

# *NoSQL databases II*

Francieli ZANON BOITO

# This class

- Last week: general motivations and principles for NoSQL
  - Today:
    - Amazon's Dynamo (2007)
    - Google's BigTable (2006)
    - Neo4j
  - Next week: lab session about Neo4j
- 
- Go deeper into the design of storage solutions  
Inspired more recent systems*

*"Dynamo: Amazon's Highly Available  
Key-Value Store"*

# *Amazon's Dynamo*

- Paper from 2007
- Key-value store (key and values are binary objects, values are usually < 1MB)
- Many services only need primary-key access
- High reliability requirements, sacrifice consistency (remember the CAP theorem!)
- Tight latency and throughput requirements
- Example: Shopping Cart System (tens of millions of requests in a single day)
  - Has to be always available, specially for writes
- Some key principles:
  - Incremental scalability
  - Heterogeneity
  - Symmetry and decentralization (no centralized control, all nodes are the same)

<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing
High availability for writes	Vector clocks with reconciliation during reads
Handling temporary failures	Sloppy quorum and hinted handoff
Recovering from permanent failures	Anti-entropy using Merkle trees
Membership and failure detection	Gossip-based membership protocol and failure detection

<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing

# Modulo-based Hashing

$N_1$

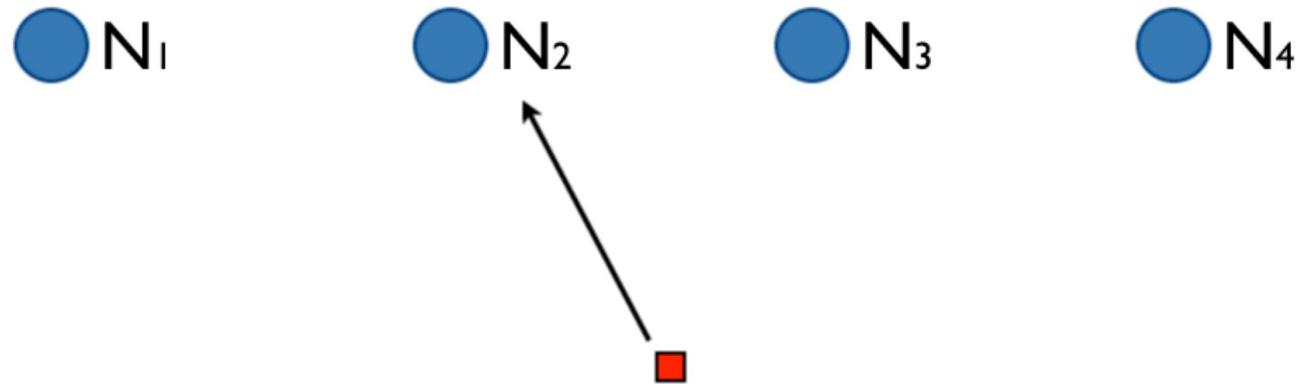
$N_2$

$N_3$

$N_4$



# Modulo-based Hashing



$\text{partition} = \text{key \% n_servers}$

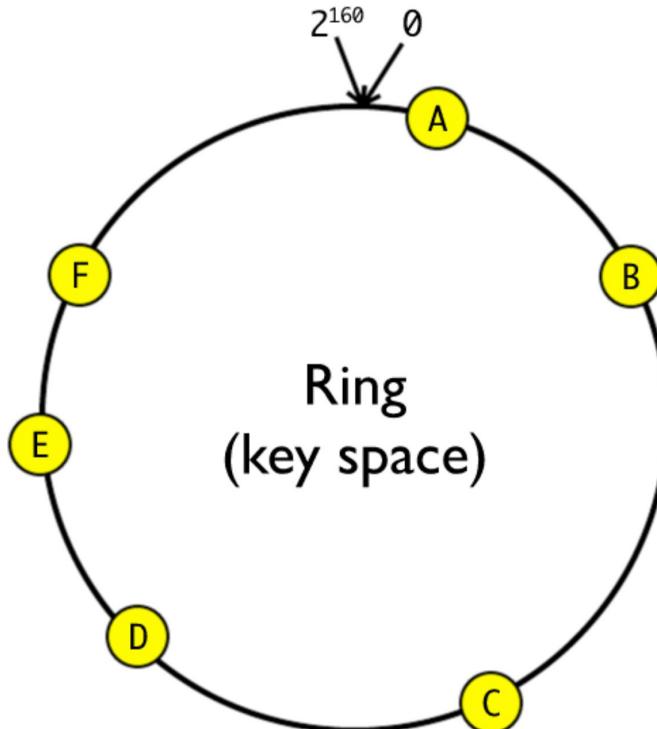
# Modulo-based Hashing



`partition = key % (n_servers - 1)`

Recalculate the hashes for all entries if n\_servers changes  
(full data redistribution when adding/removing a node)

# Consistent Hashing

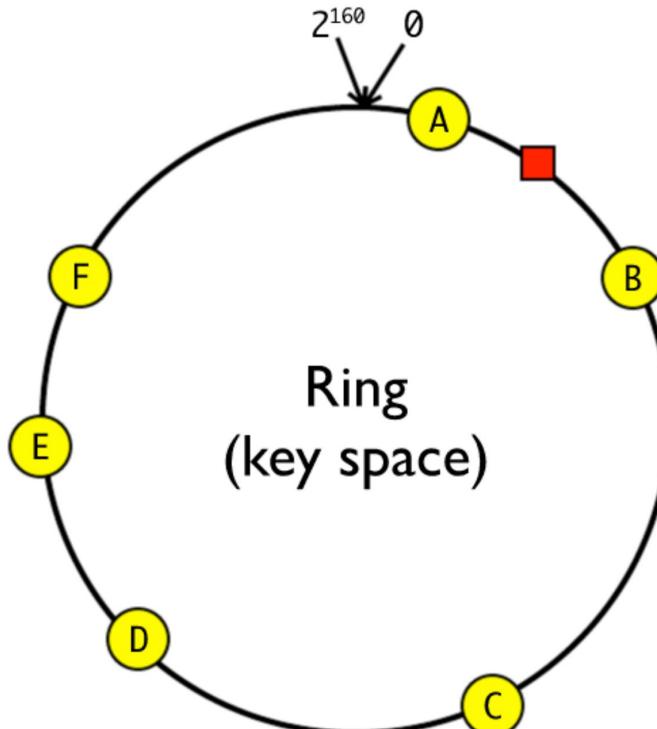


Same hash function  
for data and nodes

`idx = hash(key)`

Coordinator: next  
available clockwise  
node

# Consistent Hashing

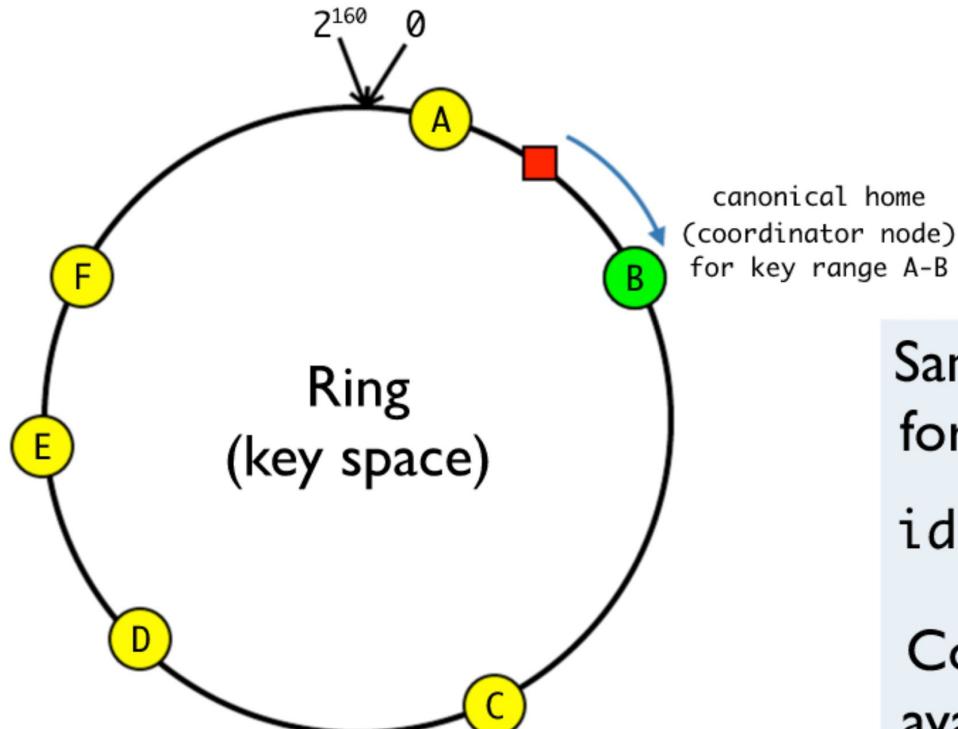


Same hash function  
for data and nodes

$\text{idx} = \text{hash}(\text{key})$

Coordinator: next  
available clockwise  
node

# Consistent Hashing

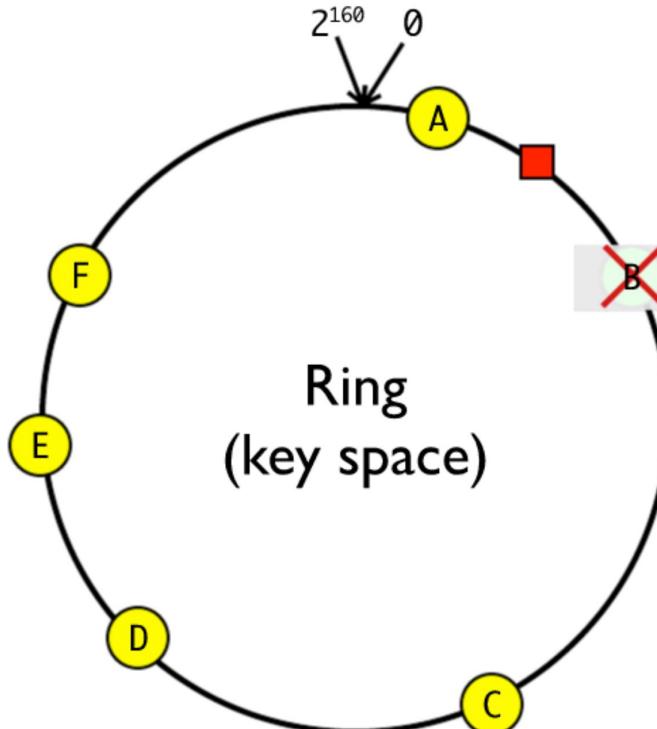


Same hash function  
for data and nodes

$\text{idx} = \text{hash}(\text{key})$

Coordinator: next  
available clockwise  
node

# Consistent Hashing

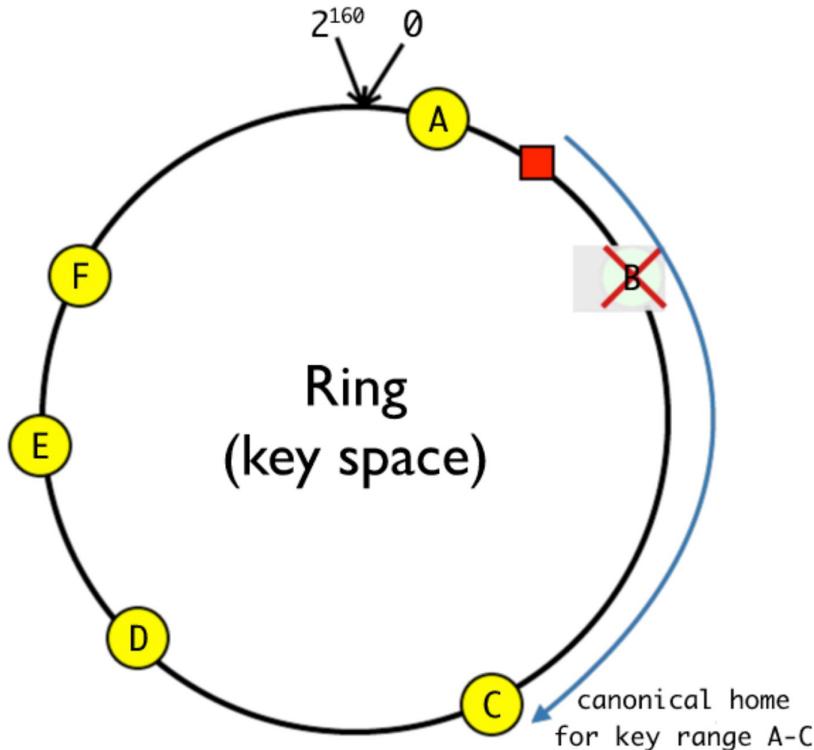


Same hash function  
for data and nodes

`idx = hash(key)`

Coordinator: next  
available clockwise  
node

# Consistent Hashing

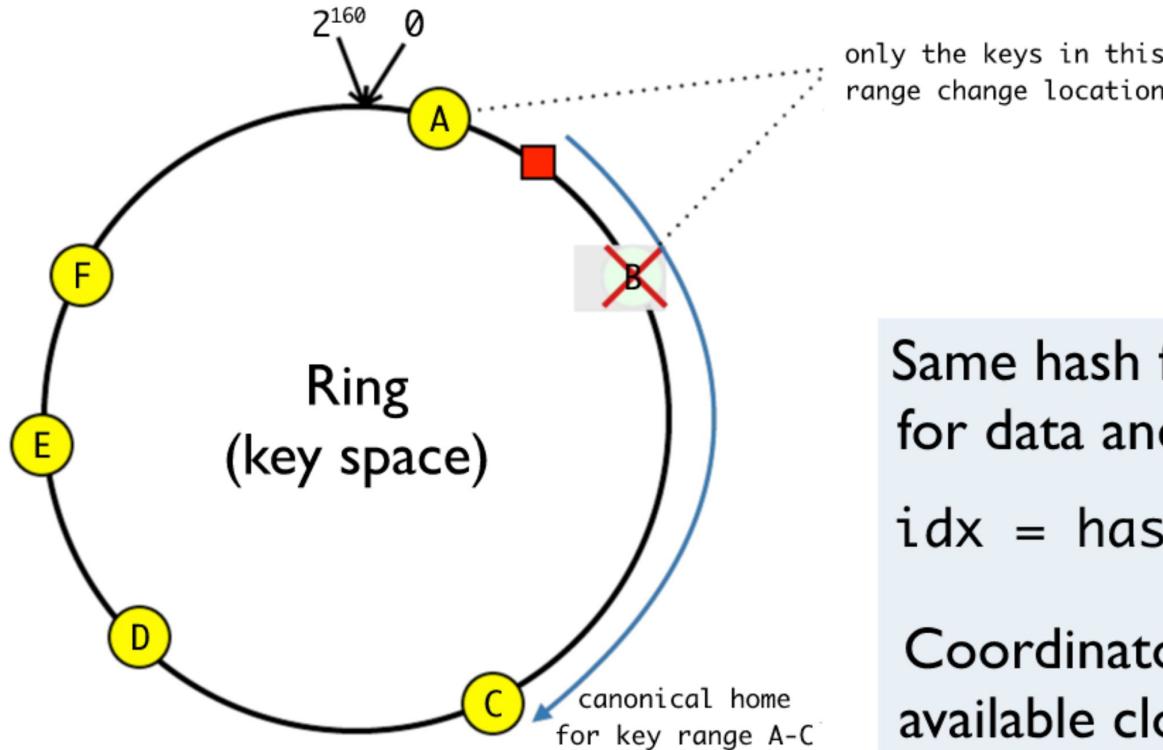


Same hash function  
for data and nodes

`idx = hash(key)`

Coordinator: next  
available clockwise  
node

# Consistent Hashing



Same hash function  
for data and nodes

`idx = hash(key)`

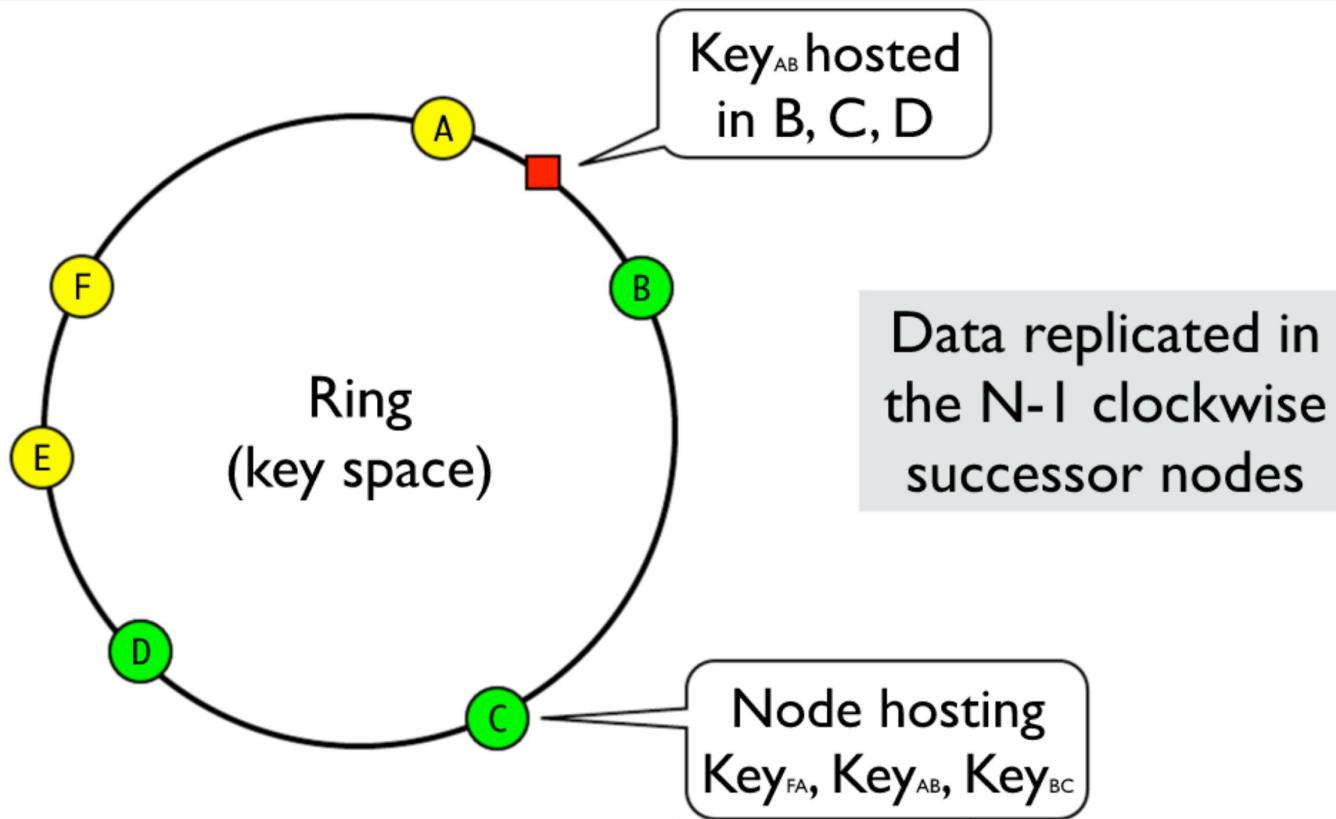
Coordinator: next  
available clockwise  
node

# *Virtual nodes variation*

- Consistent hashing using virtual nodes ( $\#\text{virtual nodes} > \#\text{nodes}$ )
- If a node disappears, its load is distributed across remaining nodes
- A new node receives load from all other nodes
- Accounts for heterogeneity

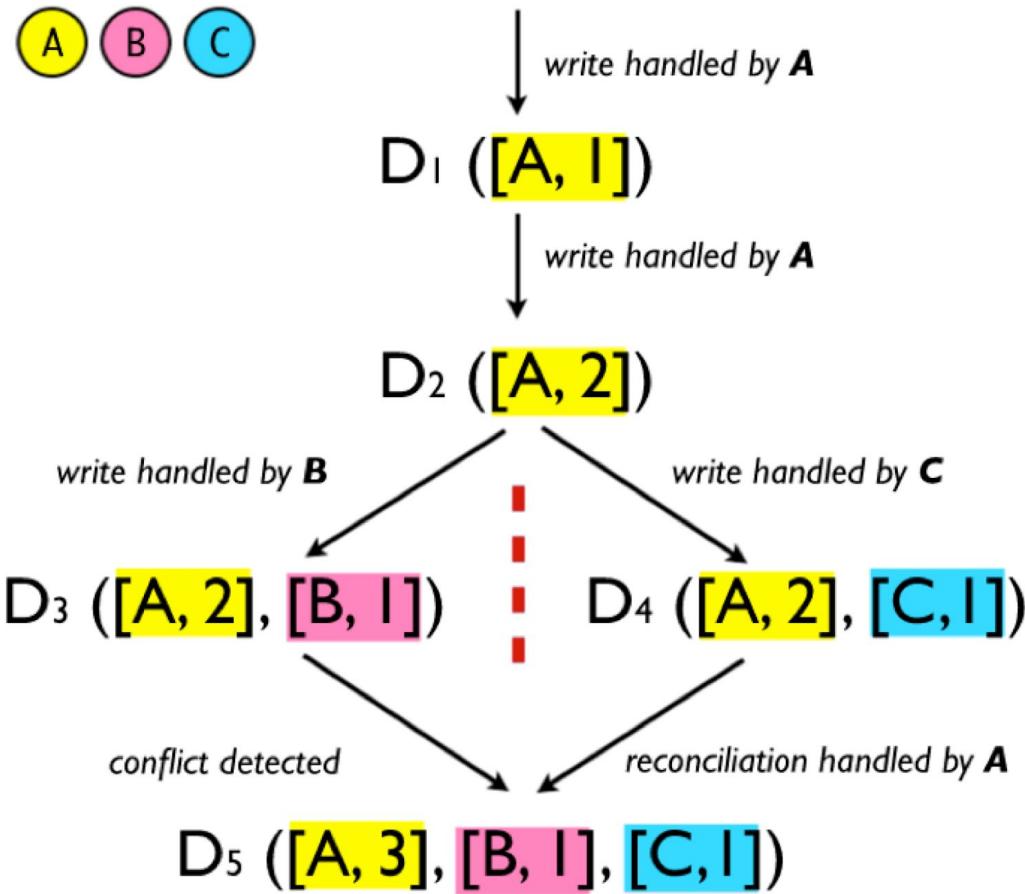
<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing
High availability for writes	Vector clocks with reconciliation during reads

# Consistent Hashing - Replication



# *Eventual consistency*

- Updates are asynchronously propagated to replicas
- Each update makes a new immutable version of data (multiple versions may co-exist)
- Ordering of versions: vector clocks
- Read returns a context structure containing the vector clock
  - Subsequent updates need to specify the version
  - If Dynamo cannot reconcile, read returns a list of versions (the application will handle)



Causality-based partial order over events that happen in the system.

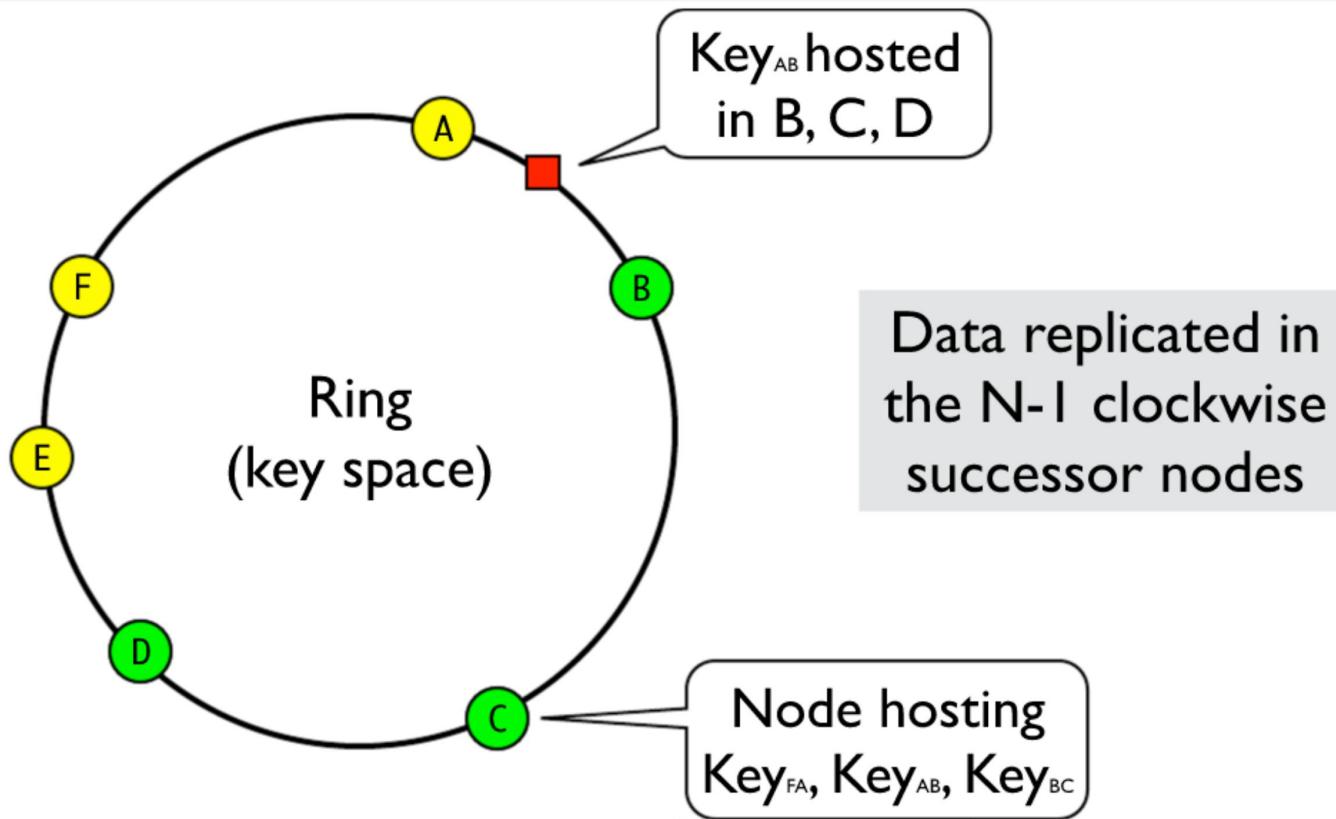
Document version

history: a counter for each node that updated the document.

If all update counters in  $V_1$  are smaller or equal to all update counters in  $V_2$ , then  $V_1$  precedes  $V_2$ .

<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing
High availability for writes	Vector clocks with reconciliation during reads
Handling temporary failures	Sloppy quorum and hinted handoff

# Consistent Hashing - Replication



# Consistency: Server-side (Quorum)

**N** = number of nodes with a replica of the data

**W** = number of replicas that must acknowledge the update<sup>(\*)</sup>

**R** = minimum number of replicas that must participate in a successful read operation

*(\*) but the data will be written to N nodes no matter what*

$W + R > N \longrightarrow$  Strong consistency (usually  $N=3, W=R=2$ )

$W = N, R = 1 \longrightarrow$  Optimised for reads

$W = 1, R = N \longrightarrow$  Optimised for writes

(durability not guaranteed in presence of failures)

$W + R \leq N \longrightarrow$  Weak consistency

# Sloppy Quorum

- Preference list: list of nodes containing replicas of a piece of data
  - Actually perform more than N replicas
  - Try to ensure replicas on different nodes and different data centers
- All nodes know the preference list! Forward requests to one of the first N
- Reads and writes involve the first N **healthy** replicas
  - “Sloppy quorum”: it might include nodes that are not in the top N (if some have failed)
  - The coordinator writes to local disk and propagates to others
    - Waits for  $(R-1)$  or  $(W-1)$  acks
    - $R+W > N$ ,  $R < N$  and  $W < N$
    - $R$  and  $W$  are configurable and adjust a trade-off between durability and availability

# *Hinted handoff*

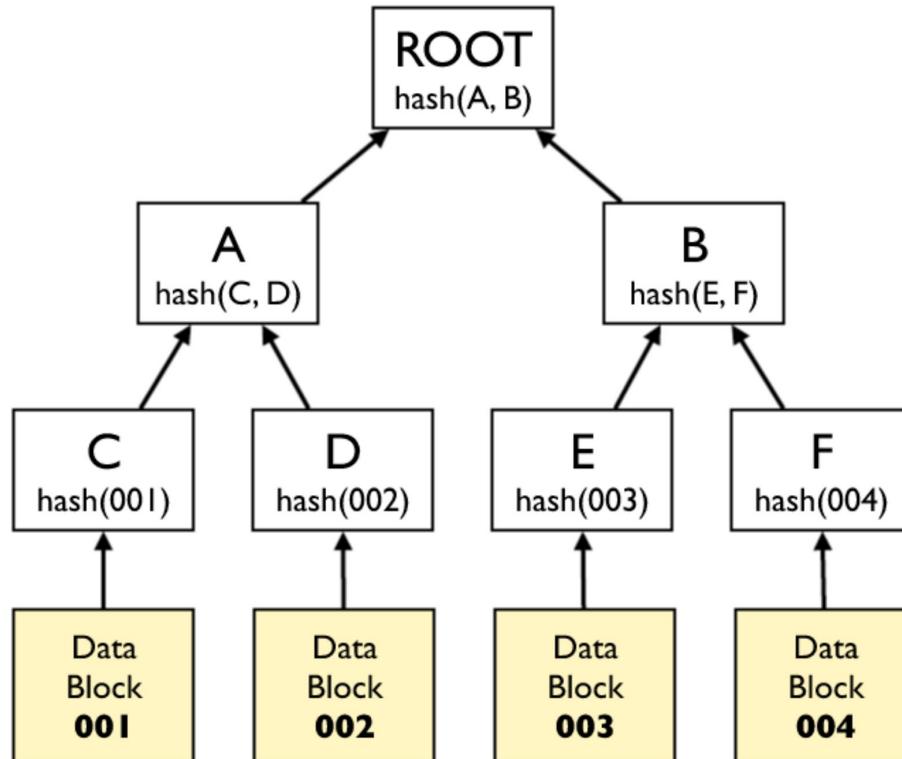
- When a node receives a write but it is not in the top N (preference list)
  - It comes with a “hint” of what node from the top N was supposed to receive that
  - Store that data separately to send to the right node when it comes back

<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing
High availability for writes	Vector clocks with reconciliation during reads
Handling temporary failures	Sloppy quorum and hinted handoff
Recovering from permanent failures	Anti-entropy using Merkle trees

# *Anti-entropy (replica synchronization)*

- Improve durability (for more permanent failures)
- A Merkle tree has stored entries as leaves (the hash of the value)
  - Each node has a hash of its children
  - To compare trees: if two nodes have the same hash, no need to check the sub-trees
    - No need to send the whole tree through the network
- Each node keeps a Merkle tree per key range it stores

# Merkle Trees (Hash Trees)



**Leaves:** hashes of data blocks.

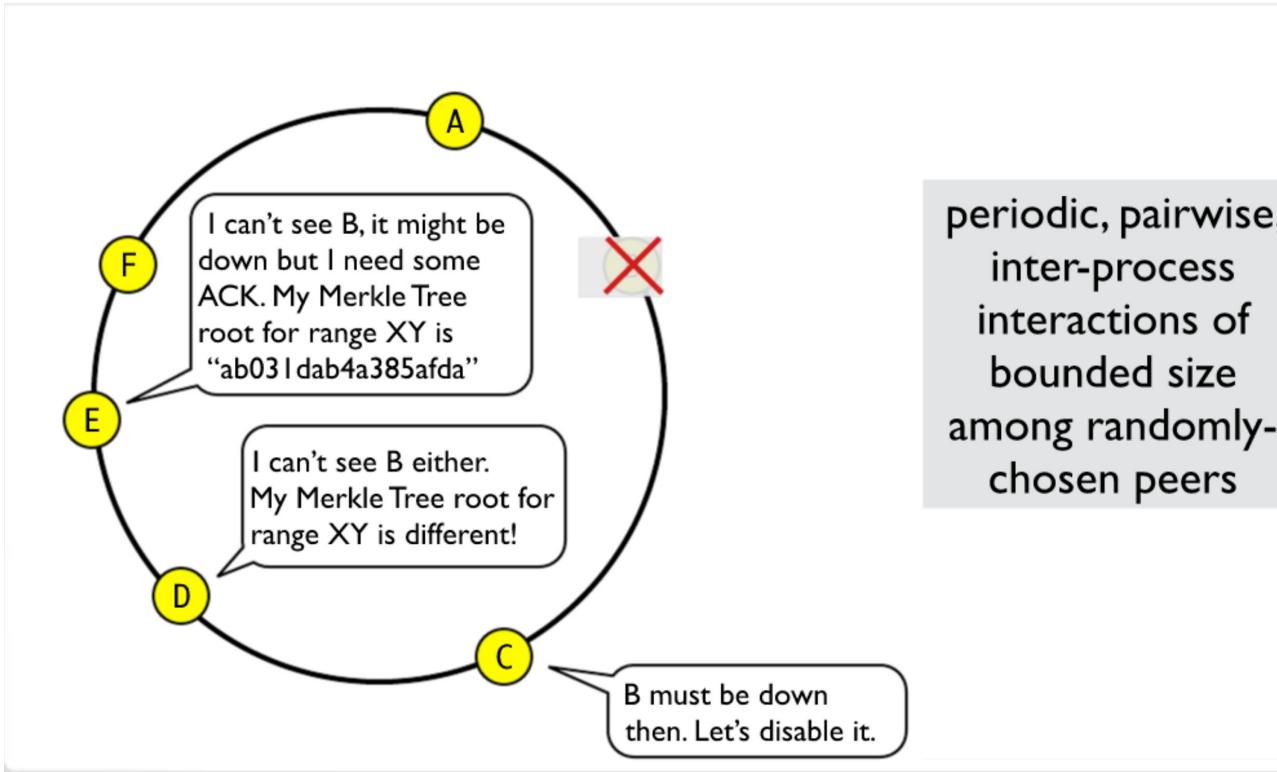
**Nodes:** hashes of their children.

Used to detect inconsistencies between replicas (anti-entropy) and to minimise the amount of transferred data

<b>Problem</b>	<b>How do they solve it?</b>
Partitioning	Consistent hashing
High availability for writes	Vector clocks with reconciliation during reads
Handling temporary failures	Sloppy quorum and hinted handoff
Recovering from permanent failures	Anti-entropy using Merkle trees
Membership and failure detection	Gossip-based membership protocol and failure detection

# Gossip strategy

- When adding or removing a node, contact any node already in the system
  - The node stores this information locally
  - The new node decides its ring locations and stores it locally
  - Some nodes can be discovered by an external mechanism (seeds)
- Periodic random exchanges between nodes to share information (gossip)
  - Nodes in the system
  - Ring formation
  - Merkle tree root
  - (that is how everybody knows the preference list for all keys)



# *Impact*

- The implementation was never released
- But inspired other implementations
  - Voldemort (powered LinkedIn until August 2018)
- Evolution: DynamoDB (key-value and documents) announced in 2012



*"Bigtable: A Distributed Storage System  
for Structured Data"*

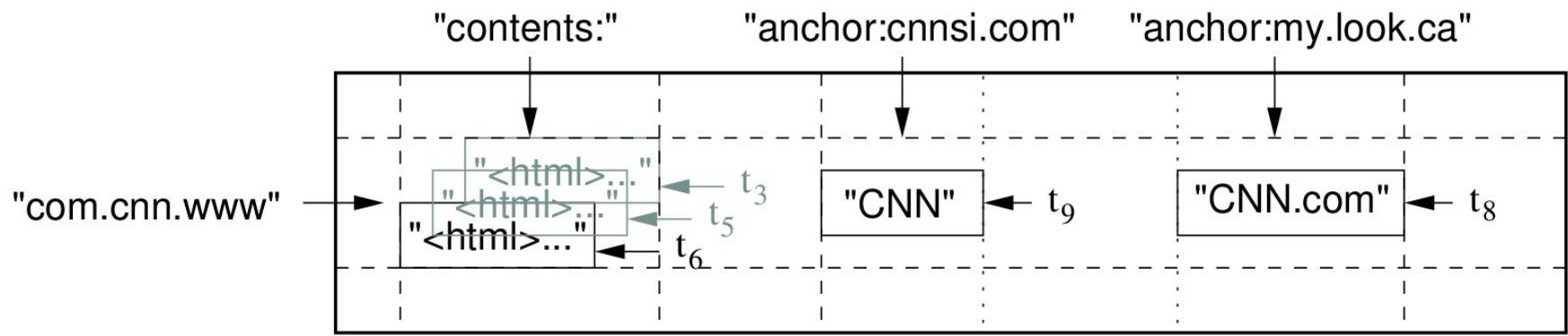
# *Bigtable*



- At the time: over 60 products and projects (Orkut, Google Analytics, Google Earth, ...)
  - Integrated with Google's MapReduce implementation
- Big table: a sparse, distributed, persistent, multidimensional sorted map
  - Indexed by row key, column key, and a timestamp (to identify a version)
  - All fields are uninterpreted arrays of bytes

# Data model

- Data is stored in lexicographic order by row key
  - Dynamically partitioned in "tablets"
  - Users can plan their keys so they will be stored together
  - Example: Webtable has reversed URLs as keys ("com.google.maps" instead of "maps.google.com")
- Column families need to be created before receiving data
  - Each family has an unbounded number of column keys
  - Access control, data locality, and garbage collection setting are decided per column family
- Versions are stored in decreasing timestamp order (finding the most recent is faster)
- Operations over a row are atomic



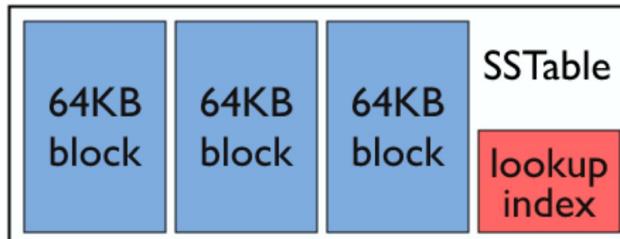
# Google BigTable: Data Structure

## SSTable

Smallest building block

Persistent immutable Map[k,v]

Operations: lookup by key / key range scan



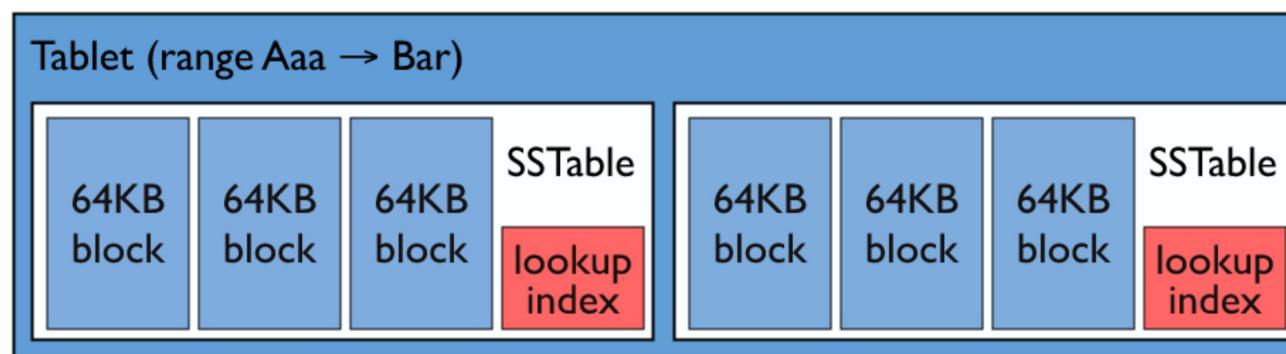
# Google BigTable: Data Structure

## Tablet

Dynamically partitioned range of rows

Built from multiple SSTables

Unit of distribution and load balancing



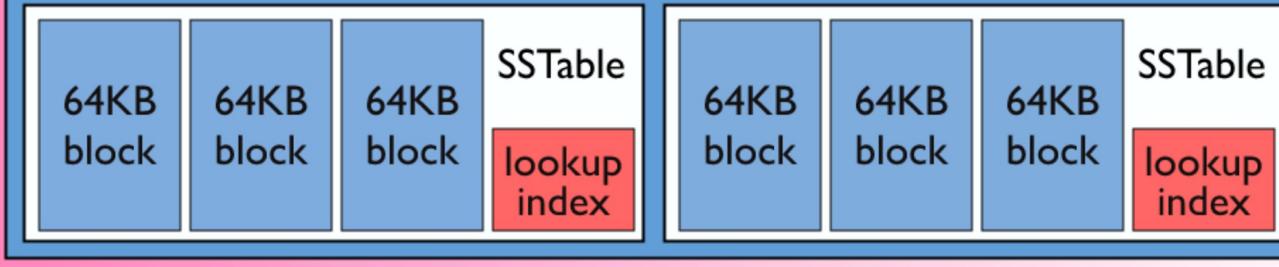
# Google BigTable: Data Structure

## Table

Multiple Tablets (table segments) make up a table

### Table

Tablet (range Aaa → Bar)



# Architecture

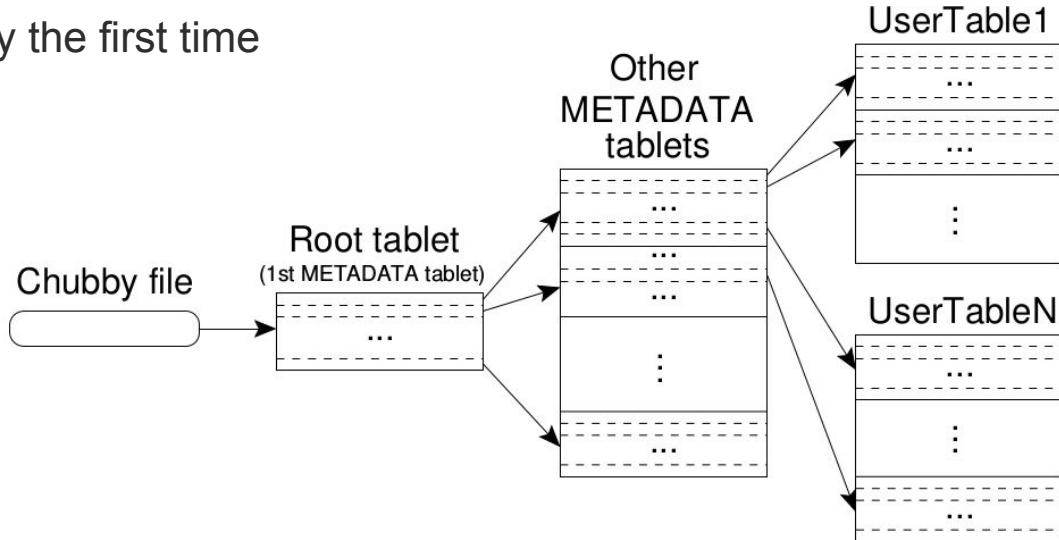
- Single master and multiple tablet servers
  - Tablet servers are added and removed to accommodate to the workload
- The master:
  - Assigns tablets to servers, load balancing
  - Detects addition or removal of servers
  - Garbage collection, table and column family creations
- Read and write DO NOT go through the master
  - Most clients never access the master

# Implementation

- Relies on Chubby: a distributed lock manager
  - A limited distributed file system for small files
  - Intended use: locks that will be kept for hours to days, not for seconds (low performance)
- Master keeps a master lock on Chubby
  - Finds existing tablet servers (then periodically contacts them)

# Metadata management

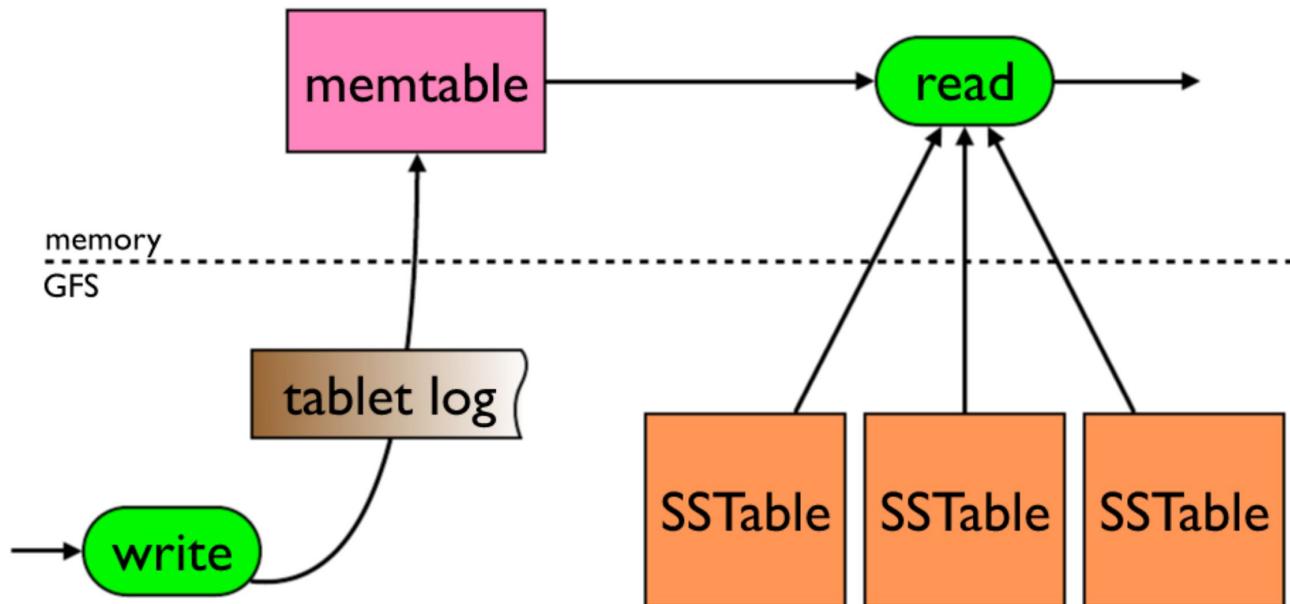
- How to know where are all the tablets?
  - A metadata table!
- The root tablet is never split to keep the hierarchy into three levels
- Clients have to access chubby the first time
  - Cache and prefetch



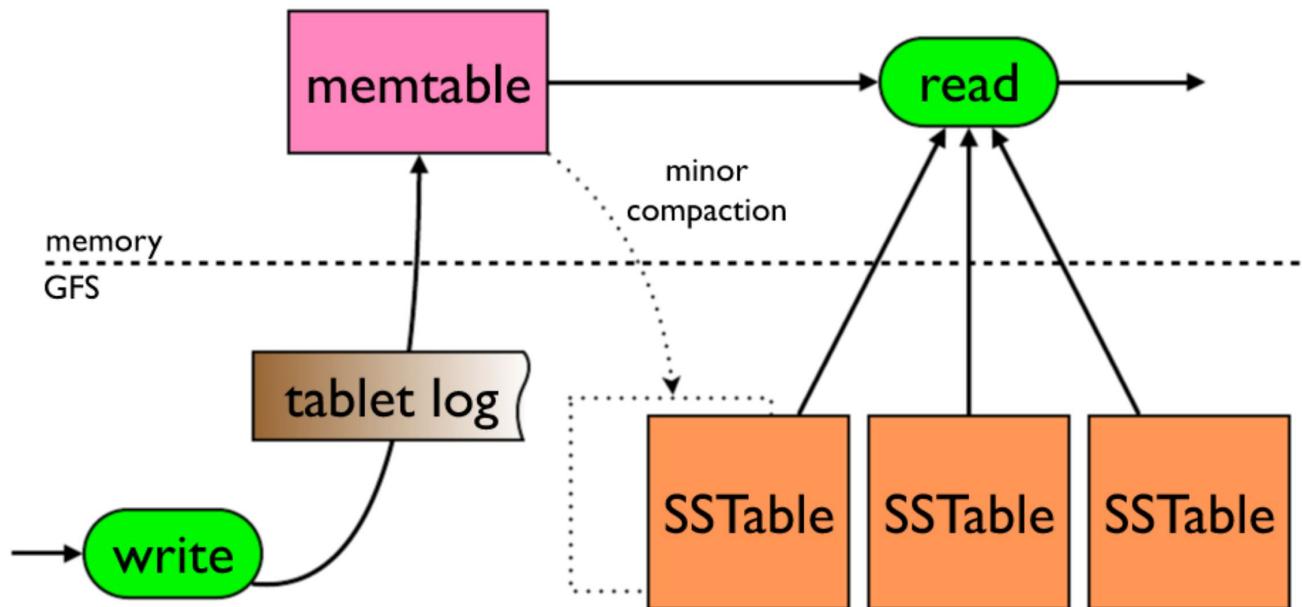
# Persistency

- SSTables are written to the Google File System (GFS)
  - Fault tolerance is provided by the GFS
  - Cached in the tablet server
- Keep a log of performed modifications (also in the GFS)
  - Tablet metadata includes pointers to these logs
  - Recent writes in memory (memtable), older ones in new SSTables

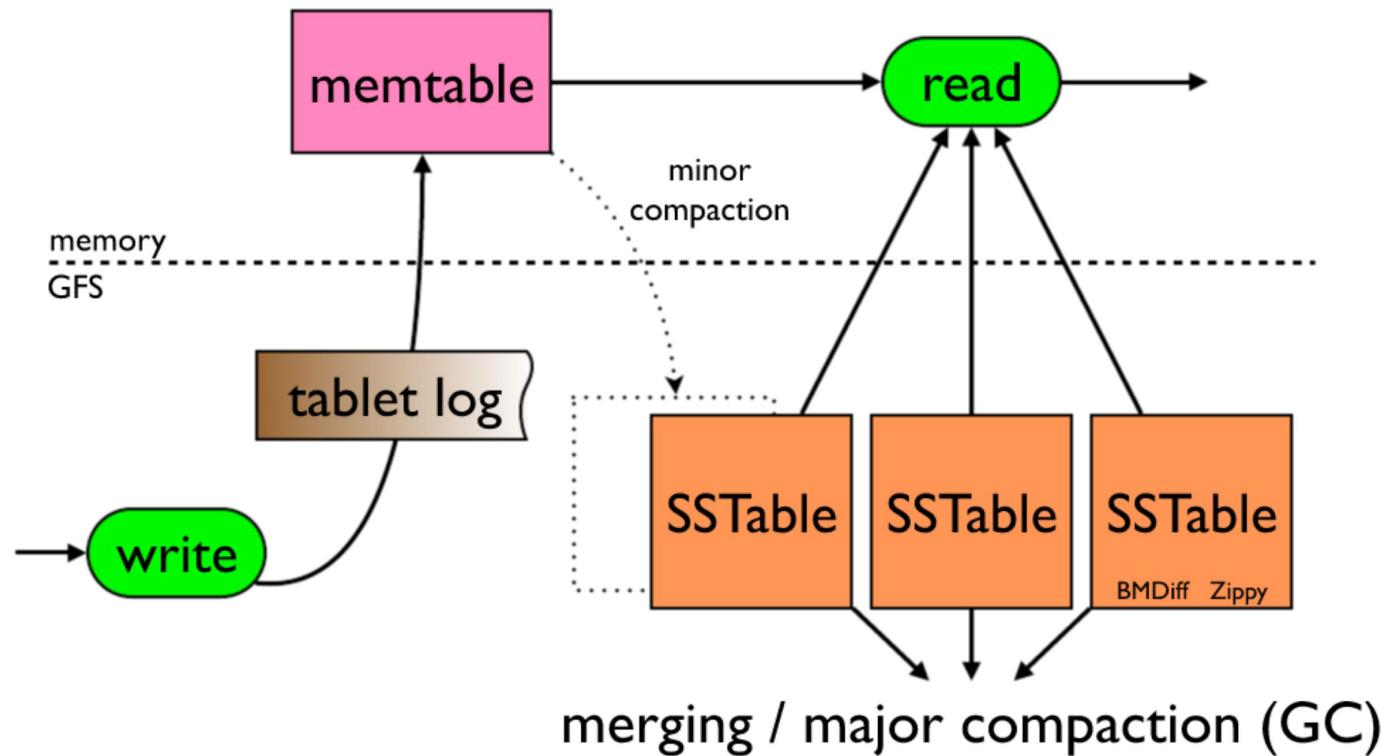
# Google BigTable: I/O



# Google BigTable: I/O



# Google BigTable: I/O



# Performance refinements

- Locality groups
  - Store column families in separate SSTables
  - Example: separate information about the webpage from its content
  - Keep it in memory (as done with the location column of the metadata table)

# *Impact*

- Open source implementation: Apache HBase
  - HDFS instead of GFS, Zookeeper instead of Chubby
- Apache Cassandra: data model of Bigtable, infrastructure of Dynamo

# *An Introduction to Neo4j*

# Neo4j



- Graph database
- Nodes, relationships, attributes of nodes and relationships
- Relationships are directional
- Example of use: recommendation systems (Walmart and Lufthansa\*)
- Cypher Query Language
- HTTP API, Bindings for .net, Java, JavaScript, and Python

\* <https://www.youtube.com/watch?v=Yzbk6VaavoM>

*Now we'll see a demo*

# Cypher Query Language

*Query statement with multiple clauses*

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

# Cypher Query Language

*MATCH = find*

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

# Cypher Query Language

"m" is an alias  
to use it later  
in the  
statement,  
like here

```
MATCH (m:Movie)-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

( ) denote a node  
":Name" is the type of node

# Cypher Query Language

*-[]-> is a relationship  
":RATED" gives the type of relationship*

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```

# Cypher Query Language

```
MATCH (m:Movie)<-[:RATED]-(u:User)  
WHERE m.title CONTAINS "Matrix"  
WITH m.title AS movie, COUNT(*) AS reviews  
RETURN movie, reviews  
ORDER BY reviews DESC  
LIMIT 5;
```

*Find relationships of type "RATED" to nodes of type "movie" and from nodes of type "user"*

# Cypher Query Language

*Filter results regarding an attribute of the movie node*

```
MATCH (m:Movie)<-[:RATED]-(u:User)  
WHERE m.title CONTAINS "Matrix"  
WITH m.title AS movie, COUNT(*) AS reviews  
RETURN movie, reviews  
ORDER BY reviews DESC  
LIMIT 5;
```

# Cypher Query Language

*COUNT(\*) is an aggregation, it counts the number of found results*

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC
LIMIT 5;
```



# Cypher Query Language

```
MATCH (m:Movie)<-[:RATED]-(u:User)  
WHERE m.title CONTAINS "Matrix"  
WITH m.title AS movie, COUNT(*) AS reviews  
RETURN movie, reviews  
ORDER BY reviews DESC  
LIMIT 5;
```

*The result is a list of movie titles with the number of reviews they have in the database*

# Cypher Query Language

```
MATCH (m:Movie)<-[:RATED]-(u:User)
WHERE m.title CONTAINS "Matrix"
WITH m.title AS movie, COUNT(*) AS reviews
RETURN movie, reviews
ORDER BY reviews DESC ← Ordered by the number of  
reviews in descending order  
(DESC)
LIMIT 5;
```

# Cypher Query Language

```
MATCH (m:Movie)<-[:RATED]-(u:User)  
WHERE m.title CONTAINS "Matrix"  
WITH m.title AS movie, COUNT(*) AS reviews  
RETURN movie, reviews  
ORDER BY reviews DESC  
LIMIT 5;
```

*Only show up to 5 results*

# *Extra materials*

Paper: Burrows et al., "The Chubby lock service for loosely-coupled distributed systems"  
(2006)

A tutorial by Neo4j: <https://www.youtube.com/watch?v=Yzbk6VaavoM>

Neo4j online sandbox: <https://neo4j.com/sandbox/>

Neo4j example with Pokémons: <https://neo4j.com/graphgist/pokemon-and-neo4j>