Key-Value Stores

Big Data Management

Definition

A **key-value** store is a simple NoSQL database using an associative array (similar to a map or dictionary) as its core data model. In this model, each key maps to exactly one arbitrary value within a collection.

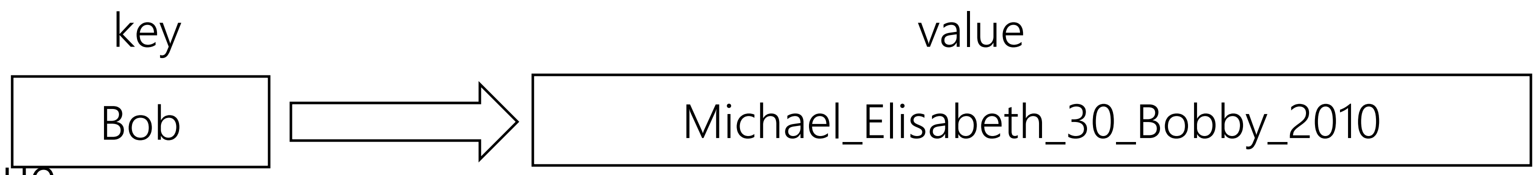
- No query language
- Simple operations: *get*, *put*, and *delete*

Wide-column stores are key-value stores that further structure the value into families, which contain groups of attributes (a.k.a. columns).

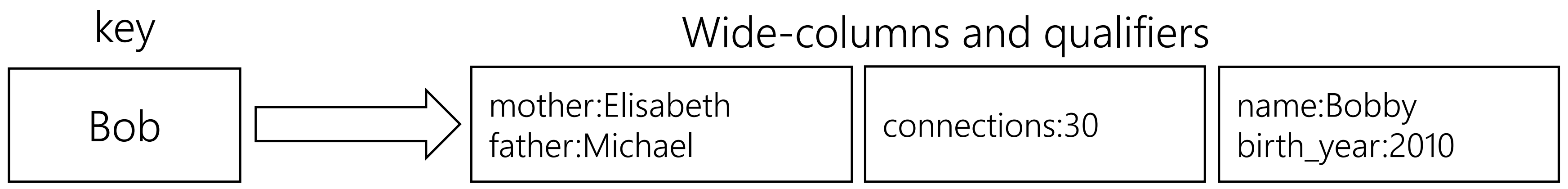
source : https://aerospike.com/glossary/what-is-a-key-value-store

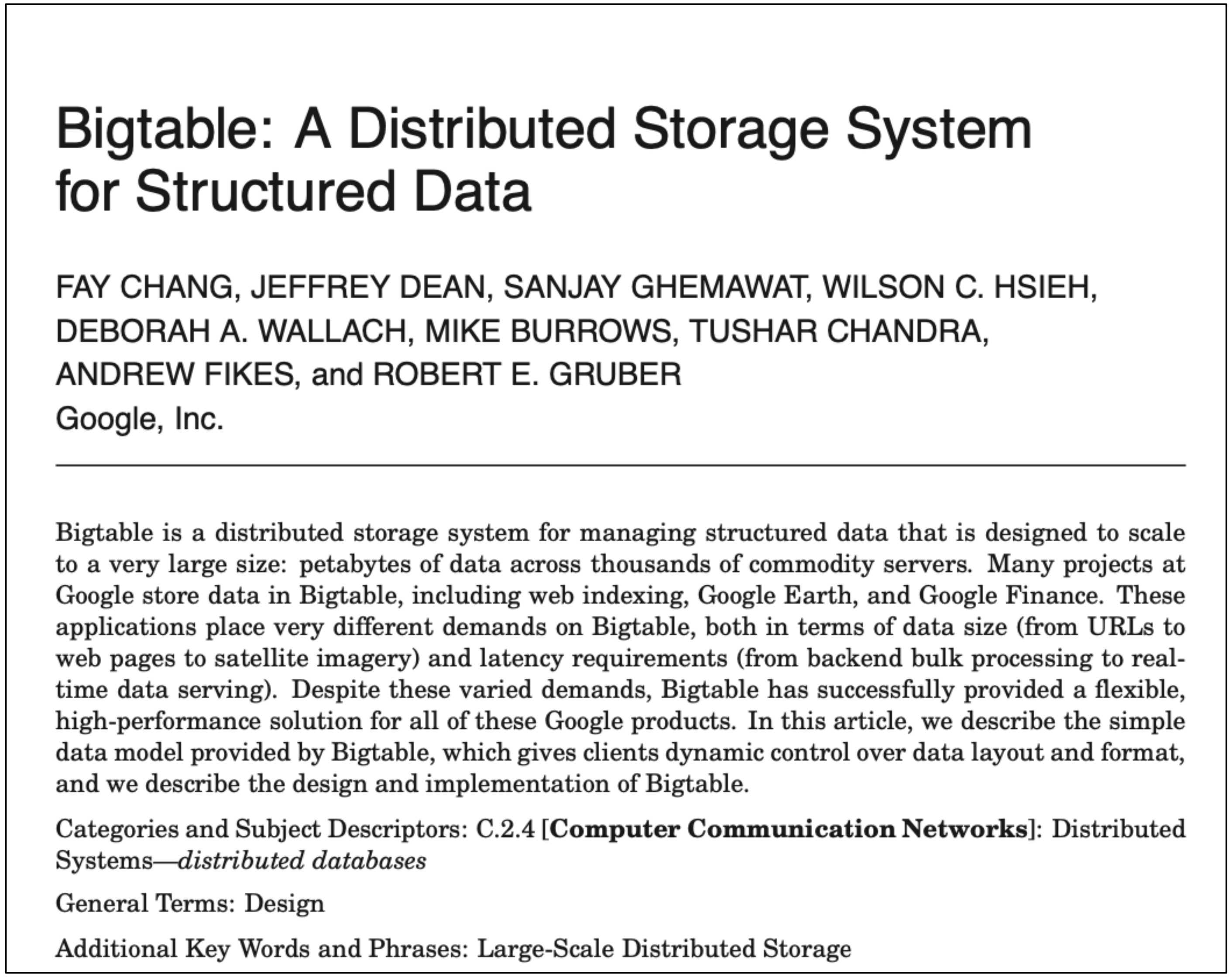
Definition

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



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





Bigtable: A Distributed Storage System for Structured Data

FAY CHANG, JEFFREY DEAN, SANJAY GHEMAWAT, WILSON C. HSIEH, DEBORAH A. WALLACH, MIKE BURROWS, TUSHAR CHANDRA, ANDREW FIKES, and ROBERT E. GRUBER
Google, Inc.

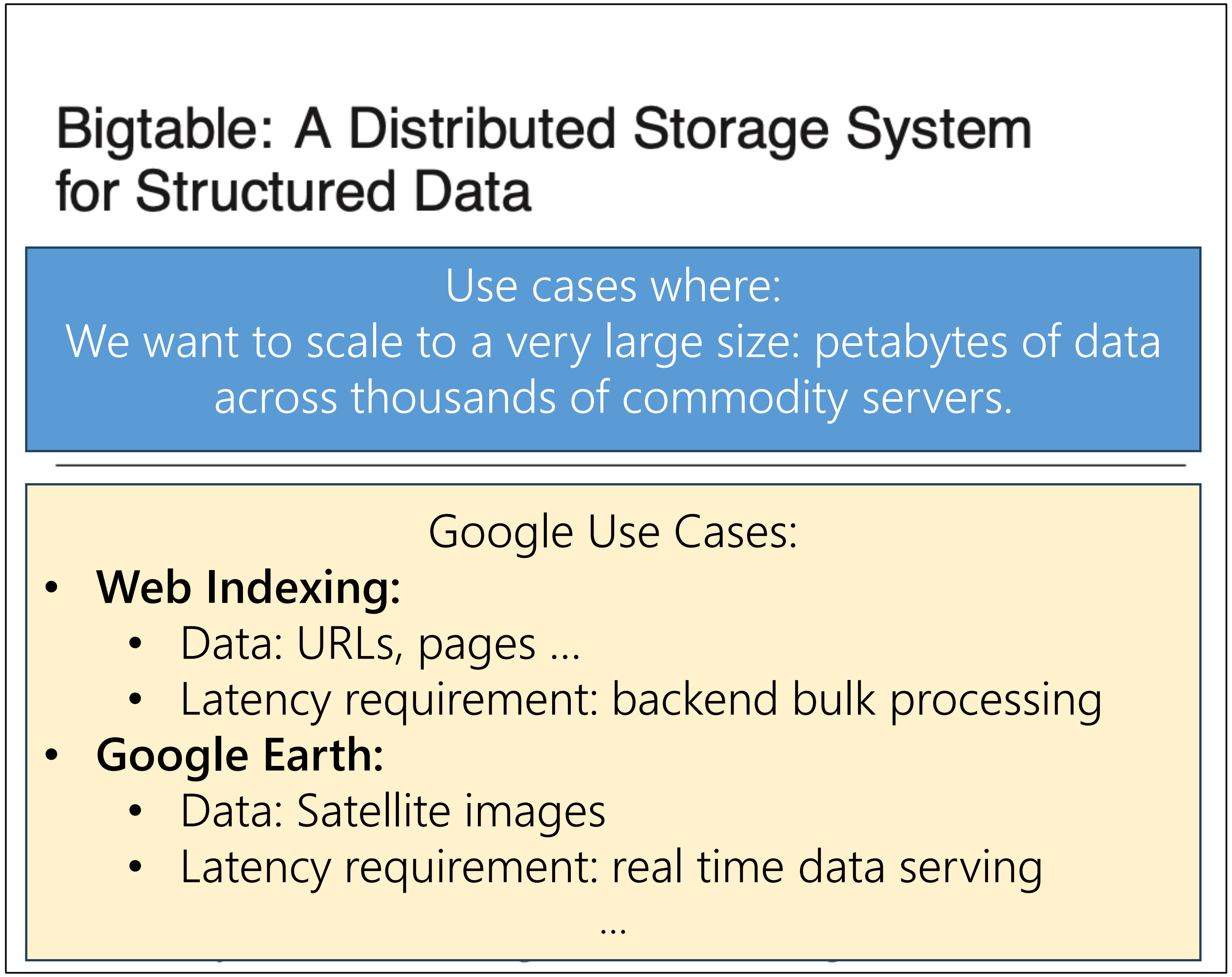
Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this article, we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed databases*

General Terms: Design

Additional Key Words and Phrases: Large-Scale Distributed Storage

OSDI '06



Bigtable: A Distributed Storage System for Structured Data

Use cases where:
We want to scale to a very large size: petabytes of data across thousands of commodity servers.

Google Use Cases:

- **Web Indexing:**
  - Data: URLs, pages …
  - Latency requirement: backend bulk processing
- **Google Earth:**
  - Data: Satellite images
  - Latency requirement: real time data serving

…

BigTable Data Model

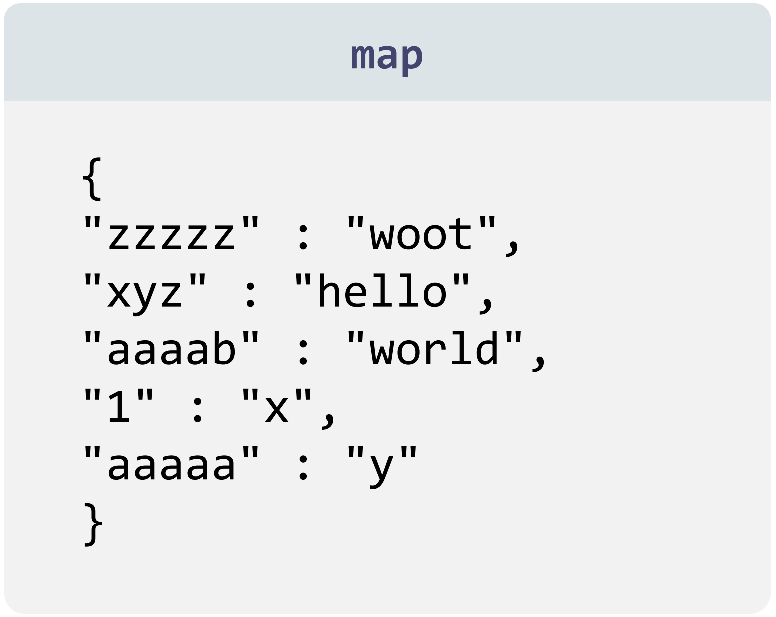
"A BigTable is a **sparse**, **distributed**, **persistent**, **multi-dimensional**, **sorted** map."

F. Chang et al.

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

map

- At its core BigTable is a map
  - In other programming languages: associative array (PHP), dictionary (Python), Hash (Ruby), or Object (Javascript)

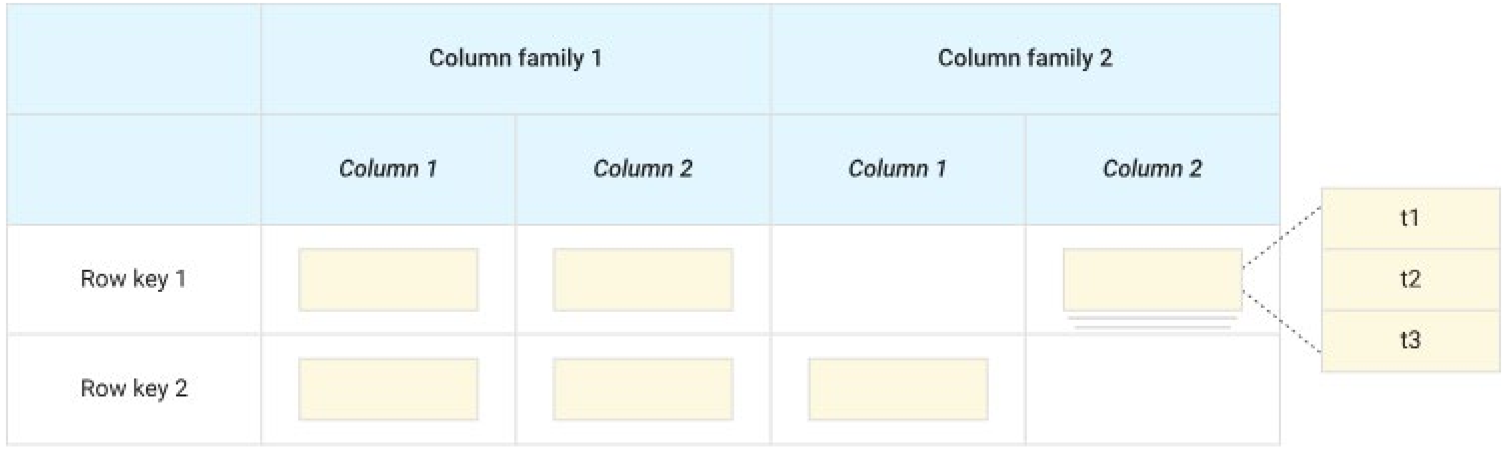


map

```
{
"zzzzz" : "woot",
"xyz" : "hello",
"aaaab" : "world",
"1" : "x",
"aaaaa" : "y"
}
```

sparse

- If a column is not used in a particular row, it does not take up any space. Optional values are not represented by placeholders or input parameters. Memory efficient.



source: https://cloud.google.com/bigtable/docs/overview

distributed

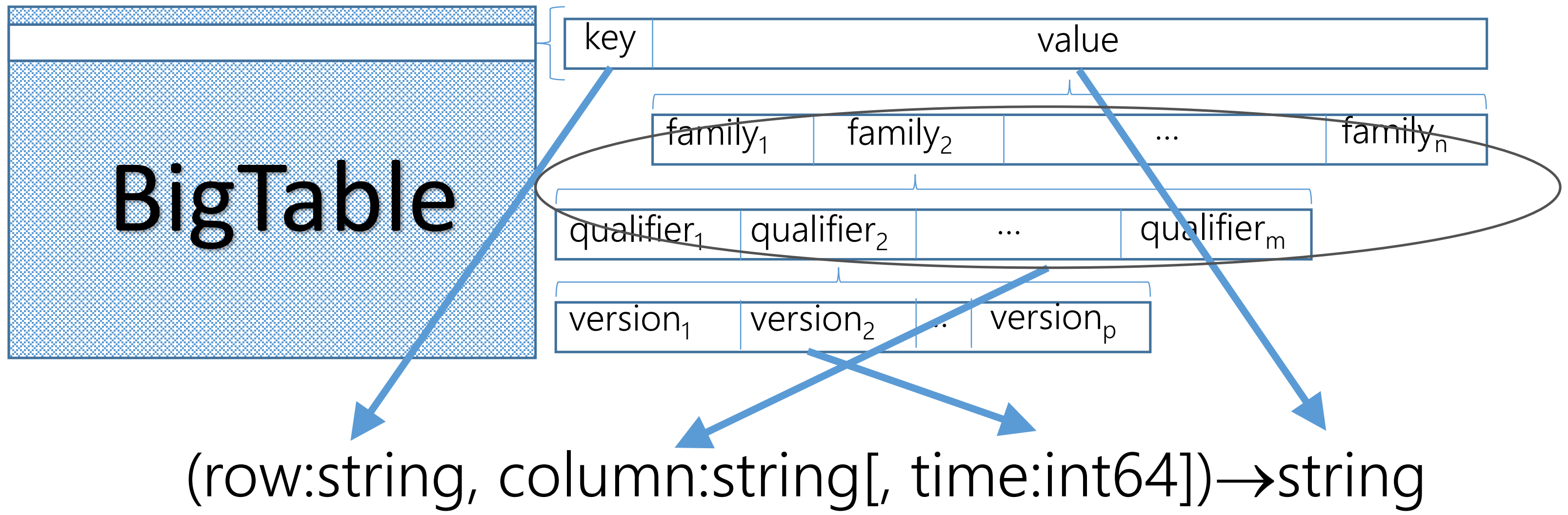
- BigTable are built upon distributed file systems, so that the underlying file storage can be spread out among an array of independent machines
    - BigTable makes use of the GFS

persistent

- Persistent means that the data you put in the special map “persists” after the program that created or accessed it finished
  - E.g., Similar to a file in a file system

multidimensional

- Different `levels' in the map
- Each cell can contain multiple versions of same data
  - (Row Key) x (Column Key) x (Timestamp) → Value
  - Column Key → family:qualifier



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

sorted

- Bigtable maintains data in lexicographic order by row key which are indexed
- The row range for a table is dynamically partitioned; each called a **tablet**
- A tablet is the unit of distribution and load balancing in BigTable

```
sorted map

{
"1" : "x",
"aaaaa" : "y",
"aaaab" : "world",
"xyz" : "hello",
"zzzzz" : "woot"
}
```

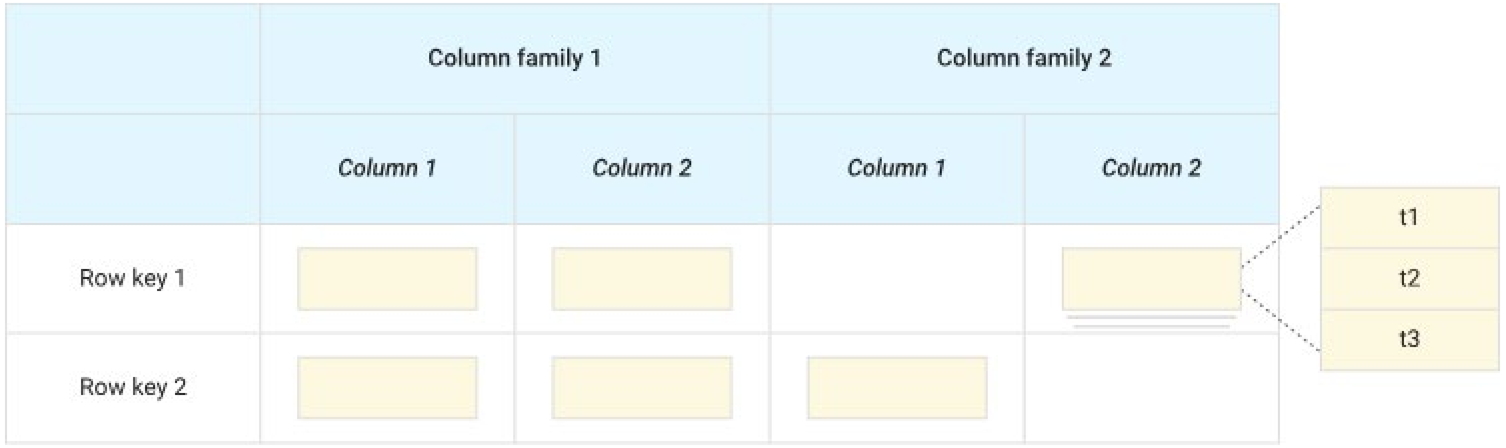
UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

source: https://dzone.com/articles/understanding-hbase-and-bigtab

DTIM www.essi.upc.edu/dtim

BigTable Data Model

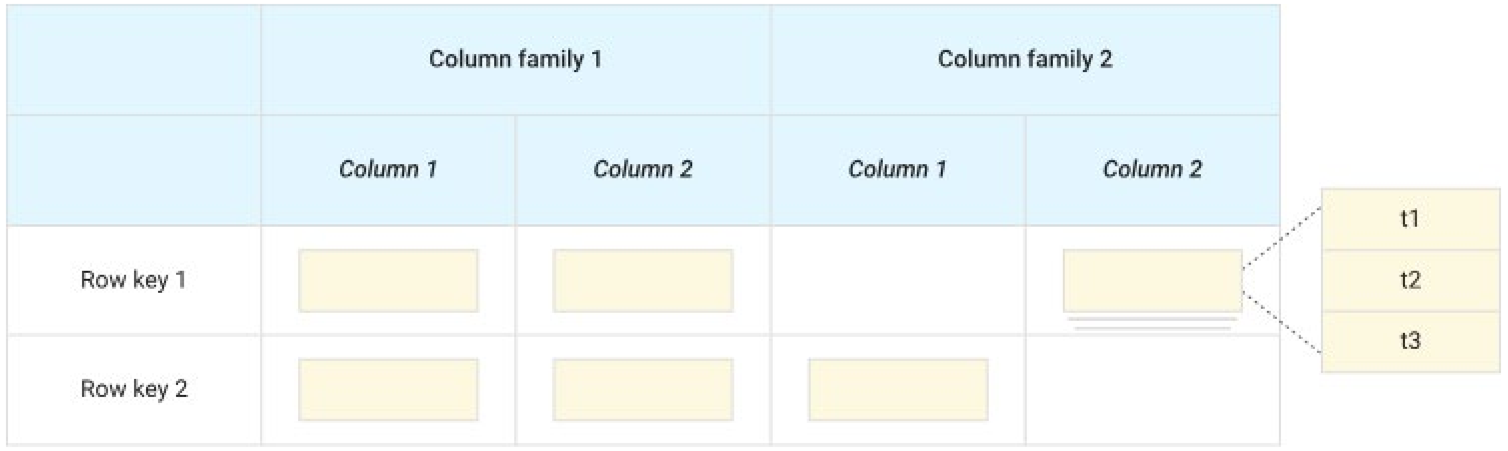
- Stores data in tables composed of indexed rows identified by **row keys**
  - Each row is a set of key-value pairs, where each key is defined by the combination of **column family**, **column qualifier**, and **timestamp**.
- Data is stored as uninterpreted strings (no predefined data types)
- Each cell can store multiple versions of data, each version identified by a unique **timestamp**.



source: https://cloud.google.com/bigtable/docs/overview

BigTable Data Model

- Columns families contain one or more often related columns.
- Families are explicitly created when table is created or updated.
- Qualifiers are column names within the column family.
- Qualifiers are declared at insertion time.



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 (mutations, such as read, write, and delete requests, are always atomic at the row level)

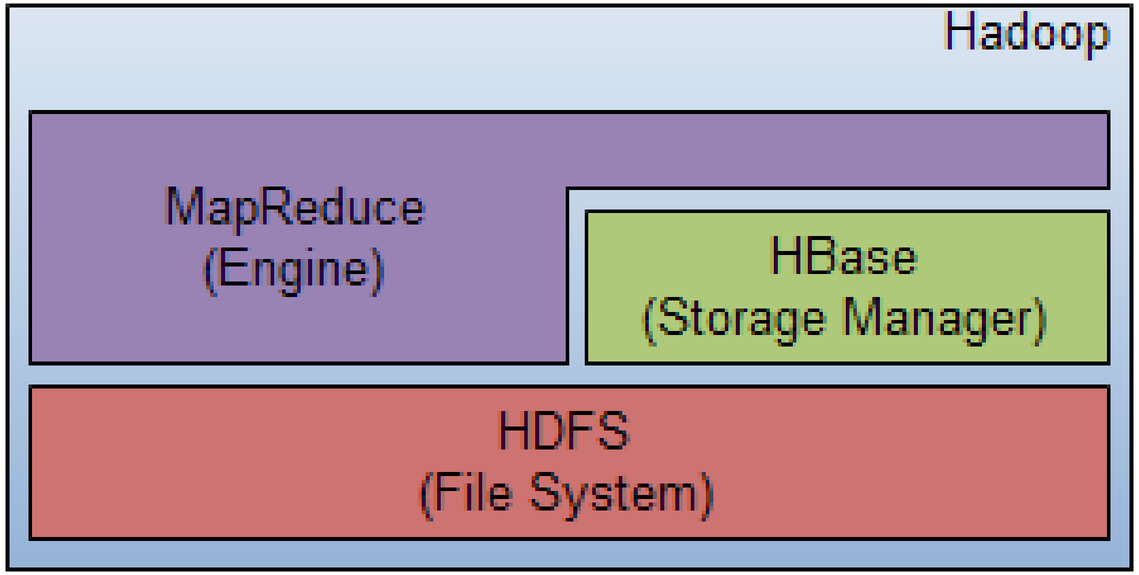
BigTable - Hbase Comparison

Feature	BigTable	HBase
Origin	Proprietary (Google)	Open-source (Apache)
Infrastructure	Cloud based	Self managed
Distribution unit	Tablets	Regions
Storage Backend	GFS (Colossus)	HDFS
Operational cost	Low (fully managed)	High (manual)

HBase Architecture

Hadoop Ecosystem

- HDFS physically stores the data in files
- **HBase manages the data, and provides basic indexing capabilities allowing random access**
- MapReduce is the query execution engine that filters, transforms and aggregates these data.

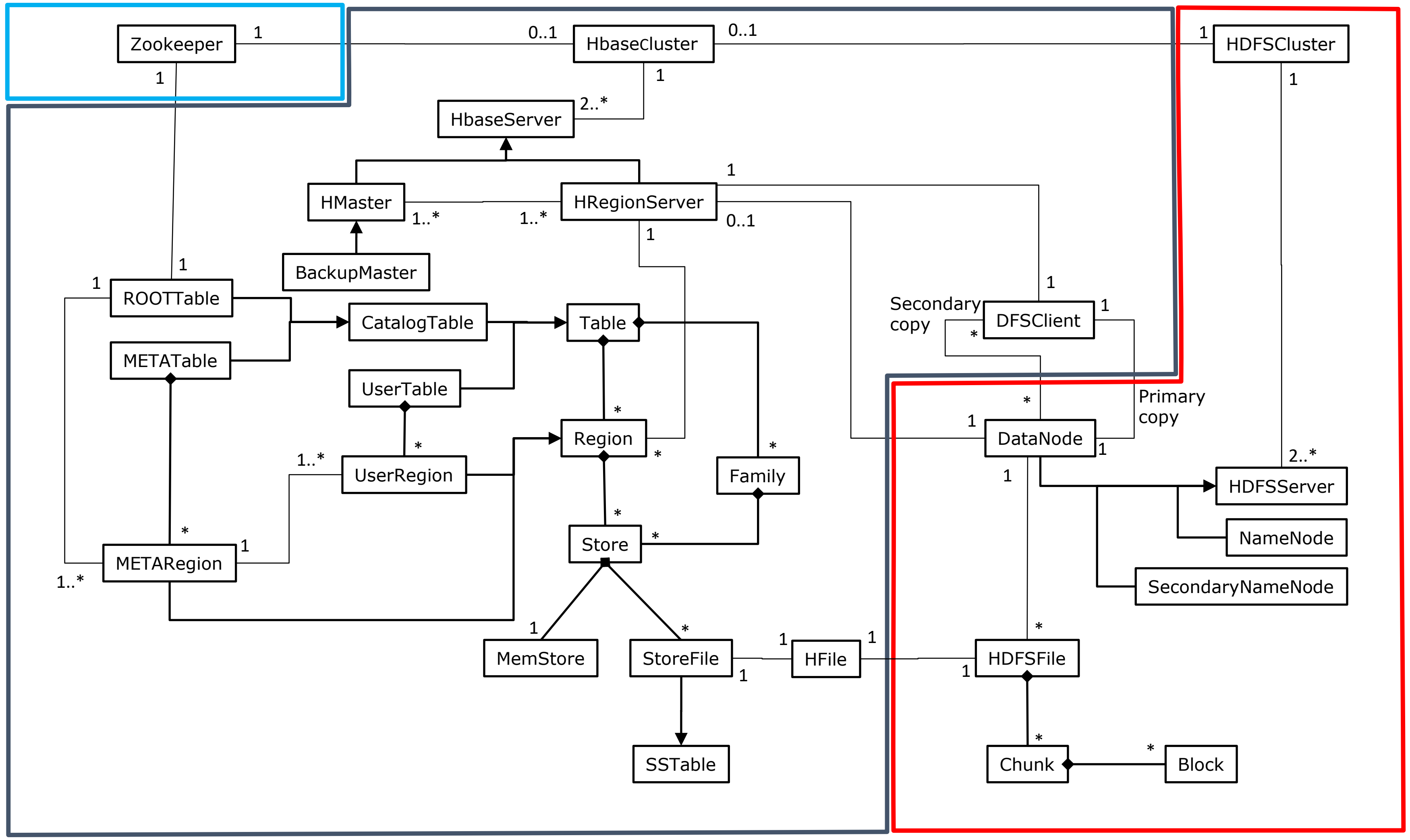


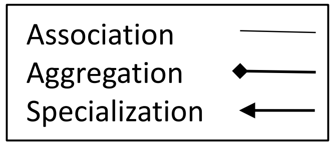
HBase shell

- `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`

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

Functional components



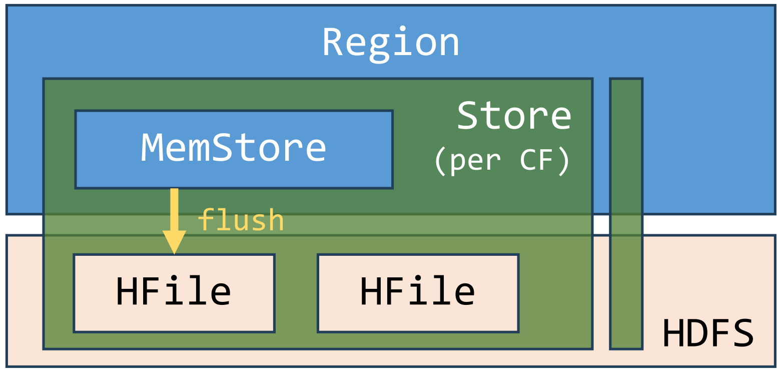


Functional components

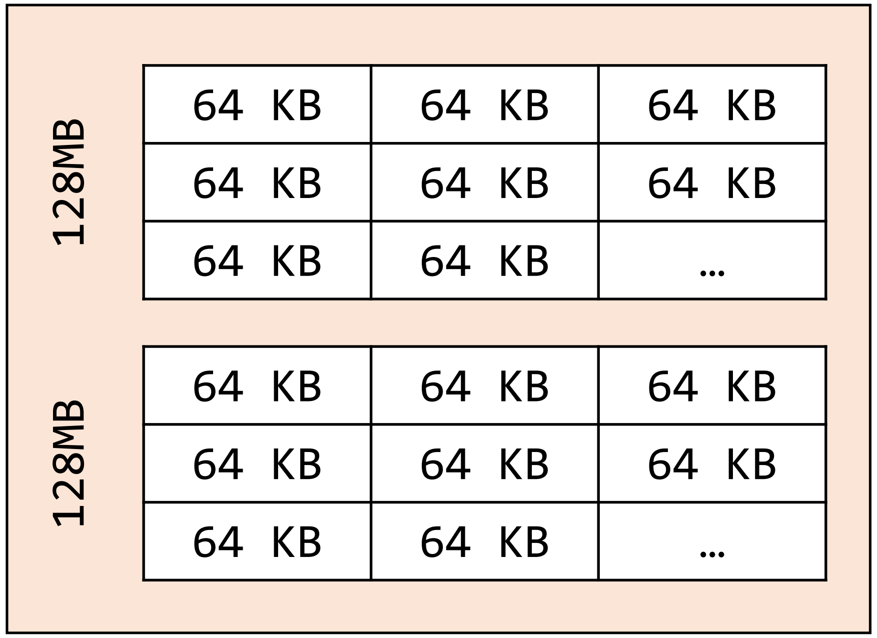
- **Zookeeper** - group of servers that stores HBase system config info
- **HMaster** - Coordinates splitting of regions/rows across nodes
  - Controls distribution of HFile chunks
- **HRegionServer** - Region Servers
  - Serves HBase client requests
    - Manage stores containing all column families of the region
  - Logs changes
  - Guarantees "atomic" updates to one column family

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)



Functional components

- Tables → Regions → Stores
- Regions are the minimal distribution unit used by HBase
- Data are physically stored in stores:
  - in in-memory buffers (memstores)
  - flushed to disk as storefiles (a.k.a. SStables)
  - storefiles are represented as HFiles (with added metadata)
- HFiles are divided into HBase blocks (64KB)
- HFiles are then written to HDFS and replicated

HBase processes

a) `Flush`

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

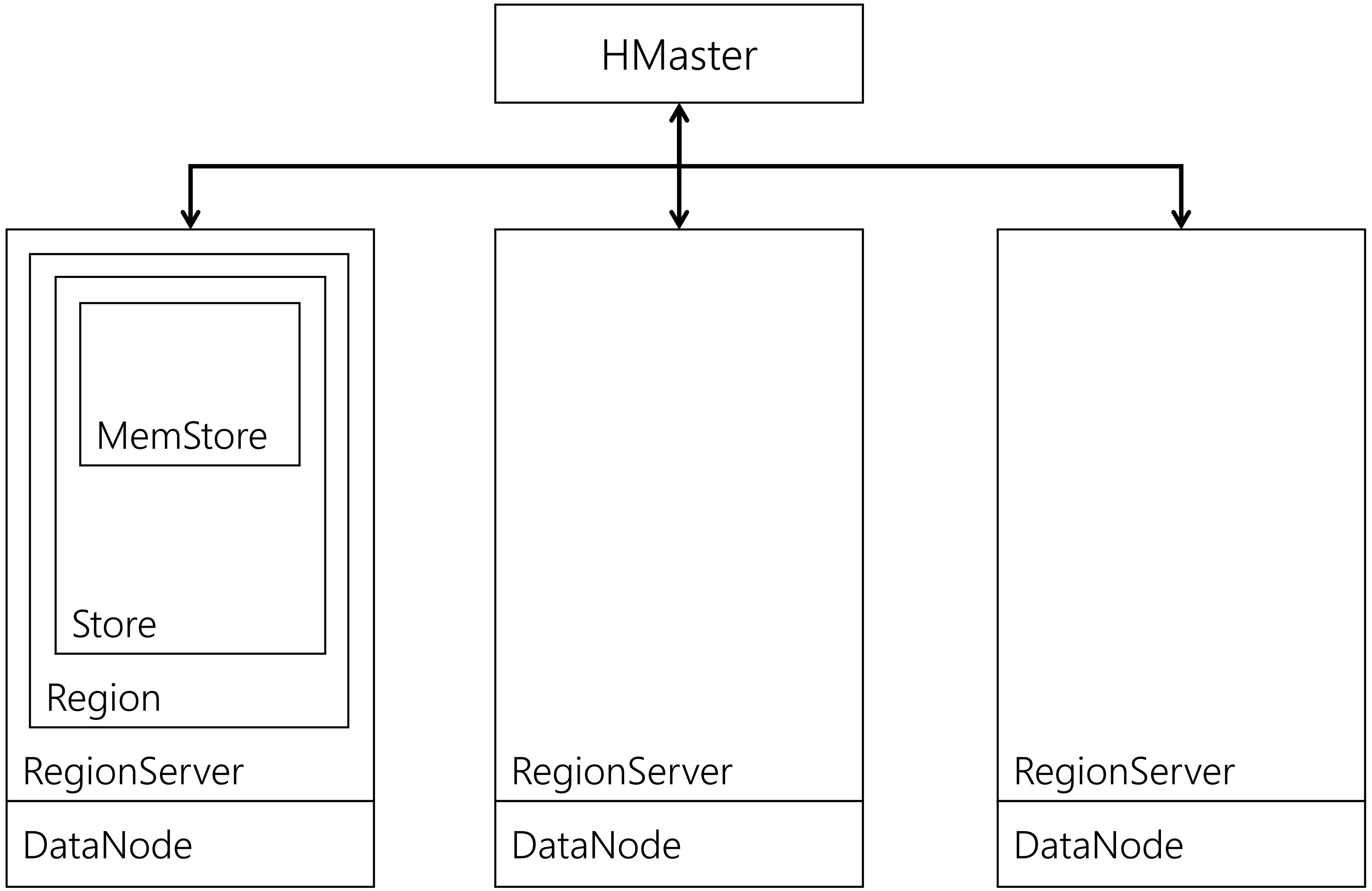
b) `Minor compactation`

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

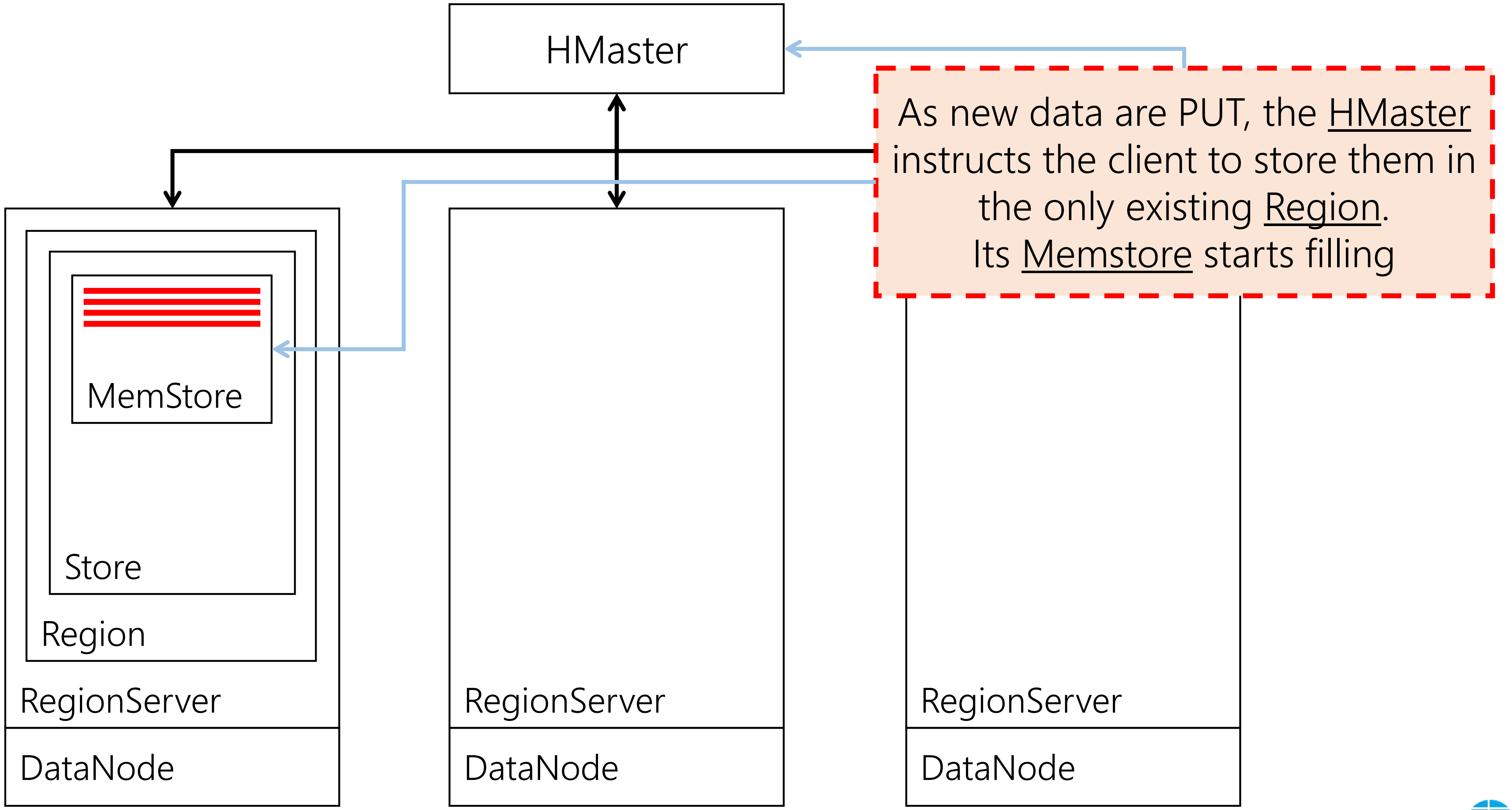
c) `Major compactation`

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

Example of Flush

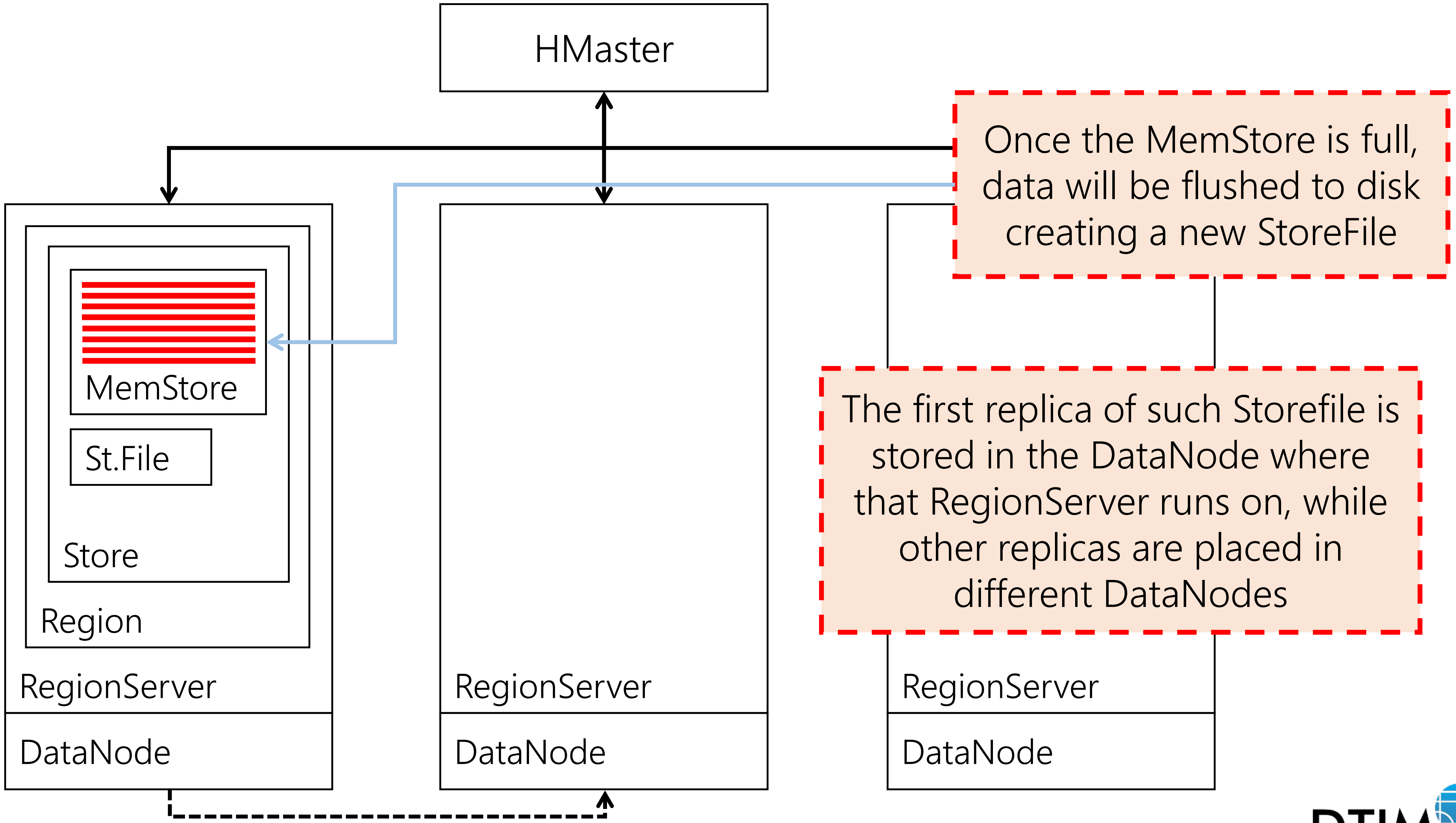


Example of Flush



As new data are PUT, the HMaster instructs the client to store them in the only existing Region.
Its Memstore starts filling

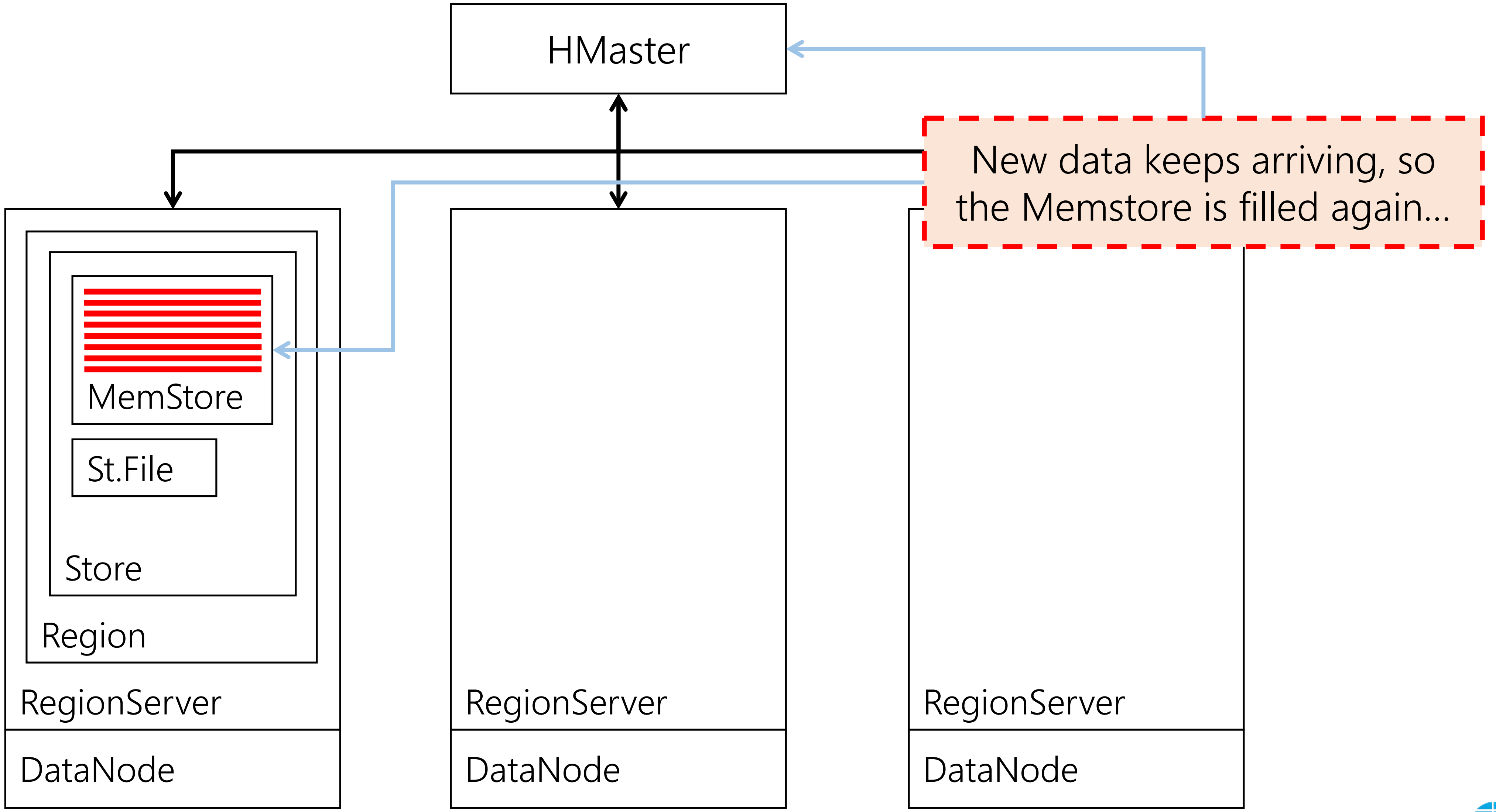
Example of Flush



Once the MemStore is full, data will be flushed to disk creating a new StoreFile

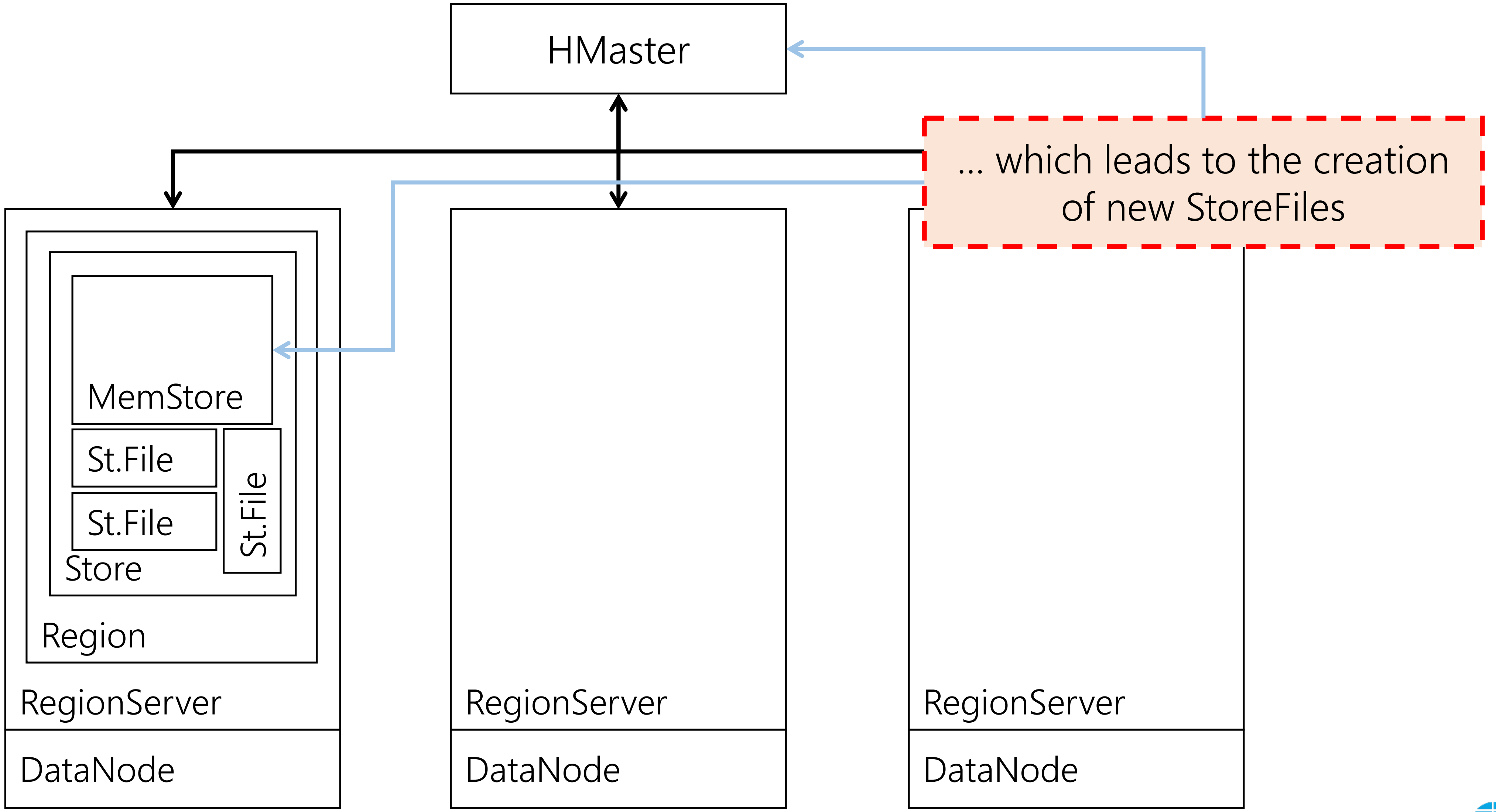
The first replica of such Storefile is stored in the DataNode where that RegionServer runs on, while other replicas are placed in different DataNodes

Example of Compactation



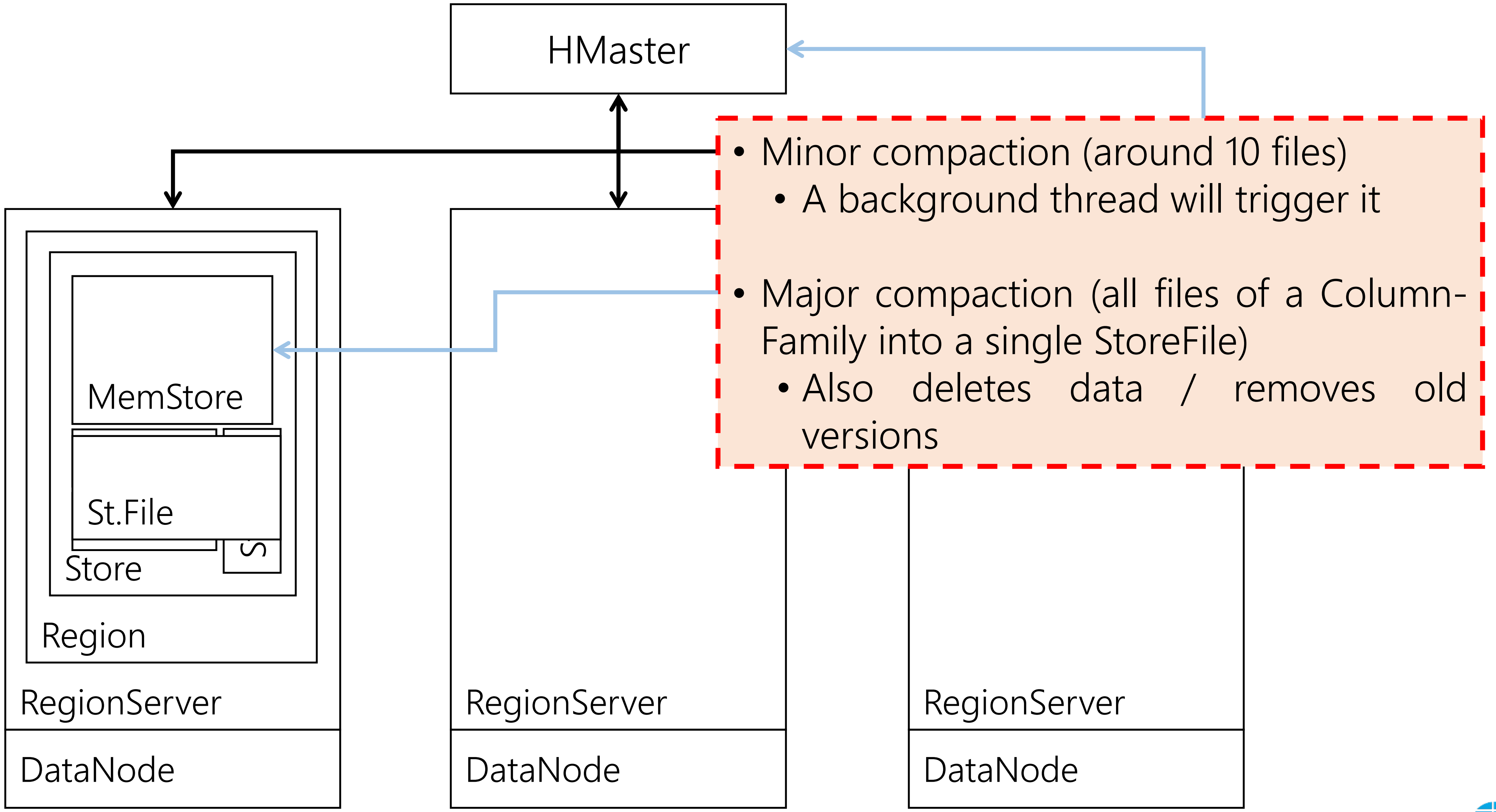
New data keeps arriving, so the Memstore is filled again…

Example of Compactation



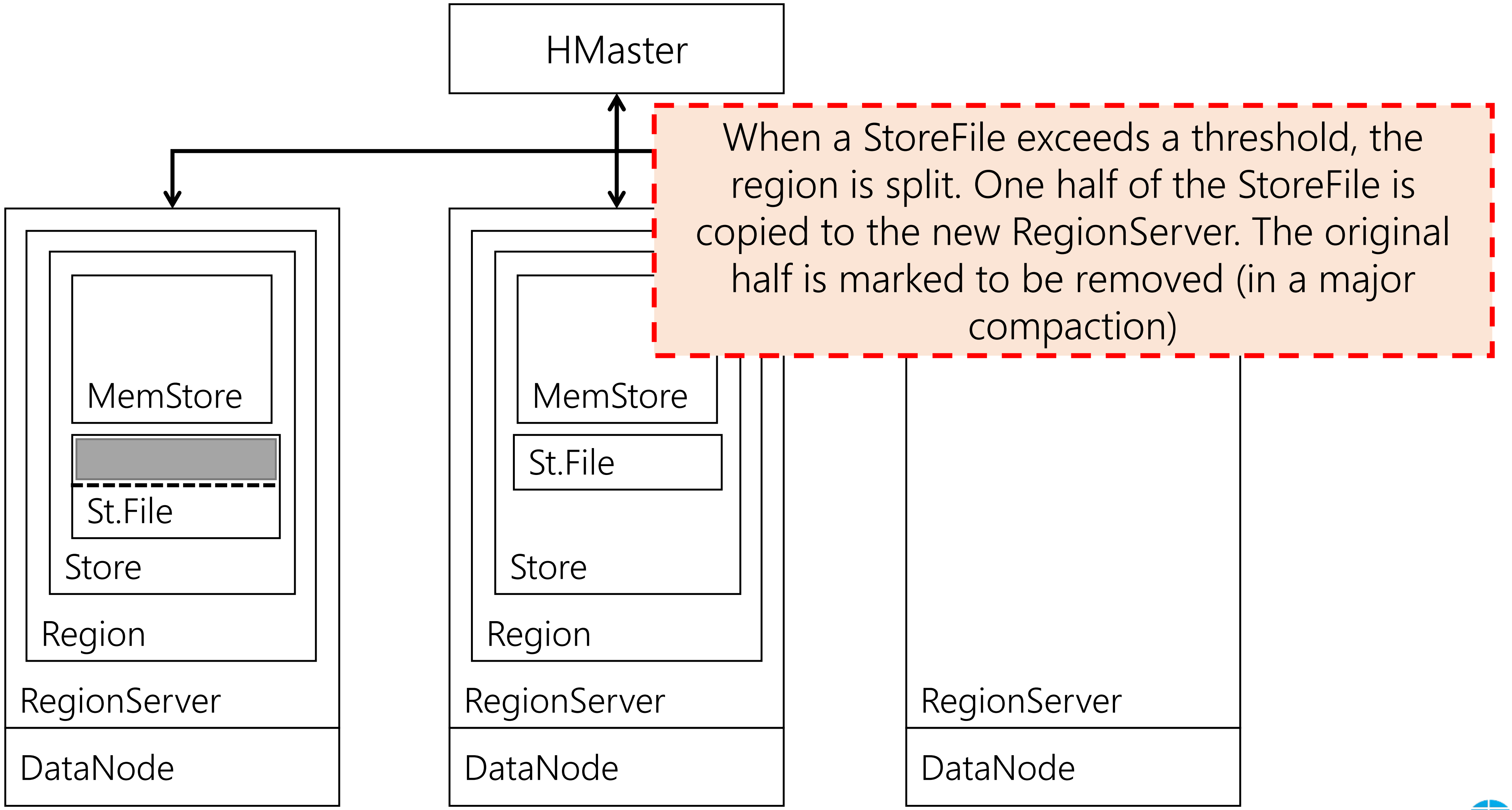
… which leads to the creation of new StoreFiles

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



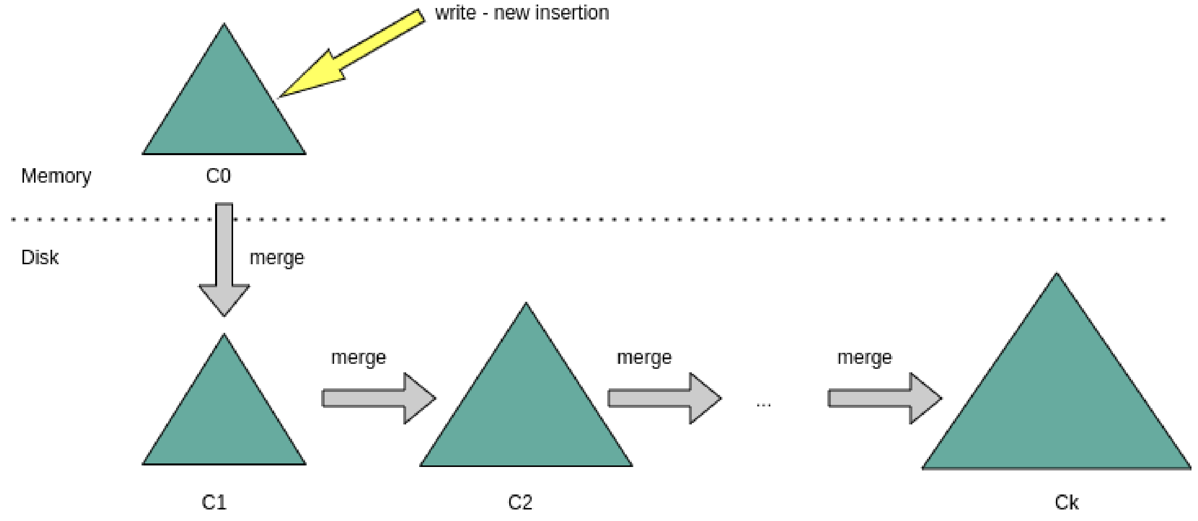
When a StoreFile exceeds a threshold, the region is split. One half of the StoreFile is copied to the new RegionServer. The original half is marked to be removed (in a major compaction)

Underlying Tree Structures

LSM-tree

Log Structured Merge-trees

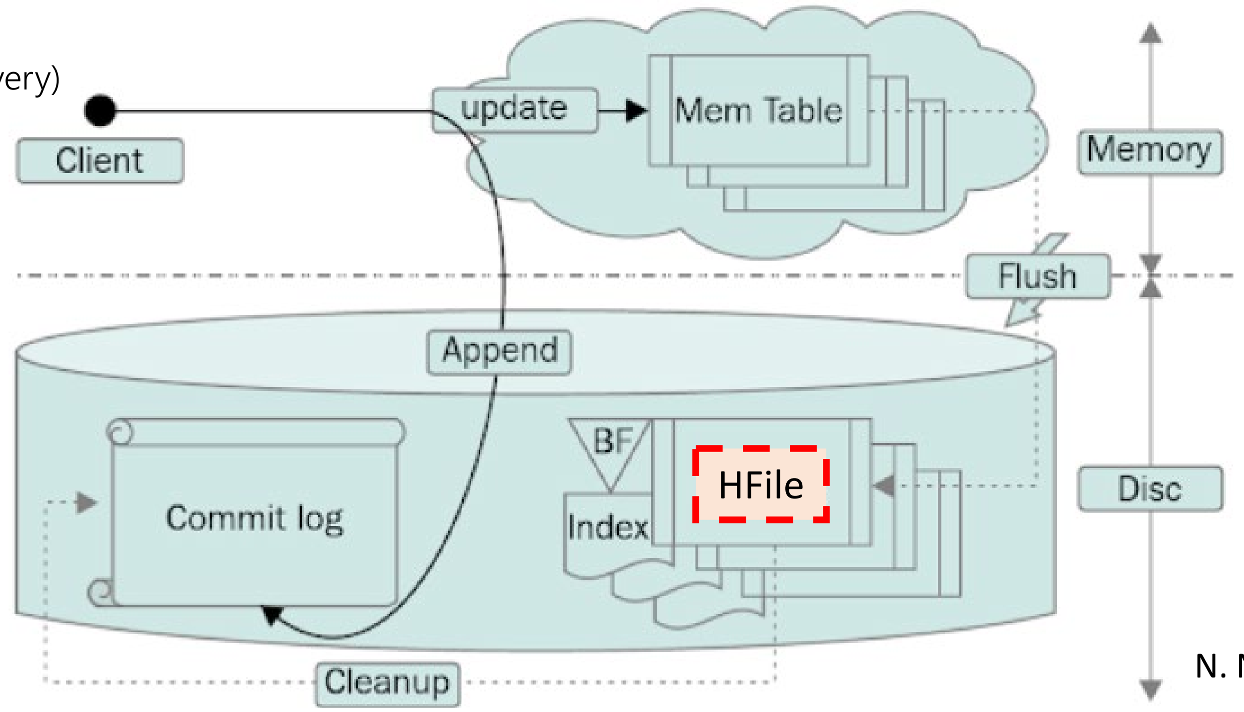
- Defers and batches index changes
- Consists on two structures
  - In Memory
    - Not necessarily a B-Tree
    - Node different from disk blocks
  - In Disk
    - Multiple components
      - Blocks 100% full
- Regularly merges trees in one level into the next one



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 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)

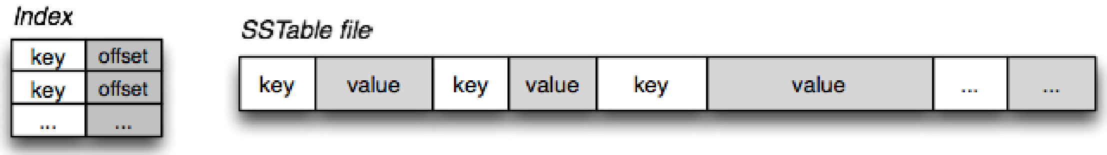


N. Neeraj

LSM-tree maintenance algorithms

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

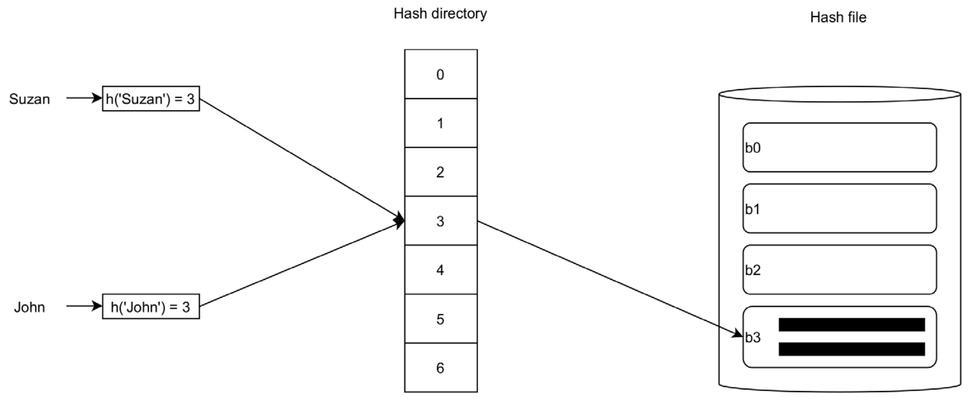
Linear hash

Consistent hash

Distributed Hashing (alternative to a tree)

- 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

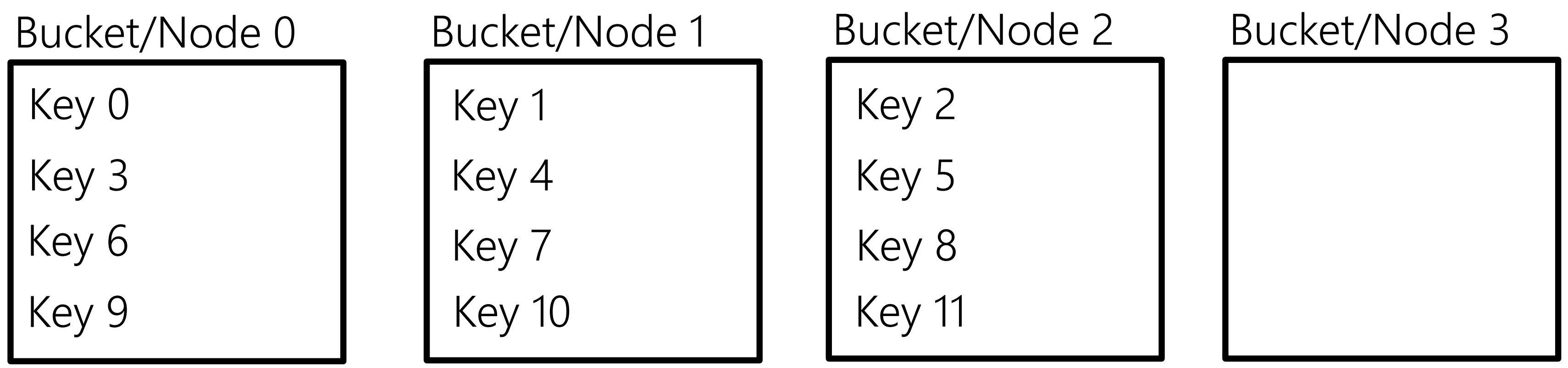


Aina Montalban, based on S. Abiteboul et al.

"%" is the mathematical modulo operation

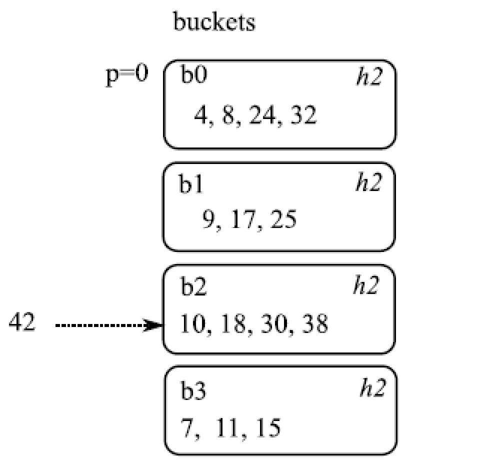
Adding a node in static hash

3 Nodes

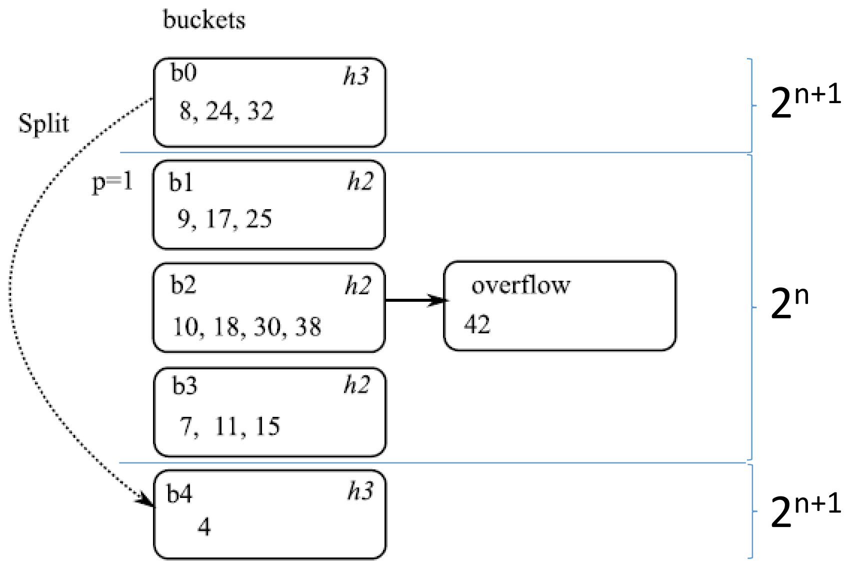


Distributed Linear Hashing (LH*)

- 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}`≤#servers<2`$^{n+1}$)
  - Overflow buckets are considered
    - When a bucket overflows the pointed bucket splits (not necessarily the overflown one)



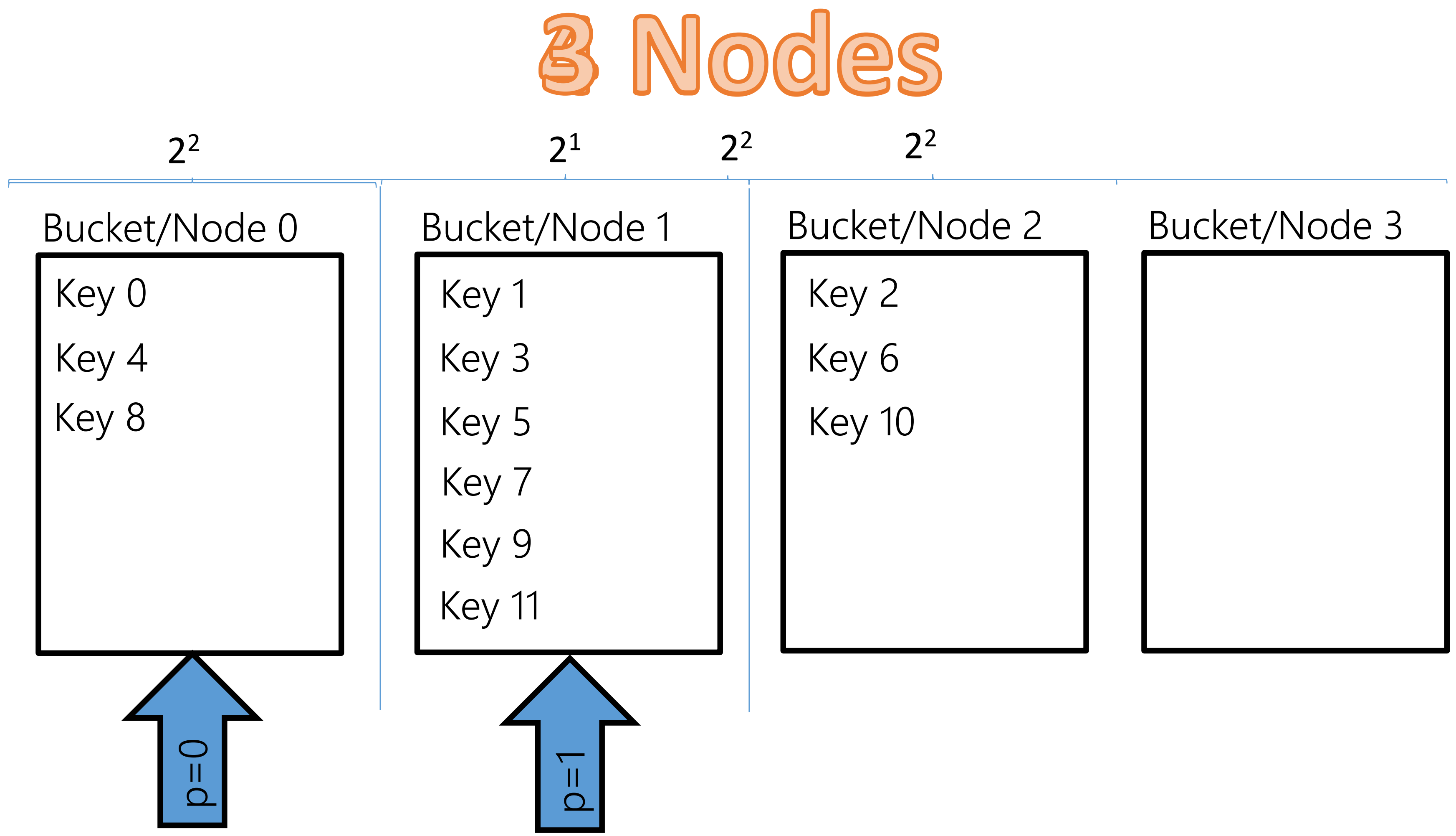
Bucket b2 receives a new object



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

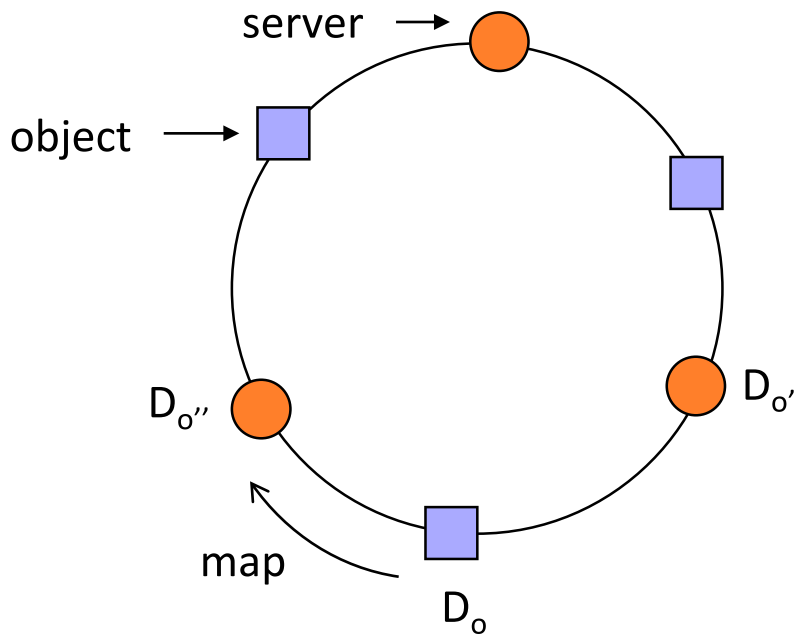
S. Abiteboul et al.

Adding a node in linear hash

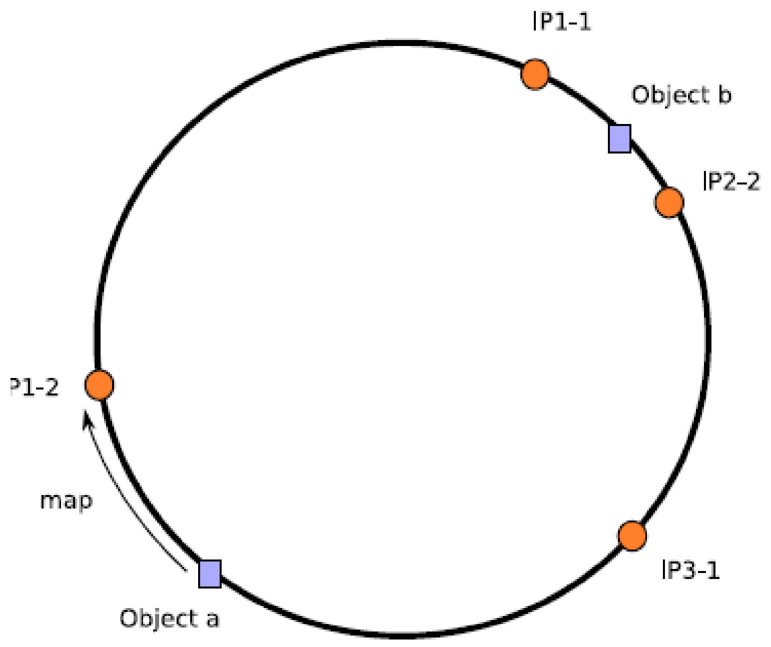


Consistent Hashing

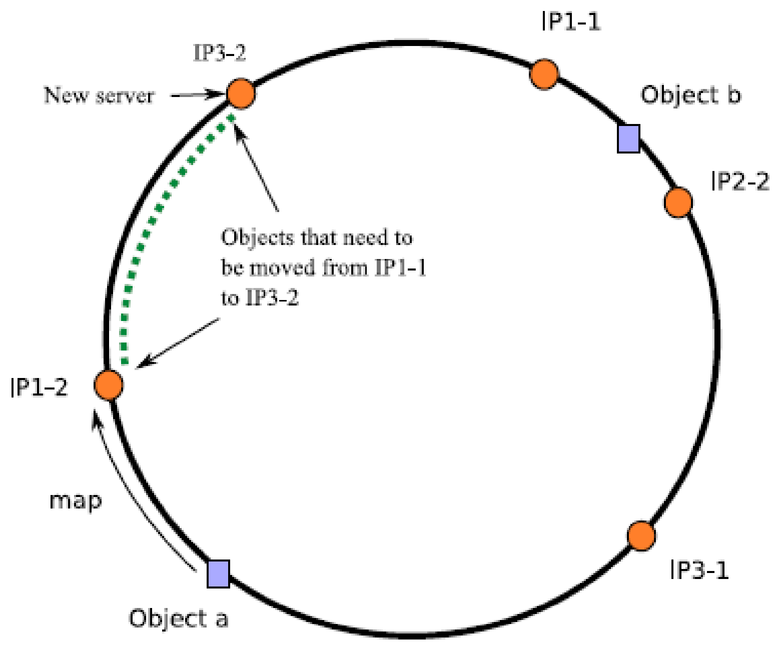
- The hash function **<u>never</u>** changes
  - Choose a very large domain ***D***
    - Map <u>server IP addresses</u> and <u>object keys</u> to such domain
  - Organize ***D*** as a ring in clockwise order
    - Each node has a successor
  - Objects are assigned as follows:
    - For an object `O, 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

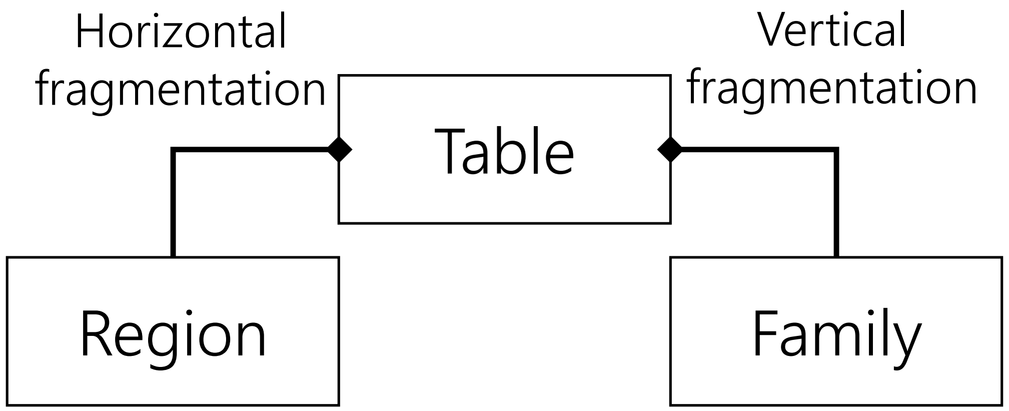
S. Abiteboul et al.

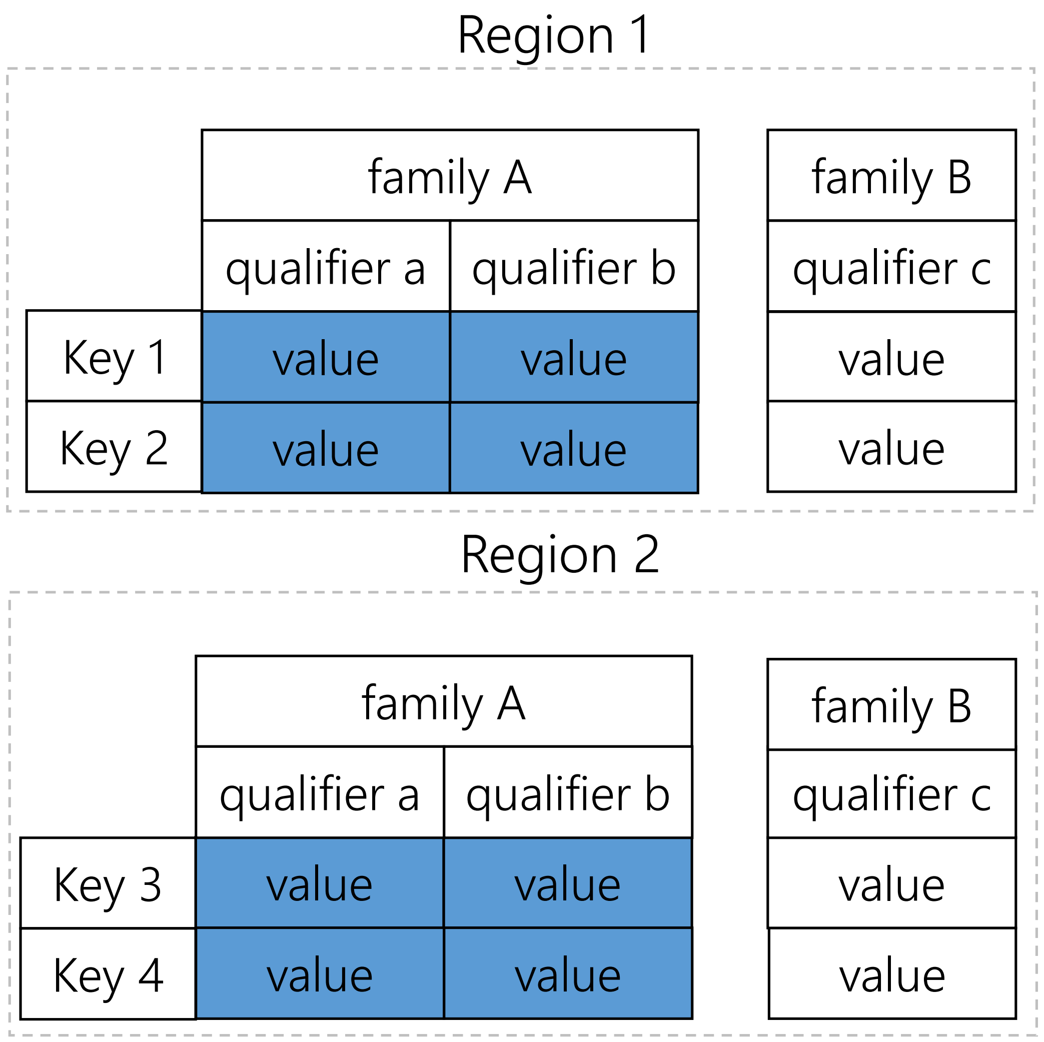
- Adding a new server is straightforward
  - It is placed in the ring and part of its successor's objects are transferred to it

Data Design

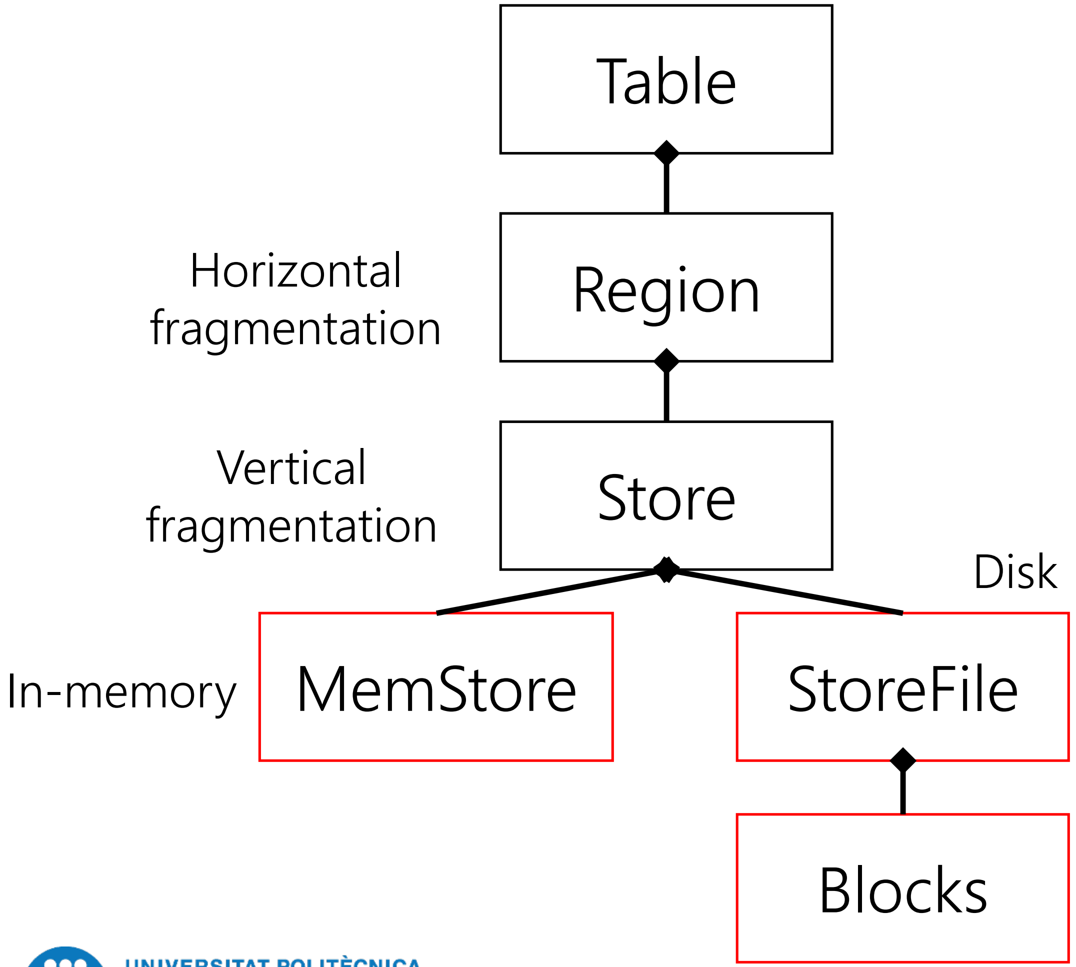
Challenge I

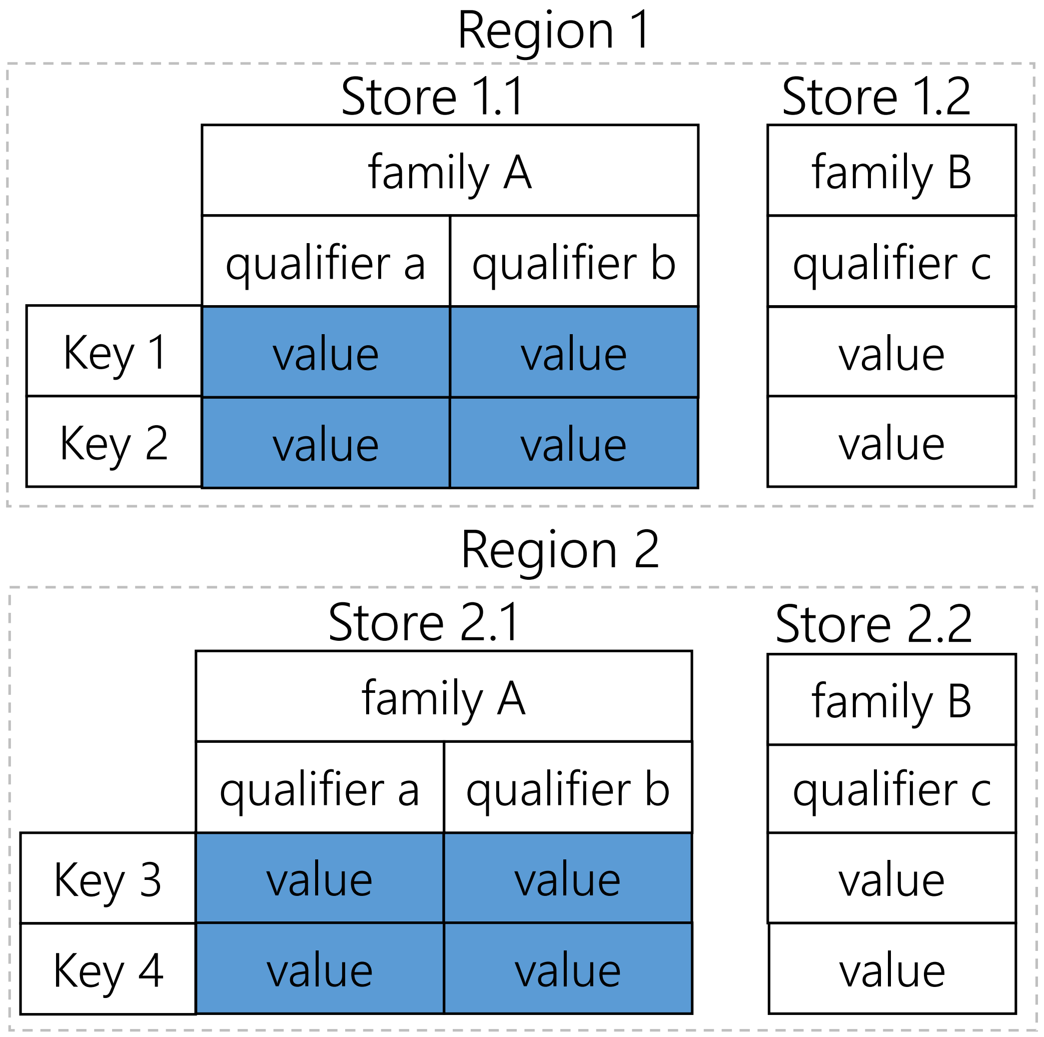
Logical structure





Physical structure

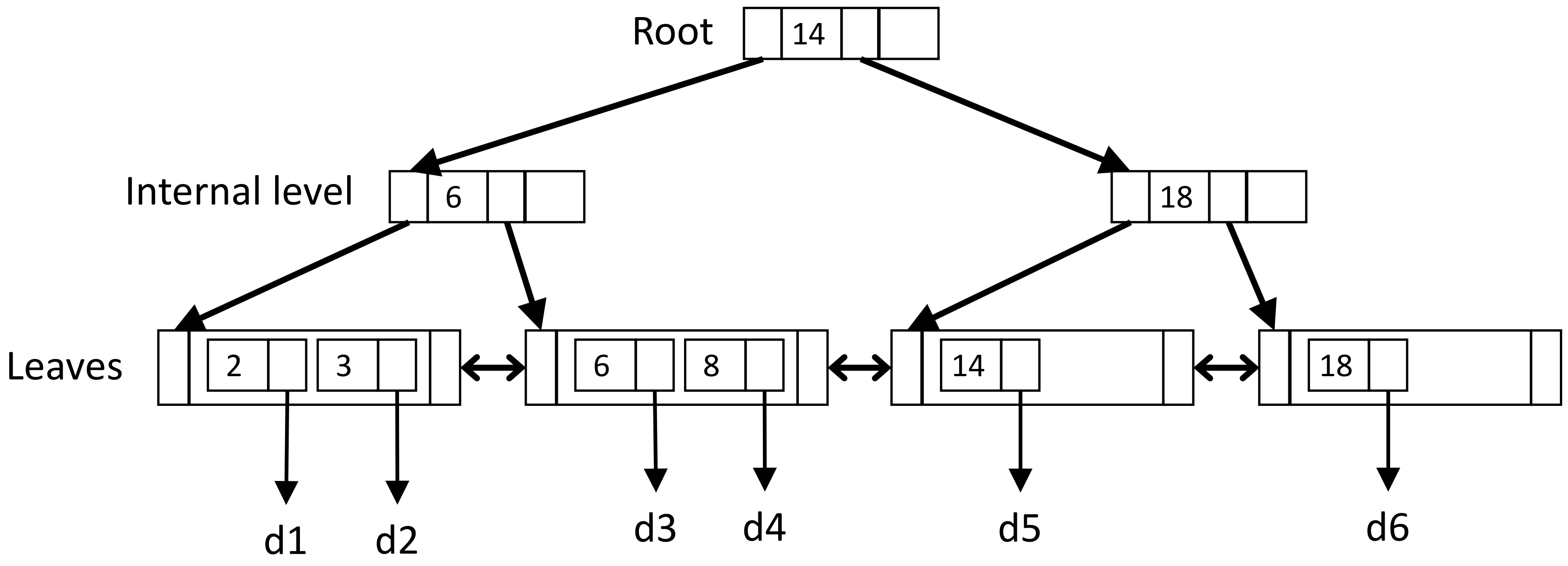




Catalog Management

Challenge II

Metadata hierarchical structure



HBase Component Roles

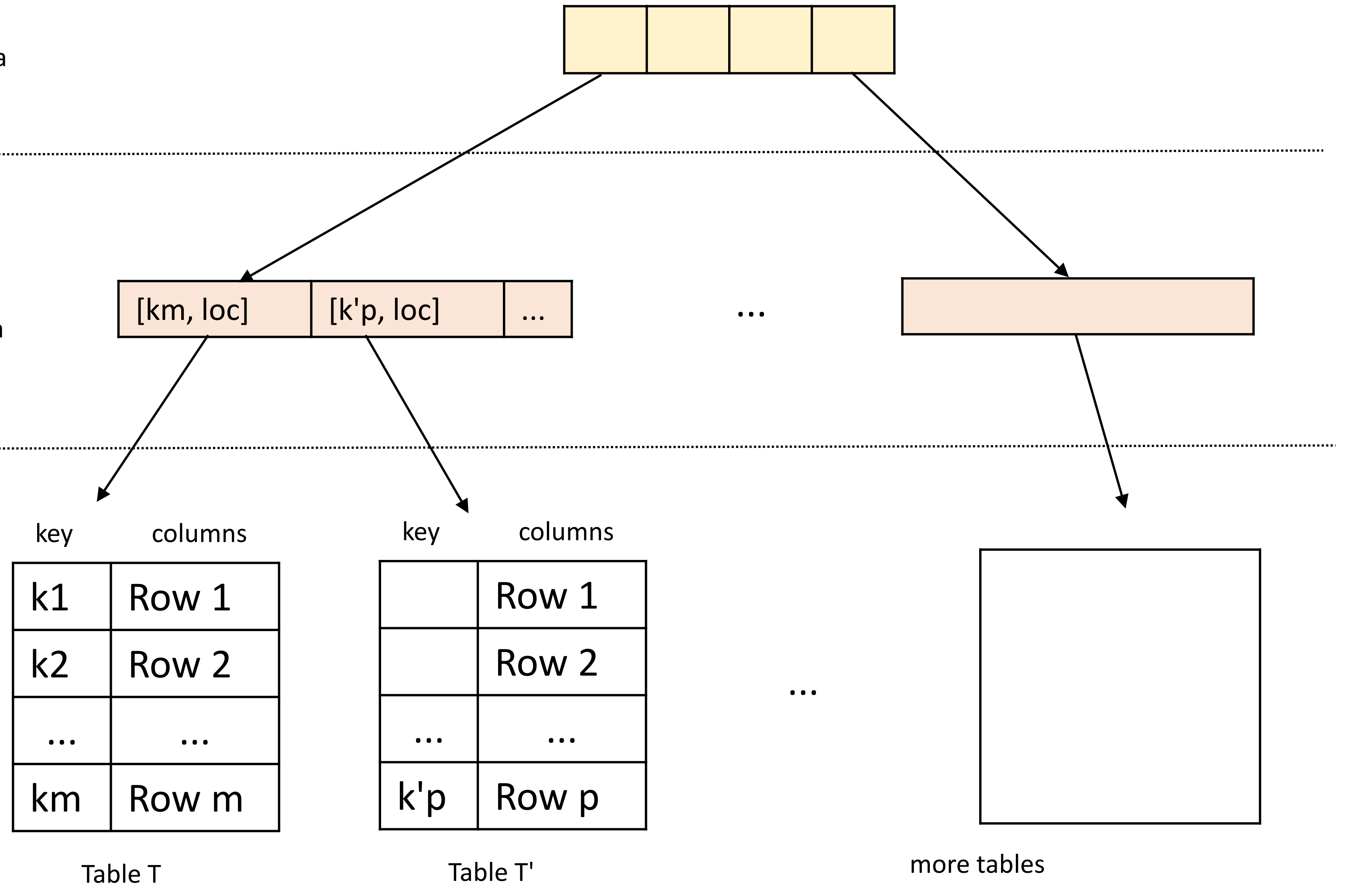
- One coordinator server
  - Maintenance of the table schemas
    - Root region
  - Monitoring of services (heartbeating)
  - Assignment of regions to servers
- Many region servers
  - Each handling around 100-1.000 regions
  - Apply concurrency and recovery techniques
  - Managing split of regions
    - 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

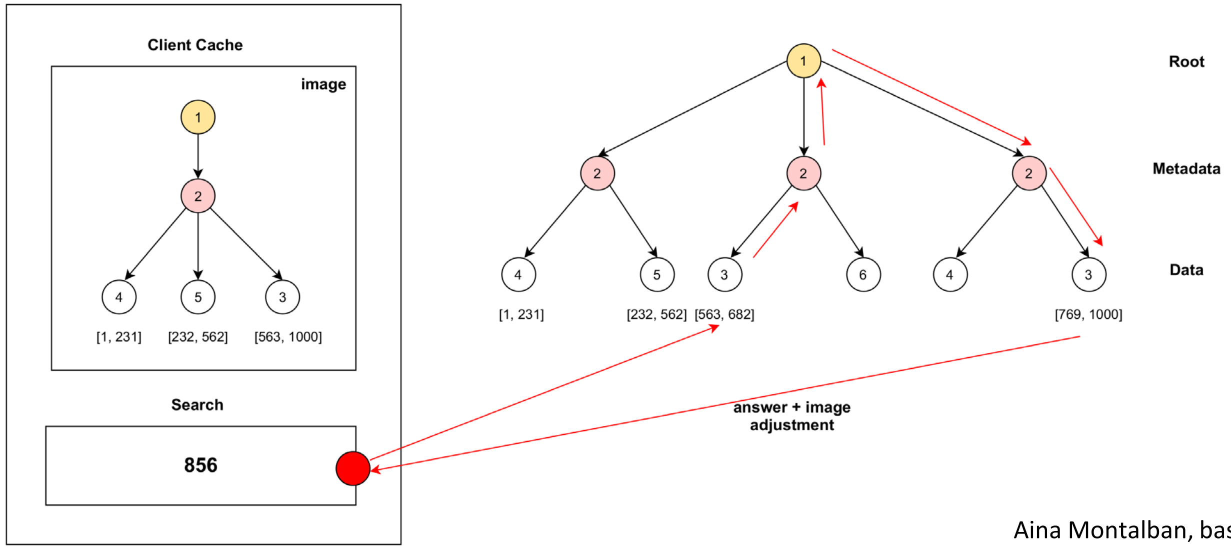
User data tables
Store data



Aina Montalban, based on S. Abiteboul et al.

Metadata synchronization in HBase

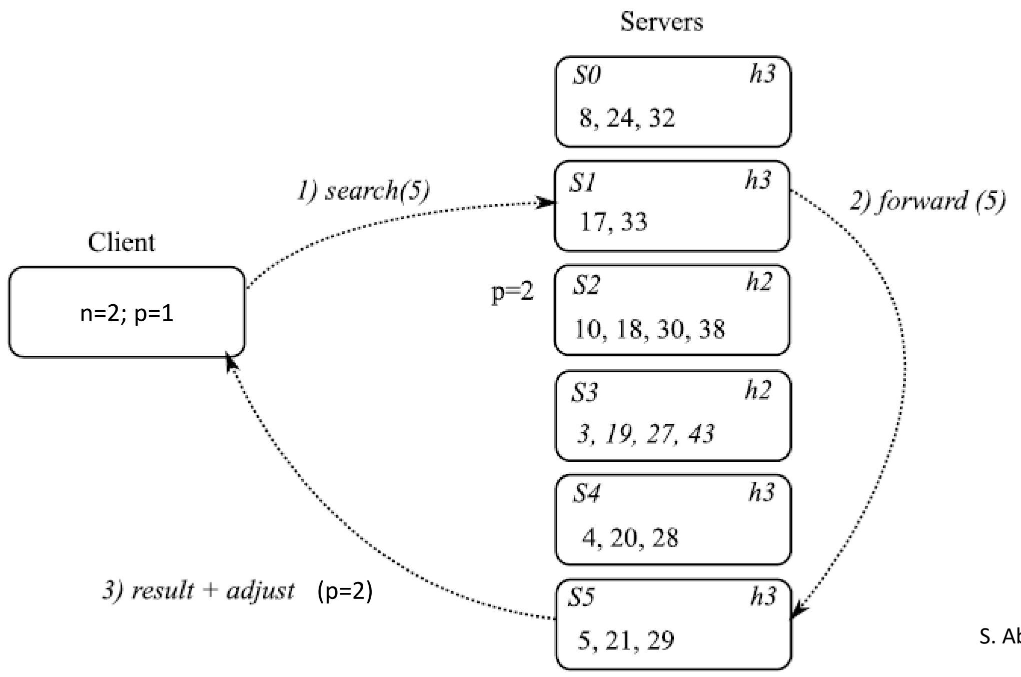
- Split and merge invalidate the cached metadata
  a) Gossiping
  b) Lazy updates
    - Discrepancies may cause out-of-range errors, which trigger a stabilization protocol (i.e., **mistake compensation**)
      - Apply forwarding path
        - If an out-of-range error is triggered, it is forwarded upward
        - 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
- 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

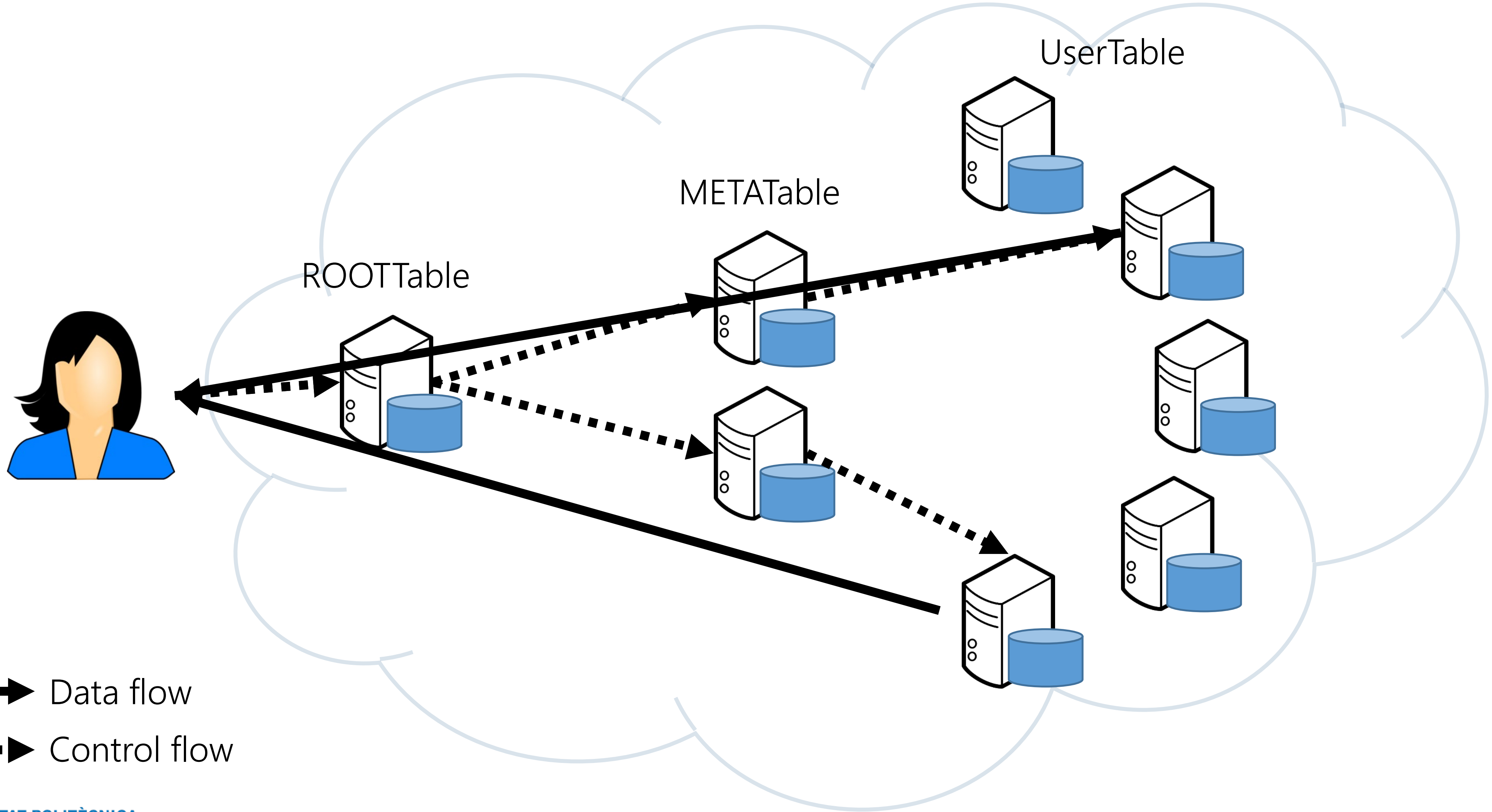
Query processing

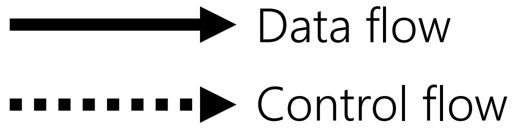
Challenge IV

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

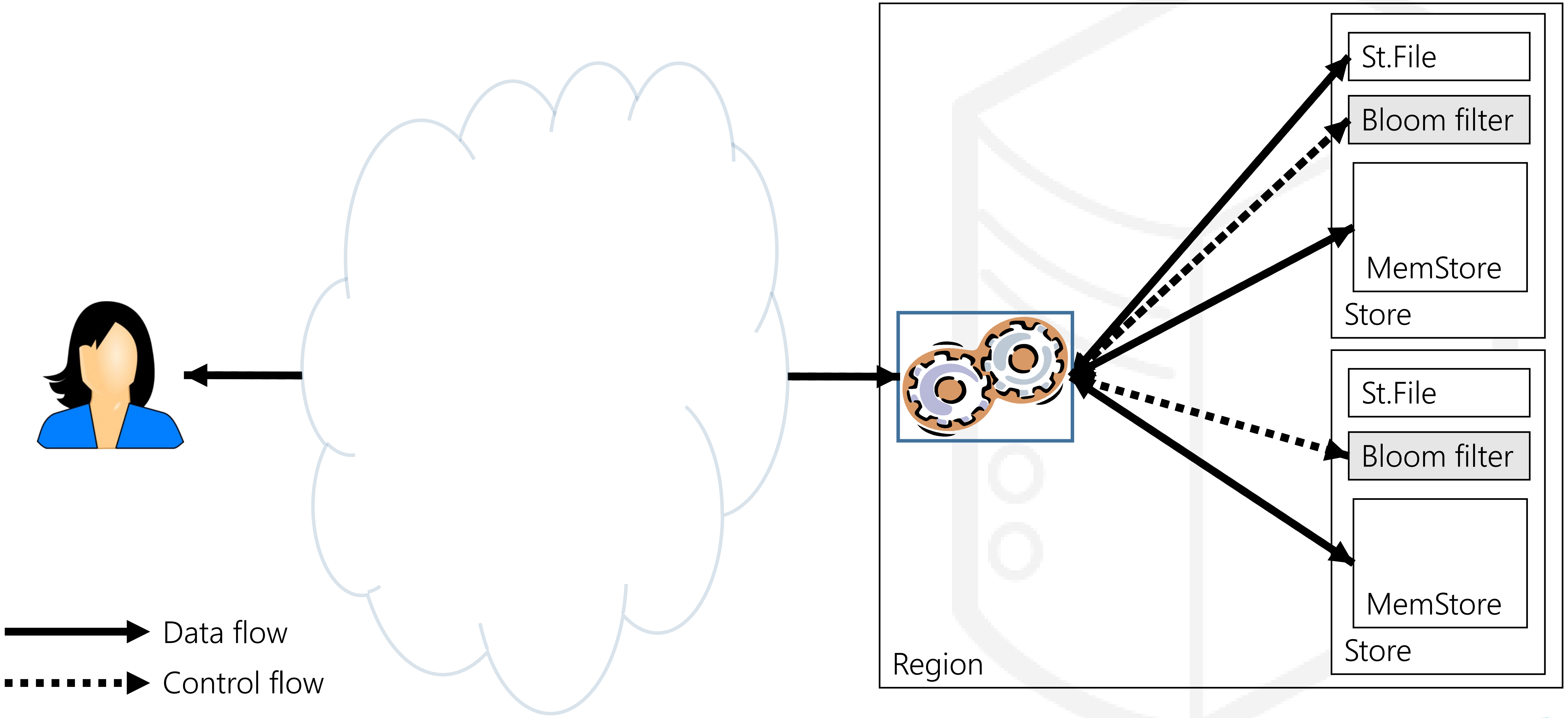
DTIM
www.essi.upc.edu/dtim

Global execution





Local execution



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