Key-Value stores (BigTable)

Knowledge objectives

1. Explain the structural components of HDFS
2. Explain how to avoid overloading the master node in HDFS
3. Explain the structural components of HBase
4. Explain the main operations available in HBase
5. Compare relational and co-relational data models
6. Explain the role of the different functional components in Hbase
7. Explain the tree structure of data in Hbase
8. Explain the cache mechanism of Hbase client
9. Compare a distributed tree against a hash structure of data
10. Explain the four kinds of replication protocols
11. Explain the three possible scenarios identified by the CAP theorem

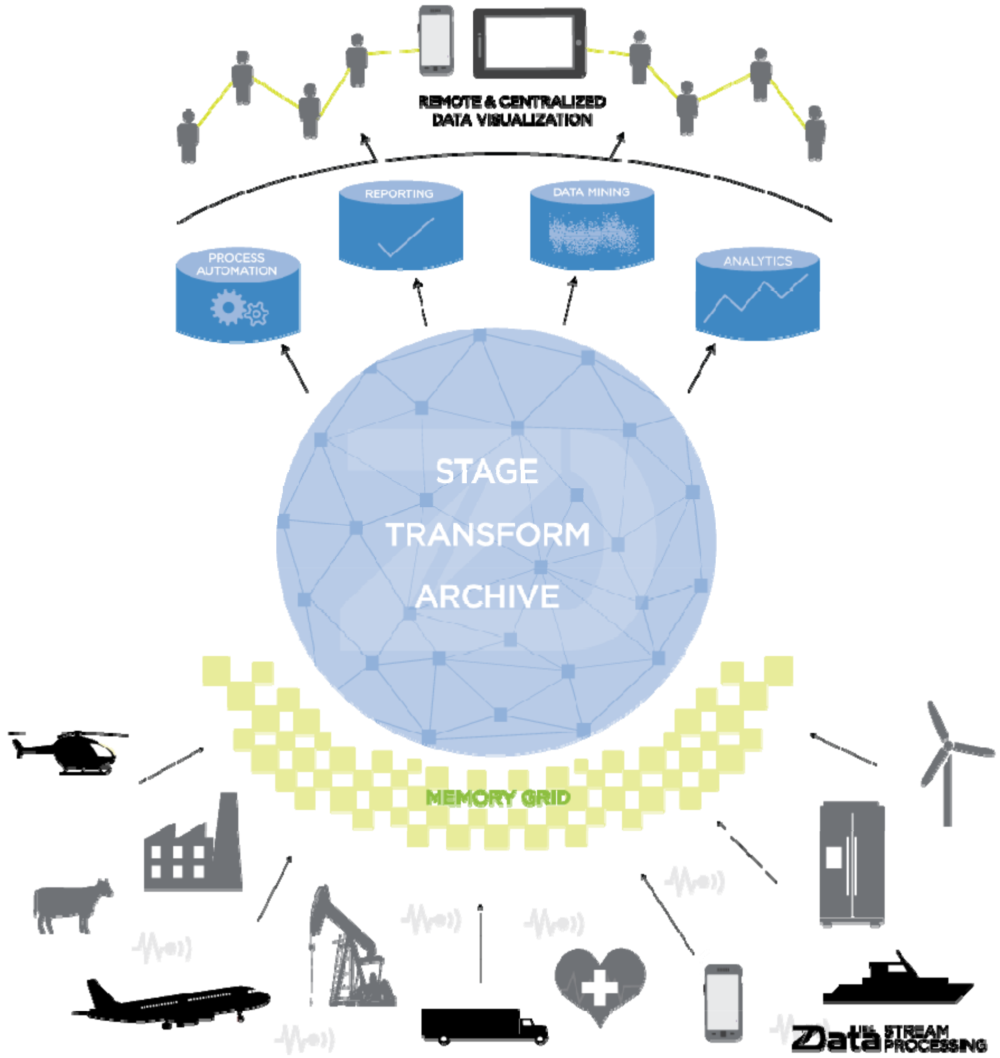
Understanding Objectives

1. Calculate the number of round trips needed in the lazy adjustment of a directory tree
2. Add a new bucket in Linear Hashing
3. Add a new node in Consistent Hashing
4. Decide the number of needed reads and writes to guarantee consistency in the presence of replicas

Goals

- Schemaless
  - No explicit schema
- Easy setup and scalability
  - Continuously evolve to support a growing amount of tasks
- Efficiency
  - How well the system performs, usually measured in terms of response *time* and *throughput*
- Reliability/Availability
  - Keep delivering service even if one of its software or hardware components fail
    - Comes to the price of relaxing consistency
- Simple usage
  - Put and Get operations

Data Lake: Load-First, Model-Later



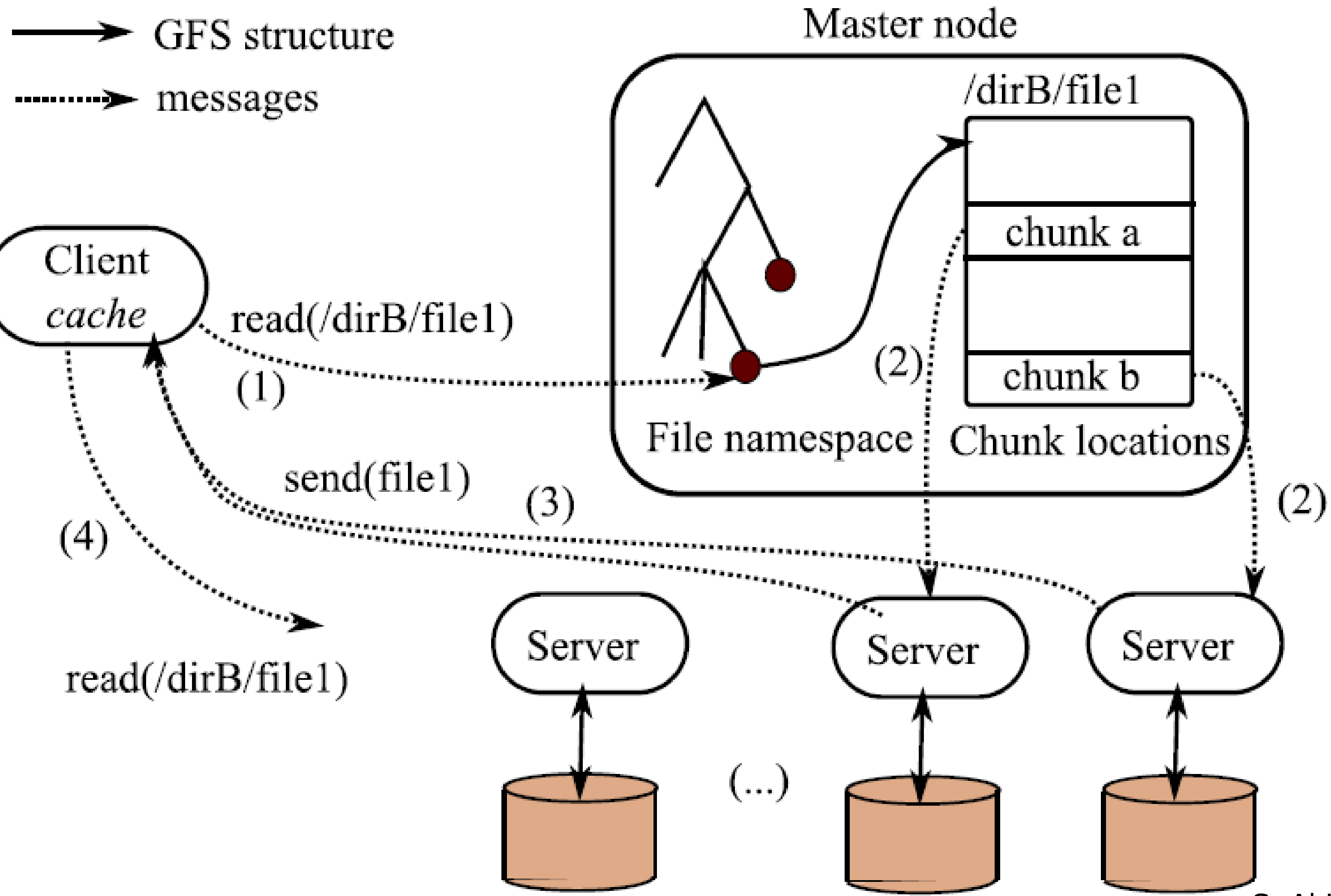
Hadoop File System (HDFS)

- Apache project
  - Based on Google File System (GFS)
- Designed to meet the following requirements:
  a) Handle very large collections of unstructured or semi-structured data
  b) Data collections are written once and read many times
  c) The infrastructure underlying consists of thousands of connected machines with high failure probability
- Traditional network file systems do partially fulfil these requirements
  - Operating Systems Vs. Database Management System
    - Balancing *query* load (e.g., by means of *fragmentation and replication*) boosts availability and reliability
      - HDFS: Equal-sized file chunks evenly distributed

HDFS in a Nutshell

- A single master (coordinator)
  - Receives client connections
  - Maintains the description of the global file system namespace
  - Keeps track of file chunks (default: 64Mb)
- Many servers
  - Receive file chunks and store them
- A single master design forfeits availability and scalability
  - Availability and reliability: Recovery system
    - Replication (a chunk always in 3 servers, by default)
    - Monitors the system with heartbeat messages to detect failures as soon as possible
    - Specific recovery system to protect the master
  - Scalability: Client cache

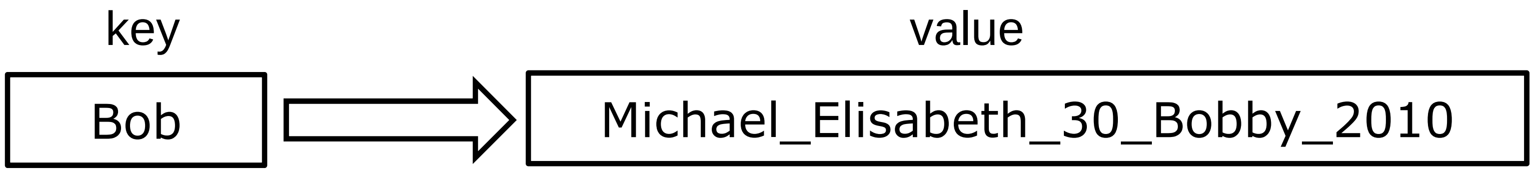
HDFS client cache



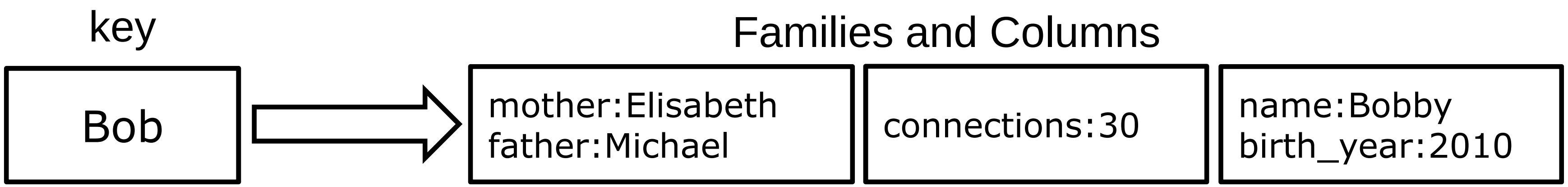
S. Abiteboul et al.

Key-Value

- Key-value stores
  - Entries in form of key-values
    - One key maps only to one value
  - Query on key only
  - Schemaless



- Column-family key-value stores
  - Entries in form of key-values
    - But now values are splitted in columns
  - Typically query on key
    - May have some support for values
  - Schemaless within a column

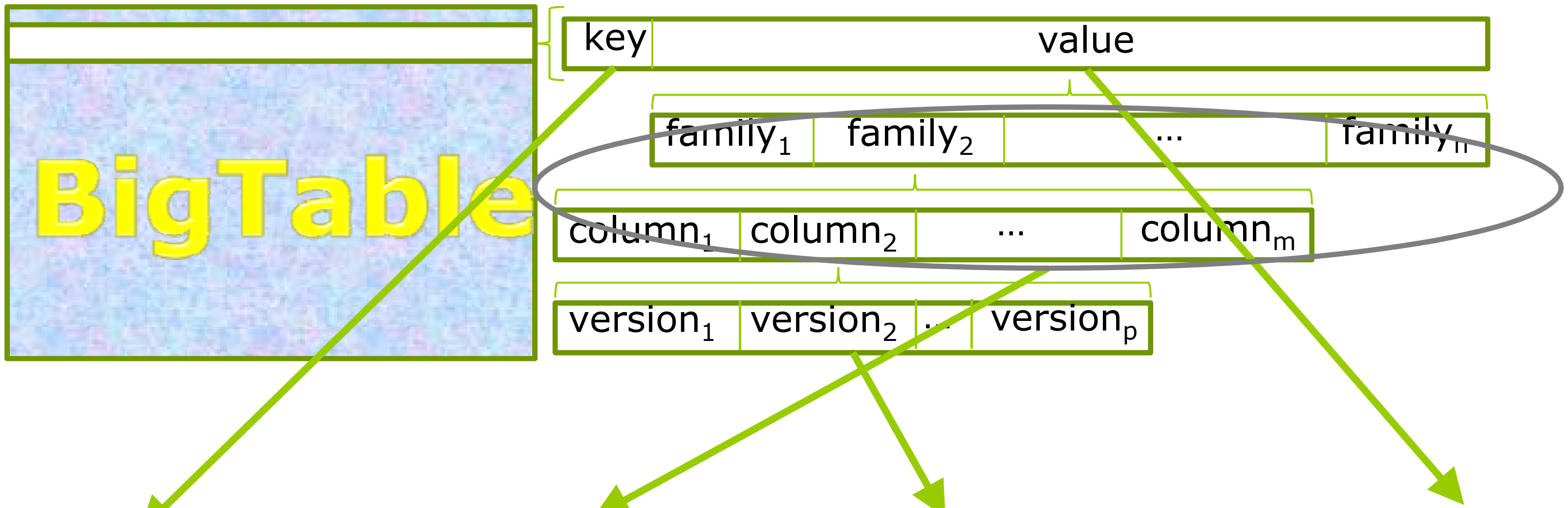


HBase

- Apache project
  - Based on Google's Bigtable
- Designed to meet the following requirements
  - Access specific data out of petabytes of data
  - It must support
    - Key search
    - Range search
    - High throughput file scans
  - It must support single row transactions
- Do it yourself database… own decisions regarding:
  - Data structure
  - Concurrency
  - Recovery availability
    - CAP trade-off

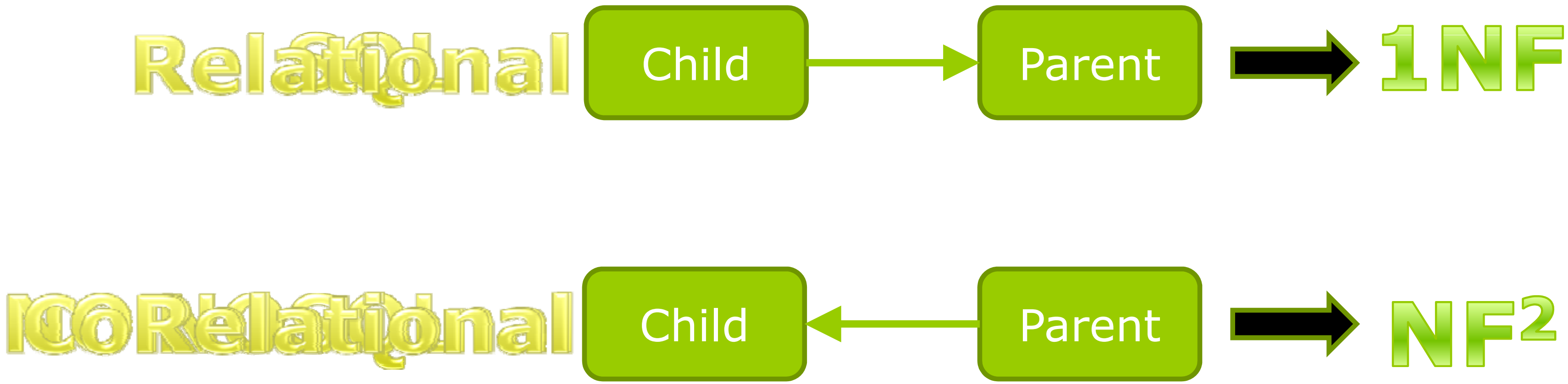
Schema elements

- Stores tables (collections) and rows (instances)
  - Data is indexed using row and column names (arbitrary strings)
- Treats data as uninterpreted strings (without data types)
- Each cell of a BigTable can contain multiple versions of the same data
  - Stores different versions of the same values in the rows
  - Each version is identified by a timestamp
    - Timestamps can be explicitly or automatically assigned



(row:string, column:string[, time:int64])→string

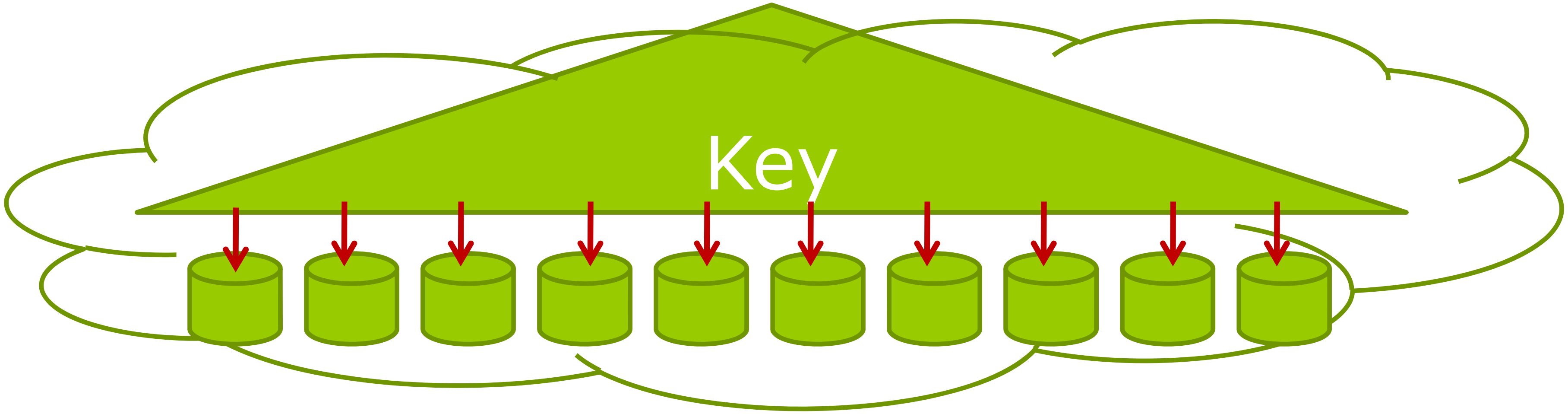
Just another point of view



HBase Shell

- ALTER <tablename>, <columnfamilyparam>
- COUNT <tablename>
- CREATE TABLE <tablename>
- DESCRIBE <tablename>
- DELETE <tablename>, <rowkey>[, <columns>]
- DISABLE <tablename>
- DROP < tablename>
- ENABLE <tablename>
- EXIT
- EXISTS <tablename>
- GET <tablename>, <rowkey>[, <columns>]
- LIST
- PUT <tablename>, <rowkey>, <columnid>, <value>[, <timestamp>]
- SCAN <tablename>[, <columns>]
- STATUS [{summary|simple|detailed}]
- SHUTDOWN

Physical implementation

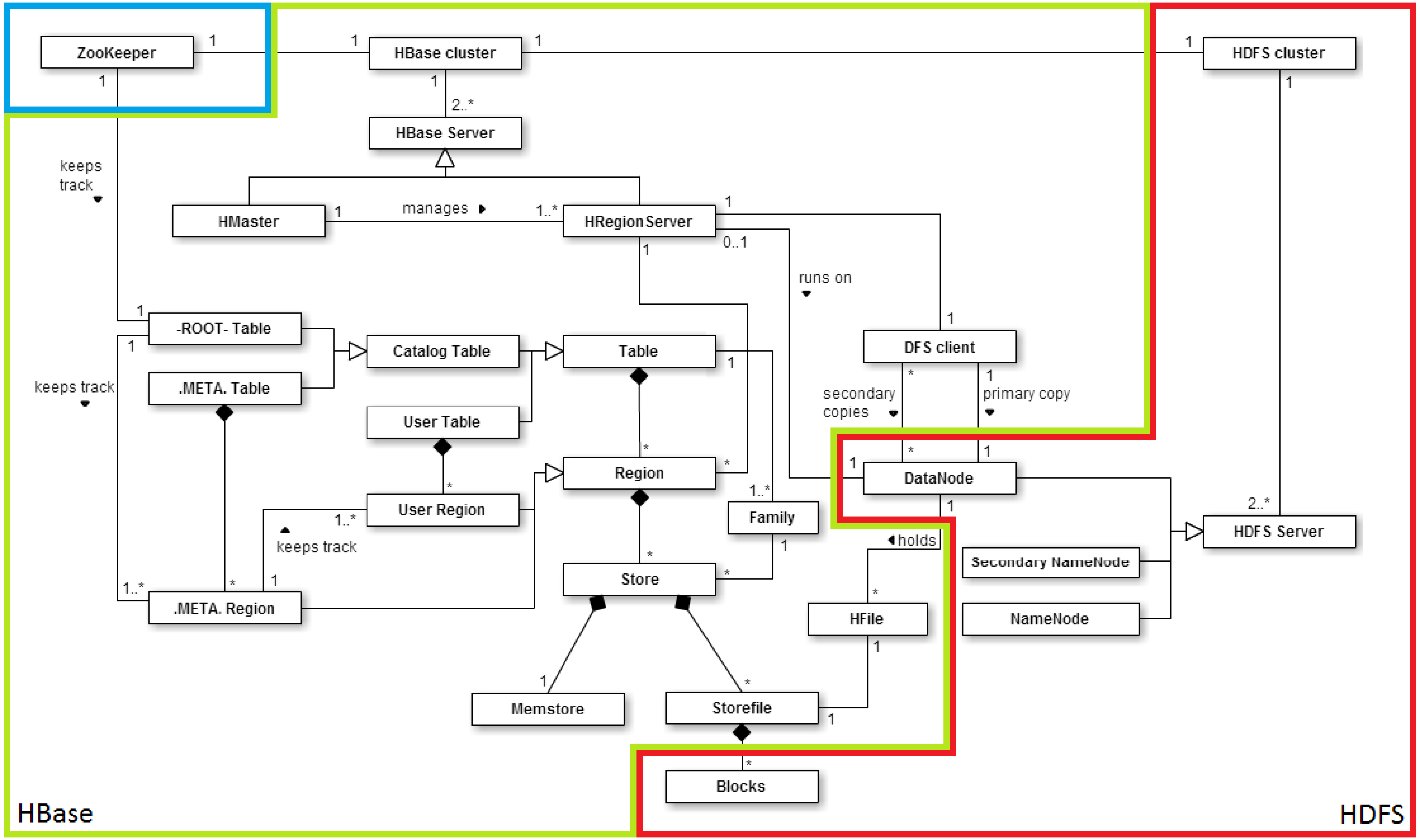


- Each table is horizontally fragmented into *tablets* (called “regions” in HBase)
  - Dynamic fragmentation
    - By default into few hundreds of Mbs
  - Distributed on a cluster of machines or cloud
- At each tablet rows are stored column-wise according to families (hybrid fragmentation)
  - Static fragmentation (the schema determines the locality of data)
    - Multiple column families can be grouped together into a locality group
      - A locality group can be “in-memory”
  - Block compression can be enabled (i.e., column families are compressed together)
- Metadata table (~ catalog)
  - Tuples are lexicographically sorted according to the key
    - Each row (entry) consists of <key, loc>
      - Key: it is the last key value in *that* tablet
      - Loc: it is the physical address of a tablet
  - This is a **distributed index cluster** (B-tree) on top of HDFS
    - It is divided into tablets and chunks
    - Supports single row transactions

Functional components of HBase (I)

- Zookeeper
  - Quorum of servers that stores HBase system config info
- Hmaster
  - Coordinates splitting of regions/rows across nodes
  - Controls distribution of HFile chunks
- Region Servers (HRegionServer)
  - Services HBase client requests
    - Manage stores containing all column families of the region
  - Logs changes
  - Guarantees "atomic" updates to one column family
  - Holds (caches) chunks of Hfile into Memstores, waiting to be written
- HFiles
  - Consist of large (e.g., 64MB) chunks
    - 3 copies of one chunk for availability (default)
- HDFS
  - Stores all data including columns and logs
    - NameNode holds all metadata including namespace
    - DataNodes store chunks of a file
  - HBase uses two HDFS file types
    - HFile: regular data files (holds column data)
    - Hlog: region's log file (allows flush/fsync for small append-style writes)
- Clients
  - Read and write chunks
    - Locality & load determine which copy to access

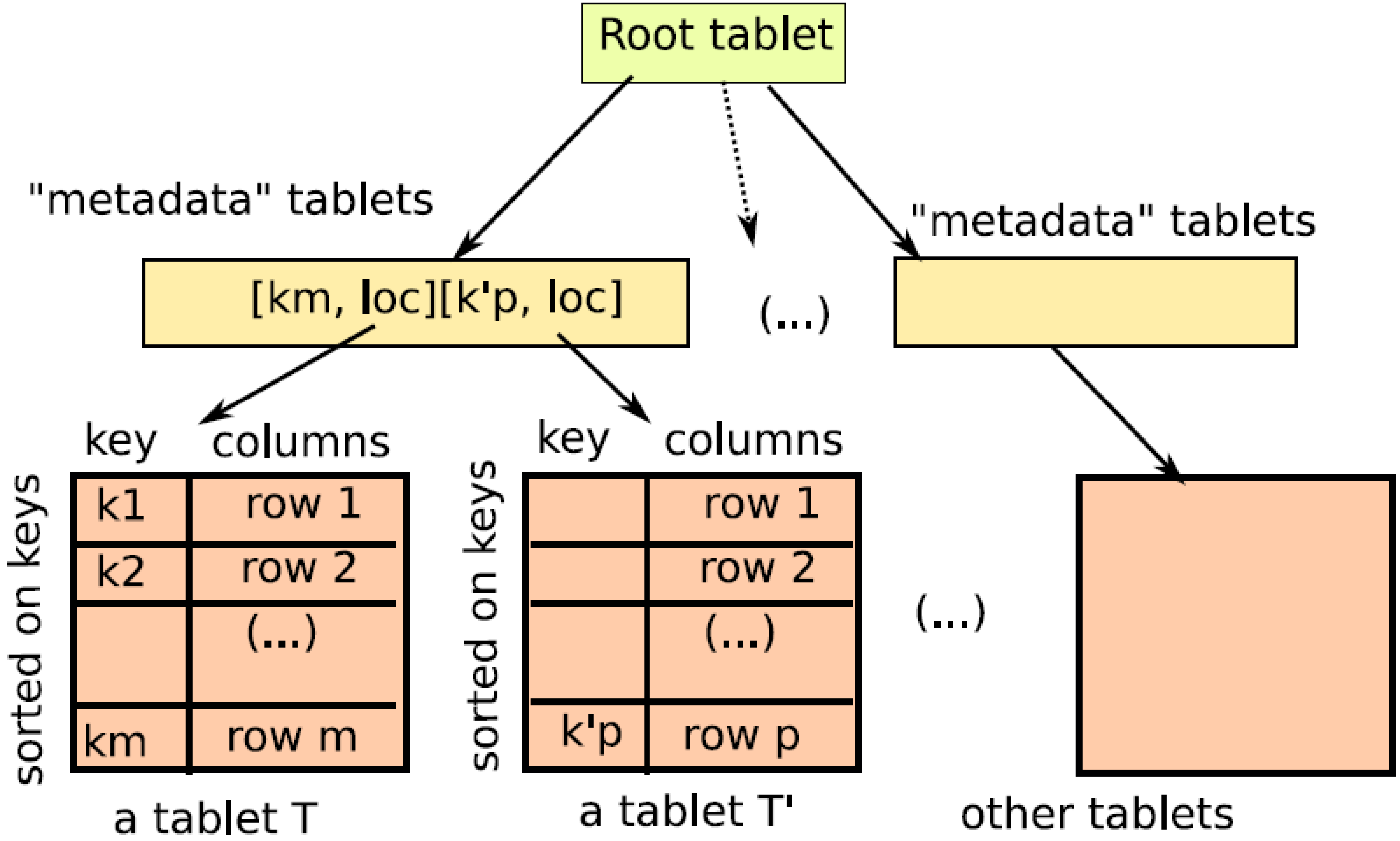
Functional components of HBase (II)



- A primary copy must be stored in the same DataNode the HRegionServer runs on.
- Secondary copies can be stored in any DataNode different from the DataNode the HRegionServer runs on.
- All the stores of a given family correspond to the same table as this family.

B —n—m— A An element of class B is associated with n elements of A, and viceversa
B ——▷ A Class B is a specialization (subtype) of class A
B ——◆ A Class A is composed by elements of class B

A Distributed Index Cluster

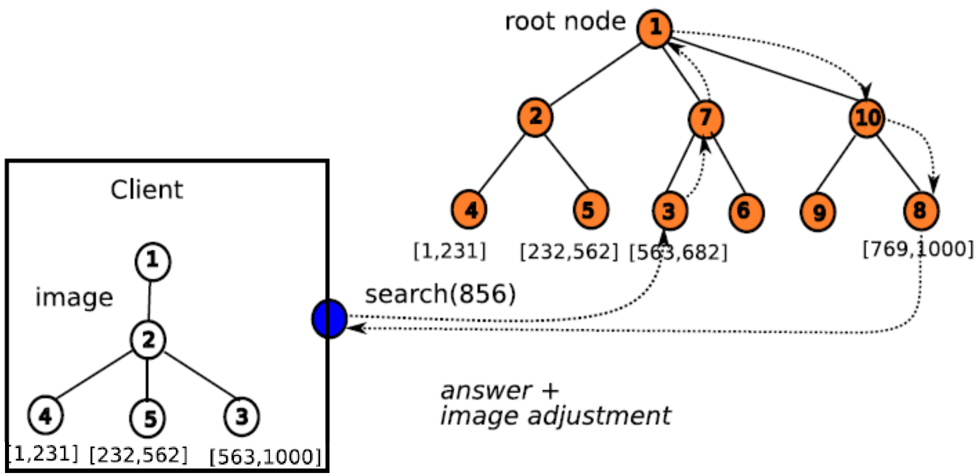


S. Abiteboul et al.

HBase Design Decisions (I)

- One master server
  - Maintenance of the table schemas
    - Root tablet
  - Monitoring of services (heartbeating)
  - Assignment of tablets to servers
- Many tablet servers
  - Each handling around 100-1.000 tablets
  - Apply concurrency and recovery techniques
  - Managing split of tablets
    - A tablet server decides to split
    - Half of its tablets are sent to another server
  - Managing merge of tablets
- Client nodes

HBase Design Decisions (II)



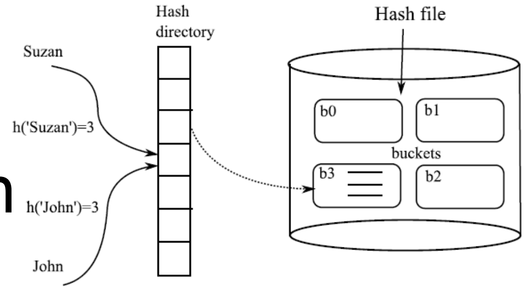
S. Abiteboul et al.

- Mistake compensation
  - The client keeps in cache the tree sent by the master and uses it to access data
  - If an out-of-range error is triggered, it is forwarded to the root
    - In the worst case, 6 network round trips

Distributed Hashing (alternative to a tree)

- Hash do neither support range queries nor nearest neighbours search
- Distributed hashing challenges
  - Dynamicity: Typical hash function

    f(x) = x % #servers
    - Adding a new server implies modifying hash function
      - Massive data transfer
      - Communicating the new function to all servers
  - Location of the hash directory: any access must go through the hash directory



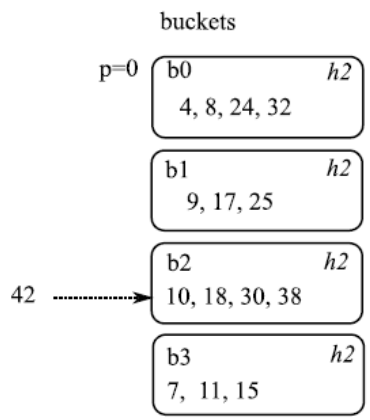
S. Abiteboul et al.

Distributed Hashing: Examples

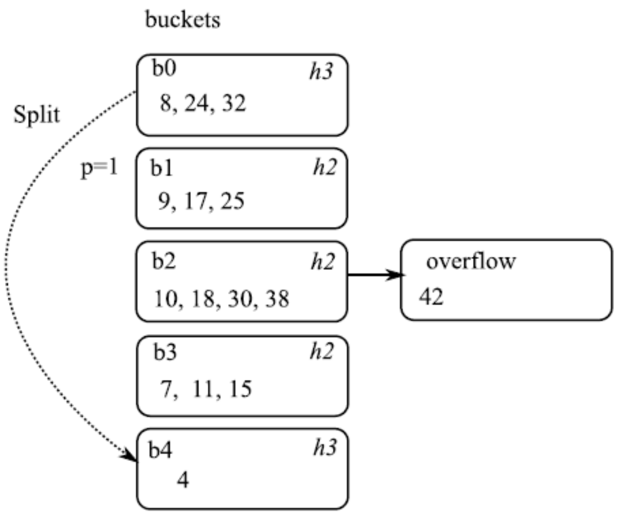
- Most current key-value (and document-stores) use distributed hashing
  - LH*
    - Memcached
    - MongoDB (past releases)
  - Consistent Hashing
    - Memcached / CouchDB
    - MongoDB (current release)
    - Cassandra
    - Dynamo / SimpleDB
    - Voldemort

Distributed Linear Hashing (LH*)

- Maintains an efficient hash in front of <u>dynamicity</u>
  - A split pointer is kept (next bucket to split)
  - A pair of hash functions are considered
    - $\%2^n$ and $\%2^{n+1}$ (being $2^n \leq \#servers < 2^{n+1}$)
  - Overflow buckets are considered
    - When a bucket overflows the bucket pointed by the split pointer splits (not the overflown one)



Bucket b2 receives a new object

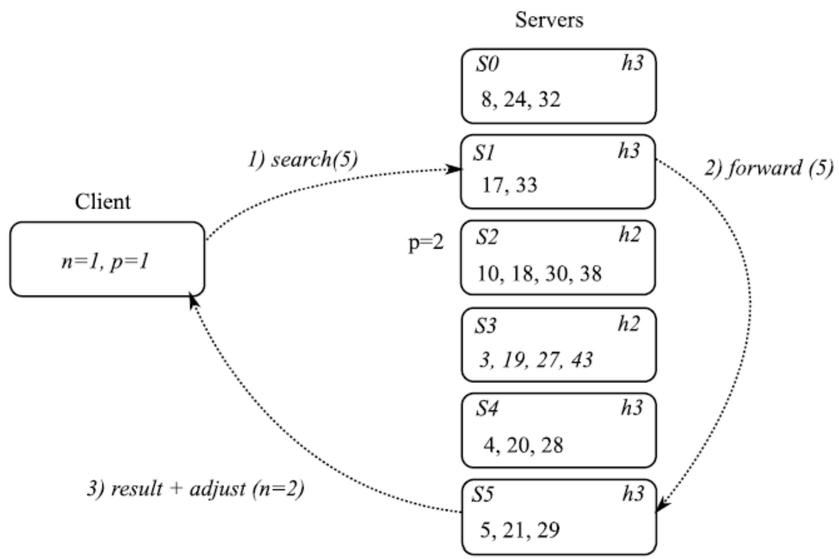


Bucket b0 splits; bucket b2 is linked to a new one

S. Abiteboul et al.

Updating the Hash Directory in LH*

- Traditionally, each participant has a copy of the hash directory
  - Changes in the hash directory (either hash functions or splits) imply *gossiping*
    - Including clients nodes
    - It might be acceptable if not too dynamic
- Alternatively, they may contain a partial representation and assume lazy adjustment
  - Apply forwarding path

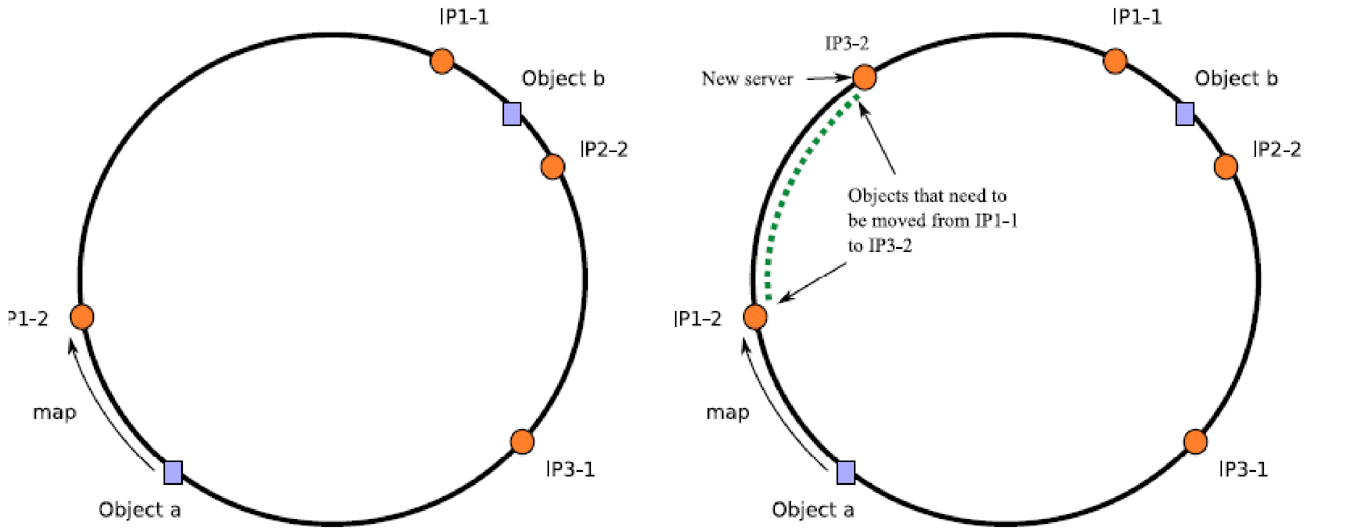


S. Abiteboul et al.

Consistent Hashing

- The hash function **never** changes
  - Choose a very large domain *D* and map server IP addresses and object keys to such domain
  - Organize *D* as a ring in clockwise order so each node has a successor
  - Objects are assigned as follows:
    - For an object O, f(O) = D_o
    - Let $D_{o'}$ and $D_{o''}$ be the two nodes in the ring such that
      - $D_{o'} < D_o <= D_{o''}$
    - O is assigned to $D_{o''}$
- Further refinements:
  - Assign to the same server several hash values (virtual servers) to balance load
  - Same considerations for the hash directory as for LH*

Adding new server in Consistent Hashing



Mapping of objects to servers   Server IP3-2 is added, with local re-hashing

S. Abiteboul et al.

- Adding a new server is straightforward
  - It is placed in the ring and part of its successors' objects are transferred to it

Activity

- *Objective: Understand the three distributed directories*
- *Tasks:*
  1. *(5') Individually solve one exercise*
  2. *(10') Explain the solution to the others*
  3. *Hand in the three solutions*

- *Roles for the team-mates during task 2:*
  a) *Explains his/her material*
  b) *Asks for clarification of blur concepts*
  c) *Mediates and **controls time***

Summary

- HDFS components
- HBase components
- Data distribution structures
  - B-Tree
  - Linear hash
  - Consistent hash

Bibliography

- S. Ghemawat et al. *The Google File System*. OSDI'03
- F. Chang et all. *Bigtable: A Distributed Storage System for Structured Data*. OSDI'06
- M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*, 3rd Ed. Springer, 2011
- P. Sadagale and M. Fowler. NoSQL distilled. Addison-Wesley, 2013
- E. Meijer and G. Bierman. *A Co-Relational model of data for large shared data banks*. Communications of the ACM 54(4), 2011
- S. Abiteboul et al. *Web data management*. Cambridge University Press, 2011
- W. Vogels. *Eventually consistent*. ACM QUEUE, October 2008

Resources

- http://hadoop.apache.org
- http://hbase.apache.org
- http://www.oracle.com/technetwork/products/nosqldb/index.html