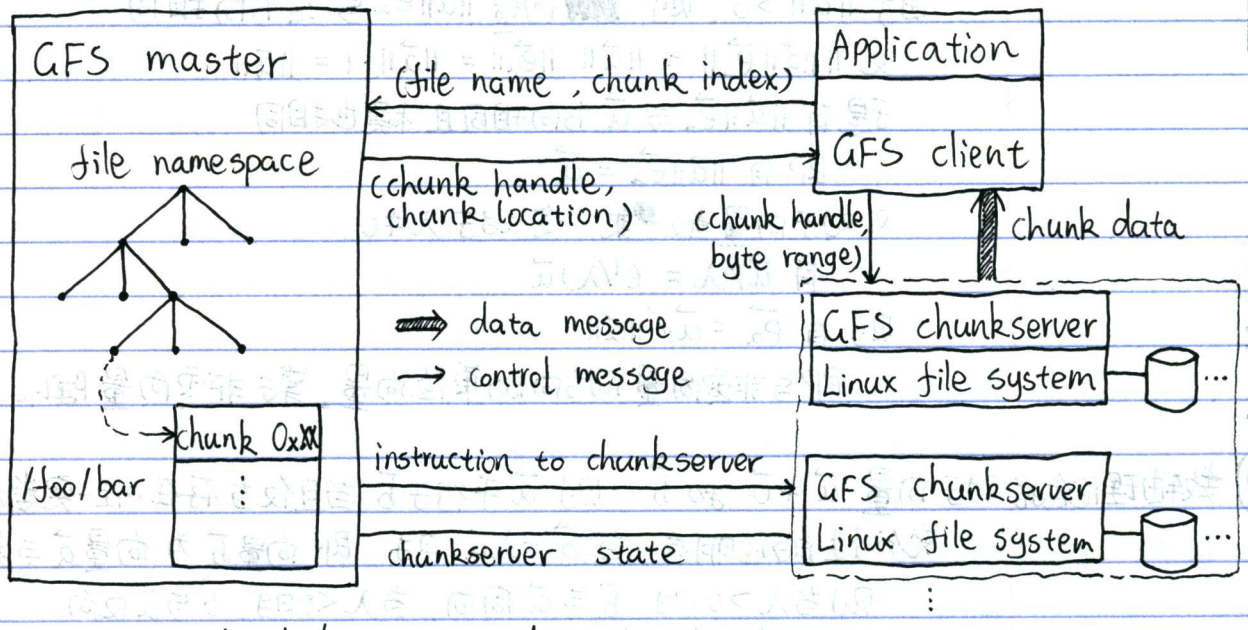
assumption
- failure common : must tolerate and recover from fault routinely and quickly
- optimize for smaller numbers of very large files
  - small files supported, may be inefficient
- workload reads are primarily : large stream reads (> 1MB)
  - small numbers of small random read
- workload writes are primarily : large, sequential, appending to end of file
  - data seldom modified after written
  - random writes supported, need not be efficient
- Semantics for multiple client appends
  - merge writes (appends) from many writers
  - atomic appends with minimal synchronization overhead
- high sequential read/write bandwidth more important than low latency

interface
- not implement standard UNIX/Linux file interface
- interface is through dedicated library API

# Database - P33

interface
operation : create , delete , open , close , read , write
snapshot : create copy of file or directory tree at low cost
record append : atomic appends from multiple clients

GFS architecture




chunk : files divided into fixed-size chunks
each has unique 64-bit chunk handle
replicated on multiple chunkservers

master : maintain all metadata :
namespace and access control lists (ACLs)
mapping from file to list of chunk handle
chunk location
control system-wide activities :
chunk lease management
orphaned chunk garbage collection
chunk migration between chunk servers

client : communicate with both master and chunkservers

data transfer : done directly between client and chunkservers
not go through the master

# Database - P34

steps for client data read:
- compute index of chunk within file
- send master file name and index of chunk (chunkserver name)
  - master replies with chunk handle and locations of replicas
- cache handle and chunkserver names
- choose chunkserver (possibly nearest)
- send read request with chunk handle and byte range to read from chunk
- chunkserver sends data to client

chunk size
- typical chunk size 64MB (2003), may be larger now due to bigger disk and faster network
- advantage of larger chunk size:
  - fewer interaction between client and master
    - multiple operations on single larger chunk (keep connection open)
      - allow fewer new network connection from client to chunkserver
  - reduce size of metadata on master
    - can keep more of the metadata in memory

metadata
- kept in memory on the master
  - fast access for better performance
  - fast scanning for garbage collection, re-replication on failure and chunk migration for load balancing
- types of metadata: namespace
  - file-to-chunk mapping
  - location of all replicas of each chunk

- changes to namespace (directory structure) and file-to-chunk mapping
  - also logged to an operation log on the master disk
  - operation log also replicated on remote machines
  - periodic checkpoints of memory state are taken
  - in case of failure, recovery load the latest checkpoint then replay and apply the operation log
  - chunk locations are recovered by polling chunkservers

# Database - P35

GFS consistency model
- namespace mutation (file creation) : atomic
- data mutation :
  - consistent file region : all client always see same data from all replicas
  - defined file region : consistent and client will see data just written
  - undefined file region : multiple concurrent writes (still consistent)

writes and record appends
- writes put data at position in file specified by client
  - regular append : write to where the client think the end of file located
- record appends cause data to be appended atomically (at least once)

consistency after mutation, mutated region
- is guaranteed to be defined and contain data written by last mutation
  - mutation applied in same order to all replicas
  - chunk version number used to detect stale replicas
    - for chunk not been updated

best practice for relaxed consistency
- client : rely on record append rather than write
  - write self-identifying, self-validating record
    - handled by client library, part of API
- typical use : multiple concurrent writers use record append
  - readers deal with occasional padding and duplicates

leases, mutation order, writing
- mutation performed on all replicas of chunk
- master assign chunk lease (with timeout) to primary replica
  - typical lease timeout : 60s, renewable with activity
  - primary determine order for all mutations of chunk
  - other replica get ordering from primary

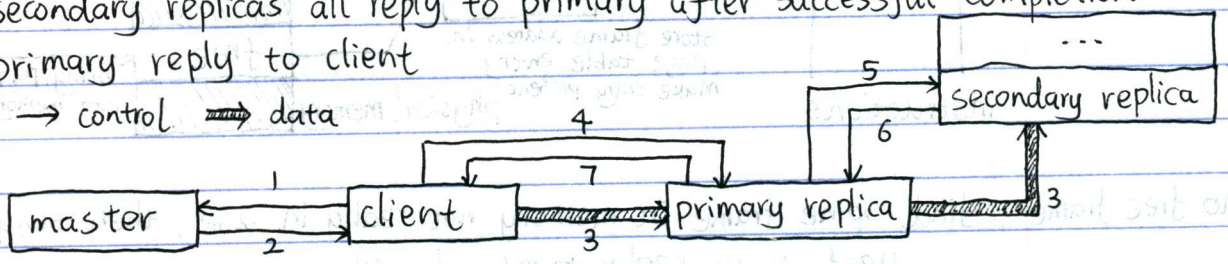
atomic record append : guarantee contiguous, defined data with single chunk
- similar to write
- data size limited to fraction of chunk size

# [redacted] and [redacted] Database - P36

## [redacted] * [redacted]

writing to a chunk

1. client ask master for address of primary for chunk
   master assign lease to a chunkserver if no active lease
2. master give client address of primary and all replicas
   client cache for future mutations of this chunk
3. client push data (serially) to all replicas (order not important)
4. client send write request to primary after all chunkserver acknowledge receipt
   primary decide on place in order for mutation
5. primary forward write request and ordering to secondary replica chunk servers
6. secondary replicas all reply to primary after successful completion
7. primary reply to client




snapshot make 'copy' of file by creating new namespace (directory) entry
pointing to the same chunks
original and snapshot become 'copy-on-write'
any chunk held by both is duplicated on write
any appended chunks only added to one of the files
concept similar to memory handling for Unix/Linux fork() function call

chunk replica placement : disk space utilization balancing
write request load balancing
spread replicas across different racks and aisles of data center
periodic reblancing / re-replication of lost replicas

deletion and garbage collection : deleted file initially just renamed folder on conventional OS
after days, file removed from namespace, orphan the chunks
chunkservers periodically report to master on subset of chunk holdings
master replies with chunks no longer in use

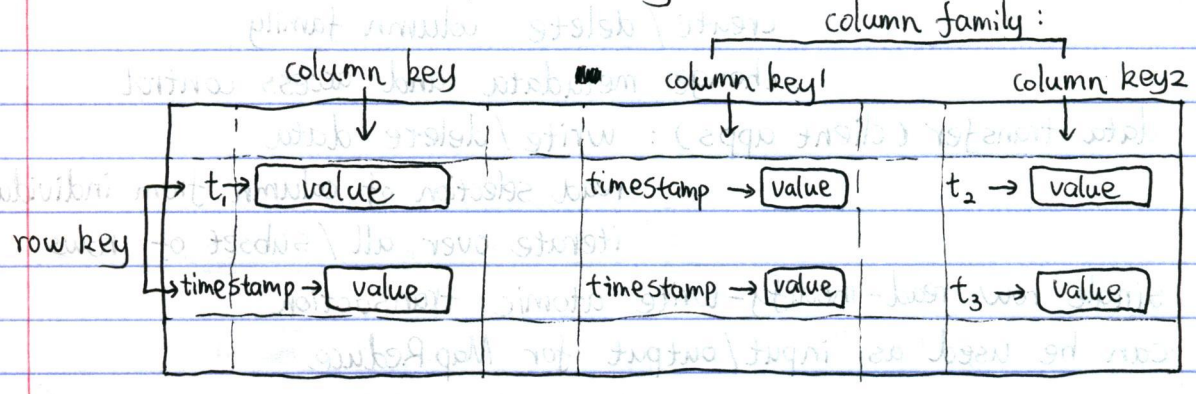
# Database - P37

master replication, master log and checkpoints replicated on multiple machines for reliability
mutation committed only after log and checkpoints replicated on multiple machines
can restart master on another machine by reading checkpoints and log
shadow masters provide read-only access when master is down
update from copy of master log

HDFS/GFS : HDFS design modeled on GFS
HDFS 'namenode' ⟷ GFS 'master'
HDFS 'datanode' ⟷ GFS 'chunkserver'

BigTable distributed structured storage
scale to petabytes of data across thousands of servers
NoSQL : similar to database, but not support retional model
simple data model : indexed using row/column name carbitrary string)
sparse data storage (empty column in row take no space)
data item : uninterpreted string (byte[])
client have control over data locality
can serve data from memory/disk per schema

data model : sparse, distributed, persistent, multi-dimensional, sorted map
indexed by row key, column key, timestamp (时间戳)
column grouped in family
(row key :: byte[], column key :: byte[], timestamp :: int64)
⟶ value :: byte[]
value is uninterpreted array of byte



# Database - P38

**Row**

row key is arbitrary string : byte[]
- atomic read/write of any row : may read/write subset of columns

maintain in key lexicographic order

tablet : Bigtable dynamically partitioned into group of rows
  unit of distribution and load balancing

read contiguous range of rows typically efficient
  require communication with limited number of tablet servers
  application can select row key to control locality

**Column**

column key grouped into Column Family : <family>:<qualifier>

column family : unit of access control
  access optimized for access
    item in a row in same column family compressed together
  must be explicitly created
    set of column family in Bigtable
      should be relatively stable, not very large
    column within one family can be created dynamically
      number of column unbounded

**Timestamp**

one cell can contain multiple version of data : indexed by timestamp
  creation clock time in microsecond / version number assigned by application

application can control garbage collection
  item expire after certain time / only certain most recent versions kept

on read, application specify timestamp / ask for most recent version

**API**

structural control : create / delete table
  create / delete column family
  change metadata and access control

data transfer (client apps) : write / delete data
  read selection of column from individual row
  iterate over all / subset of rows

single row read-modify-write atomic transaction

can be used as input/output for MapReduce

# Database - P39

SSTable data storage : file used internally to store persistent data
stored on GFS
persistent, ordered, immutable map from key to value
both are arbitrary byte string (byte[])
support lookup by key and iteration over key range
file contain index, loaded into memory on open
ordered index allow use of binary search
optionally load entire SSTable into memory

chubby distributed lock service used by Bigtable
namespace of directory and small file
each directory or file can be used as lock
atomic read from / write to file
session-based
callback for change or session expiration
Bigtable usage : ensure exactly one active master
store bootstrap location of Bigtable
manage tablet server
store table schema (column family info)
store ACLs

implementation major component : client library
single master server
multiple tablet servers
dynamically added/removed based on workload

Bigtable master: assign tablet to tablet server
detect addition and expiration of tablet server
balance tablet server load
garbage collect file
handle schema change

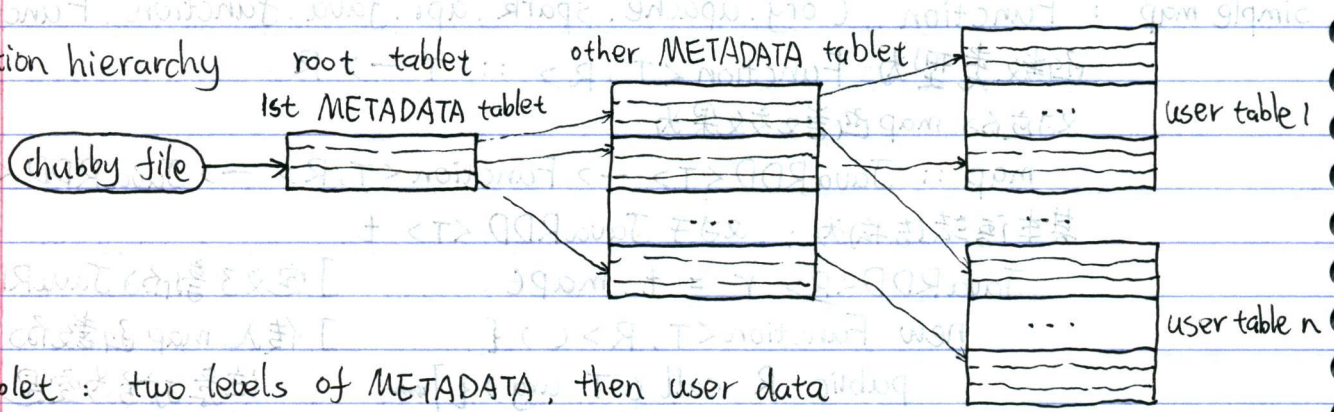
# Database - P40

Bigtable tablet server : manage set of tablet
handle read/write for each tablet
split tablet that grow to large
data directly flow between client and tablet server
no data flow through master
most clients never need to communicate with master

table/tablet : one Bigtable cluster may serve multiple tables
each table consist of set of tablets
initially single tablet
automatically split into multiple tablets as tablet grow
split by row
each tablet contain contiguous range of row

tablet splitting and merging
tablet server split tablet in two, when grow beyond specified size
update the METADATA and notify the master
serving continue uninterrupted during tablet split
deletion may shrink tablet below specified size
master can initiate merge of two adjacent tablets

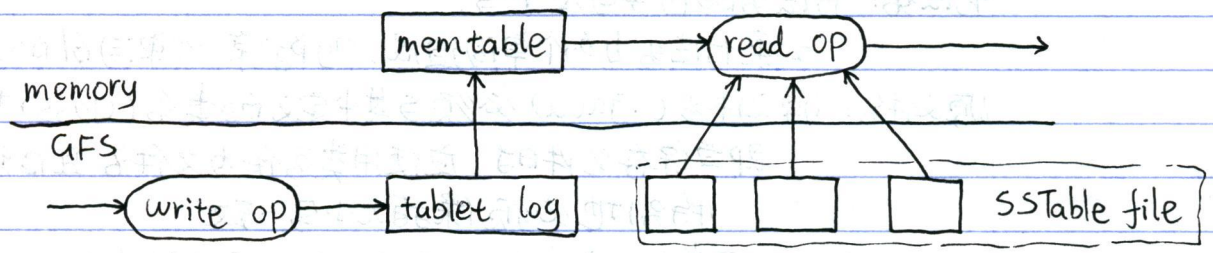
tablet location hierarchy




METADATA tablet : two levels of METADATA, then user data
typically stored in memory while active
except root, can be split as size increase
store location of tablet containing each range of user data
or lower level METADATA
client cache METADATA tablet location

# Database - P41

tablet assignment : each tablet assigned to one tablet server at one time
Master track tablet assigned to server
tablet server assignment managed by Chubby




tablet serving , memtable : tablet server keep recently updated data in memory
read : merge any update from memtable with data from GFS SSTable file
write : log the update to GFS commit log , then save update in memtable

tablet recovery , tablet server reconstruct memtable from commit log in event of restart

compaction , the memtable for tablet grow and must be compacted when update occur
read/write can still occur during compaction
update continue to new memtable
minor : current memtable converted to SSTable and written to disk
merging : merge memtable and SSTables into single new SSTable
major : merge memtable and all tablet SSTables into single new SSTable
clean out any deletion information and deleted data

table naming , table has alphanumeric name
name may be qualified by namespace
permission may be assigned by namespace
creation of table in global namespace restricted to superuser

command hbase shell : start HBase command shell
create <username>:<tablename> [,<column family>]* : create table
list <table name> : list information about table
put <table name>,<row>[,<column>,<value>]* : put data into table
scan <table name> , get <tablename>,<row> : list the data of table
disable <tablename> , drop <table name> : make table inactive and delete

# Database - P42

ACID 对于在数据库管理系统(database management system, DBMS)中
为保证写入或更新资料的事务(transaction)的正确可靠而必须具备的特性

原子性 (atomicity), 对于事务(transaction)通常由多个操作(statement)组成
guarantee that each transaction treated as single "unit"
即事务的操作视为一个单元, 或者全部完成, 或者全部不完成
如果事务执行过程中发生错误, 则执行回滚(rollback)回到事务开始前状态
也保证了事务不可分割, 不可约简
guarantee atomicity in each situation (power failure / error / crash)

一致性 (consistency), 对于数据库具有不同的有效状态(valid state)
ensure transaction can only bring database from one valid state to another
maintain database invariant (数据库完整性)
any data written to database must be valid according to all defined rules
constraint (约束), cascading rollback (级联回滚), trigger (触发器)
prevent illegal transaction, not guarantee transaction correct
referential integrity : guarantee primary key - foreign key relationship

隔离性 (isolation), 对于事务, 多个事务可能并发地执行(executed concurrently)
即数据库支持多个并发事务同时对数据的读写和修改
ensure concurrent execution leave database in the same state
would have been obtained if executed sequentially
concurrency control (并发控制) : main goal of isolation
effect of incomplete transaction might not be visible to other transaction
事务分离级别 : 读未提交(read uncommitted), 读提交(read committed)
可重复读(repeatable read), 串行化(serializable)
防止多个事务并发执行时由于交叉执行而导致的数据不一致

持久性 (durability), 对于已提交的操作处理完成后, 其修改是永久的
guarantee once transaction committed,
remain committed even in case of system failure (power outage / crash)
completed transaction / effect recorded in non-volatile memory