

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)
- Chubby
- Tablet location
- Bloom filter
- Summary

Agenda

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

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)
- Chubby
- Tablet location
- Bloom filter
- Summary

Data model - TLDR

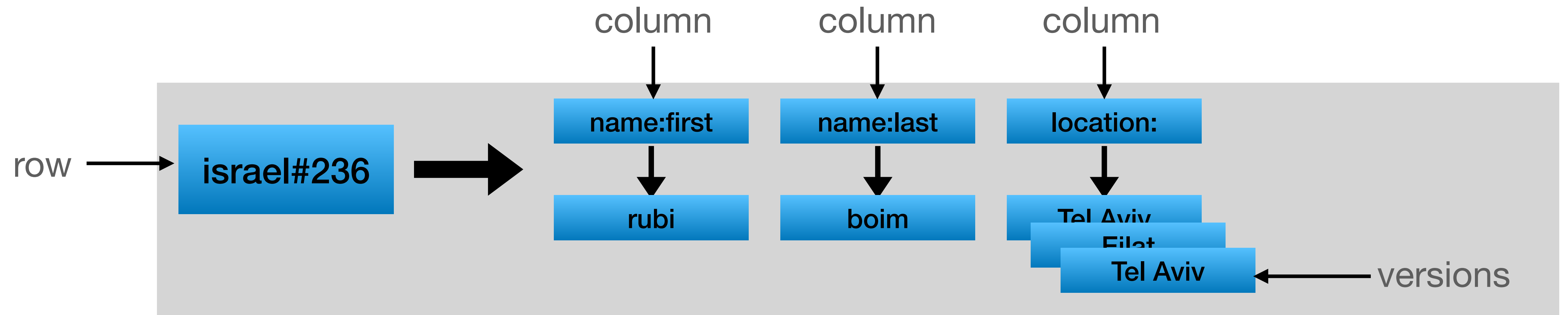
“A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map.”

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

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

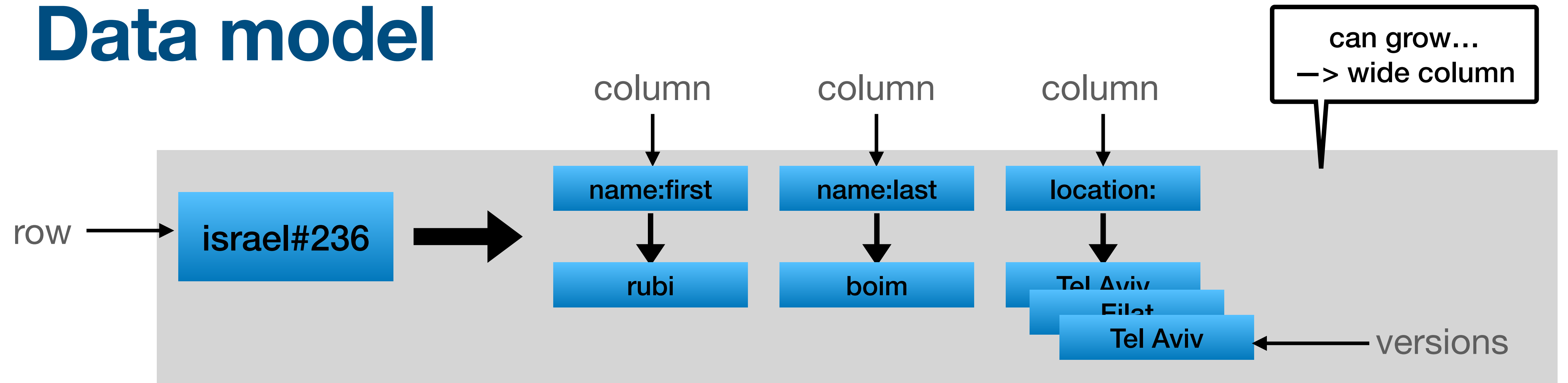
array of bytes

Data model



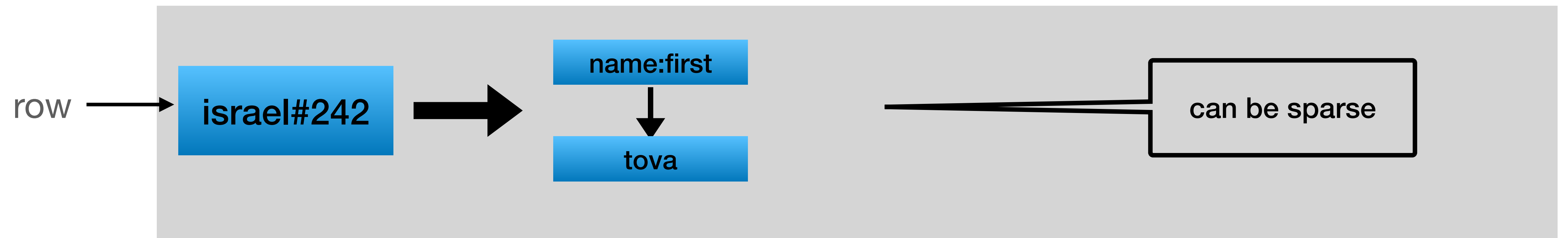
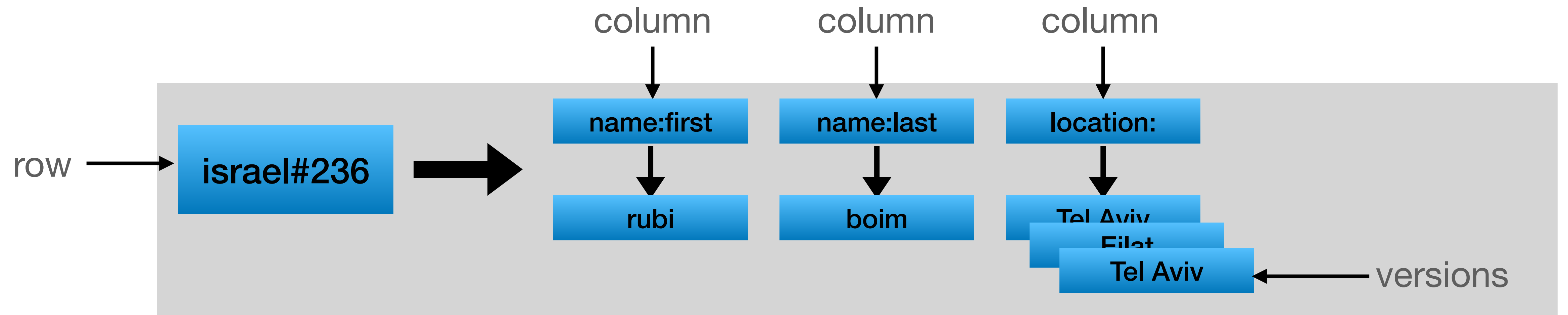
<row:string, column:string, timestamp:int64> -> string

Data model



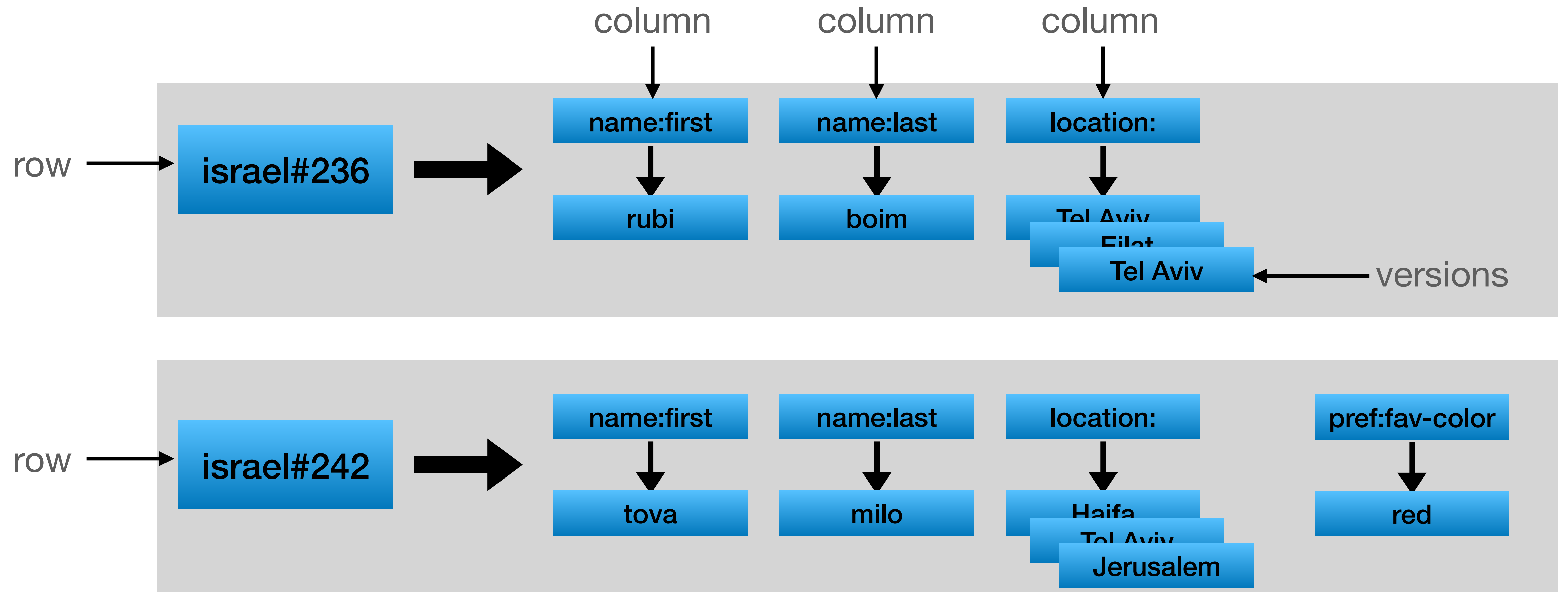
<row:string, column:string, timestamp:int64> -> string

Data model



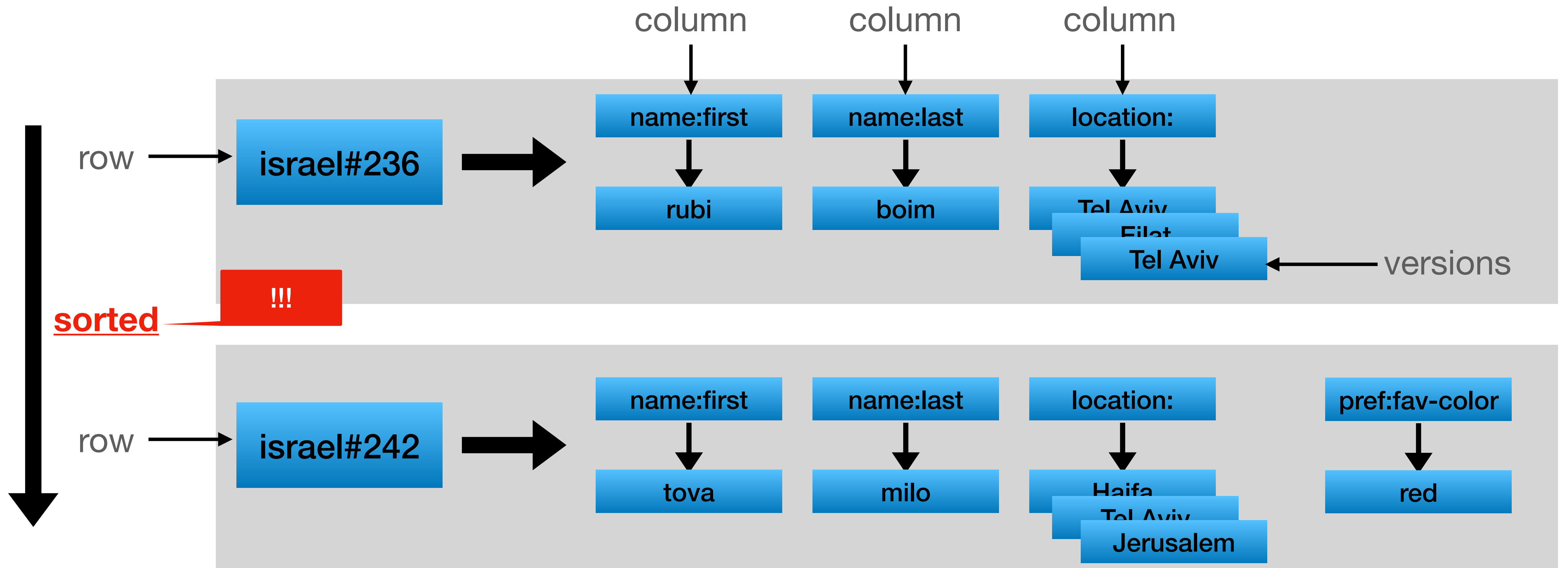
<row:string, column:string, timestamp:int64> -> string

Data model



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

Data model

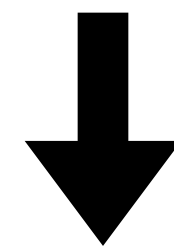


<row:string, column:string, timestamp:int64> -> string

Data model - design

- Discussion - is this model optimal?
- What will happen if we switch the order?

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

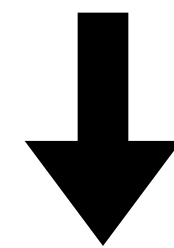


<row:string, timestamp:int64, column:string> -> string

Data model - design

- Discussion - is this model optimal?
- What will happen if we switch the order?

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



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

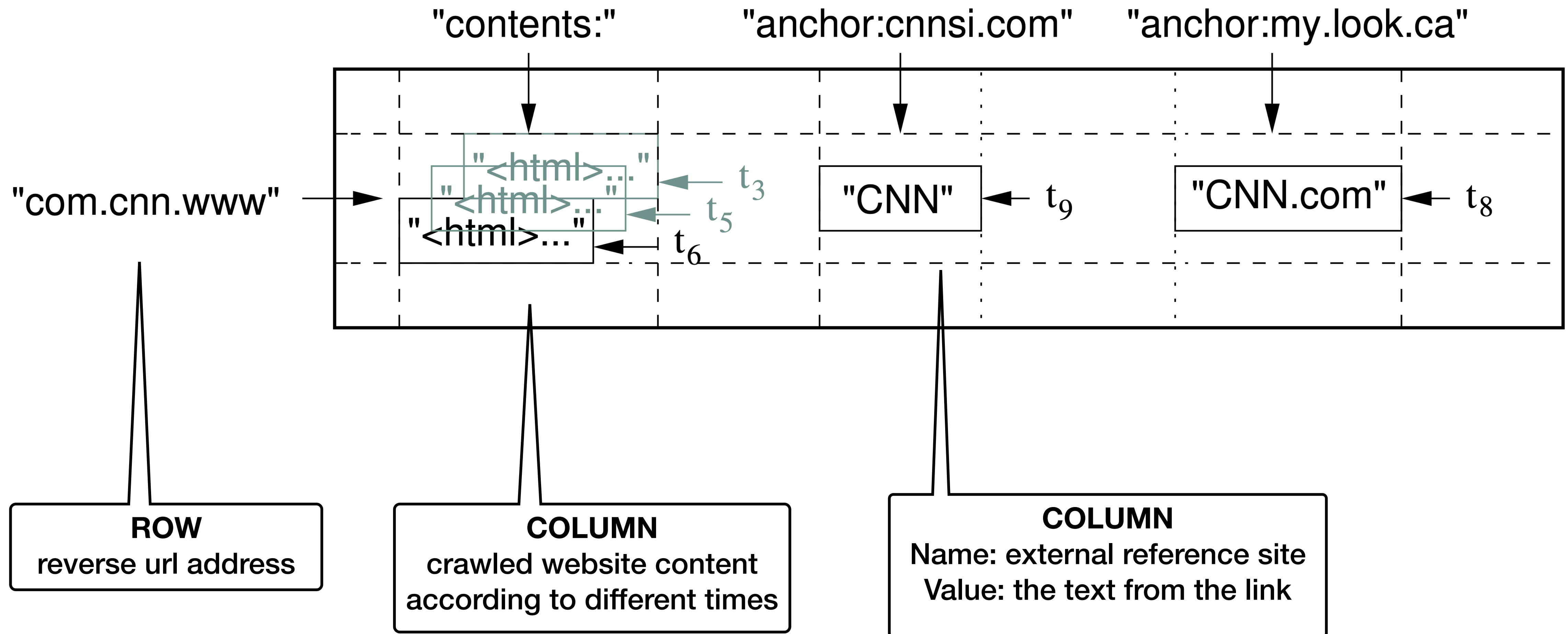
The version will apply to all columns

Data model - Google's requirements

- Bigtable is build by Google **FOR Google...**
- Optimal == Optimal for Google's requirements

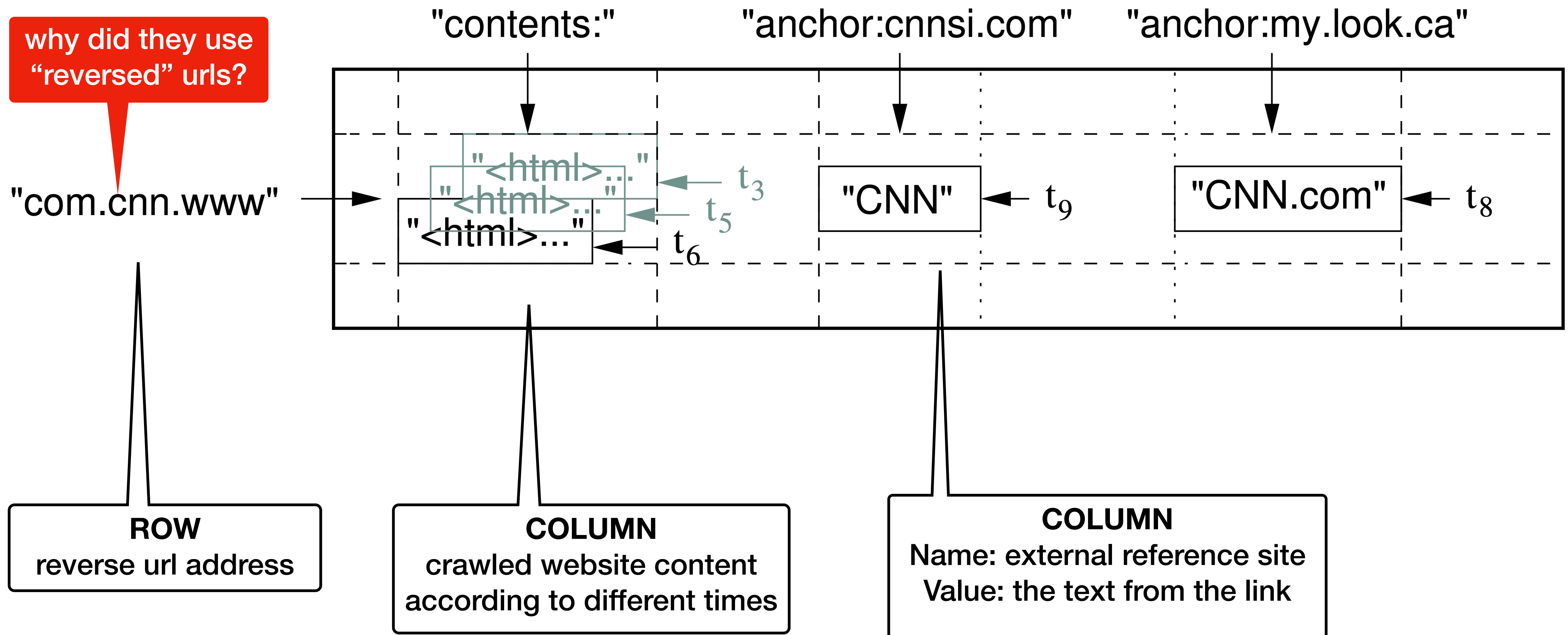
Data model - Webtable example

Used by Google's search index



Data model - Webtable example

Used by Google's search index

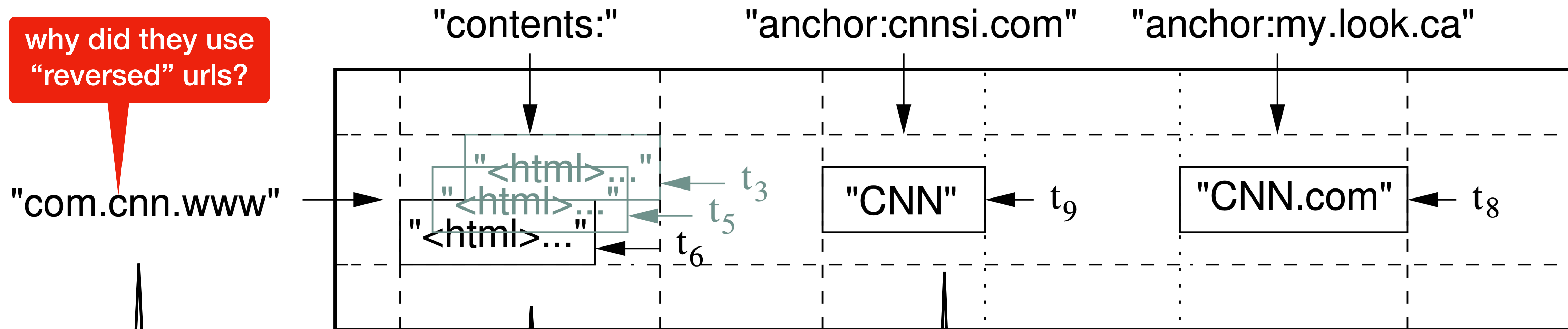


Data model - Webtable example

data is stored on column name

Used by Google's search index

why did they use "reversed" urls?



ROW
reverse url address

COLUMN
crawled website content according to different times

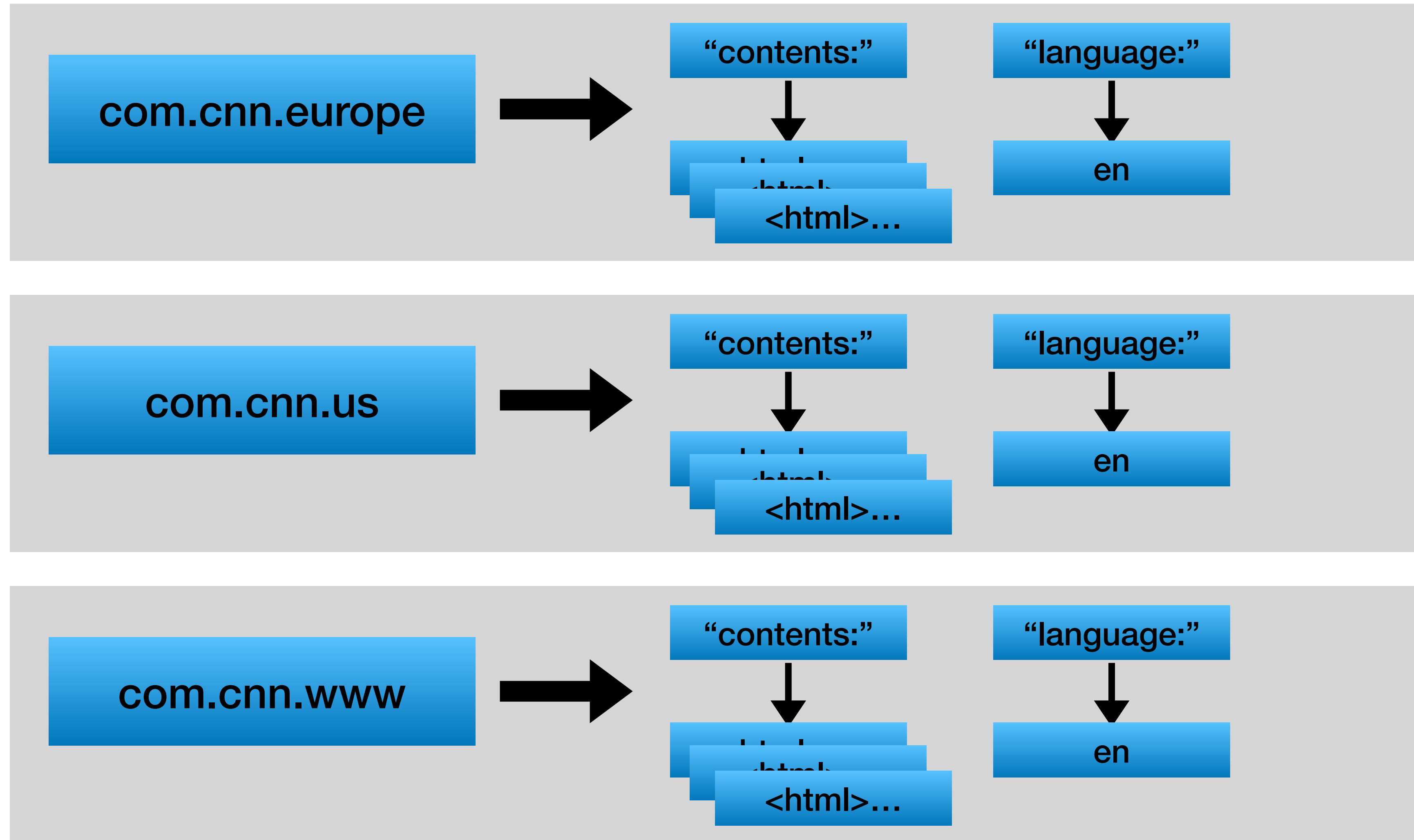
COLUMN
Name: external reference site
Value: the text from the link

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

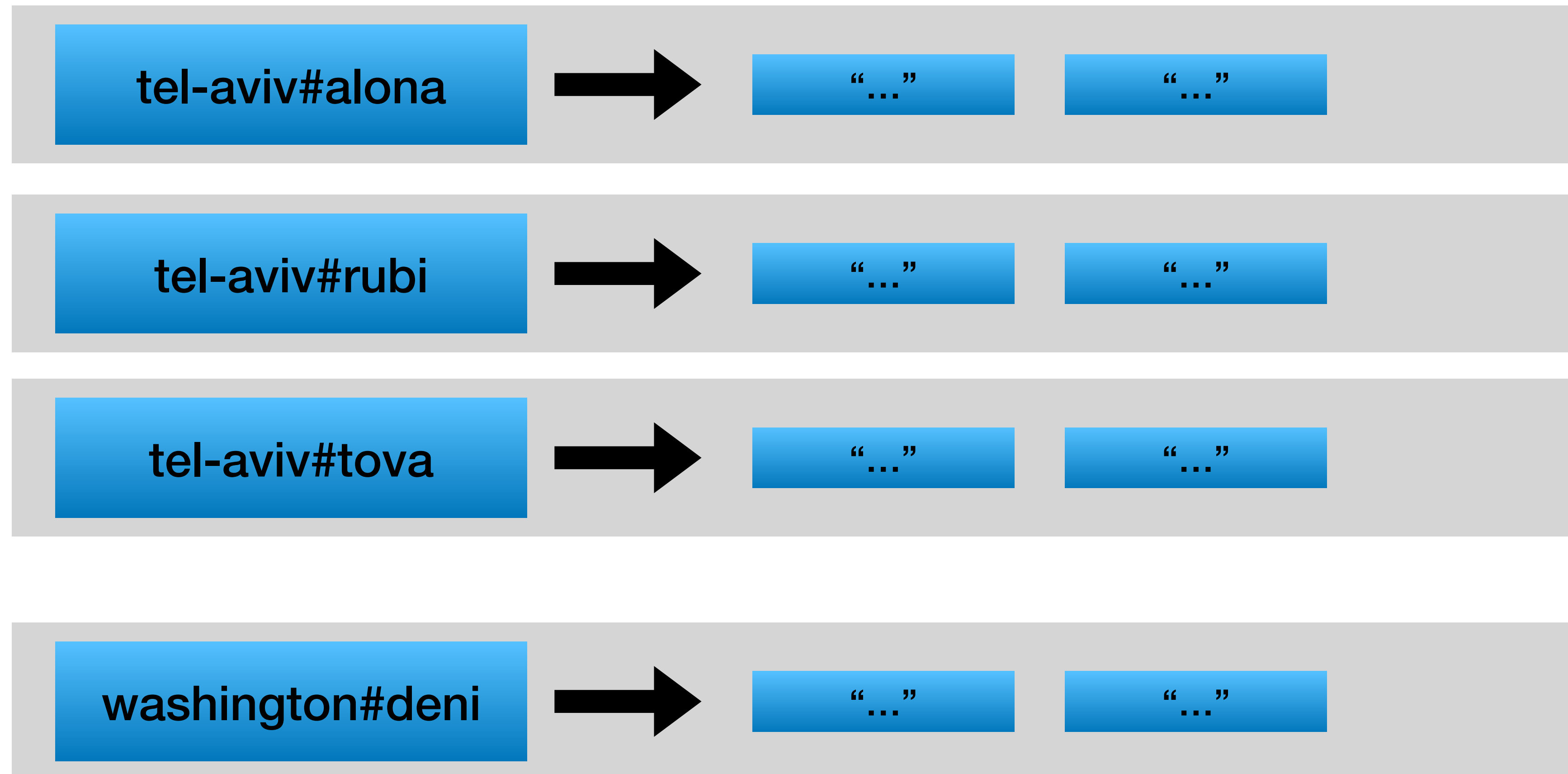
Model the data based on how data is accessed



Rows - range




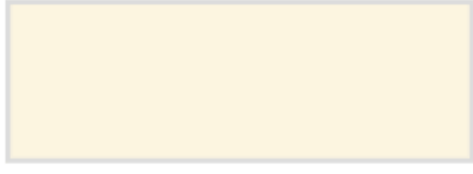
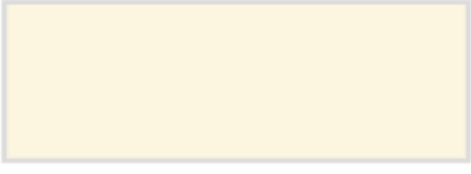
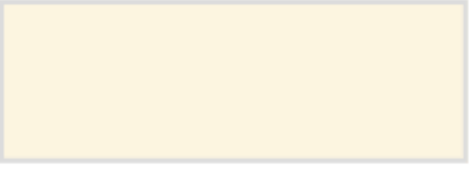
- (“short”) Rows can be read together/sequentially

read users by city



Column Family / Columns

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

	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				
Row key 2				

t1
t2
t3

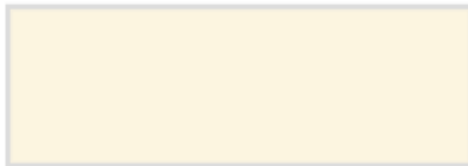
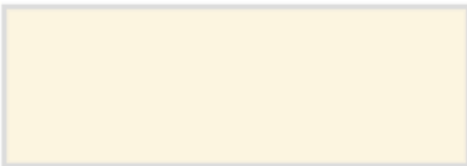

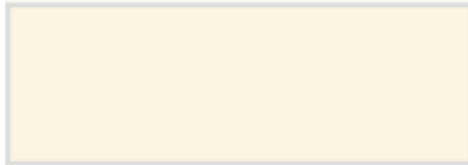
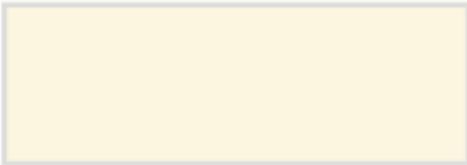
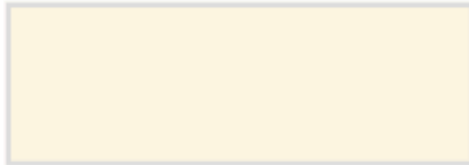
Dotted lines connect the yellow box in the 'Row key 1' cell for 'Column 2' of 'Column family 2' to the 't1' and 't2' rows of the table on the right. Another dotted line connects the 't3' row to the 'Column 2' header of 'Column family 2'.

Column Family / Columns

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

Not too much (up to ~100) families

“unlimited” columns

	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				
Row key 2				

t1

t2




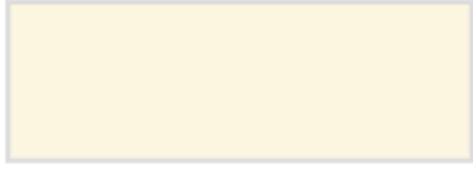
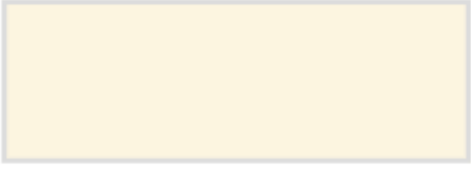
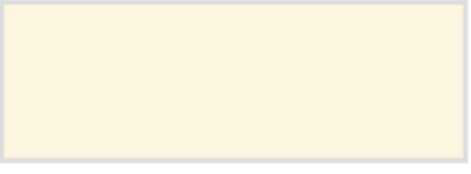
t3

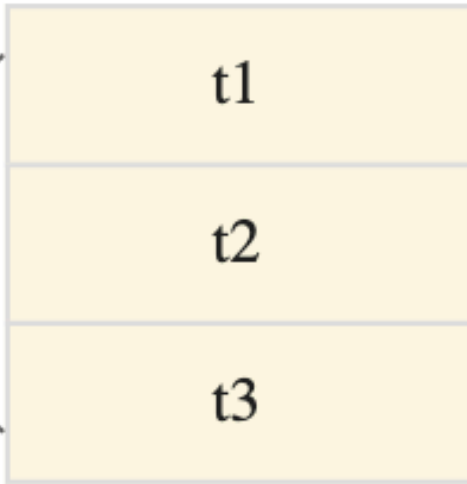
Column Family / Columns

- Column family - group of column usually of the same time for compression

Access control per column family

- Column name - `family:qualifier`

	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				
Row key 2				



t1

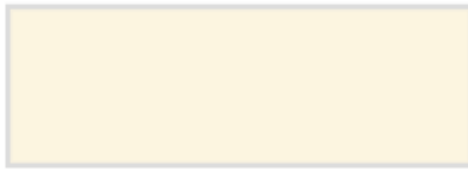
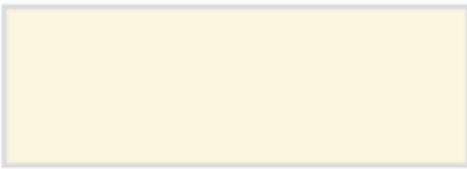

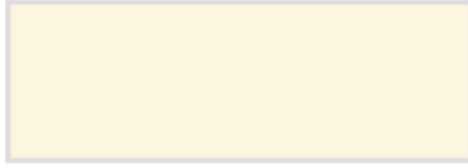
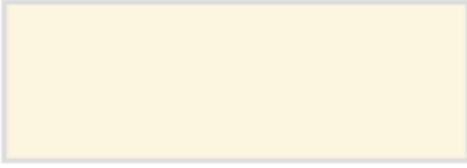
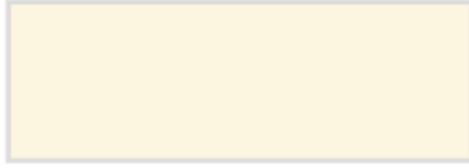
t2

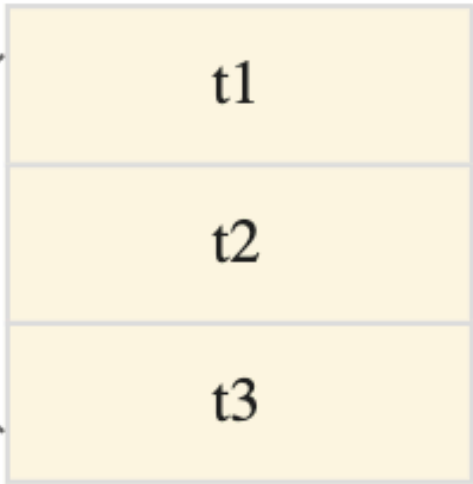
t3

Column Family / Columns

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

NOTE - we can store data in the qualifier

	Column family 1		Column family 2	
	<i>Column 1</i>	<i>Column 2</i>	<i>Column 1</i>	<i>Column 2</i>
Row key 1				
Row key 2				

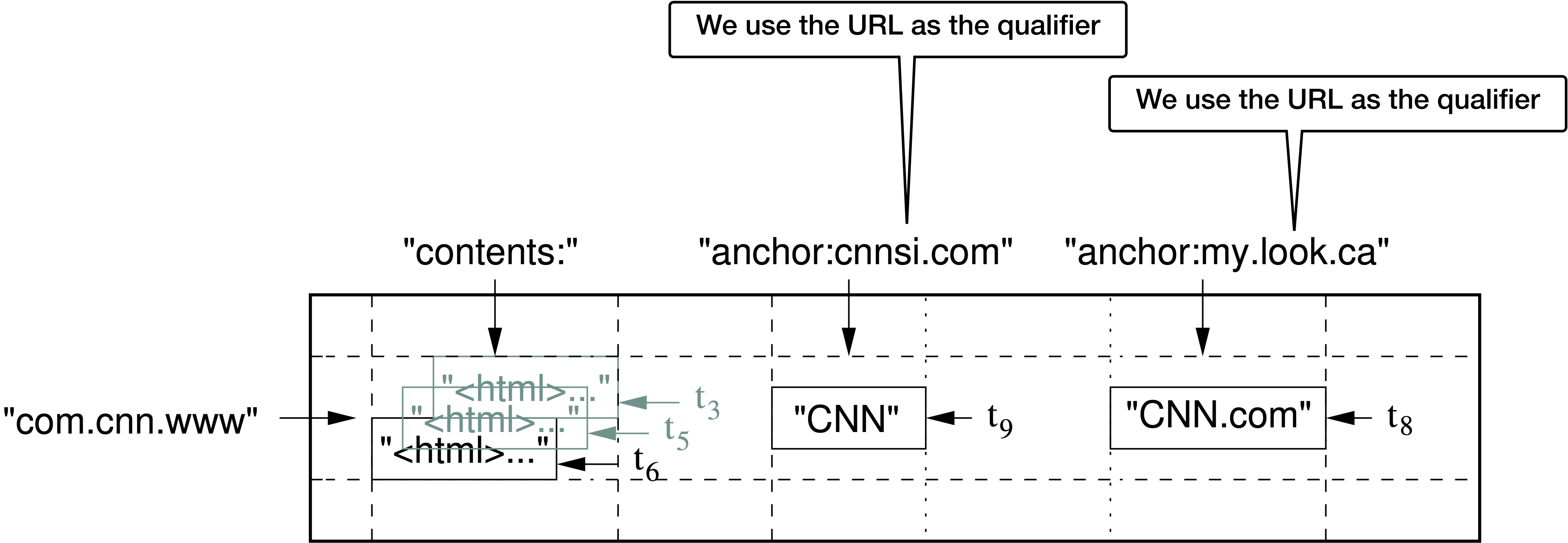


t1

t2

t3

Column Family / Columns



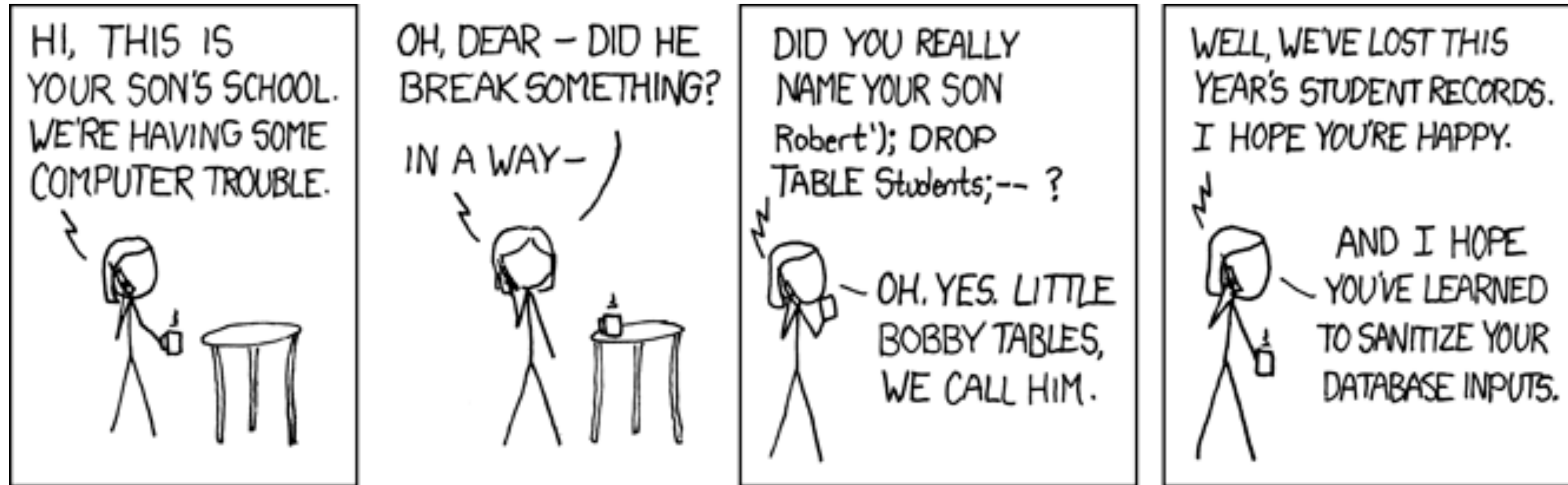
Timestamp

- 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... ;-)

Speaking about API/SQL



<https://xkcd.com/327/>

Agenda

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

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

Tablet - initialize

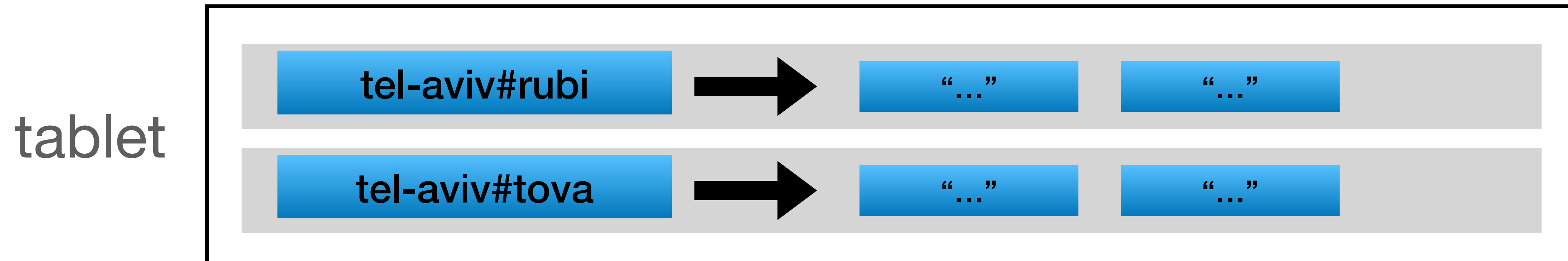
- When a table is created, there is 1 empty tablet

tablet



Tablet - initialize

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

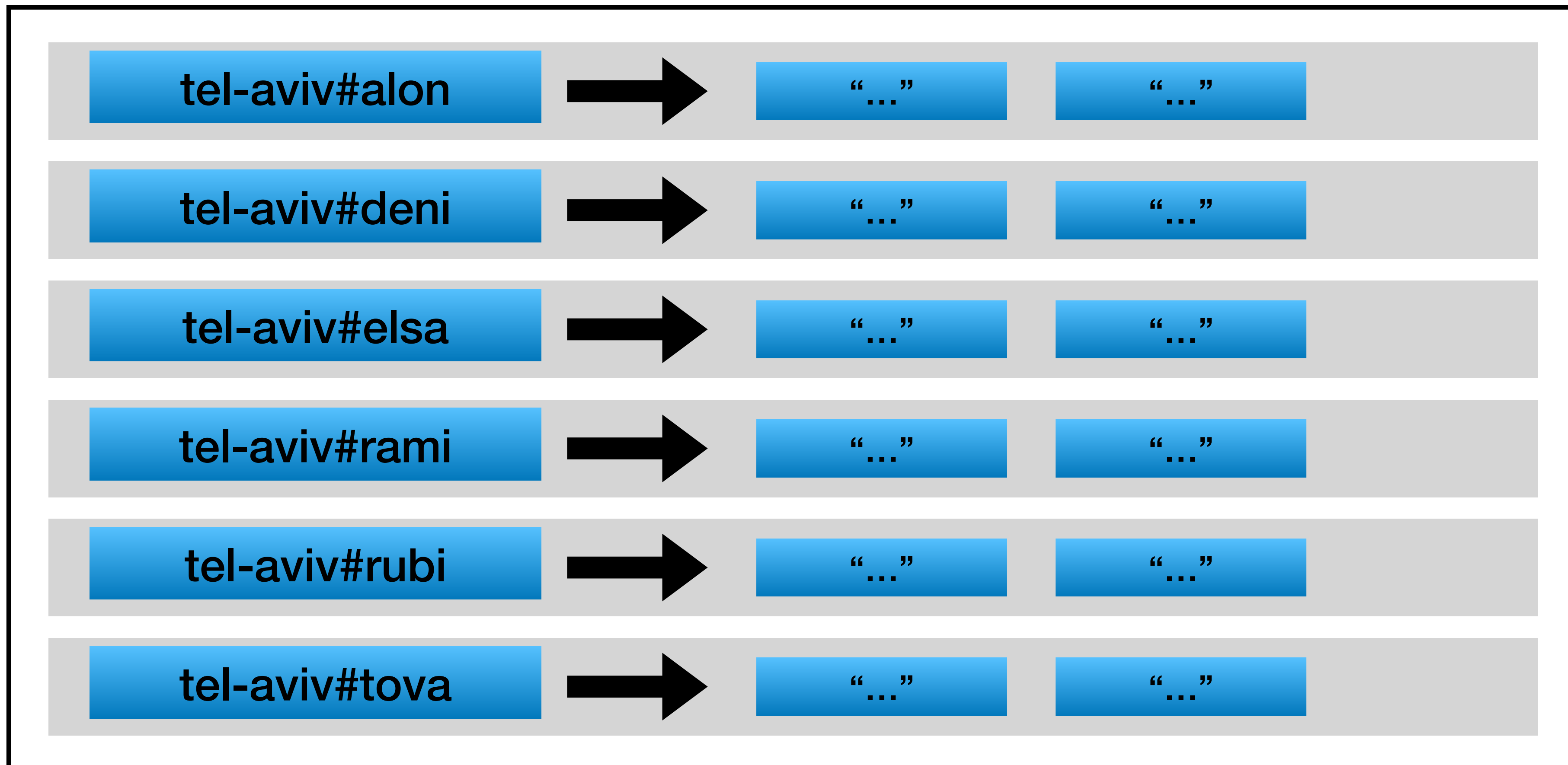


Tablet - Split

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

- When the table grows, the tablet is split

tablet

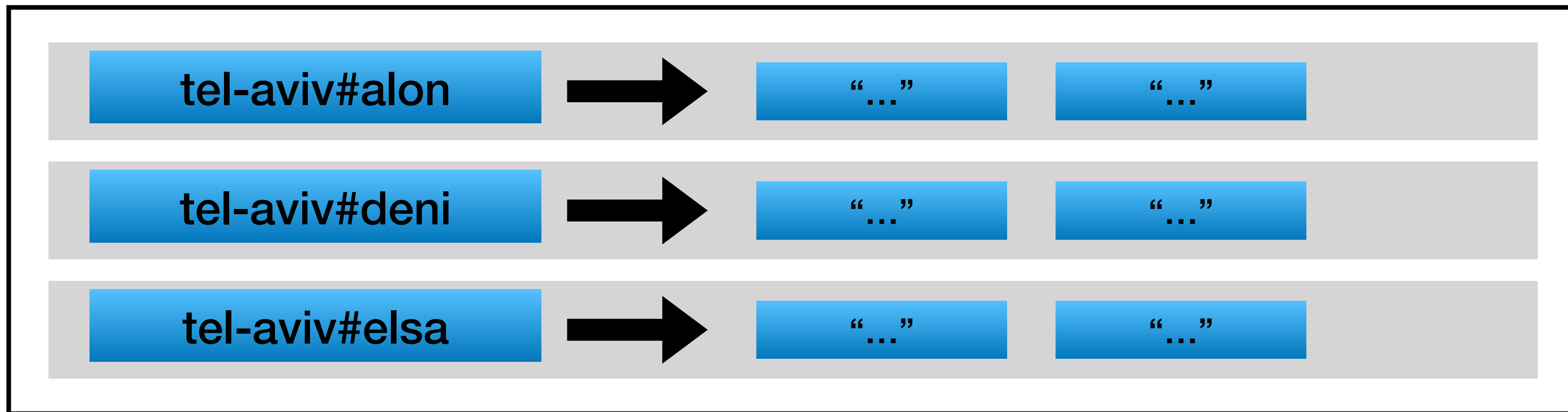


Tablet - Split

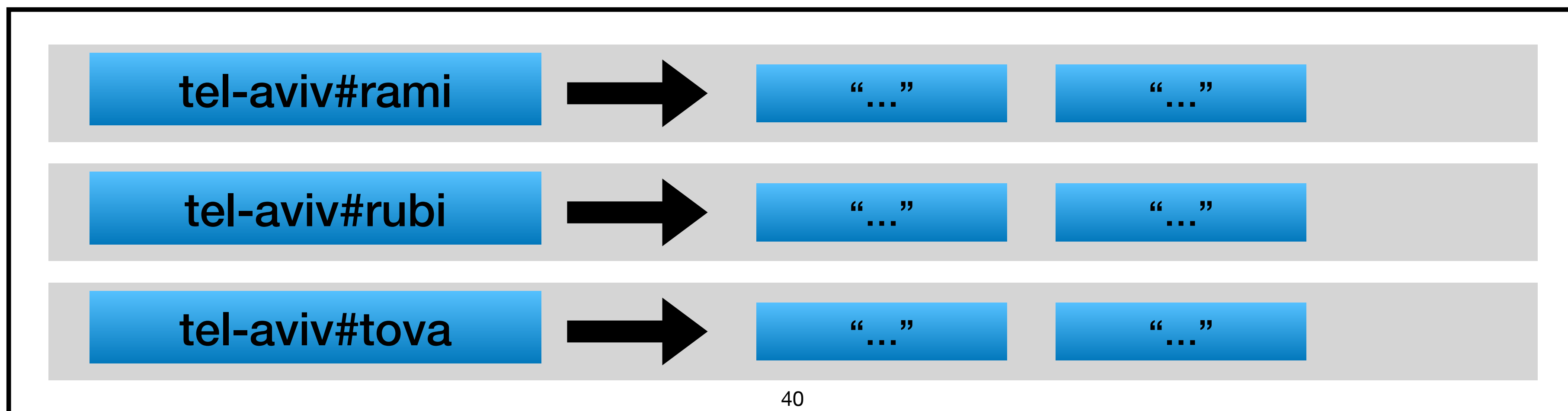
Approximate size: 100-200MB per tablet (default)

- When the table grows, the tablet is split

tablet

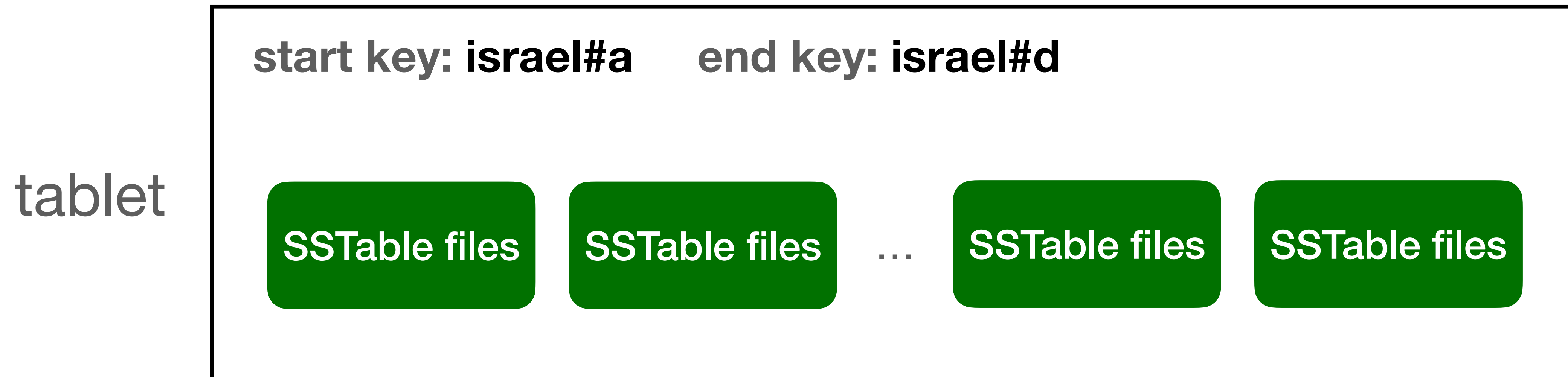


tablet



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)



“Single master distributed system”

- **Processing - Bigtable nodes**

- Manage read/writes (**without** actual storage)

- **Storage - GFS / Colossus (Google File System)**

- Manage actual storage files (SSTables)

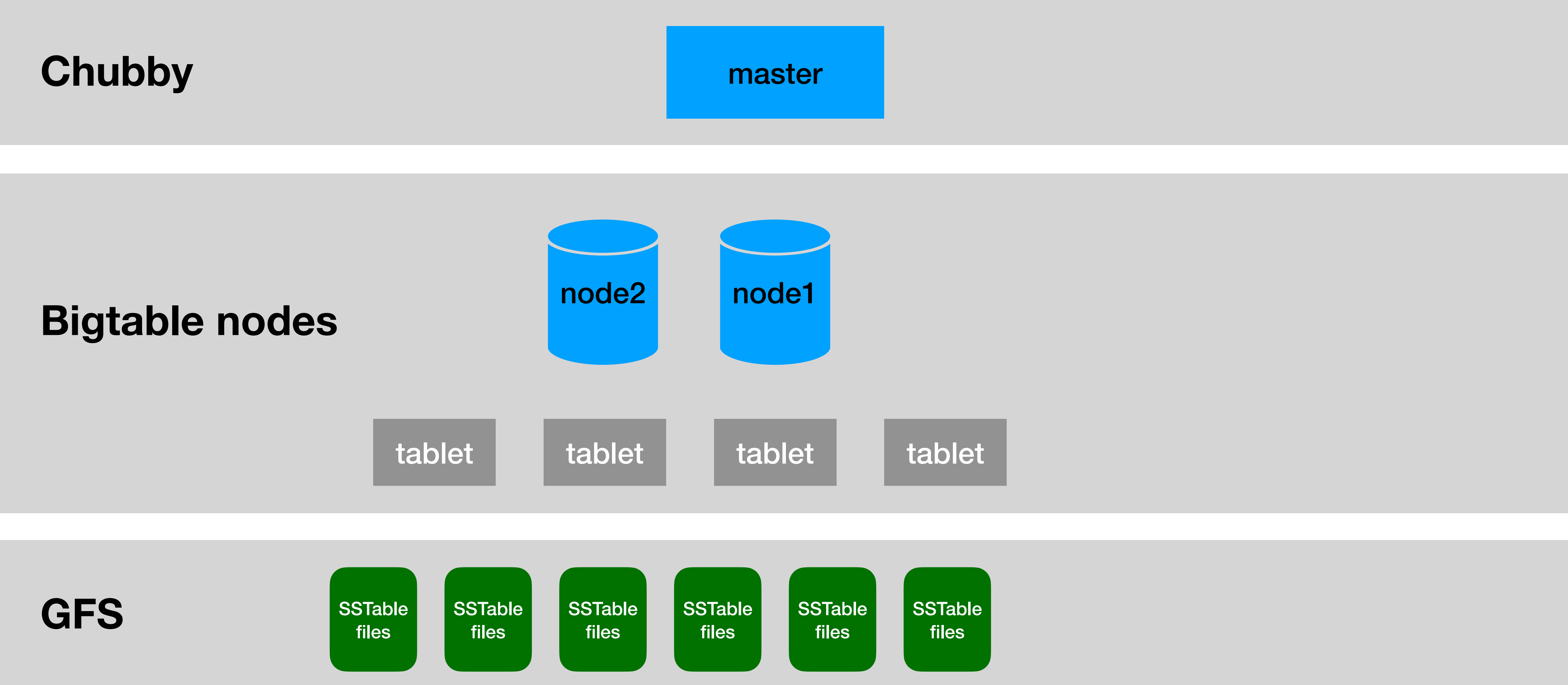
Bigtable design by components

- **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

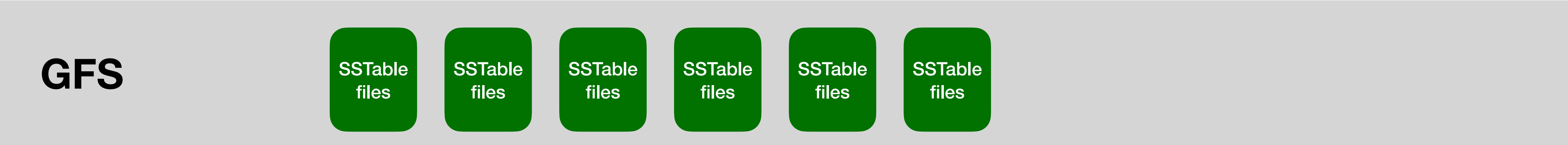
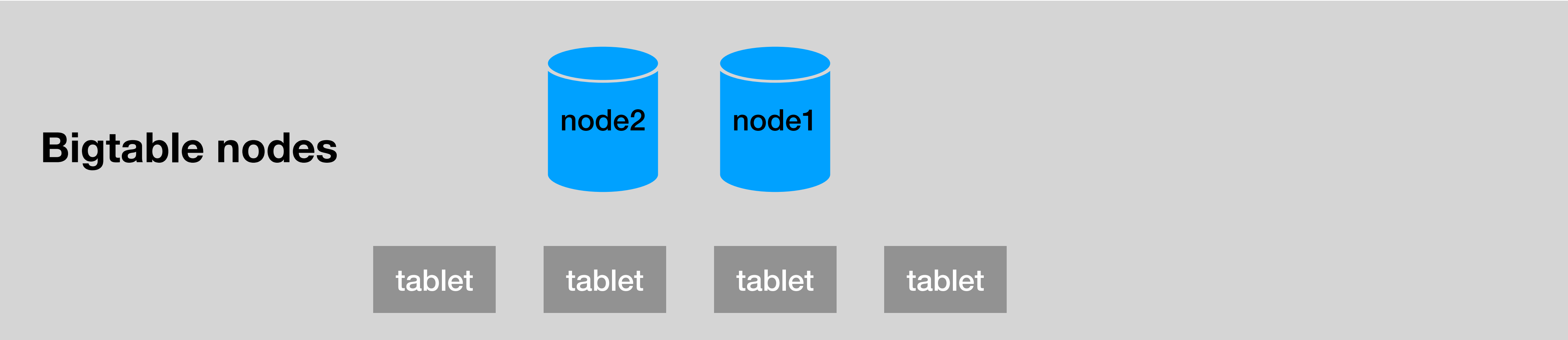
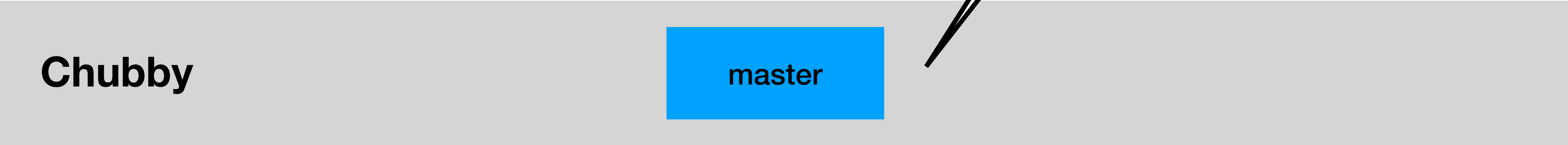
This is a BIG difference

Components by layers

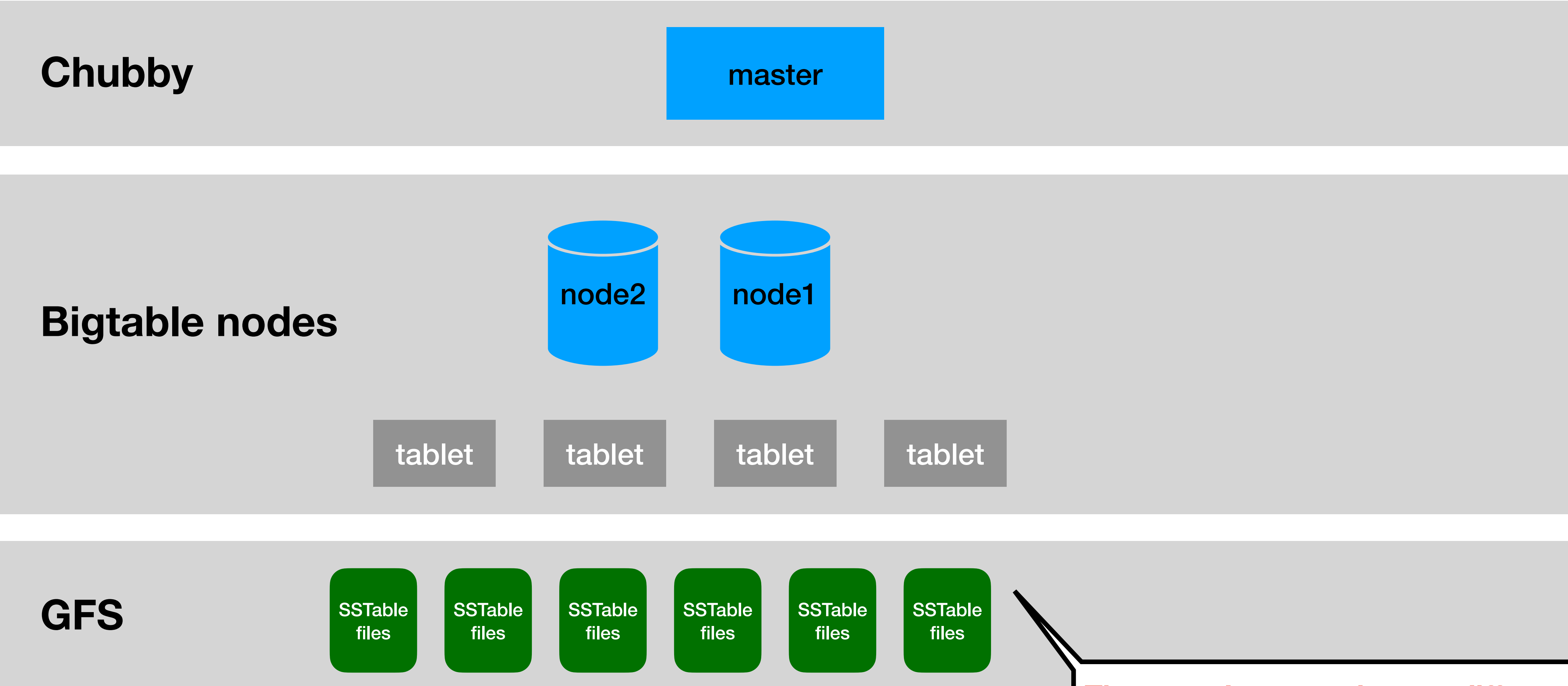


Components by layers

More than 1 server (more on this later)

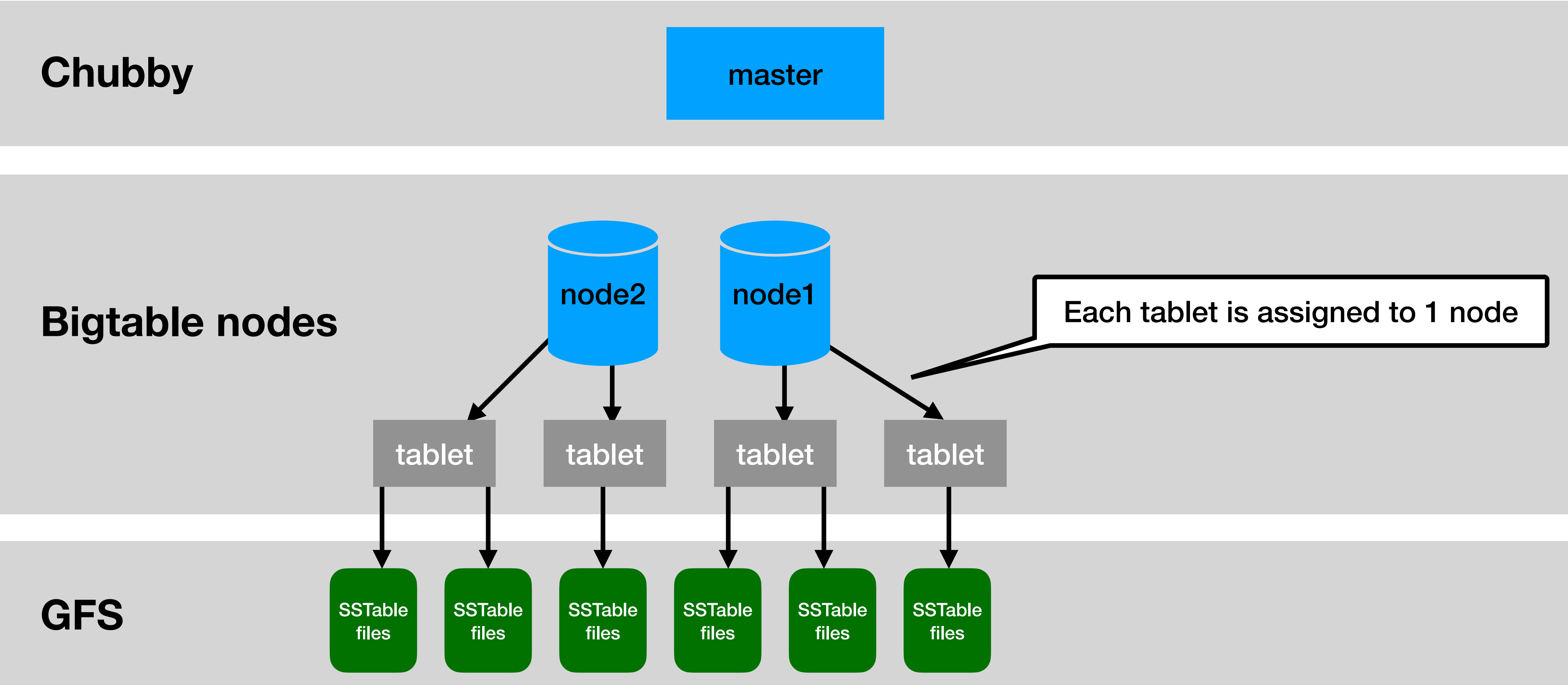


Components by layers

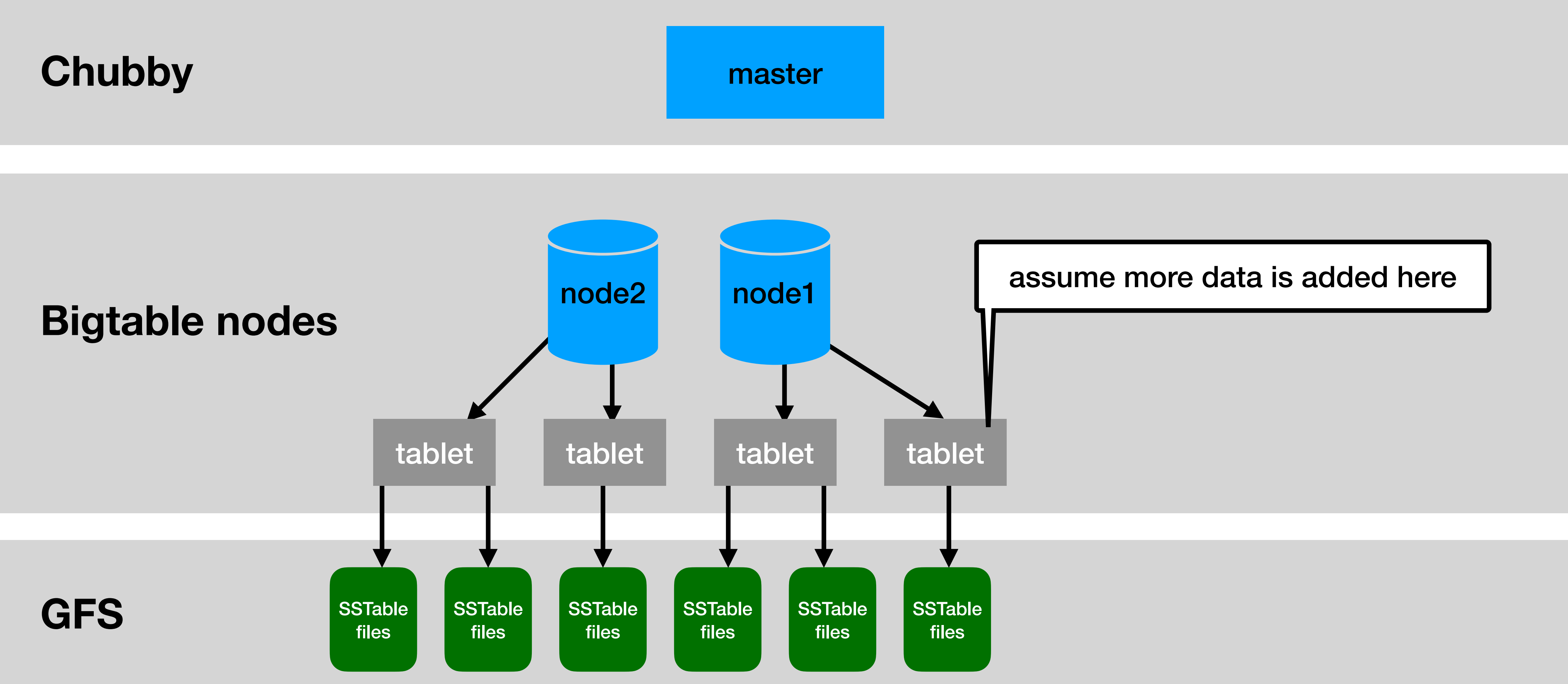


The actual storage is on a different layer from the tablets

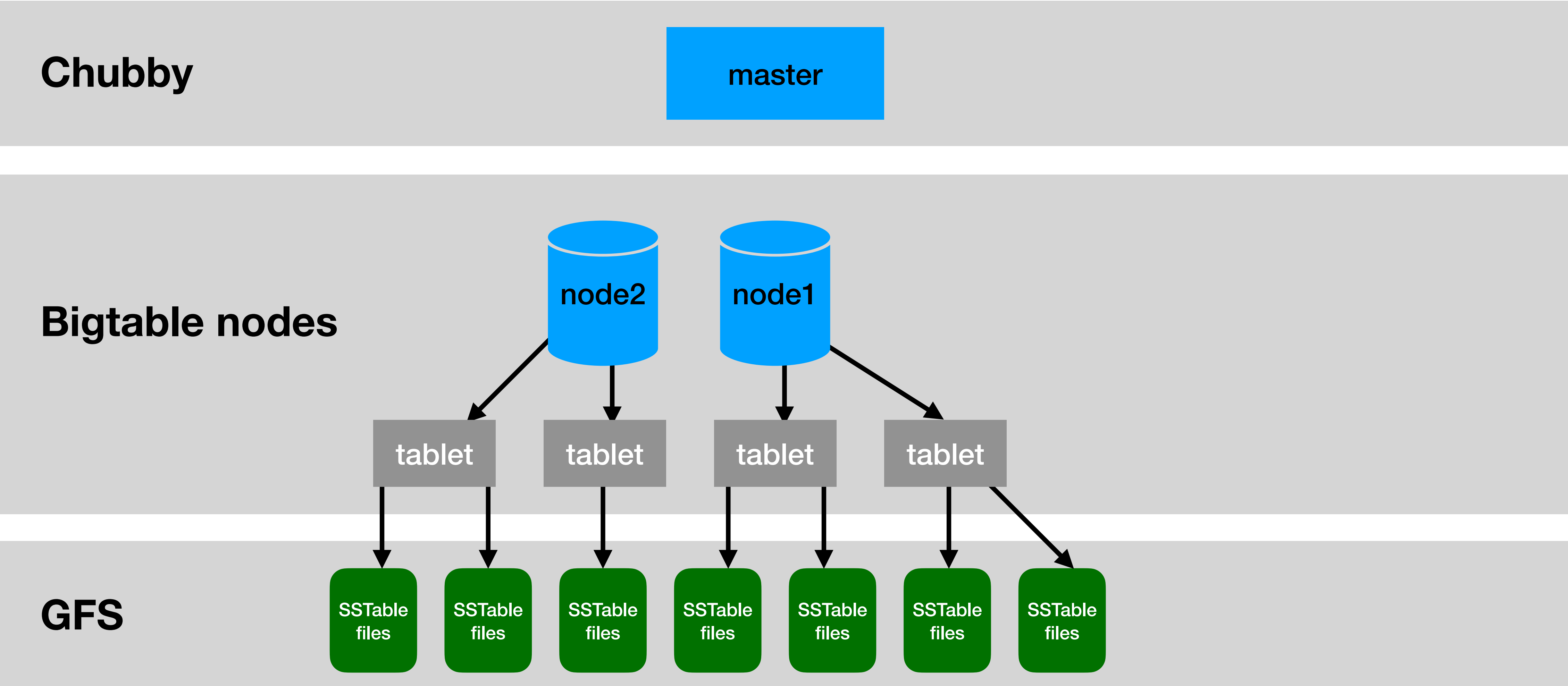
Components by layers



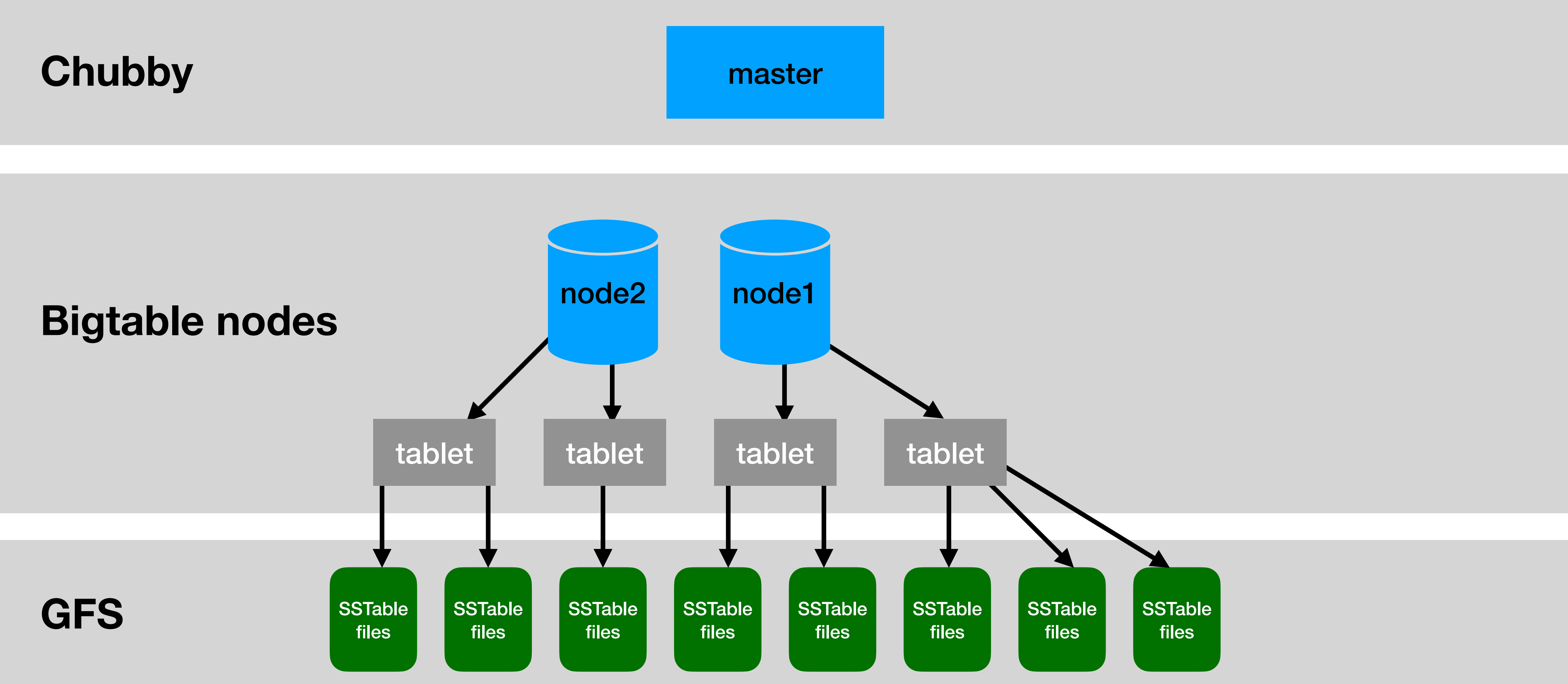
Components by layers



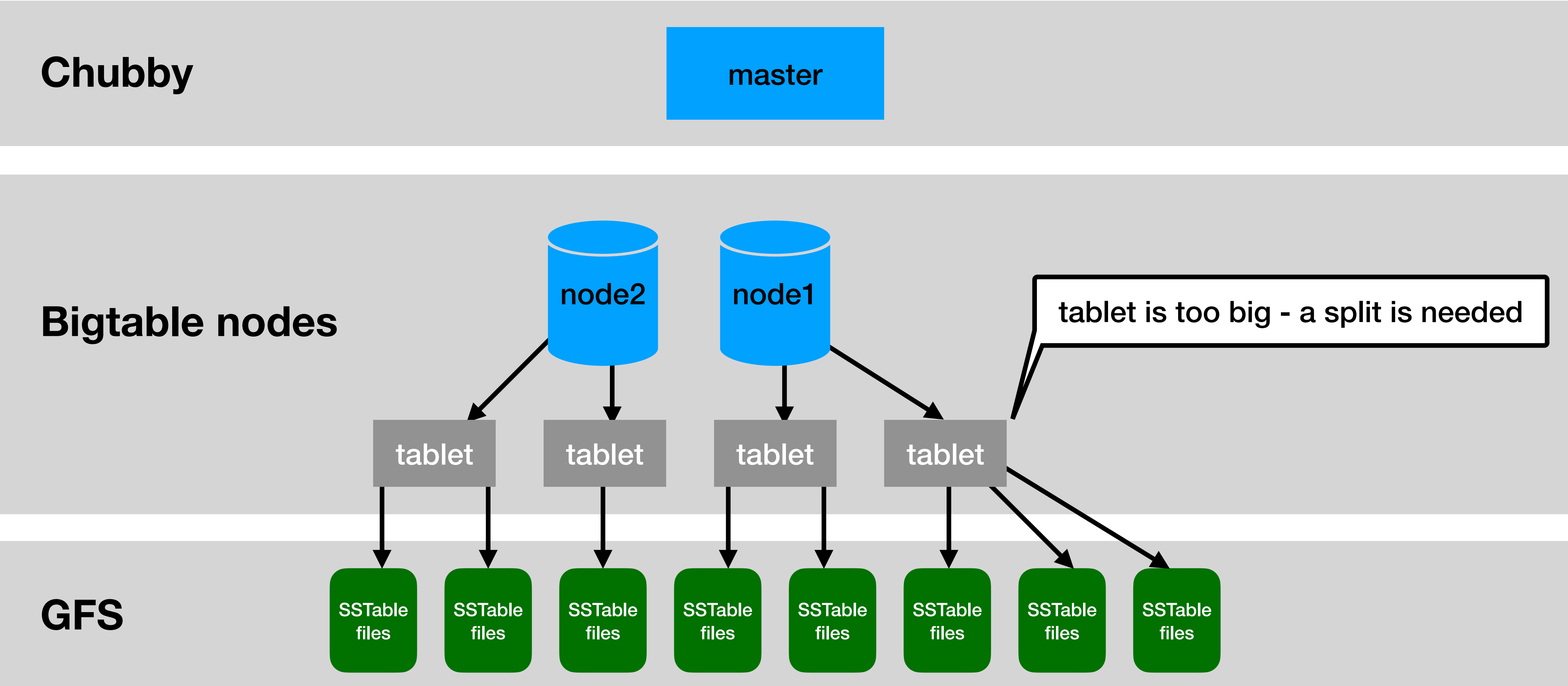
Components by layers



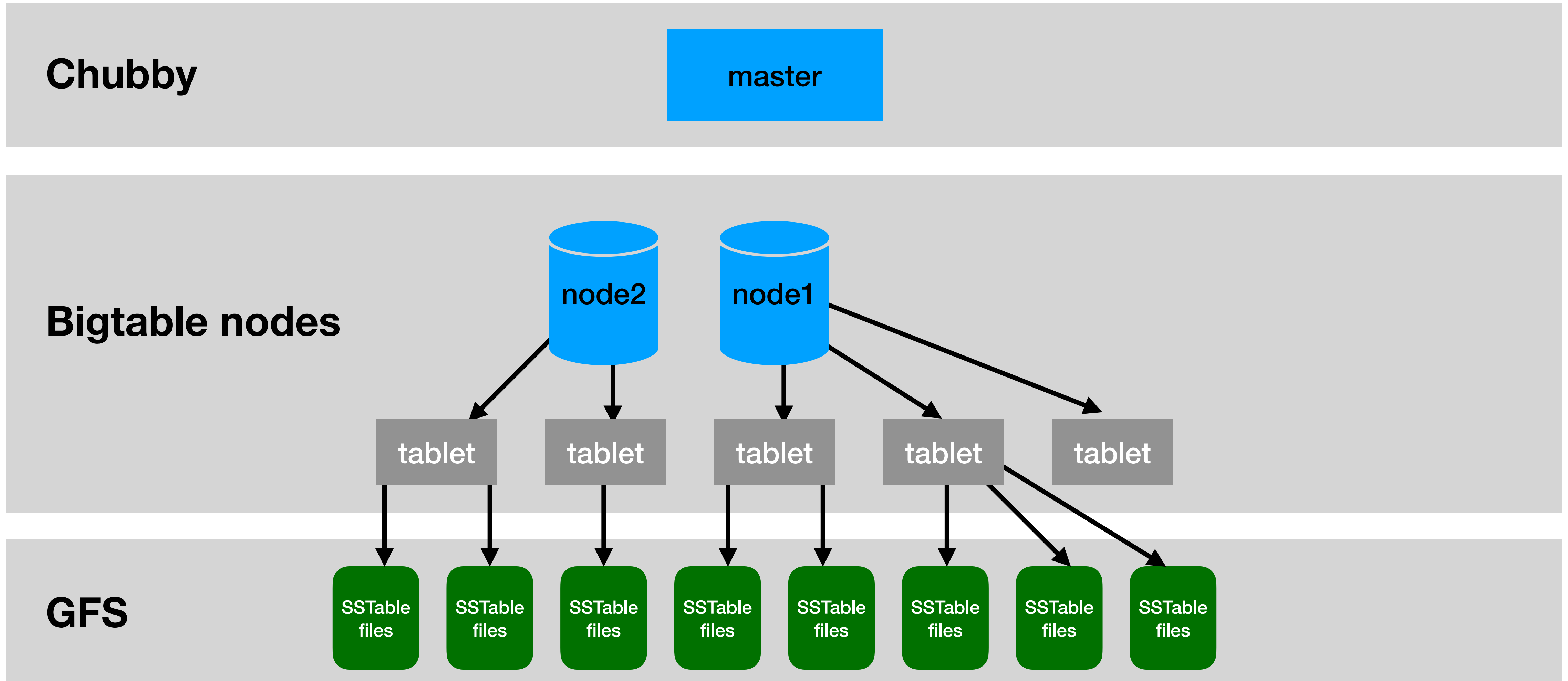
Components by layers



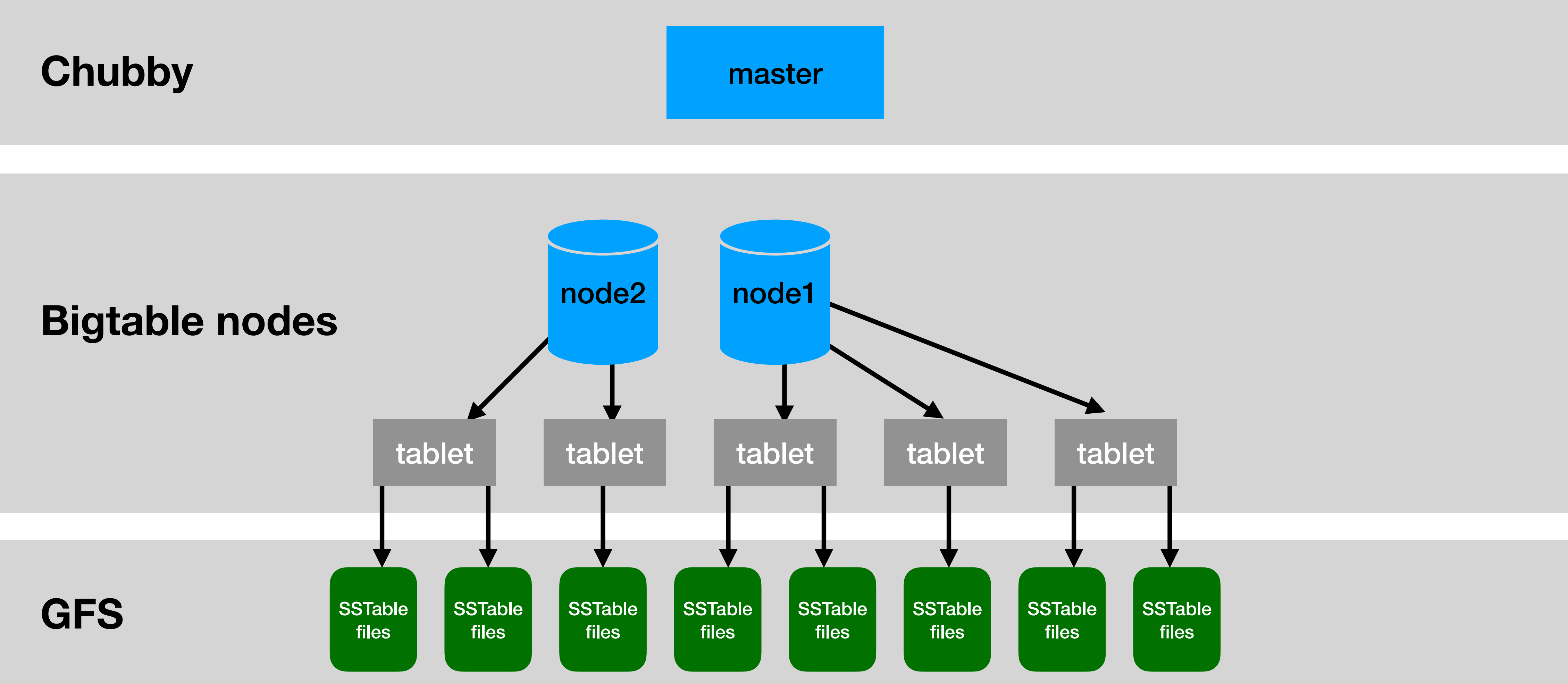
Components by layers



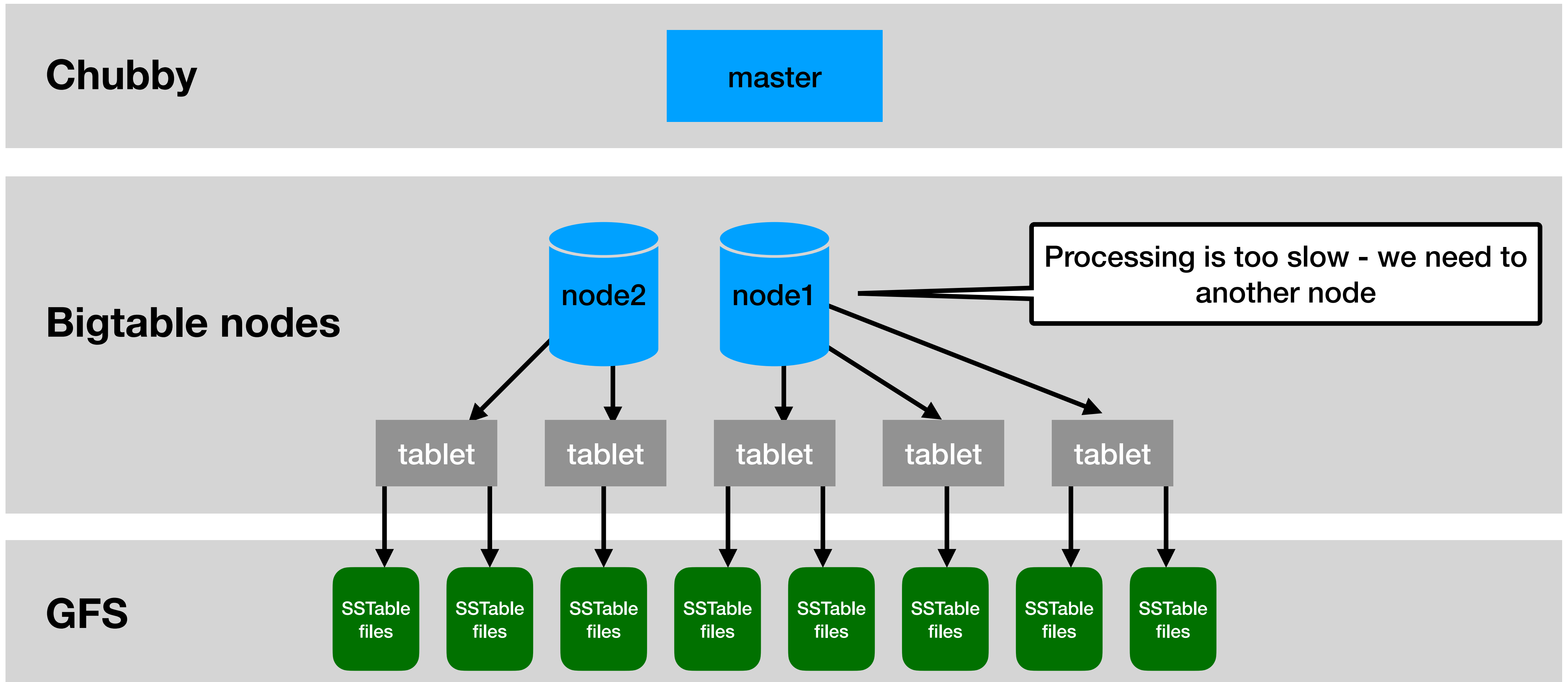
Components by layers



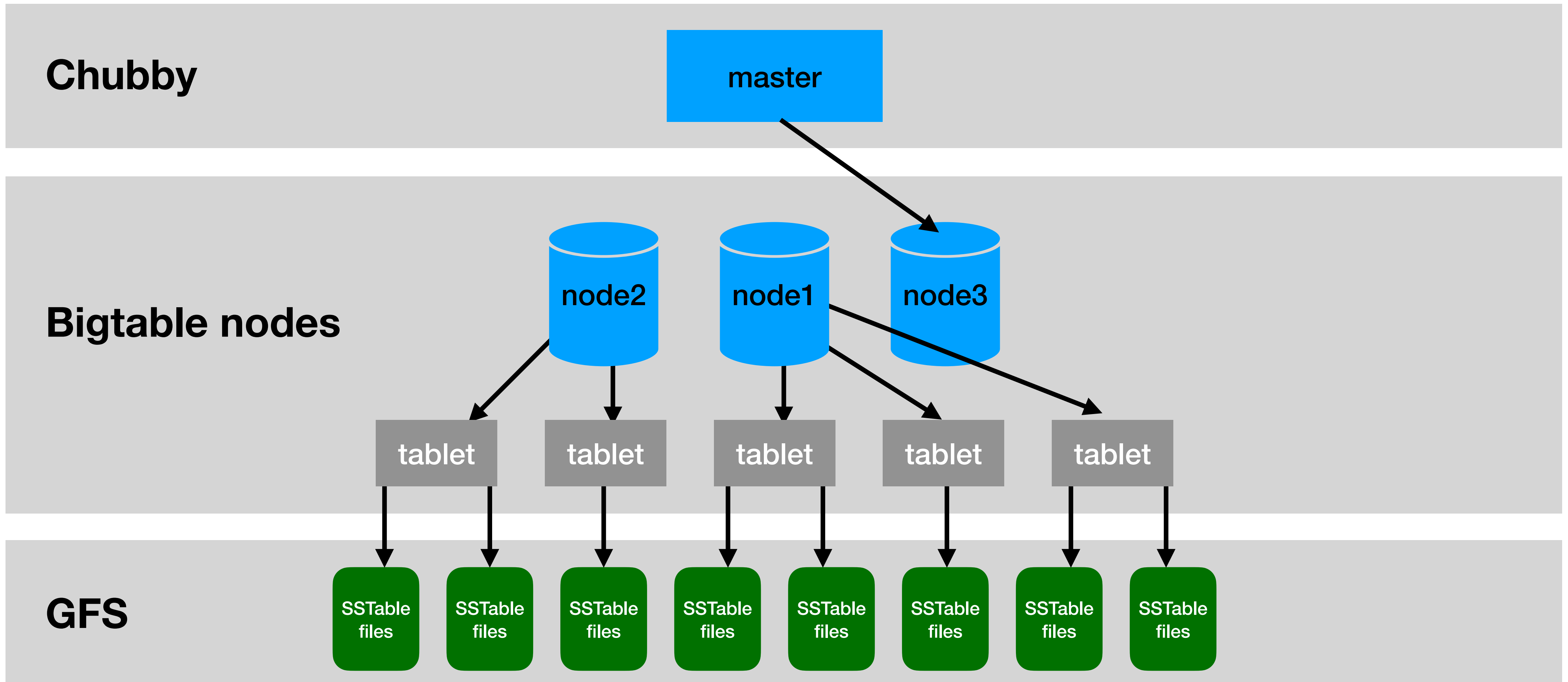
Components by layers



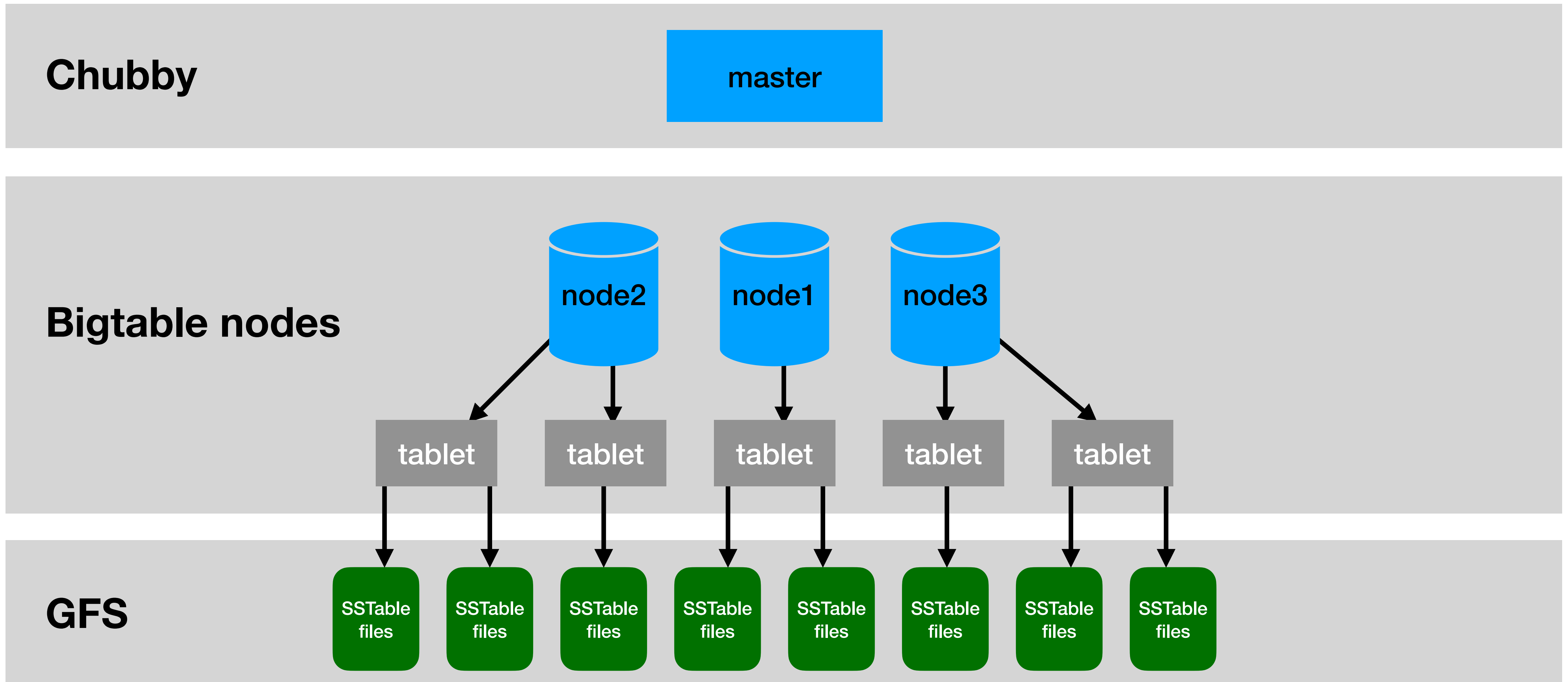
Components by layers



Components by layers



Components by layers



Agenda

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

Memtable

- 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

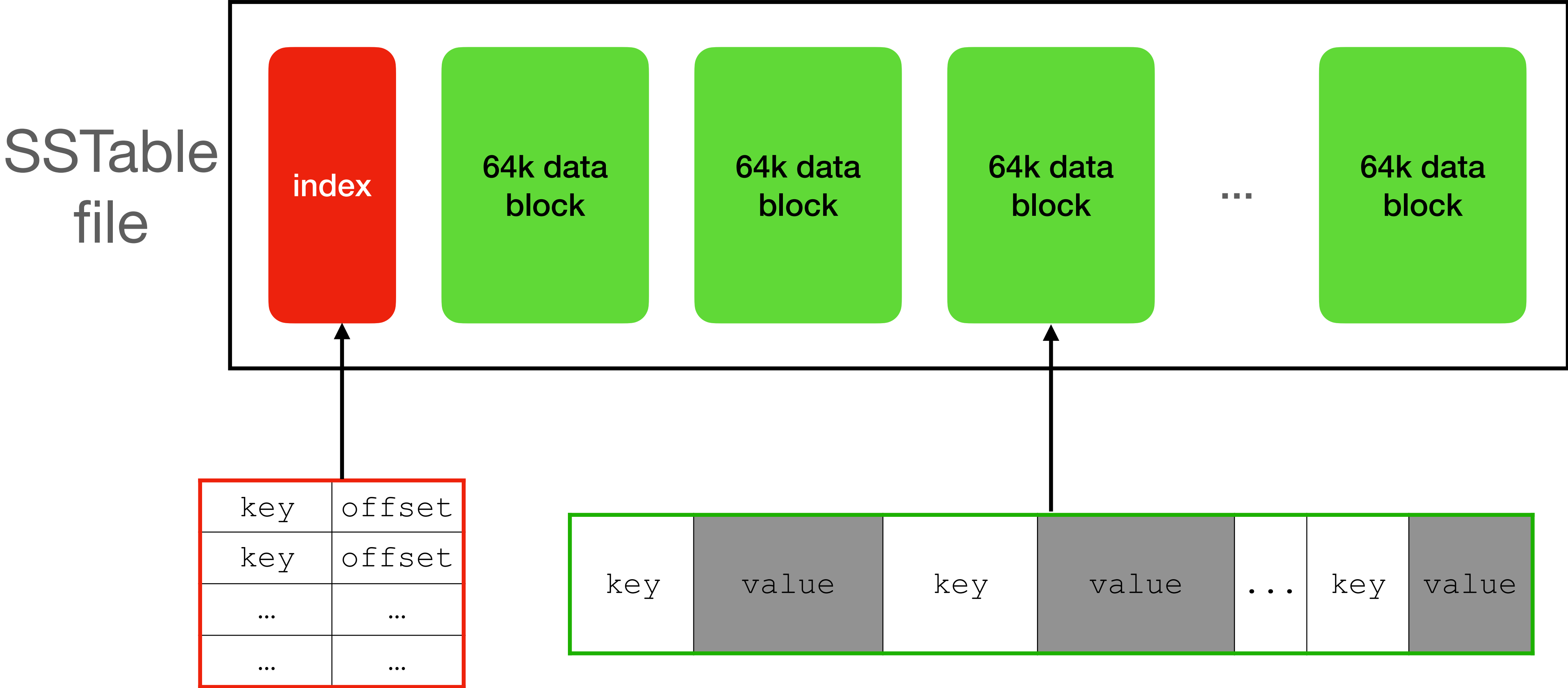


Minor compaction

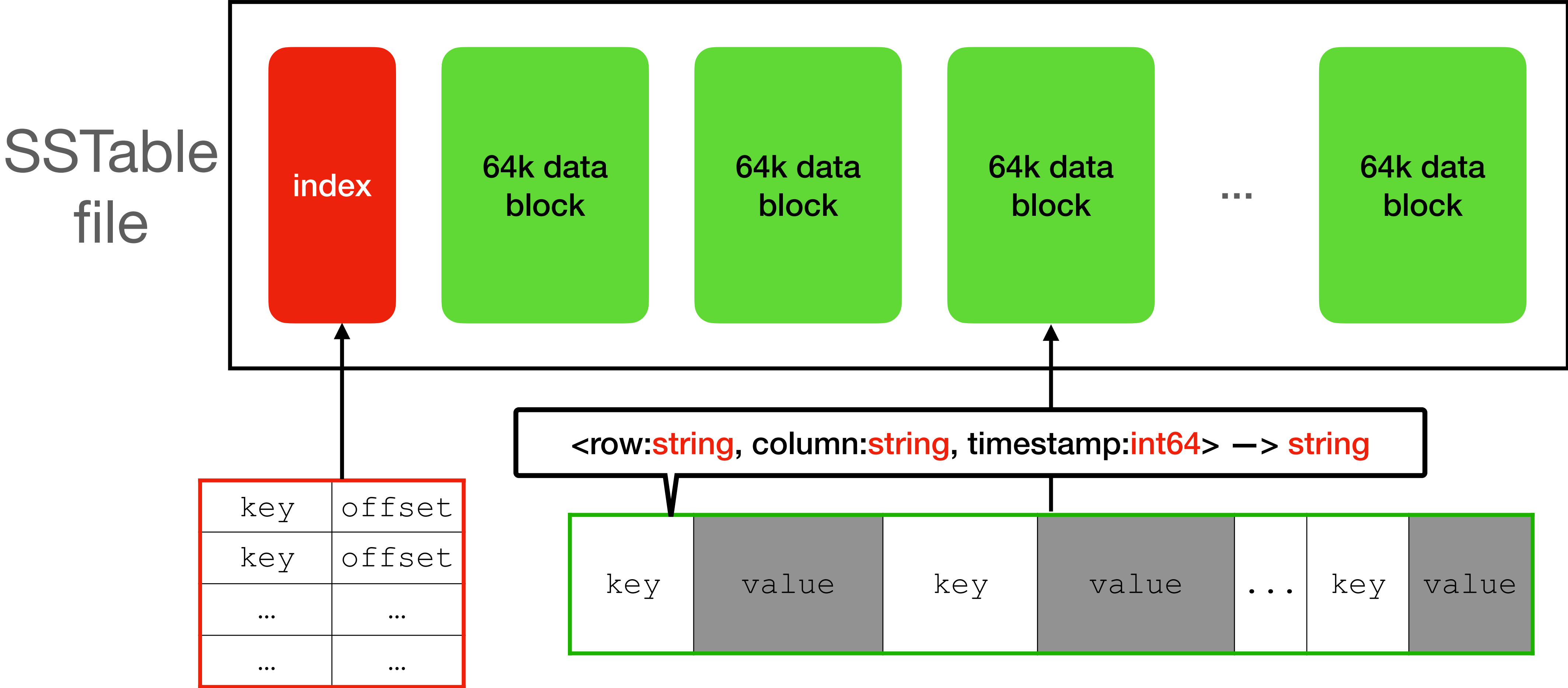
Sorted String Table (SSTable)

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

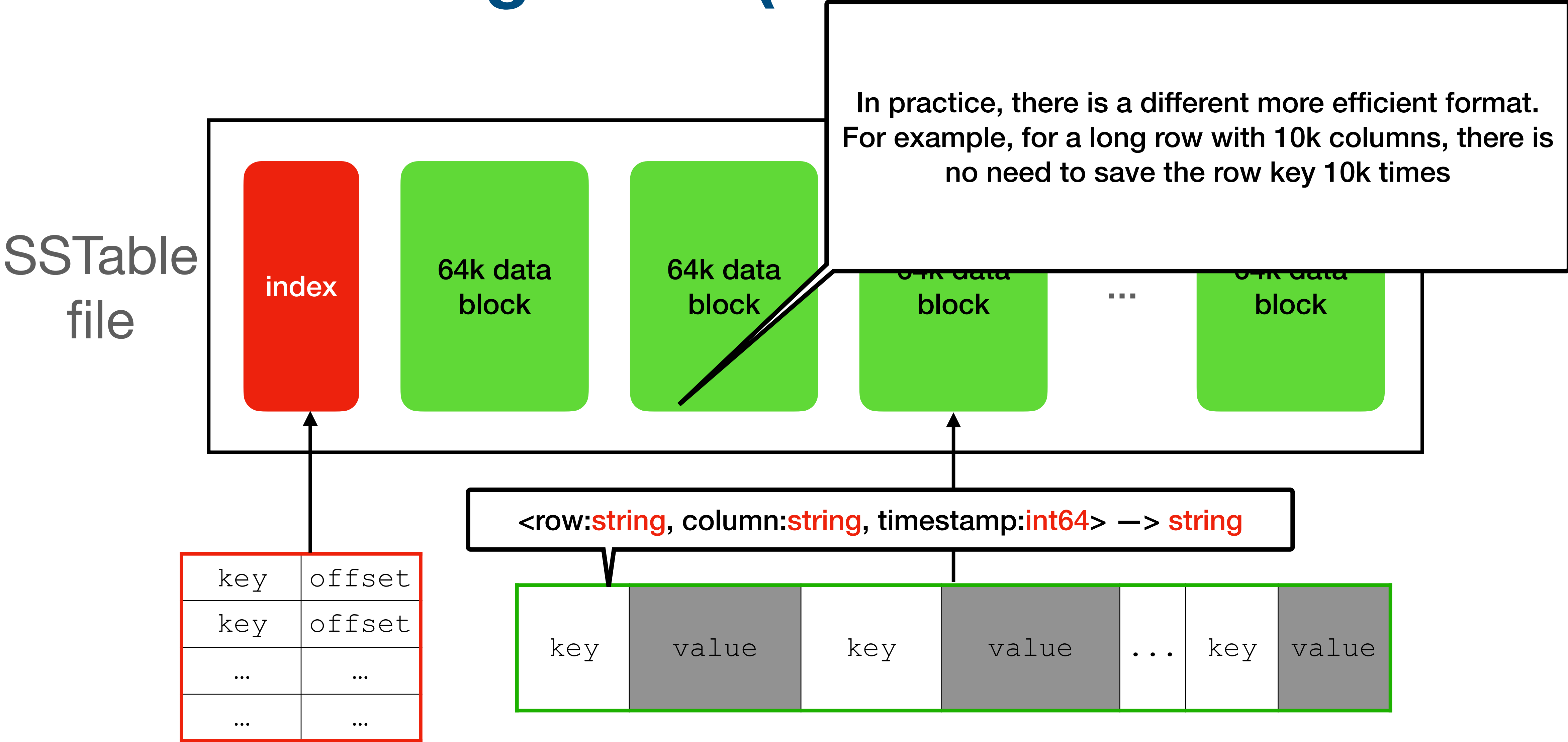
Sorted String Table (SSTable)



Sorted String Table (SSTable)



Sorted String Table (SSTable)



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)

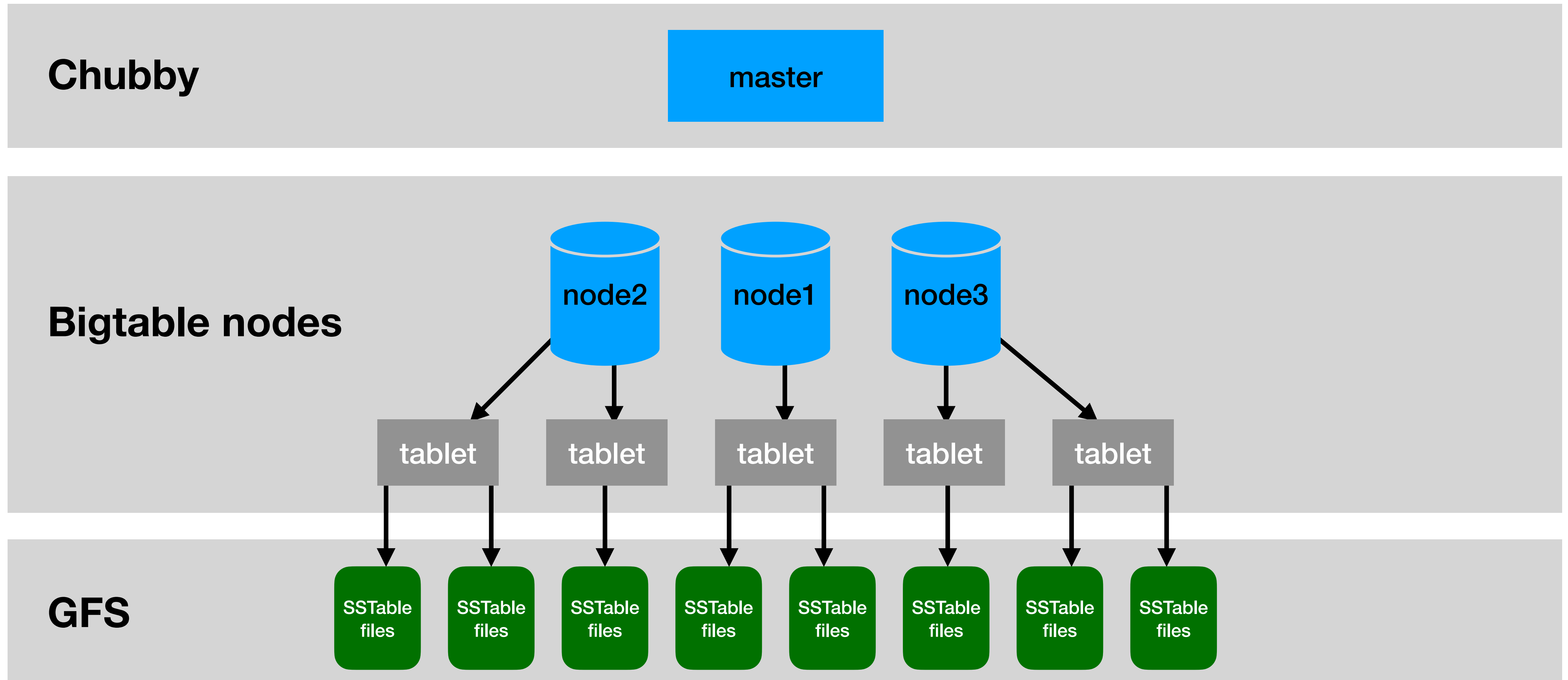


More on this later in the course

Agenda

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

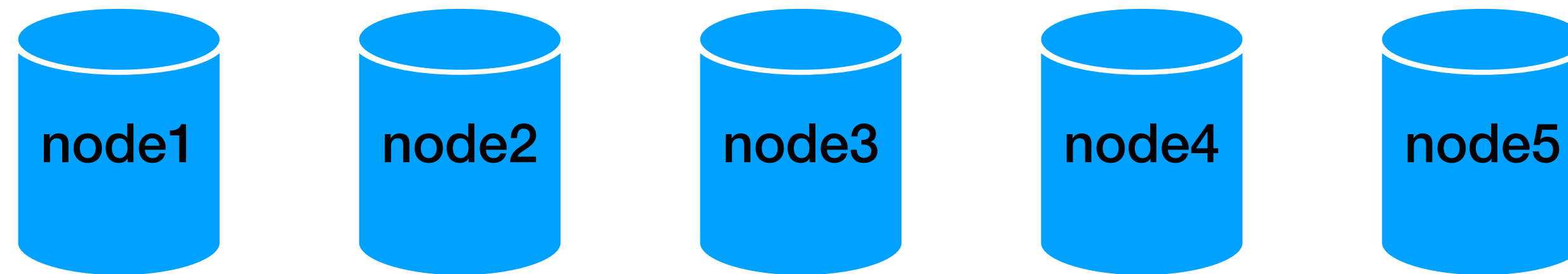
Reminder - Components by layers



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

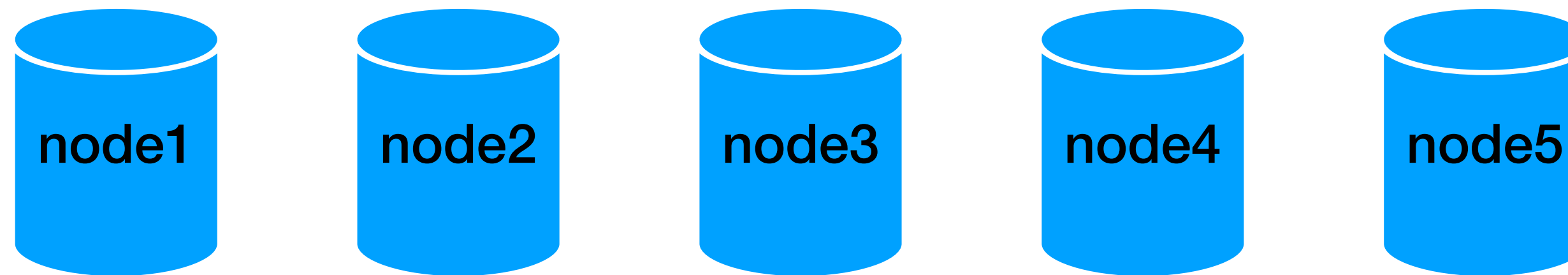
Chubby

Opensource:



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

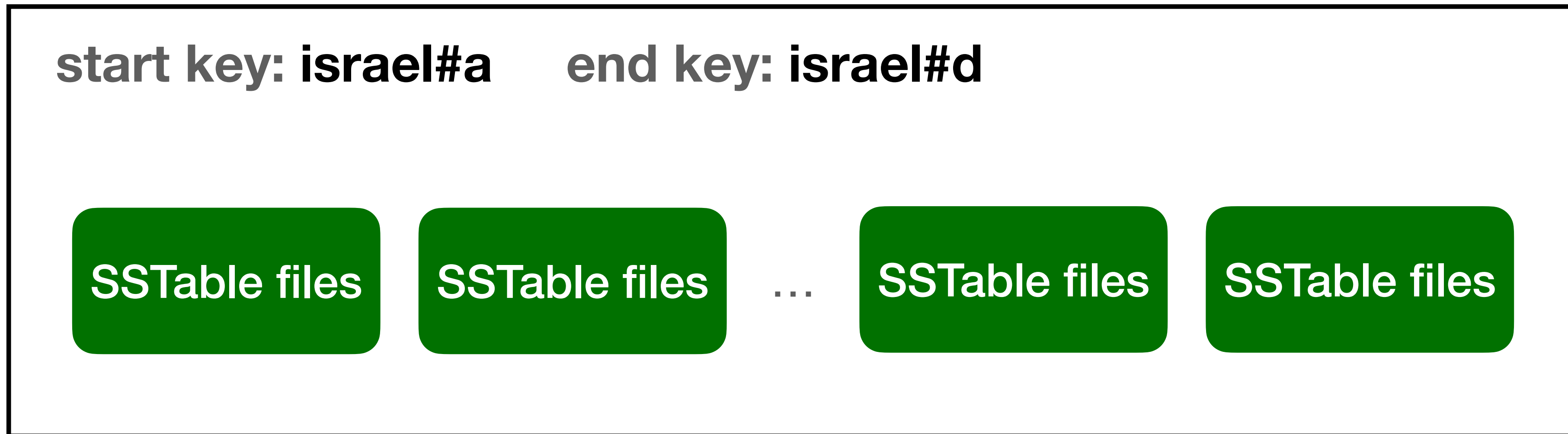
Agenda

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

Reminder - Tablet

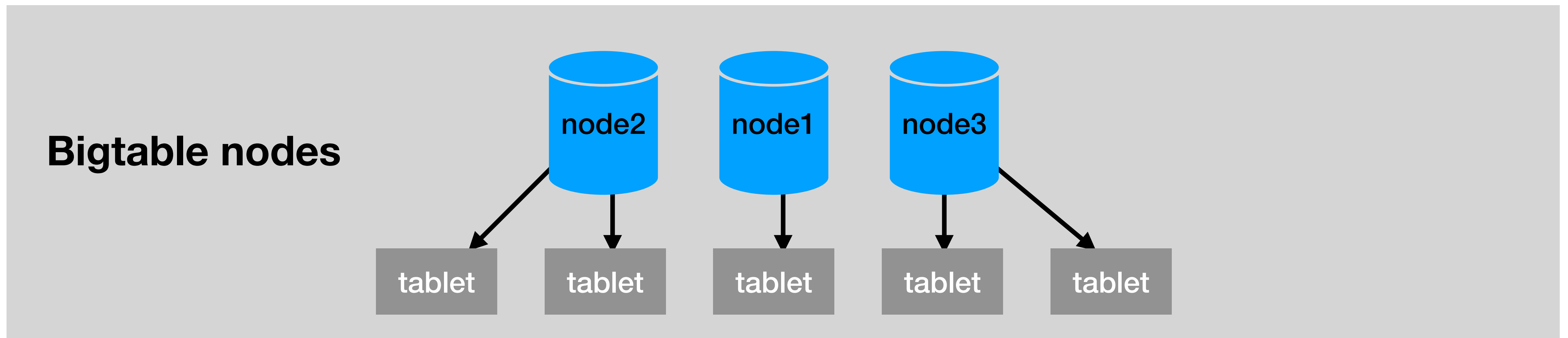
- A set of SSTables over a matching range comprise a tablet

tablet



Tablet location

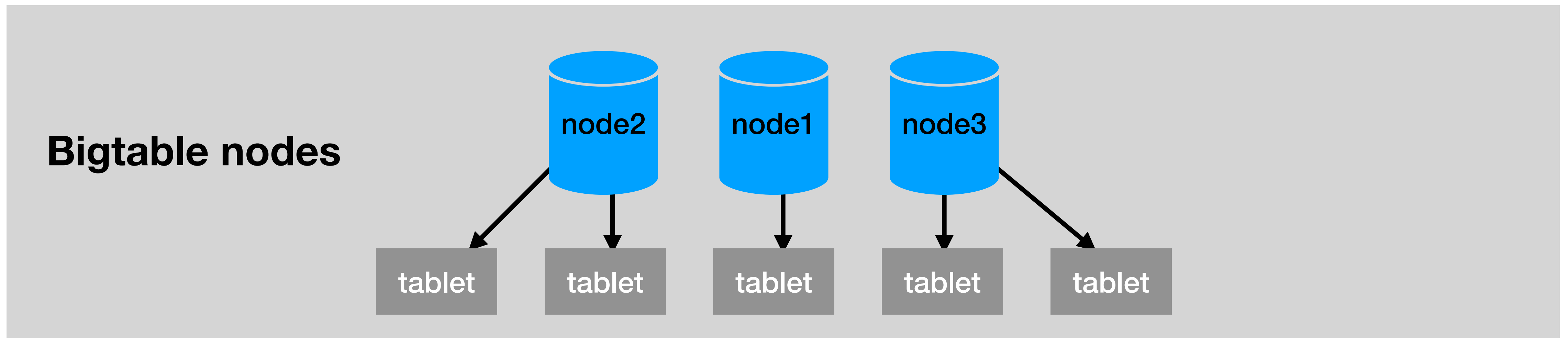
- How Bigtable stores the mapping between tablets and nodes?



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?

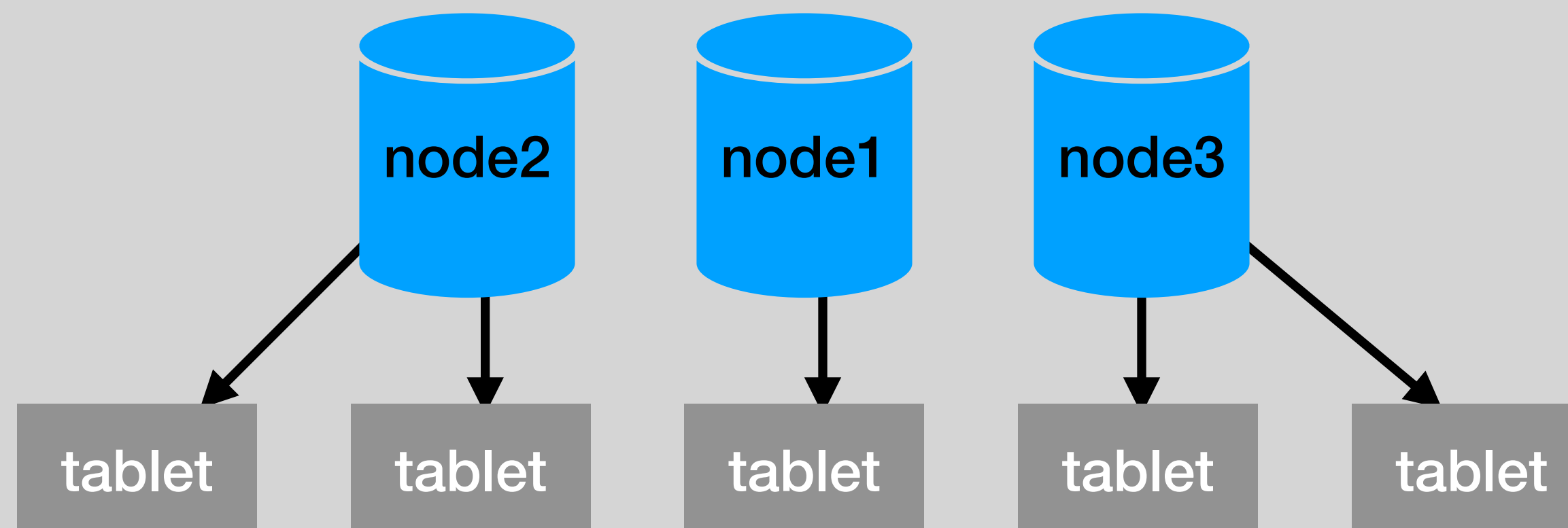


Tablet location

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”

Bigtable nodes



Tablet location

- How Bigtable and nodes?

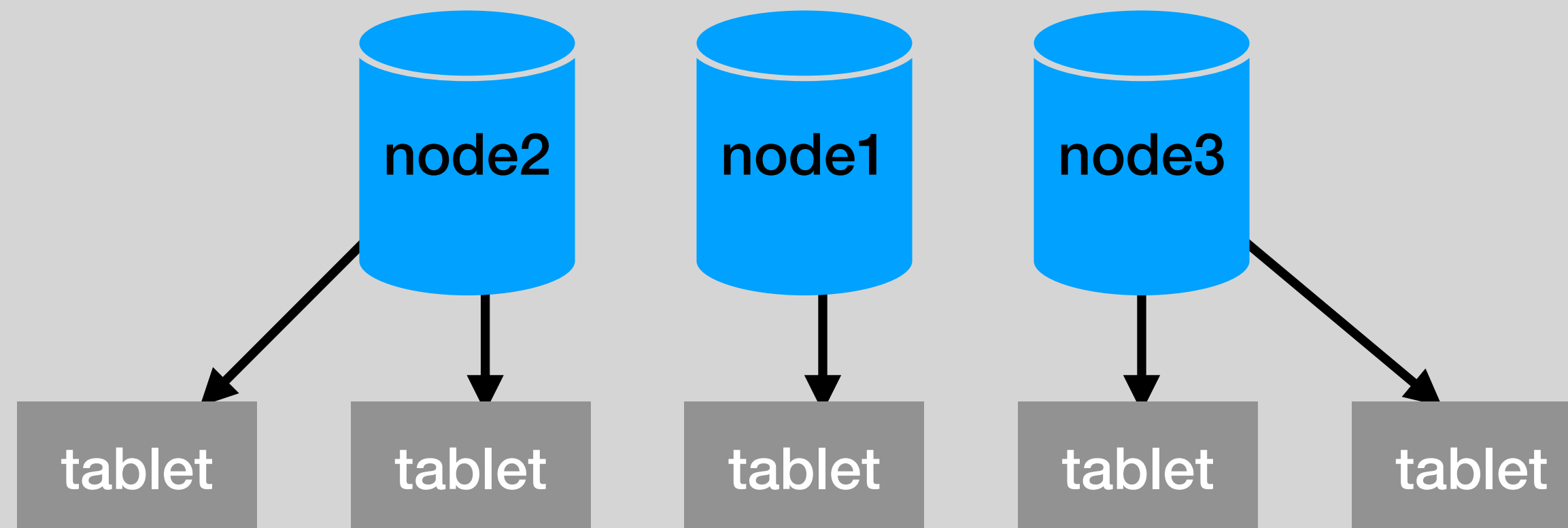
High fanout →
less I/O operation to find element →
great for indexes

between tablets

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

- Using “3-level hierarchy” index similar to B+ trees
B+ trees are search trees with “a lot of children”

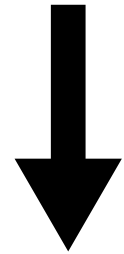
Bigtable nodes



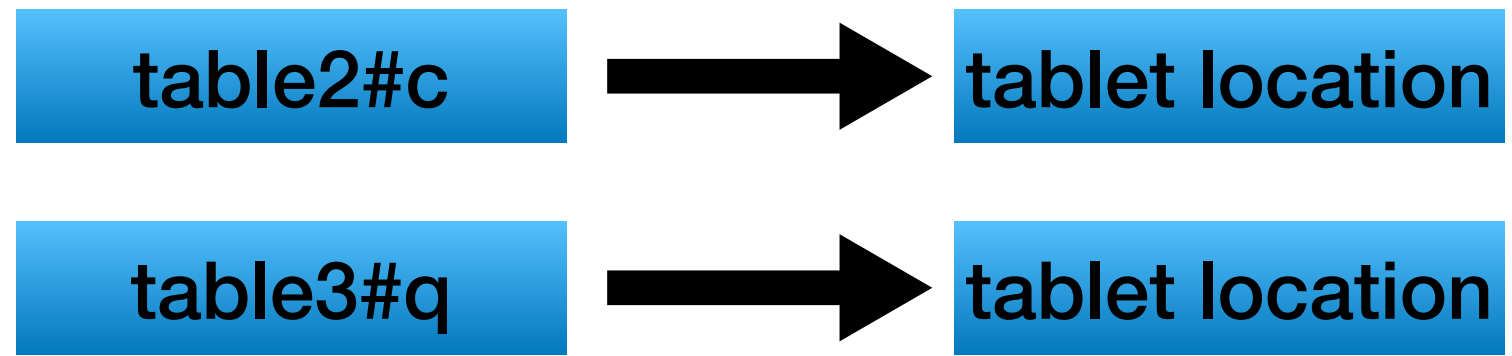
This Index is implement by

- 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

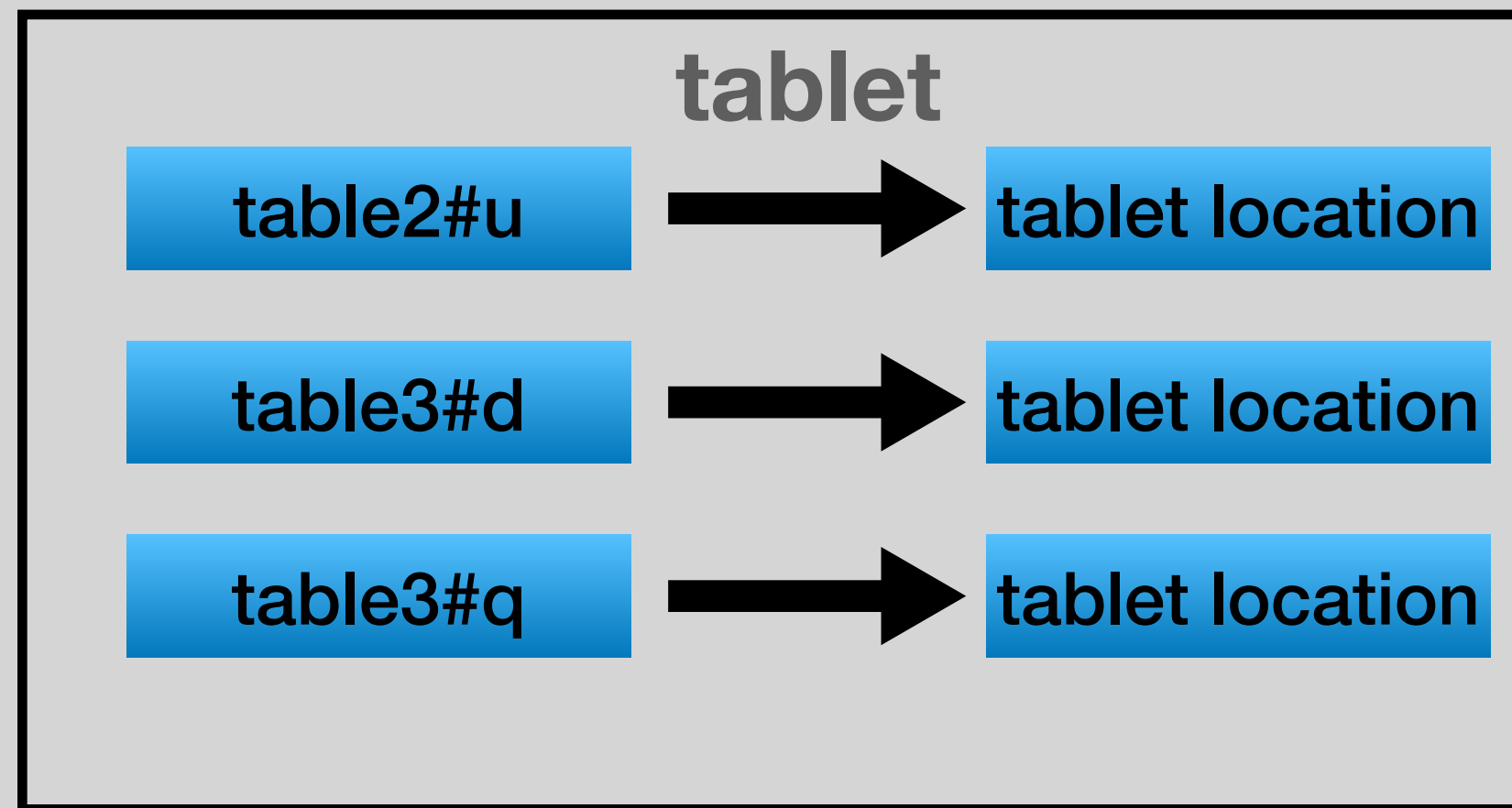
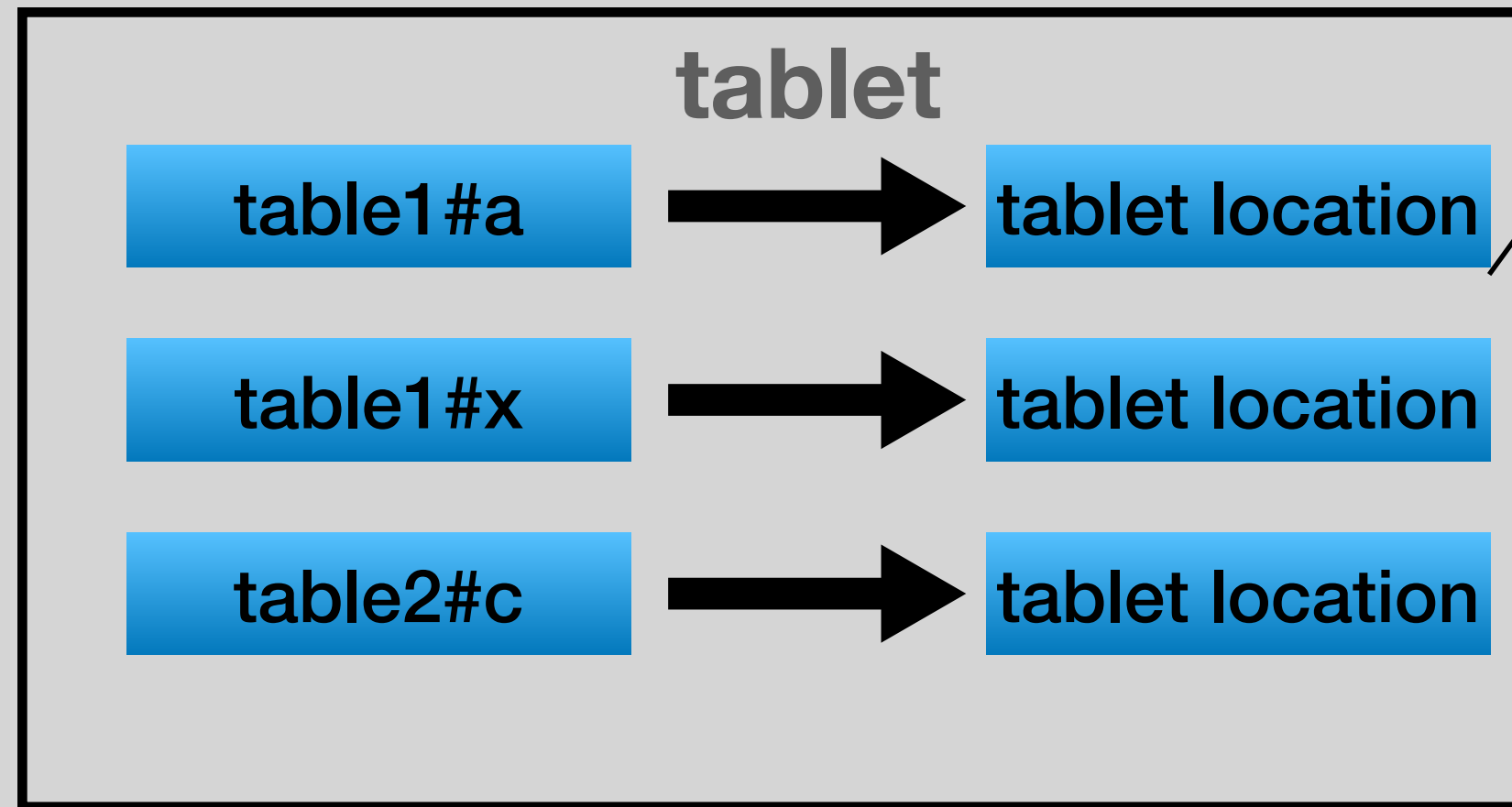
Chubby file



root tablet



METADATA table



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

Agenda

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

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

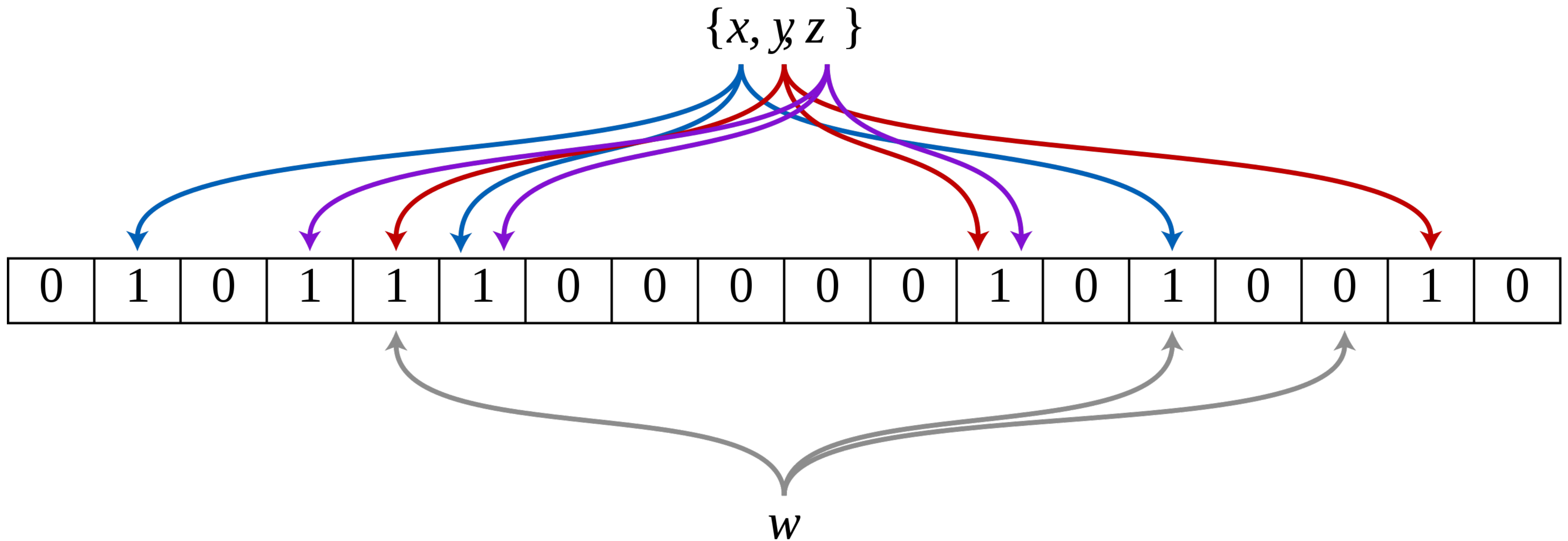
- A read operation may read from all SSTables of a tablet
can you think of an example?
- If these SSTables are not cached, a lot of disk access may happen
- To reduce these IOs, Bigtable uses Bloom filters for each SSTable (and keep them in memory) to reduce the number of IOs

Bloom filters - how they work

- Initialize (0) an **array of m bits**
- There are **k different hash functions of the rang [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)
- Chubby
- Tablet location
- Bloom filter
- **Summary**

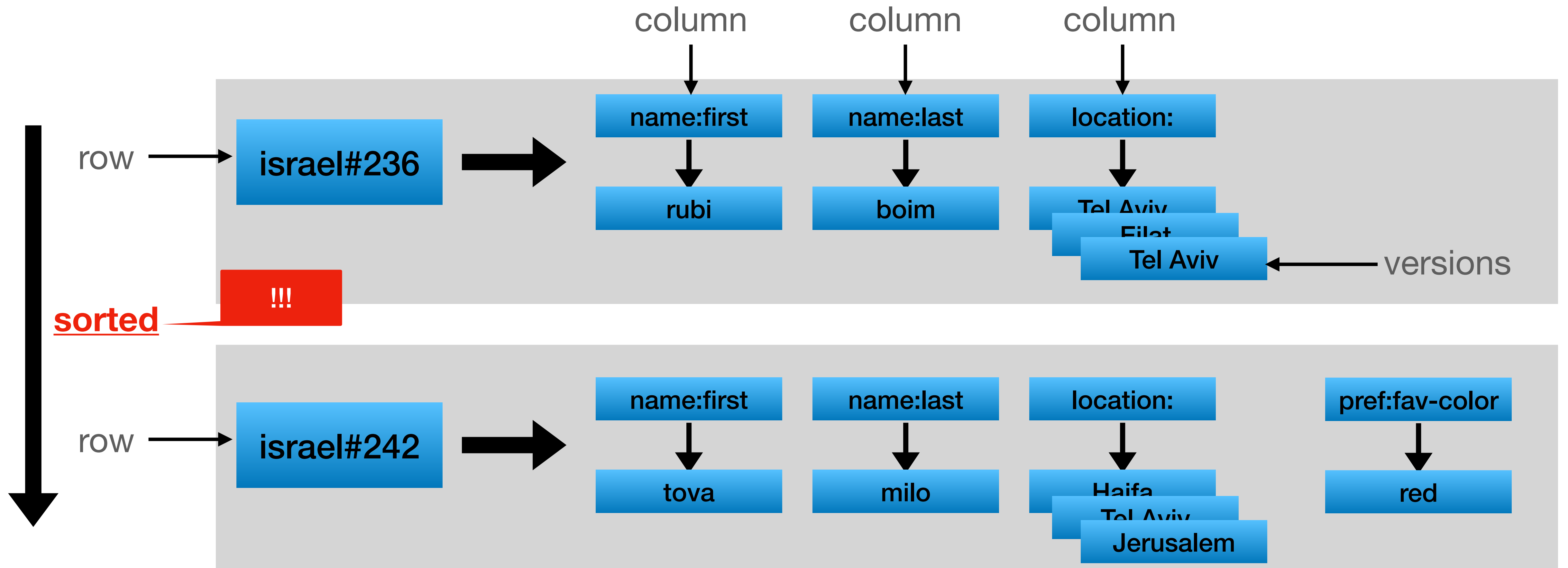
Bigtable

- “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