Bigtable

Big Data Systems

Dr. Rubi Boim

Bigtable

- Google’s (internal) main database

- In 2015 Google also offered it as a product

Motivation (for this course)

- First encounter with wide column database
- Understand basic usage / data model
  we will go much deeper later in the course (NoSQL data modeling)
- Understand Bigtable building blocks
  - Crucial for success in large scale systems
  - Many are used also by Cassandra

Agenda

- History
- Data model
- Building blocks
- SSTable (and memtable)
- Bloom filter
- Summary
- Extra - Chubby
- Extra - Tablet location

Agenda

- **History**
- Data model
- Building blocks
- SSTable (and memtable)
- Bloom filter
- Summary
- Extra - Chubby
- Extra - Tablet location

Bigtable

- Create by Google in 2004-2006
  paper: Bigtable: A Distributed Storage System for Structured Data
- The techniques developed here are used in many other systems
  not just by Google - HBase, Cassandra…
- One of (if not the) first NoSQL database

History

- Google was on hyper growth on 2004
- Web indexes for search engine took too long to build
- A lot of growing projects
  Google Search (Personalized)
  Google Earth
  Google Finance
  Google Analytics
  …
  (later on also used in gmail, maps, YouTube and many many more)

Initial requirements

Remember this was in 2004…

- Access / manage petabytes of data in real time
- Variable data size
  URLs, documents, satellite imagery…
- Wide applicability
- Highly scalable
- Highly available
- Highly compressible

Initial requirements - Data model

- Big table does NOT supports full relational model
- Simple custom API instead

Agenda

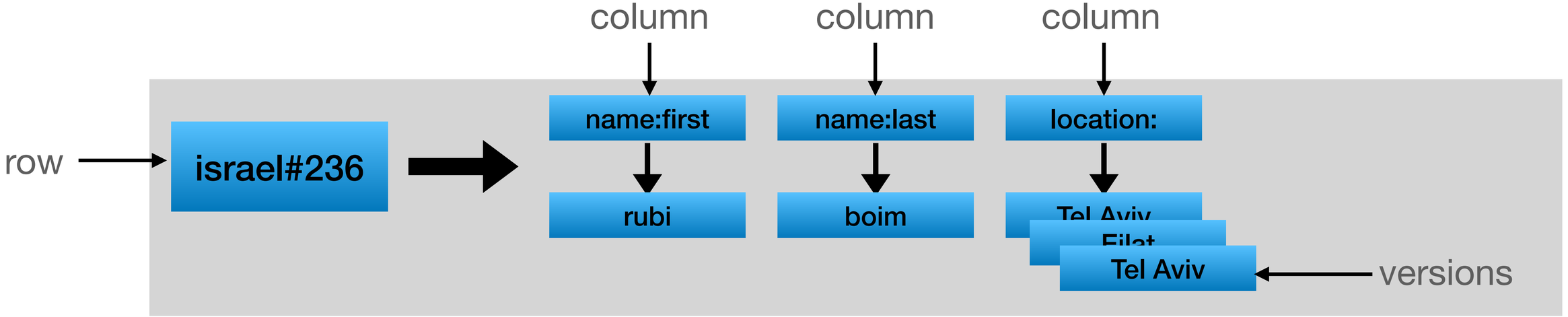
- History
- **Data model**
- Building blocks
- SSTable (and memtable)
- Bloom filter
- Summary
- Extra - Chubby
- Extra - Tablet location

Data model - TLDR

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

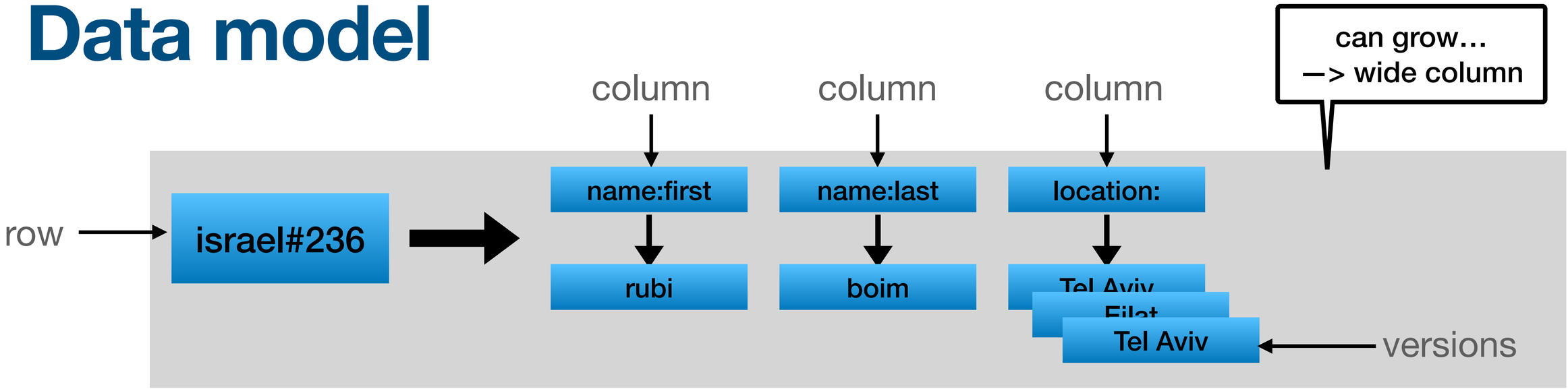
- The map is indexed by
  - Row key
  - Column key
  - Timestamp

Data model



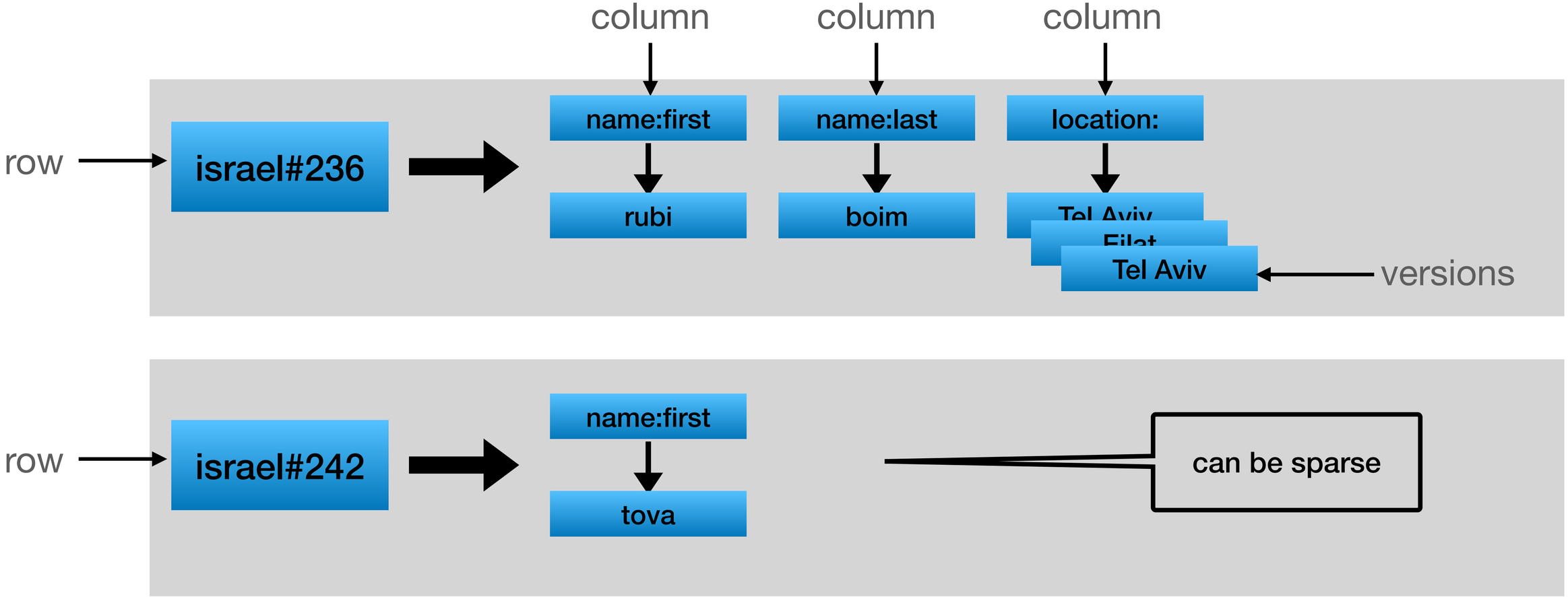
<row:string, column:string, timestamp:int64> —> string

Data model



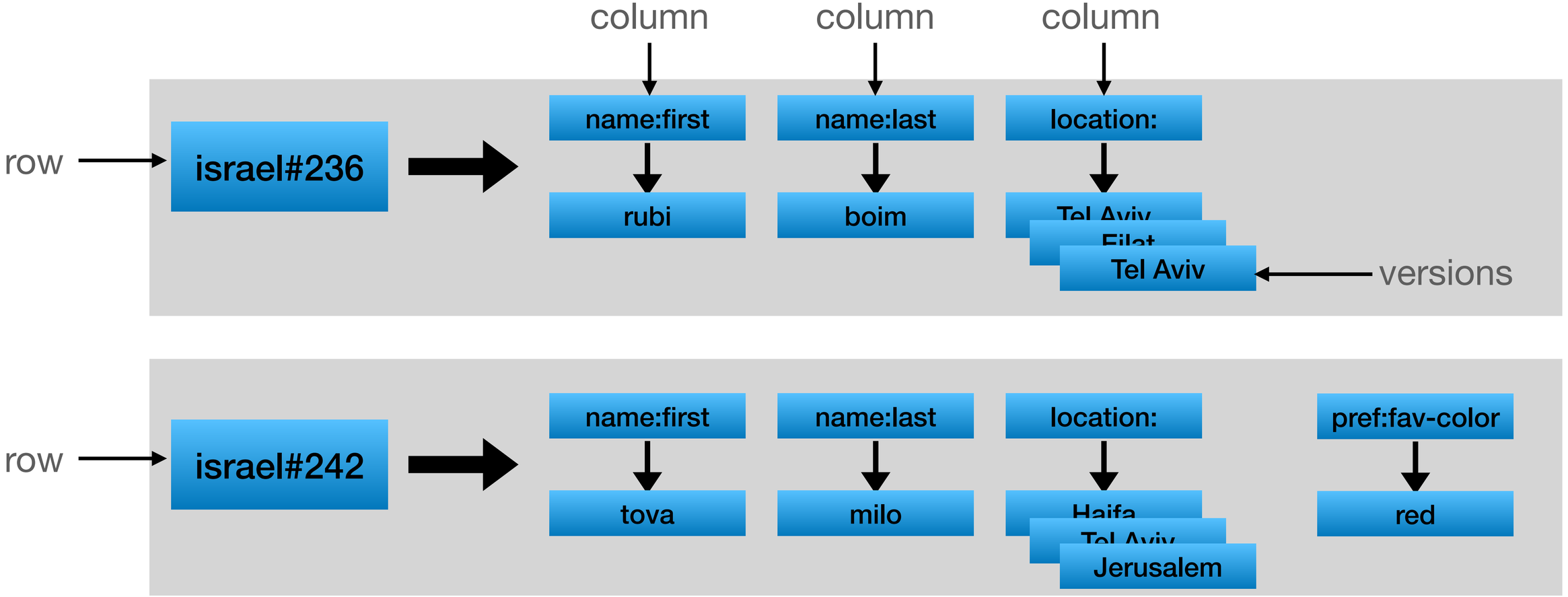
<row:string, column:string, timestamp:int64> —> string

Data model



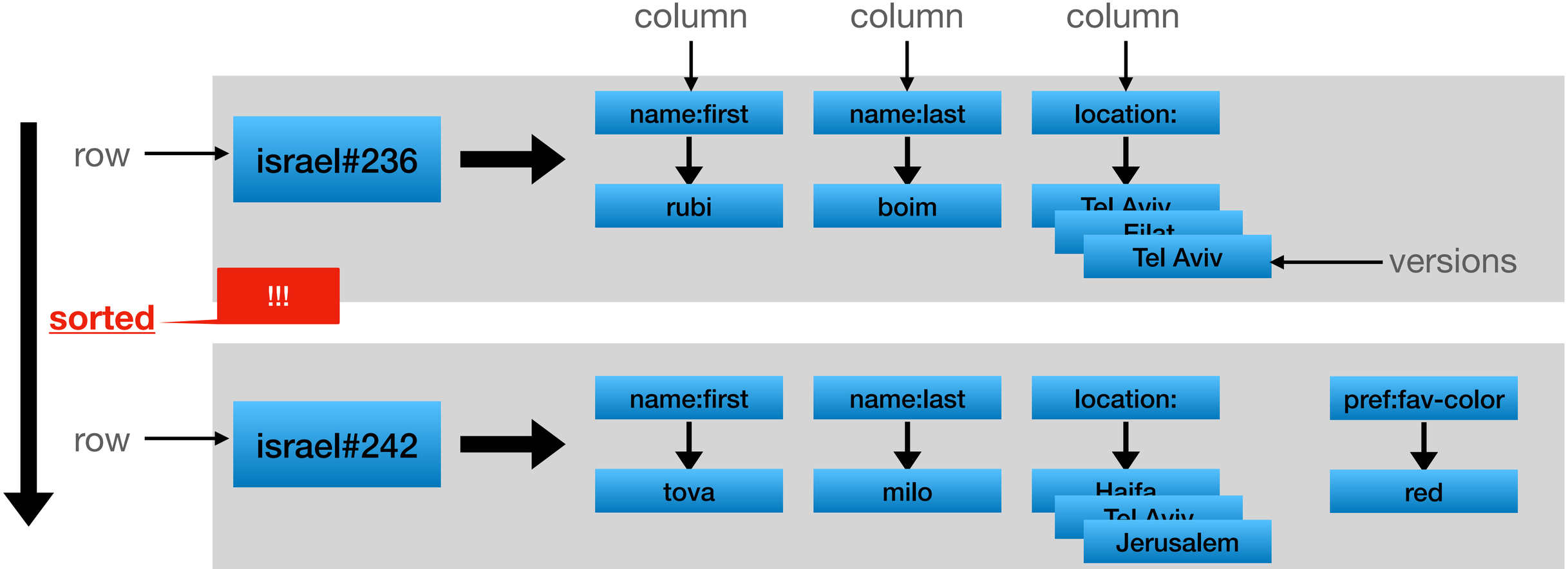
<row:string, column:string, timestamp:int64> —> string

Data model



<row:string, column:string, timestamp:int64> —> string

Data model

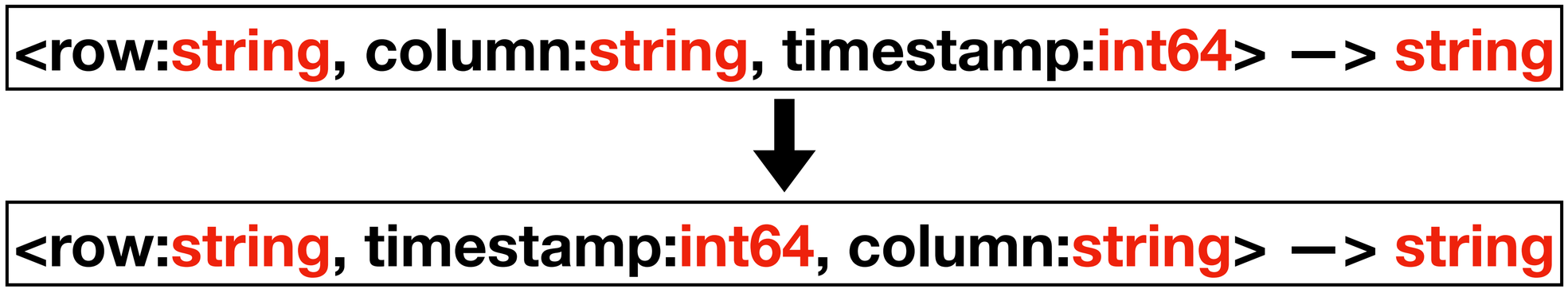


<row:string, column:string, timestamp:int64> —> string

Data model - design

- Discussion - is this model optimal?

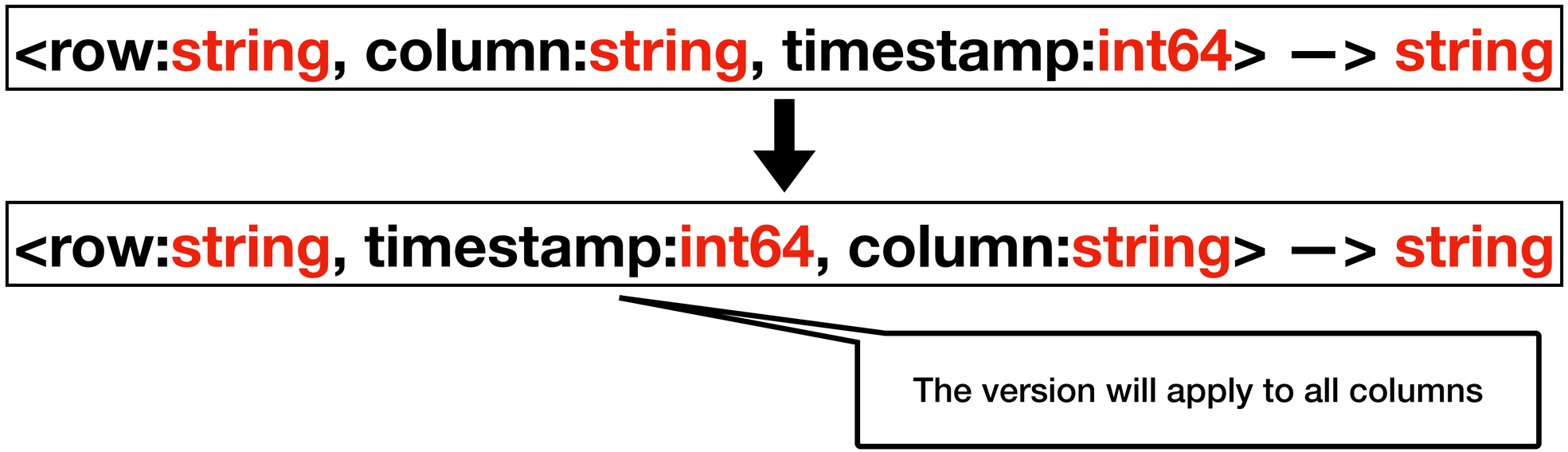
- What will happen if we switch the order?



Data model - design

- Discussion - is this model optimal?

- What will happen if we switch the order?



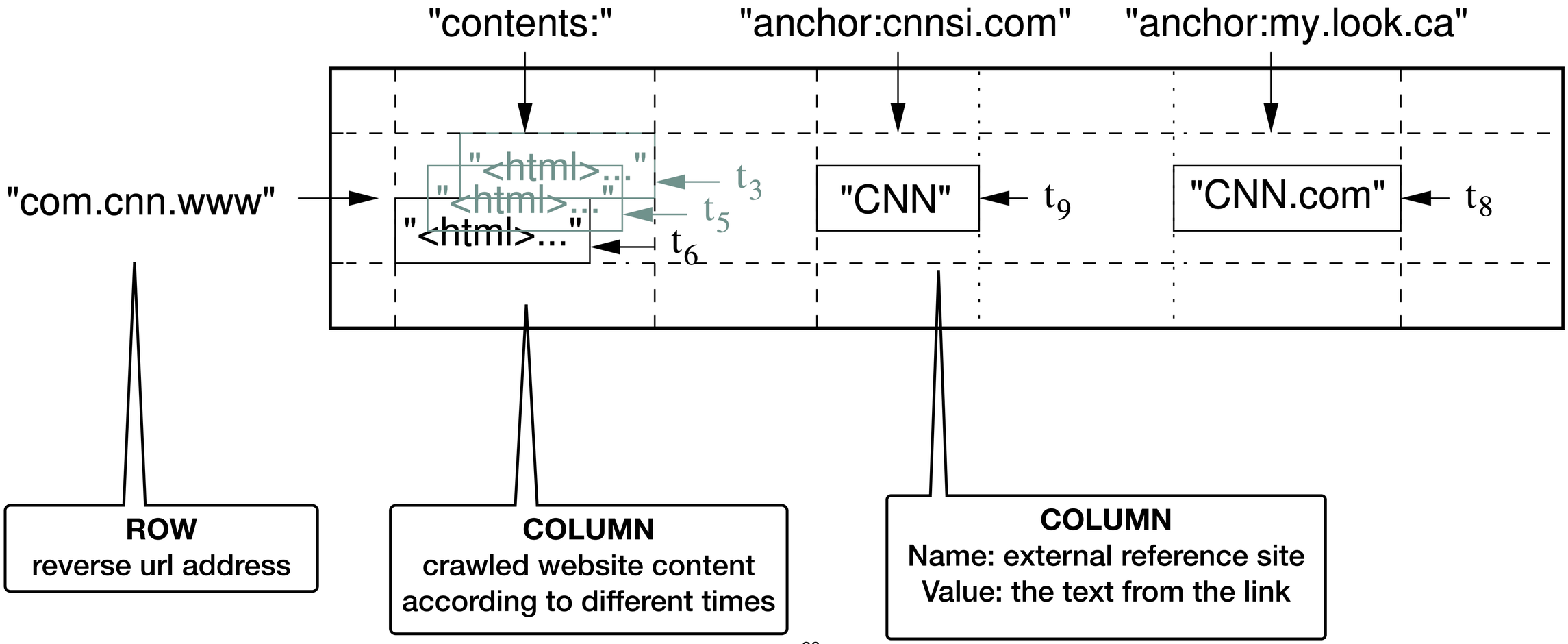
Data model - Google’s requirements

- Bigtable is build by Google **FOR Google**…

- Optimal == Optimal for Google’s requirements

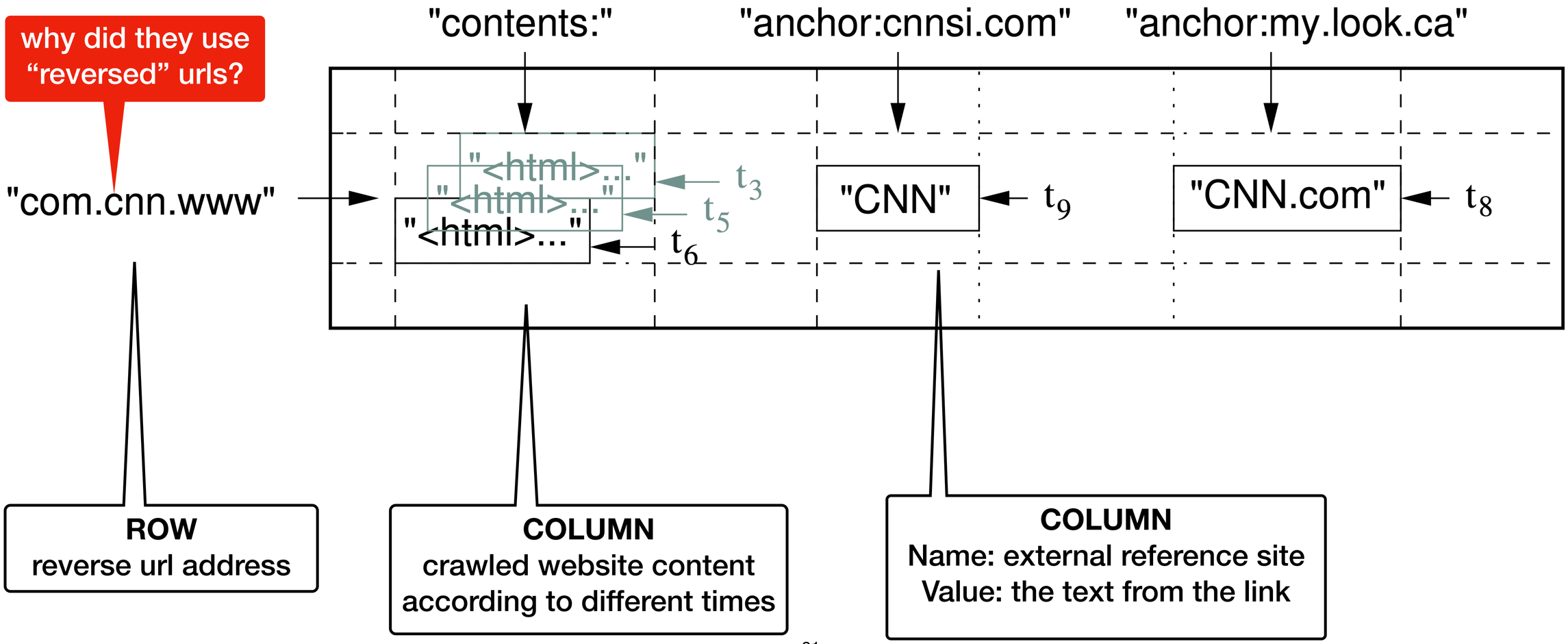
Data model - Webtable example

Used by Google's search index



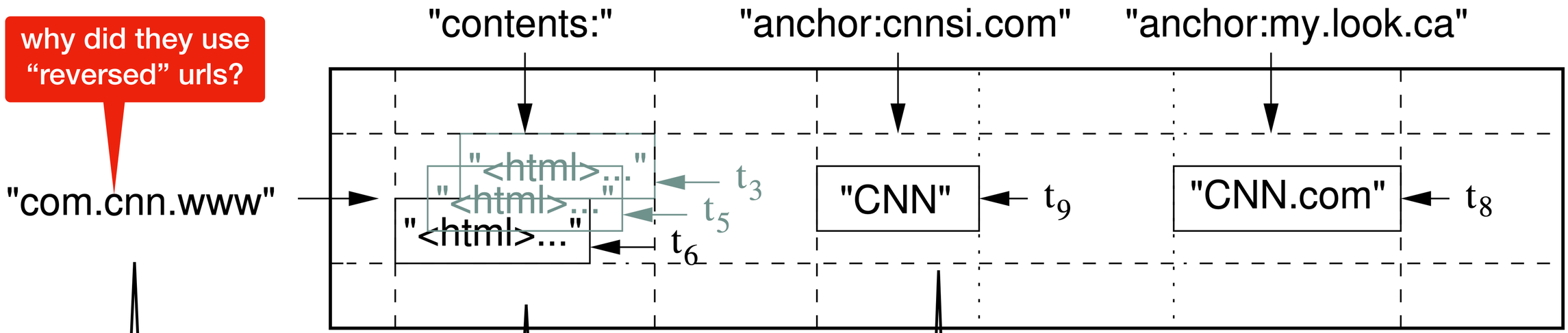
Data model - Webtable example

Used by Google's search index



Data model - Webtable example

Used by Google's search index



ROW
reverse url address

COLUMN
crawled website content
according to different times

COLUMN
Name: external reference site
Value: the text from the link

Rows

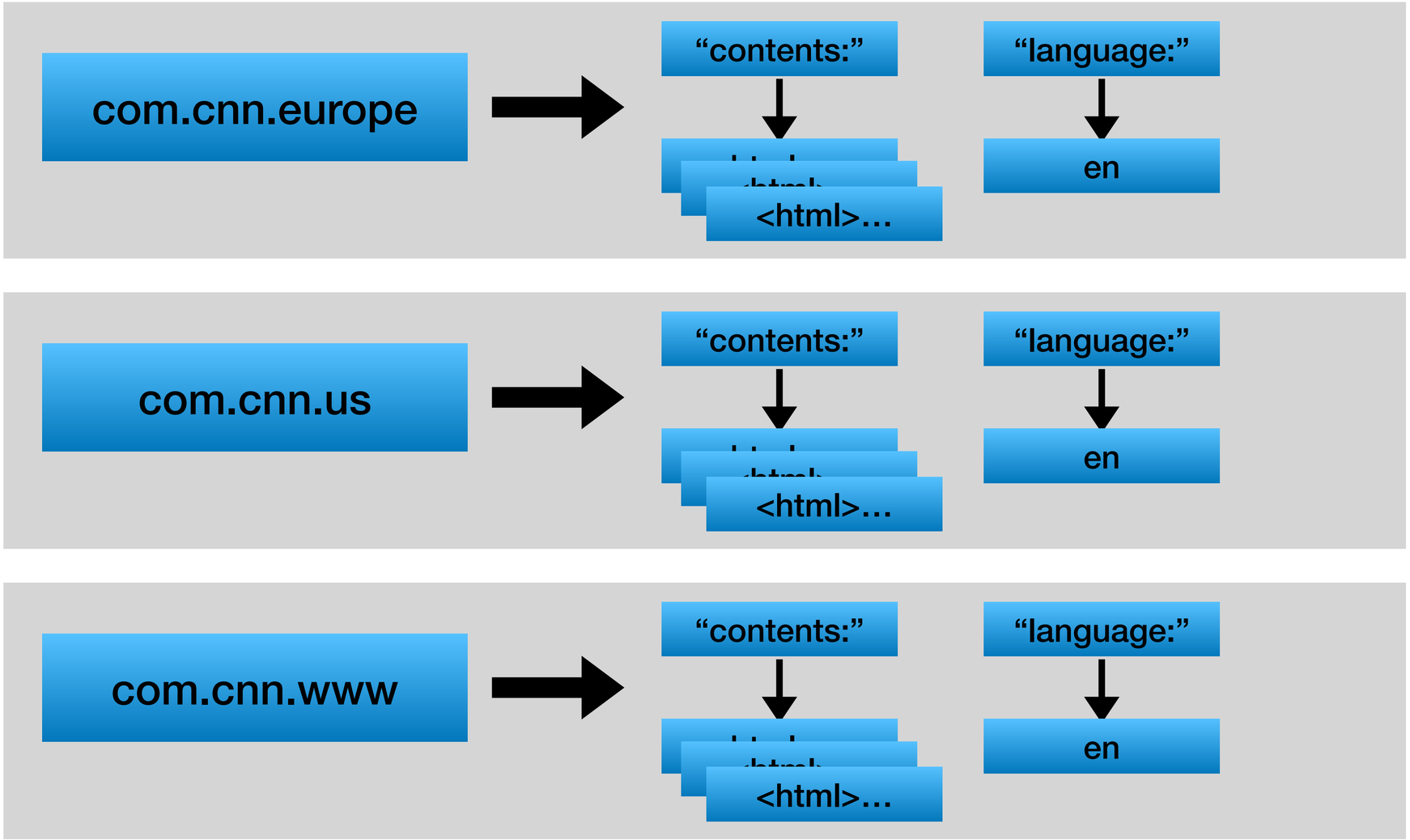
- Row key is up to 64KB (usually 10-100 bytes)

- Every read/write of data under a single row is atomic
  regardless to the number of columns read/written

- Stored by lexicographic order of row key
  —> read of short rows are efficient
  (can be on the same server)
  more on tablets later on

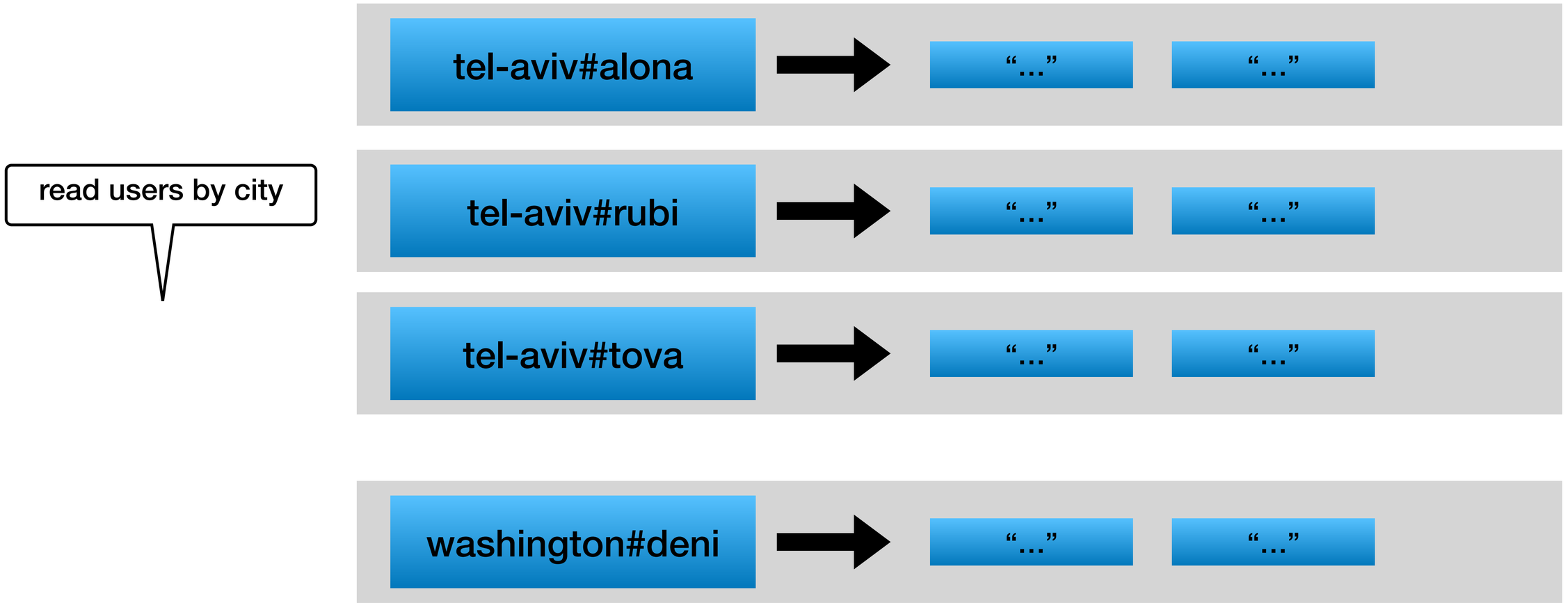
Rows - locality exploit

Model the data based on how data is accessed



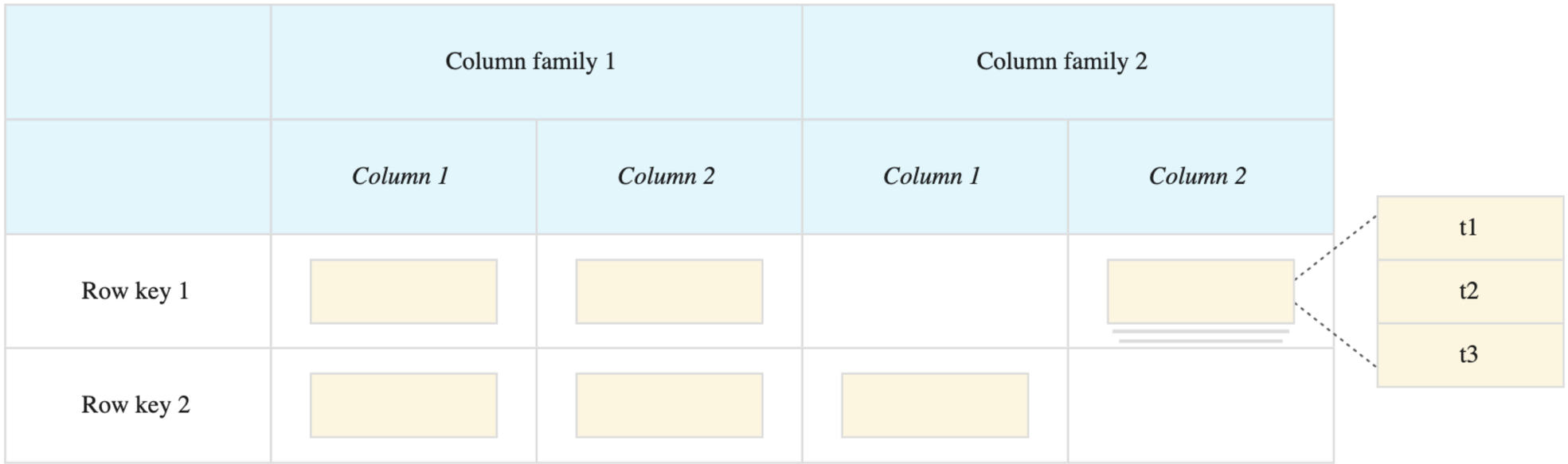
Rows - range

- (“short”) Rows can be read together/sequentially



Column Family / Columns

- Column family - group of column
  usually of the same type for compression
- Column name - `family:qualifier`

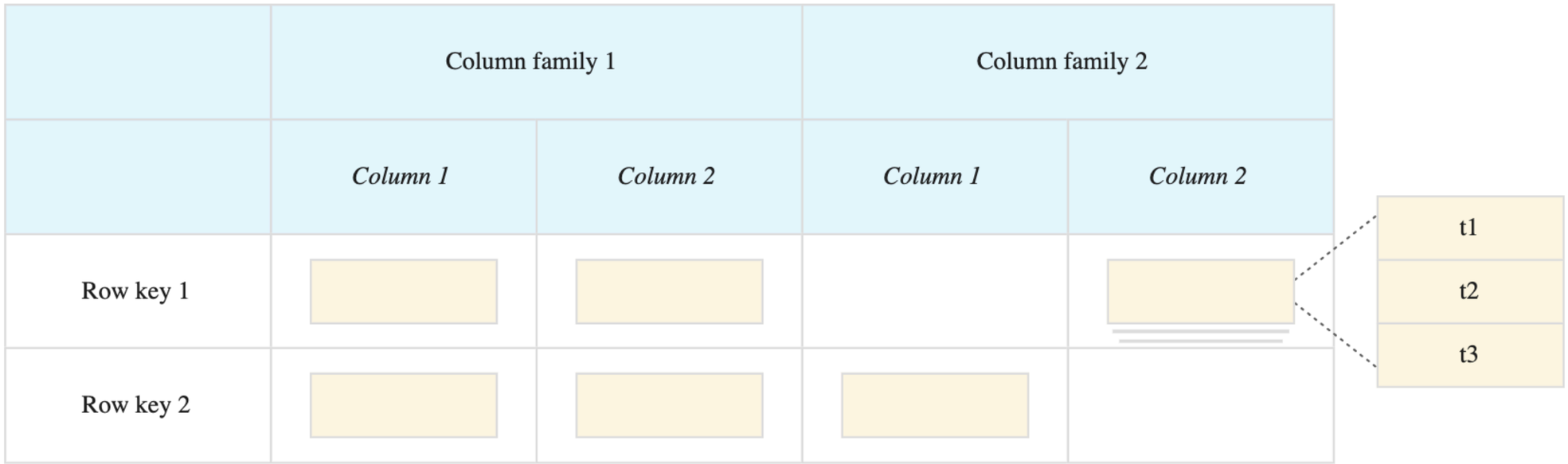


Column Family / Columns

- Column family - group of column
  usually of the same type for compression
- Column name - `family:qualifier`

Not too much (up to ~100) families

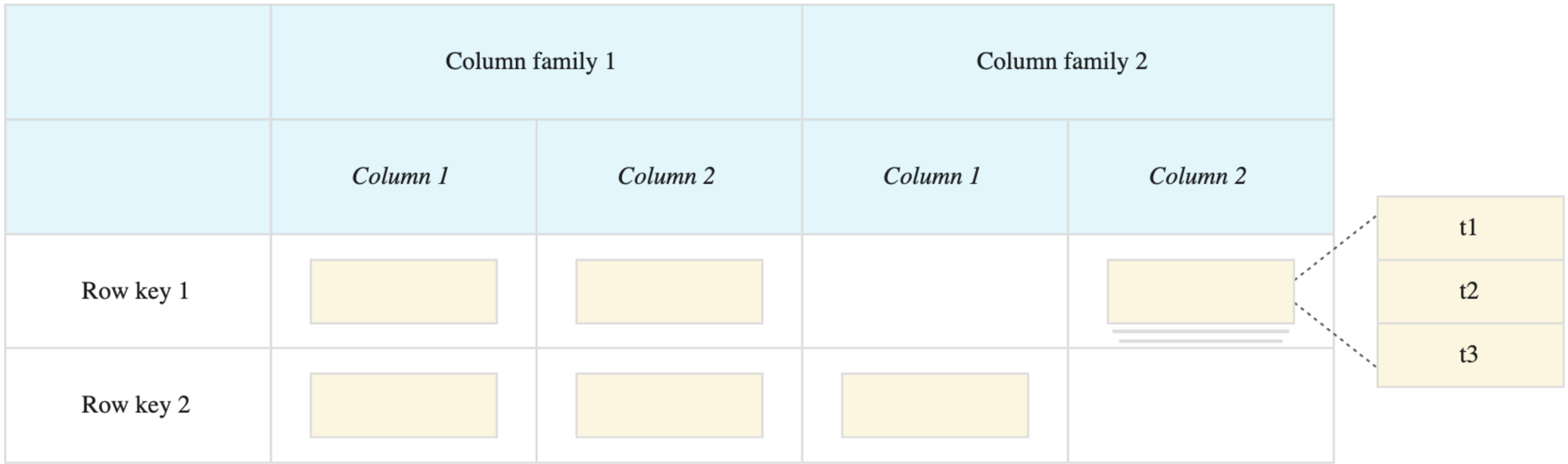
“unlimited” columns



Column Family / Columns

- Column family - group of column
  usually of the same type for compression
- Column name - `family:qualifier`

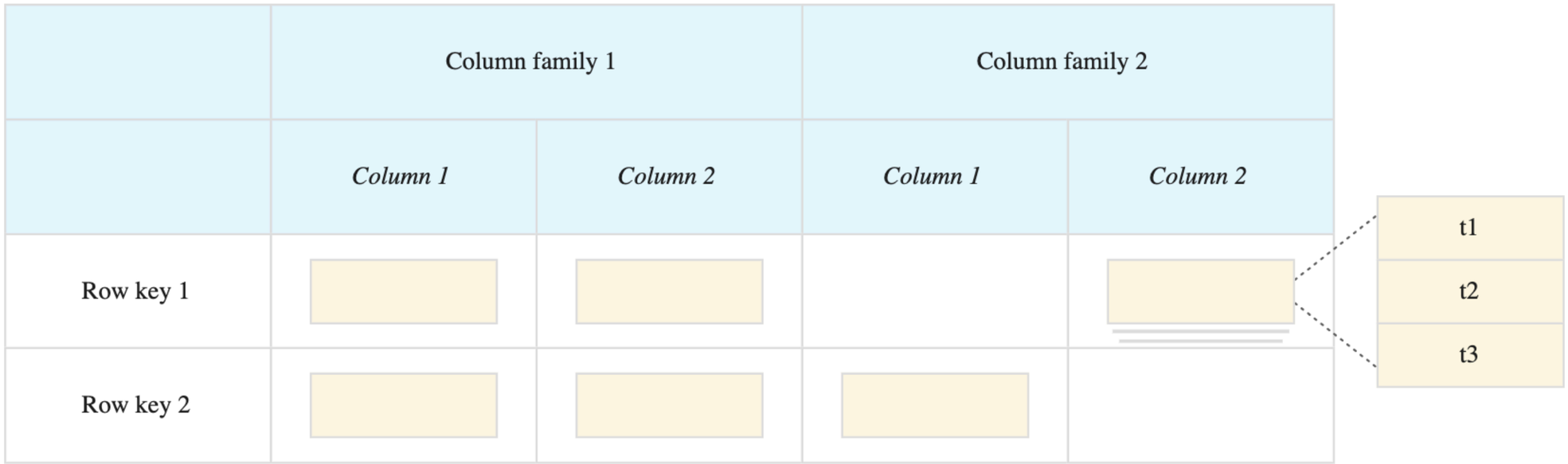
Access control per column family



Column Family / Columns

- Column family - group of column
  usually of the same type for compression
- Column name - `family:qualifier`

NOTE - we can store data in the qualifier

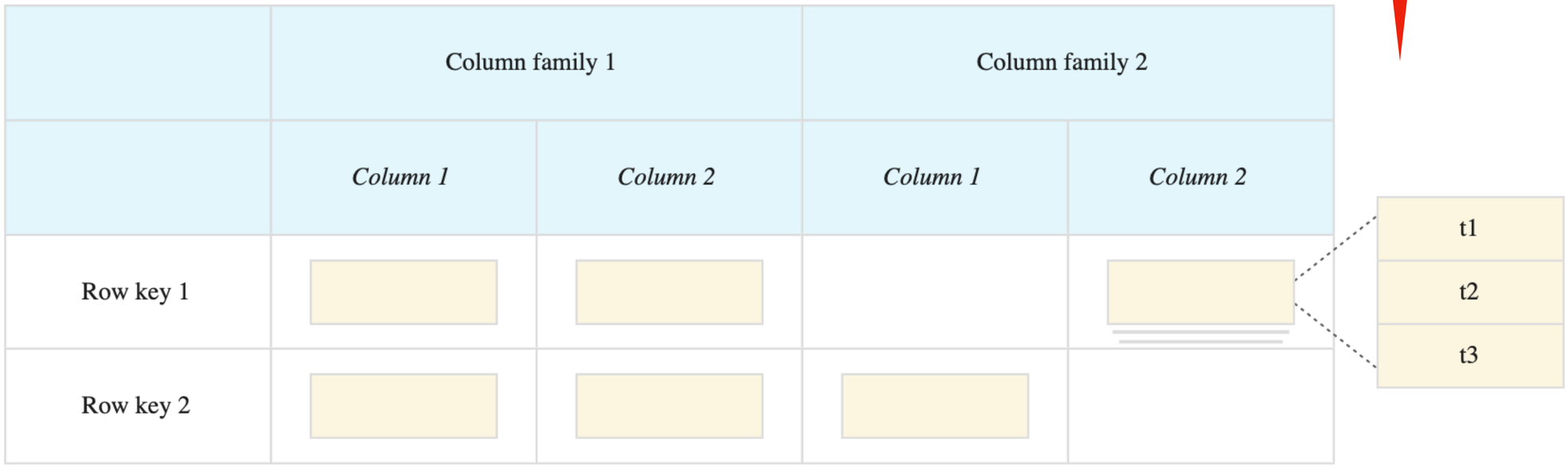


Column Family / Columns

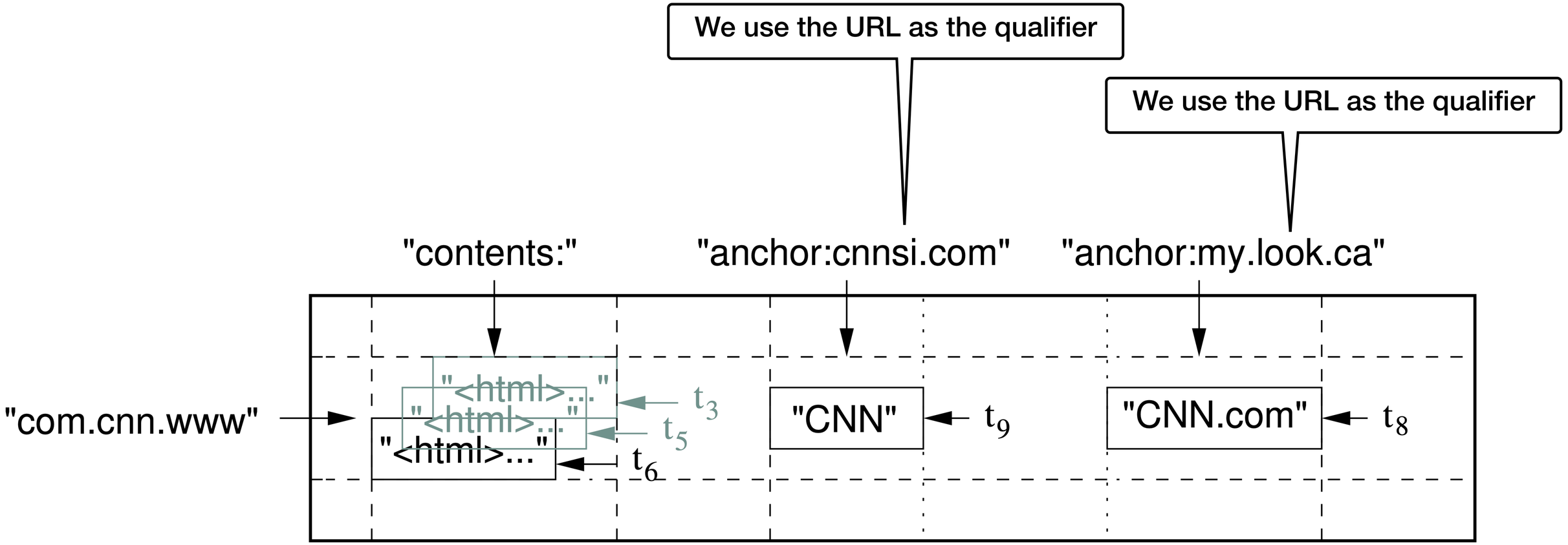
- Column family - group of column
  usually of the same type for compression
- Column name - `family:qualifier`

columns are sorted within each column family

column families are NOT sorted between other column families



Column Family / Columns



Timestamp

- Used to store different version of the same cell
  optional - current time is used if not passed

- For reads:
  - return all versions
  - return top k recent versions
  - return all versions between timestamps

- Automatic “garbage collect”
  - save only top k versions
  - save only versions in the past 7 days

Bigtable API

- It is not SQL
- Basic management / data manipulation
- BUT also support querying range of rows
- RTFM… ;-)

Big

- It i
- Ba
- BU
- RT

Databases

Bigtable transforms the developer experience with SQL support

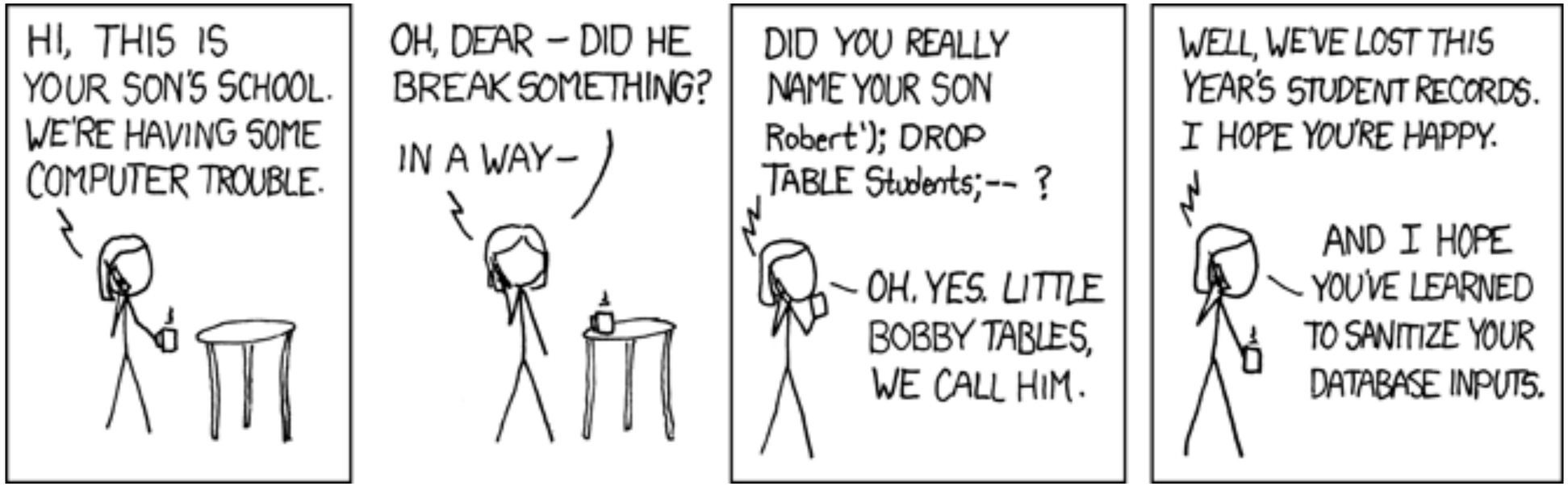
August 3, 2024

Christopher Crosbie
Group Product Manager, Google

Gary Elliott
Engineering Manager, Bigtable

Bigtable is a fast, flexible, NoSQL database that powers core Google services such as Search, Ads, and YouTube, as well as critical applications for customers such as PLAID and Mercari. Today, we're announcing Bigtable support for GoogleSQL, an ANSI-compliant SQL dialect used by Google products such as Spanner and BigQuery. Now you can use the same SQL with Bigtable to write applications for AI, fraud detection, data mesh, recommendations, or any other application that would benefit from real-time data.

Speaking about API/SQL



https://xkcd.com/327/

Agenda

- History
- Data model
- **Building blocks**
- SSTable (and memtable)
- Bloom filter
- Summary
- Extra - Chubby
- Extra - Tablet location

Bigtable Building blocks

- How to manage rows across servers?
- How to manage servers?
- How to manage replication?
- How to manage actual data?

Tablet

- A range of rows is called a **tablet**
- Data is stored on special files - **SSTables** (later on this)
- A set of SSTables and a range comprise a tablet

38

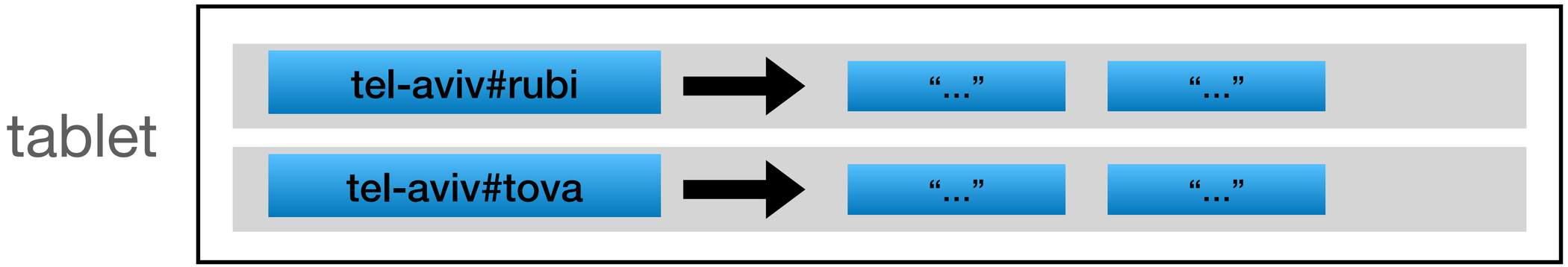
Tablet - initialize

- When a table is created, there is 1 empty tablet

39

Tablet - initialize

- When a table is created, there is 1 empty tablet

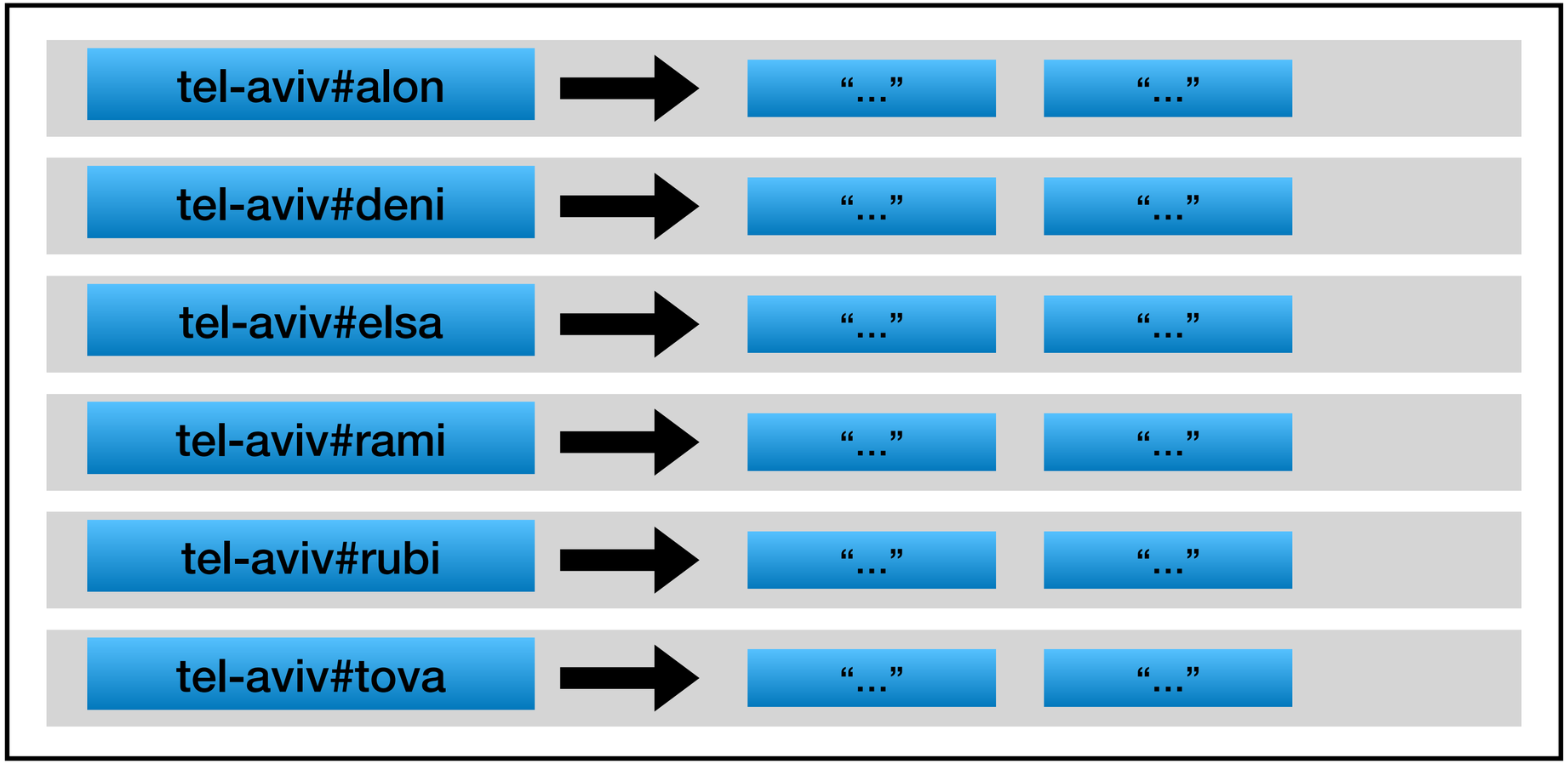


40

Tablet - Split

Approximate size: 100-200MB per tablet (default)

- When the table grows, the tablet is split



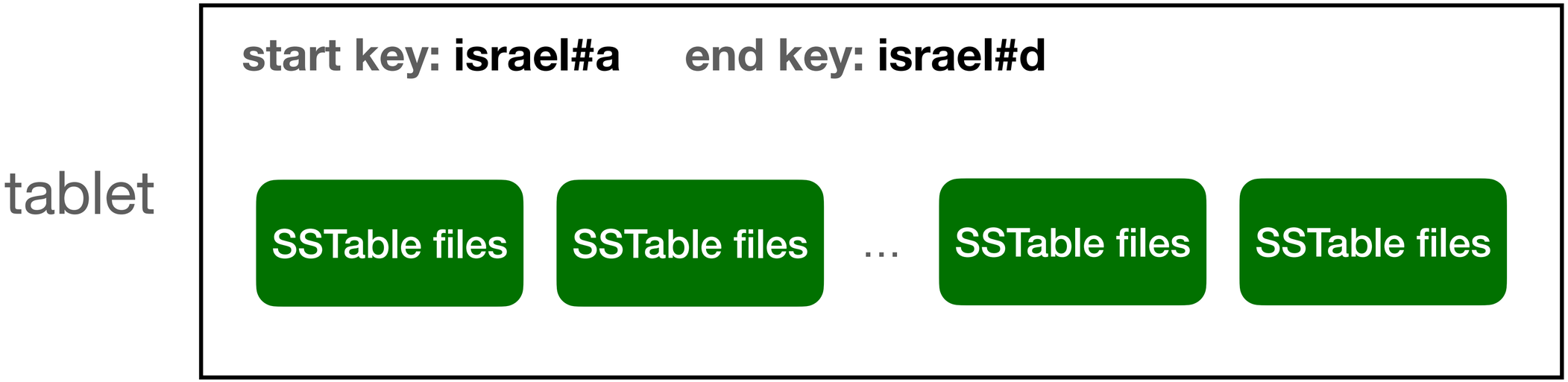
Tablet - Split

Approximate size: 100-200MB per tablet (default)

- When the table grows, the tablet is split

Tablet - components

- SSTable - the files that stored the tablet's data
  more on this later
- A set of SSTables over a matching range comprise a tablet



Tablet - mapping

- Each tablet is **assigned** to a **single node**
  also known as “Bigtable node” / “tablet server”

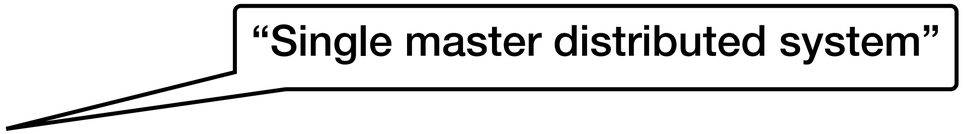
- But what is a Bigtable node???

Bigtable design by components

- Bigtable is built on several different layers
  - Management
  - Processing
  - Storage

Bigtable design by components

- **Management - Master node (Cubby)**
  - Manage Bigtable nodes
  - Manage Data mapping (tablets —> nodes)



- **Processing - Bigtable nodes**
  - Manage read/writes (without actual storage)

- **Storage - GFS / Colossus (Google File System)**
  - Manage actual storage files (SSTables)

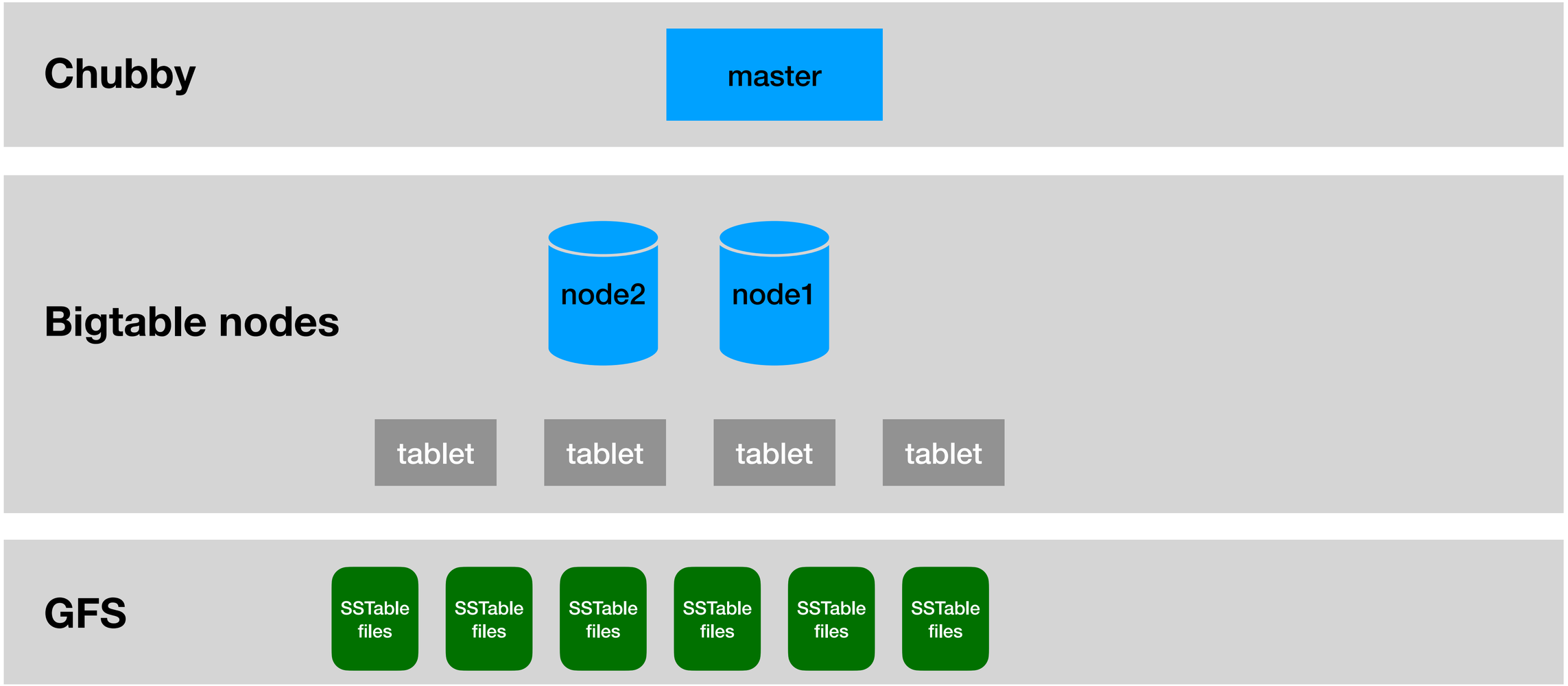
Bigtable design by components

- **Management - Master node (Cubby)**
  - Manage Bigtable nodes
  - Manage Data mapping (tablets —> nodes)
- **Processing - Bigtable nodes**
  - Manage read/writes (without actual storage)
- **Storage - GFS / Colossus (Google File System)**
  - Manage actual storage files (SSTables)

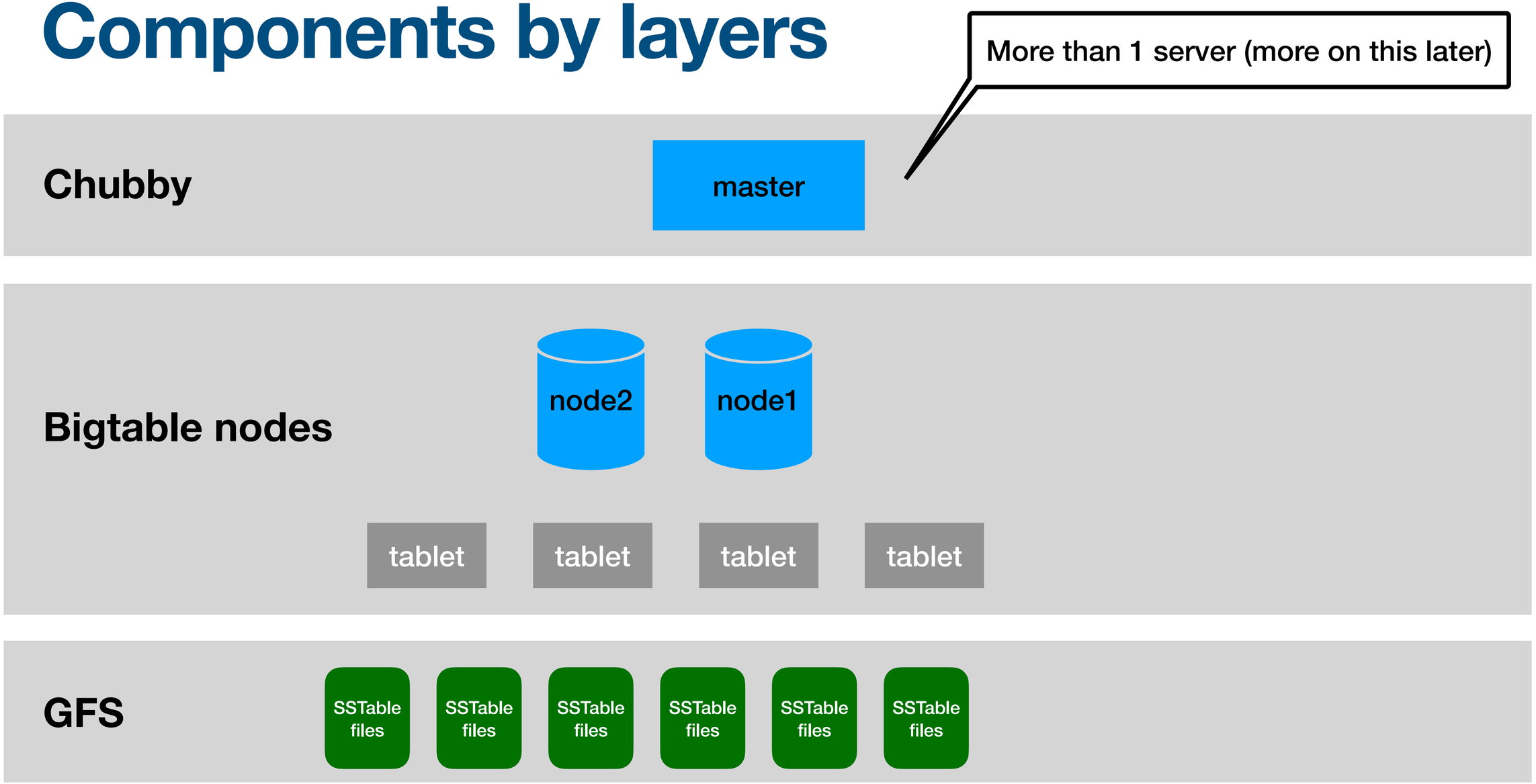
In Dynamo / Cassandra
each node handles everything

This is a BIG difference

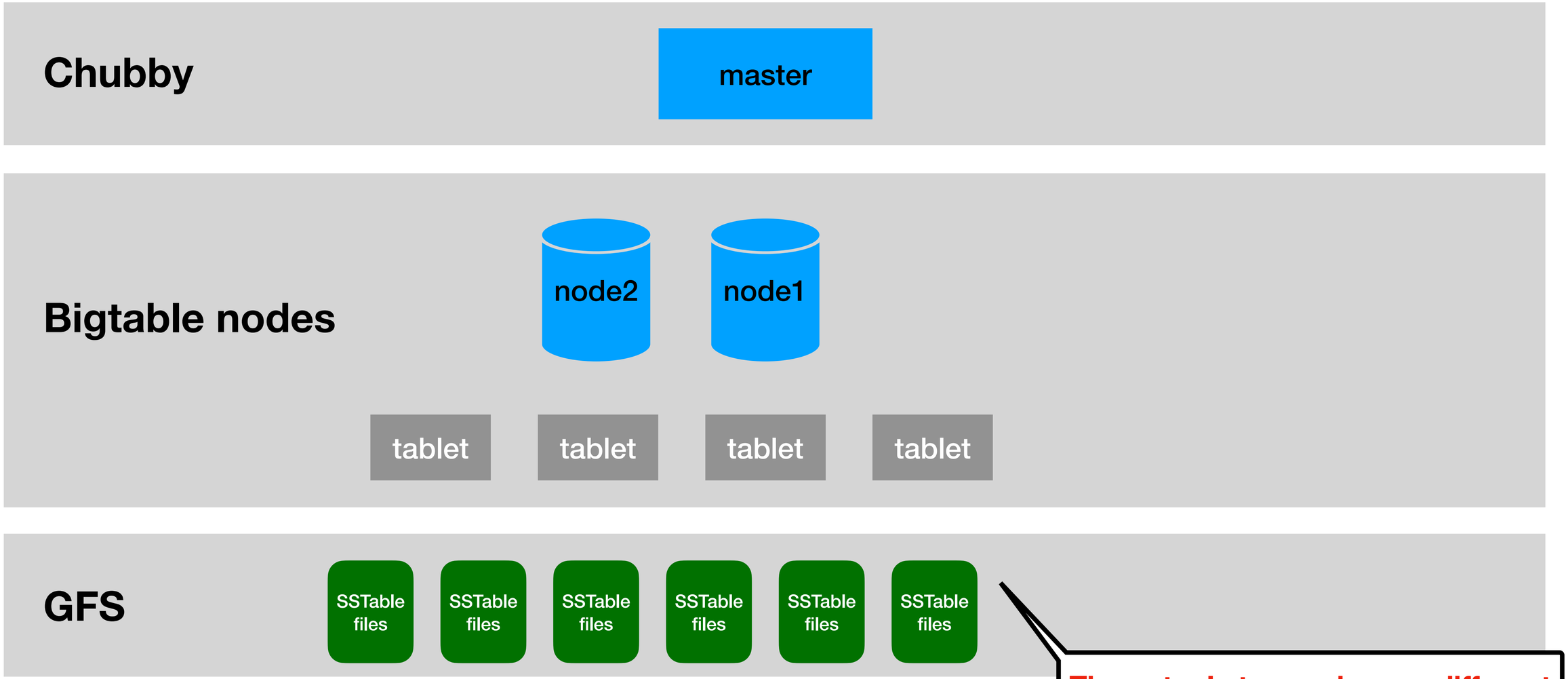
Components by layers



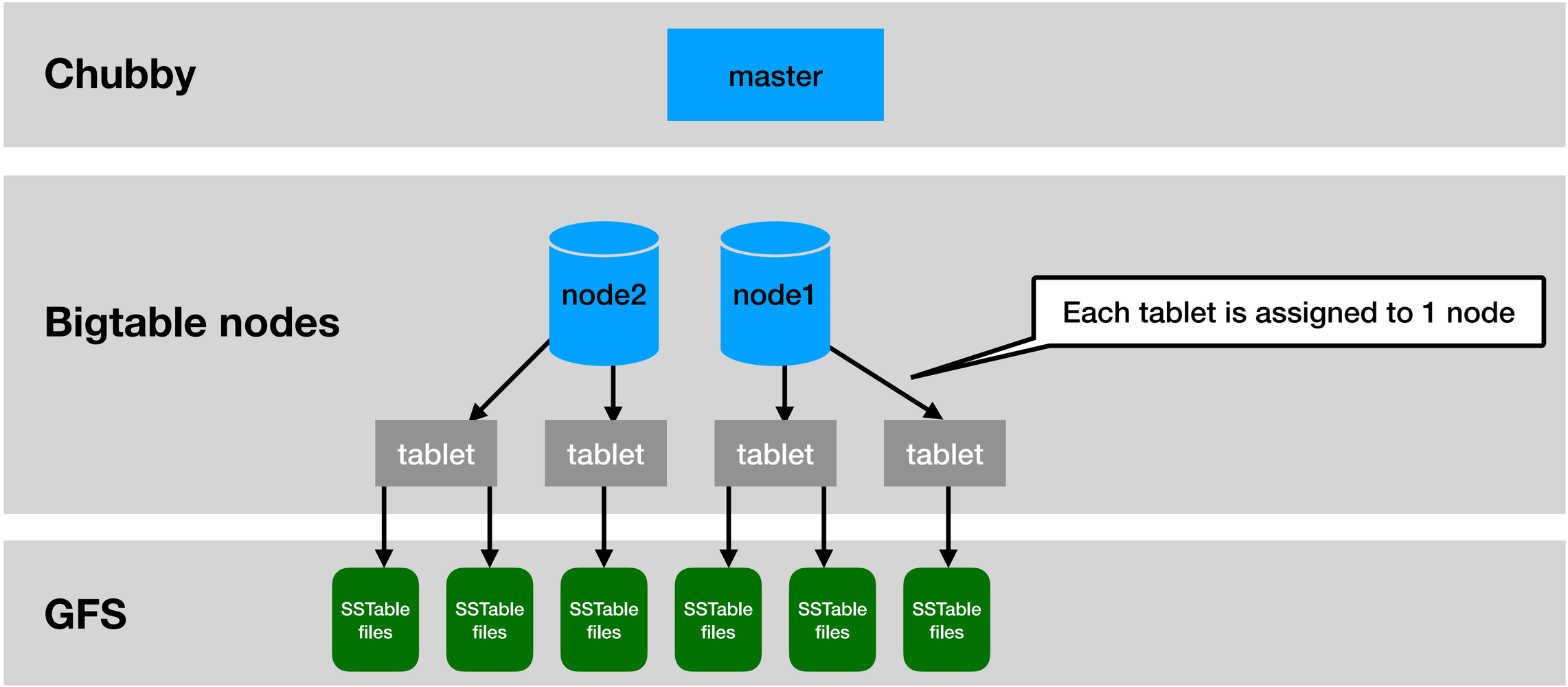
Components by layers



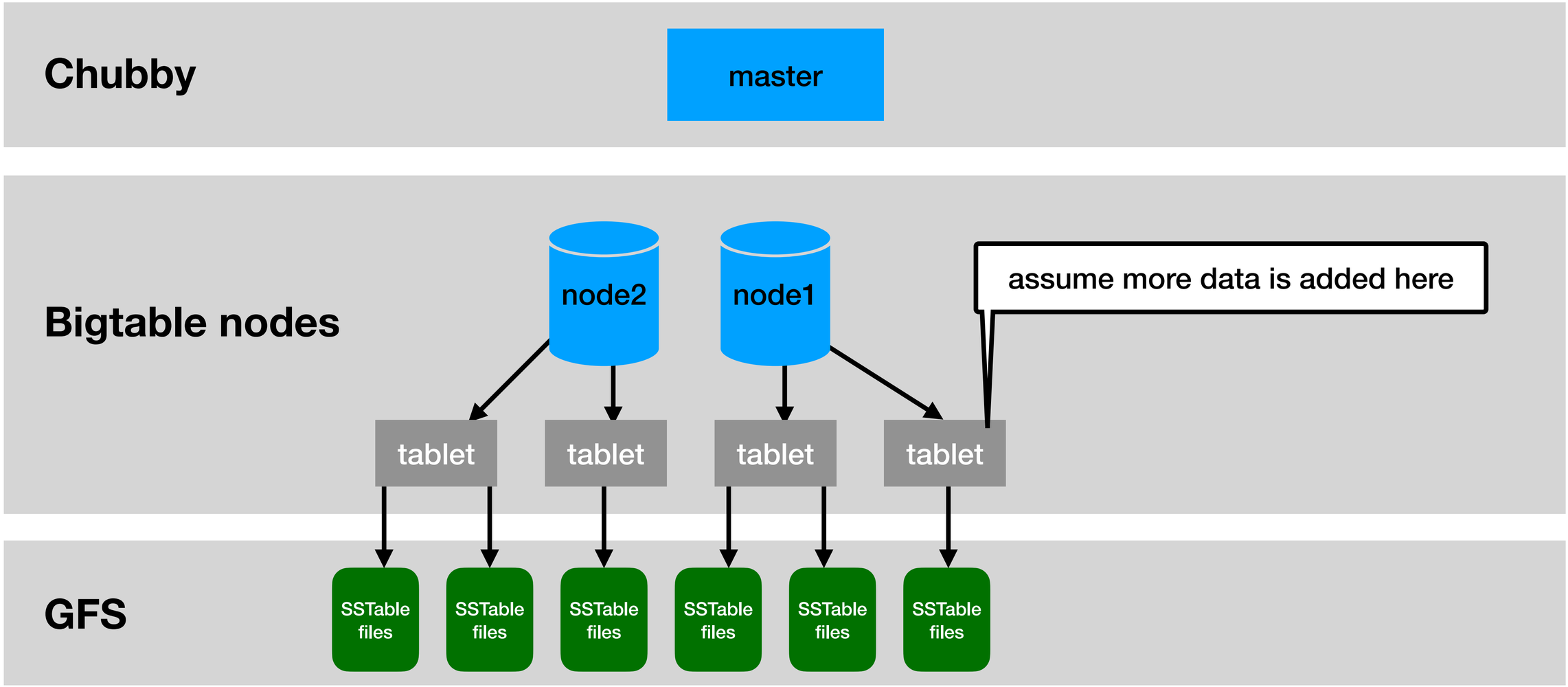
Components by layers



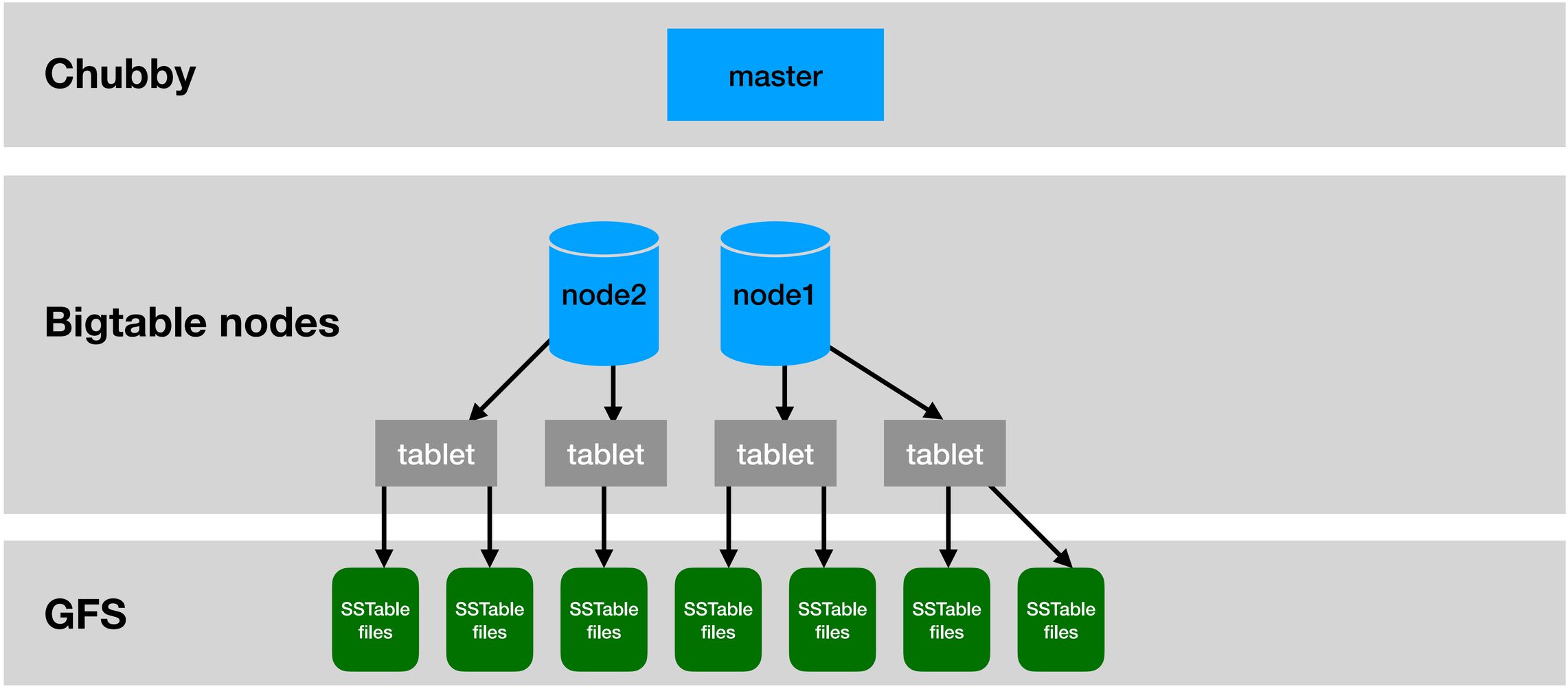
Components by layers



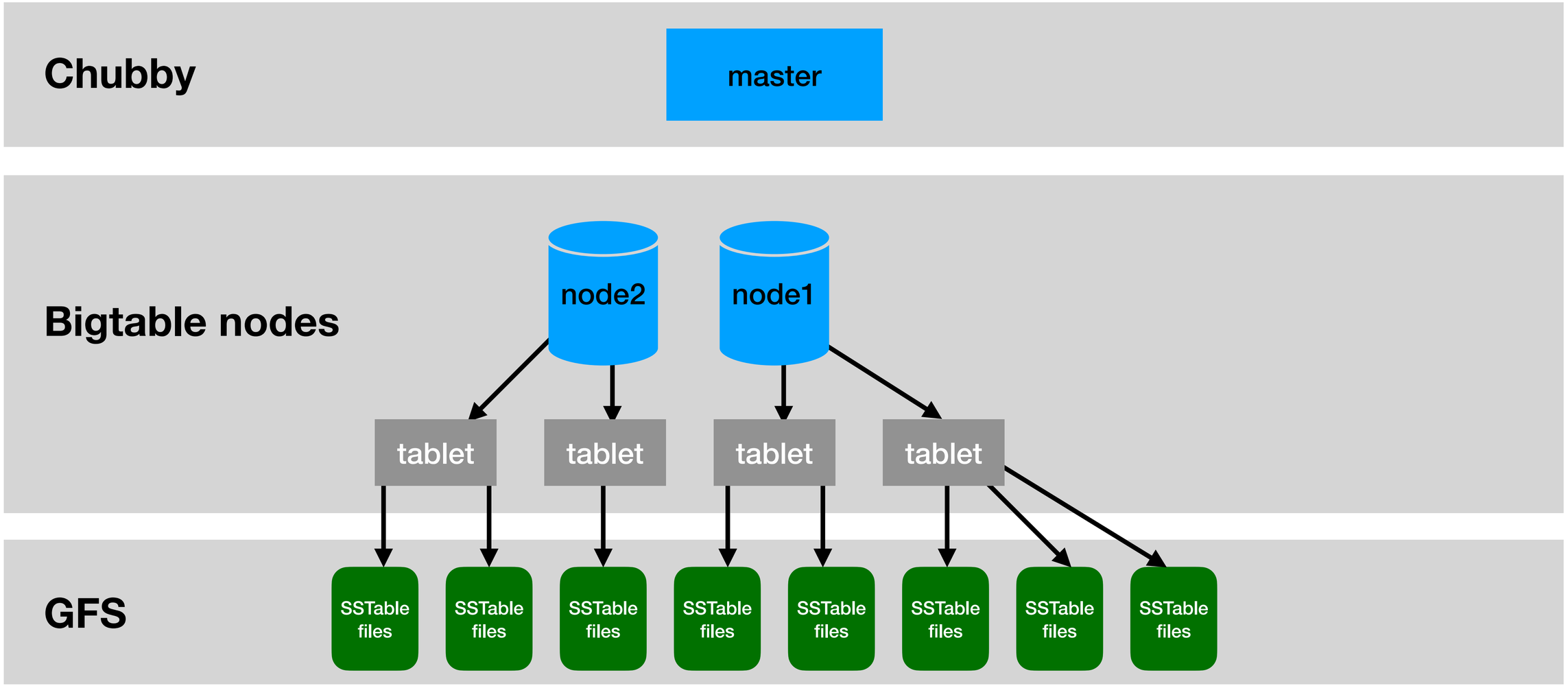
Components by layers



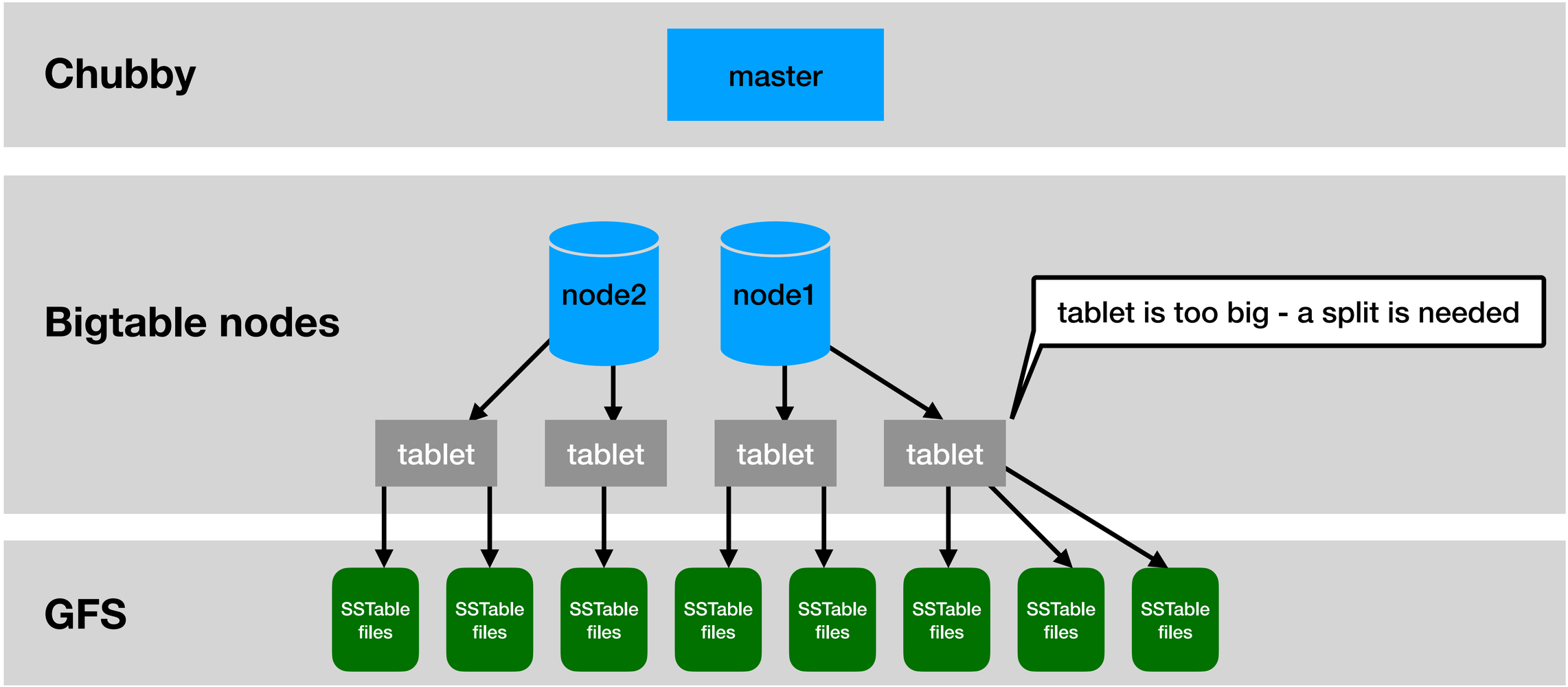
Components by layers



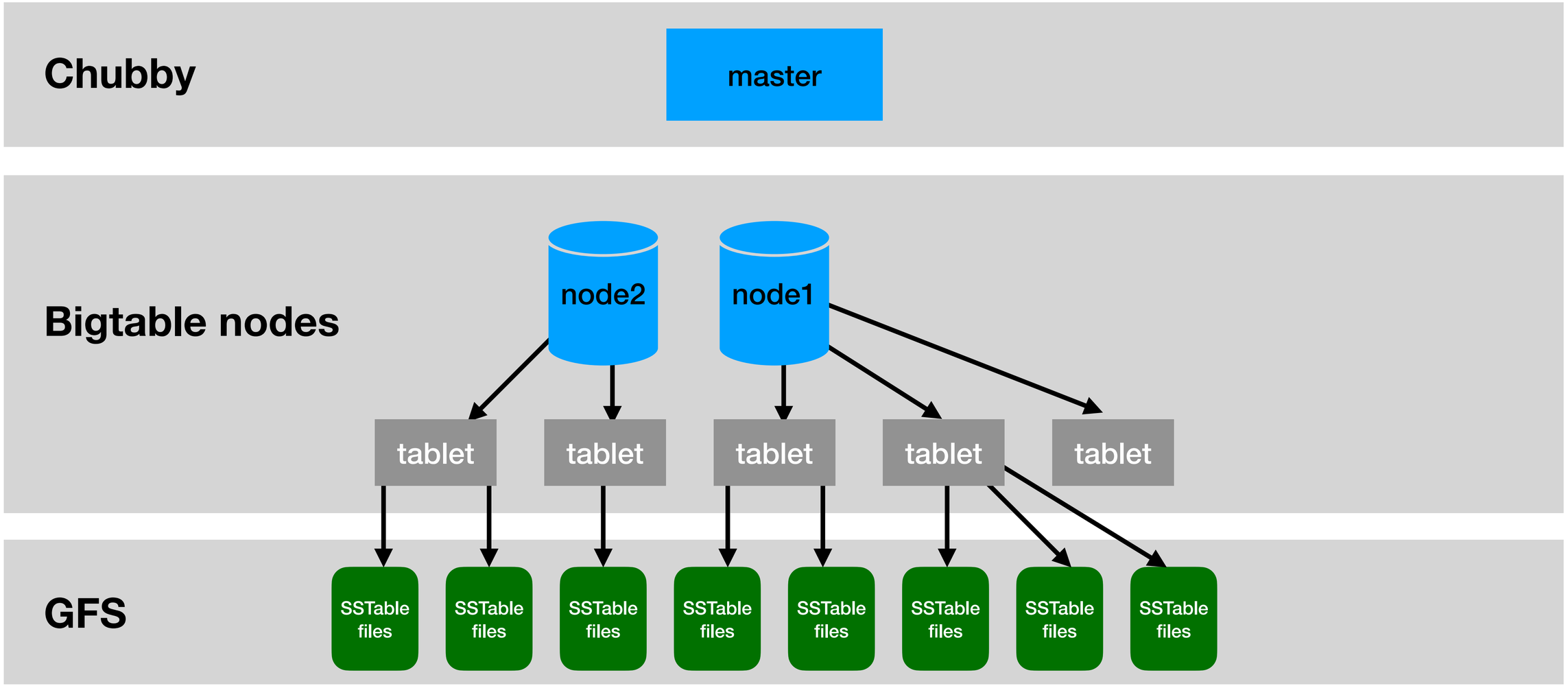
Components by layers



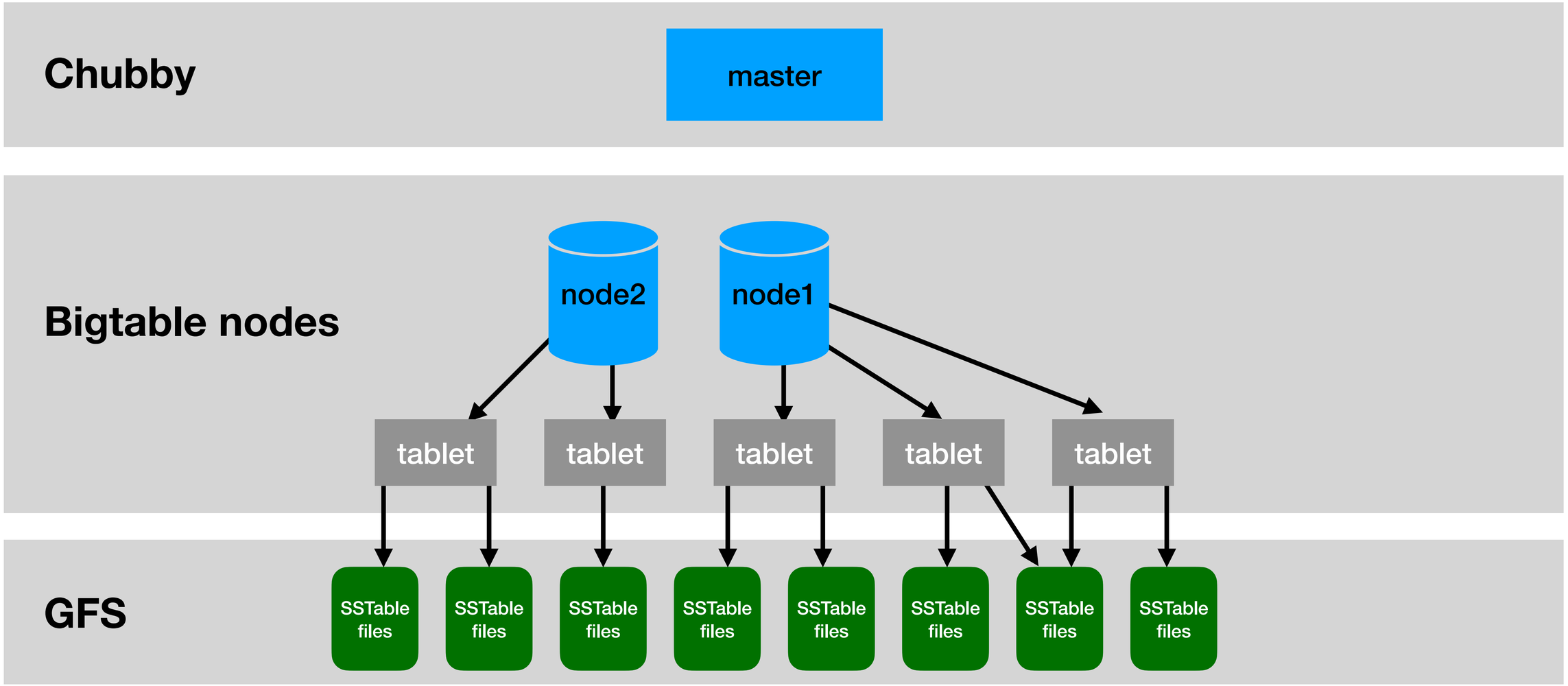
Components by layers



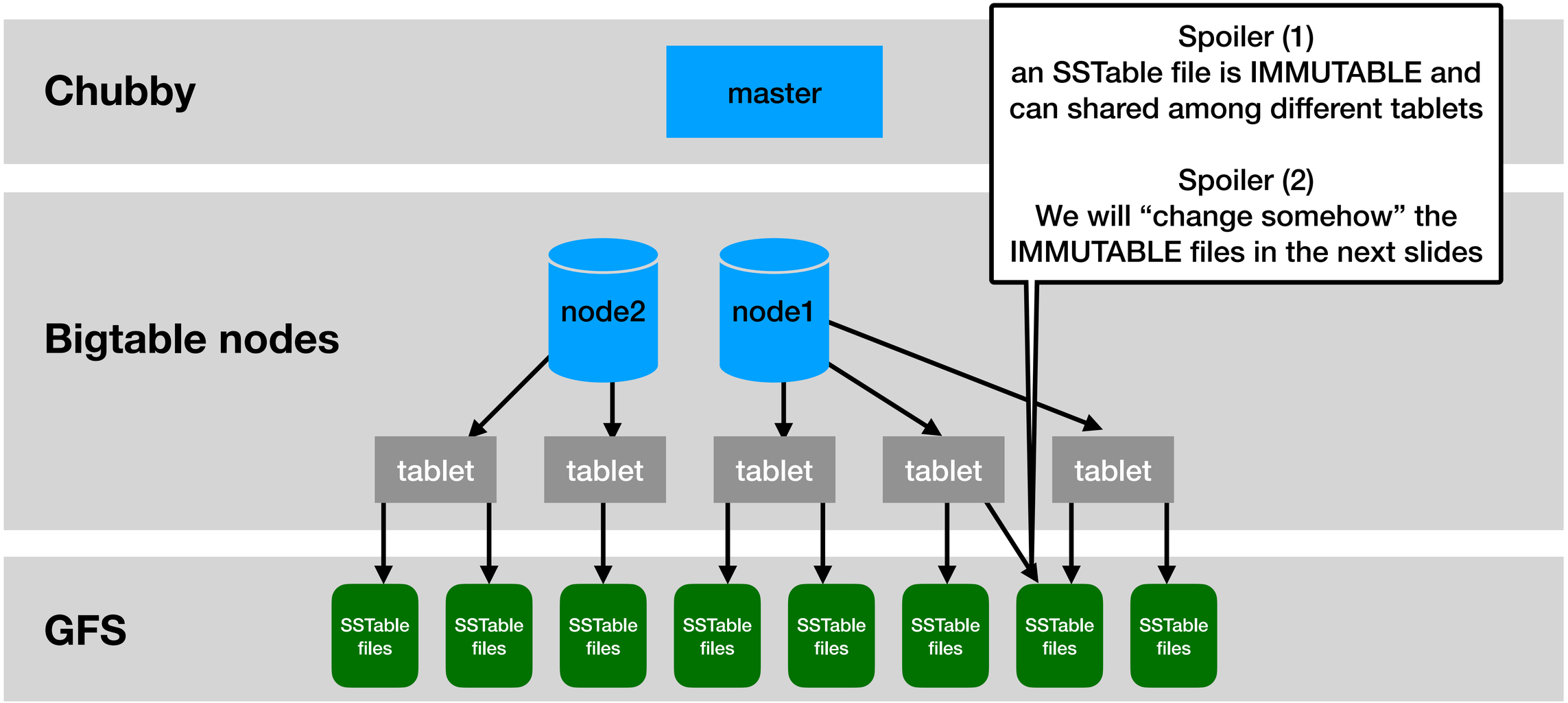
Components by layers



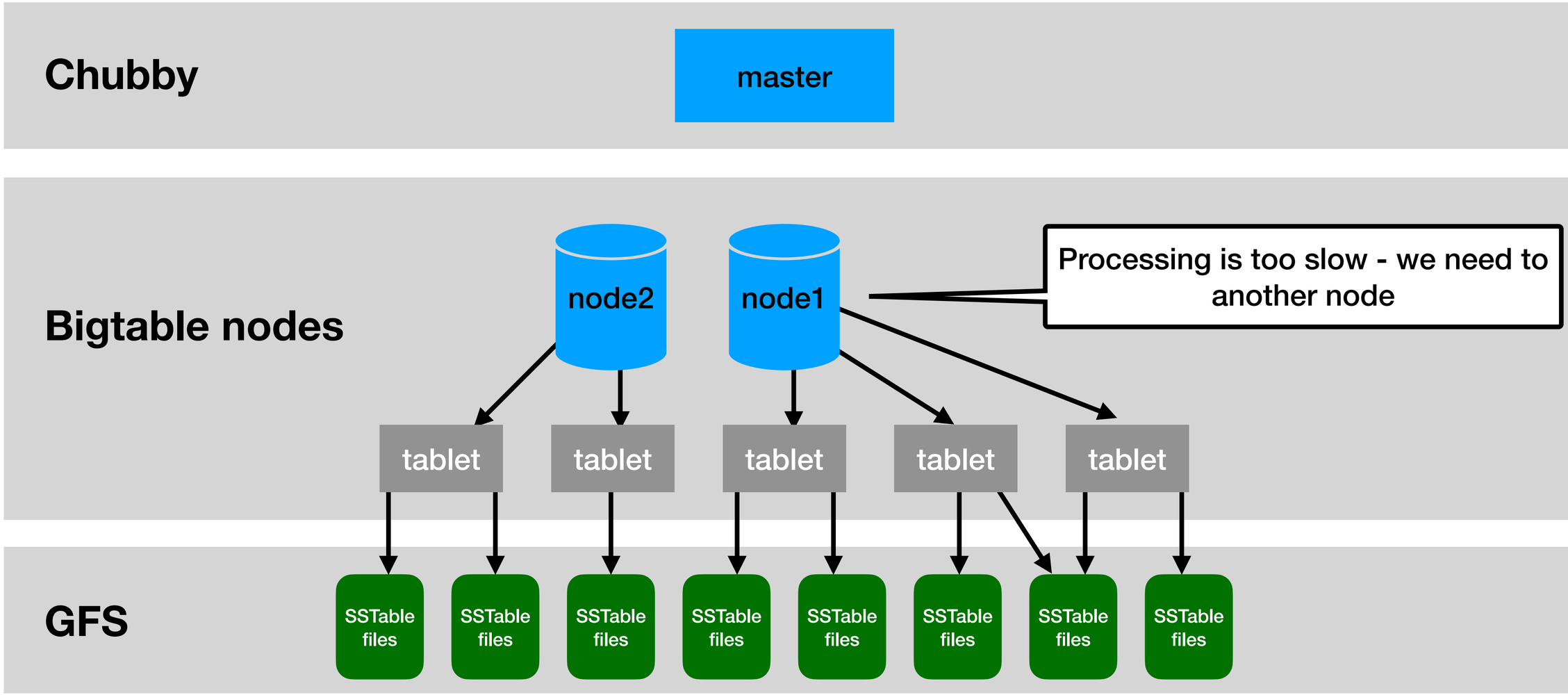
Components by layers



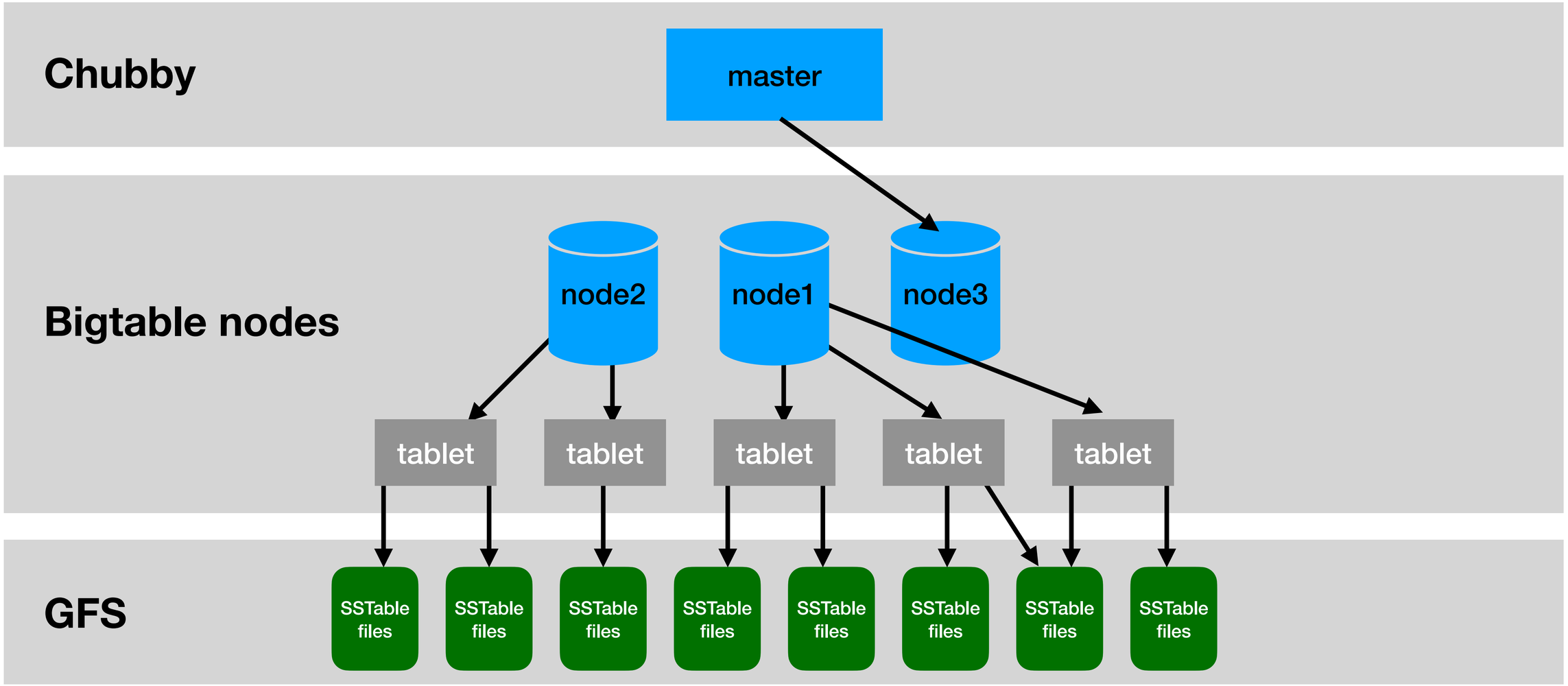
Components by layers



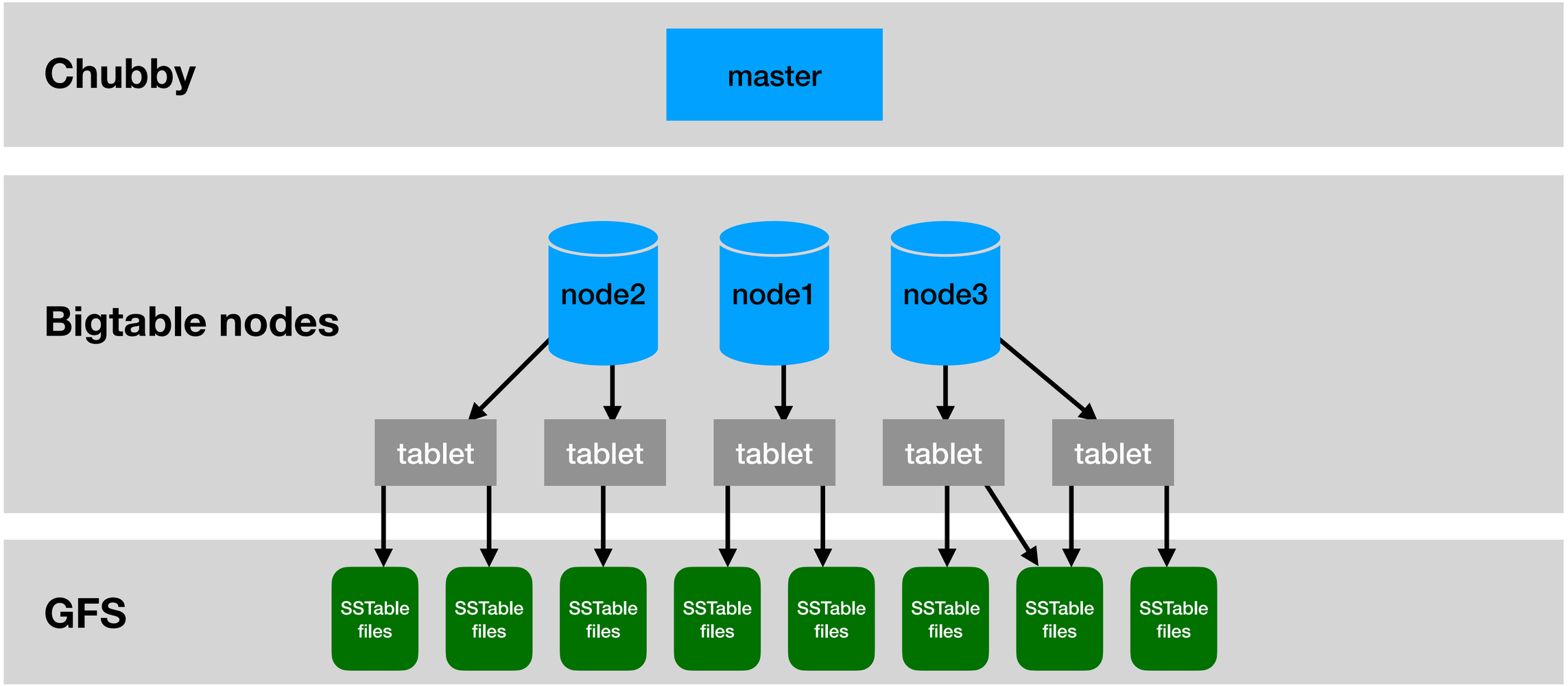
Components by layers



Components by layers



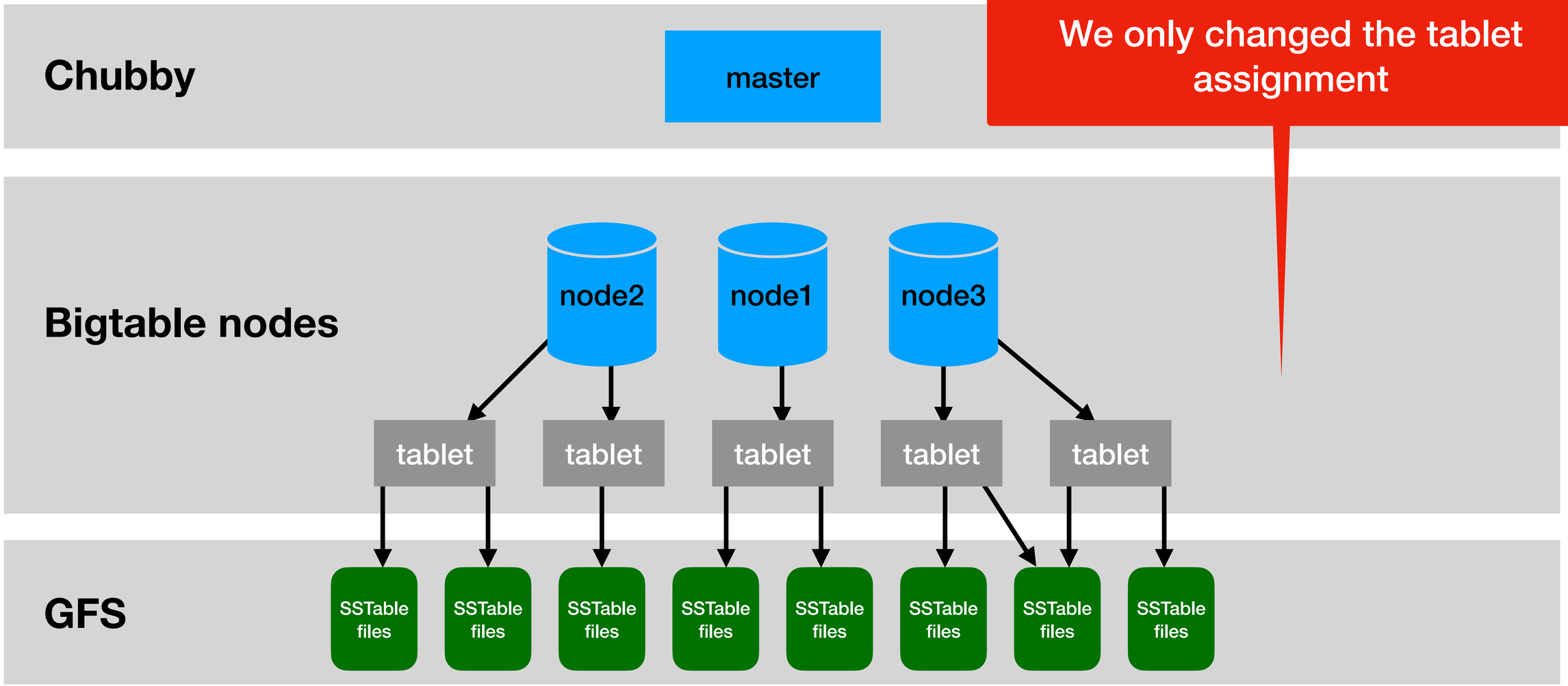
Components by layers



Components by layers

note - we did NOT copy any SSTable when we added a node.

We only changed the tablet assignment



Agenda

- History
- Data model
- Building blocks
- **SSTable (and memtable)**
- Bloom filter
- Summary
- Extra - Chubby
- Extra - Tablet location

“Querying” a tablet

- On updates (insert/update/delete):
  - Writes to a log (to redo on failures)
  - Updates the memtable

Memtable: a sorted buffer in memory

- Once the memtable reaches a threshold
  - it is saved to an **immutable** SSTable file
  - A new empty one is initialized

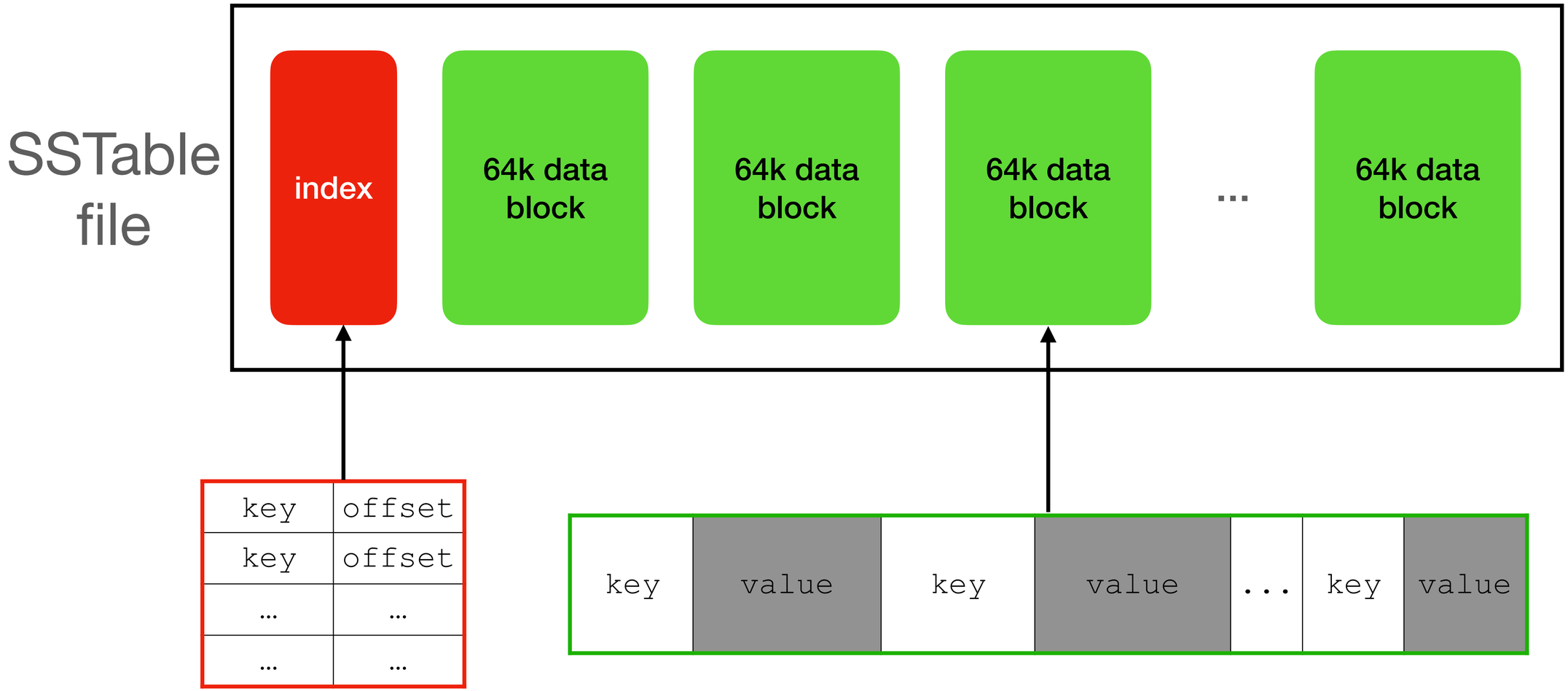
Minor compaction

- On read, we first search the value in the memtable,
  then (if not found) in all other SSTables by their order (last one first)

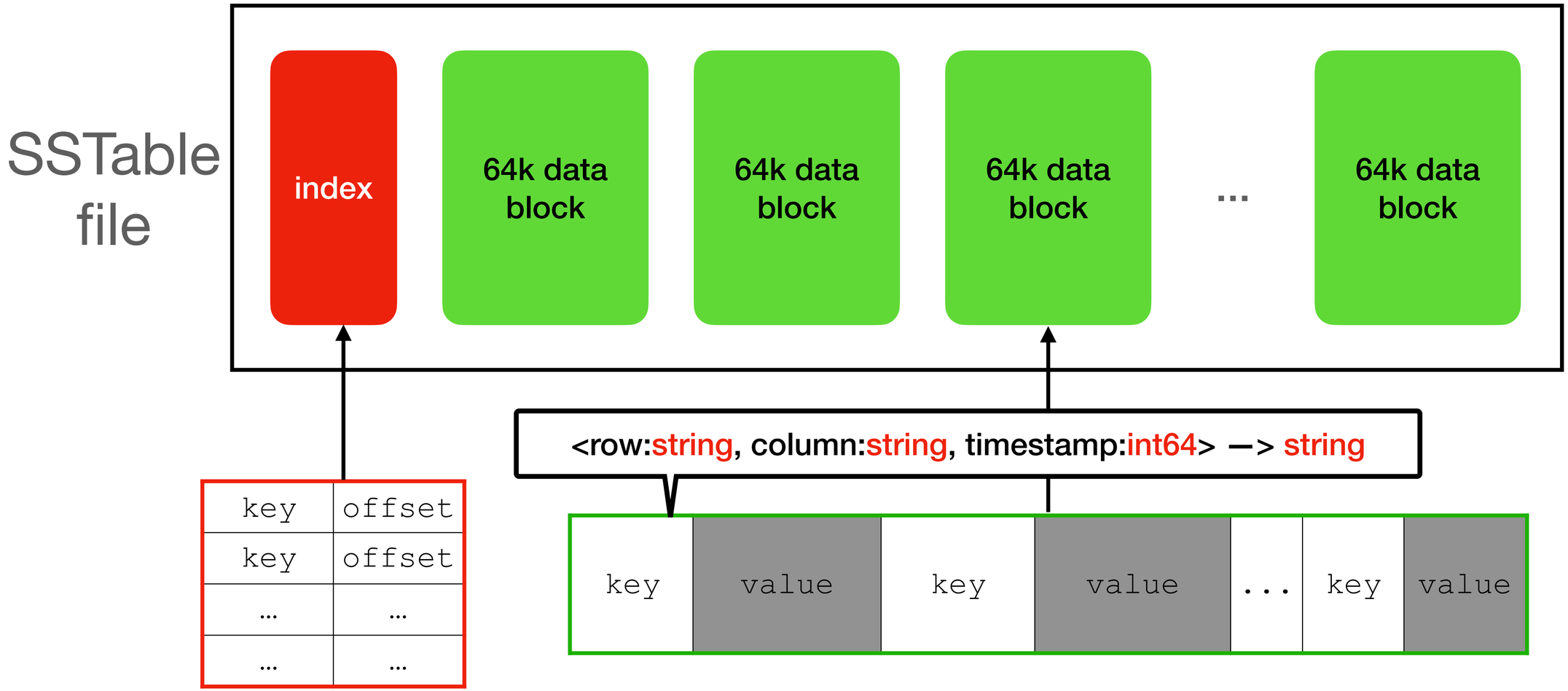
Sorted String Table (SSTable)

- A file format
- **Immutable**
- Provides a persistent ordered map (key-value)

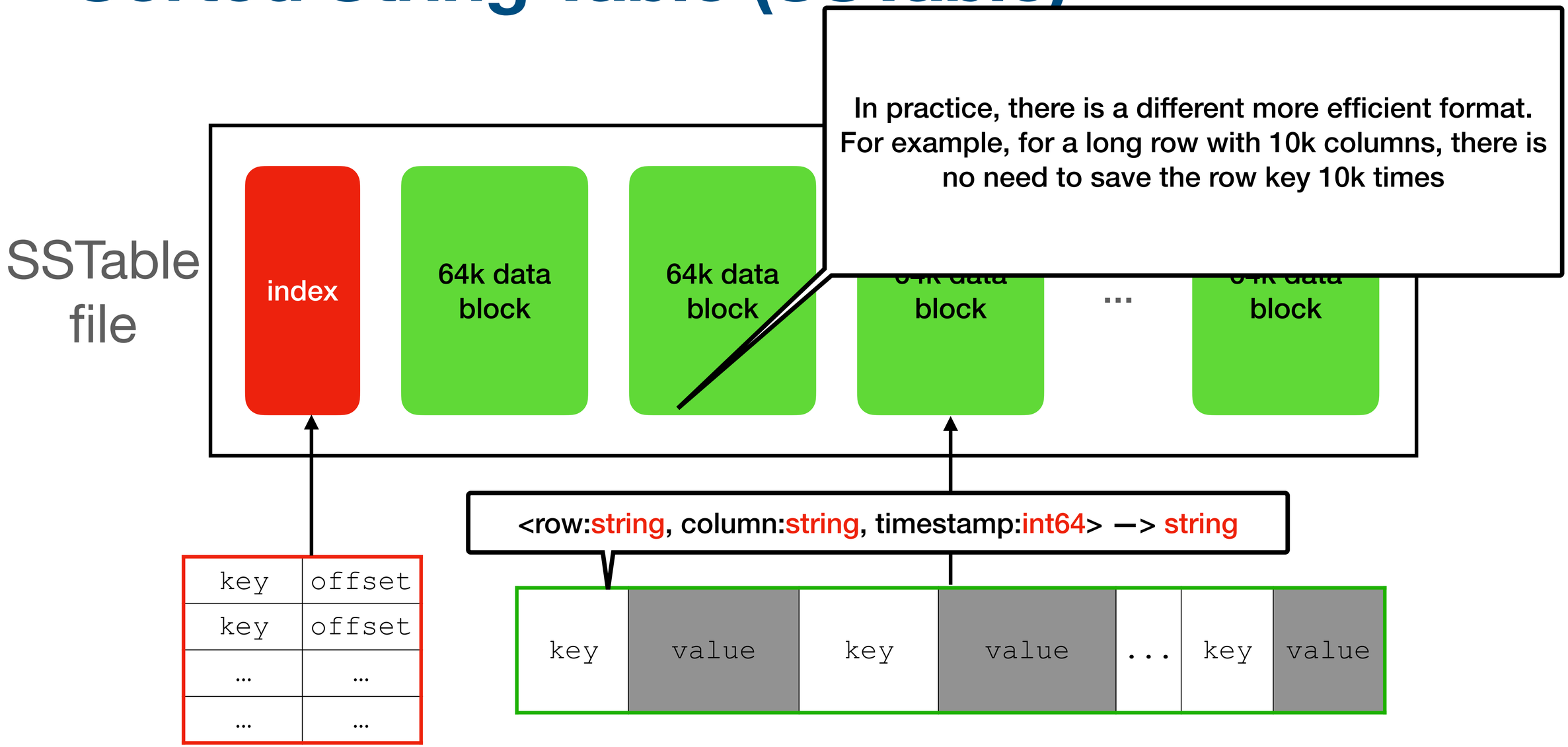
Sorted String Table (SSTable)



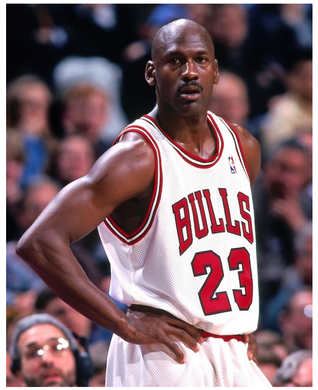
Sorted String Table (SSTable)

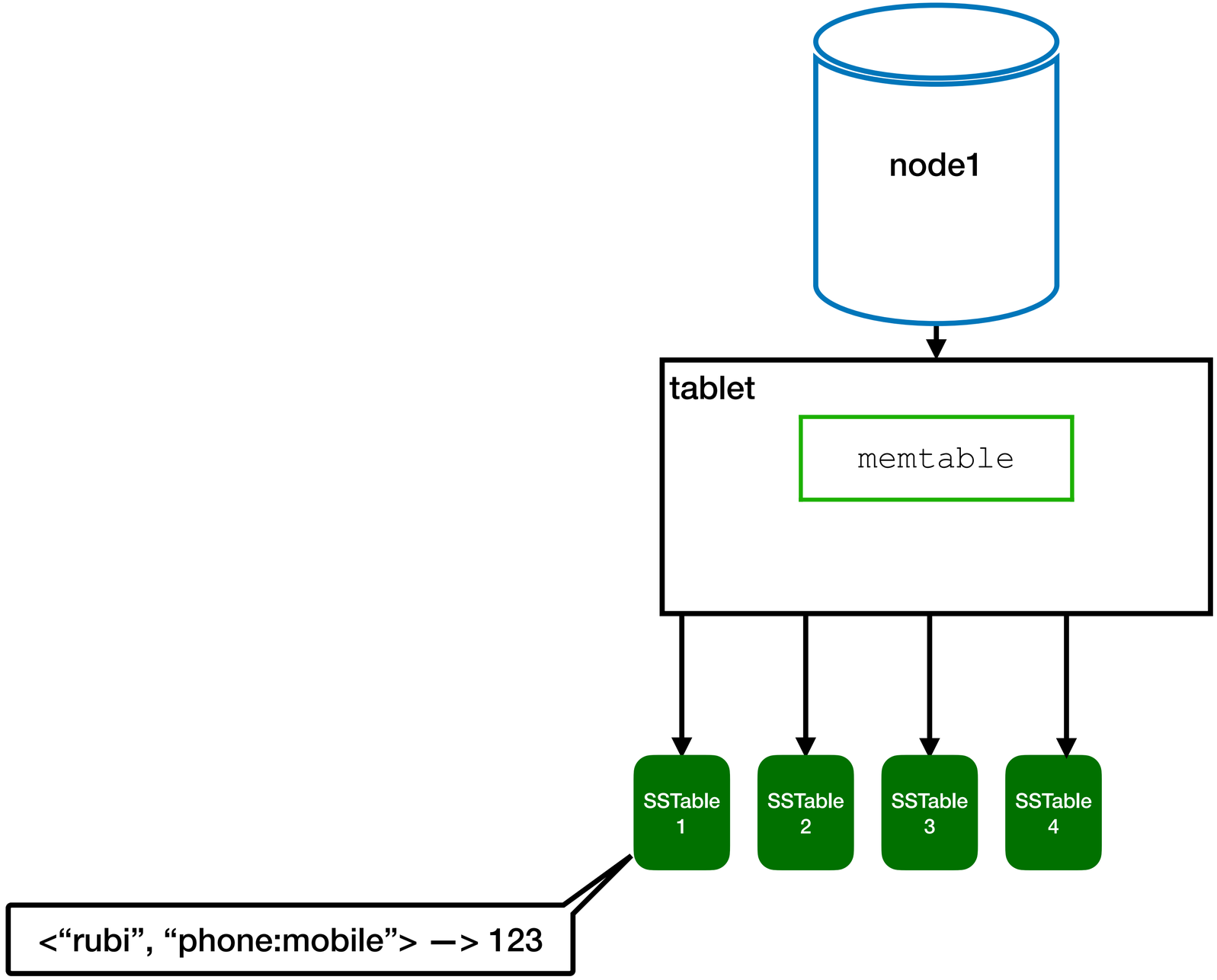


Sorted String Table (SSTable)

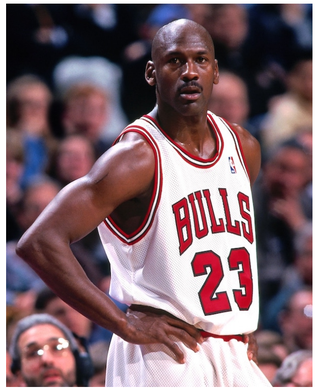


Example

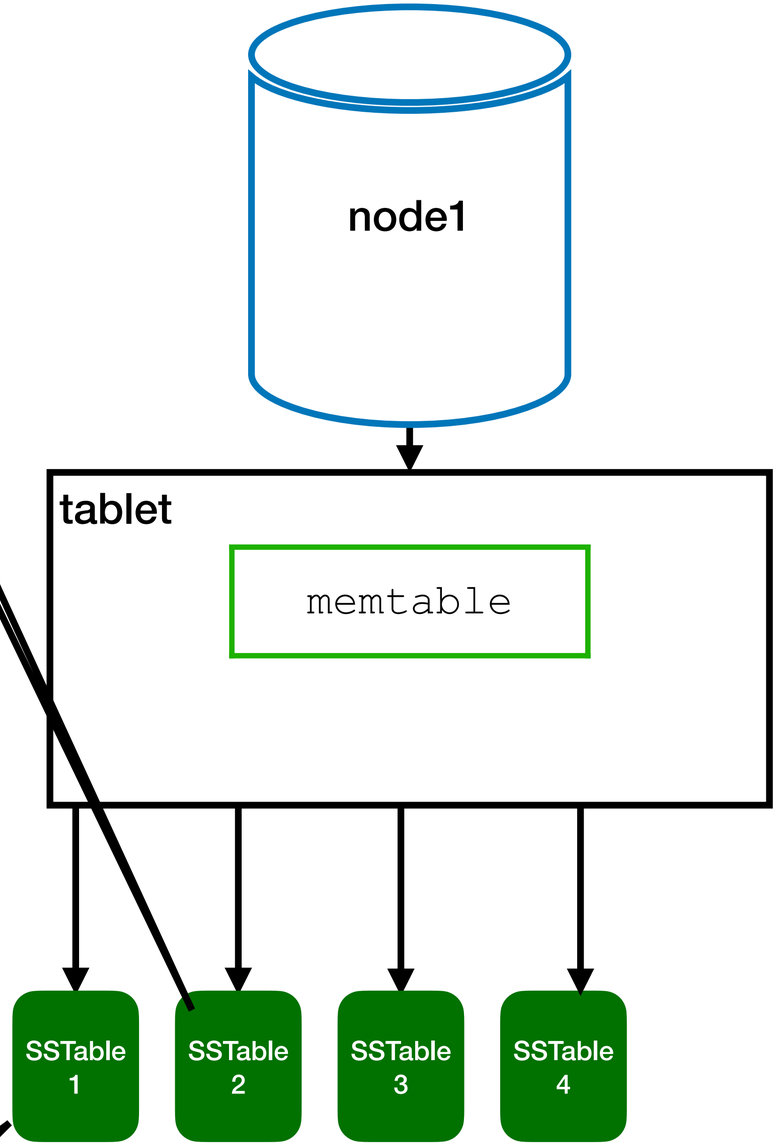

BULLS
23


node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
<“rubi”, “phone:mobile”> —> 123

Example

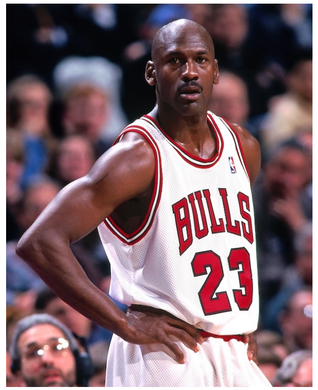


Note - we keep track of the “order” of the SSTables



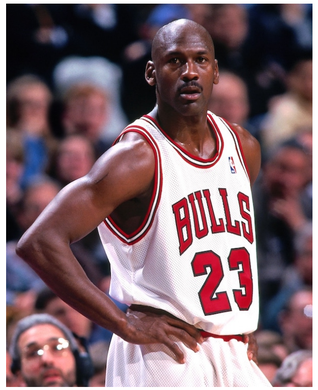
<“rubi”, “phone:mobile”> —> 123

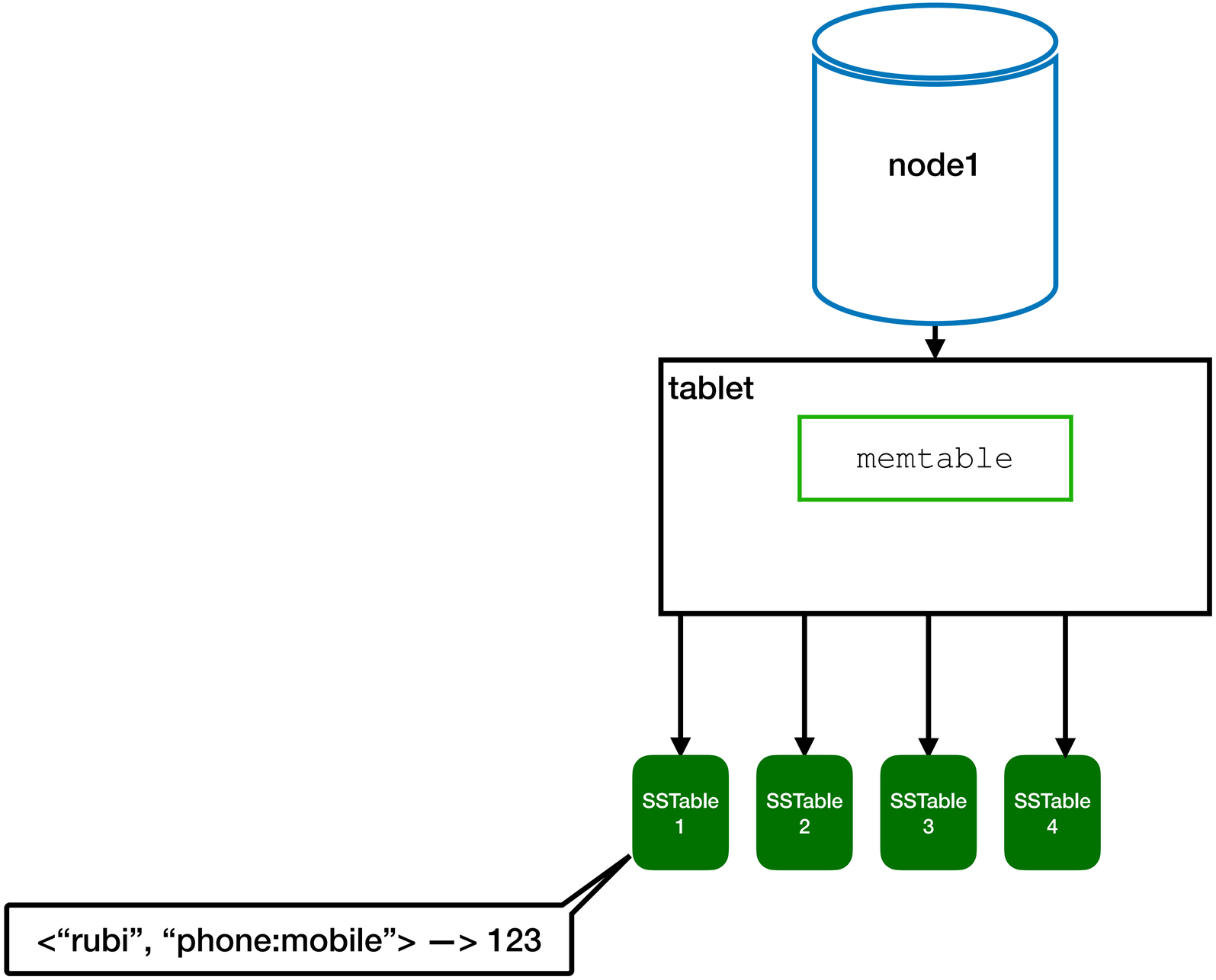
Example


BULLS
23

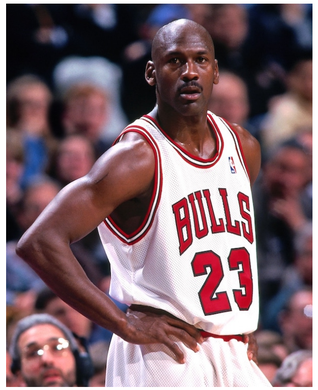
node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
A timestamp should also be here. For simplicity we ignore for now
<“rubi”, “phone:mobile”> —> 123

Example

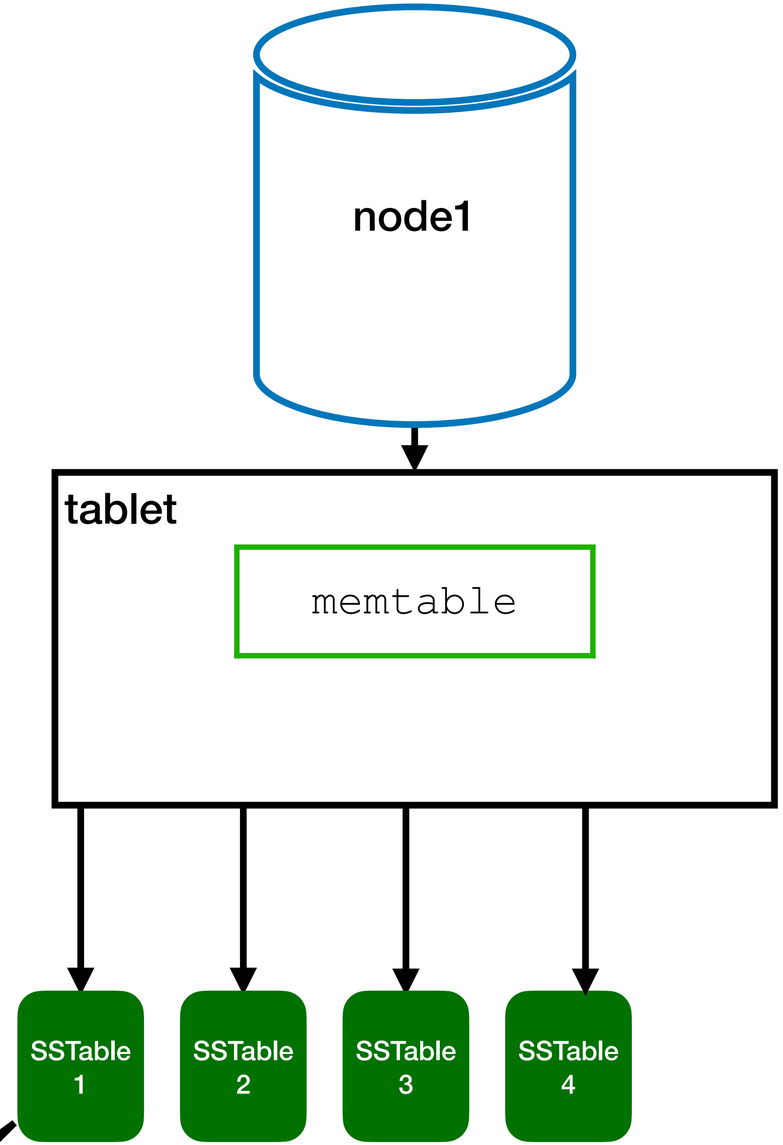

BULLS
23


node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
<“rubi”, “phone:mobile”> —> 123

Example

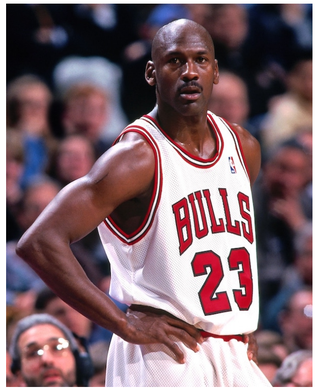


So what is Rubi's mobile number?

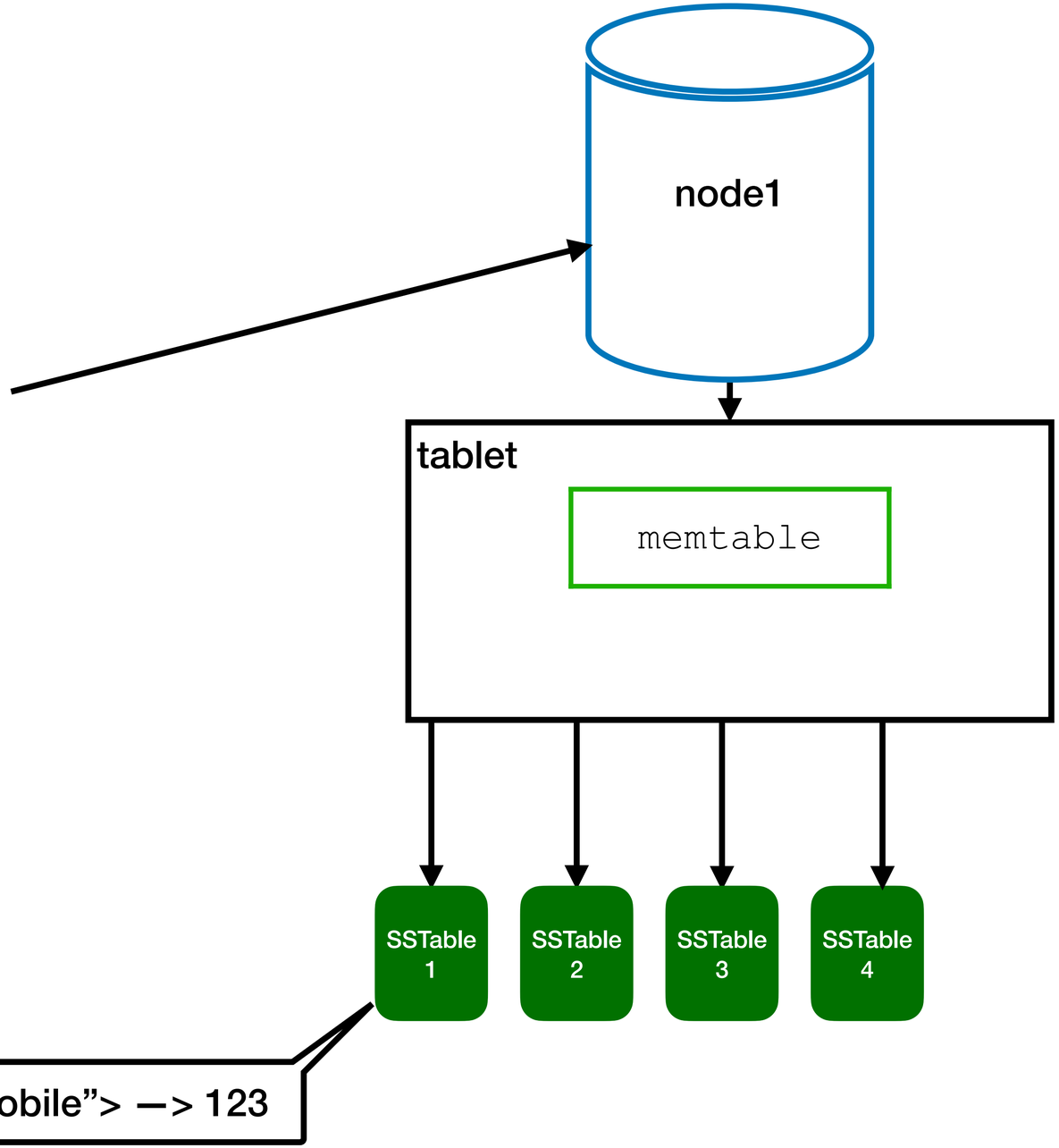


<"rubi", "phone:mobile"> —> 123

Example

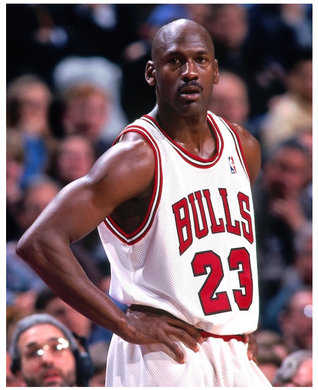


query: <“rubi”, “phone:mobile”>

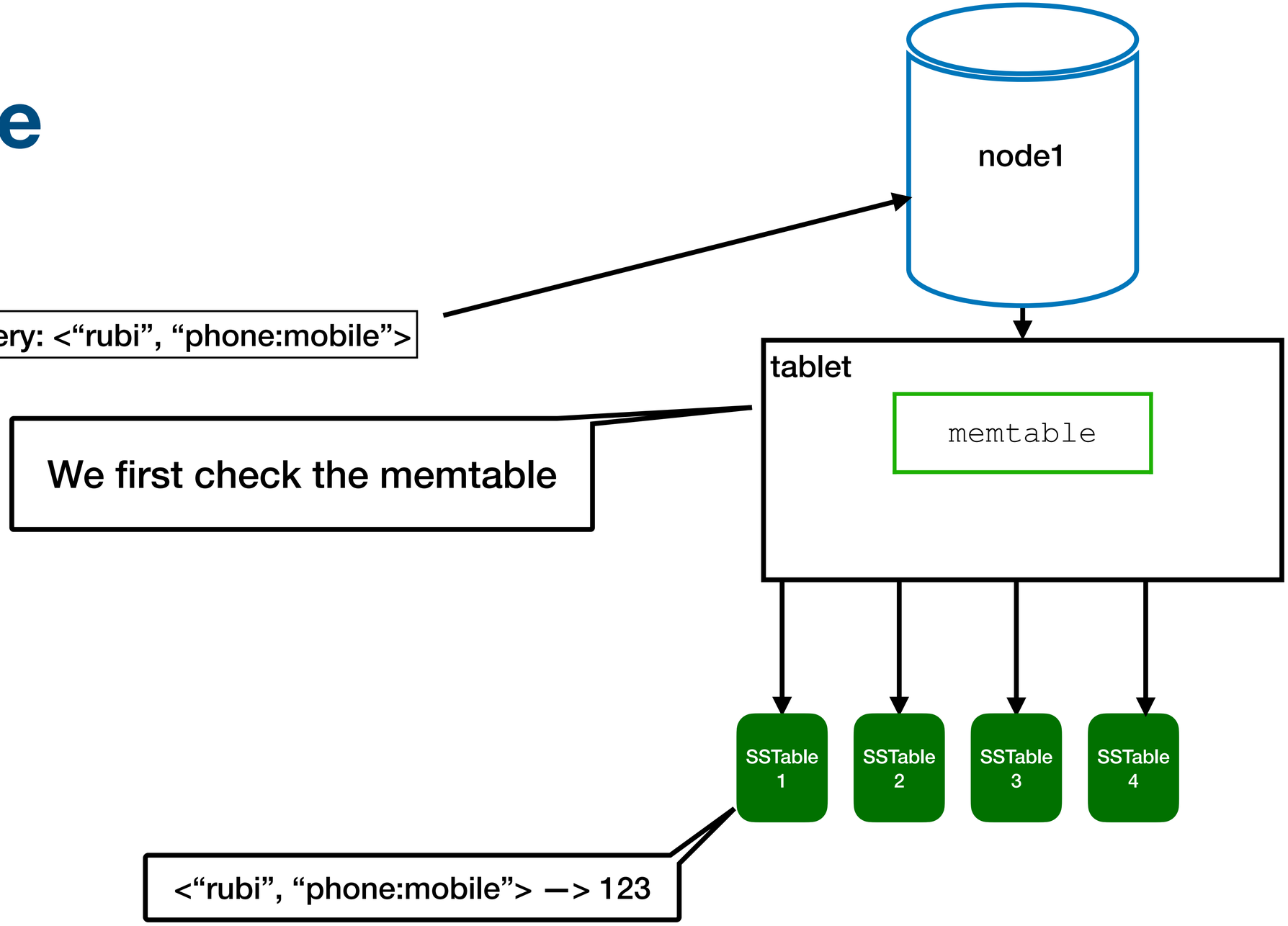


<“rubi”, “phone:mobile”> —> 123

Example



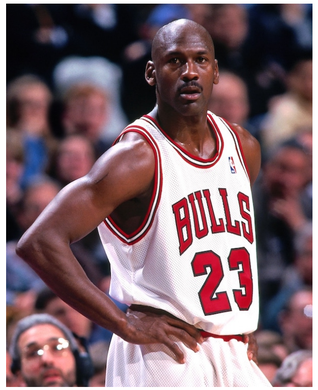
query: <“rubi”, “phone:mobile”>



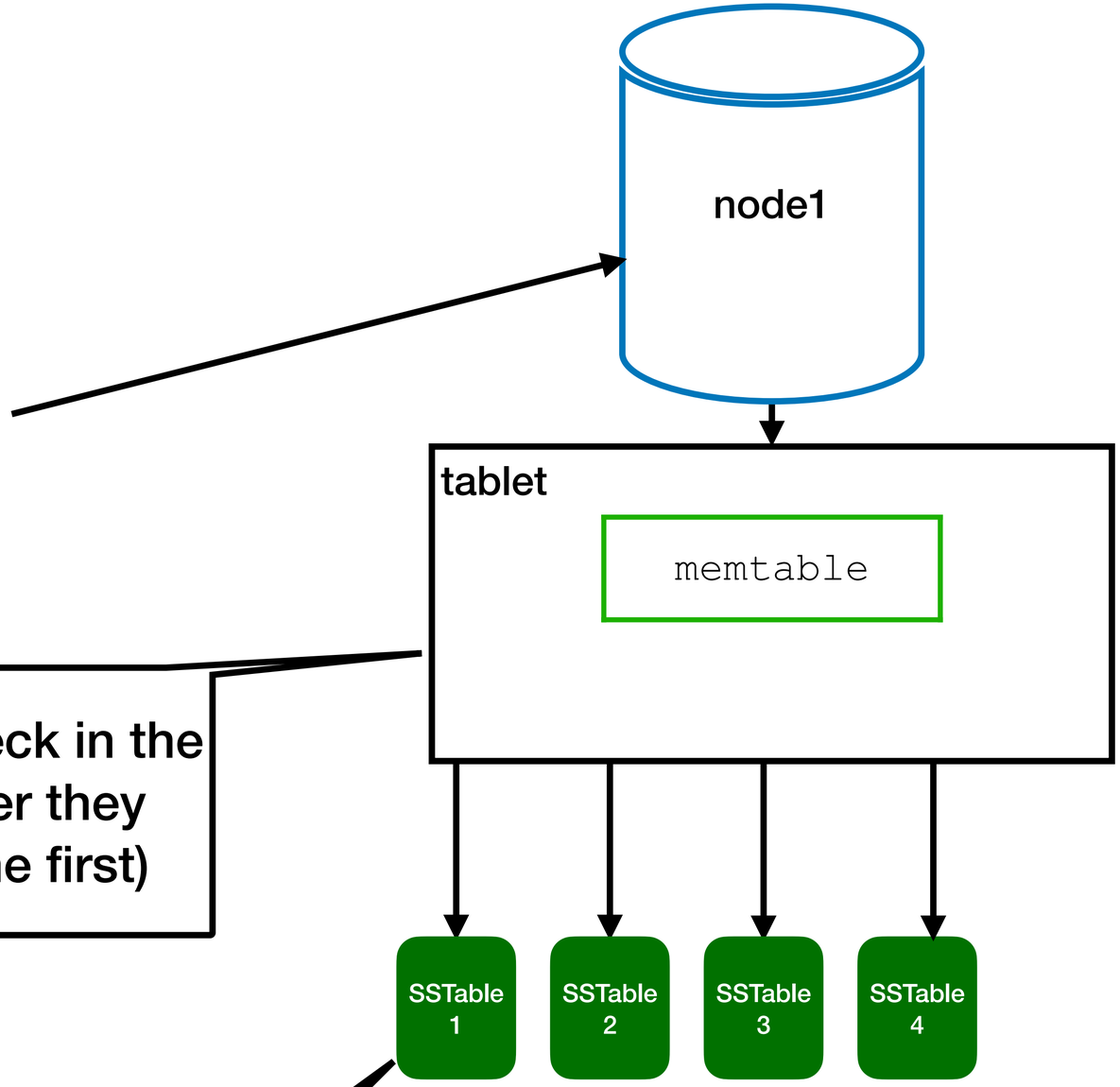
We first check the memtable

<“rubi”, “phone:mobile”> —> 123

Example



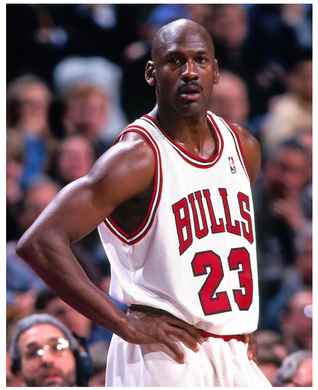
query: <“rubi”, “phone:mobile”>



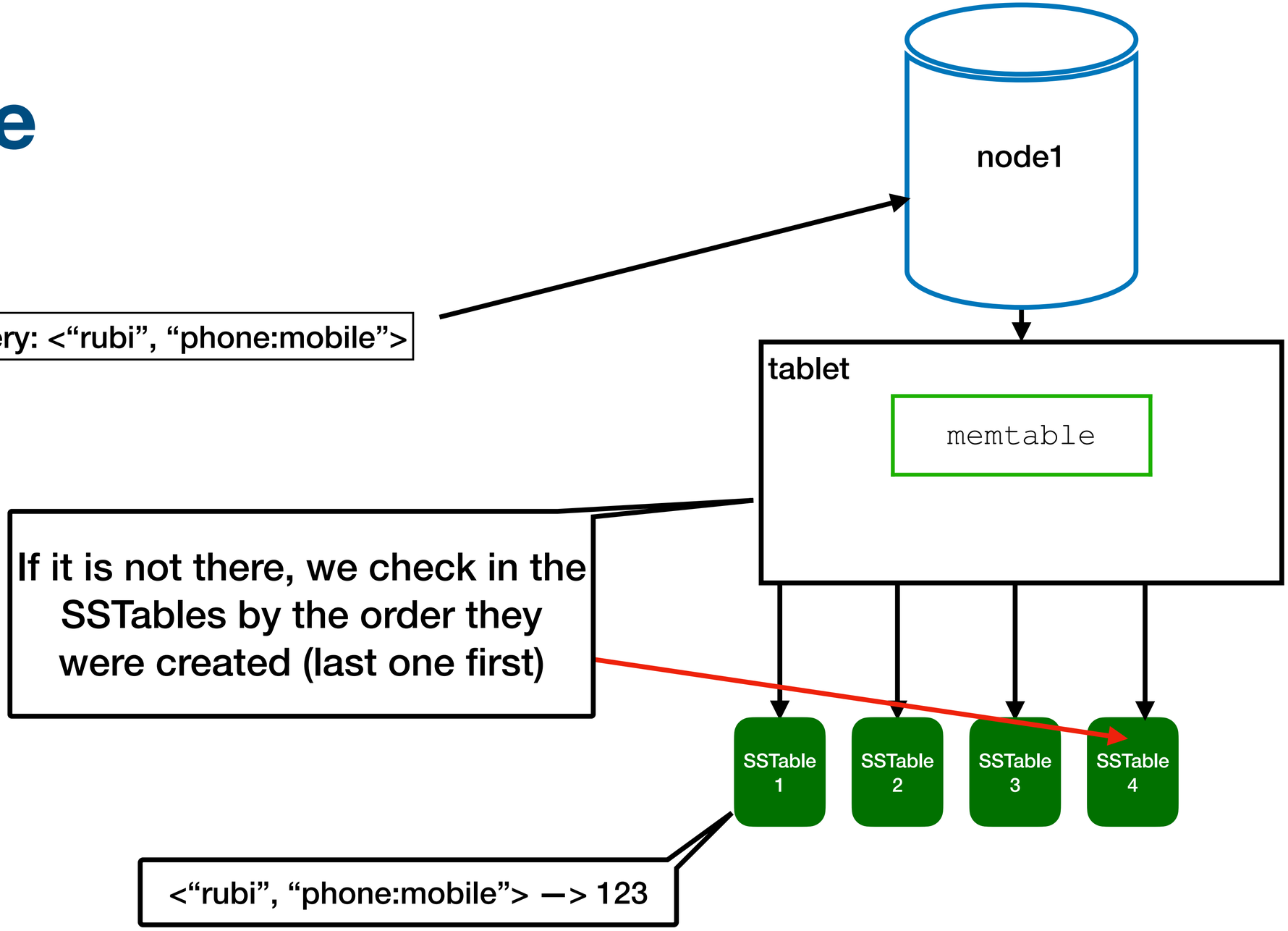
If it is not there, we check in the SSTables by the order they were created (last one first)

<“rubi”, “phone:mobile”> —> 123

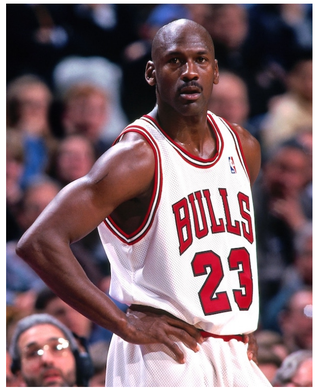
Example



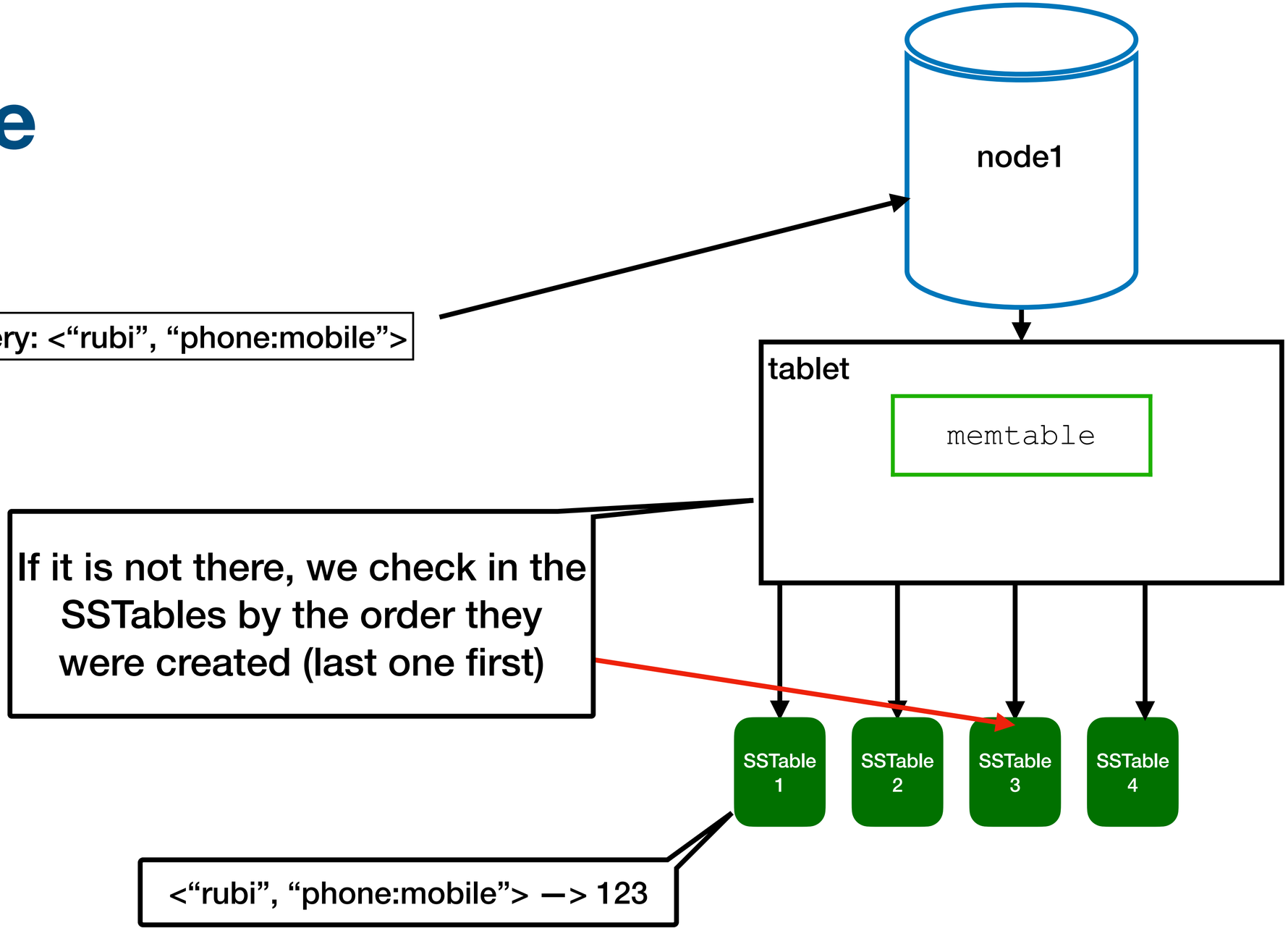
query: <“rubi”, “phone:mobile”>



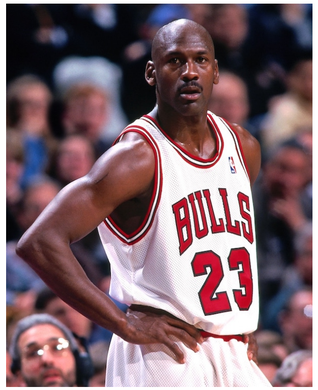
Example



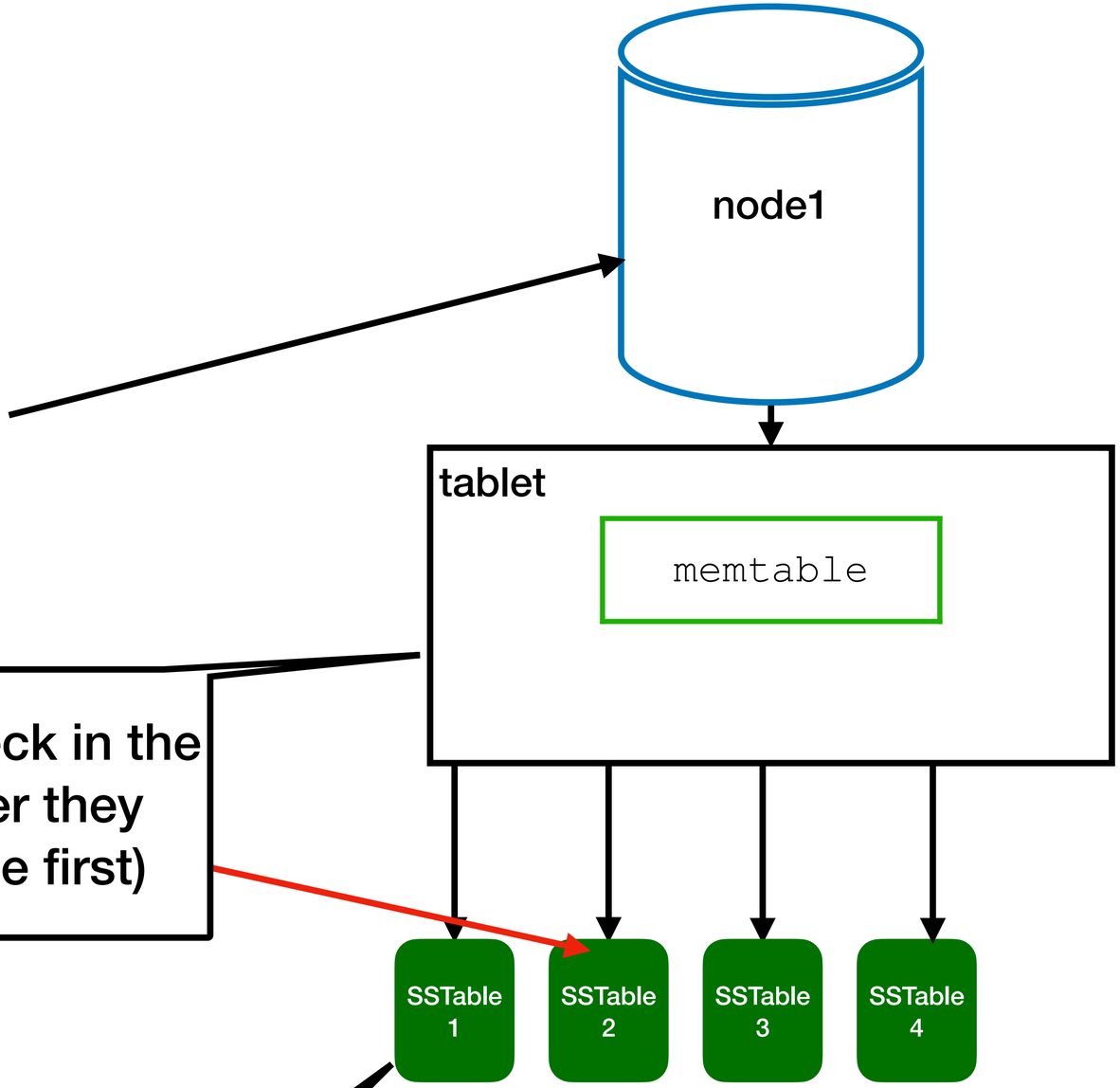
query: <“rubi”, “phone:mobile”>



Example



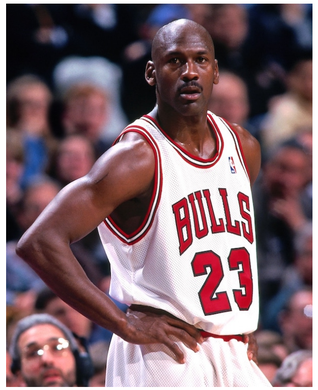
query: <“rubi”, “phone:mobile”>



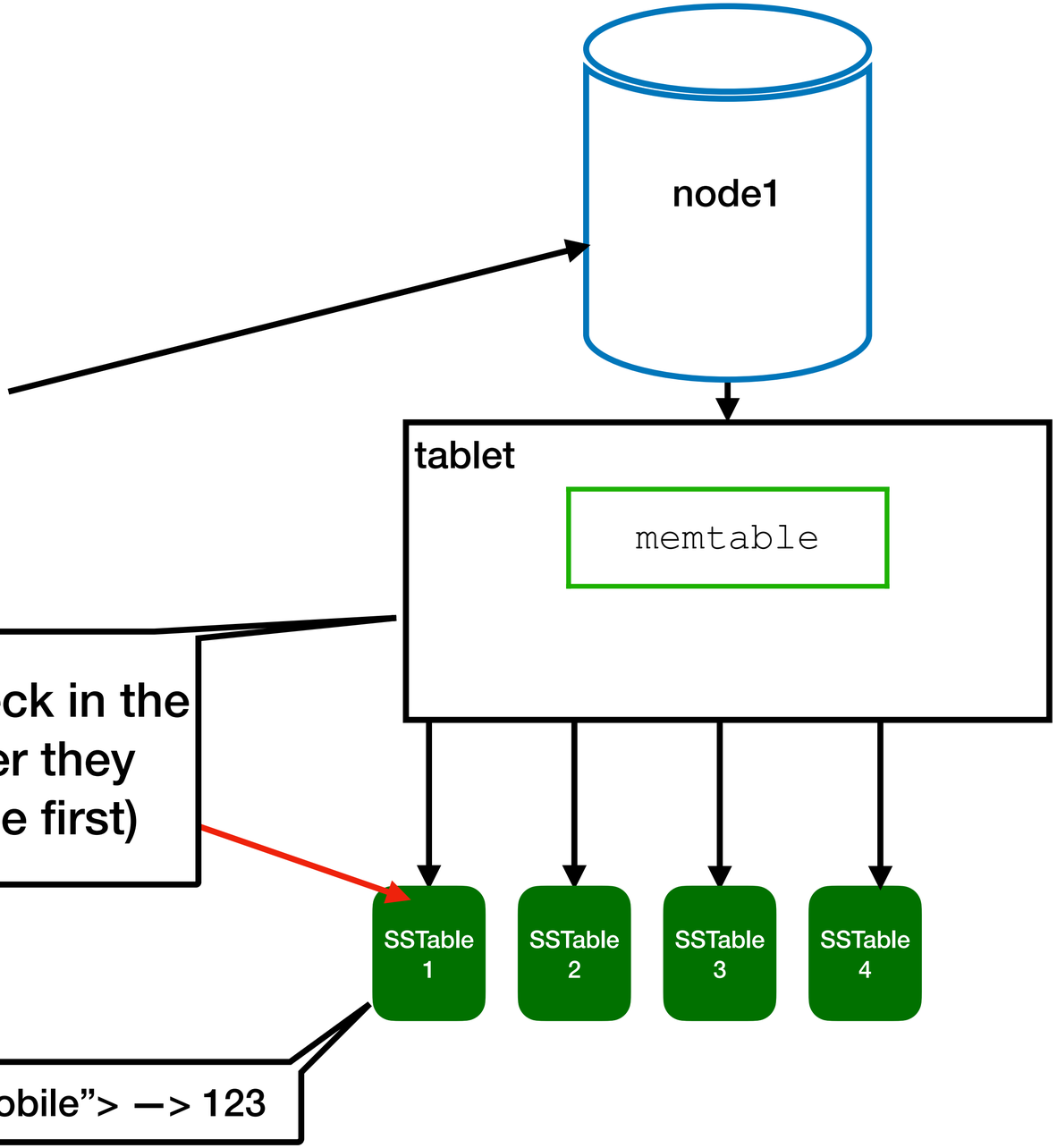
If it is not there, we check in the SSTables by the order they were created (last one first)

<“rubi”, “phone:mobile”> —> 123

Example



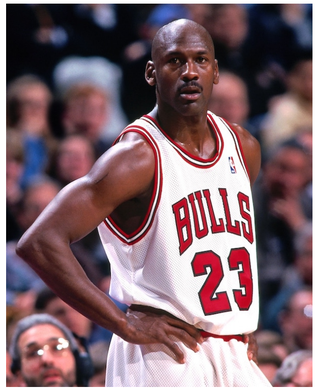
query: <“rubi”, “phone:mobile”>



If it is not there, we check in the SSTables by the order they were created (last one first)

<“rubi”, “phone:mobile”> —> 123

Example



query: <“rubi”, “phone:mobile”>

123

node1

tablet

memtable

SSTable 1

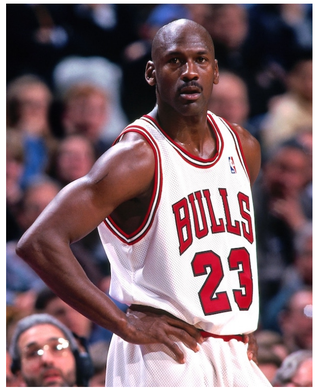
SSTable 2

SSTable 3

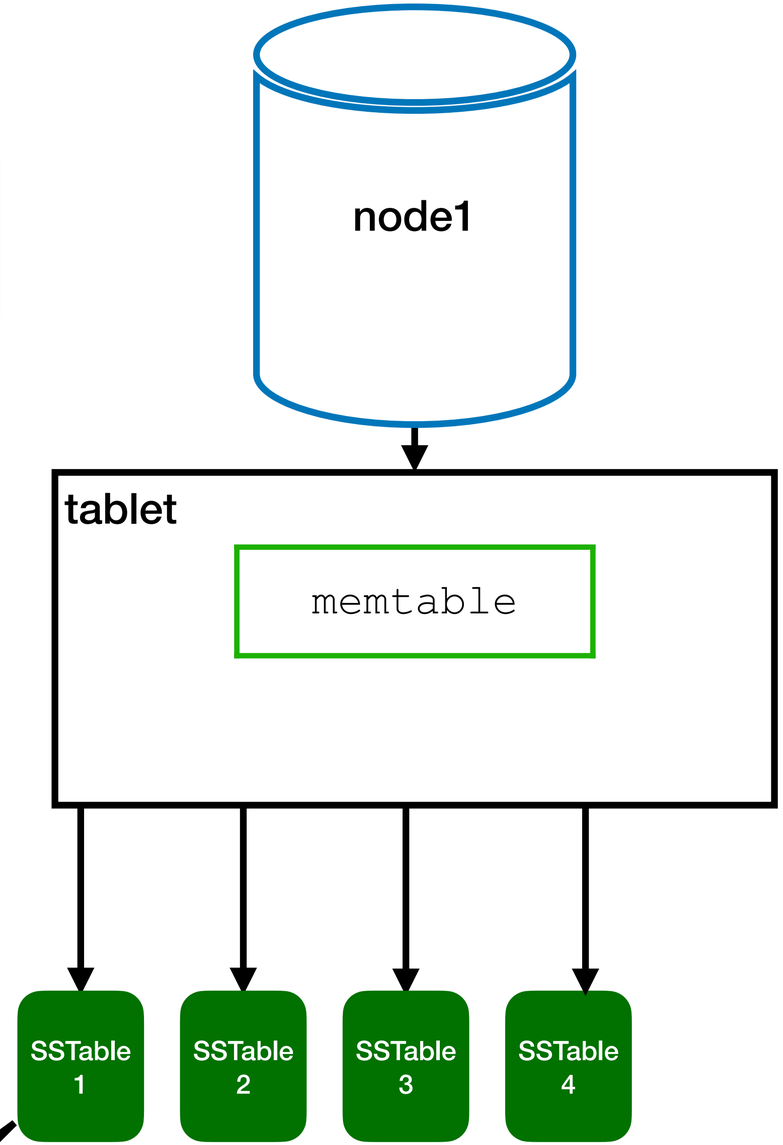
SSTable 4

<“rubi”, “phone:mobile”> —> 123

Example

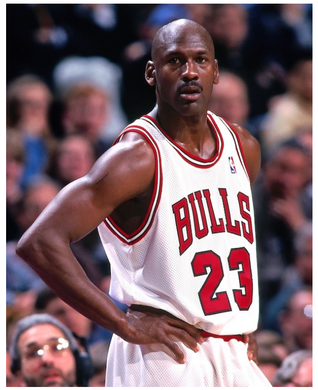


Rubi updates his mobile number

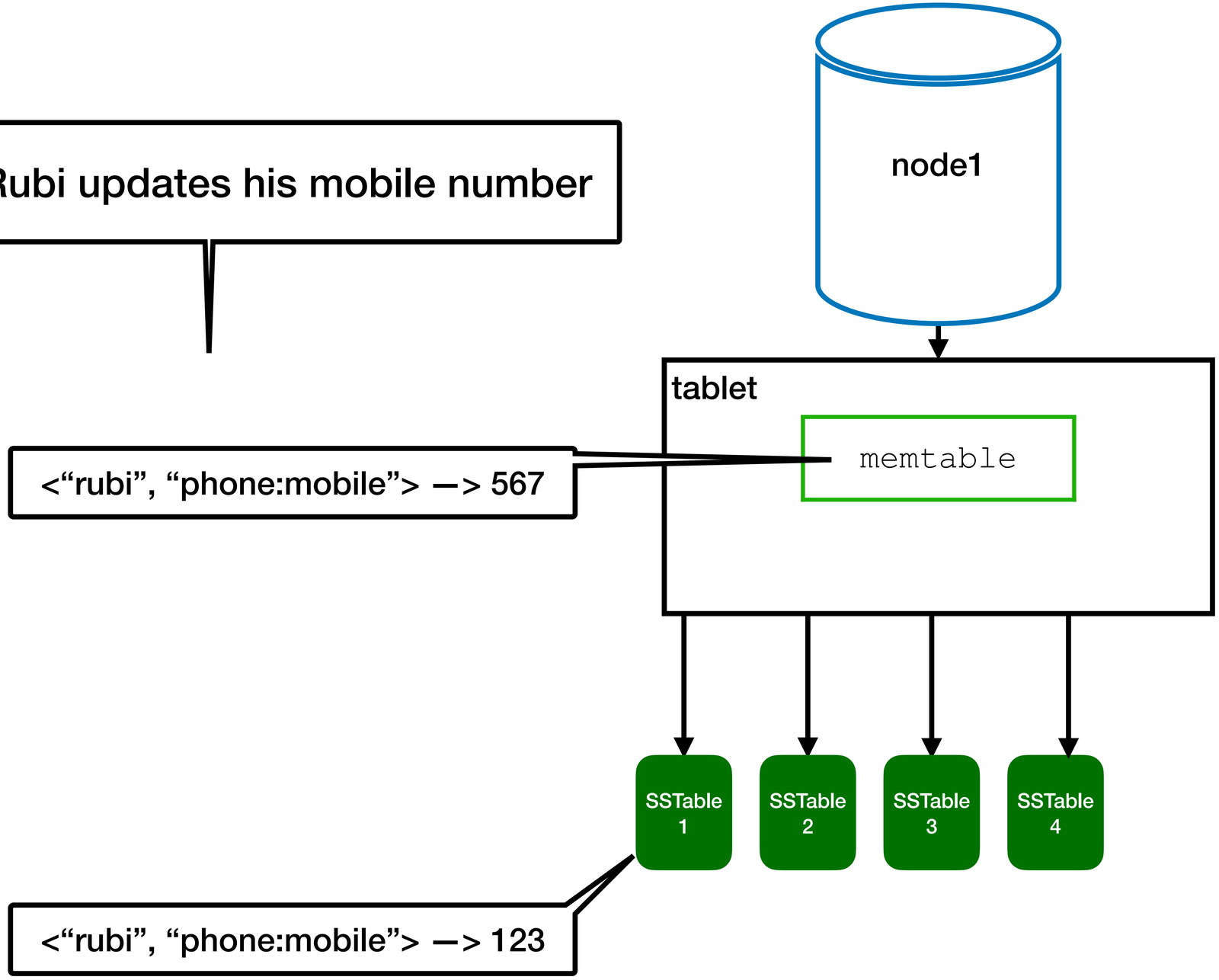


<“rubi”, “phone:mobile”> —> 123

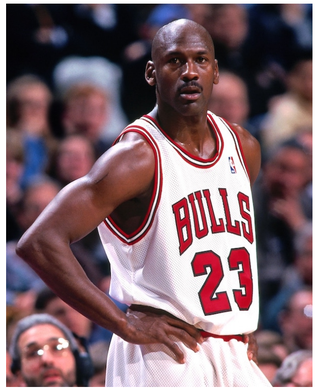
Example



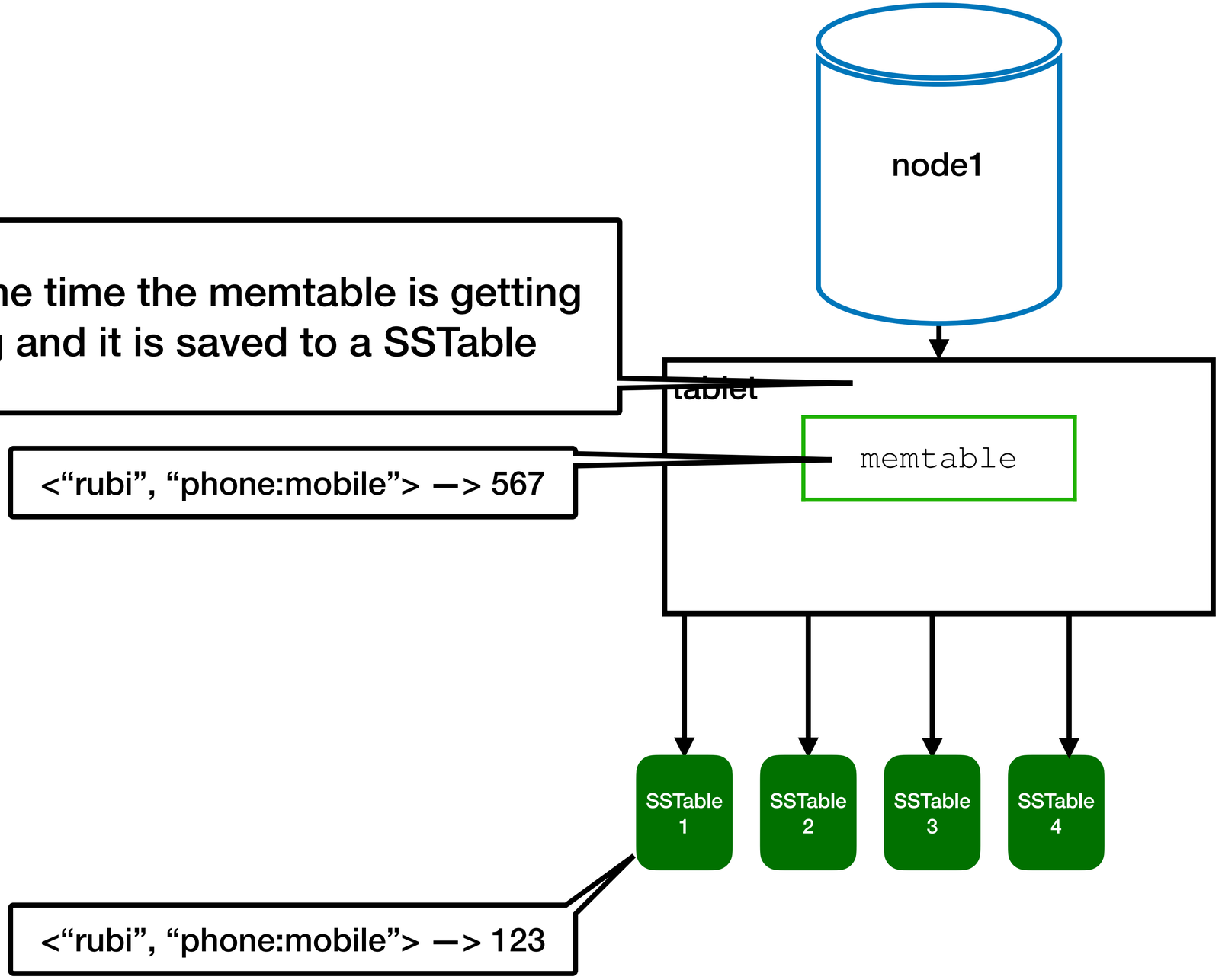
Rubi updates his mobile number



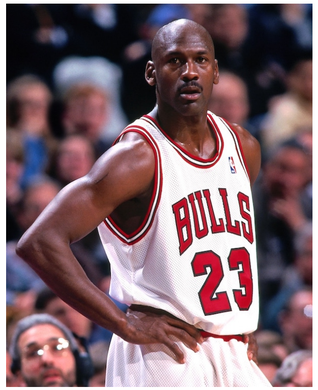
Example

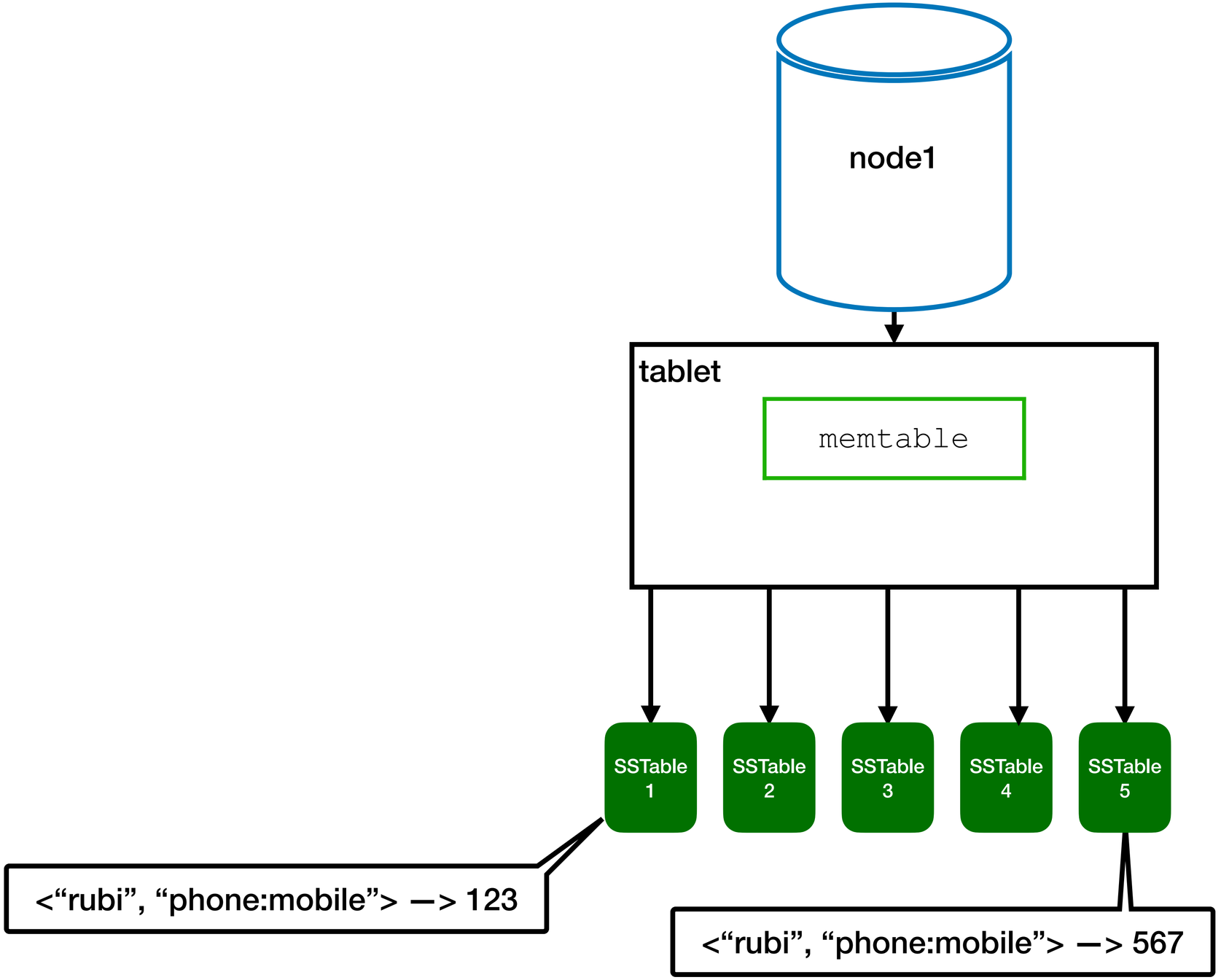


After some time the memtable is getting too big and it is saved to a SSTable

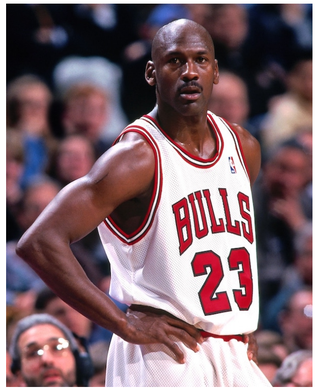


Example

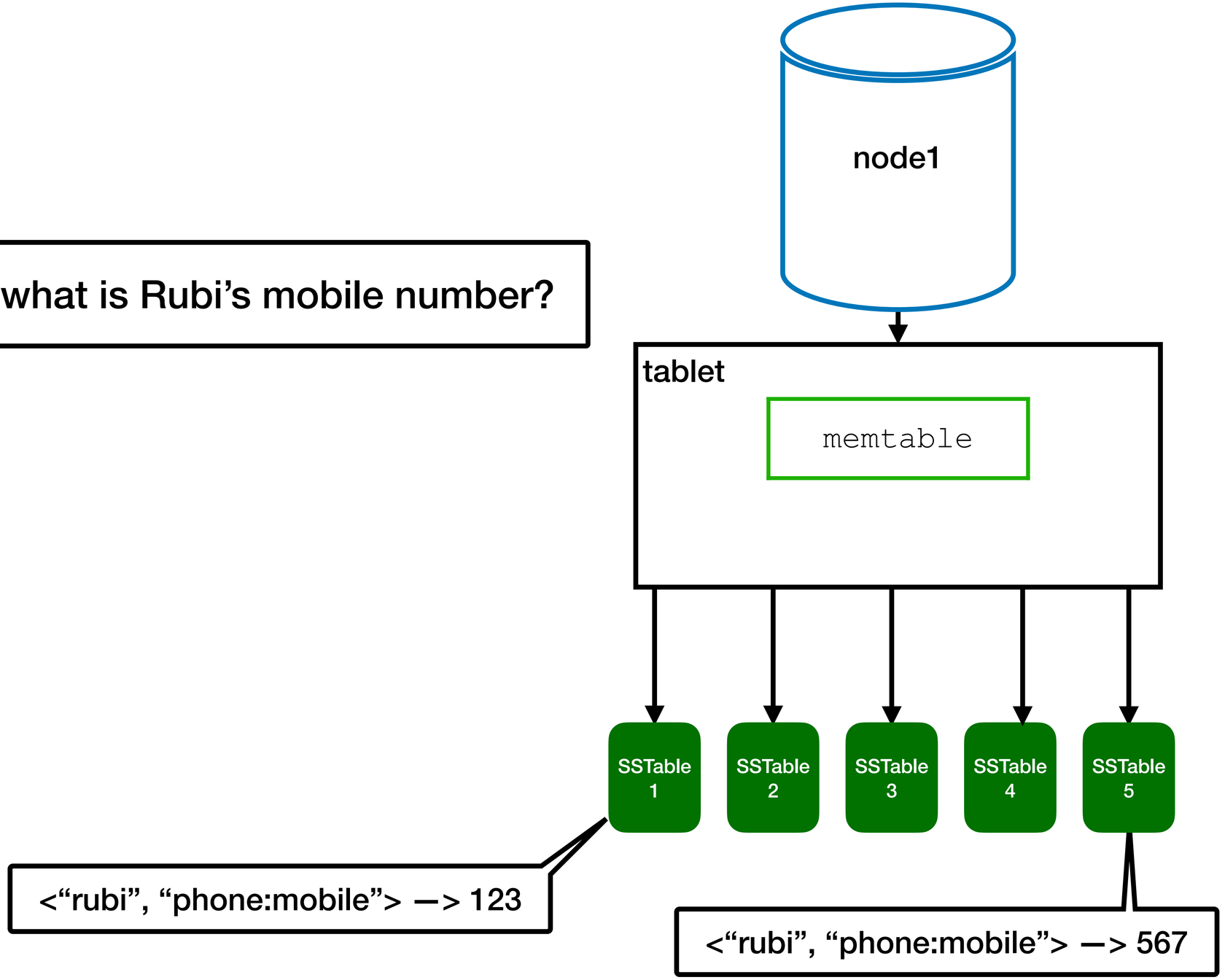

BULLS
23


node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
SSTable 5
<“rubi”, “phone:mobile”> —> 123
<“rubi”, “phone:mobile”> —> 567

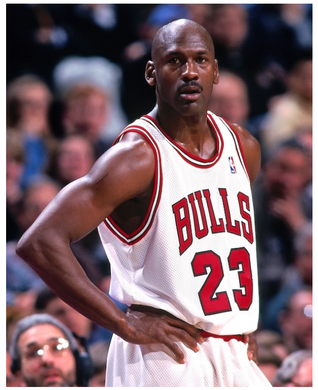
Example



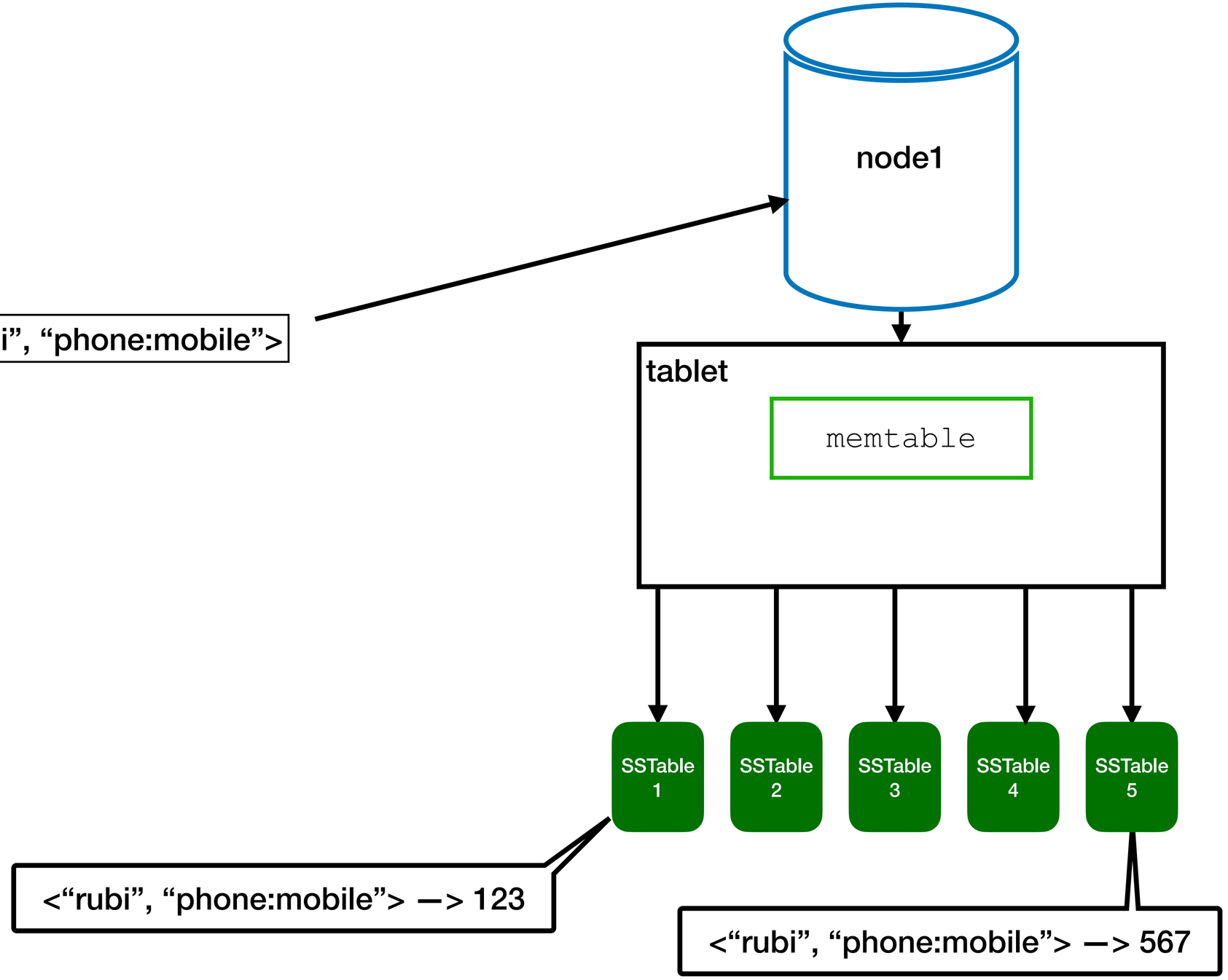
So what is Rubi's mobile number?



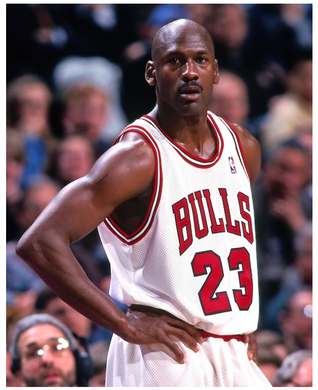
Example



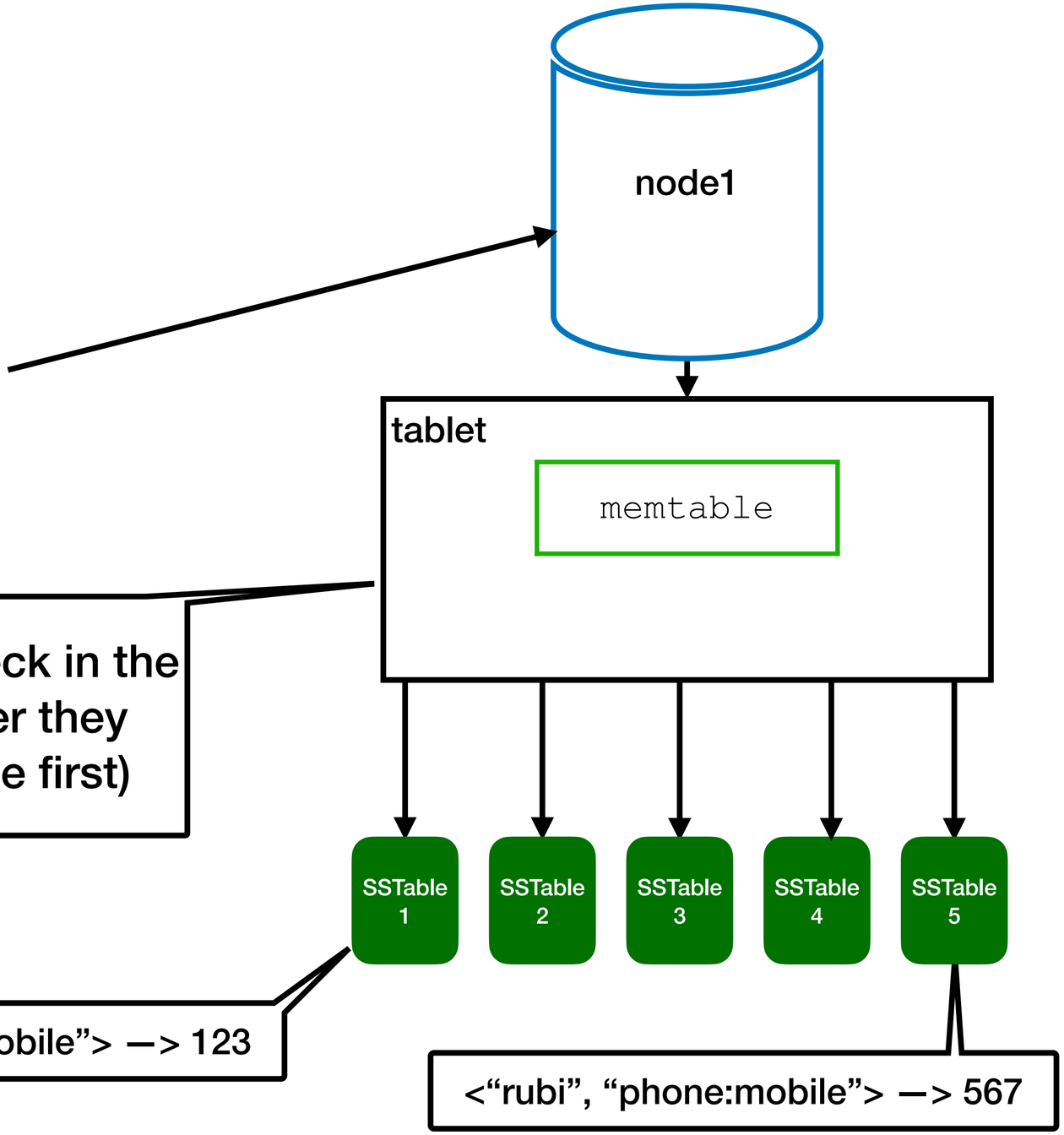
query: <“rubi”, “phone:mobile”>



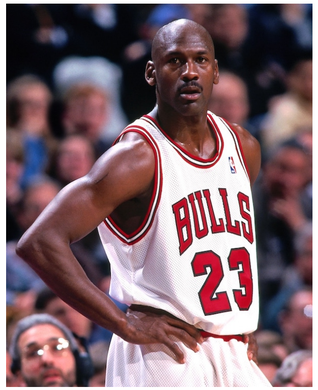
Example



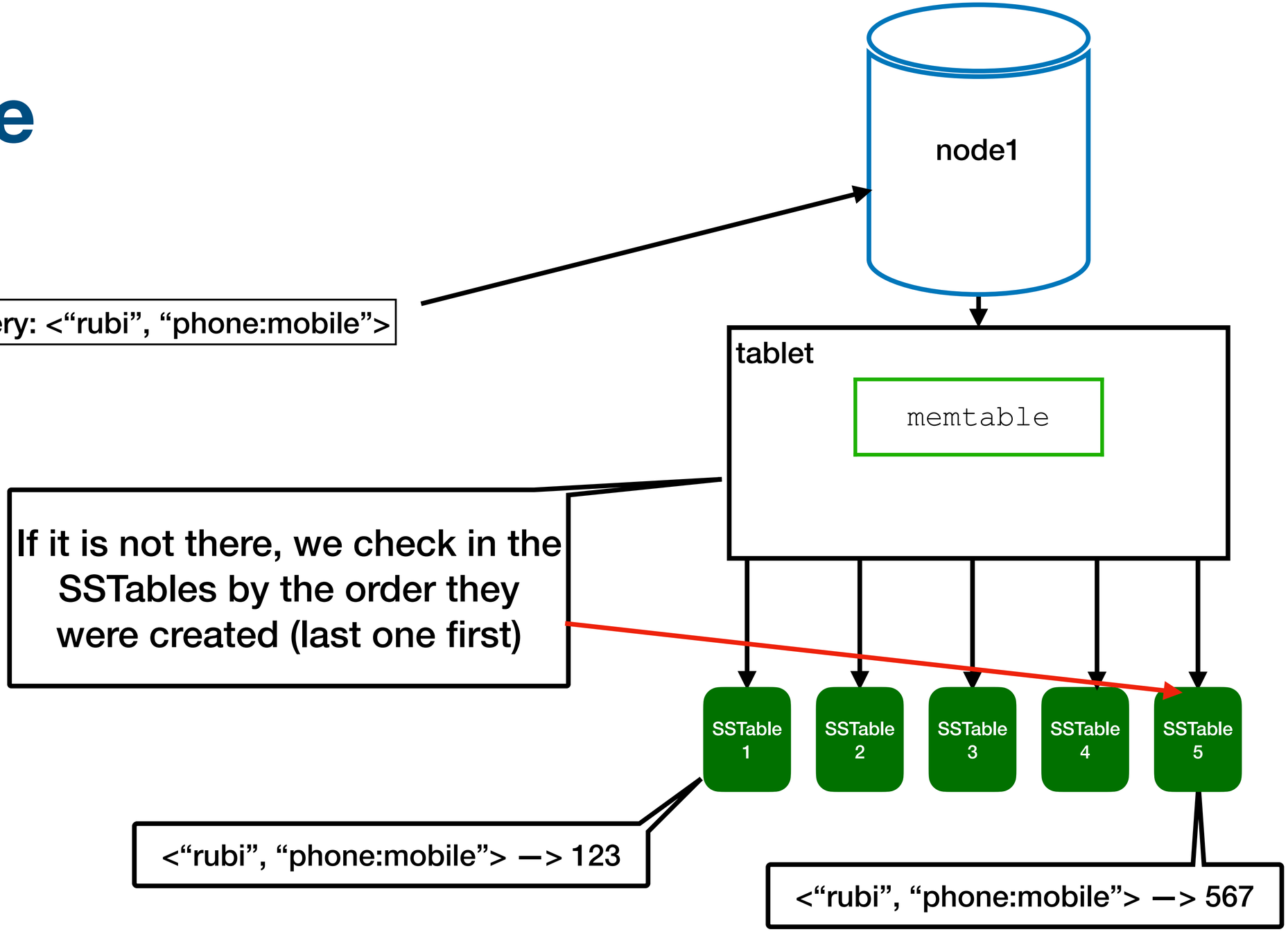
query: <“rubi”, “phone:mobile”>



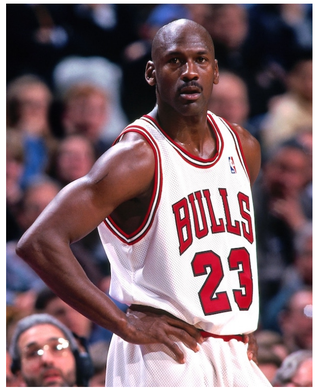
Example



query: <“rubi”, “phone:mobile”>



Example



Note - we do NOT need to read more SSTables - we should return the most recent value

query: <“rubi”, “phone:mobile”>

node1

tablet

memtable

If it is not there, we check in the SSTables by the order they were created (last one first)

SSTable 1

SSTable 2

SSTable 3

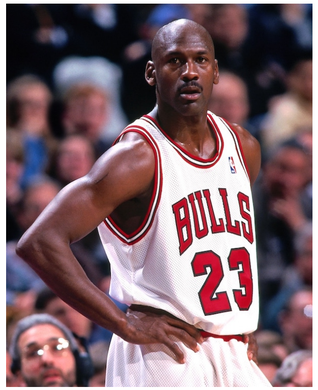
SSTable 4

SSTable 5

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567

Example



query: <“rubi”, “phone:mobile”>

567

node1

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

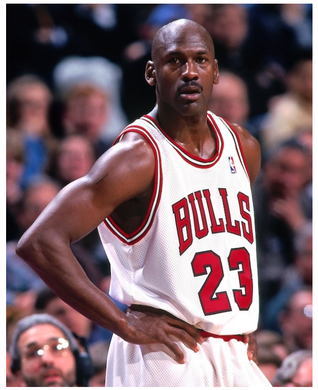
SSTable 4

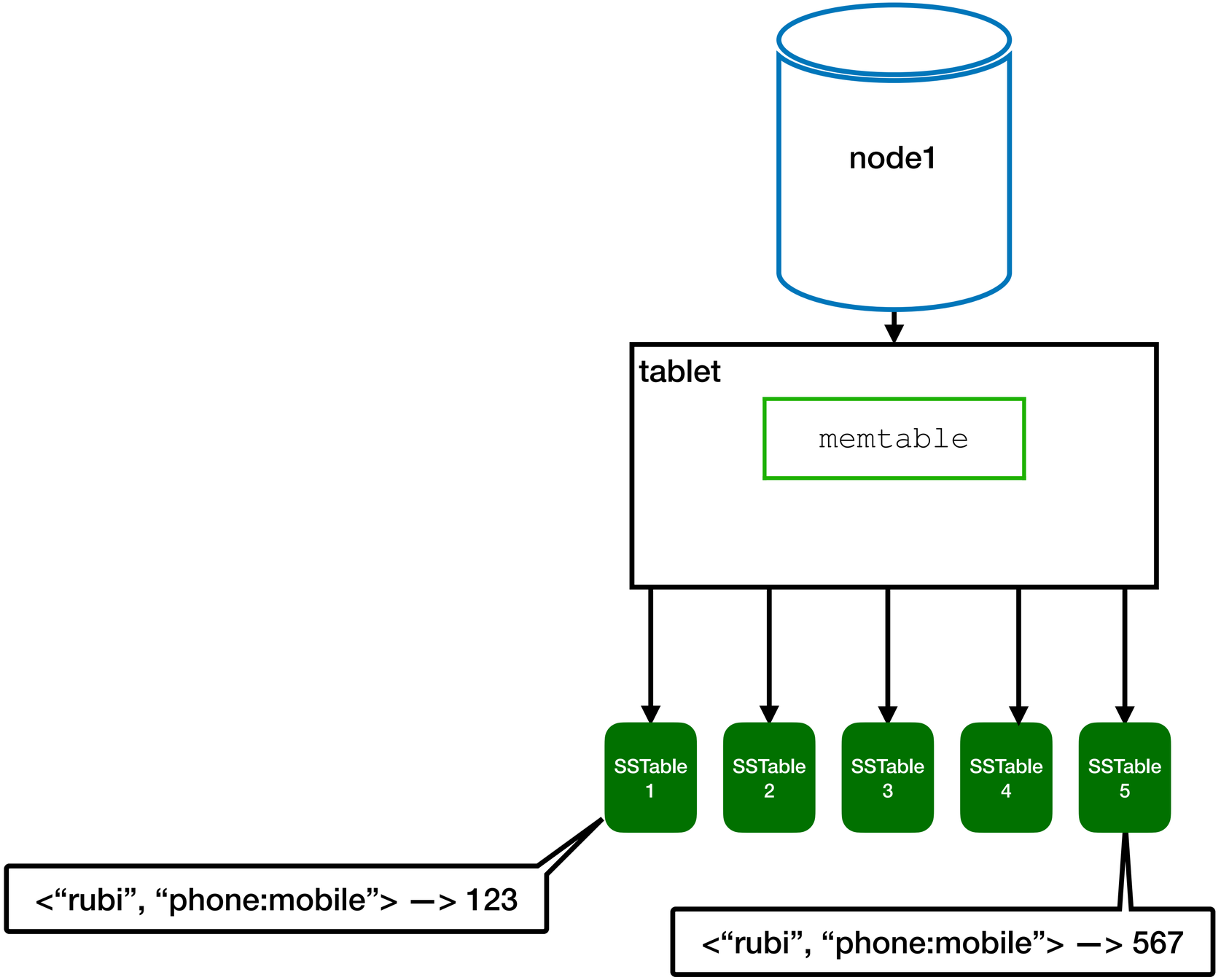
SSTable 5

<“rubi”, “phone:mobile”> —> 123

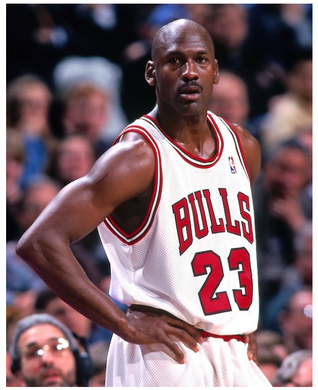
<“rubi”, “phone:mobile”> —> 567

Example


BULLS
23

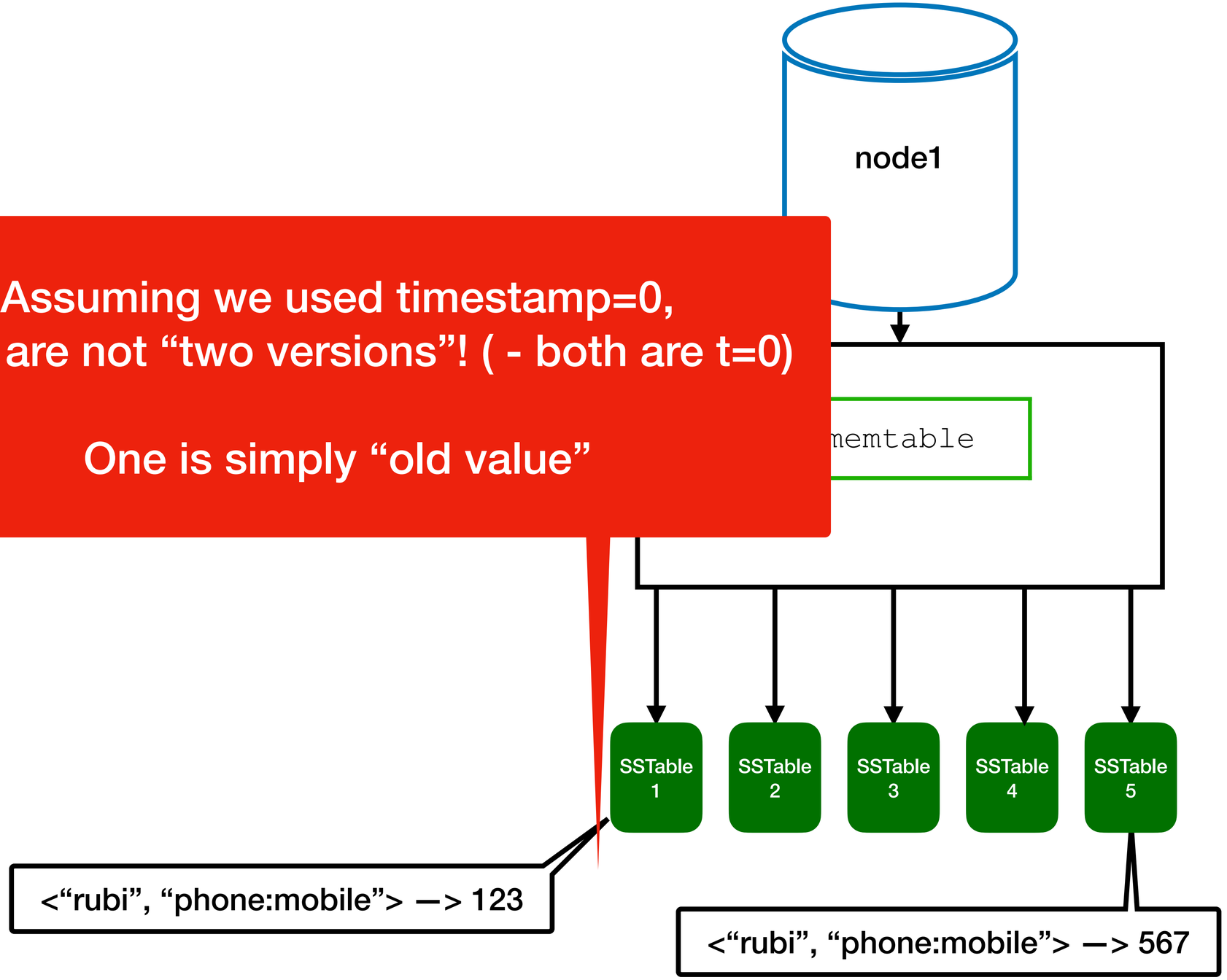

node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
SSTable 5
<“rubi”, “phone:mobile”> —> 123
<“rubi”, “phone:mobile”> —> 567

Example

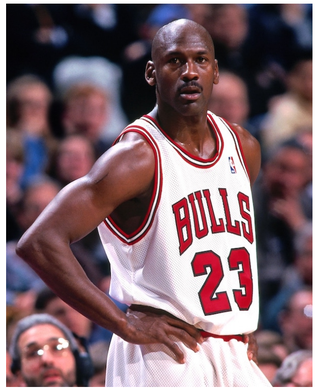


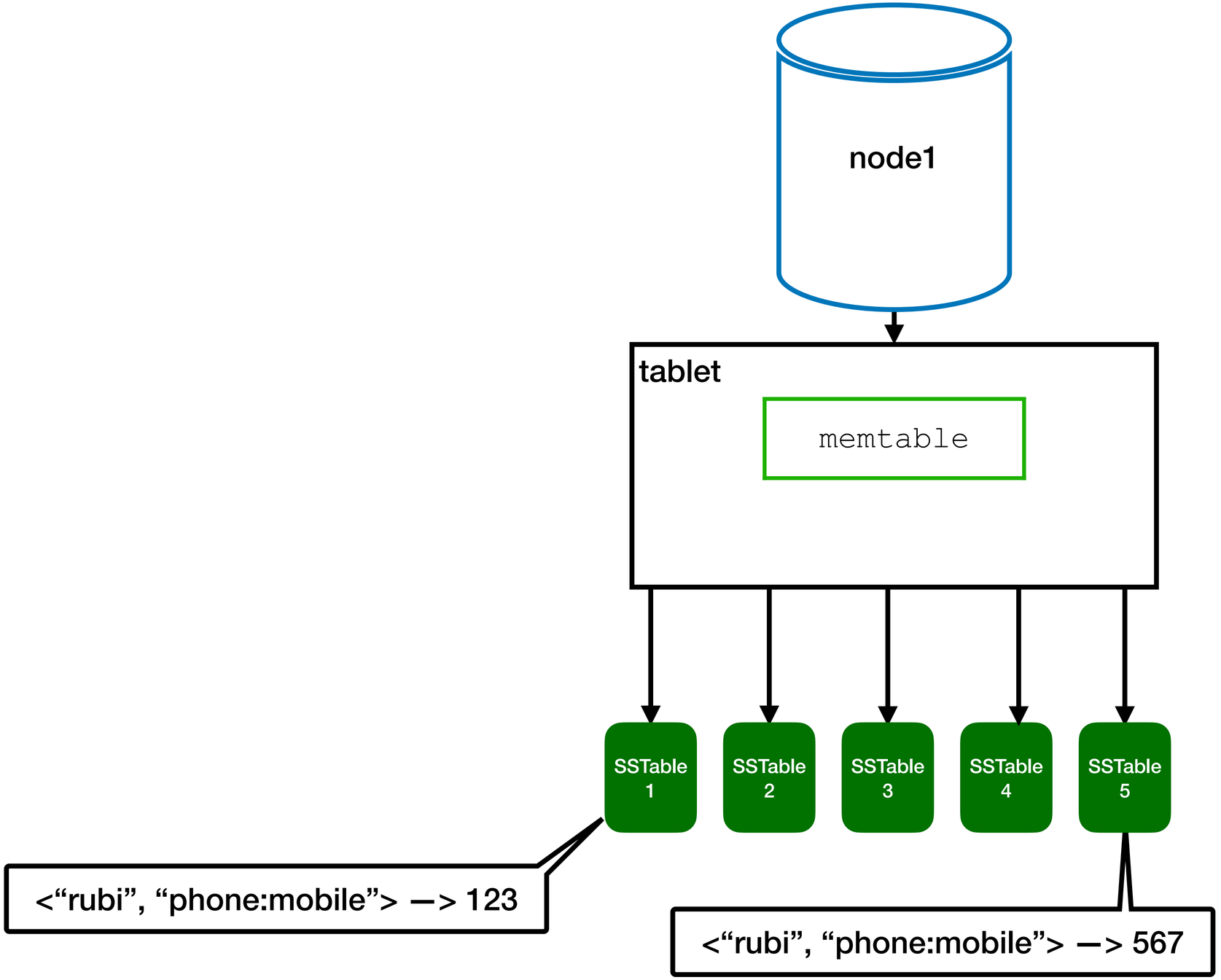
Assuming we used timestamp=0,
these are not “two versions”! (- both are t=0)

One is simply “old value”

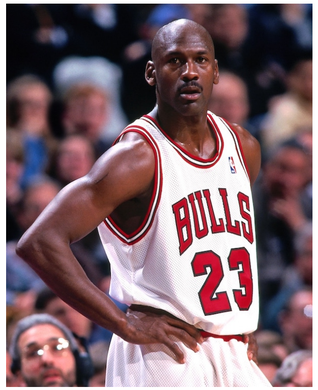


Example

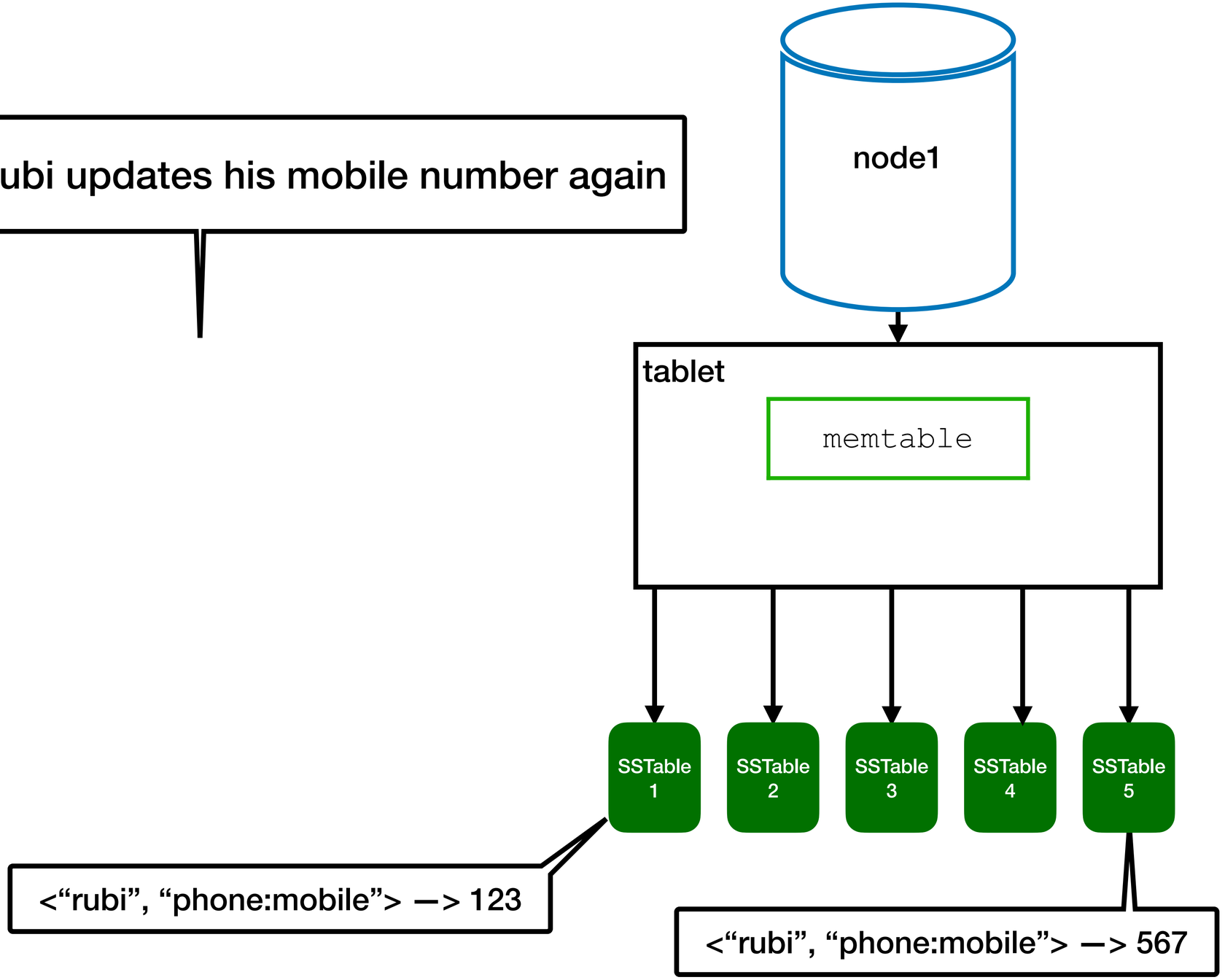

BULLS
23


node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
SSTable 5
<“rubi”, “phone:mobile”> —> 123
<“rubi”, “phone:mobile”> —> 567

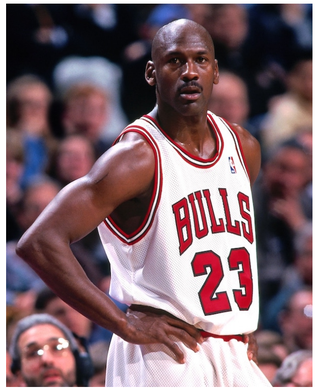
Example



Rubi updates his mobile number again



Example



Rubi updates his mobile number again

node1

tablet

memtable

<“rubi”, “phone:mobile”> —> 890

SSTable 1

SSTable 2

SSTable 3

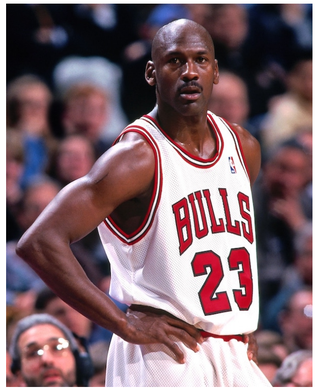
SSTable 4

SSTable 5

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567

Example



Now he deletes his mobile

node1

tablet

memtable

<“rubi”, “phone:mobile”> —> 890

SSTable 1

SSTable 2

SSTable 3

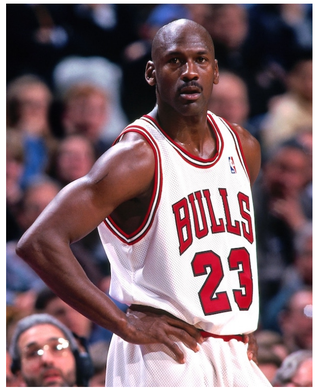
SSTable 4

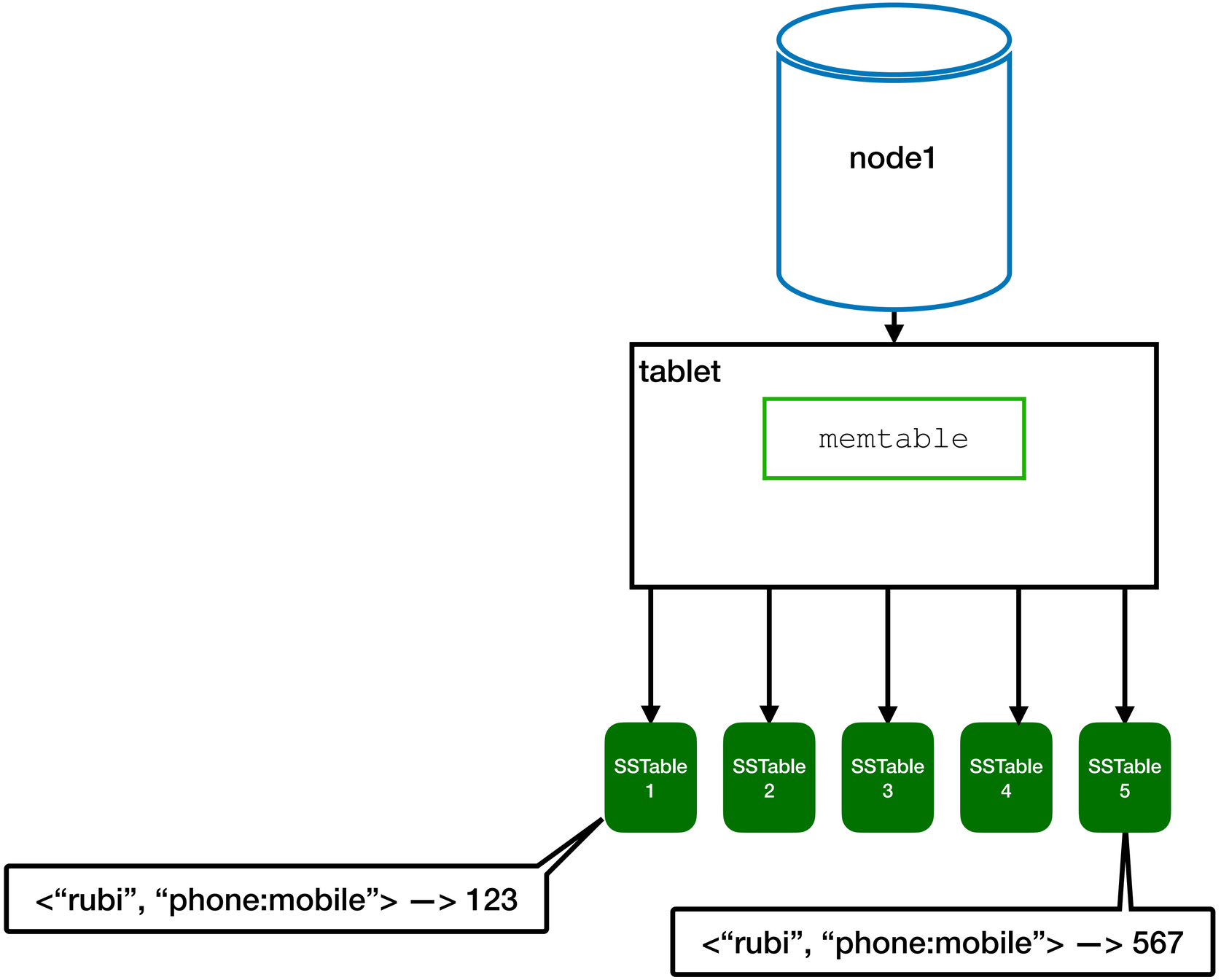
SSTable 5

<“rubi”, “phone:mobile”> —> 123

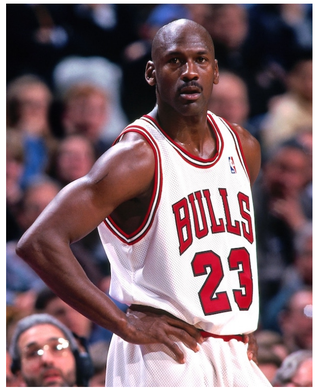
<“rubi”, “phone:mobile”> —> 567

Example

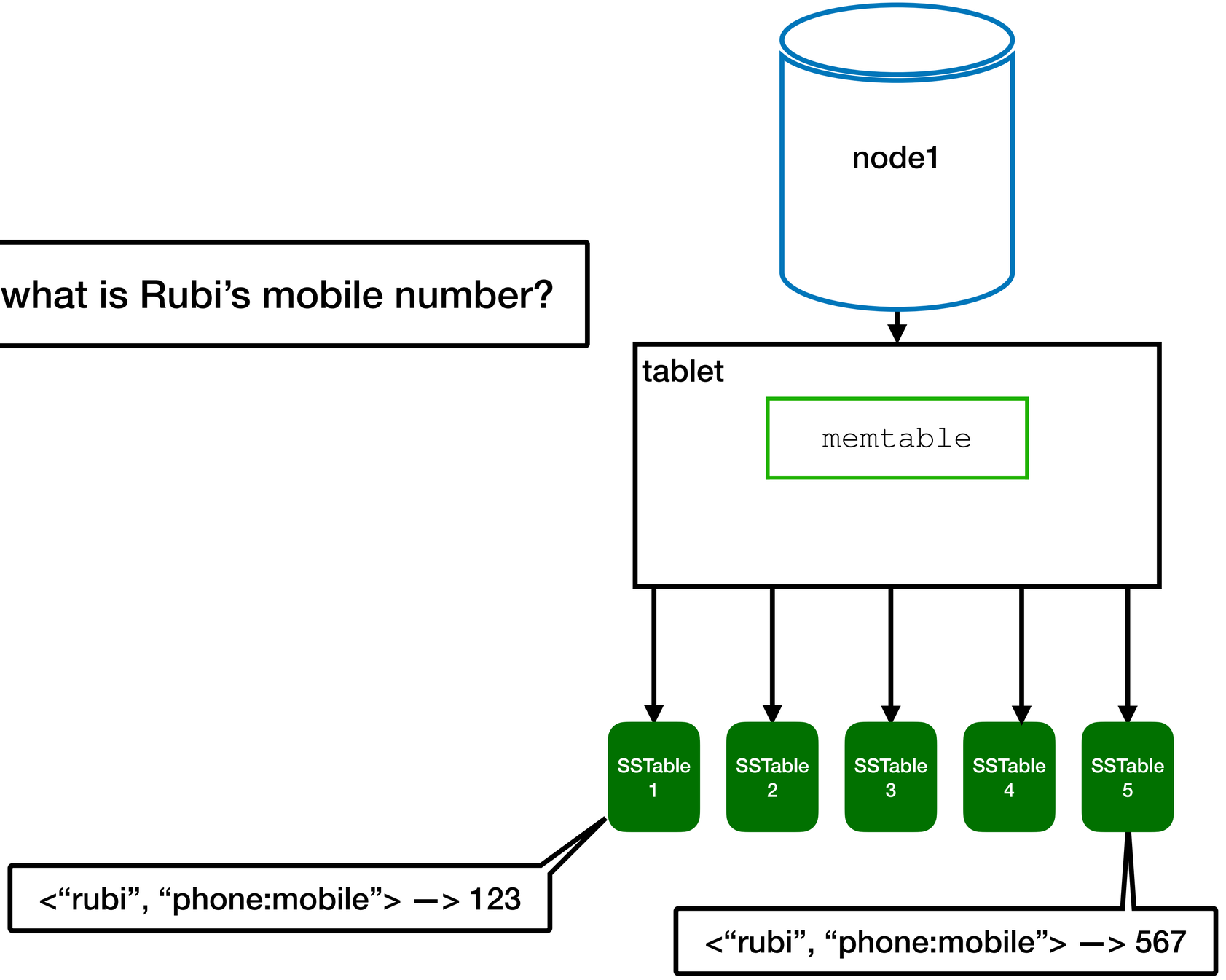

BULLS
23


node1
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
SSTable 5
<“rubi”, “phone:mobile”> —> 123
<“rubi”, “phone:mobile”> —> 567

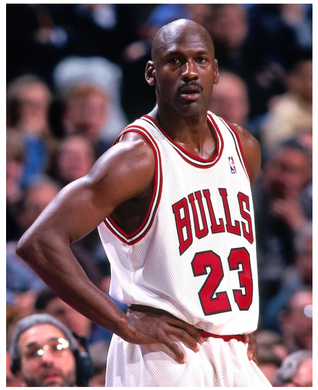
Example



So what is Rubi's mobile number?



Example



query: <“rubi”, “phone:mobile”>

567

node1

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

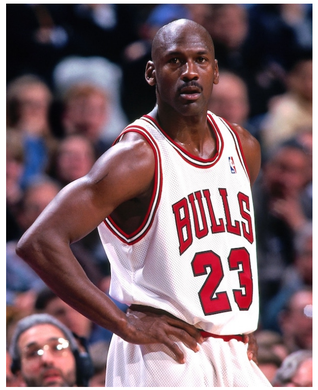
SSTable 4

SSTable 5

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567

Example



query: <“rubi”, “phone:mobile”>

567

BUG?

node1

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

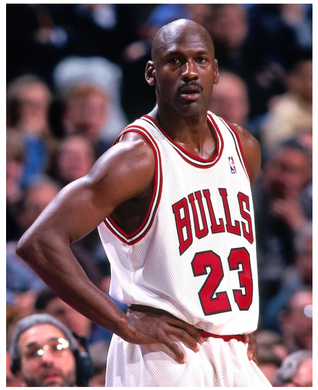
SSTable 4

SSTable 5

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567

Example



query: <“rubi”, “phone:mobile”>

567

BUG?
NO!
In deletes we “insert” a flag
(tombstone)

node1

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

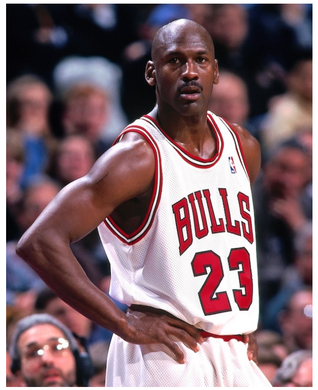
SSTable 4

SSTable 5

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567

Example



Now he deletes his mobile

node1

tablet

memtable

<“rubi”, “phone:mobile”> —> tombstone

SSTable 1

SSTable 2

SSTable 3

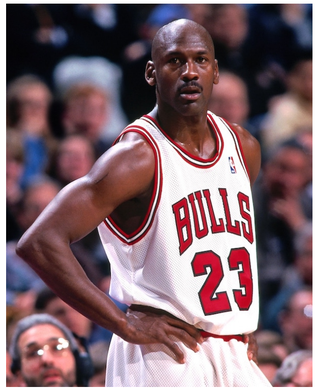
SSTable 4

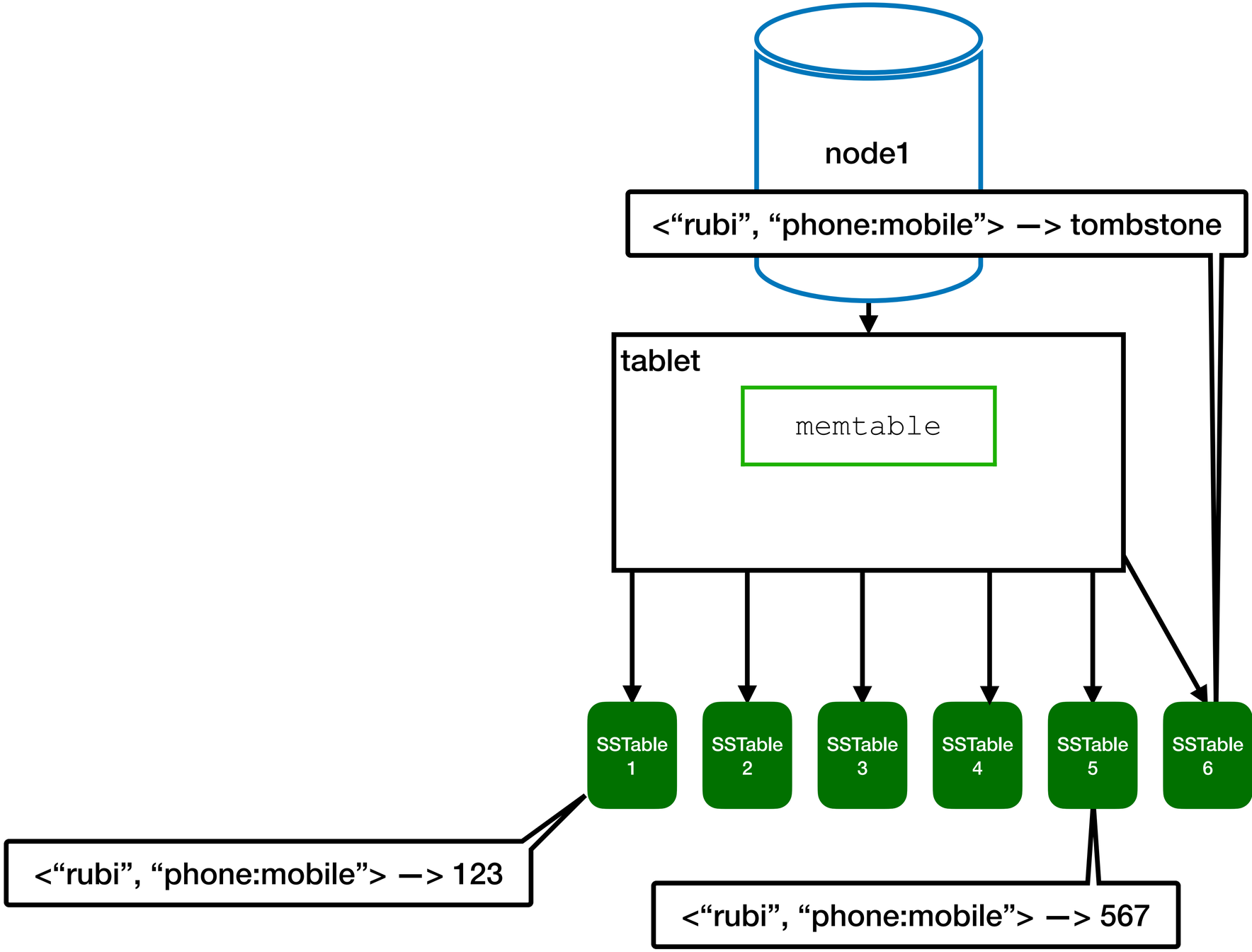
SSTable 5

<“rubi”, “phone:mobile”> —> 123

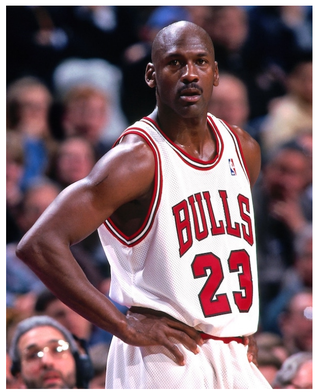
<“rubi”, “phone:mobile”> —> 567

Example


BULLS
23


node1
<“rubi”, “phone:mobile”> —> tombstone
tablet
memtable
SSTable 1
SSTable 2
SSTable 3
SSTable 4
SSTable 5
SSTable 6
<“rubi”, “phone:mobile”> —> 123
<“rubi”, “phone:mobile”> —> 567

Example



So what is Rubi's mobile number?

node1

<"rubi", "phone:mobile"> —> tombstone

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

SSTable 4

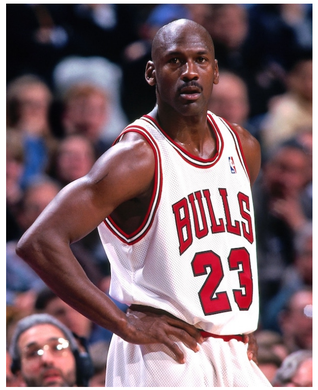
SSTable 5

SSTable 6

<"rubi", "phone:mobile"> —> 123

<"rubi", "phone:mobile"> —> 567

Example



query: <“rubi”, “phone:mobile”>

No results

node1

<“rubi”, “phone:mobile”> —> tombstone

tablet

memtable

SSTable 1

SSTable 2

SSTable 3

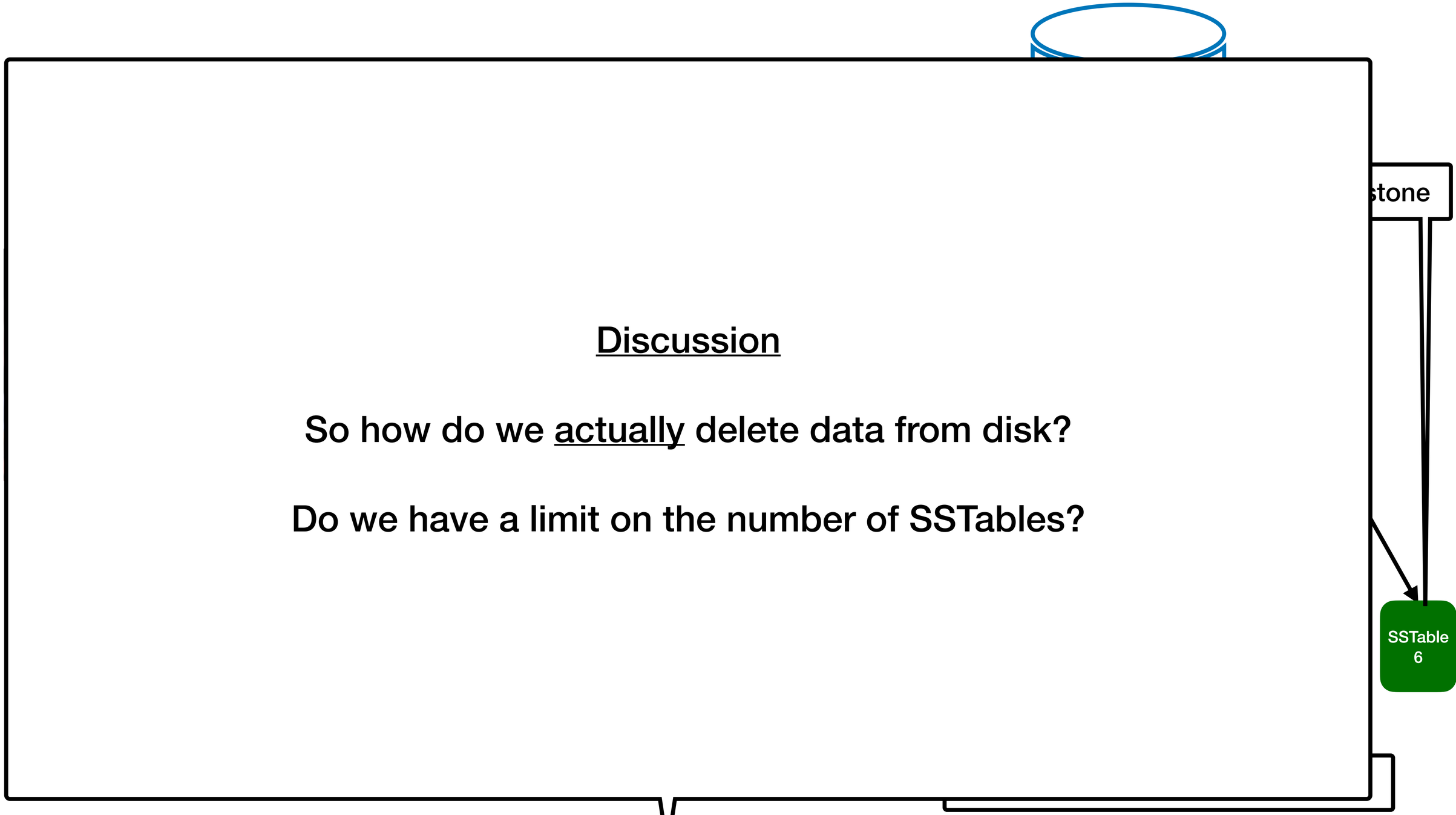
SSTable 4

SSTable 5

SSTable 6

<“rubi”, “phone:mobile”> —> 123

<“rubi”, “phone:mobile”> —> 567



Discussion

So how do we actually delete data from disk?

Do we have a limit on the number of SSTables?

Minor Compaction

The process of saving the memtable into an SSTable

- Goals:
  - Shrinks the memory usage of the node
  - Reduce the data that needs to be read from the log on failures

Minor Compaction

The process of saving the memtable into an SSTable

- Goals:
  - Shrinks the memory usage of the node
  - Reduce the data that needs to be read from the log on failures

How many SSTables would we have over time?

Merging Compaction

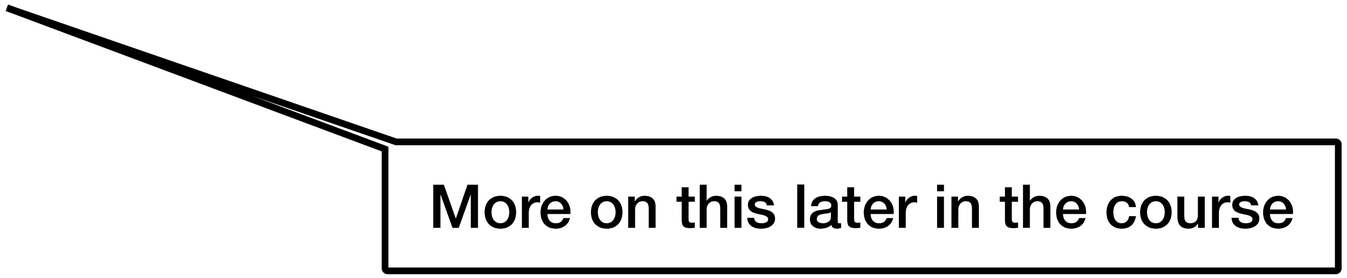
The process of merging two (or more) SSTables into a single new file

- A process that runs automatically in the background
- Optimization - can read also from the memtable
- The old SSTables (and maybe the memtable) can be deleted once merging compaction completes

Major Compaction

The process of merging **all** SSTables into a single new file

- Data is actually deleted only on major compactions
  - before that, deleted values are only flag (by tombstones)



A note on Compaction

- It is a “background” process

Process runs on Bigtable node

- So why do we (as users) should care?

- **Because it dramatically affects the performance**

More on this later in the course

Agenda

- History
- Data model
- Building blocks
- SSTable (and memtable)
- **Bloom filter**
- Summary
- Extra - Chubby
- Extra - Tablet location

Bloom filters

- Probabilistic data structure that used to test whether an element is a member of a set

- If the filter returns true - the element is present with high probability, but not 100% (false positive)
- If the filter return false - the element is NOT in the set

Bloom filters in Bigtable

- A read operation may read from all SSTables of a tablet
  can you think of an example?

- If these SSTables (indexes) are not cached, a lot of disk access may happen

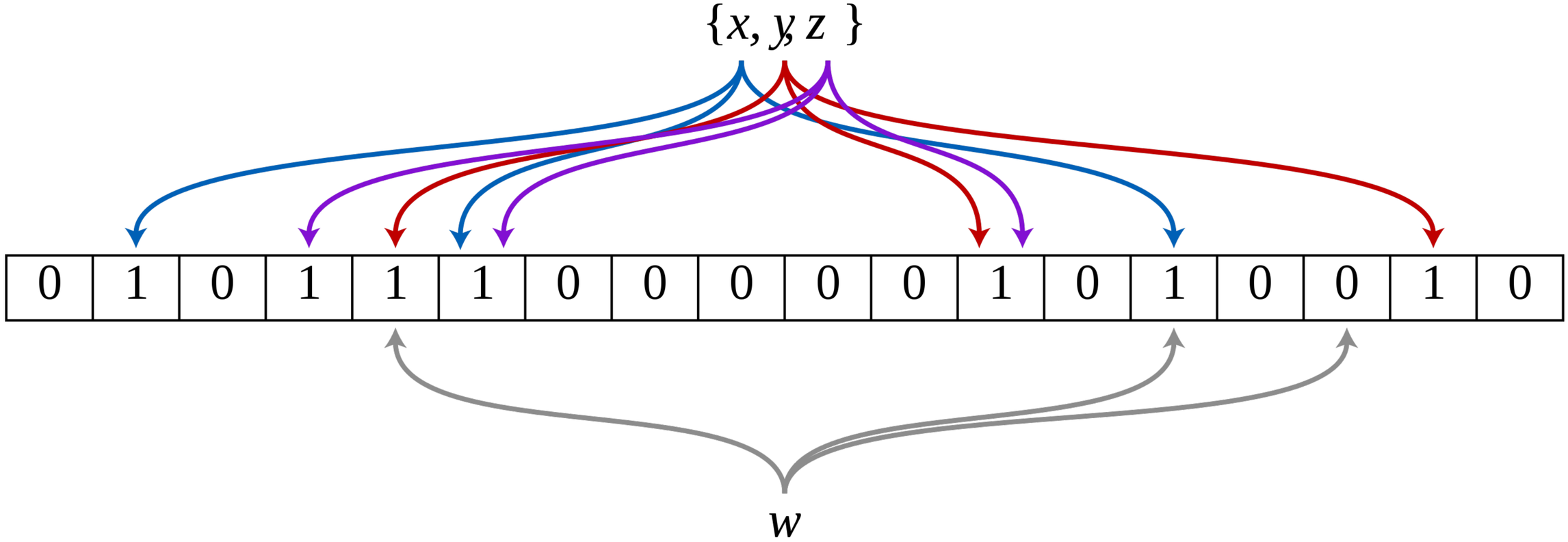
- To reduce these IOs, Bigtable uses Bloom filters for each SSTable (and keep them in memory) to reduce the number of IOs

Bloom filters - how they work

- Initialize (0) an array of m bits
- There are k different hash functions of the range [0, m-1]
- For every element added to the set, apply the k hash functions and mark the matching bits in the array
- To check if an element exists, run the k hash functions and check the matching bits
  - If all are flagged, return true.
  - If any of the bits are 0, return false

Bloom filters - example

- m=18, k=3



Agenda

- History
- Data model
- Building blocks
- SSTable (and memtable)
- Bloom filter
- **Summary**
- Extra - Chubby
- Extra - Tablet location

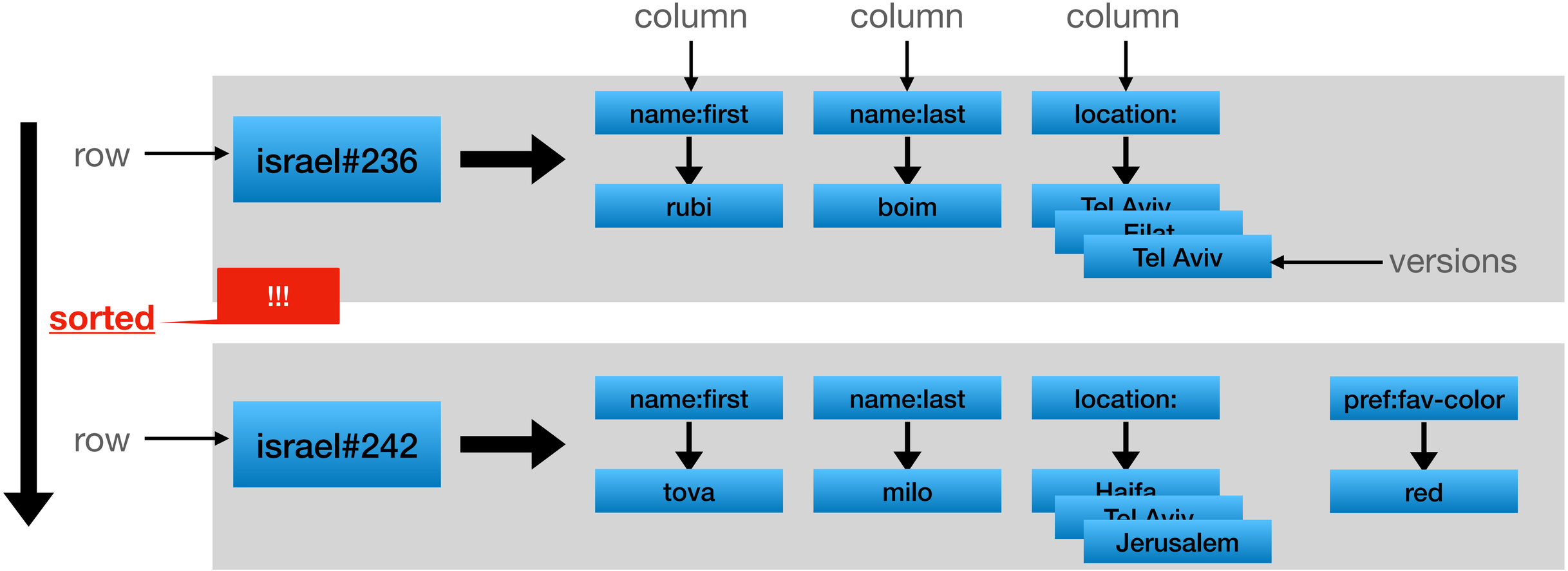
Bigtable

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

<row:string, column:string, timestamp:int64> —> string

- Built on 3 different layers
  - Management (Chubby)
  - Processing (Bigtable nodes)
  - Storage (GFS)

Data model



<row:string, column:string, timestamp:int64> —> string

Schema design points (1)

- Bigtable is a key/value store, not relational
  no joins, atomic operation only within a single row

- Each table has only one index, the row key
  no secondary indexes

- Rows are sorted lexicographically by row key
  from the lowest to the highest byte string

Schema design points (2)

- Column families are not stored in any specific order.

- Columns are grouped by column family and sorted in lexicographic order within the column family

- The intersection of a row and column can contain multiple timestamped cells
  different versions

Schema design points (3)

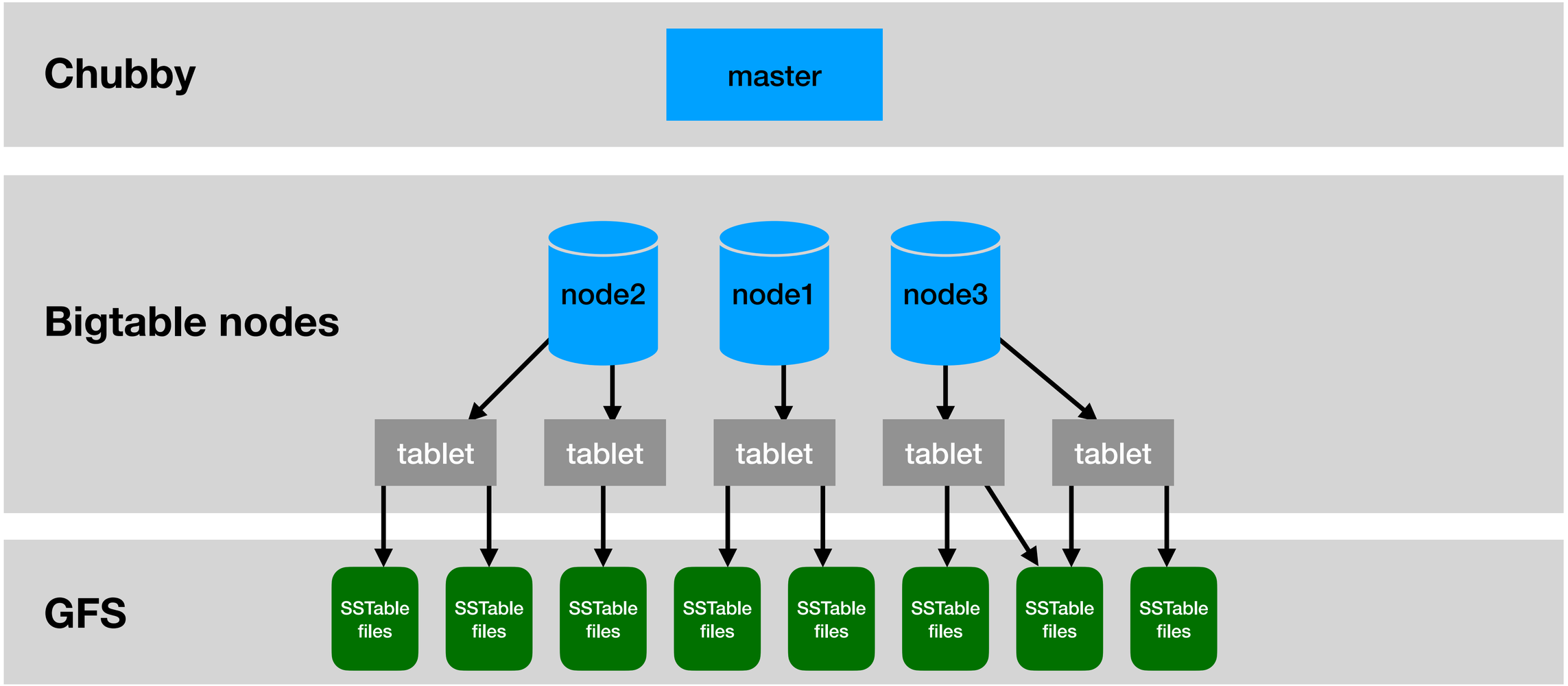
- Ideally, both reads and writes should be distributed evenly
  across the row space of a table

- Bigtable tables are sparse
  A column doesn't take up any space in a row that doesn't use the column

Agenda

- History
- Data model
- Building blocks
- SSTable (and memtable)
- Bloom filter
- Summary
- **Extra - Chubby**
- Extra - Tablet location

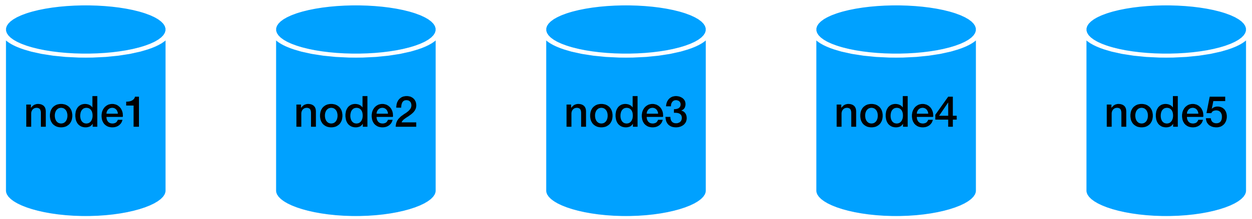
Reminder - Components by layers



Chubby

A **highly available** and persistent **distributed lock** service

- 5 servers, uses the PAXOS algorithm for consistency



- **Provides a namespace for directories and small files**
- API for read/write (atomic) and locks
  on directories / files

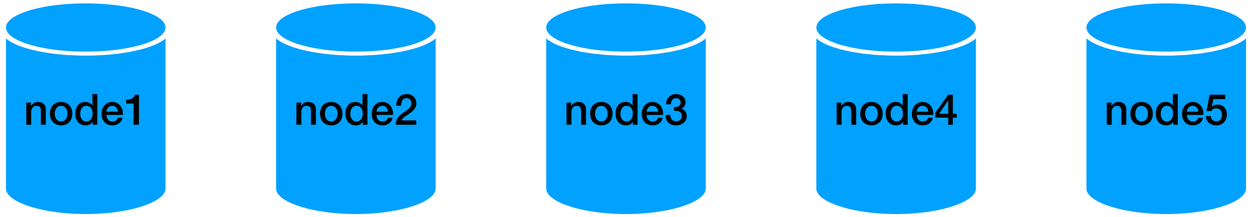
Chubby

Opensource:

APACHE ZooKeeper™

A **highly available** and persistent **distributed lock** service

- 5 servers, uses the PAXOS algorithm for consistency



- **Provides a namespace for directories and small files**
- API for read/write (atomic) and locks
  on directories / files

Chubby - Bigtable usage

Bigtable uses chubby to:

1. Select a node (from Chubby) as **Master**
   - this is done by creating a “lock” on a fixed file
2. Stores bootstraps data (new cluster/table)
3. Stores schema data (table / column families)
4. Discover / manage Bigtable nodes
   - There is a directory “servers” and each server has a matching file with a lock
   - As long as the lock is active, the server is live
   - If the sessions with Chubby is lost, the lock is released and the Bigtable server is considered down

Chubby - Bigtable usage

Bigtable uses chubby to:

1. Select a node (from Chubby) as **Master**
   - this is done by creating a “lock” on a fixed file
2. Stores bootstraps data (new cluster/table)
3. Stores schema data (table / column families)
4. Discover / manage Bigtable nodes
   - There is a directory “servers” and each server has a matching file with a lock
   - As long as the lock is active, the server is live
   - If the sessions with Chubby is lost, the lock is released and the Bigtable server is considered down

If Chubby becomes unavailable for an extended period of time —> Bigtable becomes unavailable

Master node

The master node is responsible to

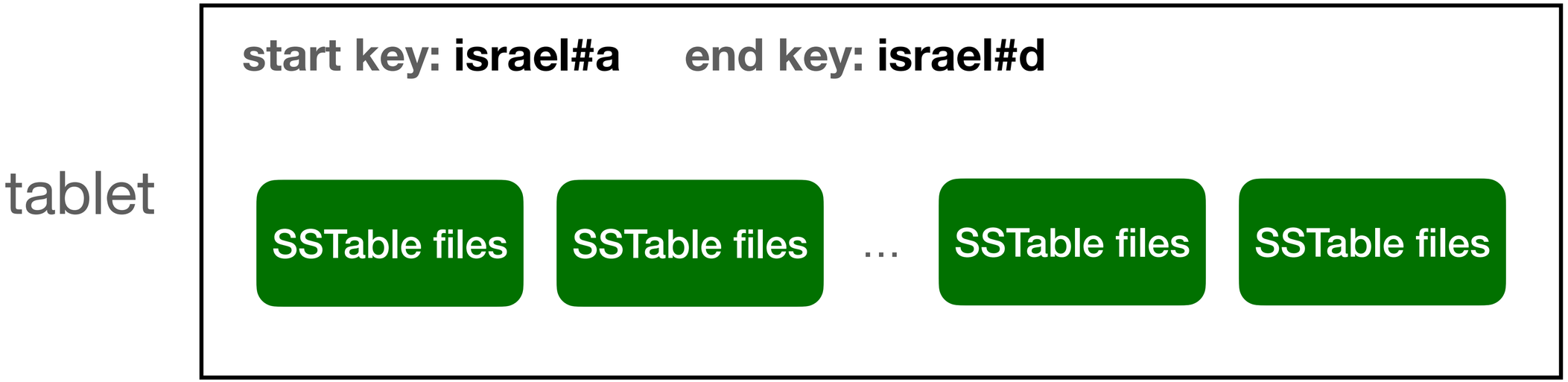
1. Assigning tablets to Bigtable nodes
   root tablet for METADATA table - more on this next
2. Detecting the addition / expiration of Bigtable nodes
3. Balancing Bigtable nodes
   moving tablets
4. Schema management
   tables / column families

Agenda

- History
- Data model
- Building blocks
- SSTable (and memtable)
- Bloom filter
- Summary
- Extra - Chubby
- **Extra - Tablet location**

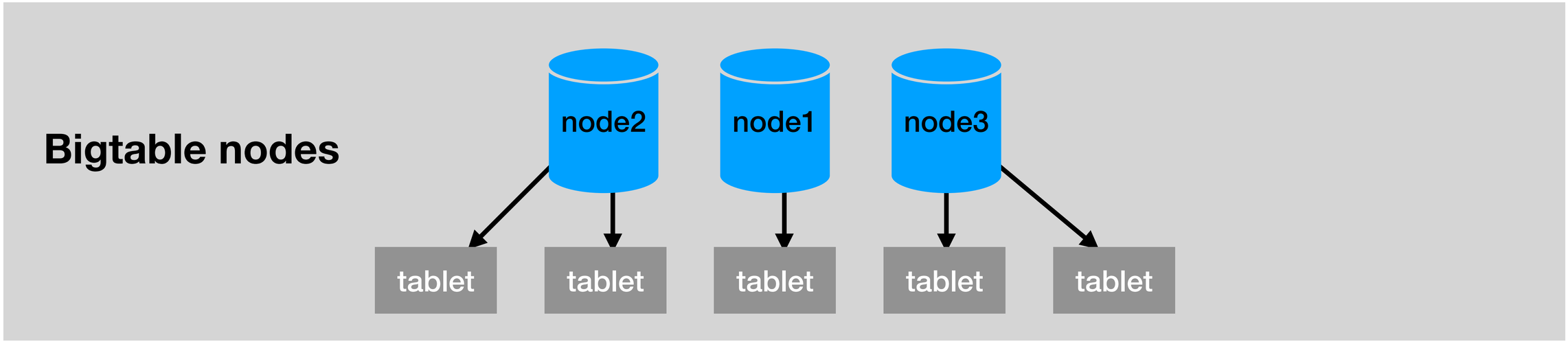
Reminder - Tablet

- A set of SSTables over a matching range comprise a tablet



Tablet location

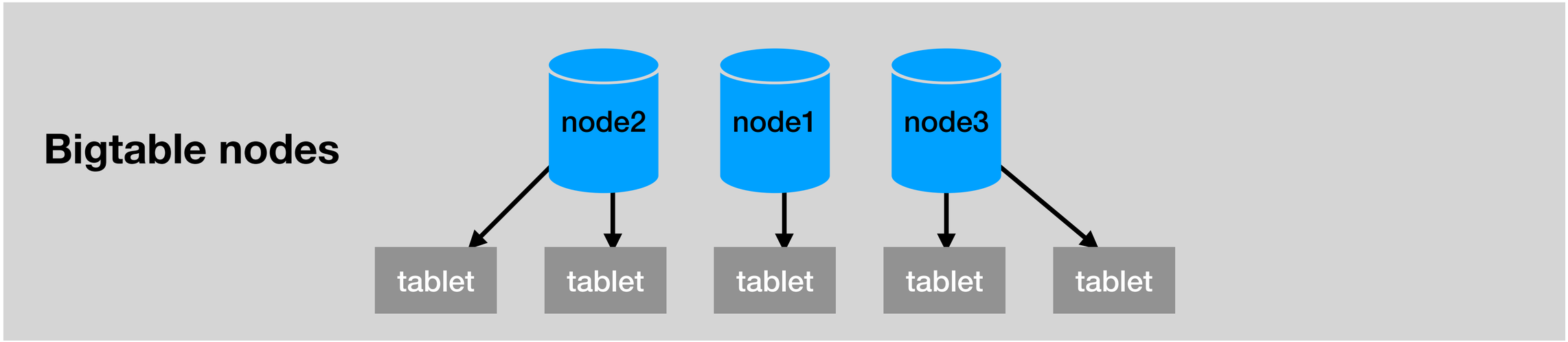
- How Bigtable stores the mapping between tablets and nodes?



Tablet location

For example, where is the tablet for the key “tel-aviv#rubi” for table users?

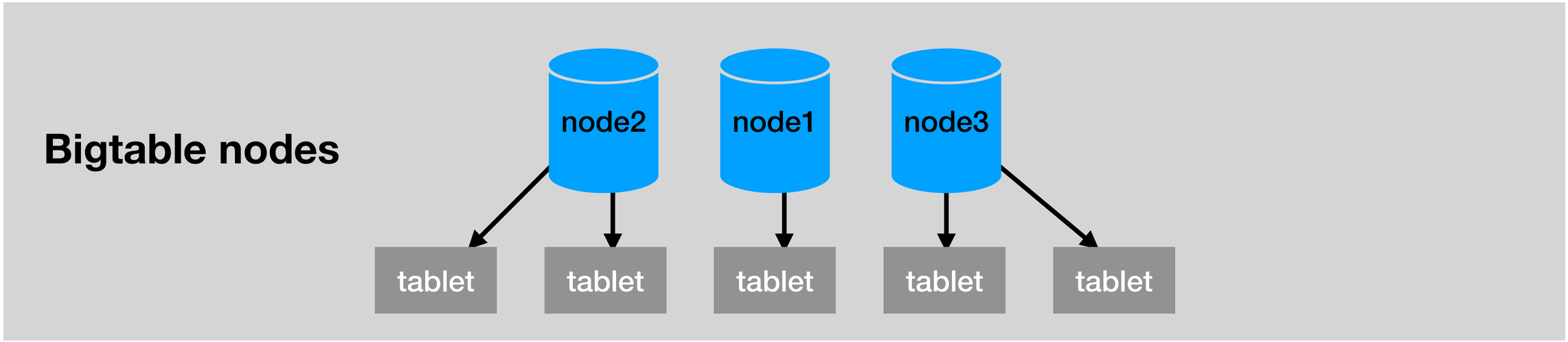
- How Bigtable stores the mapping between tablets and nodes?



Tablet location

For example, where is the tablet for the key “tel-aviv#rubi” for table users?

- How Bigtable stores the mapping between tablets and nodes?
- Using “3-level hierarchy” index similar to B+ trees
  B+ trees are search trees with “a lot of children”

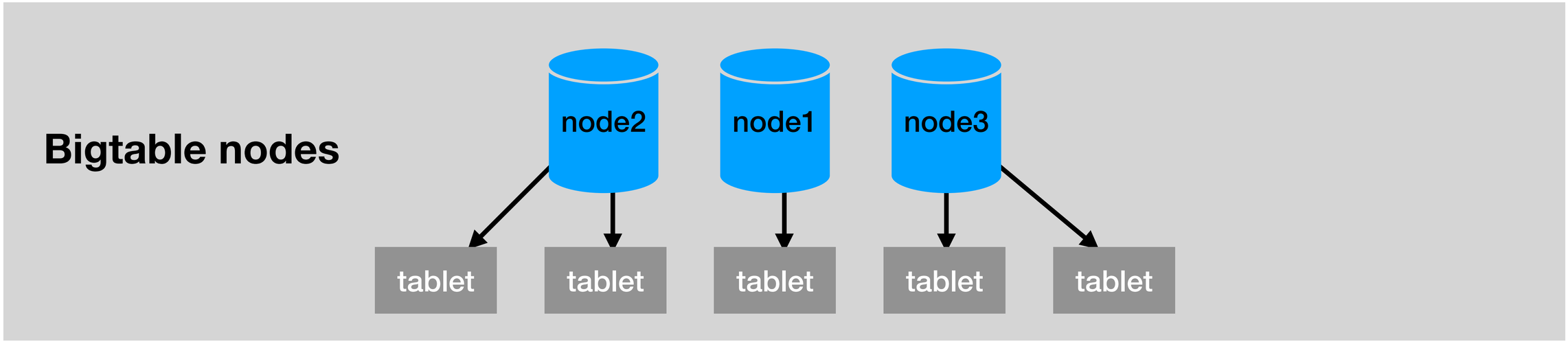


Tablet location

For example, where is the tablet for the key “tel-aviv#rubi” for table users?

High fanout —>
less I/O operation to find element —>
great for indexes

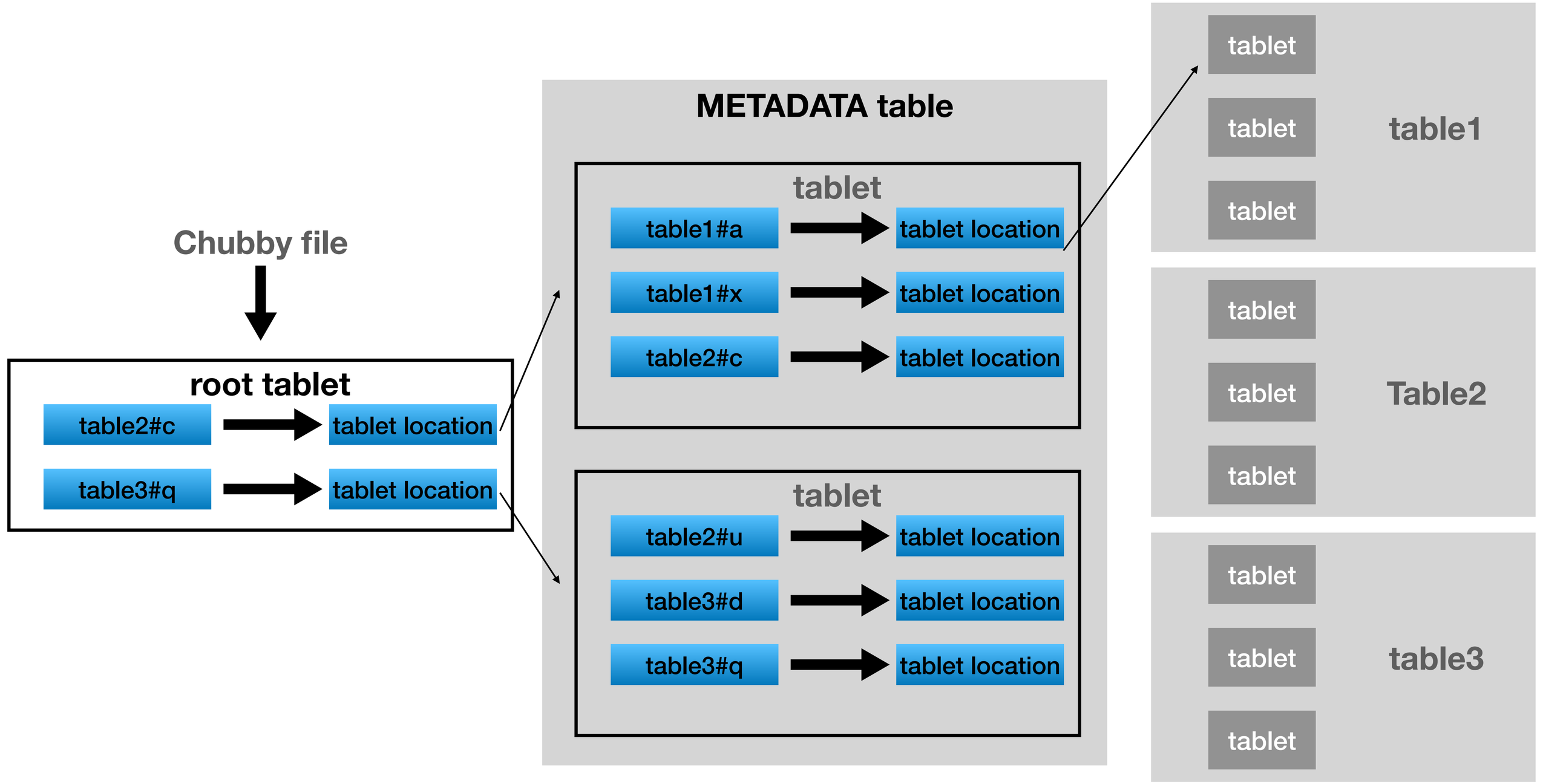
- How Bigtab between tablets and nodes?
- Using “3-level hierarchy” index similar to B+ trees
  B+ trees are search trees with “a lot of children”



This Index is implement by

- A system Bigtable table (METADATA)
  - the row key is [table]#[last range] of a user tablet

- A Chubby file (root tablet)
  - A single file holding the tablet of METADATA tablet
  - It is never split


Chubby file
root tablet
table2#c
tablet location
table3#q
tablet location
METADATA table
tablet
table1#a
tablet location
table1#x
tablet location
table2#c
tablet location
tablet
table2#u
tablet location
table3#d
tablet location
table3#q
tablet location
tablet
tablet
tablet
table1
tablet
tablet
tablet
Table2
tablet
tablet
tablet
table3

Some numbers

- Each METADATA row stores ~1KB
- Assume 128MB per METADATA tablet
  - 2^{17} records per tablet
- 3 level hierarchy - 2^{34} tablets
  - 17,179,869,184 user tablets