Key-Value Stores

Big Data Management

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

DTIM
www.essi.upc.edu/dtim

Knowledge objectives

1. Give the definition of the BigTable data model
2. Explain what a map structure is
3. Explain the difference between a KeyValue and a Wide-Column store
4. Enumerate the main schema elements of Hbase
5. Explain the main operations available in HBase
6. Enumerate the main functional components of HBase
7. Explain the role of the different functional components in Hbase
8. Explain the tree structure of data in HBase
9. Explain the 3 basic algorithms of Hbase
10. Explain the main components and behaviour of an LSM-tree
11. Compare a distributed tree against a hash structure of data
12. Justify the need of dynamic hashing
13. Explain the structure of HBase catalog
14. Explain the mistake compensation mechanism of the cache in HBase client
15. Enumerate the ACID guarantees provided by HBase
16. Explain the execution flow of an HBase query both at global and local levels

Understanding objectives

1. Given few queries, define the best logical structure of a table considering its physical implications in terms of performance
2. Given the data in two leafs of a Log-Structured Merge-tree, merge them
3. Given the current structure of a Linear Hash, modify it according to insertions potentially adding buckets
4. Given the current structure of a Consistent Hash, modify it in case of adding a bucket
5. Calculate the number of round trips needed in case of mistake compensation of the tree metadata

Application objectives

1. Use HBase shell to create a table and access it
2. Use HBase API to create a table and access it

Key-Value

BigTable

BigTable Data Model

Data should be stored in disk

"A Bigtable is a sparse, distributed, persistent, multi-dimensional, sorted map."

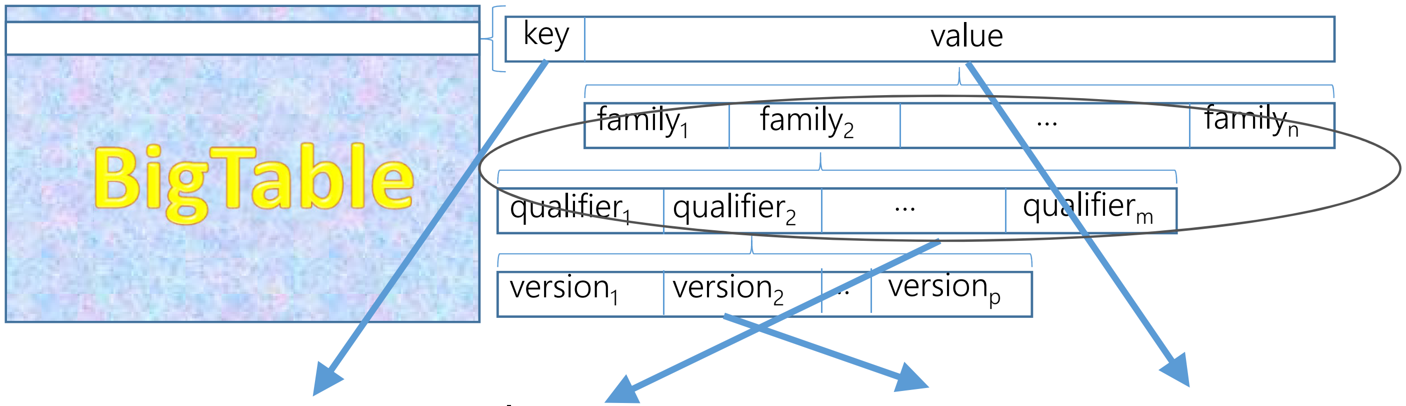
F. Chang et al.

- Sparse: most elements are unknown
- Distributed: cluster parallelism
- Persistent: disk storage (HDFS)
- Multi-dimensional: values with columns
- Sorted: sorting lexicographically by primary key
- Map: lookup by primary key (a.k.a. finite map)

Families must be explicitly created by modifying the schema of the table.
One qualifier belongs to a family and it is only declared at insertion time.
Qualifiers are not fixed. Families are fixed.

BigTable 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

We can limit the number of versions we can keep. After that limit is reached, older version data is deleted.

Key-Value

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

key		value
Bob	⇨	Michael_Elisabeth_30_Bobby_2010

- Bigtable (Wide-column key-value stores)
  - Entries in form of key-values
    - But now values are split in wide-columns
  - Typically query on key
    - May have some support for values
  - Schemaless within a wide-column

System is aware of partitions
But not aware of schema inside partition

key		Wide-columns and qualifiers		
Bob	⇨	mother:Elisabeth father:Michael	connections:30	name:Bobby birth_year:2010

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

HBase Architecture

HBase shell

Unique from Relational - GET, PUT

- ALTER <tablename>, <columnfamilyparam>
- COUNT <tablename>
- CREATE TABLE <tablename>
- DESCRIBE <tablename> Returns families of the table
- DELETE <tablename>, <rowkey>[, <columns>]
- DISABLE <tablename> Since, we are in highly distributed env, To avoid interference with other user, we need to disable table before alter or drop
- DROP < tablename>
- ENABLE <tablename> Not needed to disable for PUT and GET
- EXIT Leave the terminal
- EXISTS <tablename> Check if given table exists in the system
- GET <tablename>, <rowkey>[, <columns>]
- LIST List all tables in the system
- PUT <tablename>, <rowkey>, <columnid>, <value>[, <timestamp>]
- SCAN <tablename>[, <columns>] Scan entire table
- STATUS [{<u>summary</u>|simple|detailed}]
- SHUTDOWN

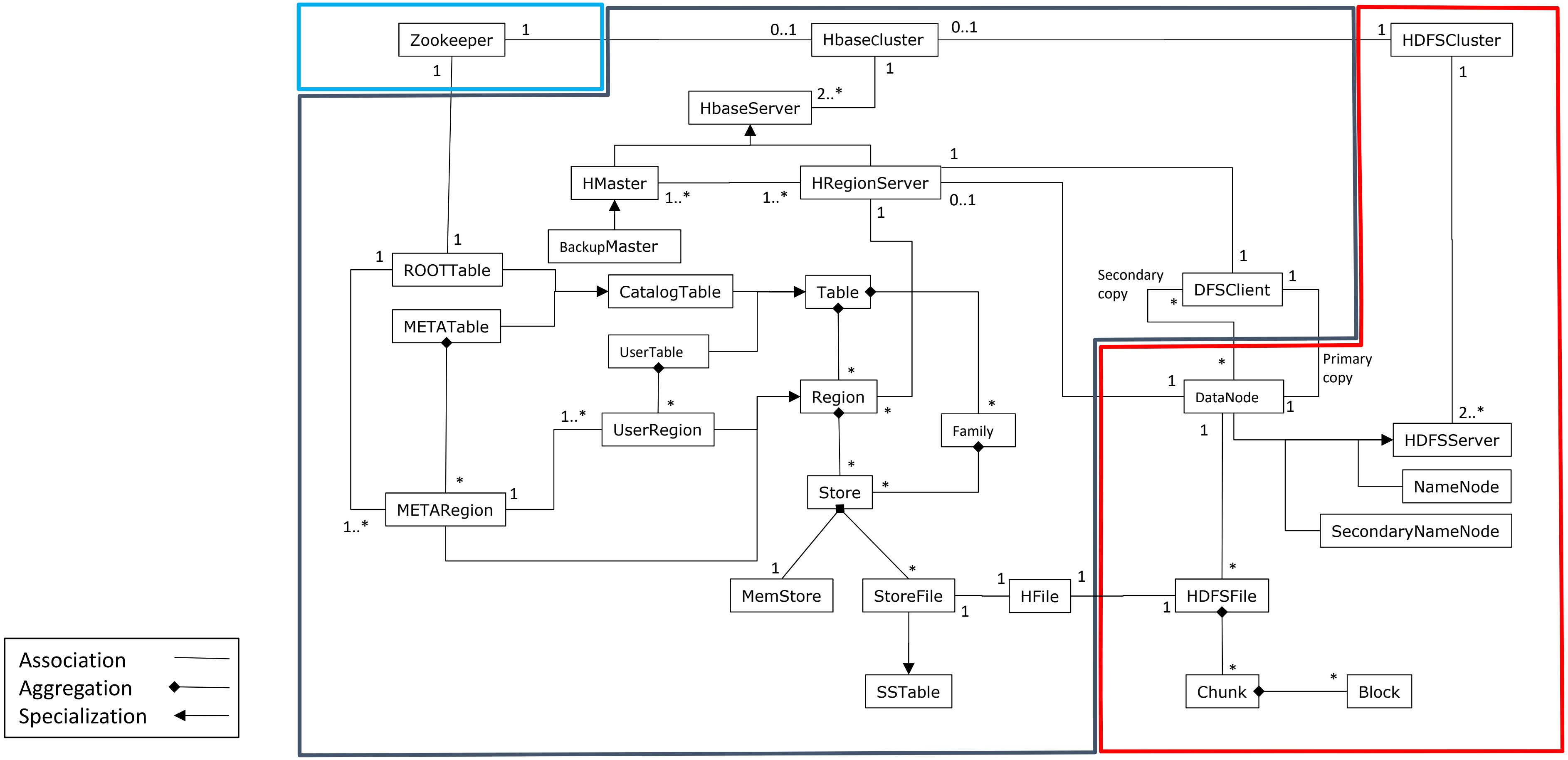
https://learnhbase.wordpress.com/2013/03/02/hbase-shell-commands

Functional components (I) Region -Distribution unit of HBase

- Zookeeper - Quorum of servers that stores HBase system config info
- HMaster - Coordinates splitting of regions/rows across nodes
  - Controls distribution of HFile chunks
- HRegionServer - Region Servers
  - Serves HBase client requests
    - Manage stores containing all column families of the region
  - Logs changes
  - Guarantees "atomic" updates to one column family
- HFiles - Consist of large (e.g., 128MB) chunks
- 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 formats
    - HFile: regular data files (holds column data)
      - 3 copies of one chunk for availability (default)
    - HLog: region's log file (allows flush/fsync for small append-style writes)

Functional components (II)

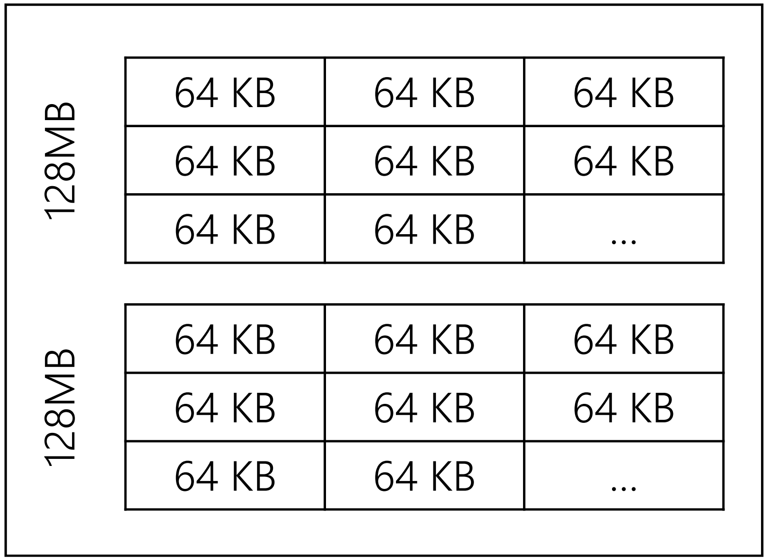
Vertically, there are multiple families.
Horizontally, there are multiple regions.
In the intersection of families and regions, there are stores.



StoreFiles

- When the MemStore is full (128MB), data are flushed to HDFS
- A StoreFile is generated
  - Format HFile
- An HFile stores data into HDFS chunks
  - Chunks are structured into HBase blocks
    - Size 64 KB

Storefile
(HFile format)



HBase processes

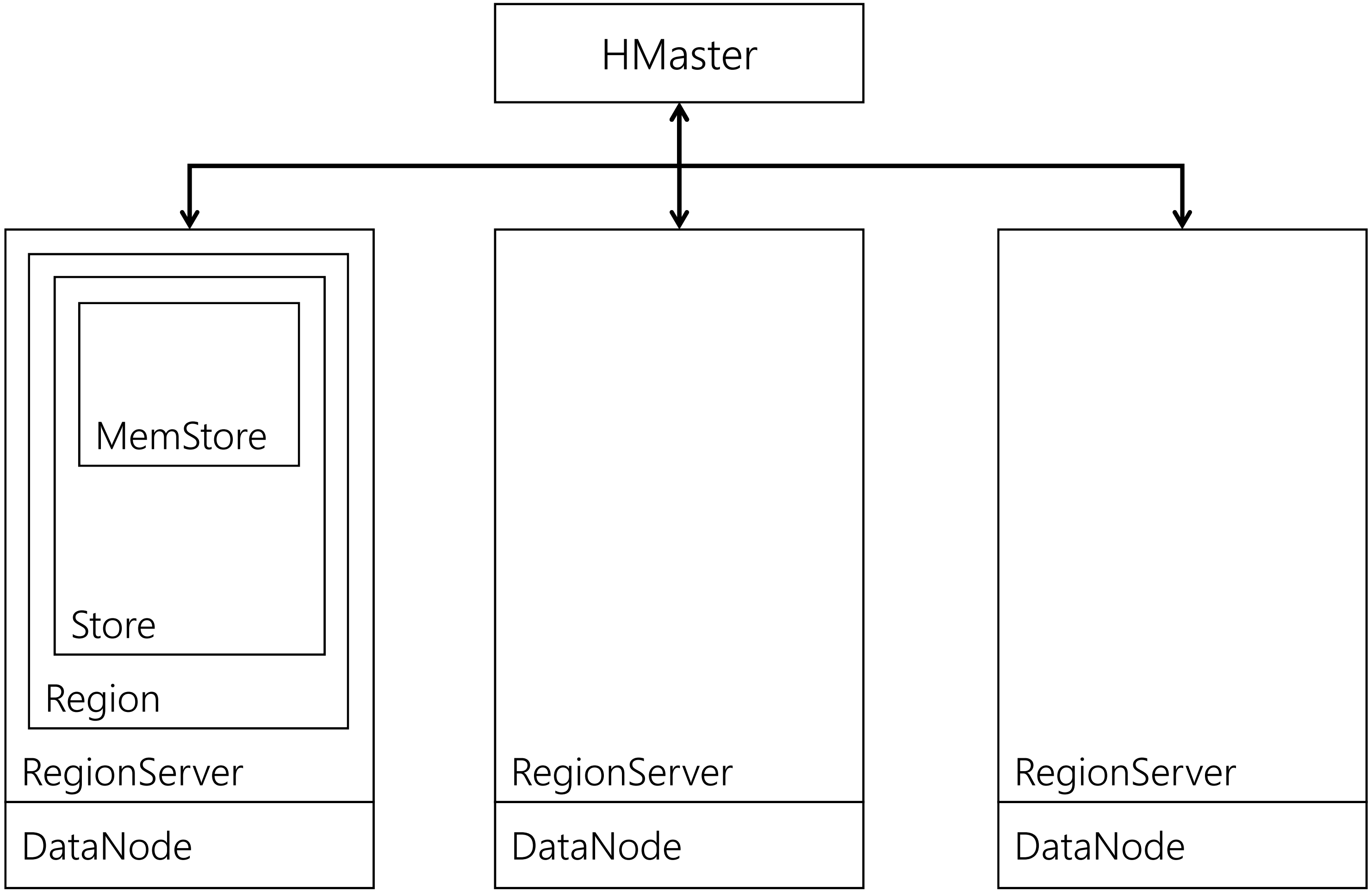
a) Flush
   - On memory structure reaching threshold
   - Takes memory content and store it into an SSTable
   - Generates different disk versions of the same record

b) Minor compactation
   - Runs regularly in the background
   - Merges a given number of equal size SSTables into one
   - Does not remove all record versions (only some)

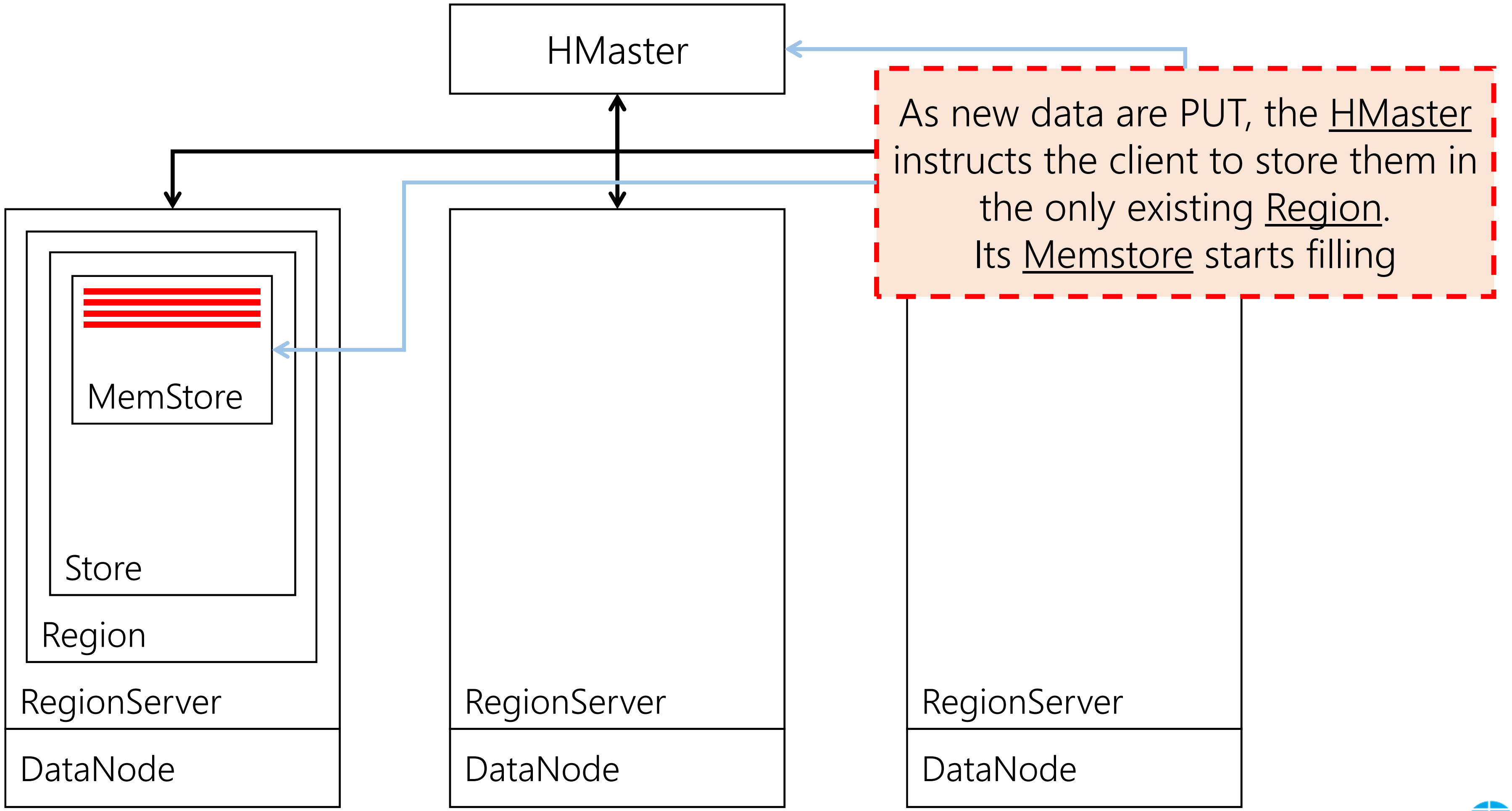
c) Major compactation
   - Triggered manually
   - Merges all SSTables
   - Leaves one single SSTable
   - All versions of a record are merged into one

Can be slow

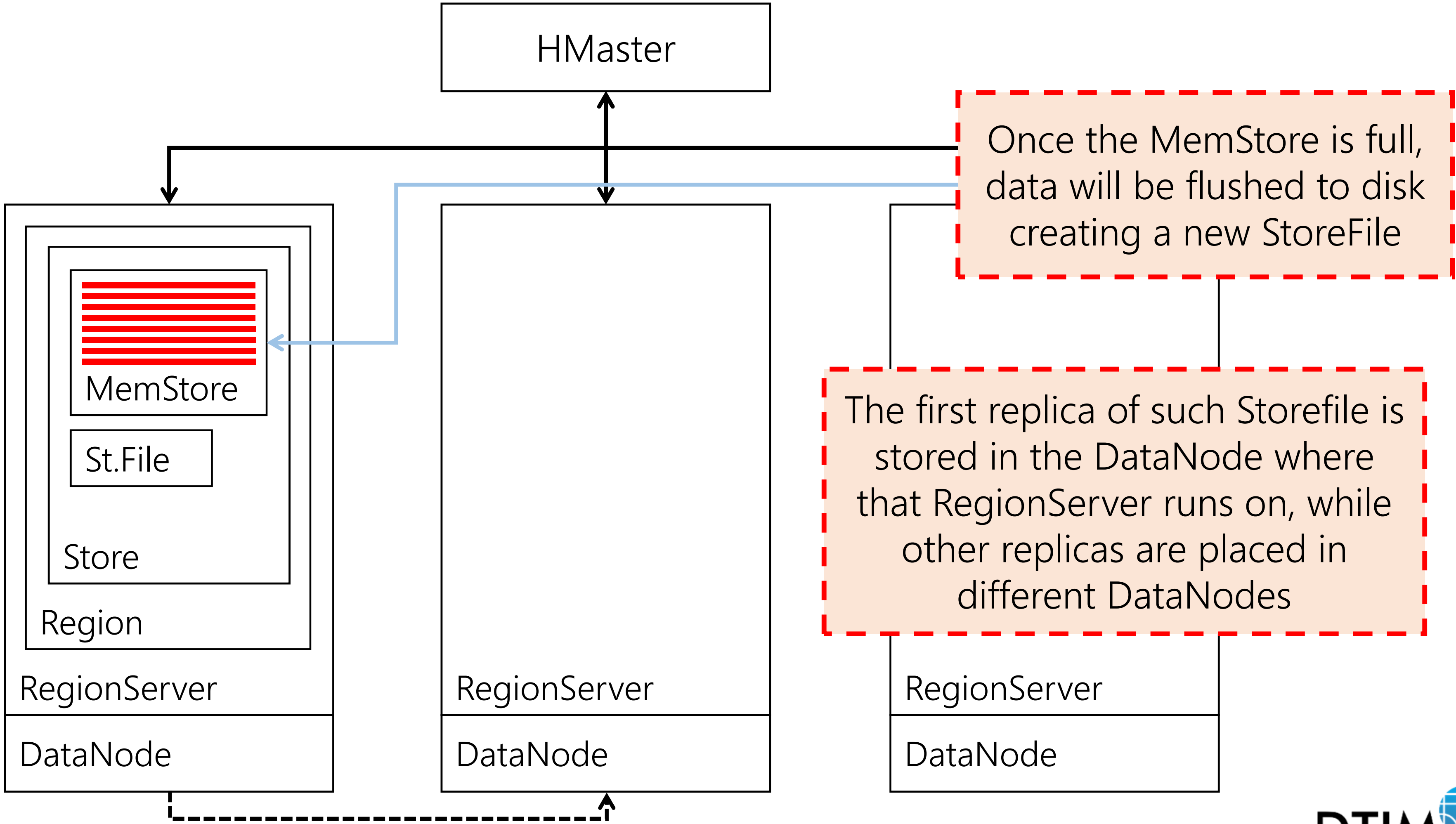
Example of Flush



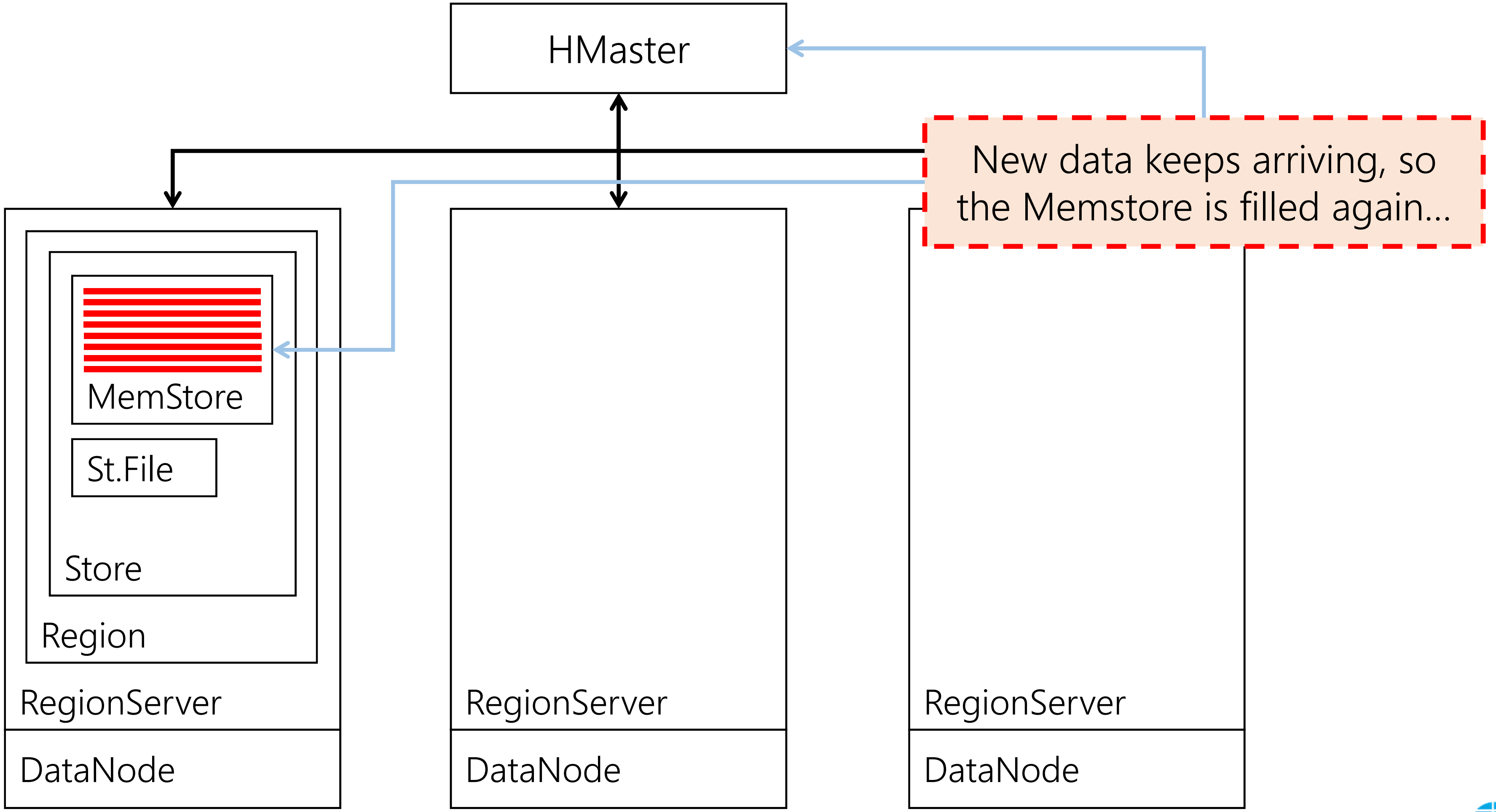
Example of Flush



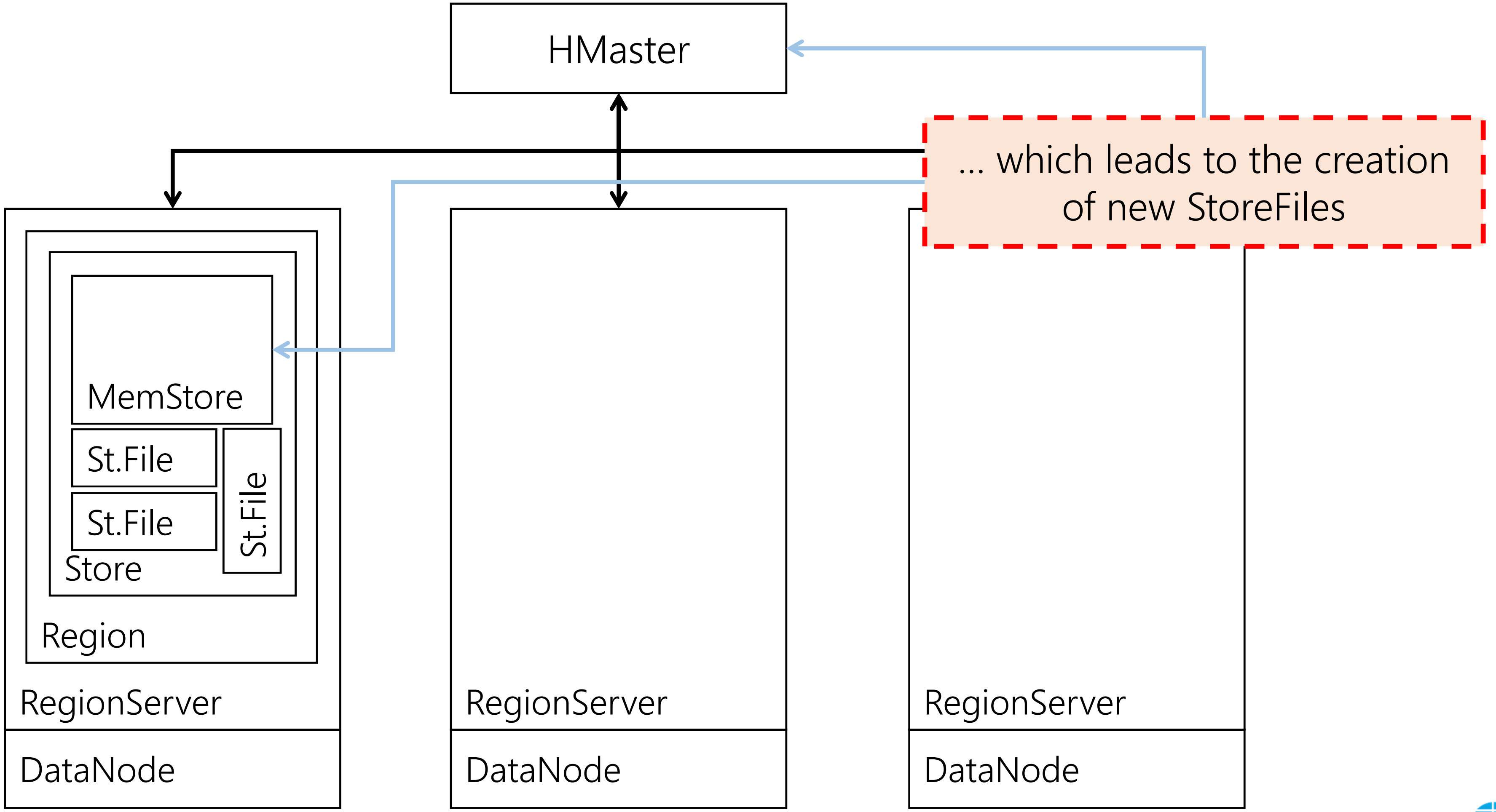
Example of Flush



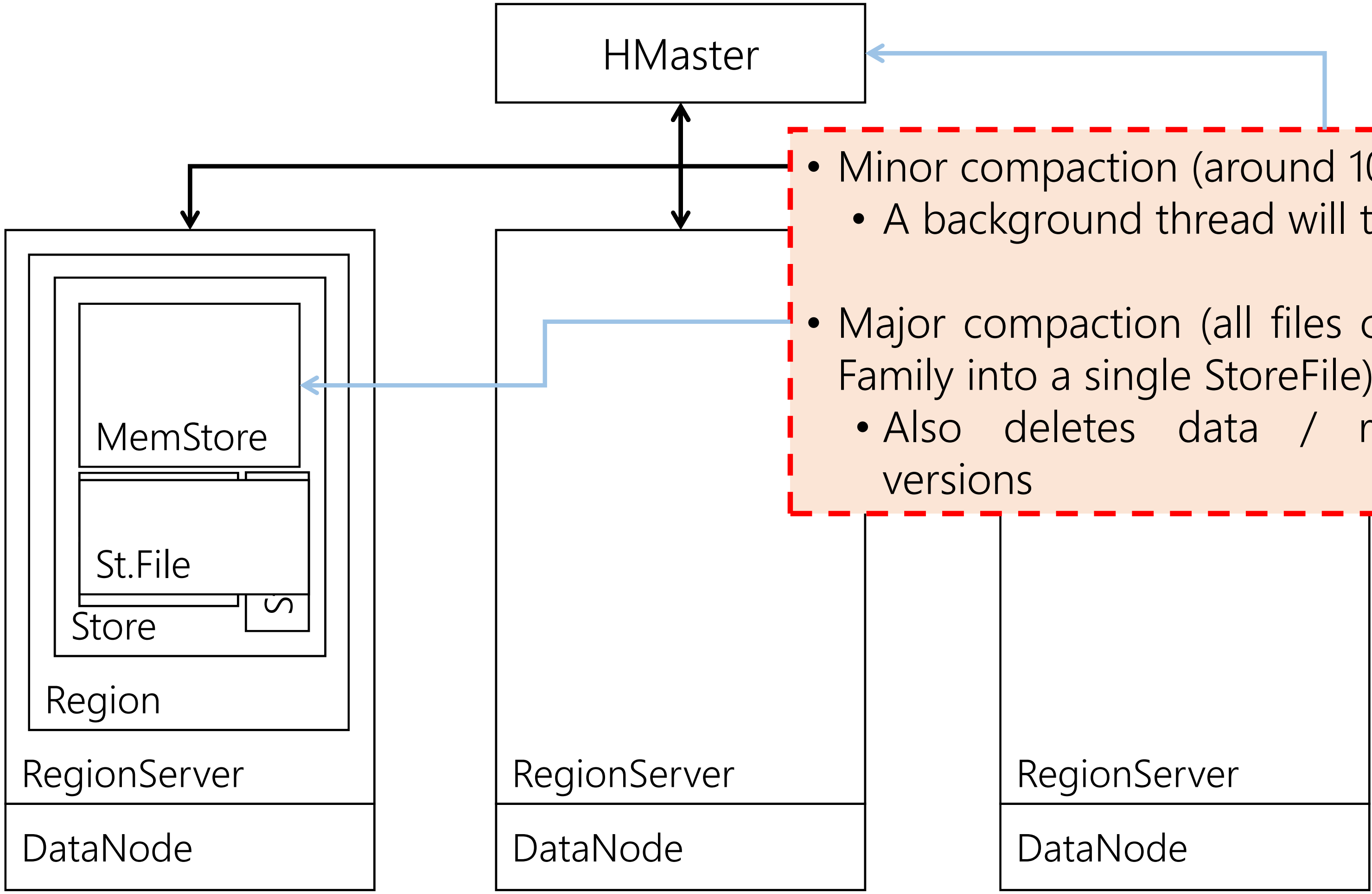
Example of Compactation



Example of Compactation

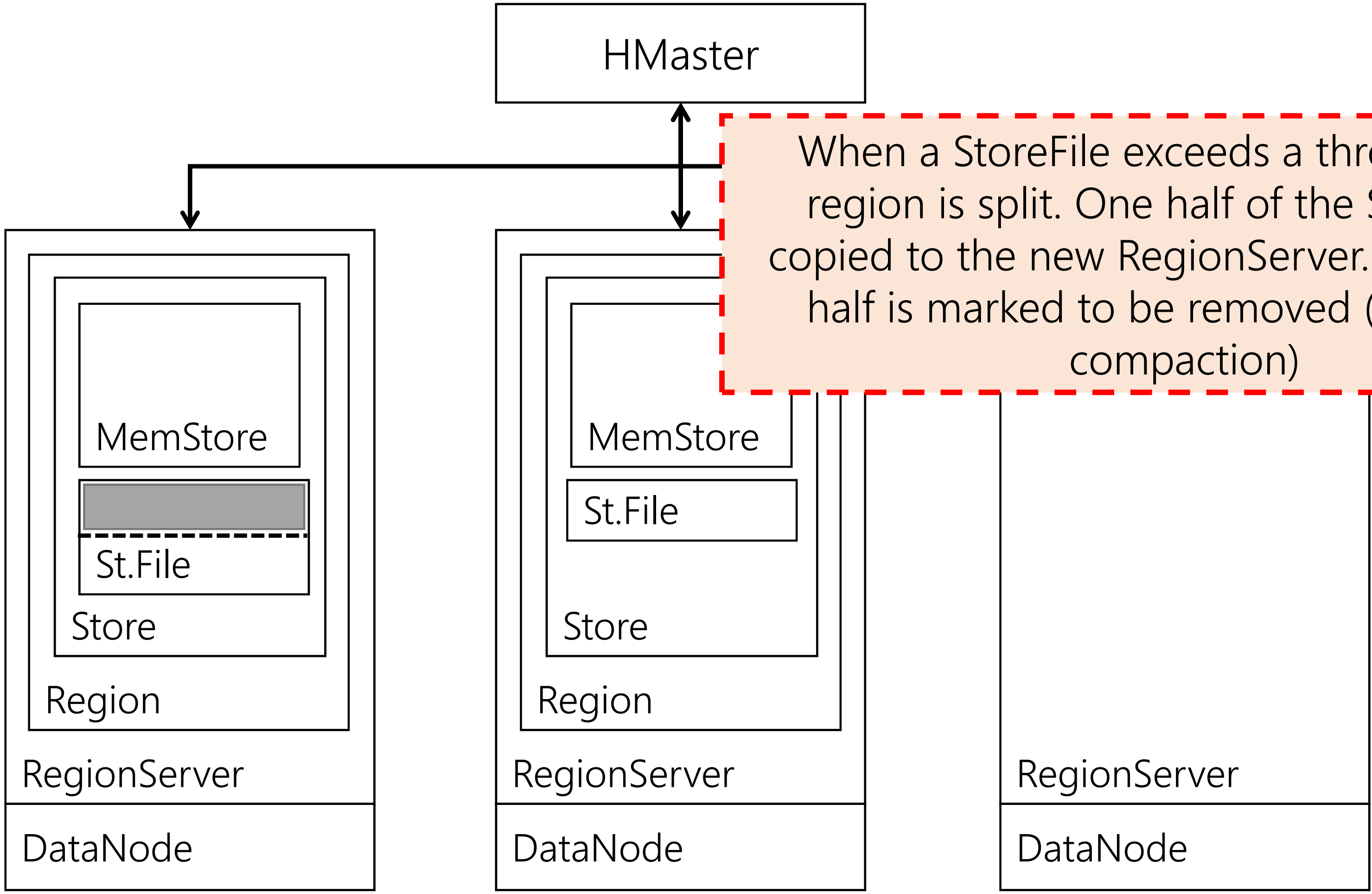


Example of Compactation



- Minor compaction (around 10 files)
  - A background thread will trigger it
- Major compaction (all files of a Column-Family into a single StoreFile)
  - Also deletes data / removes old versions

Example of Split



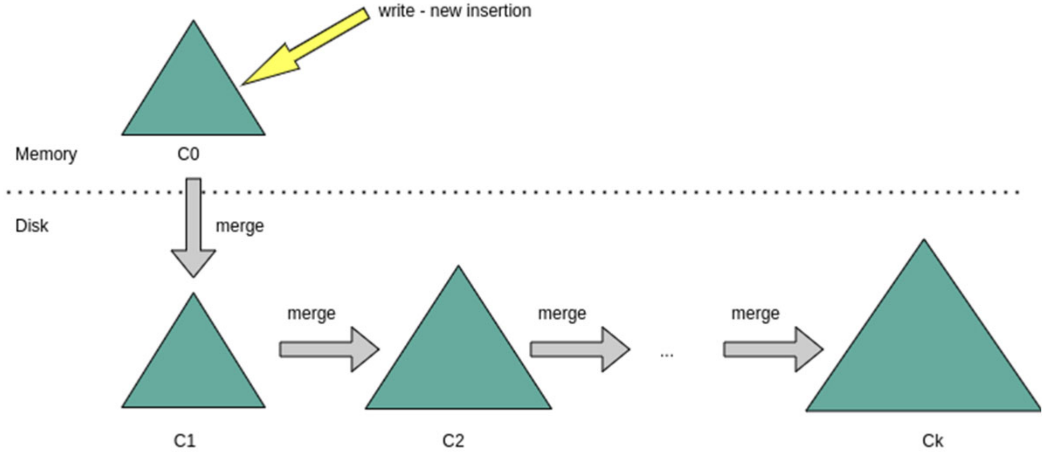
When a StoreFile exceeds a threshold, the region is split. One half of the StoreFile is copied to the new RegionServer. The original half is marked to be removed (in a major compaction)

Underlying Tree Structures

LSM-tree

Log Structured Merge-trees

- Defers and batches index changes
- Consists on two structures
  - In Memory
    - Not necessarily a B-Tree
    - Node different from disk blocks
  - In Disk
    - Multiple components
      - Blocks 100% full
- Regularly merges trees in one level into the next one

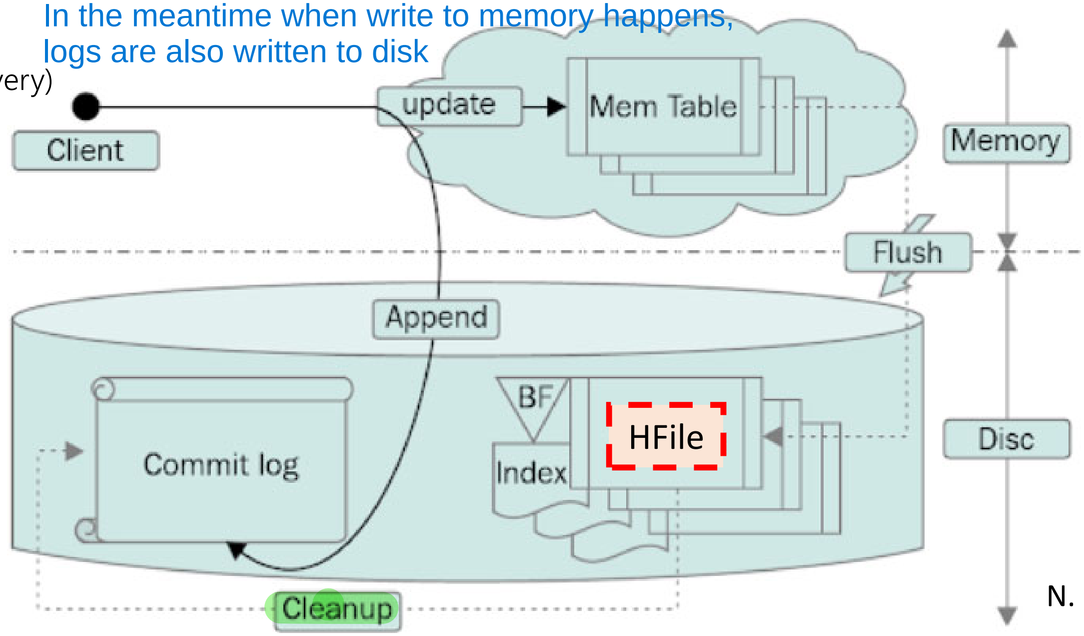


Aina Montalban, based on N. Neeraj

HBase LSM-tree implementation

- In Memory (MemStore)
  - Holds the most recent updates for fast lookup
  - Sorted by key
- In Disk (StoreFiles) immutable
  - Immutable Sorted-String Tables (with Bloom Filters and Indexes)
    - May contain different versions of the same row
      - All of them need to be accessed at query time
    - Regularly performs a size-tiered merge process
      - Old SSTables not overwritten (available for recovery)

In the meantime when write to memory happens, logs are also written to disk



Happens after data is written to disc

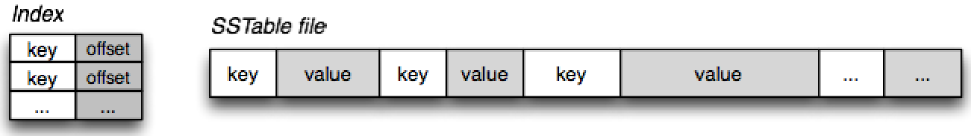
N. Neeraj

LSM-tree maintenance algorithms

When memory structure reaches threshold, data is flushed into SSTable.
Here data is stored in the form of key value and index is maintained to store offset (position) of the value.

On memory structure reaching threshold:

1. Take next <u>in memory leafs</u>
2. Flush them to an SSTable (of the stablished size)



On triggering a compactation

1. Take *n* SStables and merge them
2. Put the merge in an <u>in-memory buffer</u>
3. If buffer size exceeds chunk size
   1. Write one chunk to disk
   2. Purge buffer
   3. Keep exceeds in the buffer

Underlying Hash Structures

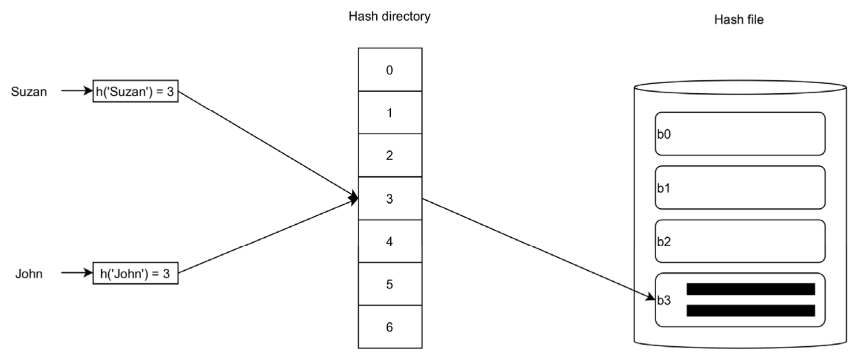
Linear hash

Consistent hash

Distributed Hashing (alternative to a tree)

Idea
To have a function so that Given the key, where should the value of this key be stored?
Two similar keys must be stored in very different machines

- Hash does neither support range queries nor nearest neighbours search
- Distributed hashing challenges
  - Dynamicity:
    - Typical hash function
      h(x) = f(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



Aina Montalban, based on S. Abiteboul et al.

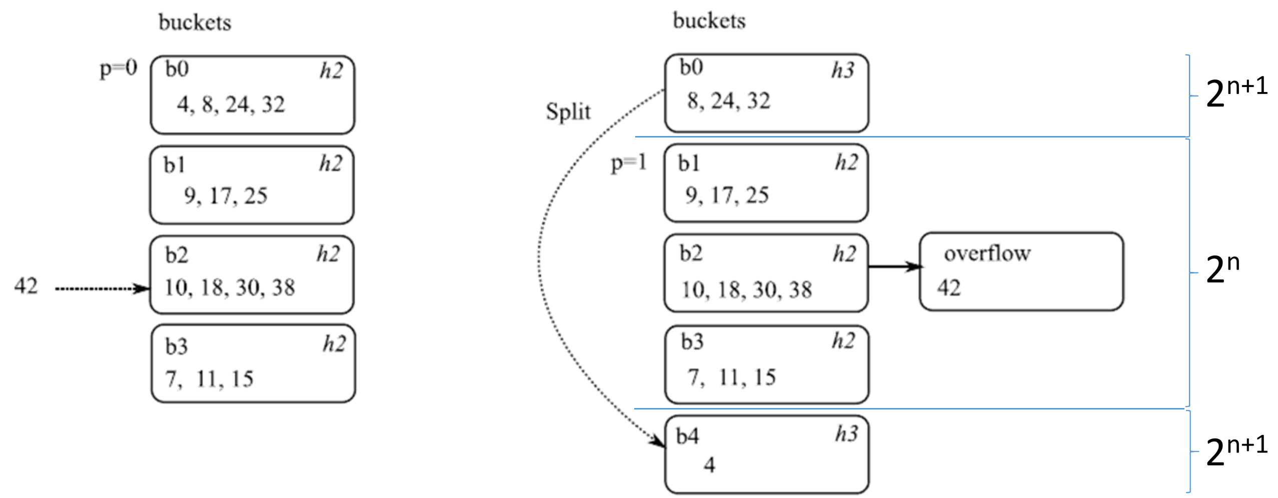
Adding a node in static hash

3 Nodes

Bucket/Node 0	Bucket/Node 1	Bucket/Node 2	Bucket/Node 3
Key 0	Key 1	Key 2	
Key 3	Key 4	Key 5	
Key 6	Key 7	Key 8	
Key 9	Key 10	Key 11	

Distributed Linear Hashing (LH*)

- Maintains an efficient hash in front of dynamicity
  - 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 pointed bucket splits (not necessarily the overflown one)



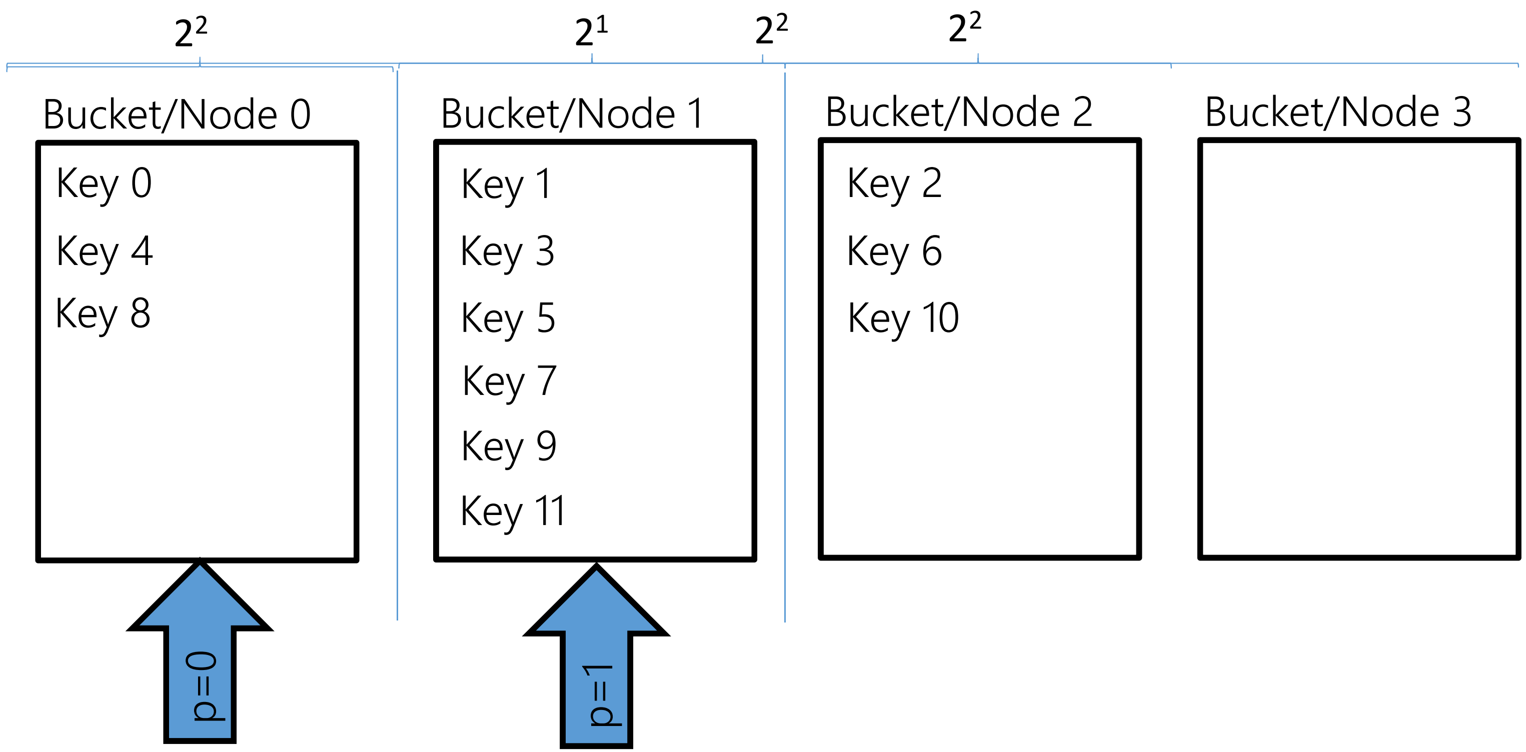
Bucket b2 receives a new object *Bucket b0 splits; bucket b2 is linked to a new one*

S. Abiteboul et al.

This actually does not solve the problem of overflow. But eventually, as more and more bucket overflows, pointer moves and overflowing bucket splits.

Adding a node in linear hash

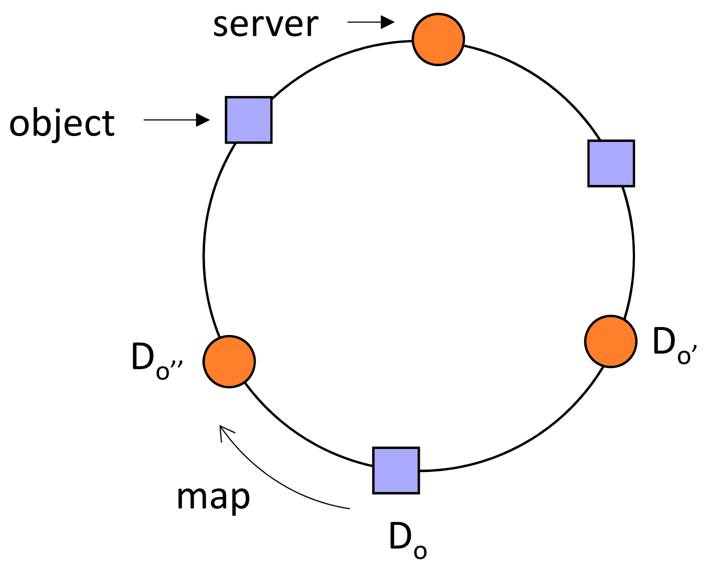
3 Nodes



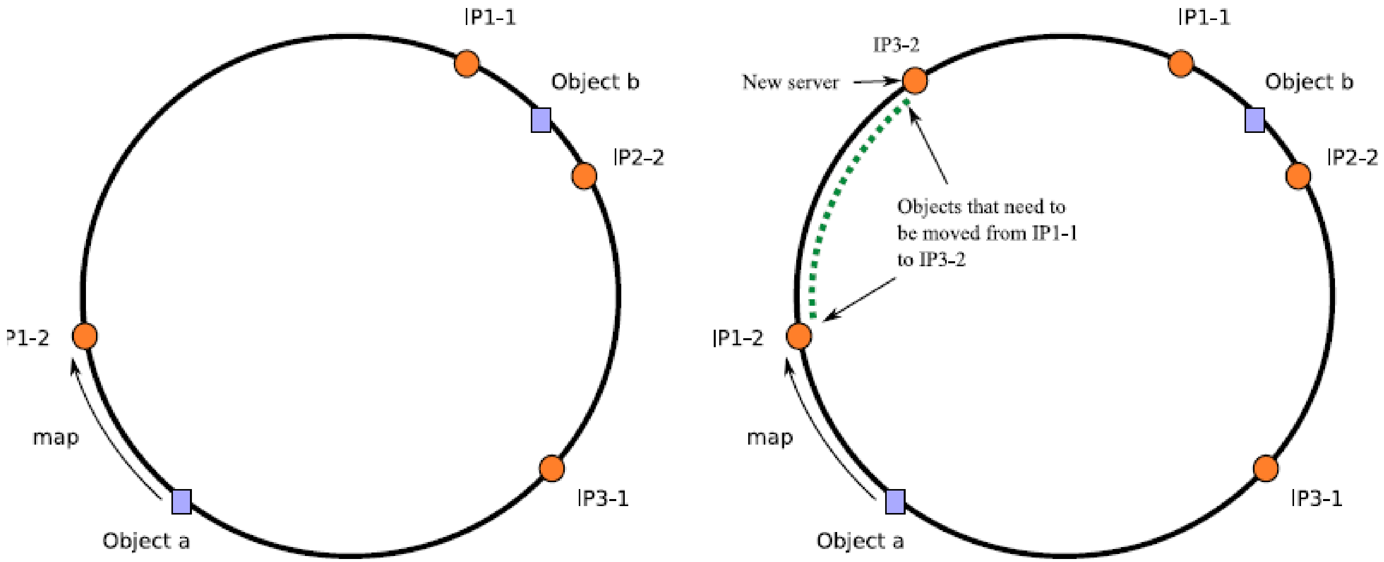
Consistent Hashing

we apply the module a number much larger than the maximum number of buckets we can ever reach

- The hash function <u>never</u> changes
  - Choose a very large domain *D*
    - Map server IP addresses and object keys to such domain
  - Organize *D* as a ring in clockwise order
    - 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 a new server in Consistent Hashing



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

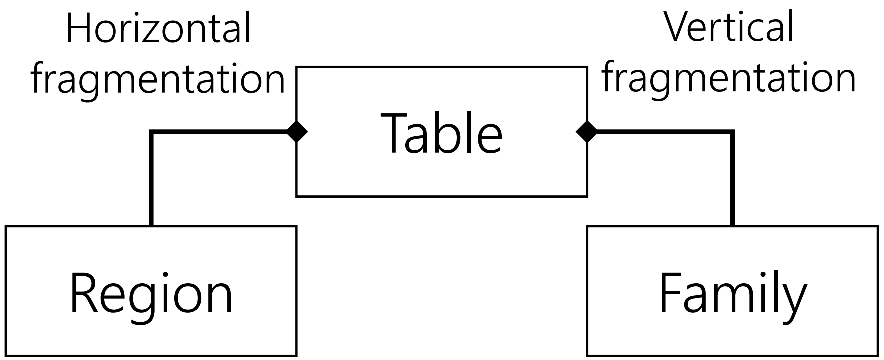
S. Abiteboul et al.

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

Data Design

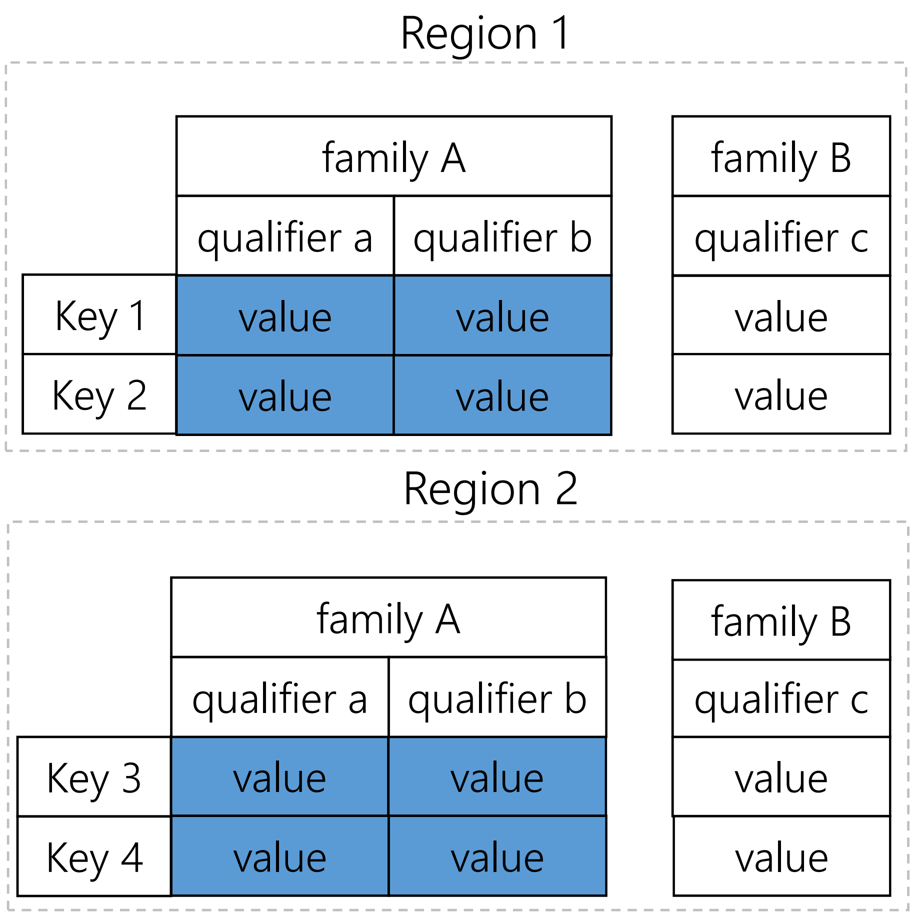
Challenge I

Logical structure

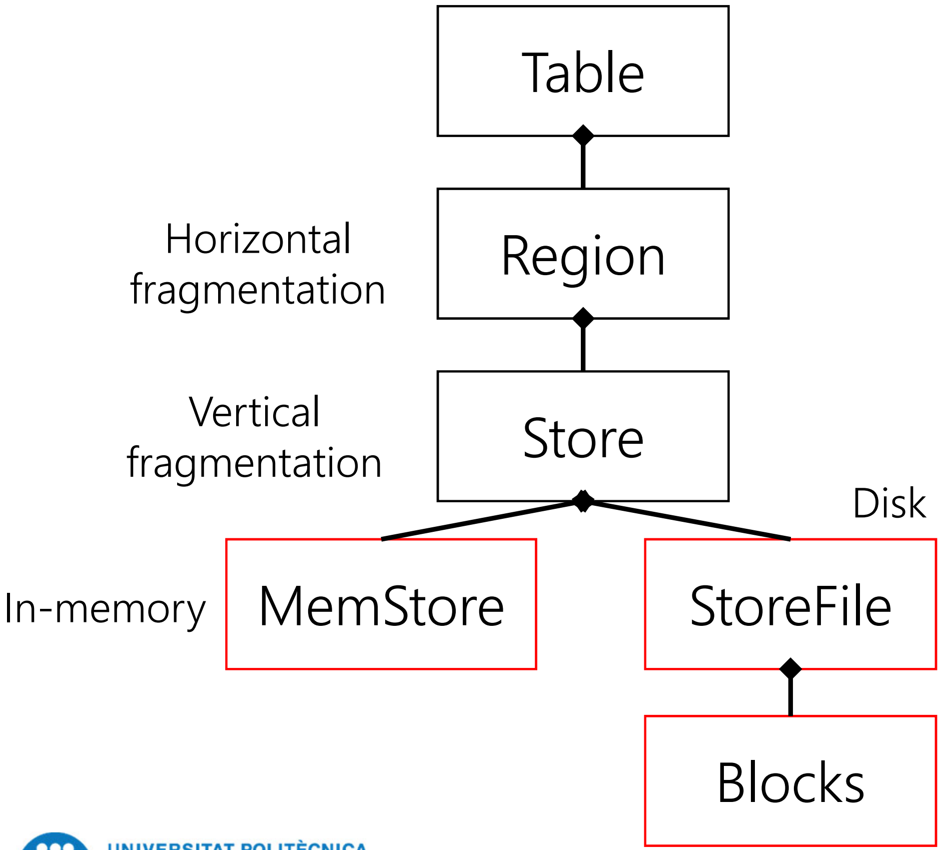


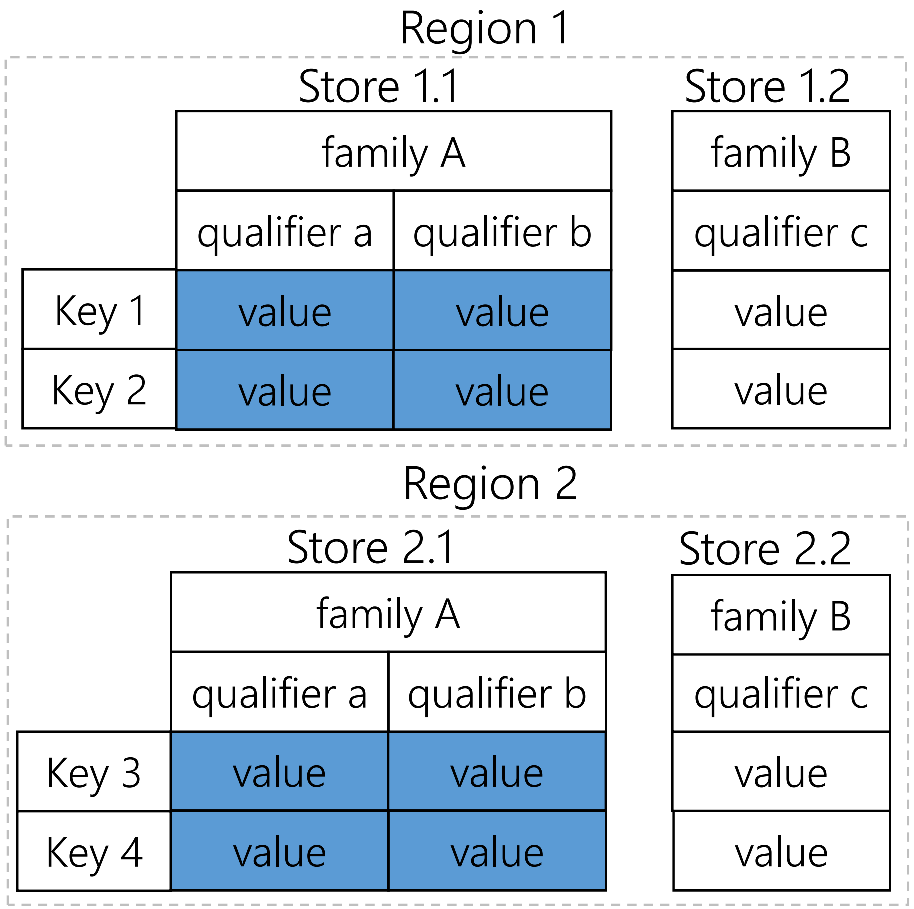
Distribution Unit
Different region potentially goes to different server.

Set of attributes that are typically accessed together



Physical structure

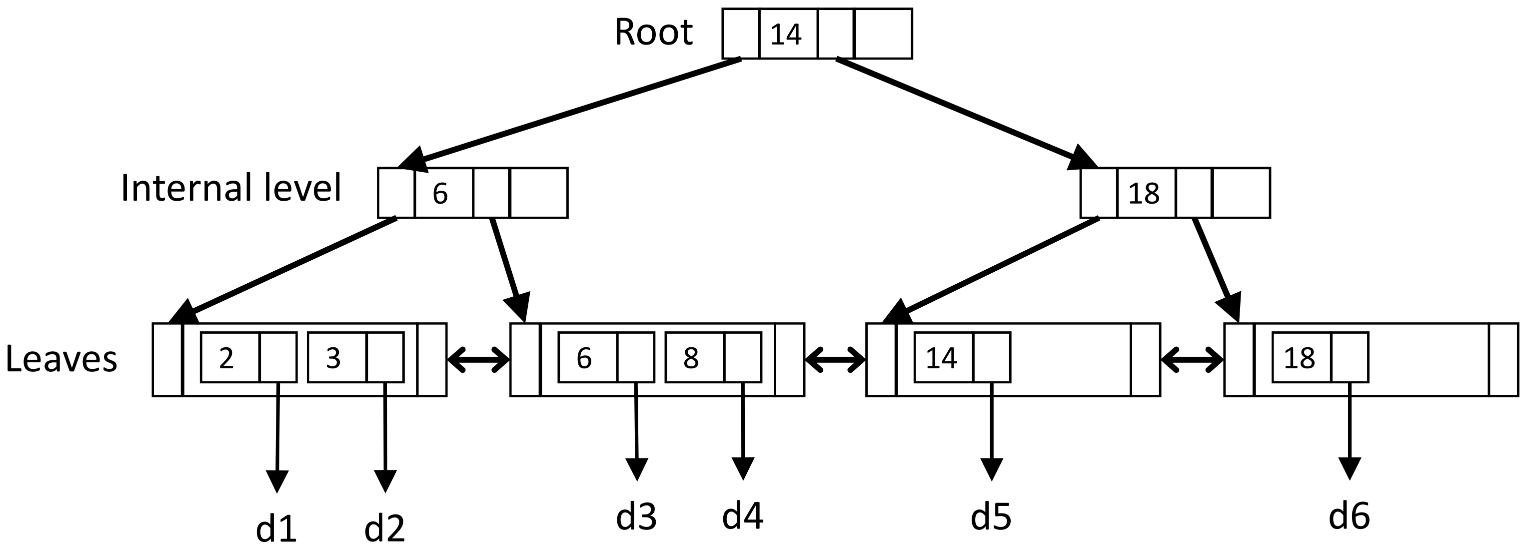




Catalog Management

Challenge II

Metadata hierarchical structure



HBase Component Roles

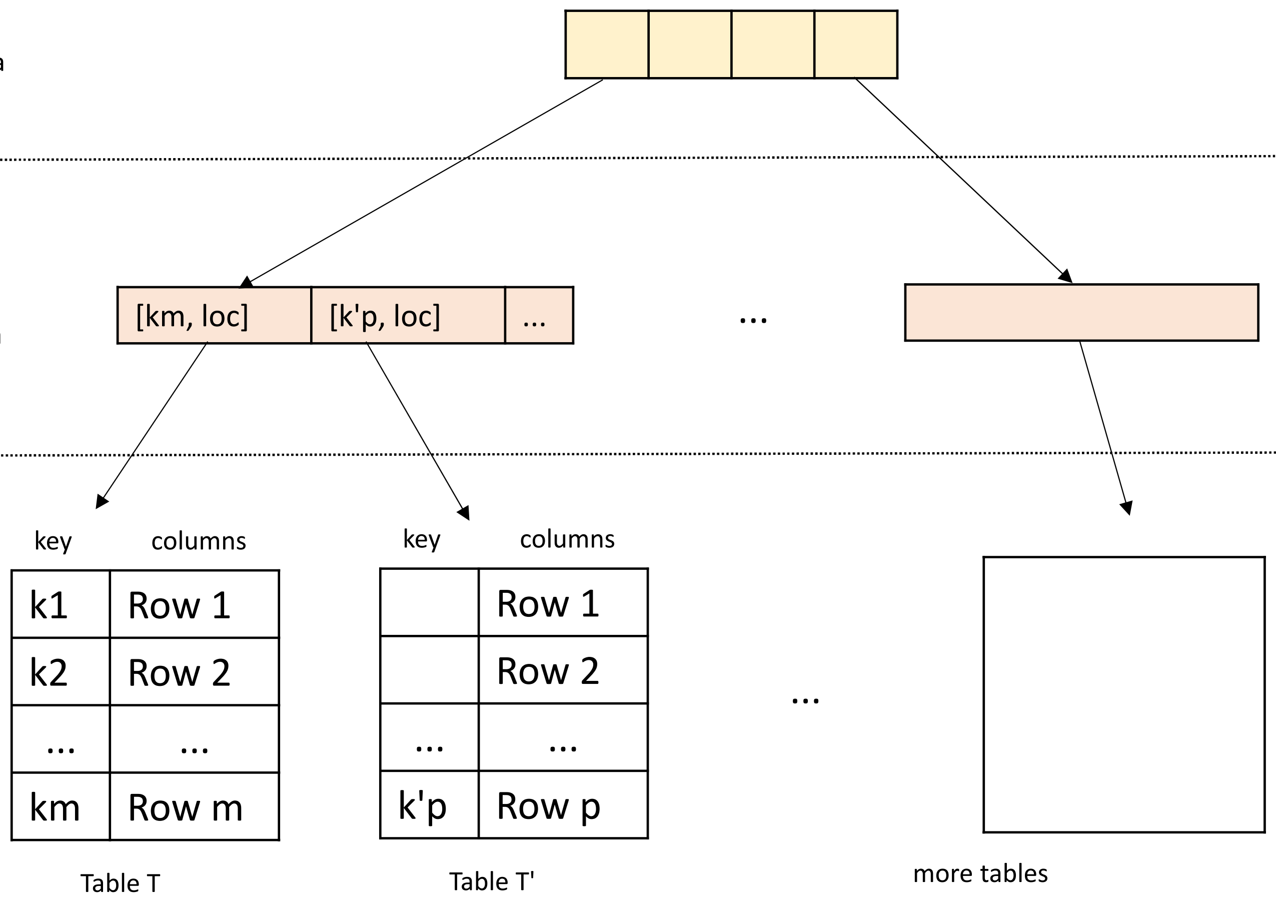
- One coordinator server
  - Maintenance of the table schemas
    - Root region
  - Monitoring of services (heartbeating)
  - Assignment of regions to servers
- Many region servers
  - Each handling around 100-1.000 regions
  - Apply concurrency and recovery techniques
  - Managing split of regions
    - Regions can be sent to another server (load balancer)
  - Managing merge of regions
- Client nodes
  - Cache the metadata sent by the region servers

Metadata hierarchical structure

Root table
Store locations of metadata tables

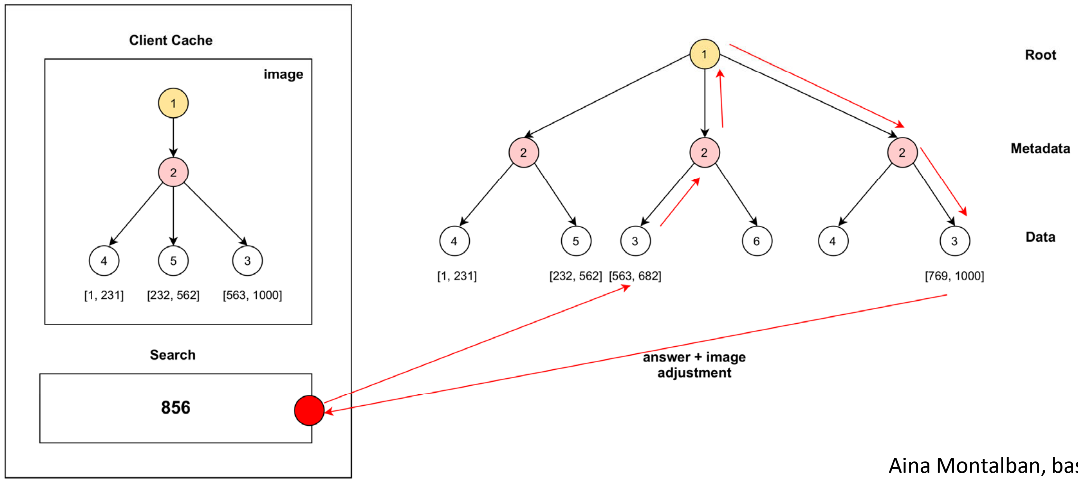
Metadata tables
Store locations of user data tables

User data tables
Store data



Metadata synchronization in HBase

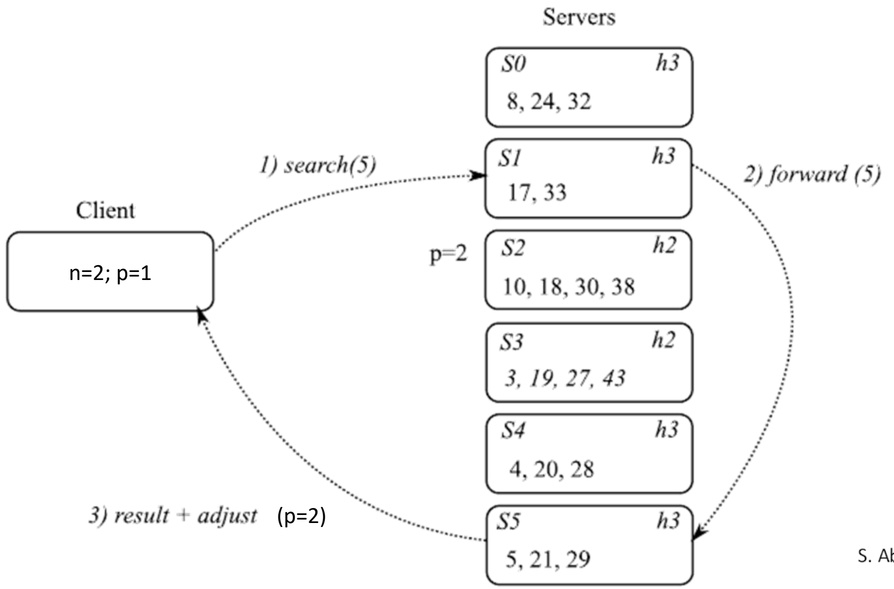
- Split and merge invalidate the cached metadata
  a) Gossiping
  when anything changes, changes are communicated to every other servers
  Generates lots of messages
  not usually used in HBase
  b) Lazy updates
  - Discrepancies may cause out-of-range errors, which trigger a stabilization protocol (i.e., mistake compensation)
    - Apply forwarding path
      - If an out-of-range error is triggered, it is forwarded upward
      - In the worst case (i.e., reaching the root), 6 network round trips



Aina Montalban, based on 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.

Transaction Management

Challenge III

Single row guarantees

- Atomicity
  - Only single row guarantees (even across families)
    - … since different families of a row are not distributed
- Consistency
  - Replication relies on HDFS

Does not provide any type of consistency check like primary key, foreign key
Relies on replication of HDFS

- Isolation
  - Locking mechanism at row level
    - Does not guarantee snapshot isolation (only read committed)
      - Rows (all families) are separately consistent
      - Sets of rows may not be consistent as a whole
- Durability
  - Write Ahead Log implementation

https://hbase.apache.org/acid-semantics.html

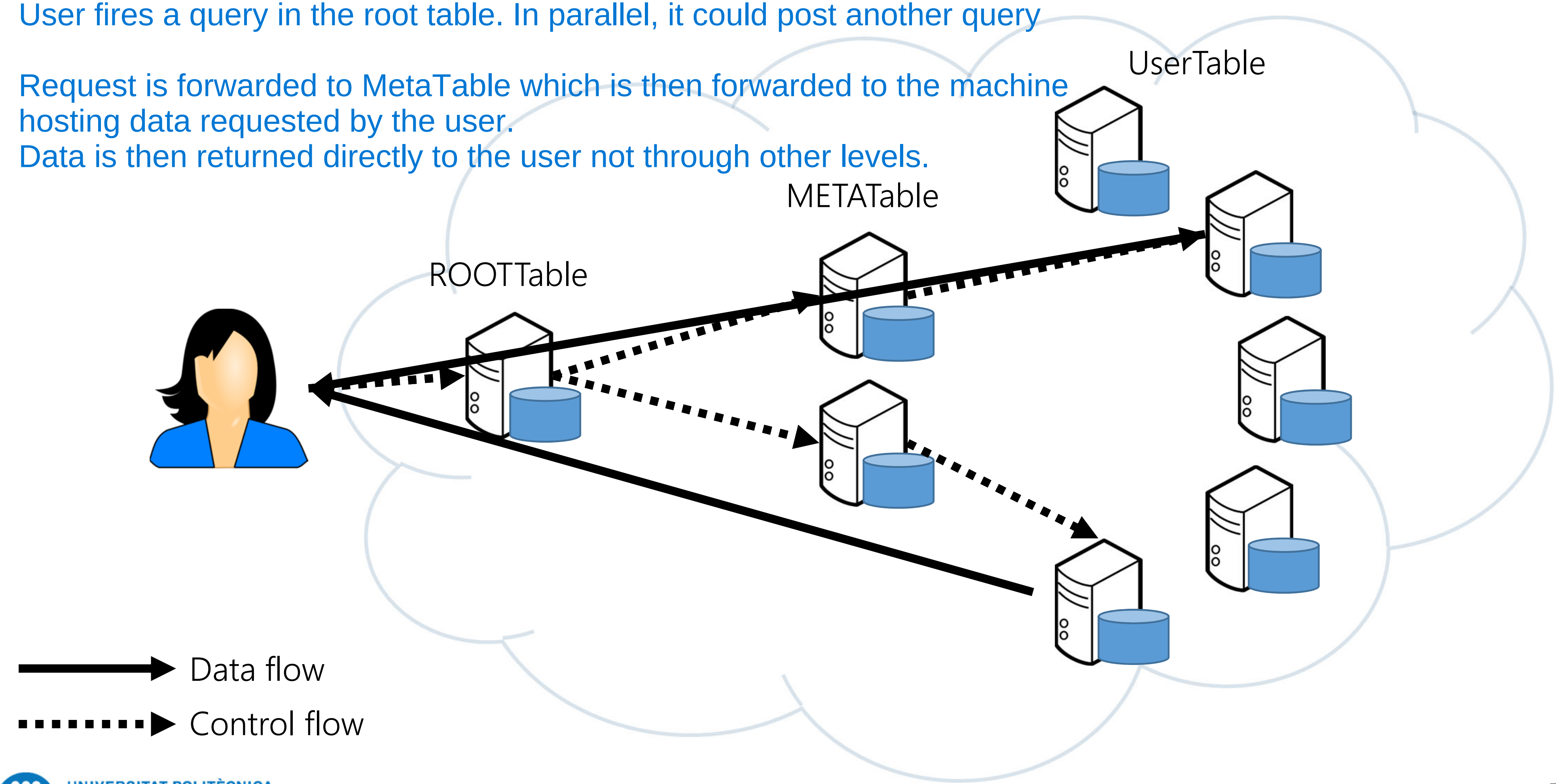
Query processing

Challenge IV

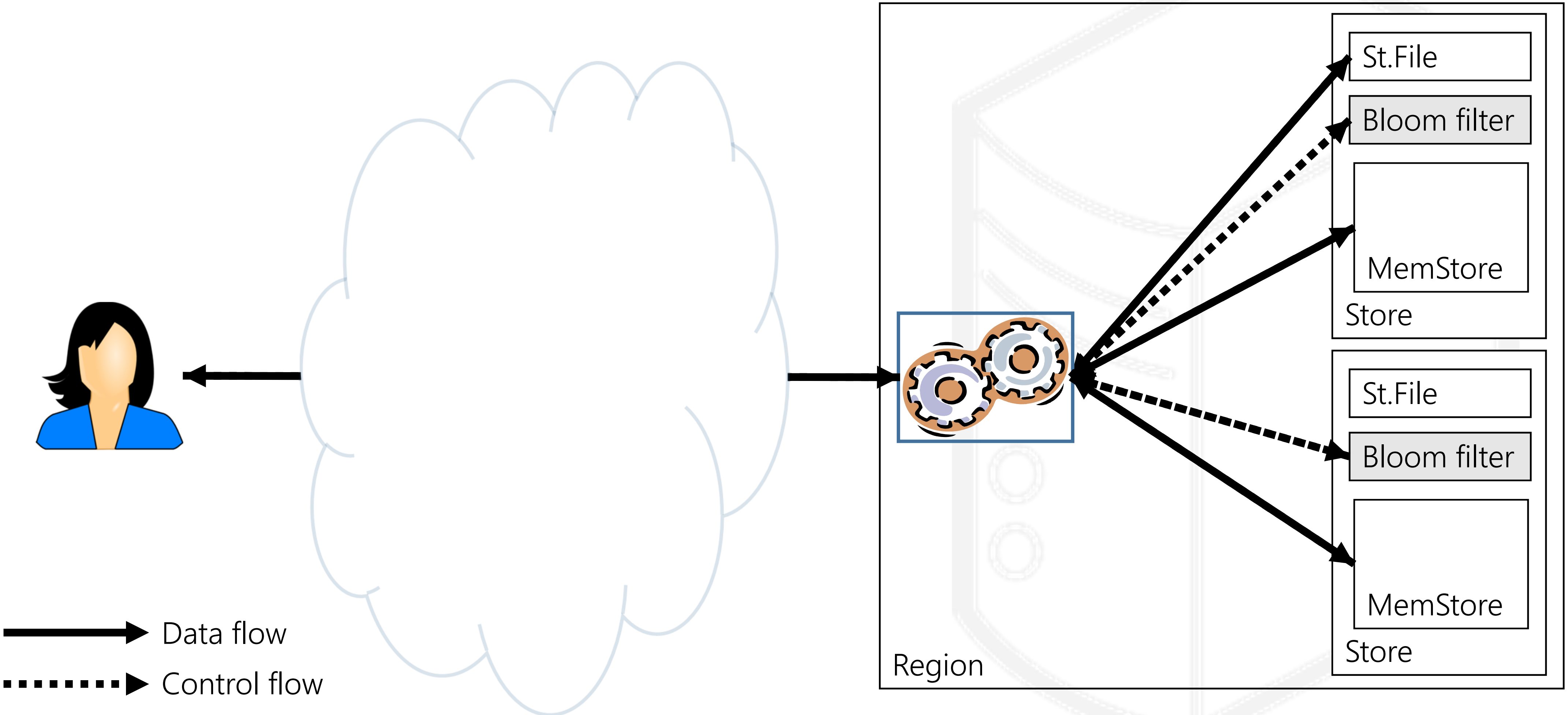
Global execution

User fires a query in the root table. In parallel, it could post another query

Request is forwarded to MetaTable which is then forwarded to the machine hosting data requested by the user.
Data is then returned directly to the user not through other levels.



Local execution



Once the request of the user reaches the execution manager in the region.
Execution Manager checks if the data is in the MemStore. If present it fetched data from MemStore and from SSTable. before looking into SSTable, it goes into the bloom filter. This is the filter that probabilistically says if the key can be present in Store File or not.

Closing

Summary

- Key-value abstraction
- HBase functional components
- Directory caching
  - Mistake compensation
- Data distribution structures
  - LSM-Tree
  - Linear hash
  - Consistent hash

References

- P. O'Neil et al. *The log-structured merge-tree (LSM-tree).* Acta Informatica, 33(4). Springer, 1996
- F. Chang et al. *Bigtable: A Distributed Storage System for Structured Data*. OSDI'06
- S. Abiteboul et al. Web Data Management. Cambridge University Press, 2011. http://webdam.inria.fr/Jorge
- N. Neeraj. *Mastering Apache Cassandra*. Packt, 2015
- O. Romero et al. *Tuning small analytics on Big Data: Data partitioning and secondary indexes in the Hadoop ecosystem*. Information Systems, 54. Elsevier, 2016
- A. Petrov. *Algorithms Behind Modern Storage Systems*. Communications of the ACM 61(8), 2018