Big Data and Internet Thinking

Chentao Wu
Associate Professor
Dept. of Computer Science and Engineering
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Download lectures

- ftp://public.sjtu.edu.cn
- User: wuct
- Password: wuct123456

- http://www.cs.sjtu.edu.cn/~wuct/bdit/

Schedule

- lec1: Introduction on big data, cloud computing & IoT
- lec2: Parallel processing framework (e.g., MapReduce)
- lec3: Advanced parallel processing techniques (e.g., YARN, Spark)
- lec4: Cloud & Fog/Edge Computing
- lec5: Data reliability & data consistency
- lec6: Distributed file system & objected-based storage
- lec7: Metadata management & NoSQL Database
- lec8: Big Data Analytics

Collaborators

Google 谷歌

阿里云 aliyun.com

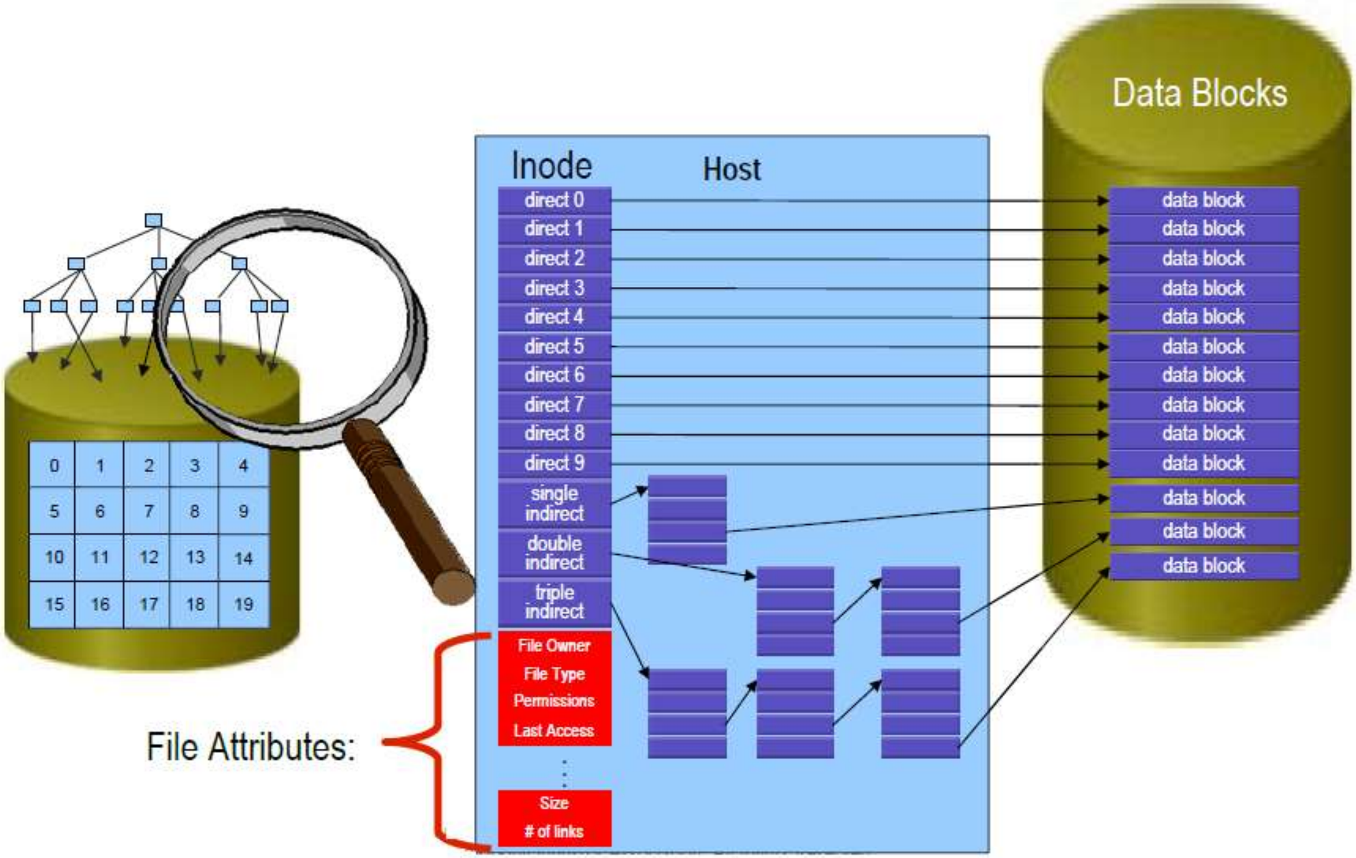
DELL EMC

1

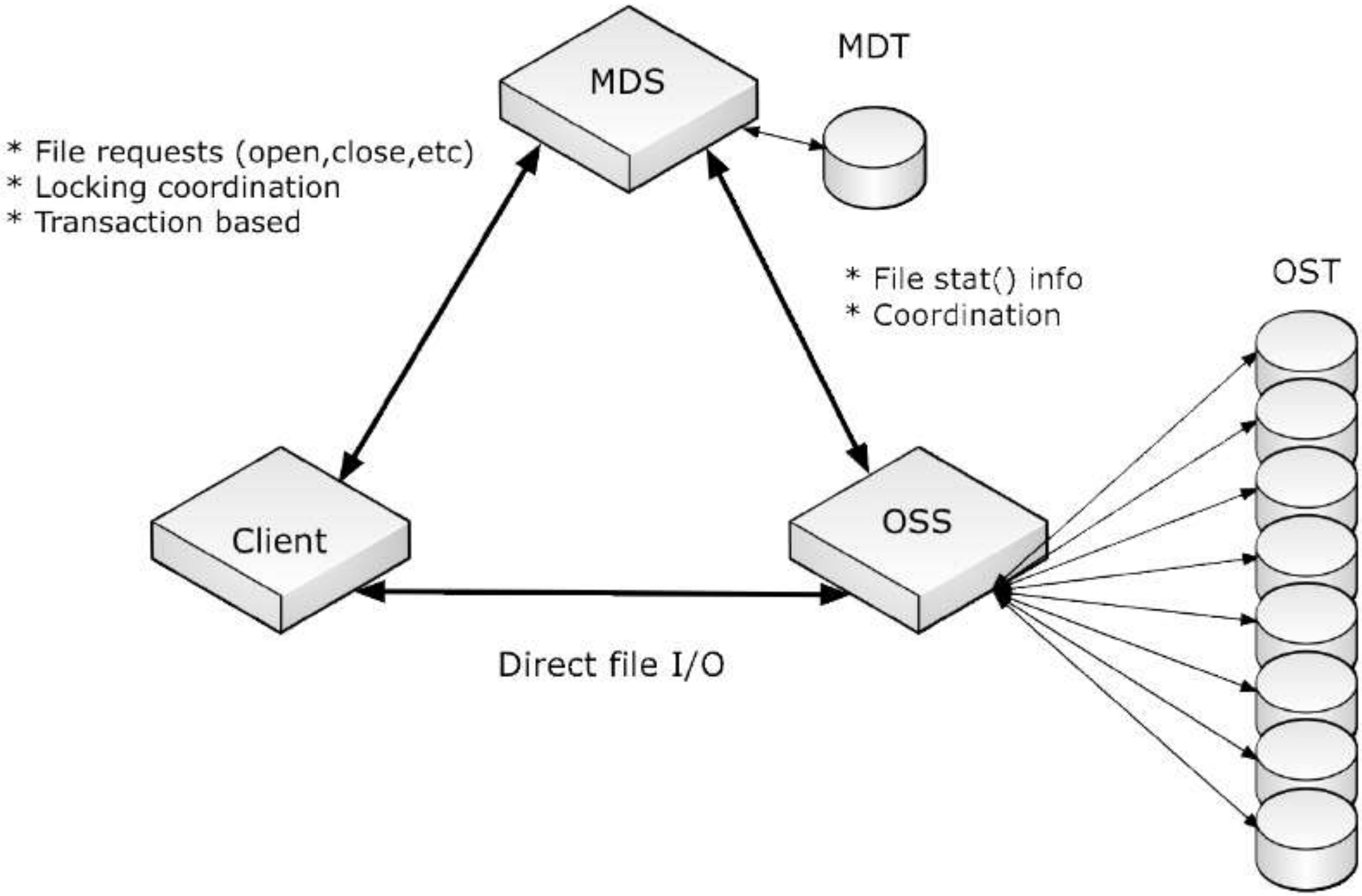
Metadata in DFS

Metadata

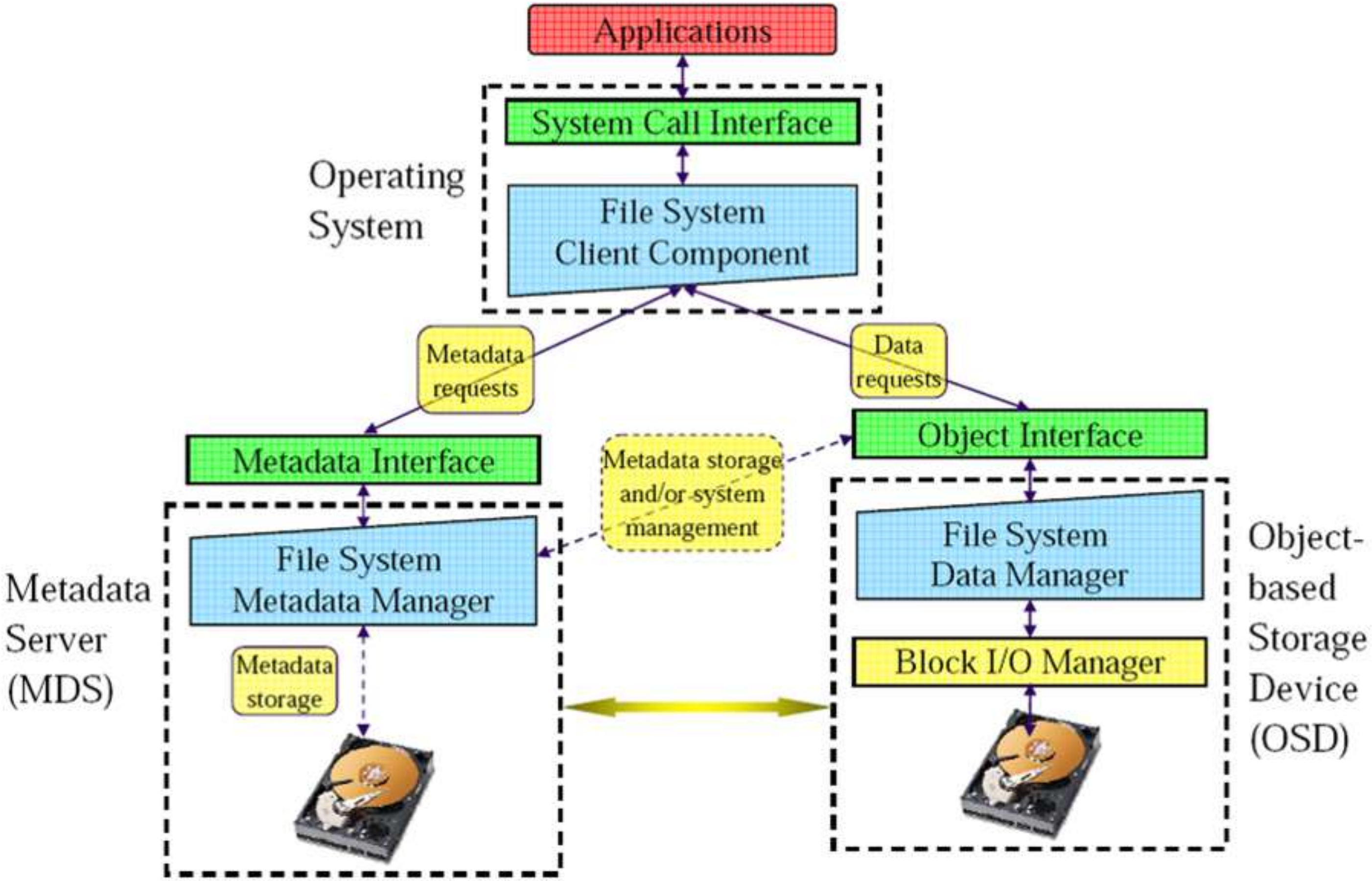
- **Metadata = structural information**
  - File/Objects: attributes in inode/onode
  - Main problem for metadata in DFS: indexing



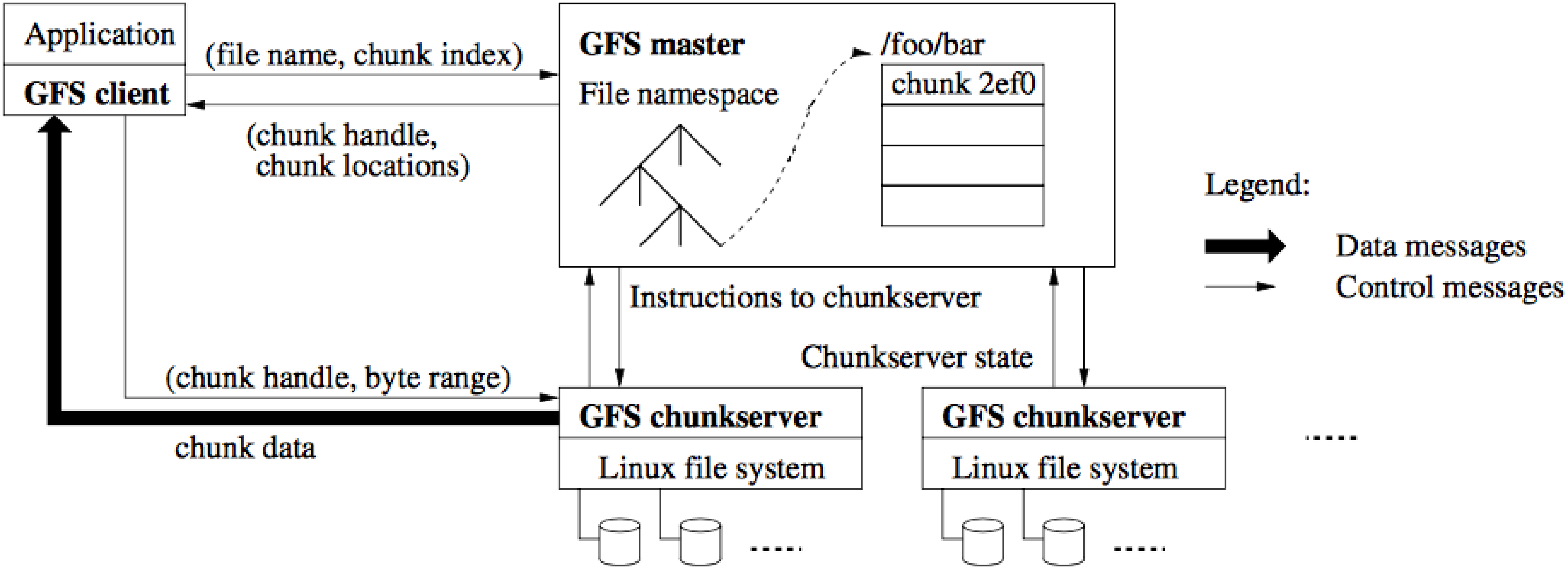
Metadata Server in DFS (Lustre)



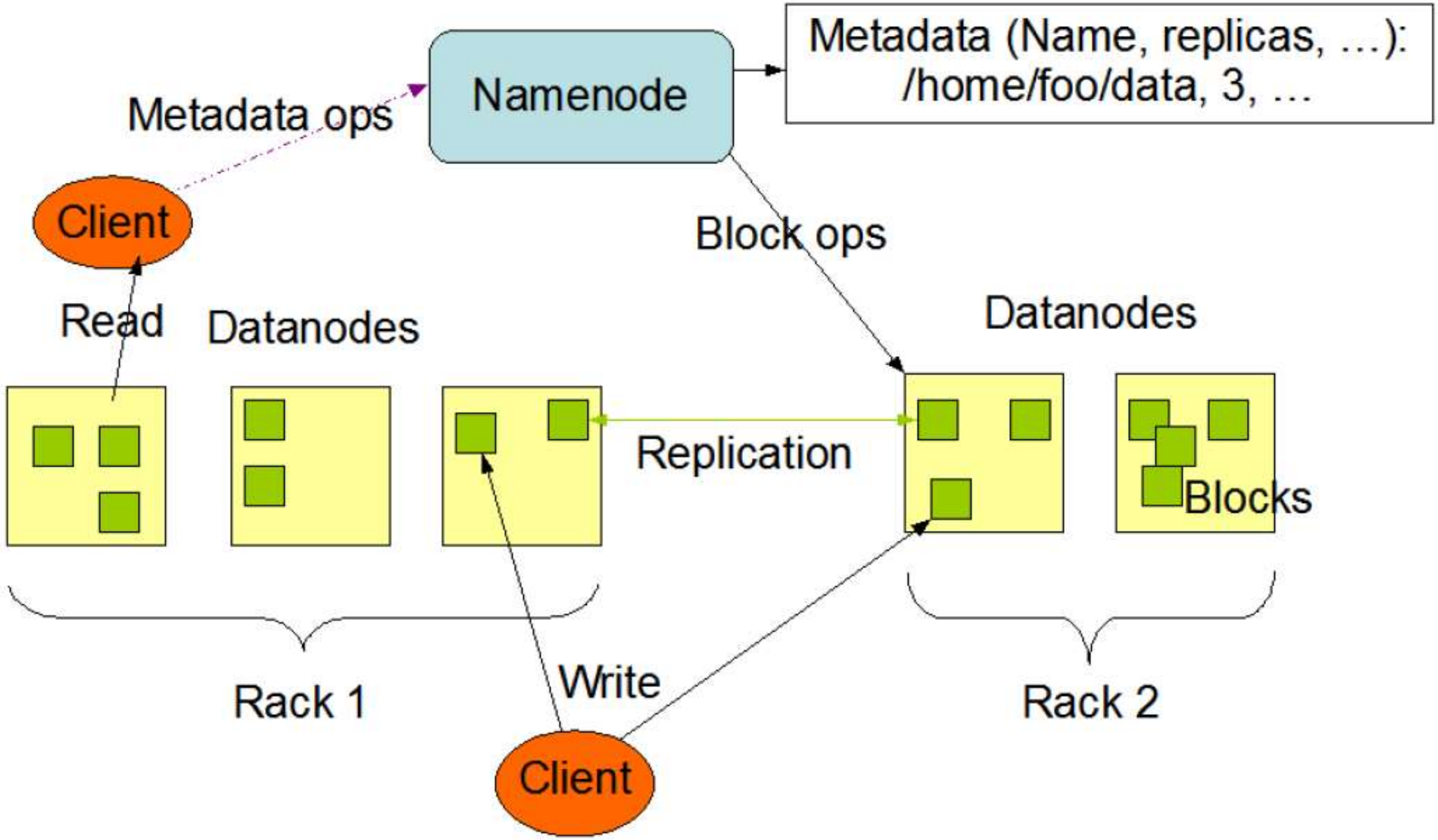
Metadata Server in DFS (Ceph)



Metadata Server in DFS (GFS)



Metadata Server in DFS (HDFS)



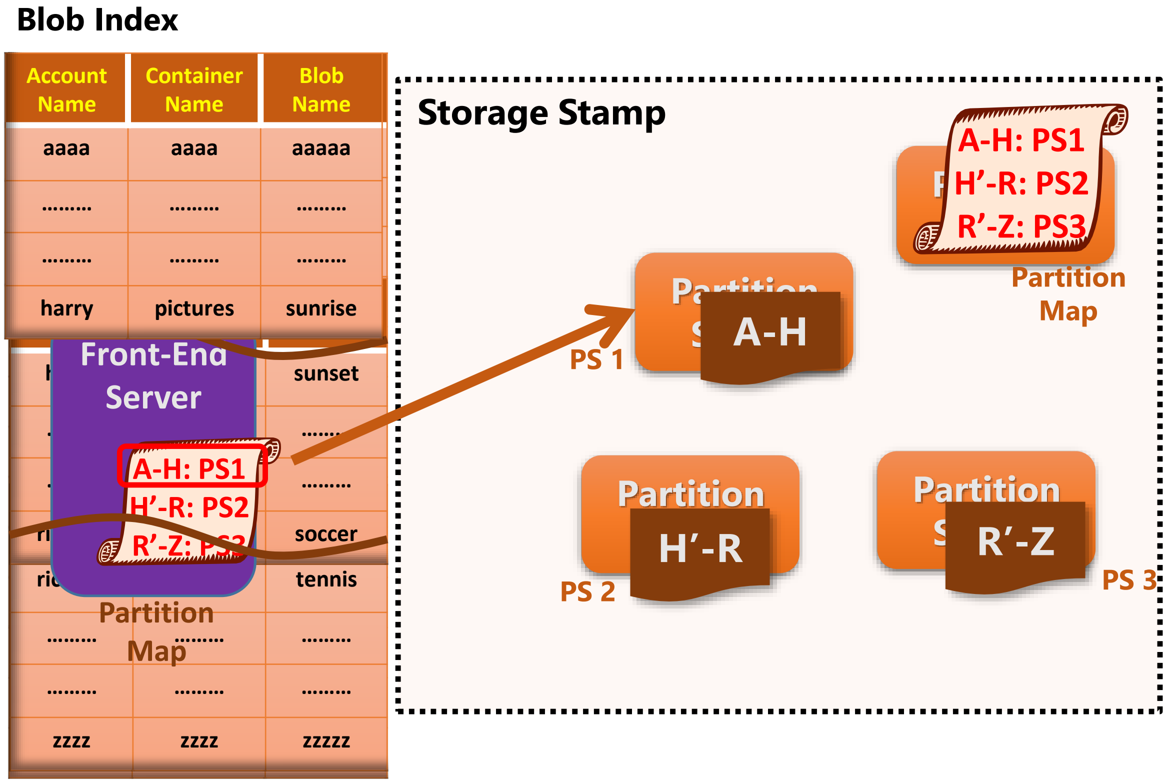
NameNode Metadata in HDFS

- **Metadata in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **Types of Metadata**
  - List of files
  - List of Blocks for each file
  - List of DataNodes for each block
  - File attributes, e.g creation time, replication factor
- **A Transaction Log**
  - Records file creations, file deletions. etc

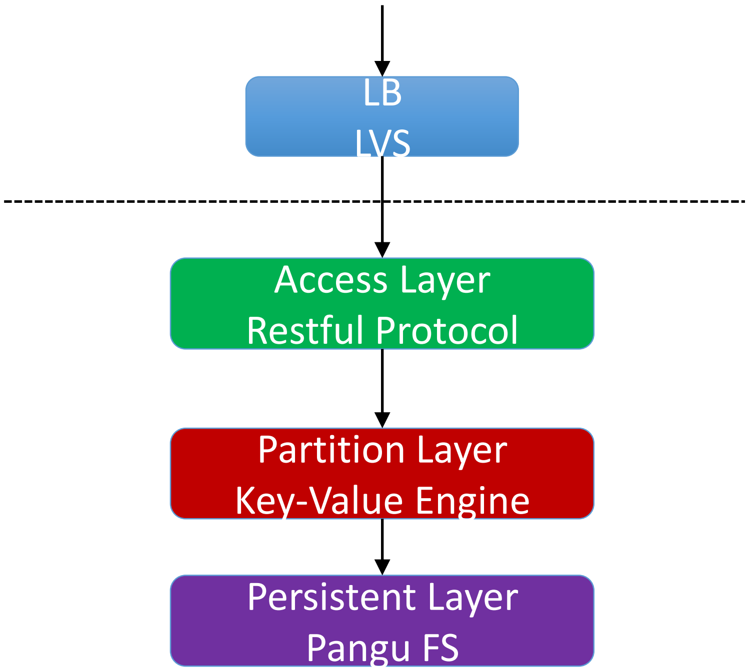
Metadata level in DFS (Azure)
Partition Layer – Index Range Partitioning

Microsoft Azure

- Split index into RangePartitions based on load
- Split at PartitionKey boundaries
- PartitionMap tracks Index RangePartition assignment to partition servers
- Front-End caches the PartitionMap to route user requests
- Each part of the index is assigned to only one Partition Server at a time



Metadata level in DFS (Pangu) Partition layer



Load Balancing

Protocol Manager & Access Control

Partition & Index

Persistent, Redundancy & Fault-Tolerance

2

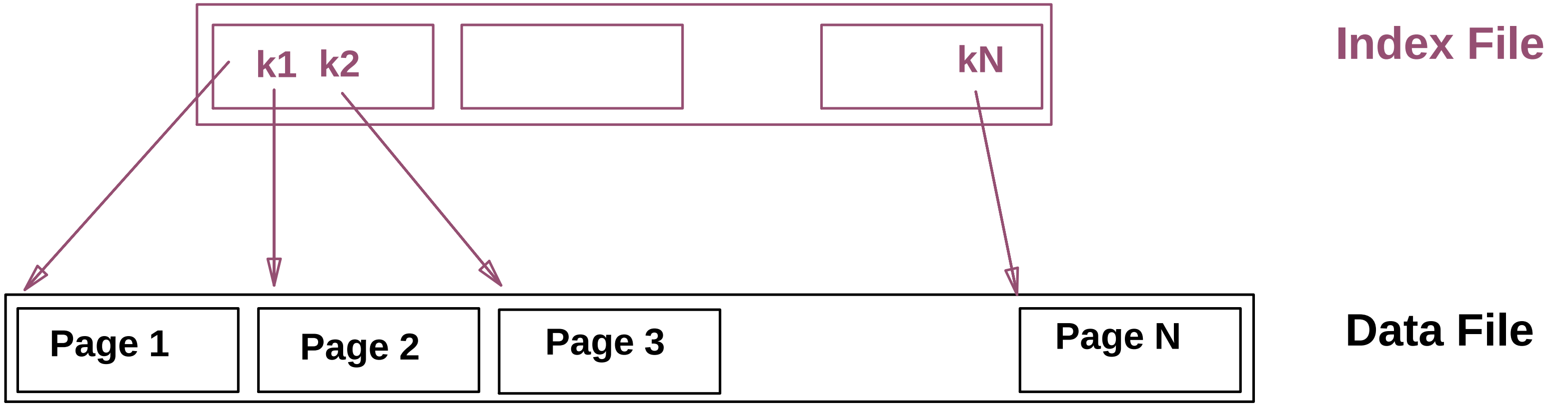
ISAM & B+ Tree

Tree Structures Indexes

- *Recall: 3 alternatives for data entries* k*:
  - Data record with key value **k**
  - <**k**, rid of data record with search key value **k**>
  - <**k**, list of rids of data records with search key **k**>
- Choice is orthogonal to the *indexing technique* used to locate data entries k*.
- Tree-structured indexing techniques support both *range searches* and *equality searches*.
  - *ISAM (Indexed Sequential Access Method)*: static structure
  - *B+ tree*: dynamic, adjusts gracefully under inserts and deletes.

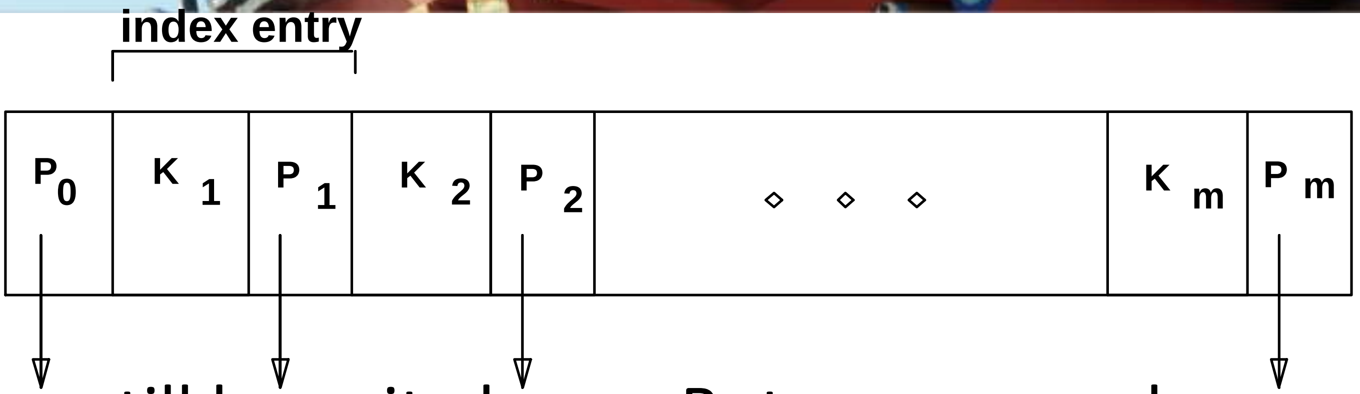
Range Searches

- Choose``*Find all students with gpa > 3.0''*
    - If data is in sorted file, do binary search to find first such student, then scan to find others.
    - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.
    - Level of indirection again!

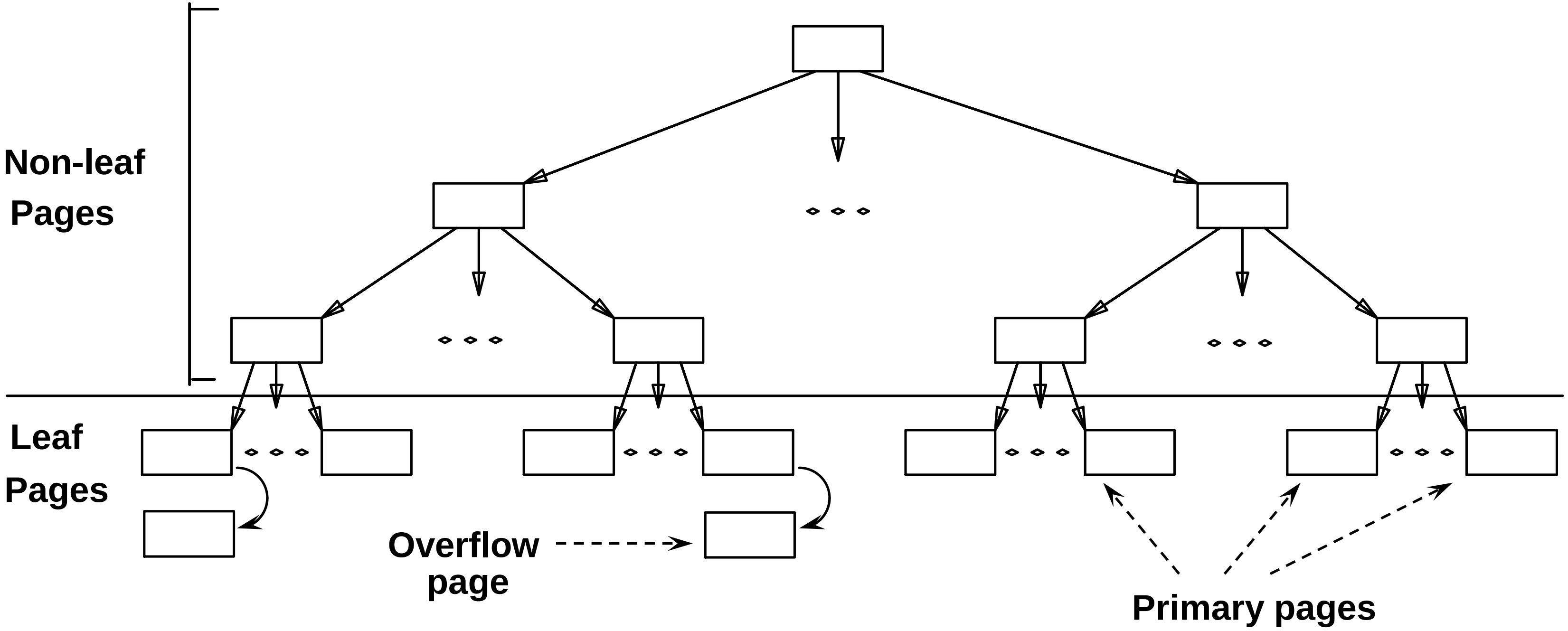


Can do binary search on (smaller) index file!

ISAM



- Index file may still be quite large. But we can apply the idea repeatedly!



Leaf pages contain data entries

Comments on ISAM

Data Pages

Index Pages

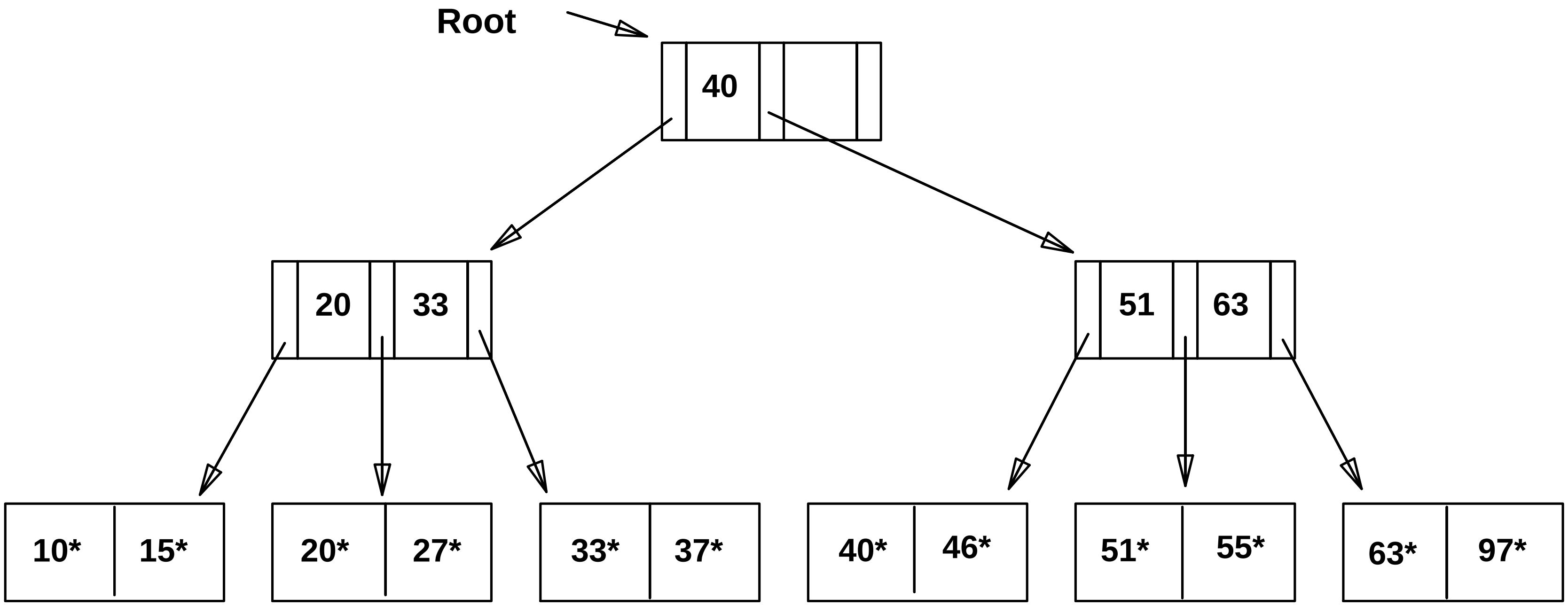
Overflow pages

- *File creation*: Leaf (data) pages allocated sequentially, sorted by search key.
  Then index pages allocated.
  Then space for overflow pages.
- *Index entries*: <search key value, page id>; they `direct' search for *data entries*, which are in leaf pages.
- *Search*: Start at root; use key comparisons to go to leaf. Cost $\log_F N$; F = # entries/index pg, N = # leaf pgs
- *Insert*: Find leaf where data entry belongs, put it there. (Could be on an overflow page).
- *Delete*: Find and remove from leaf; if empty overflow page, de-allocate.

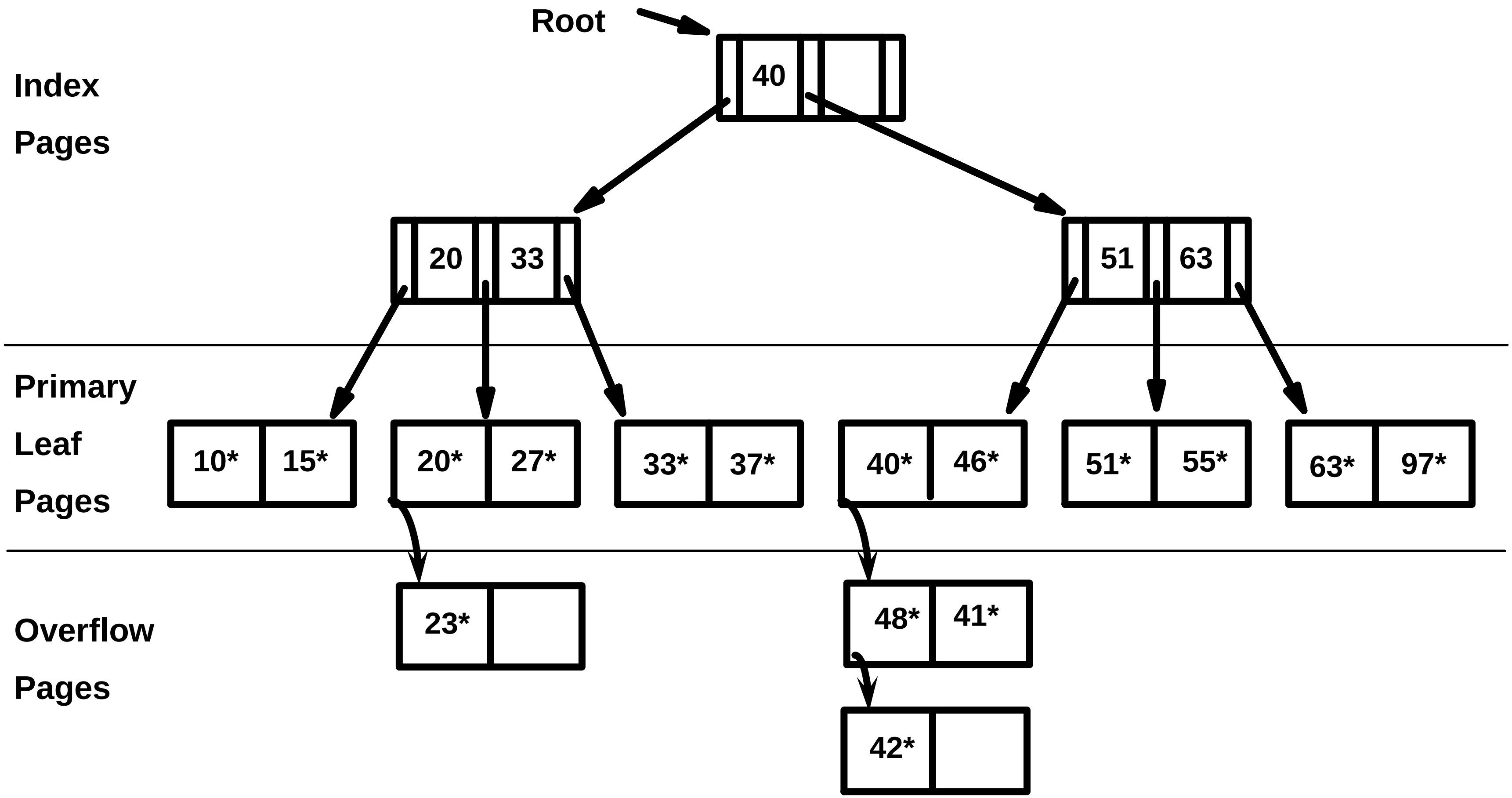
Static tree structure: *inserts/deletes affect only leaf pages*.

Example ISAM Tree

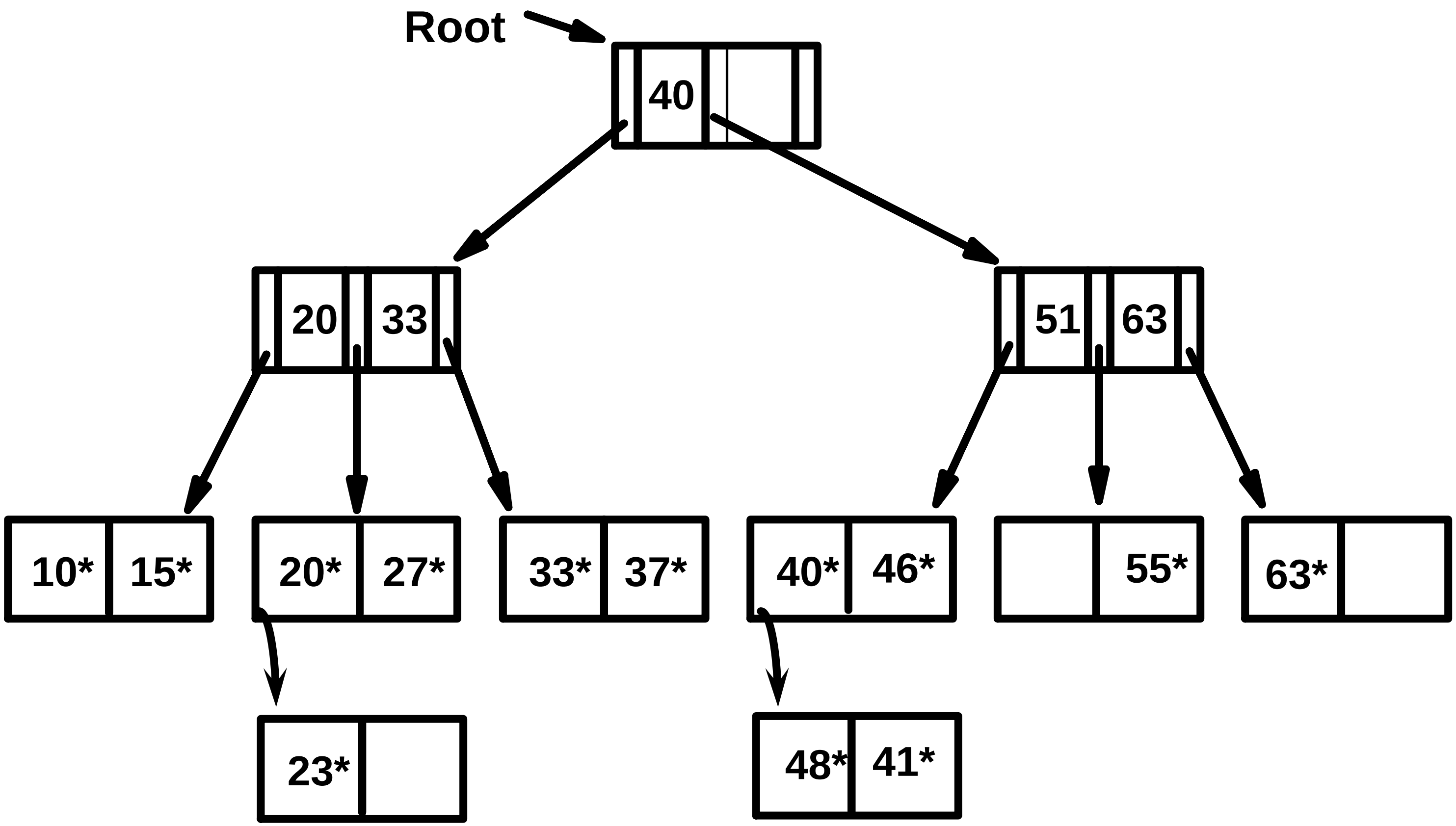
- Each node can hold 2 entries; no need for `next-leaf-page' pointers.



After Inserting 23*, 48*, 41*, 42* ...



... then Deleting 42*, 51*, 97*



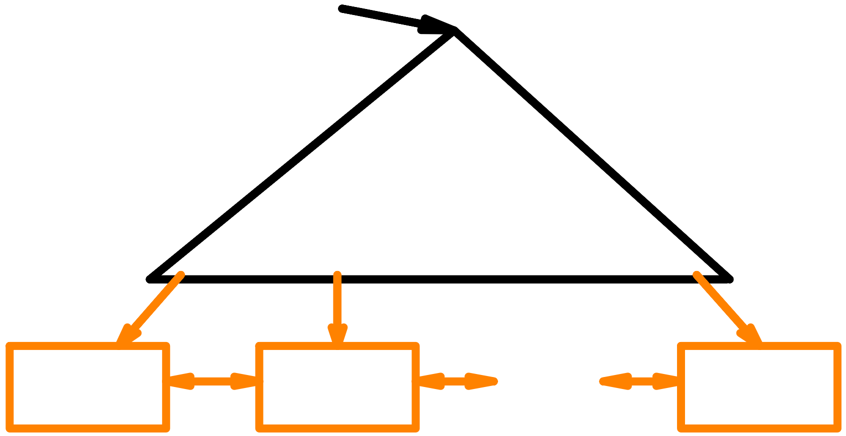
Note that 51 appears in index levels , but 51 not in leaf!*

Pros, Cons & Usage

- Pros
  - Simple and easy to implement
- Cons
  - Unbalanced overflow pages
  - Index redistribution
- Usage
  - MS Access
  - Berkeley DB
  - MySQL (before 3.23) → MyISAM (not real ISAM)

B+ Tree: The Most Widely Used Index

- Insert/delete at log $_F$ N cost; keep tree *height-balanced*. (F = fanout, N = # leaf pages)
- Minimum 50% occupancy (except for root). Each node contains d <= $\underline{m}$ <= 2d entries. The parameter d is called the *order* of the tree.
- Supports equality and range-searches efficiently.



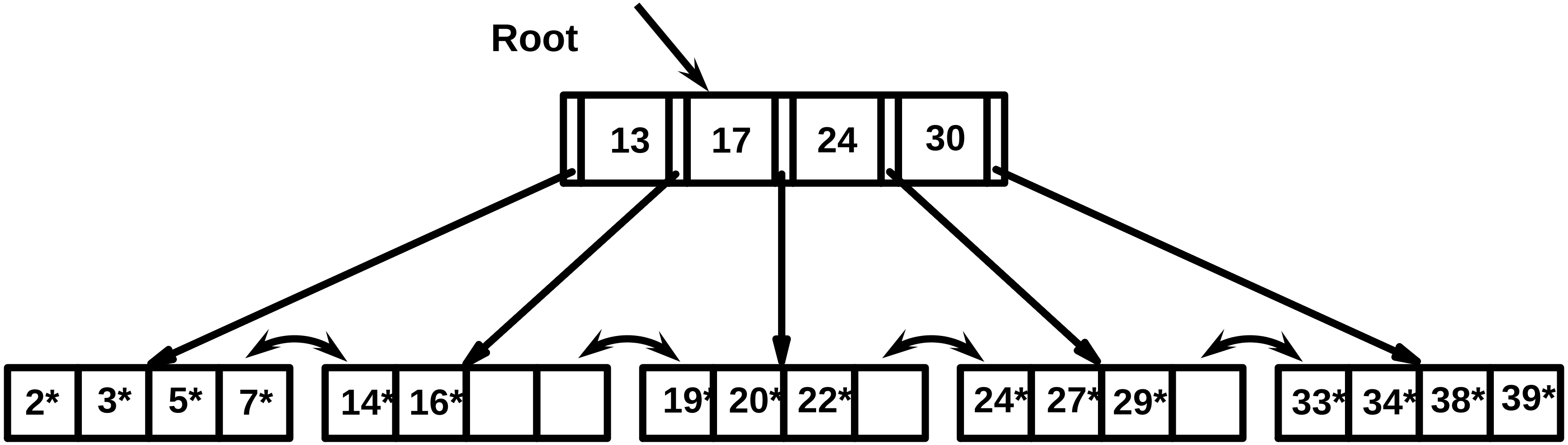
Index Entries

(Direct search)

Data Entries
("Sequence set")

Example B+ Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM).
- Search for 5*, 15*, all data entries >= 24* ...



Based on the search for 15, we know it is not in the tree!*

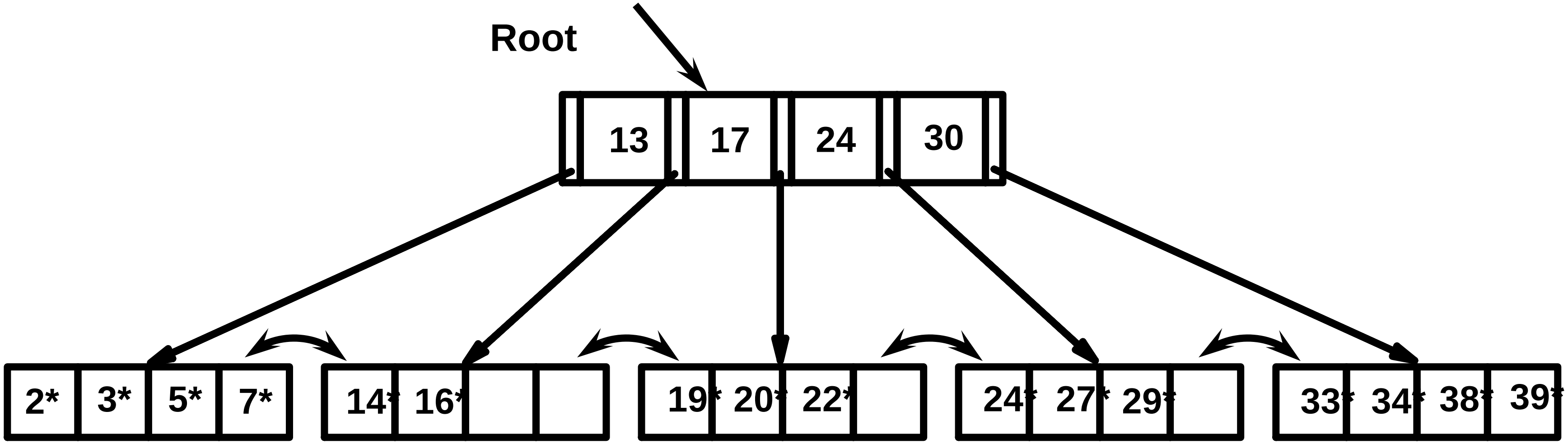
B+ Tree in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

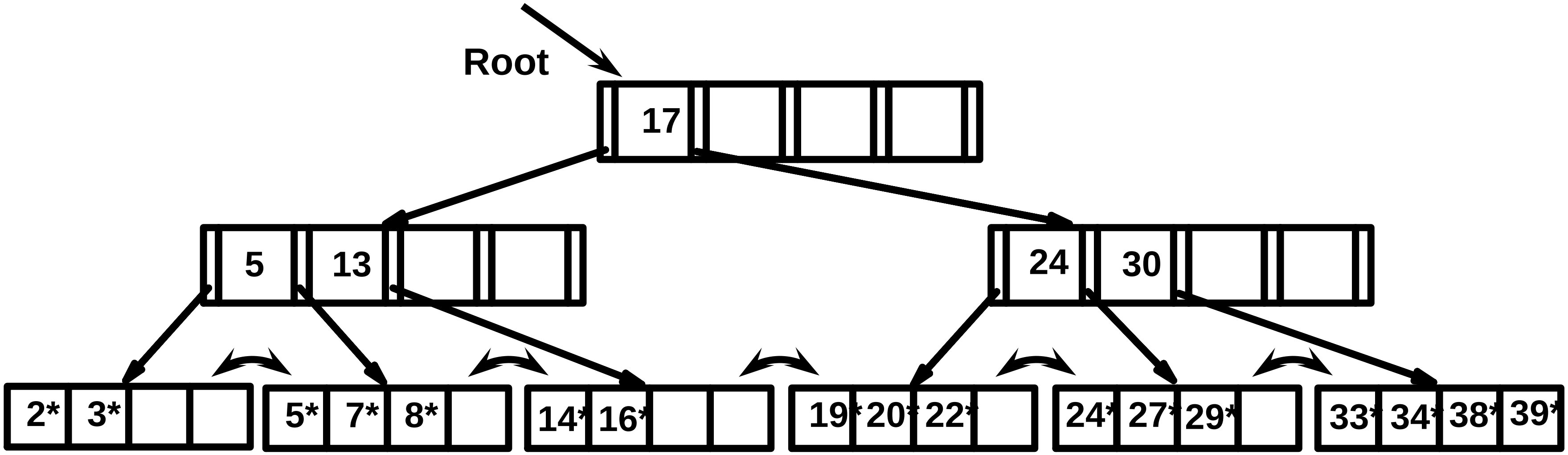
Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits “grow” tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Example B+ Tree - Inserting 8*



Example B+ Tree - Inserting 8*

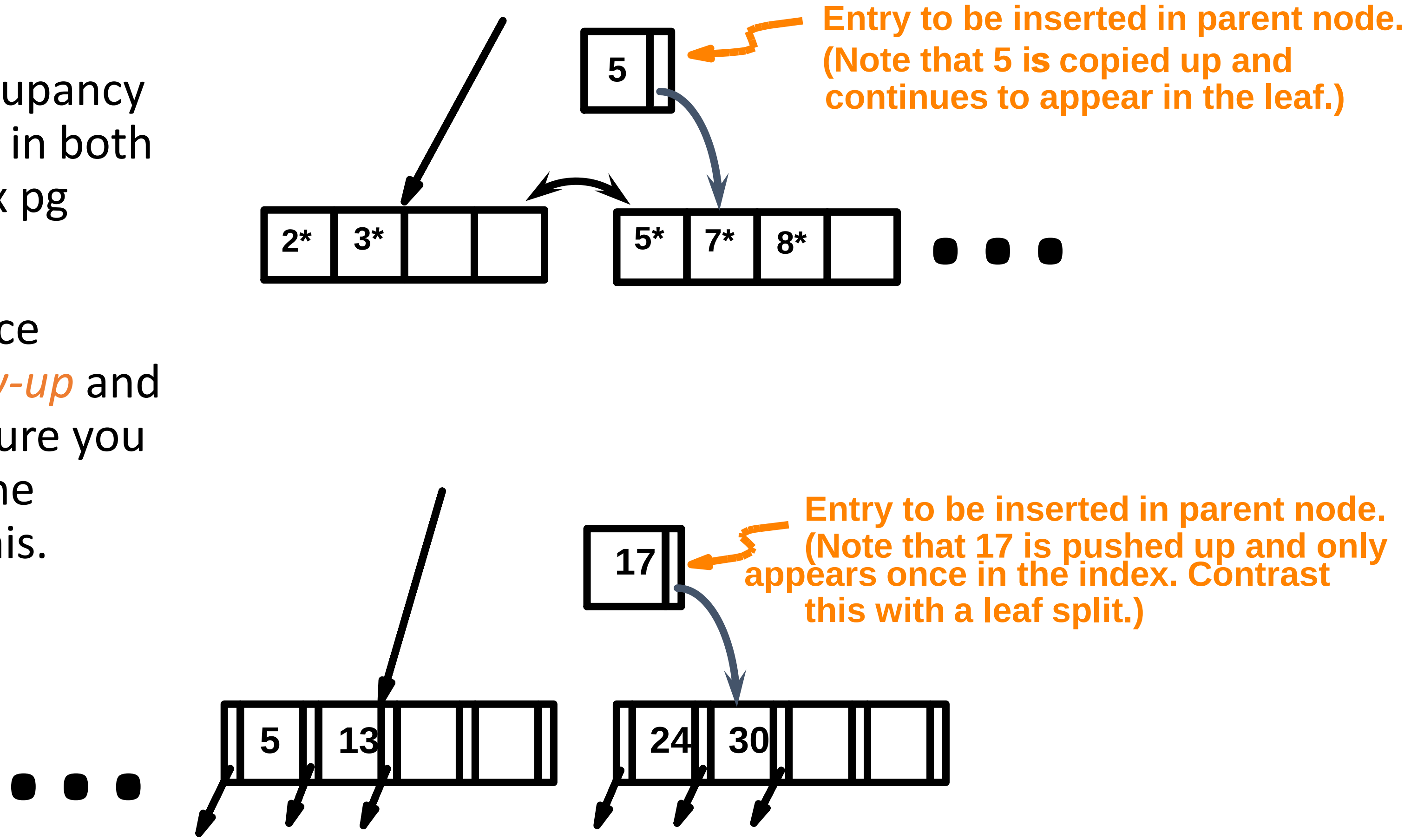


Notice that root was split, leading to increase in height.

In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

Inserting 8* into Example B+ Tree

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.



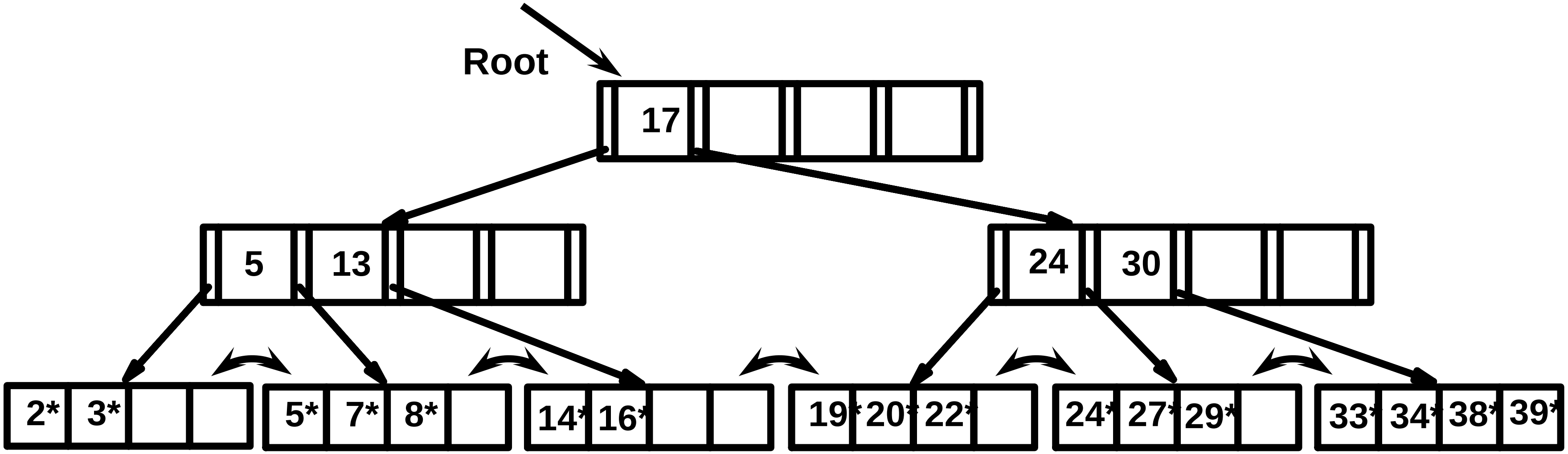
Entry to be inserted in parent node. (Note that 5 is copied up and continues to appear in the leaf.)

Entry to be inserted in parent node. (Note that 17 is pushed up and only appears once in the index. Contrast this with a leaf split.)

Deleting a Data Entry from a B+ Tree

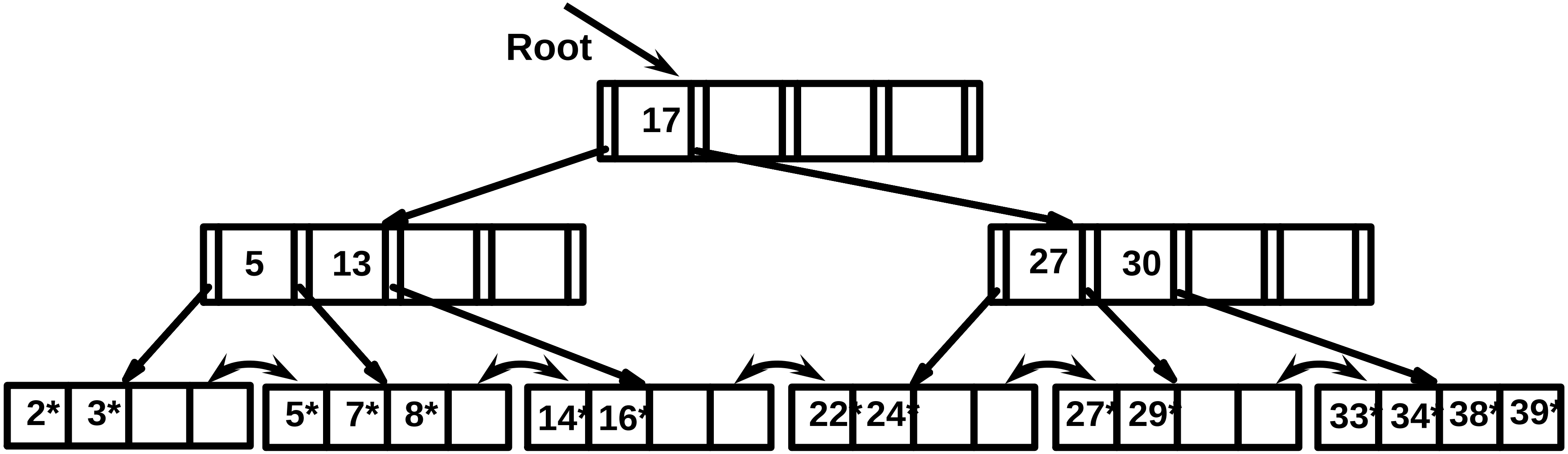
- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute, borrowing from *sibling (adjacent node with same parent as L)*.
    - If re-distribution fails, *merge* *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Example Tree (including 8*)
Delete 19* and 20* ...



- Deleting 19* is easy.

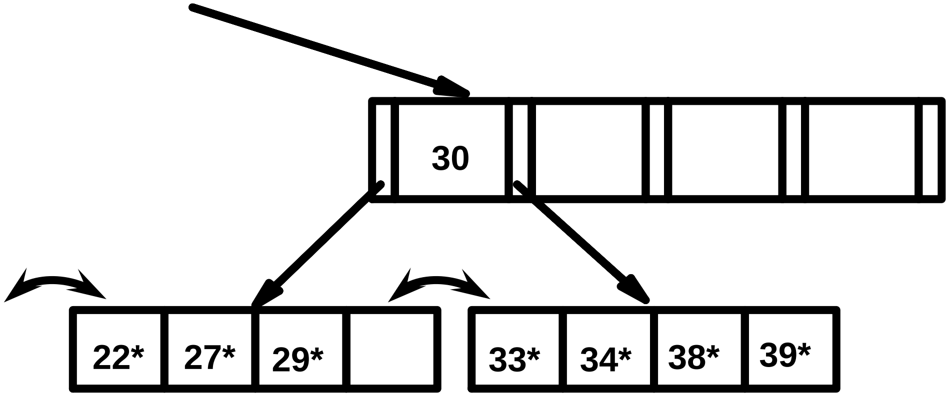
Example Tree (including 8*)
Delete 19* and 20* ...

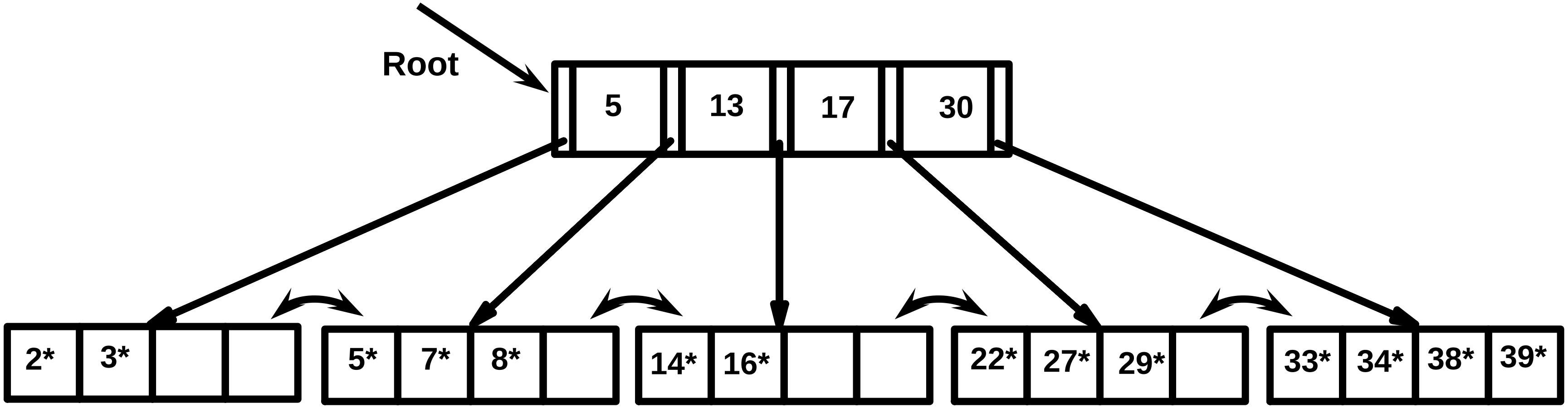


- Deleting 19* is easy.
- Deleting 20* is done with re-distribution. Notice how middle key is *copied up*.

... And Then Deleting 24*

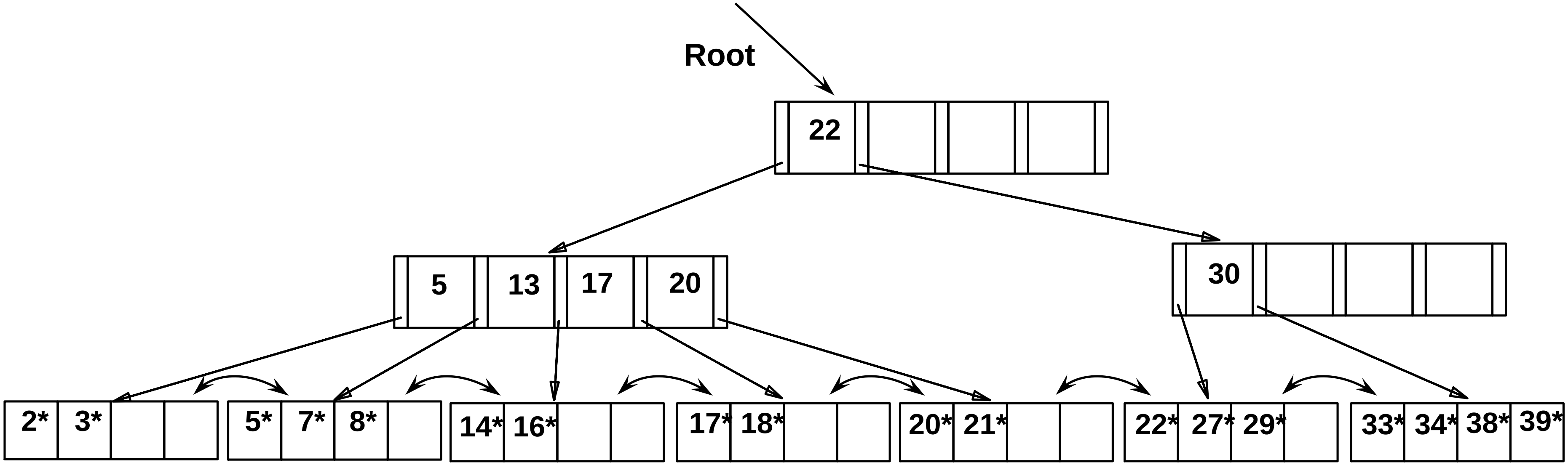
- Must merge.
- Observe *`toss'* of index entry (on right), and *`pull down'* of index entry (below).





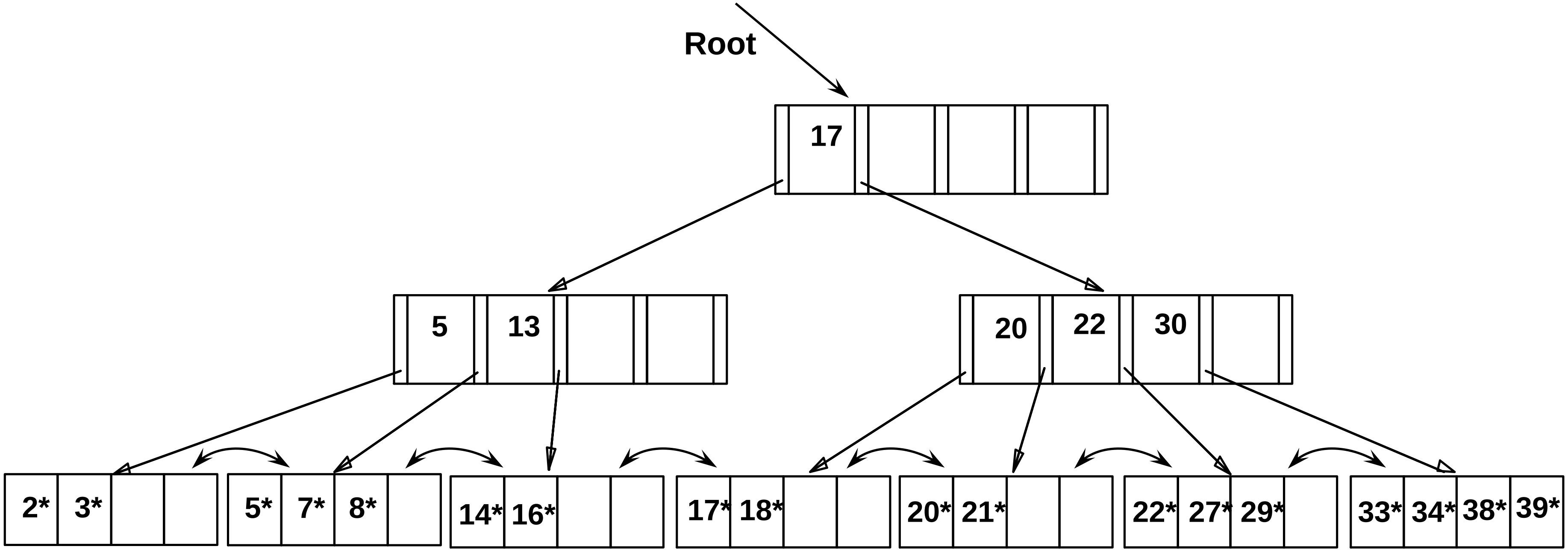
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



After Re-distribution

- Intuitively, entries are re-distributed by `*pushing through*’ the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we’ve re-distributed 17 as well for illustration.

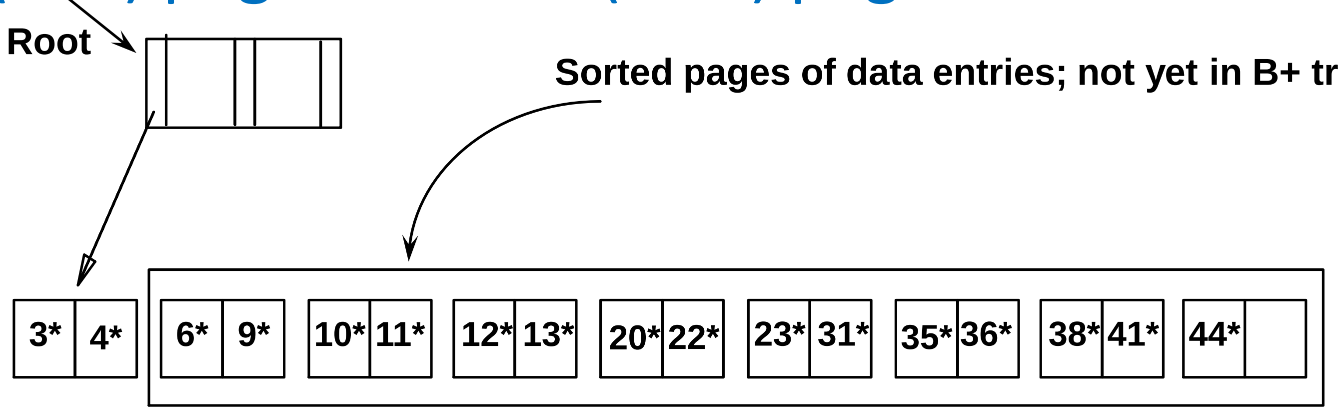


Prefix Key Compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

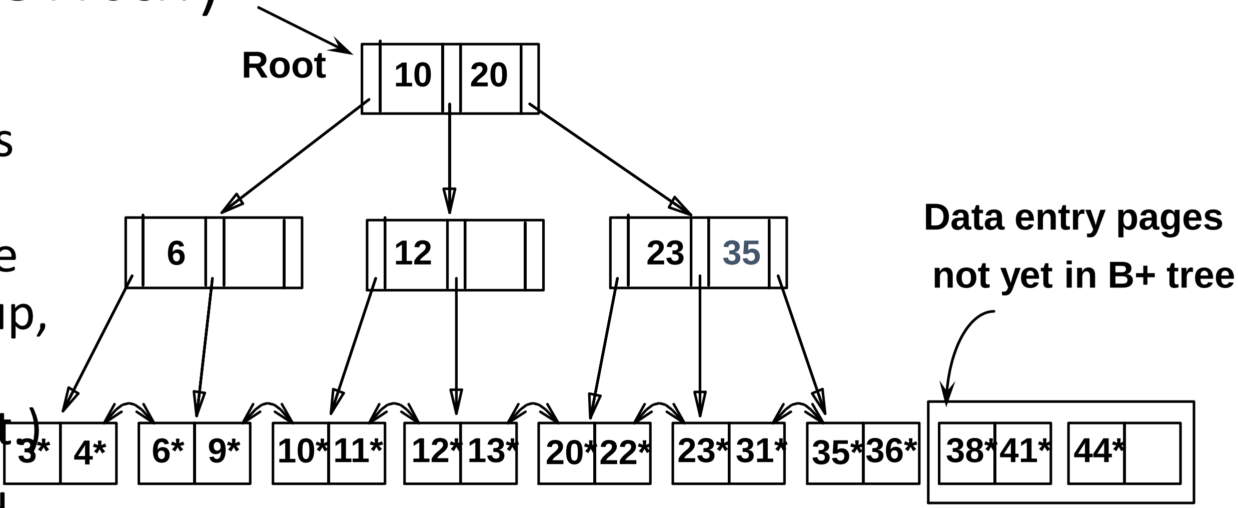
Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
  - Also leads to minimal leaf utilization --- why?
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.

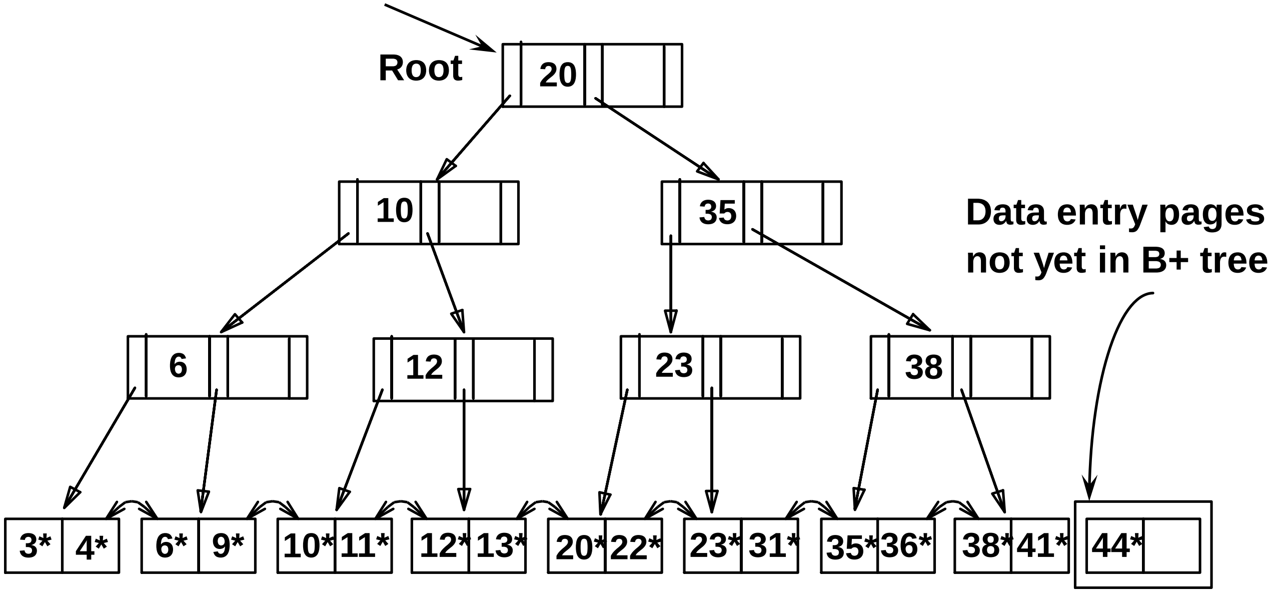


Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



Data entry pages not yet in B+ tree



Data entry pages not yet in B+ tree

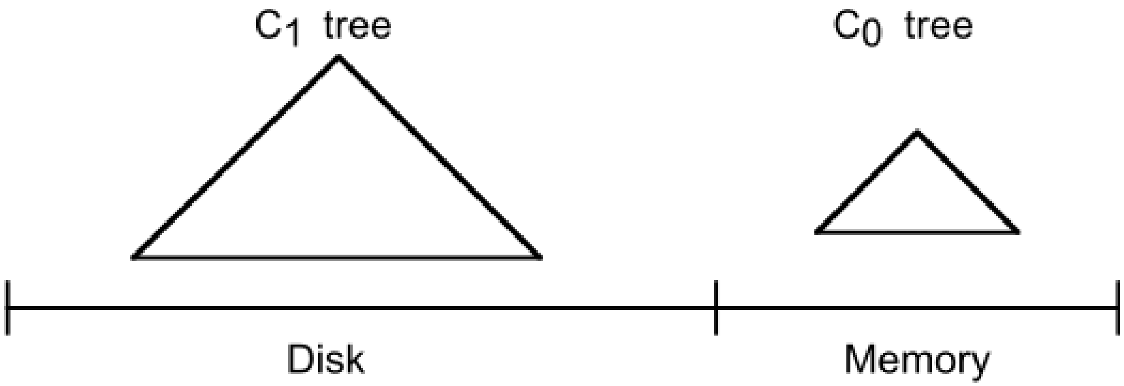
Summary of Bulk Loading

- Option 1: multiple inserts.
  - Slow.
  - Does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - Has advantages for concurrency control.
  - Fewer I/Os during build.
  - Leaves will be stored sequentially (and linked, of course).
  - Can control “fill factor” on pages.

3 Log Structured Merge (LSM) Tree

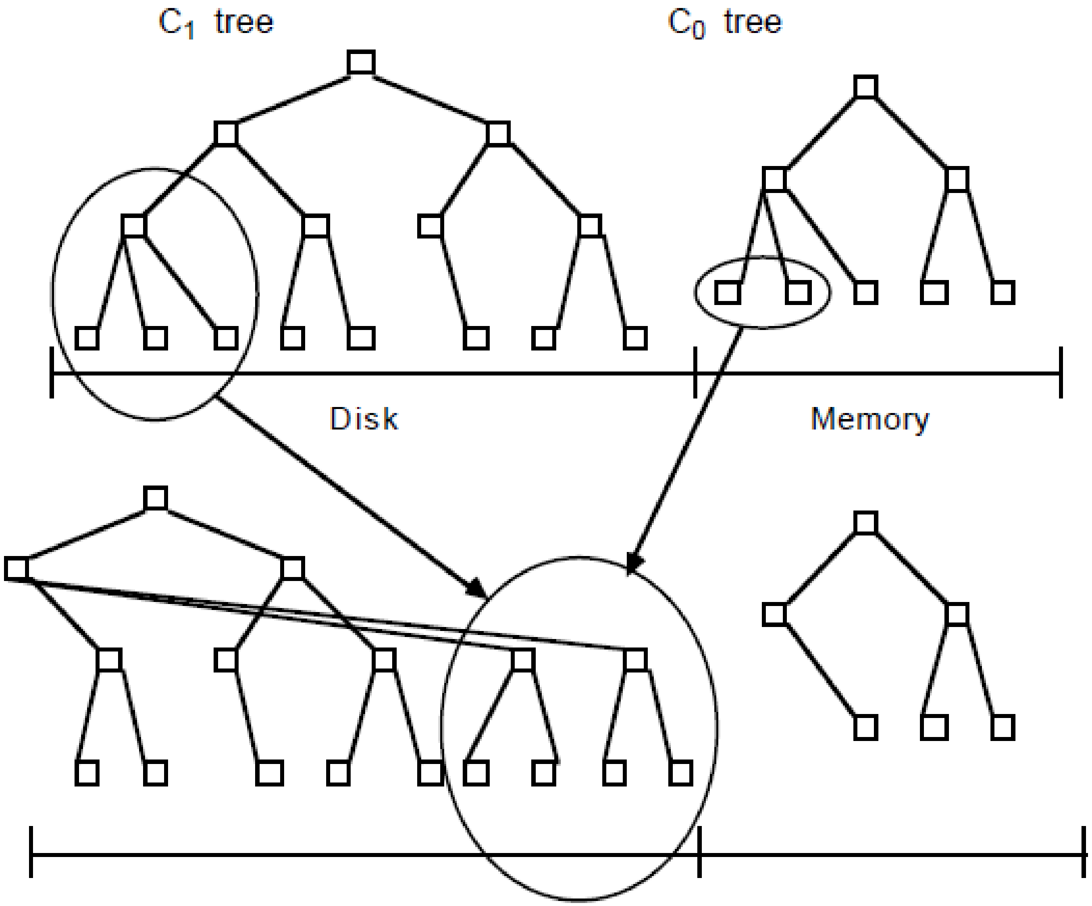
Structure of LSM Tree

- Two trees
  - C_0 tree: memory resident (smaller part)
  - C_1 tree: disk resident (whole part)



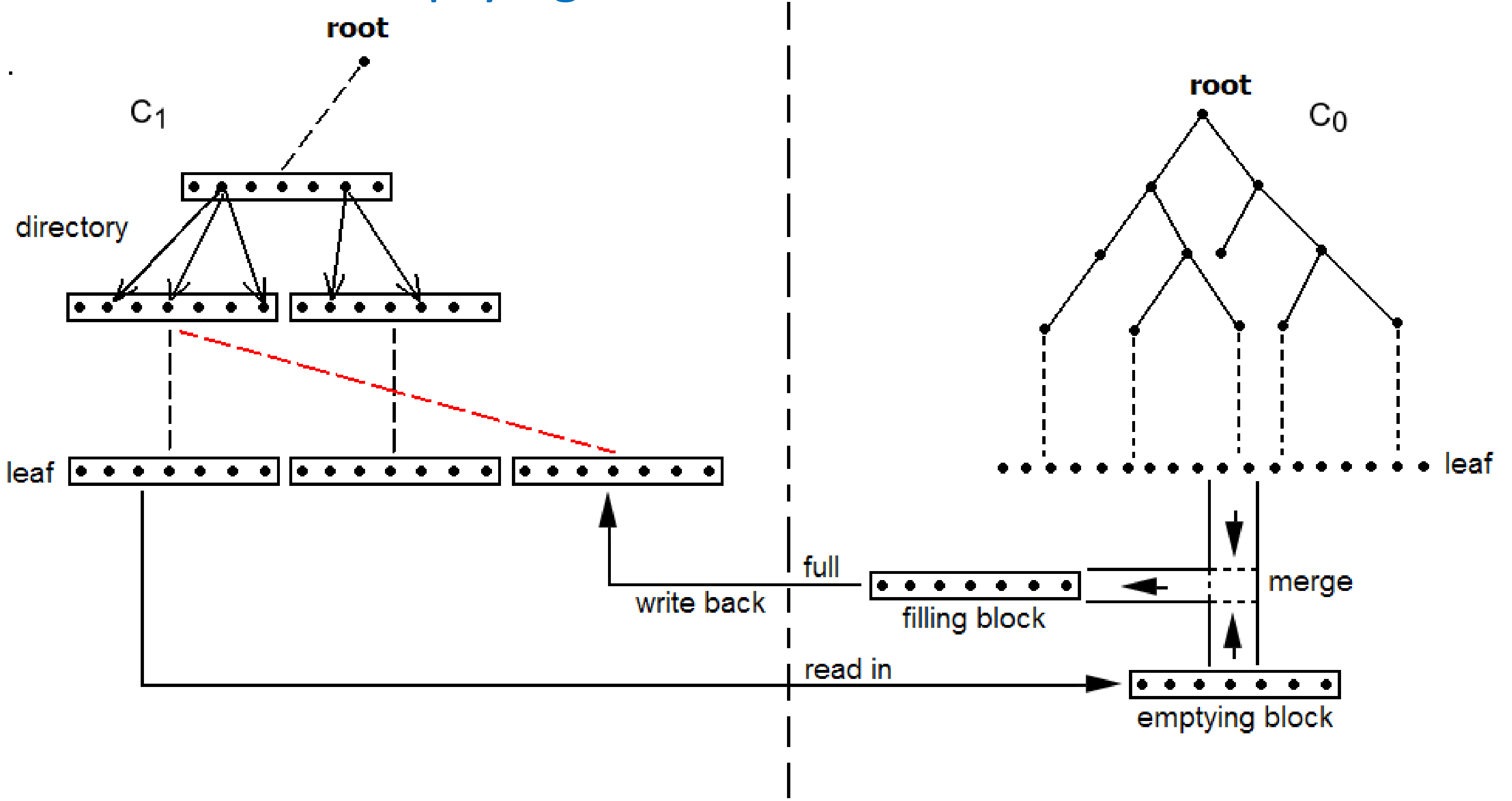
Rolling Merge (1)

- Merge new leaf nodes in C_0 tree and C_1 tree



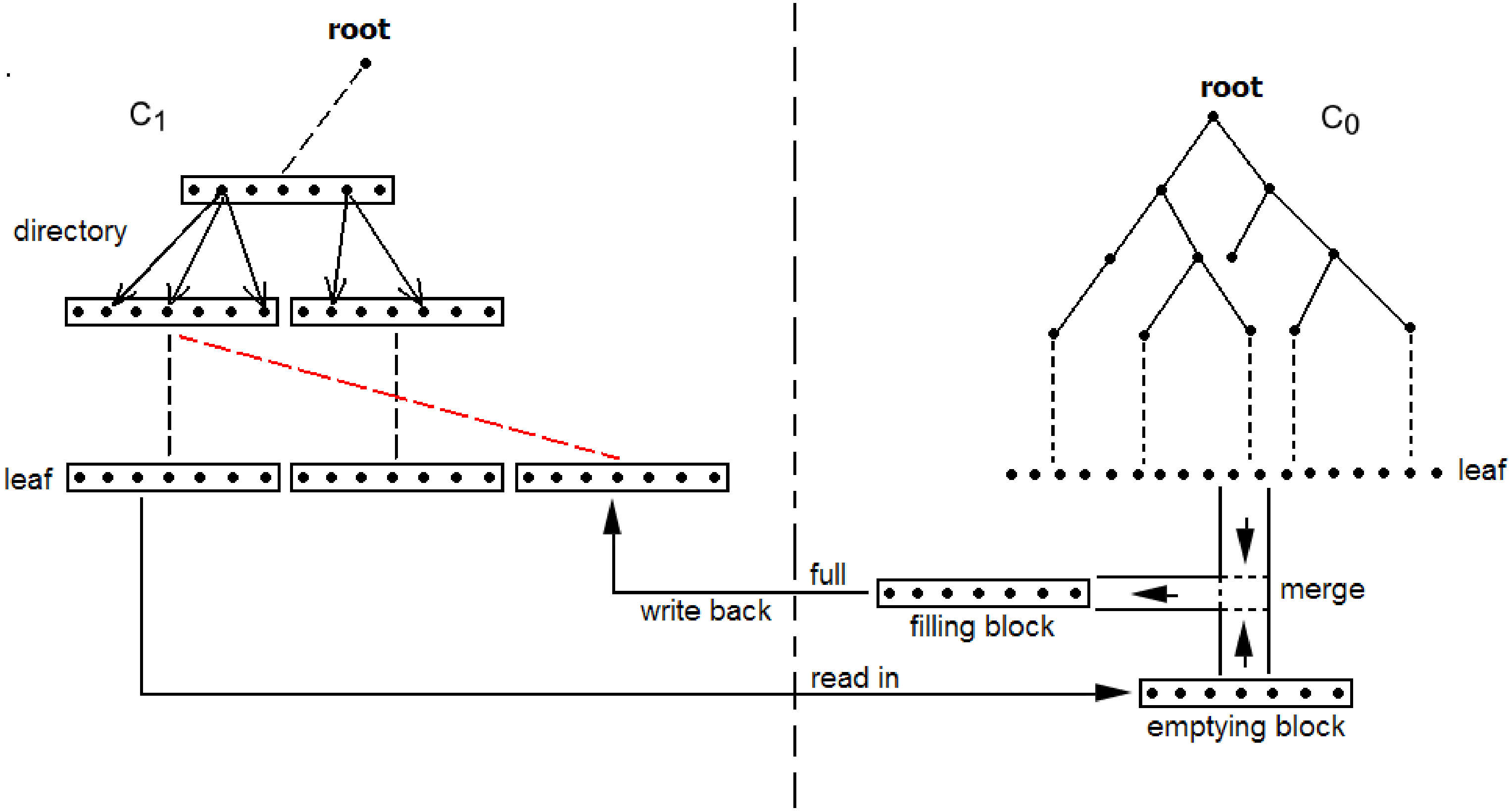
Rolling Merge (2)

- Step 1: read the new leaf nodes from C_1 tree, and store them as emptying block in memory
- Step 2: read the new leaf nodes from C_0 tree, and make merge sort with the emptying block



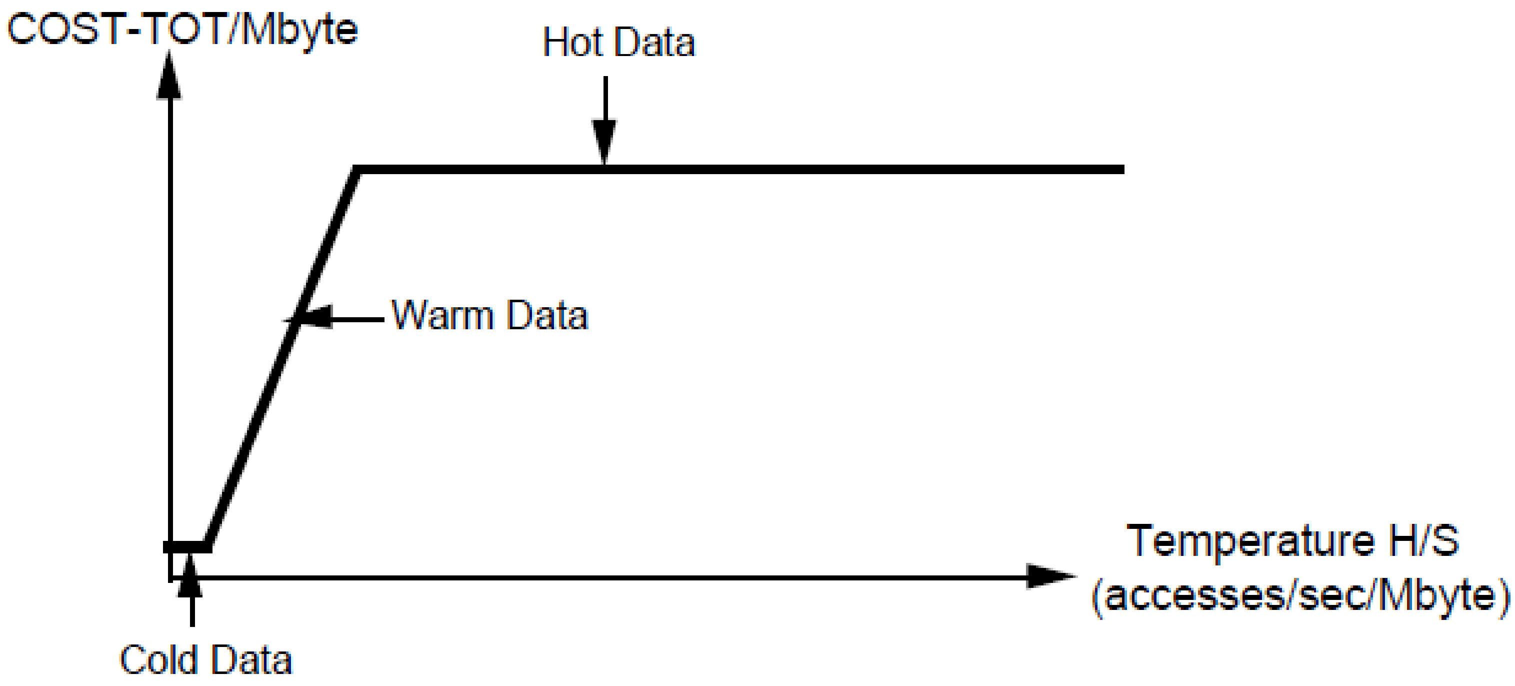
Rolling Merge (3)

- Step 3: write the merge results into filling block, and delete the new leaf nodes in $C_{0.}$
- Step 4: repeat step 2 and 3. When the filling block is full, write the filling block into C_1 tree, and delete the corresponding leaf nodes.
- Step 5: after all new leaf nodes in C_0 and C_1 are merged, finish the rolling merge process.



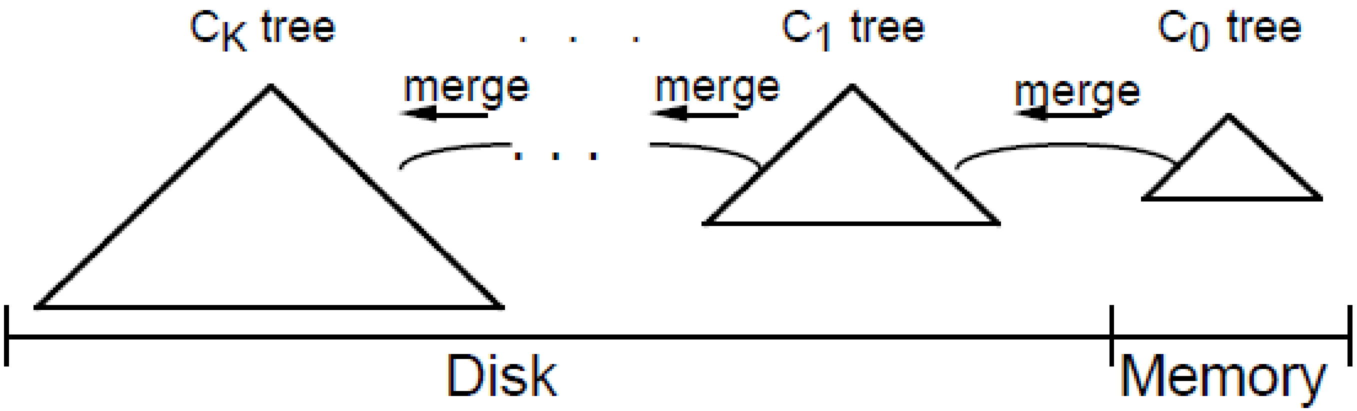
Data temperature

- Data Type
  - Hot/Warm/Cold Data → different trees



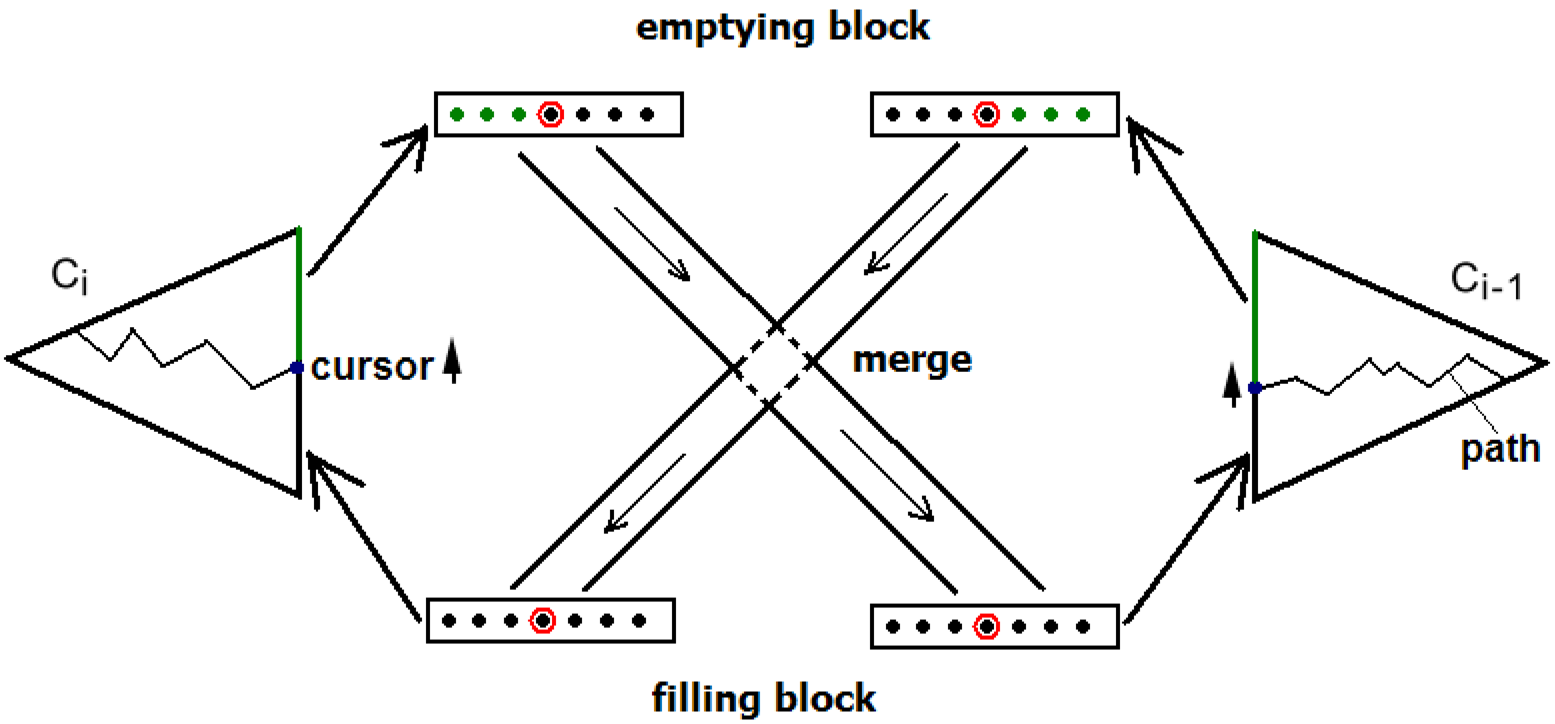
A LSM tree with multiple components

- Data Type
  - Hottest data → C_0 tree
  - Hotter data → C_1 tree
  - ……
  - Coldest data → C_K tree



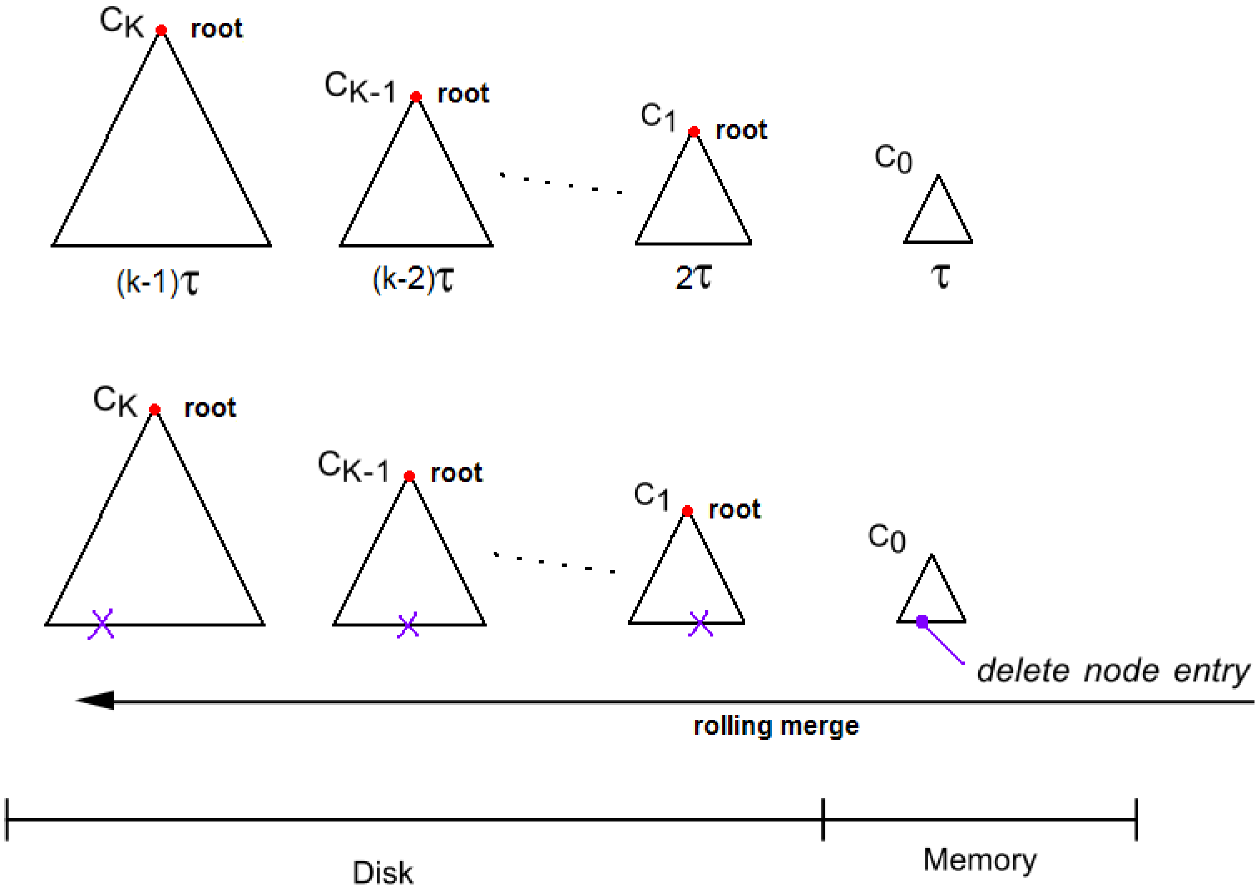
Rolling Merge among Disks

- Two emptying blocks and filling blocks
- New leaf nodes should be locked (write lock)



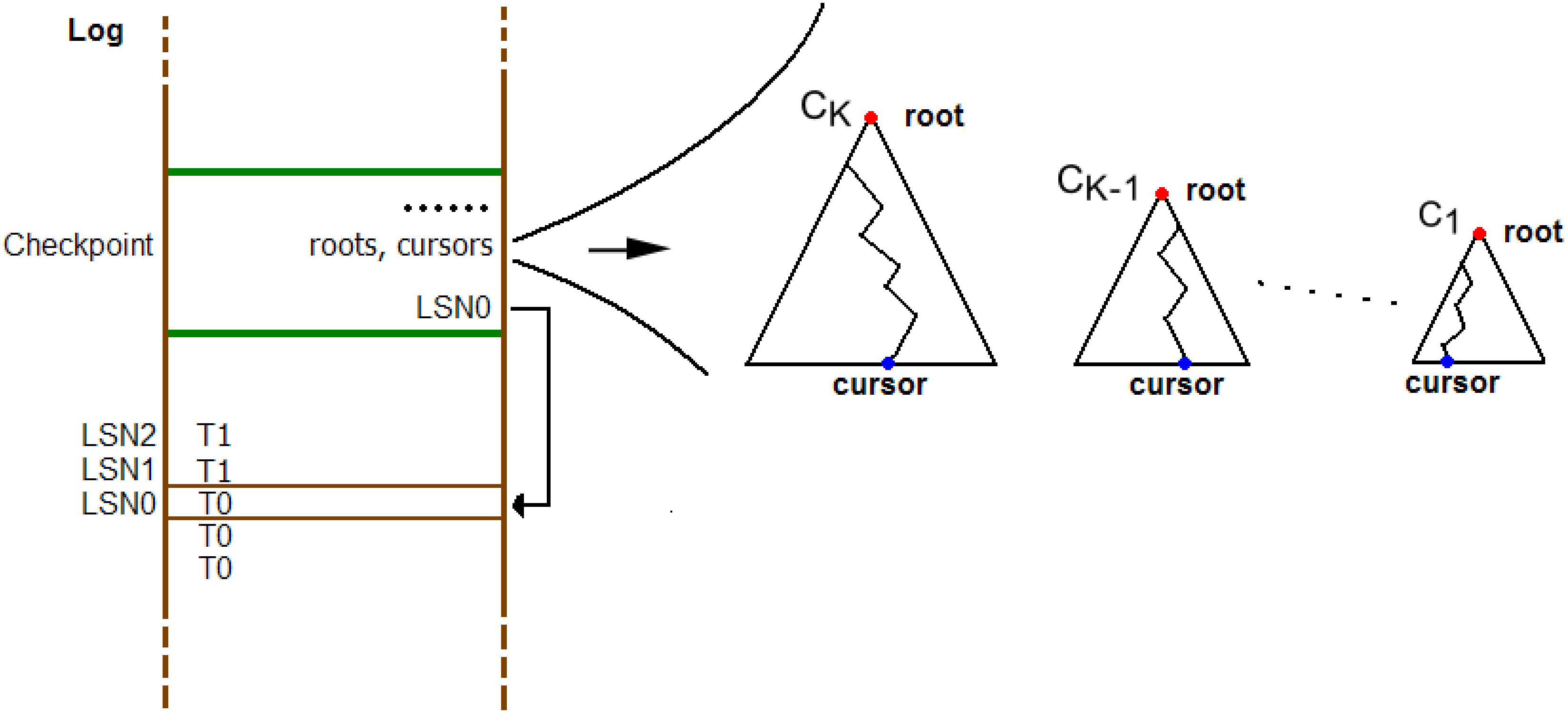
Search and deletion (based on temporal locality)

- Lastest T (0- T) accesses are in C_0 tree
- T - 2T accesses are in C_1 tree
- ……



Checkpointing

- Log Sequence Number (LSN0) of last insertion at Time T_0
- Root addresses
- Merge cursor for each component
- Allocation information



4

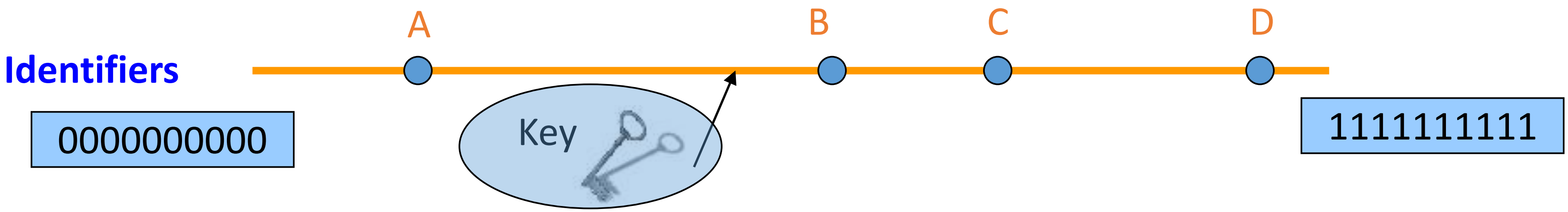
Distributed Hash & DHT

Definition of a DHT

- Hash table → supports two operations
  - `insert(key, value)`
  - `value = lookup(key)`
- Distributed
  - Map hash-buckets to nodes
- Requirements
  - Uniform distribution of buckets
  - Cost of `insert` and `lookup` should *scale* well
  - Amount of local state (routing table size) should *scale* well

Fundamental Design Idea - I

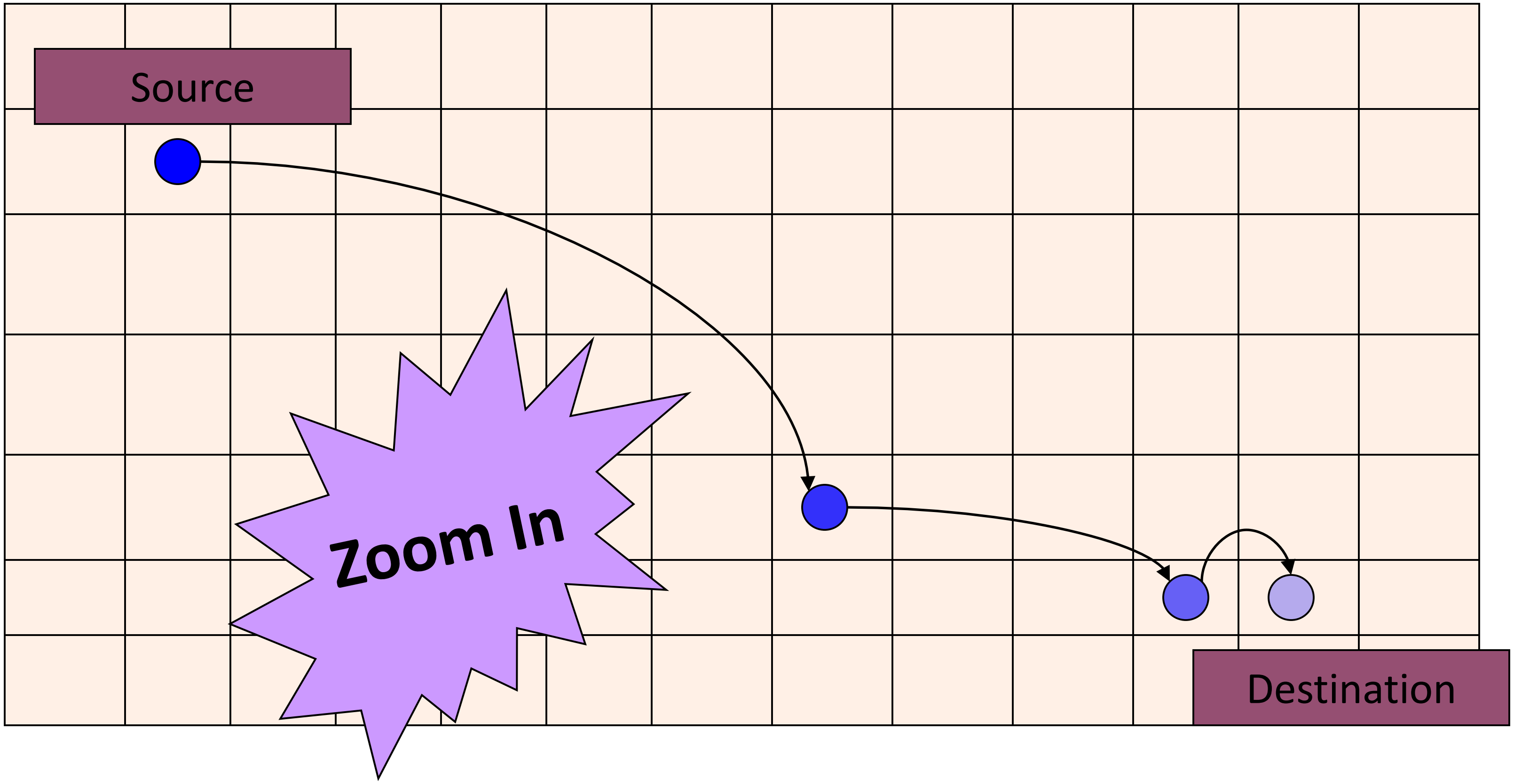
- Consistent Hashing
  - Map keys *and* nodes to an *identifier* space; implicit assignment of responsibility



- Mapping performed using hash functions (e.g., SHA-1)
  - Spread nodes and keys *uniformly* throughout

Fundamental Design Idea - II

- Prefix / Hypercube routing



But, there are so many of them!

- Scalability trade-offs
  - Routing table size at each node vs.
  - Cost of lookup and insert operations
- Simplicity
  - Routing operations
  - Join-leave mechanisms
- Robustness
- DHT Designs
  - Plaxton Trees, Pastry/Tapestry
  - Chord
  - Overview: CAN, Symphony, Koorde, Viceroy, etc.
  - SkipNet

Plaxton Trees Algorithm (1)

1. Assign labels to objects and nodes
   - using randomizing hash functions

9	A	E	4

Object

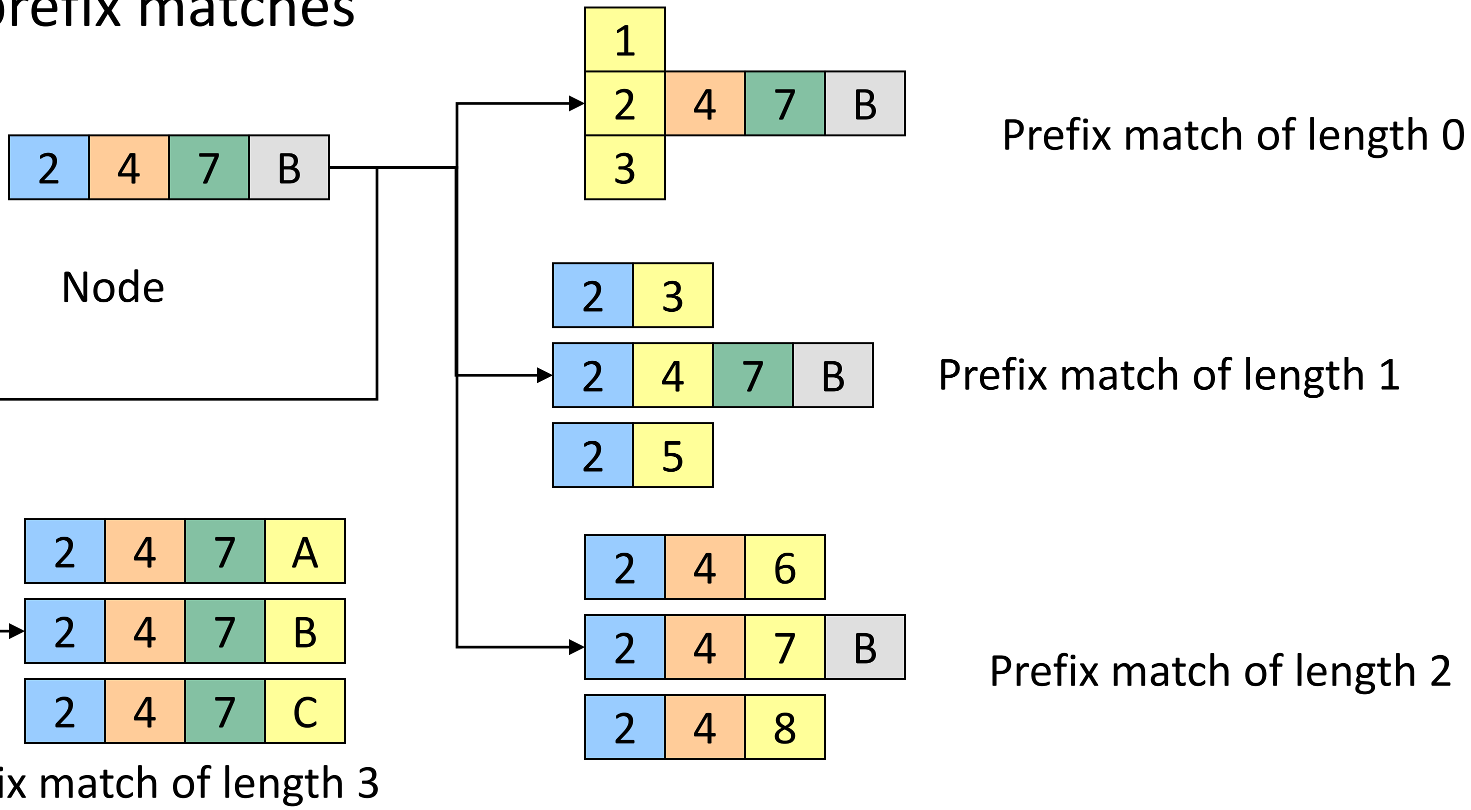
2	4	7	B

Node

Each label is of $\log_2^b n$ digits

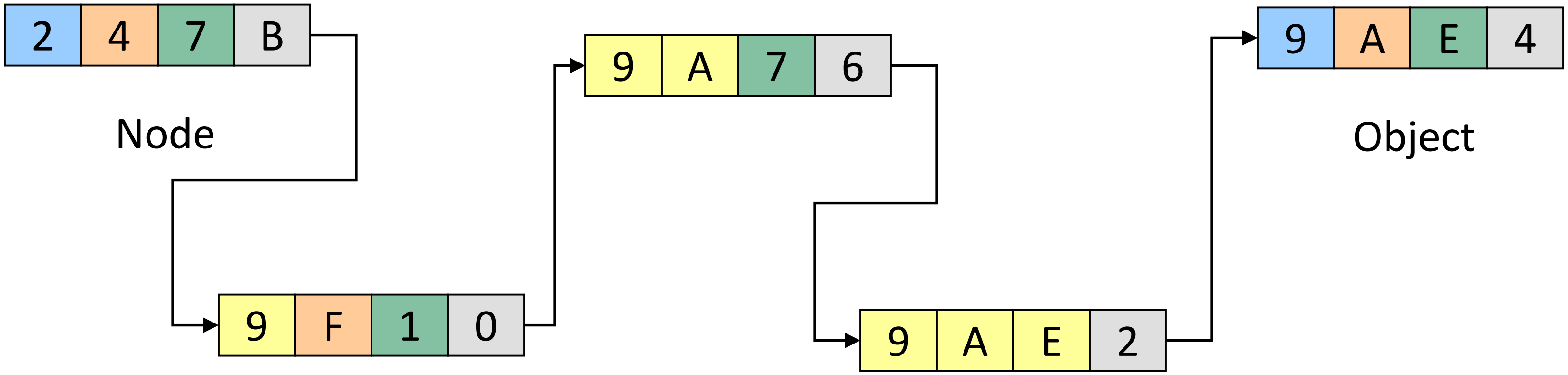
Plaxton Trees Algorithm (2)

2. Each node knows about other nodes with varying prefix matches



Plaxton Trees Algorithm (3)
Object Insertion and Lookup

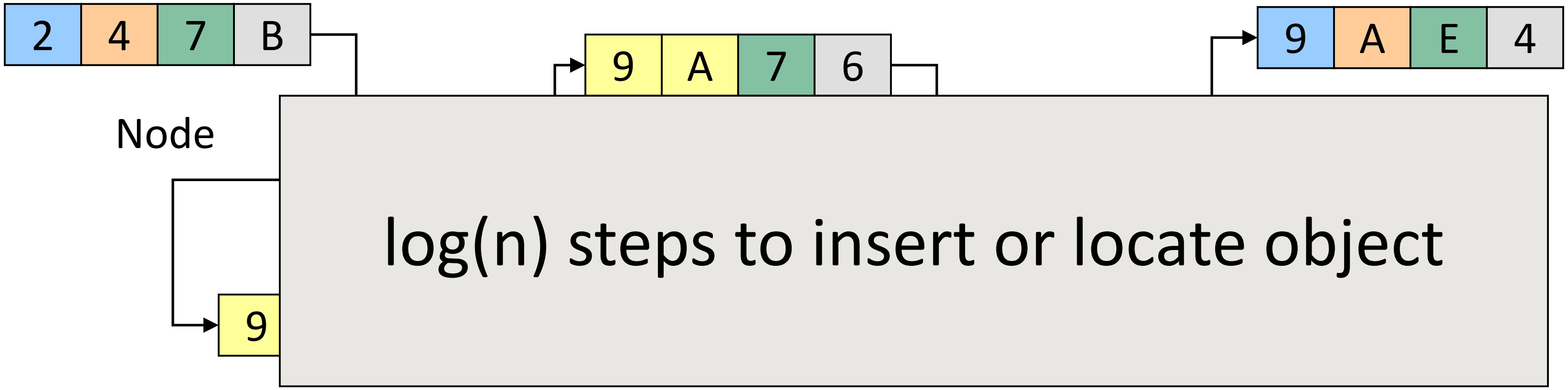
Given an object, route successively towards nodes with greater prefix matches



Store the object at each of these locations

Plaxton Trees Algorithm (4)
Object Insertion and Lookup

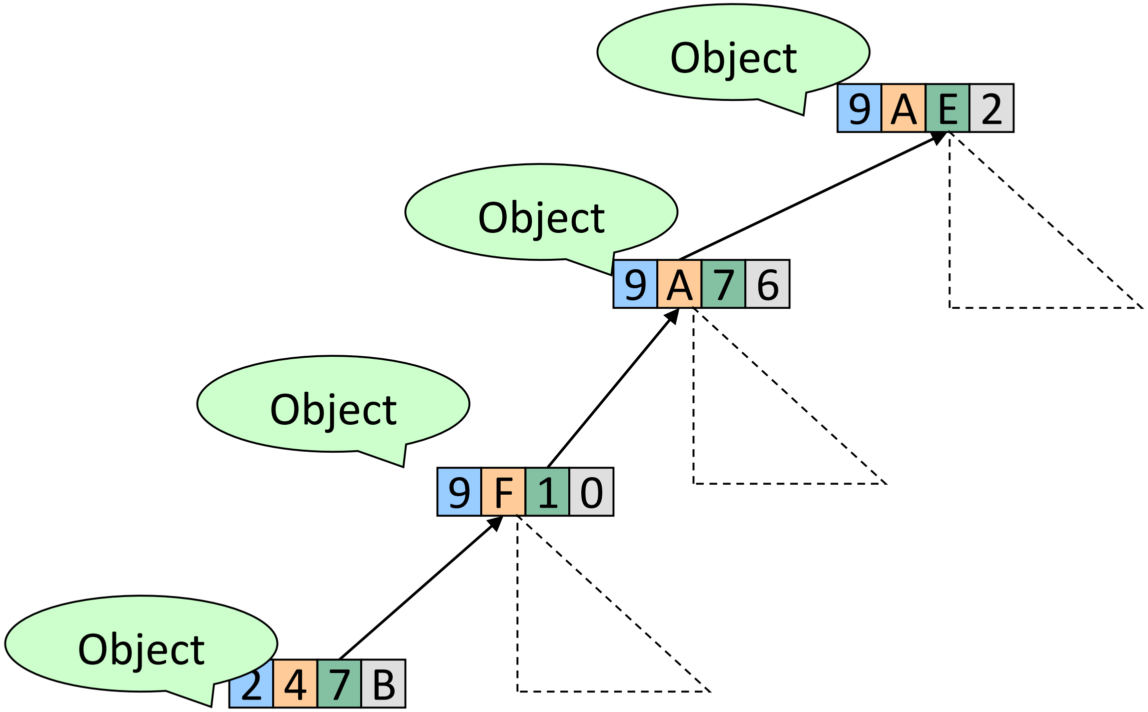
Given an object, route successively towards nodes with greater prefix matches



log(n) steps to insert or locate object

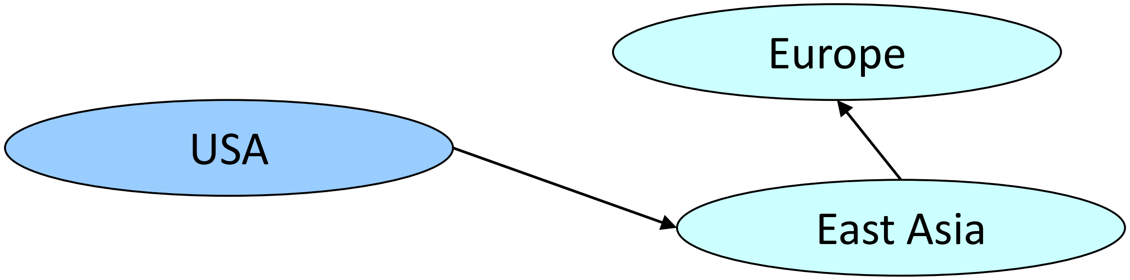
Store the object at each of these locations

Plaxton Trees Algorithm (5)
Why is it a tree?



Plaxton Trees Algorithm (6)
Network Proximity

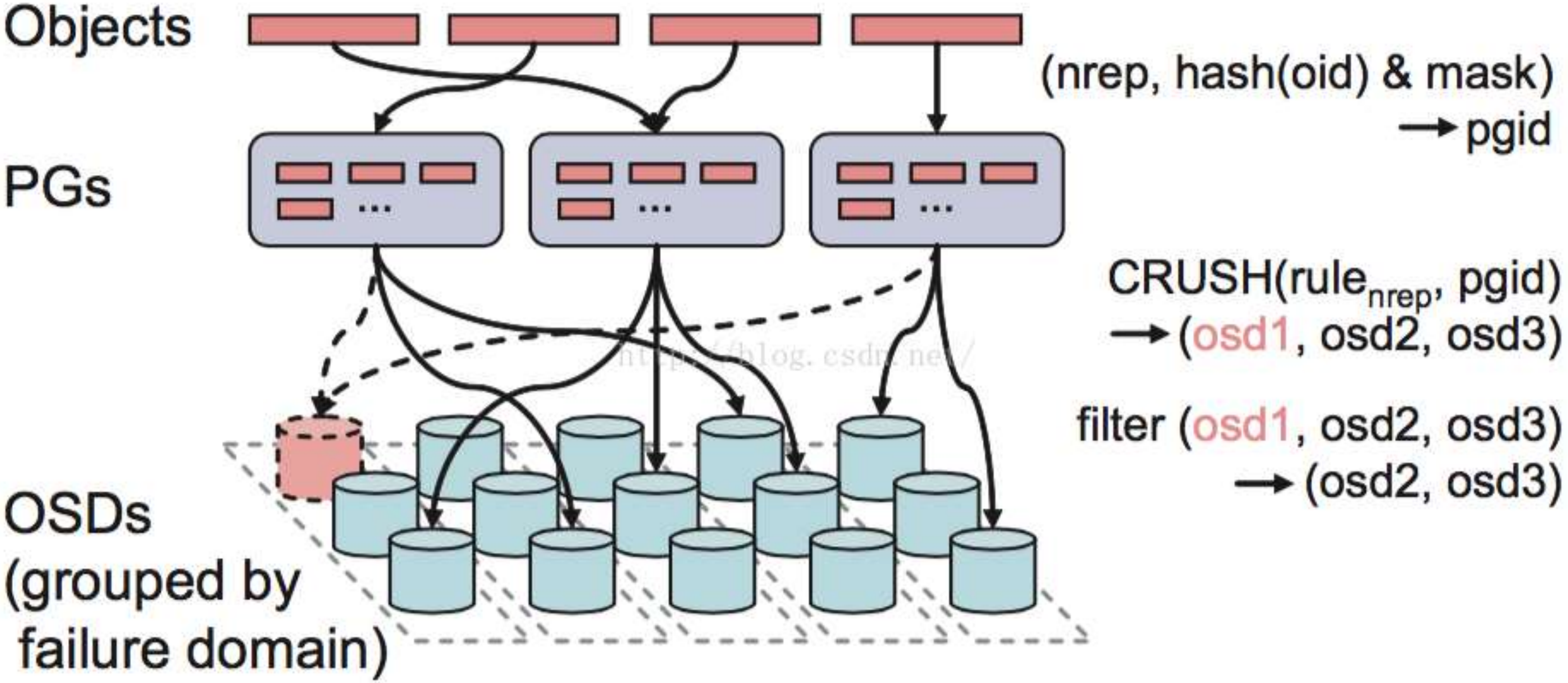
- Overlay tree hops could be totally unrelated to the underlying network hops



- Plaxton trees guarantee constant factor approximation!
  - Only when the topology is *uniform* in some sense

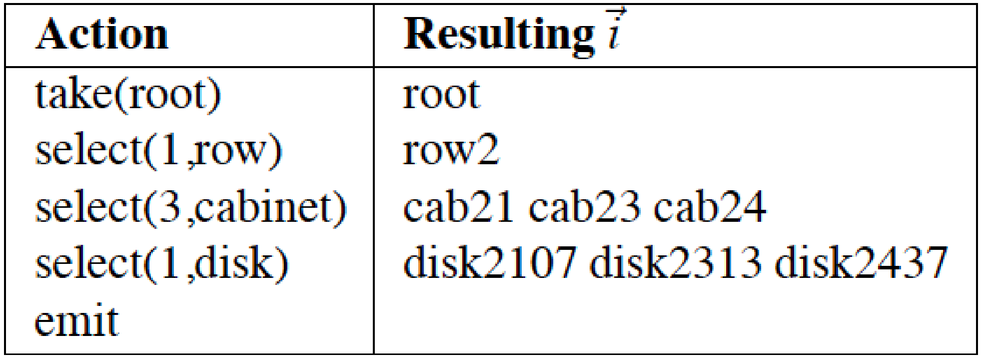
Ceph Controlled Replication Under Scalable Hashing (CRUSH) (1)

- CRUSH algorithm: pgid → OSD ID?
- Devices: leaf nodes (weighted)
- Buckets: non-leaf nodes (weighted, contain any number of devices/buckets)

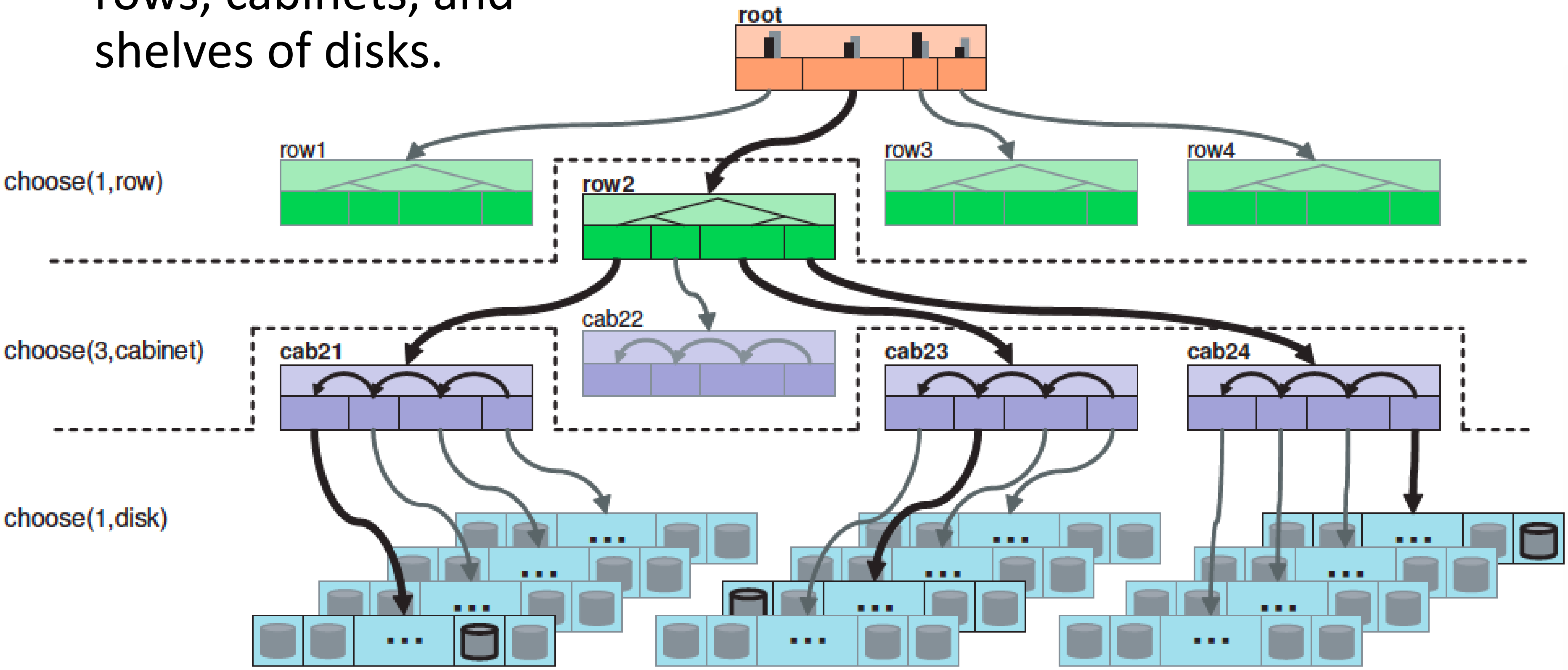


CRUSH (2)

- A partial view of a four-level cluster map hierarchy consisting of rows, cabinets, and shelves of disks.

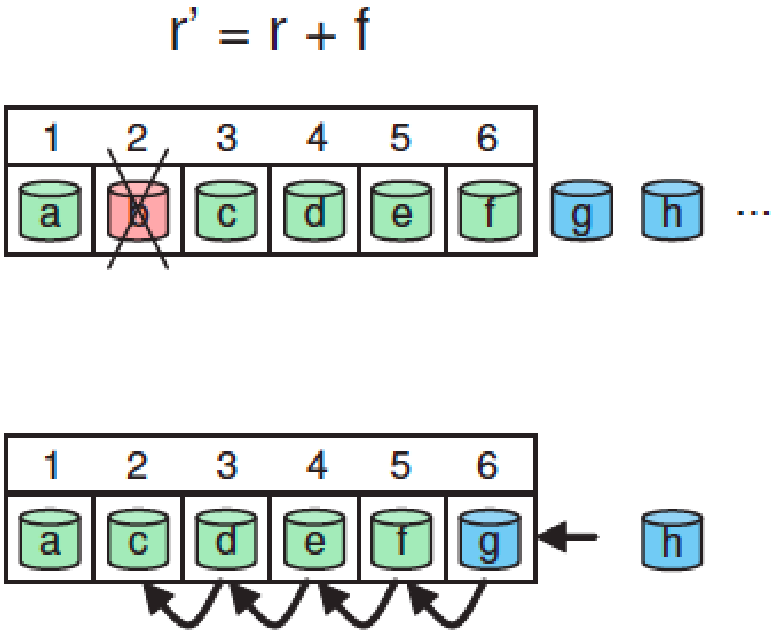


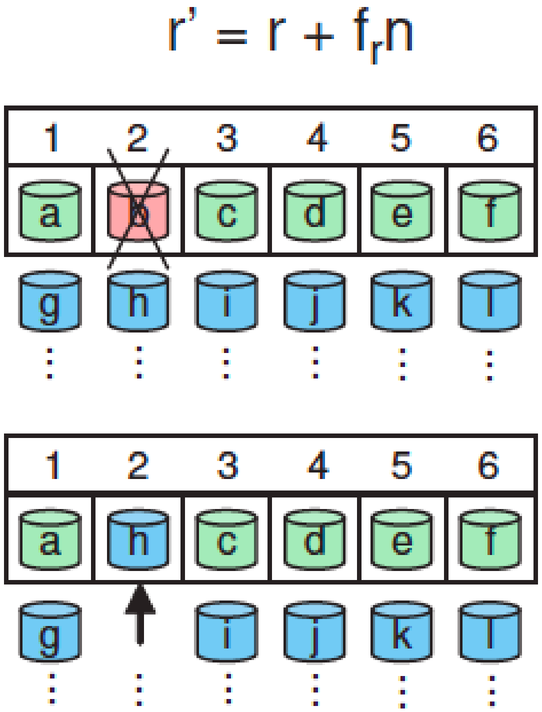
Action	**Resulting $\vec{i}$**
take(root)	root
select(1,row)	row2
select(3,cabinet)	cab21 cab23 cab24
select(1,disk)	disk2107 disk2313 disk2437
emit	



CRUSH (3)

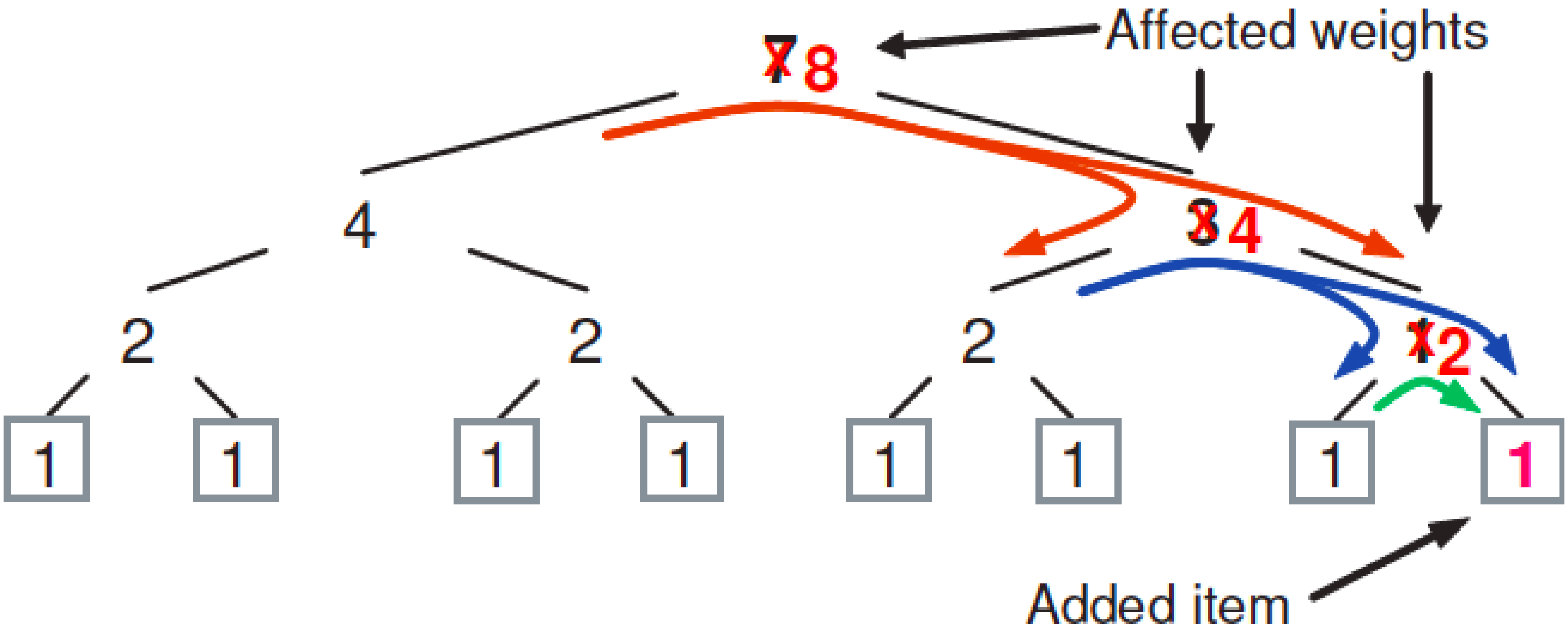
- Reselection behavior of select(6,disk) when device r = 2 (b) is rejected, where the boxes contain the CRUSH output R of n = 6 devices numbered by rank. The left shows the “first n” approach in which device ranks of existing devices (c,d,e,f) may shift. On the right, each rank has a probabilistically independent sequence of potential targets; here $f_r = 1$, and $r' = r + f_r n = 8$ (device h).





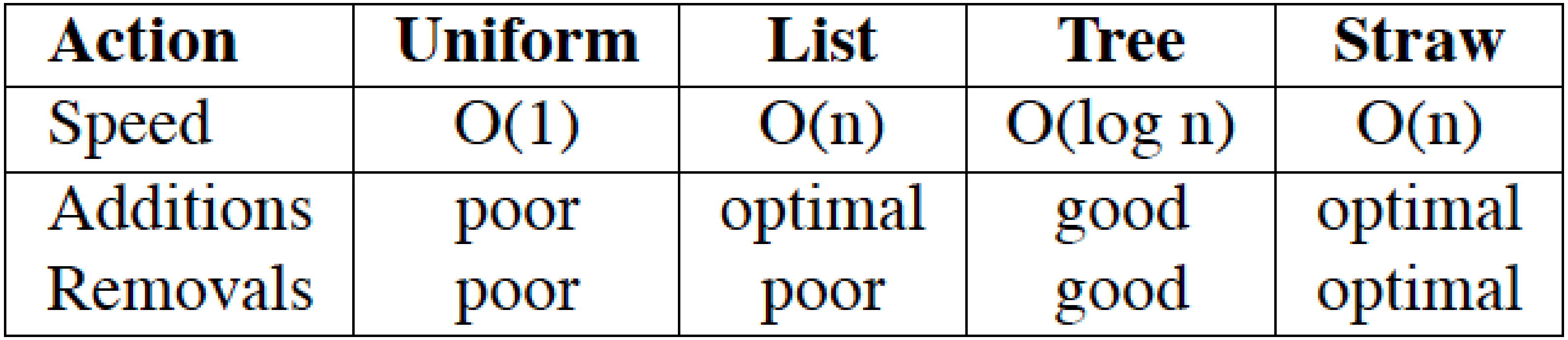
CRUSH (4)

- Data movement in a binary hierarchy due to a node addition and the subsequent weight changes.



CRUSH (5)

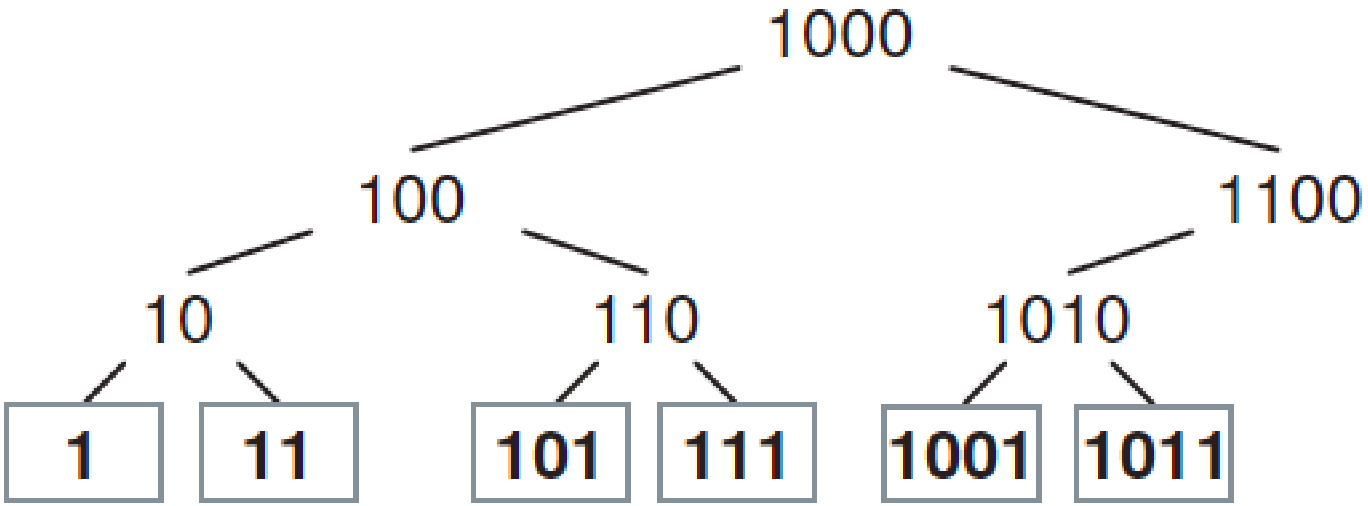
- Four types of Buckets
  - Uniform buckets
  - List buckets
  - Tree buckets
  - Straw buckets
- Summary of mapping speed and data reorganization efficiency of different bucket types when items are added to or removed from a bucket.



Action	Uniform	List	Tree	Straw
Speed	O(1)	O(n)	O(log n)	O(n)
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

CRUSH (6)

- Node labeling strategy used for the binary tree comprising each tree bucket



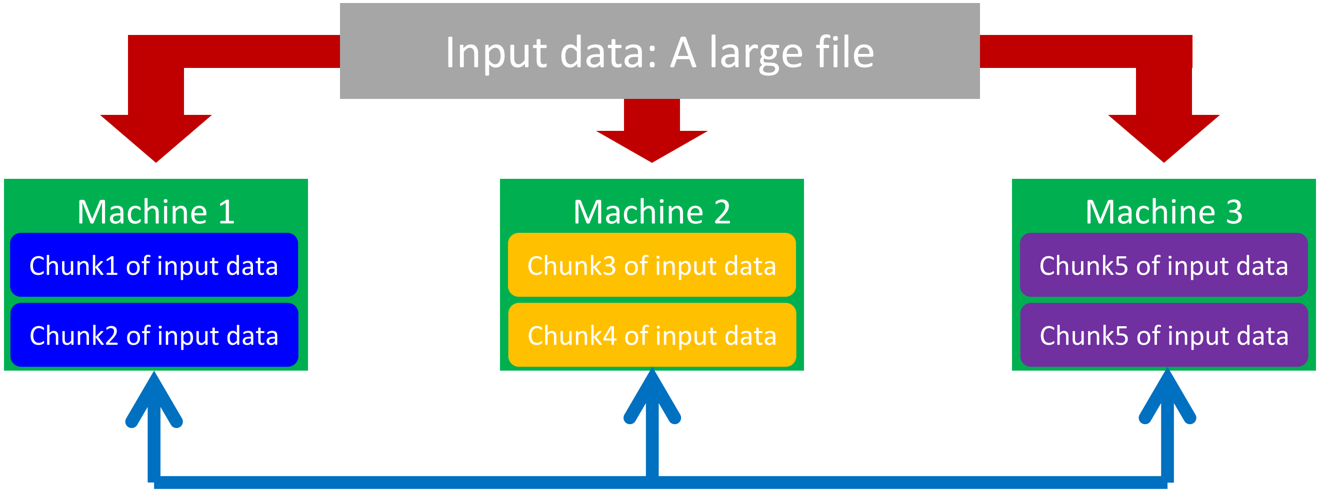
5 Motivation of NoSQL Databases

Big Data →Scaling Traditional Databases

- Traditional RDBMSs can be either scaled:
  - Vertically (or Scale Up)
    - Can be achieved by hardware upgrades (e.g., faster CPU, more memory, or larger disk)
    - Limited by the amount of CPU, RAM and disk that can be configured on a single machine
  - Horizontally (or Scale Out)
    - Can be achieved by adding more machines
    - Requires database *sharding* and probably *replication*
    - Limited by the Read-to-Write ratio and communication overhead

Big Data →Improving the Performance of Traditional Databases

- Data is typically *striped* to allow for concurrent/parallel accesses



Input data: A large file

Machine 1

Chunk1 of input data

Chunk2 of input data

Machine 2

Chunk3 of input data

Chunk4 of input data

Machine 3

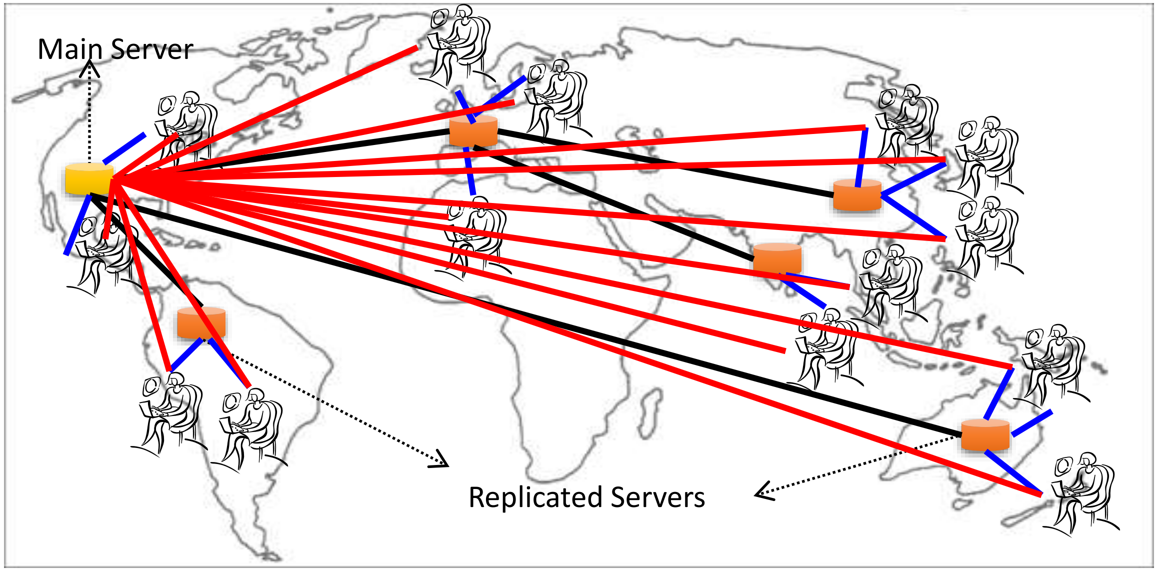
Chunk5 of input data

Chunk5 of input data

E.g., Chunks 1, 3 and 5 can be accessed in parallel

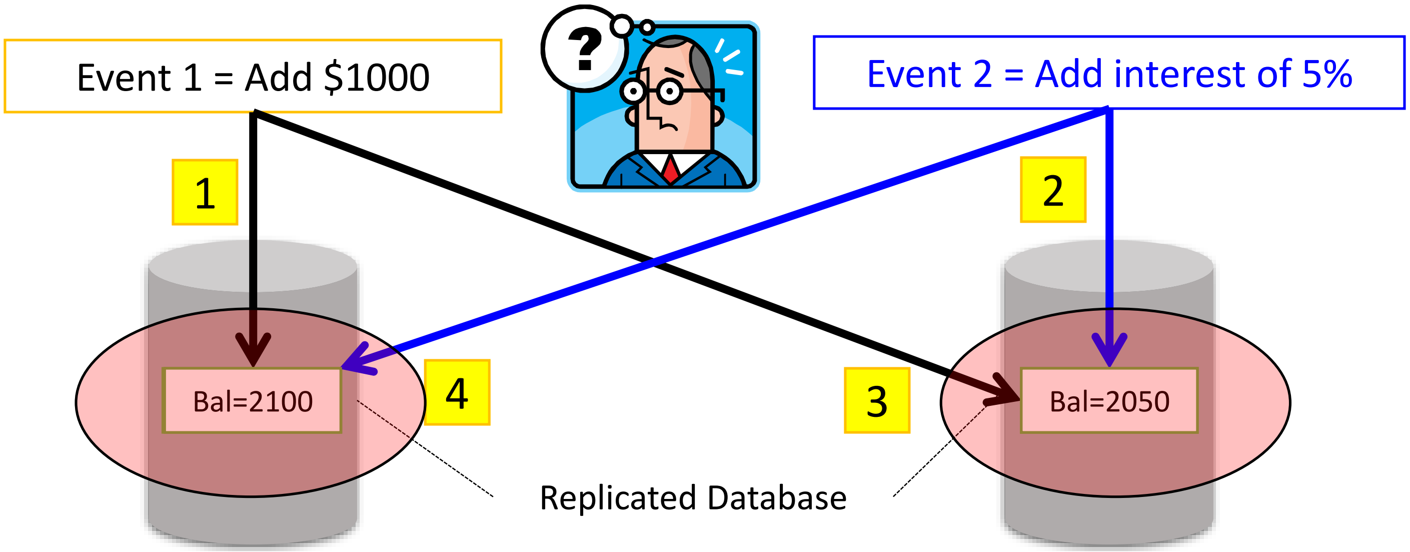
Why Replicating Data?

- Replicating data across servers helps in:
  - Avoiding performance bottlenecks
  - Avoiding single point of failures
  - And, hence, enhancing scalability and availability



But, Consistency Becomes a Challenge

- An example:
    - In an e-commerce application, the bank database has been replicated across two servers
    - Maintaining consistency of replicated data is a challenge



6 Introduction to NoSQL Databases

What’s NoSQL

- Stands for Not Only SQL
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the CAP/ACID properties

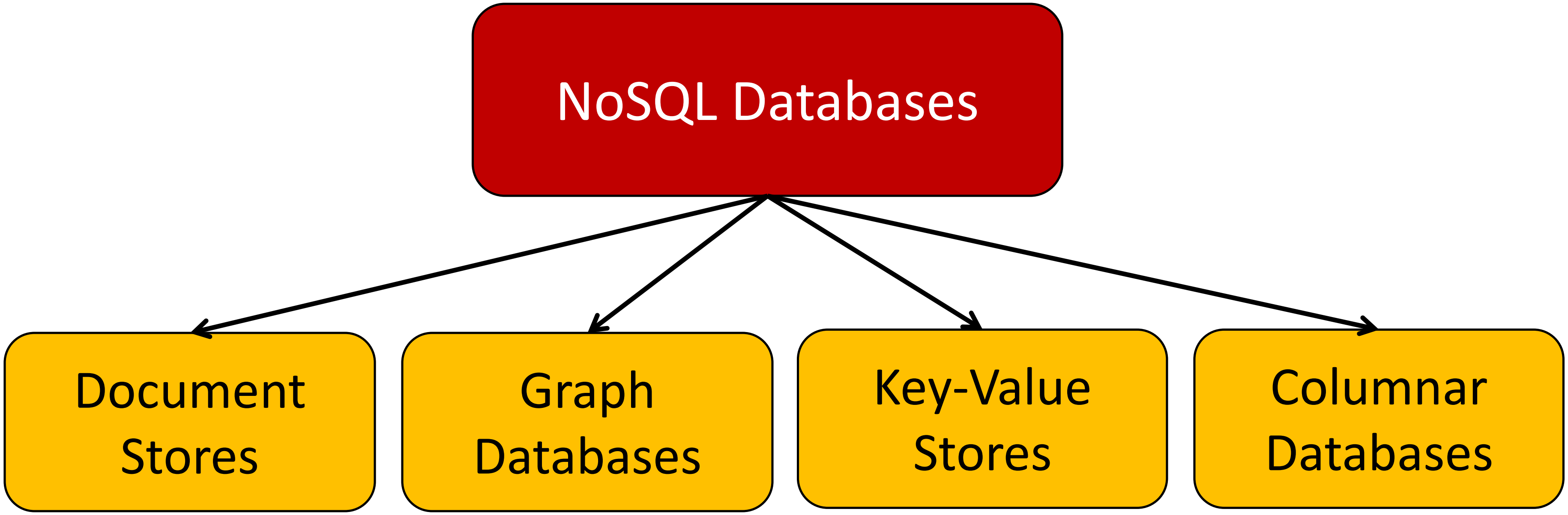
NoSQL Databases

- To this end, a new class of databases emerged, which mainly follow the BASE properties
  - These were dubbed as NoSQL databases
  - E.g., Amazon’s Dynamo and Google’s Bigtable

- Main characteristics of NoSQL databases include:
  - No strict schema requirements
  - No strict adherence to ACID properties
  - Consistency is traded in favor of Availability

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



NoSQL Databases

Document Stores

Graph Databases

Key-Value Stores

Columnar Databases

Document Stores

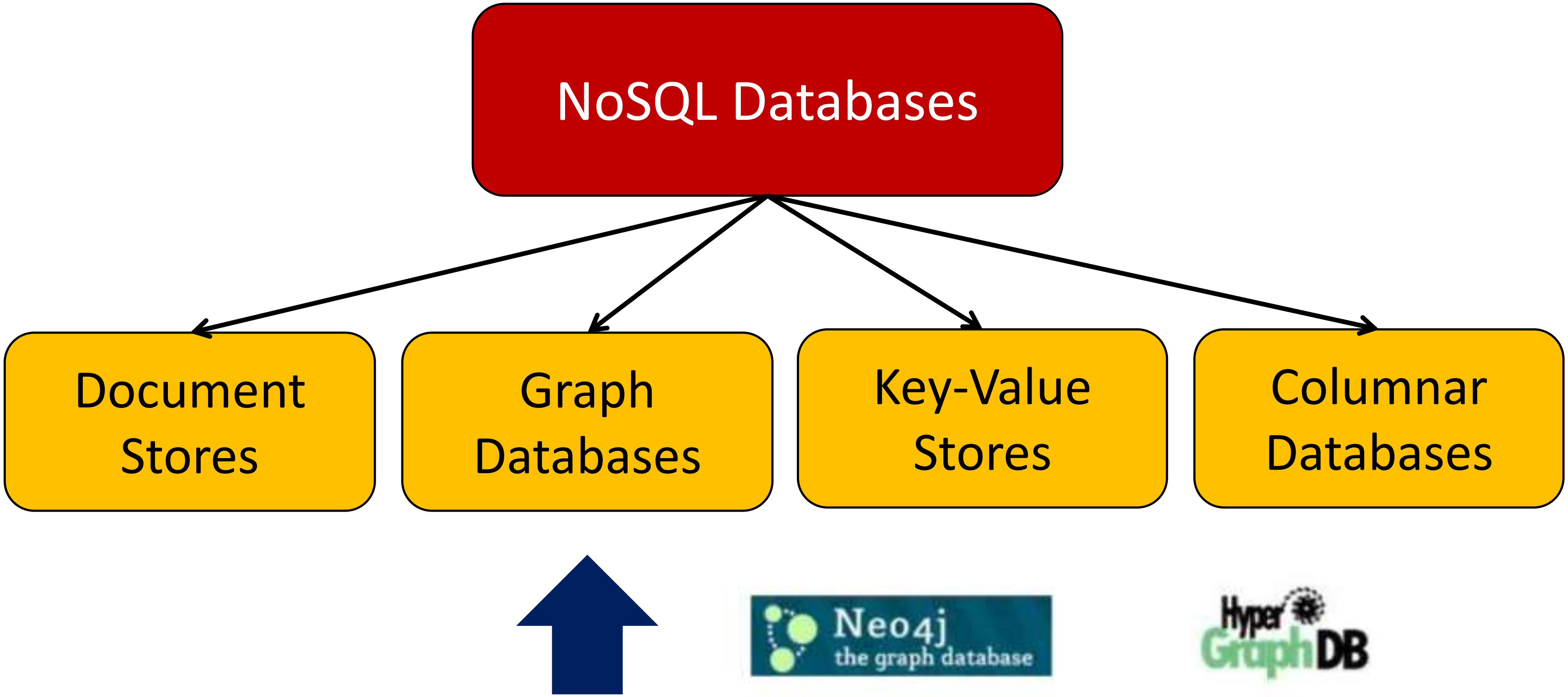
- Documents are stored in some standard format or encoding (e.g., XML, JSON, PDF or Office Documents)
  - These are typically referred to as Binary Large Objects (BLOBs)
- Documents can be indexed
  - This allows document stores to outperform traditional file systems
- E.g., MongoDB and CouchDB (both can be queried using MapReduce)

mongoDB

CouchDB
relax

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



NoSQL Databases

Document Stores

Graph Databases

Key-Value Stores

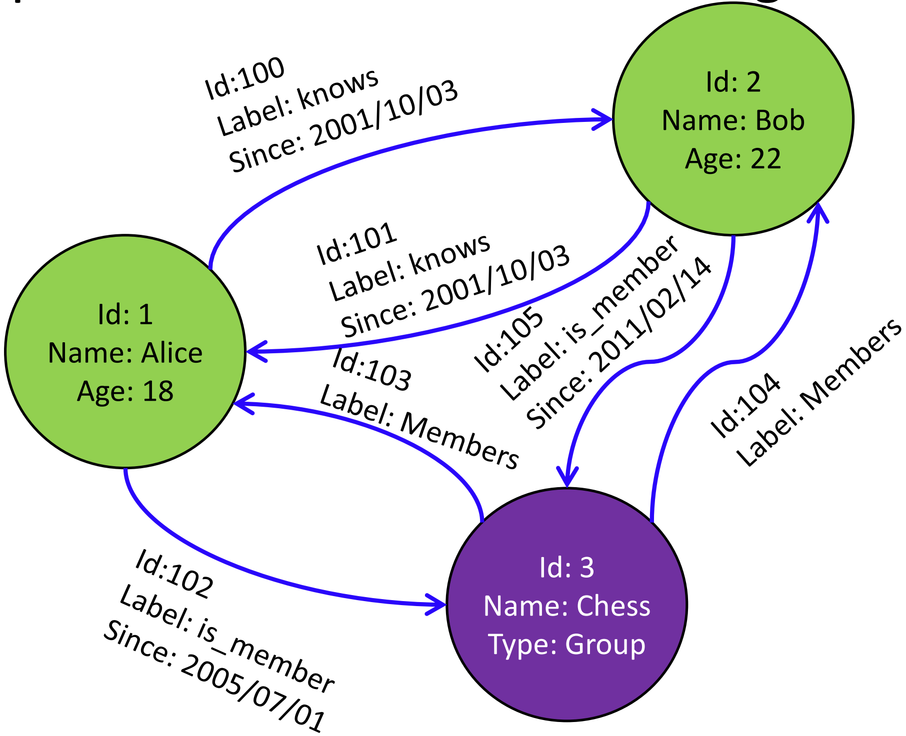
Columnar Databases

Neo4j
the graph database

HyperGraphDB

Graph Databases

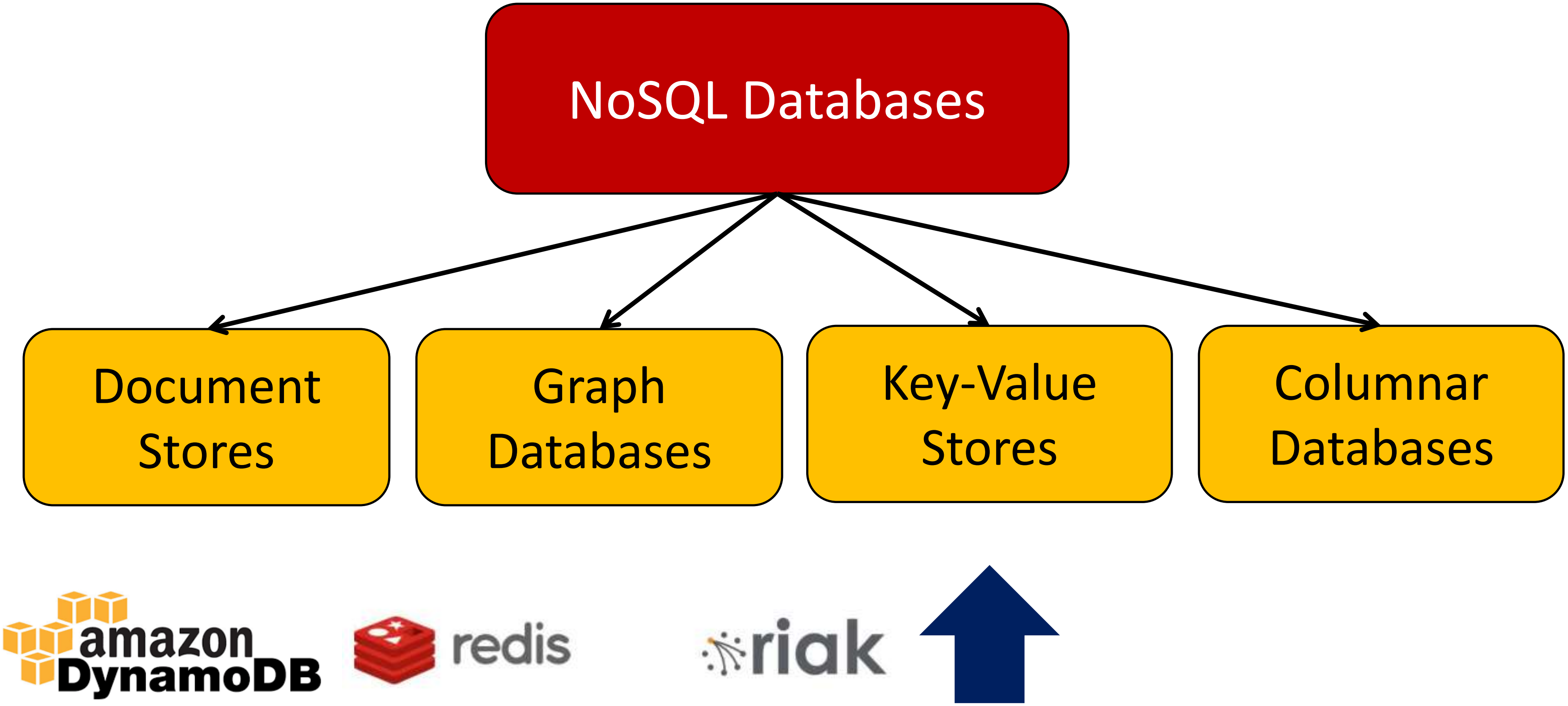
- Data are represented as vertices and edges



- Graph databases are powerful for graph-like queries (e.g., find the shortest path between two elements)
- E.g., Neo4j and VertexDB

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



NoSQL Databases

Document Stores

Graph Databases

Key-Value Stores

Columnar Databases

amazon DynamoDB

redis

riak

Key-Value Stores

- Keys are mapped to (possibly) more complex value (e.g., lists)
- Keys can be stored in a hash table and can be distributed easily
- Such stores typically support regular CRUD (create, read, update, and delete) operations
    - That is, no joins and aggregate functions

- E.g., Amazon DynamoDB and Apache Cassandra

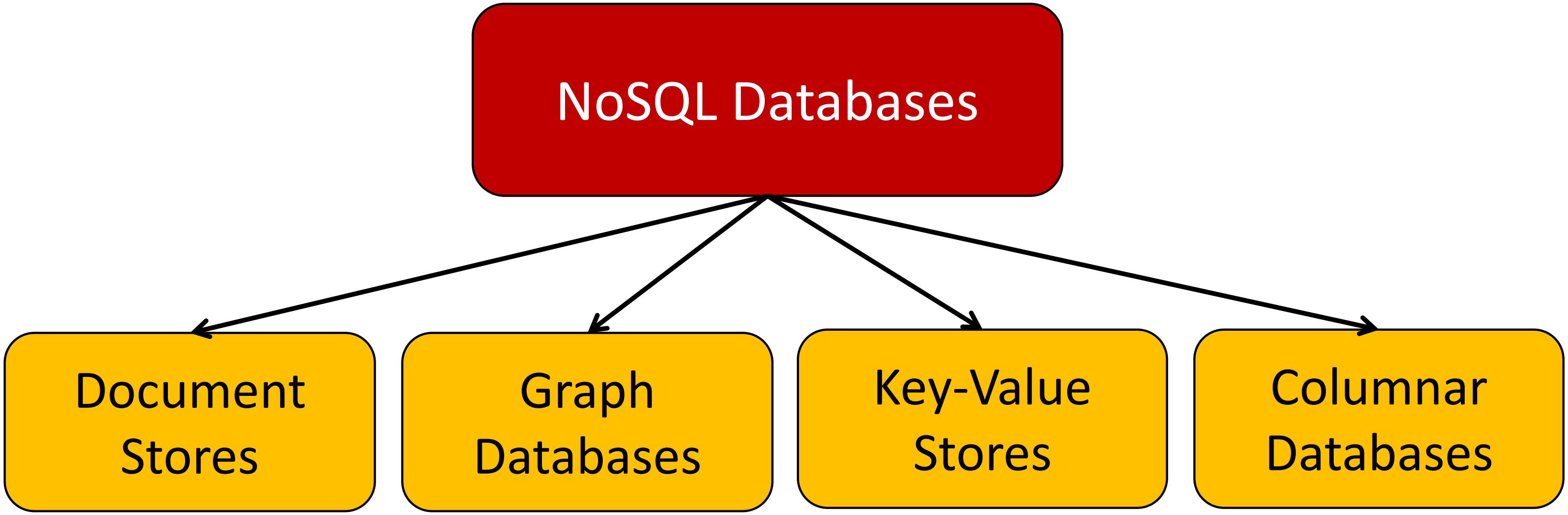
redis

riak

amazon DynamoDB

Types of NoSQL Databases

- Here is a limited taxonomy of NoSQL databases:



NoSQL Databases

Document Stores

Graph Databases

Key-Value Stores

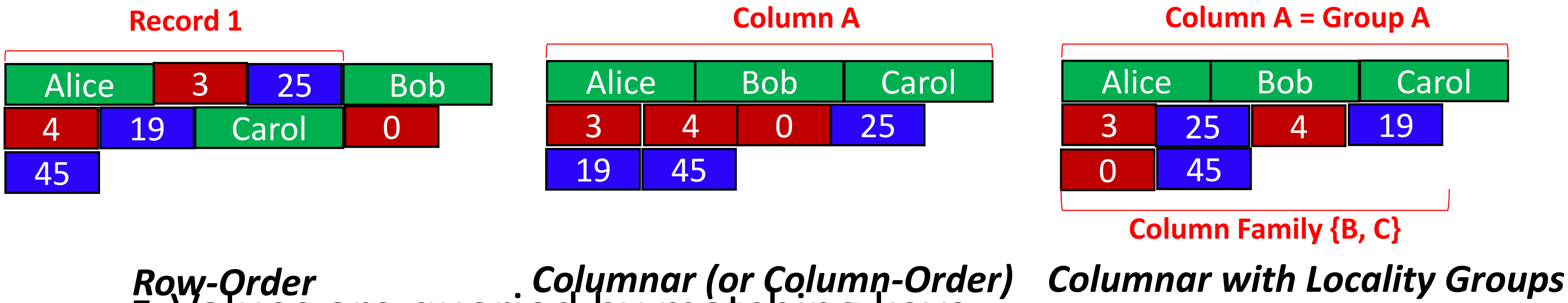
Columnar Databases

Cassandra

HBASE

Columnar Databases

- Columnar databases are a hybrid of RDBMSs and Key-Value stores
  - Values are stored in groups of zero or more columns, but in Column-Order (as opposed to Row-Order)



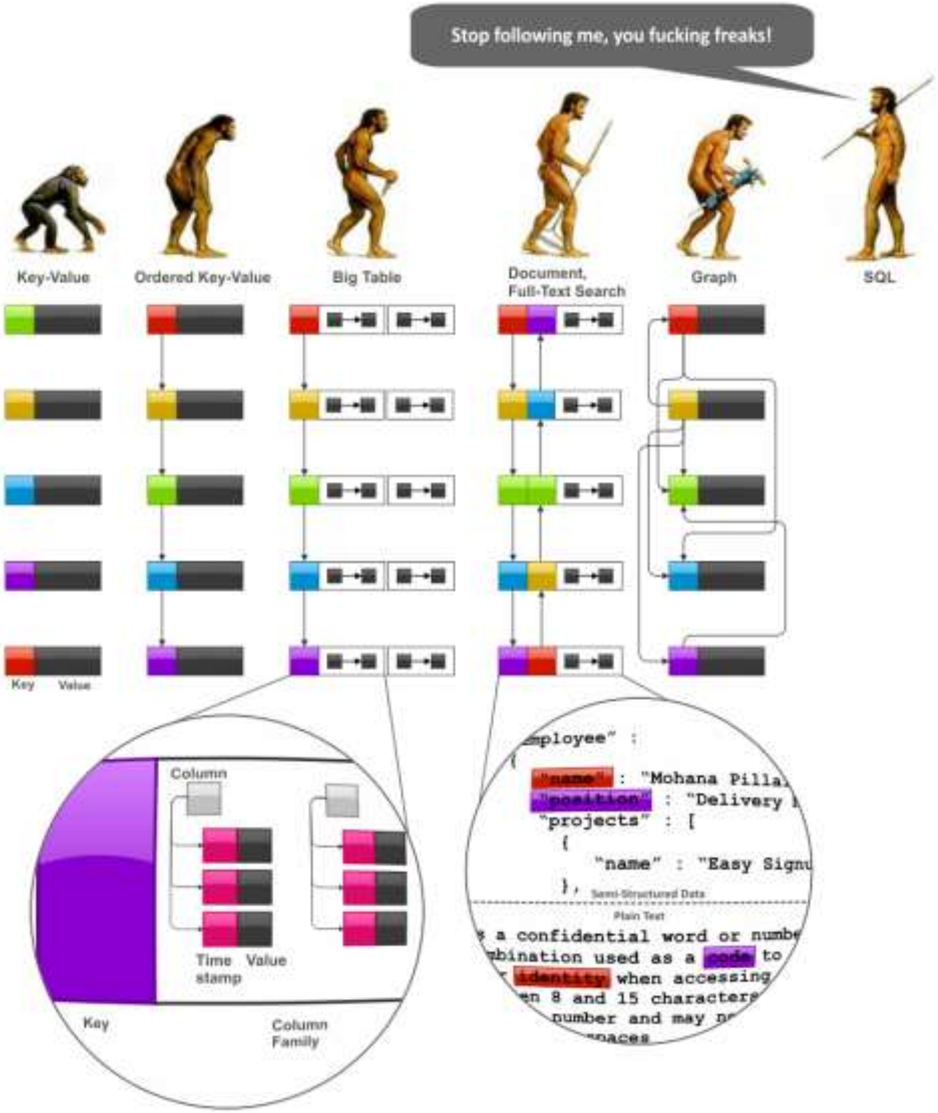
Row-Order

Columnar (or Column-Order)

Columnar with Locality Groups

  - Values are queried by matching keys
- E.g., HBase and Vertica

Revolution of Databases



7

Typical NoSQL Databases

Google BigTable

Google Cloud

- BigTable is a distributed storage system for managing structured data.
- Designed to scale to a very large size
  - Petabytes of data across thousands of servers
- Used for many Google projects
  - Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …
- Flexible, high-performance solution for all of Google's products

Motivation of BigTable Google Cloud

- Lots of (semi-)structured data at Google
  - URLs:
    - Contents, crawl metadata, links, anchors, pagerank, …
  - Per-user data:
    - User preference settings, recent queries/search results, …
  - Geographic locations:
    - Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, …
- Scale is large
  - Billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands or q/sec
  - 100TB+ of satellite image data

Design of BigTable

Google Cloud

- Distributed multi-level map
- Fault-tolerant, persistent
- Scalable
    - Thousands of servers
    - Terabytes of in-memory data
    - Petabyte of disk-based data
    - Millions of reads/writes per second, efficient scans
- Self-managing
    - Servers can be added/removed dynamically
    - Servers adjust to load imbalance

Building Blocks

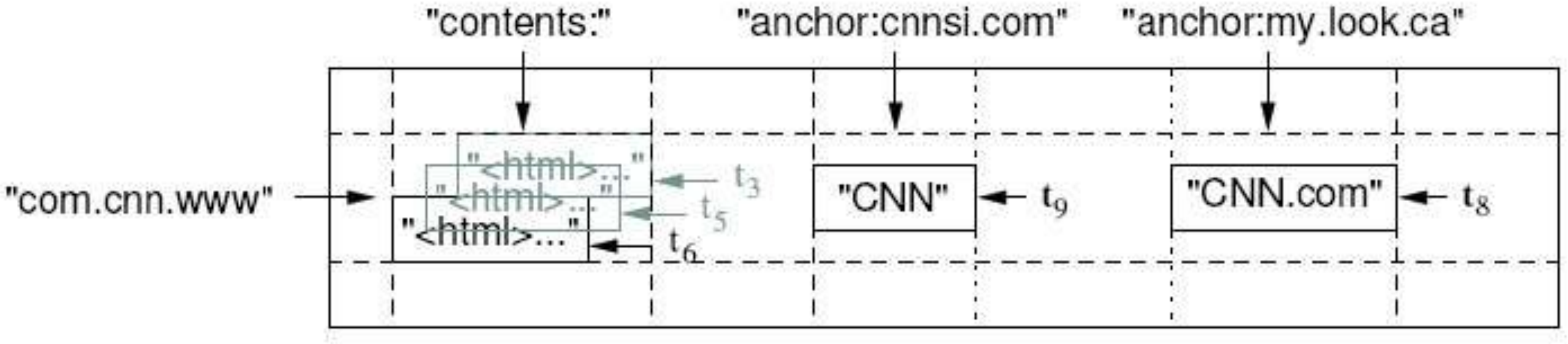
Google Cloud

- Building blocks:
  - Google File System (GFS): Raw storage
  - Scheduler: schedules jobs onto machines
  - Lock service: distributed lock manager
  - MapReduce: simplified large-scale data processing
- BigTable uses of building blocks:
  - GFS: stores persistent data (SSTable file format for storage of data)
  - Scheduler: schedules jobs involved in BigTable serving
  - Lock service: master election, location bootstrapping
  - Map Reduce: often used to read/write BigTable data

Basic Data Model

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

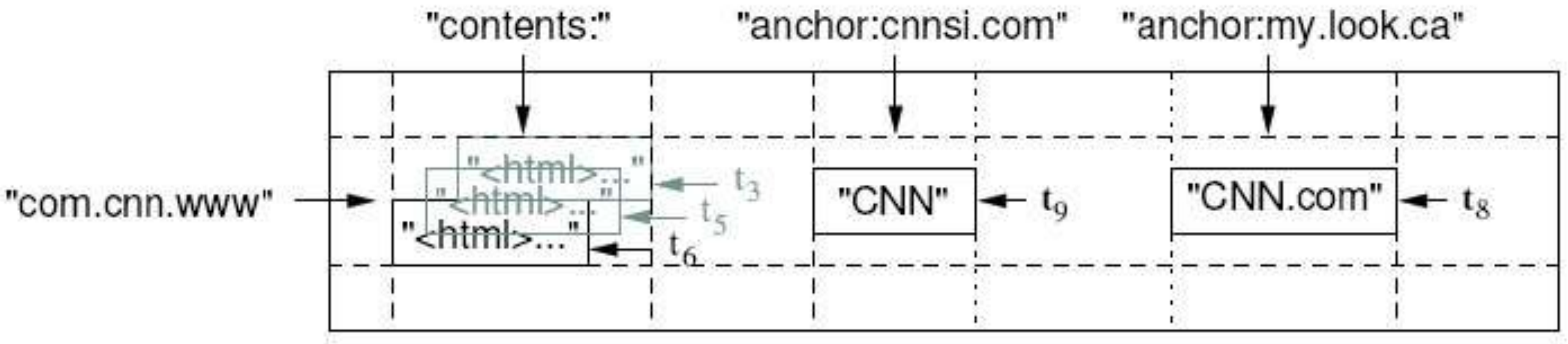
(row, column, timestamp) -> cell contents



- Good match for most Google applications

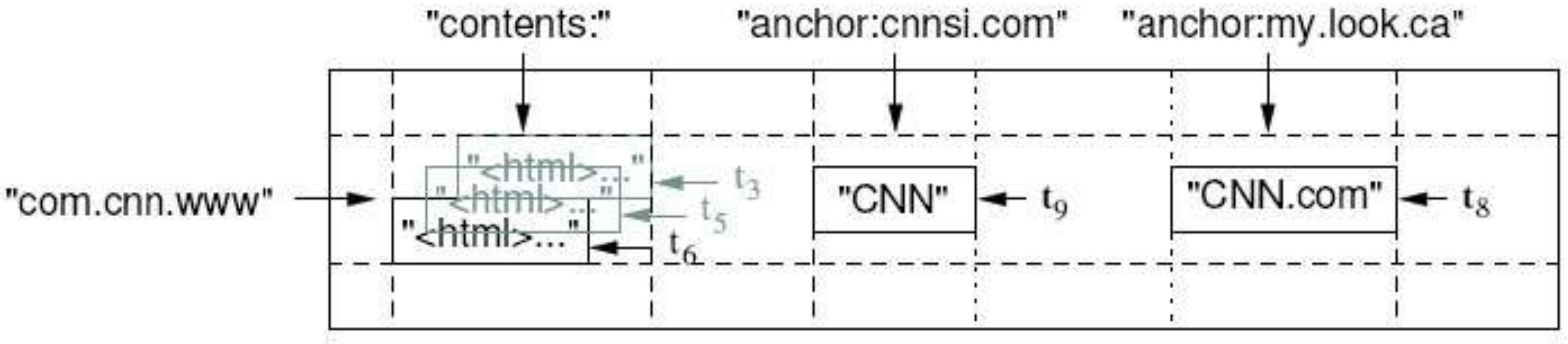
WebTable Example

Google Cloud



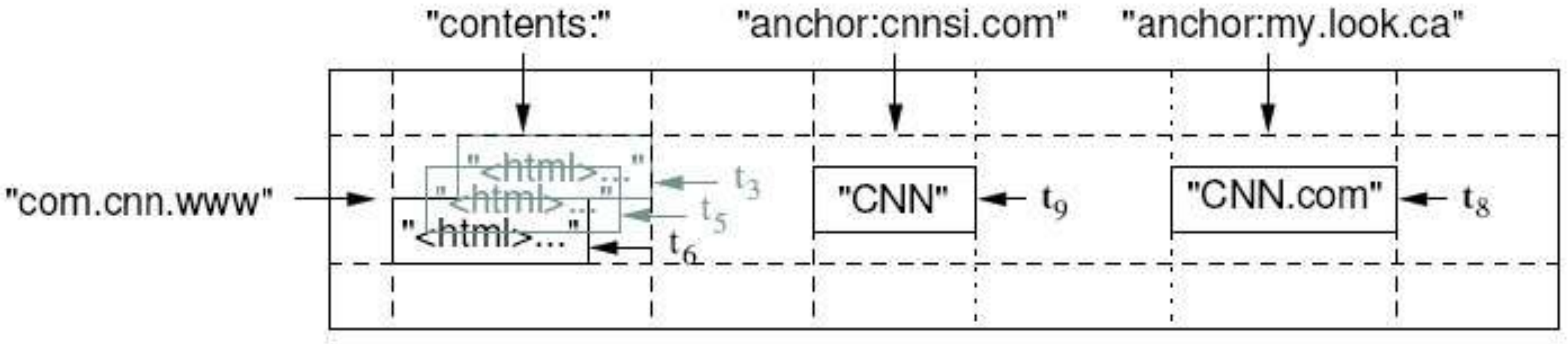
- Want to keep copy of a large collection of web pages and related information
- Use URLs as row keys
- Various aspects of web page as column names
- Store contents of web pages in the `contents:` column under the timestamps when they were fetched.

Rows



- Name is an arbitrary string
    - Access to data in a row is atomic
    - Row creation is implicit upon storing data
- Rows ordered lexicographically
    - Rows close together lexicographically usually on one or a small number of machines
- Reads of short row ranges are efficient and typically require communication with a small number of machines.

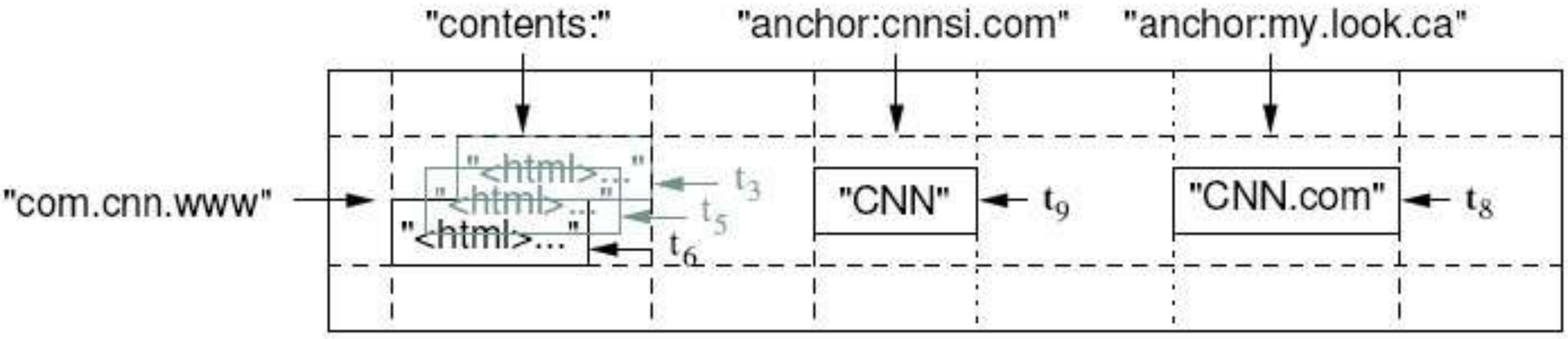
Columns



- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - Unit of access control
  - Has associated type information
- Qualifier gives unbounded columns
  - Additional levels of indexing, if desired

Timestamps

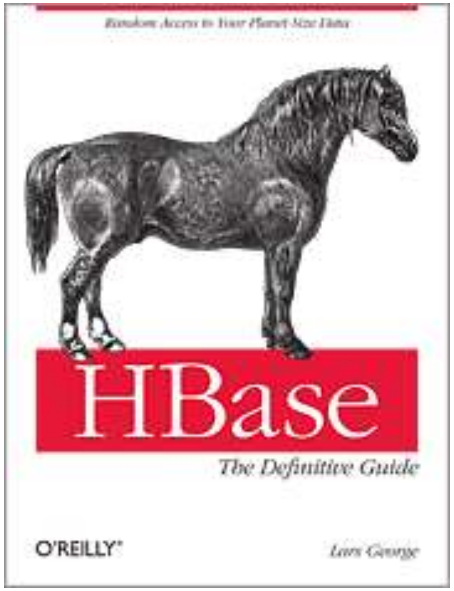
Google Cloud



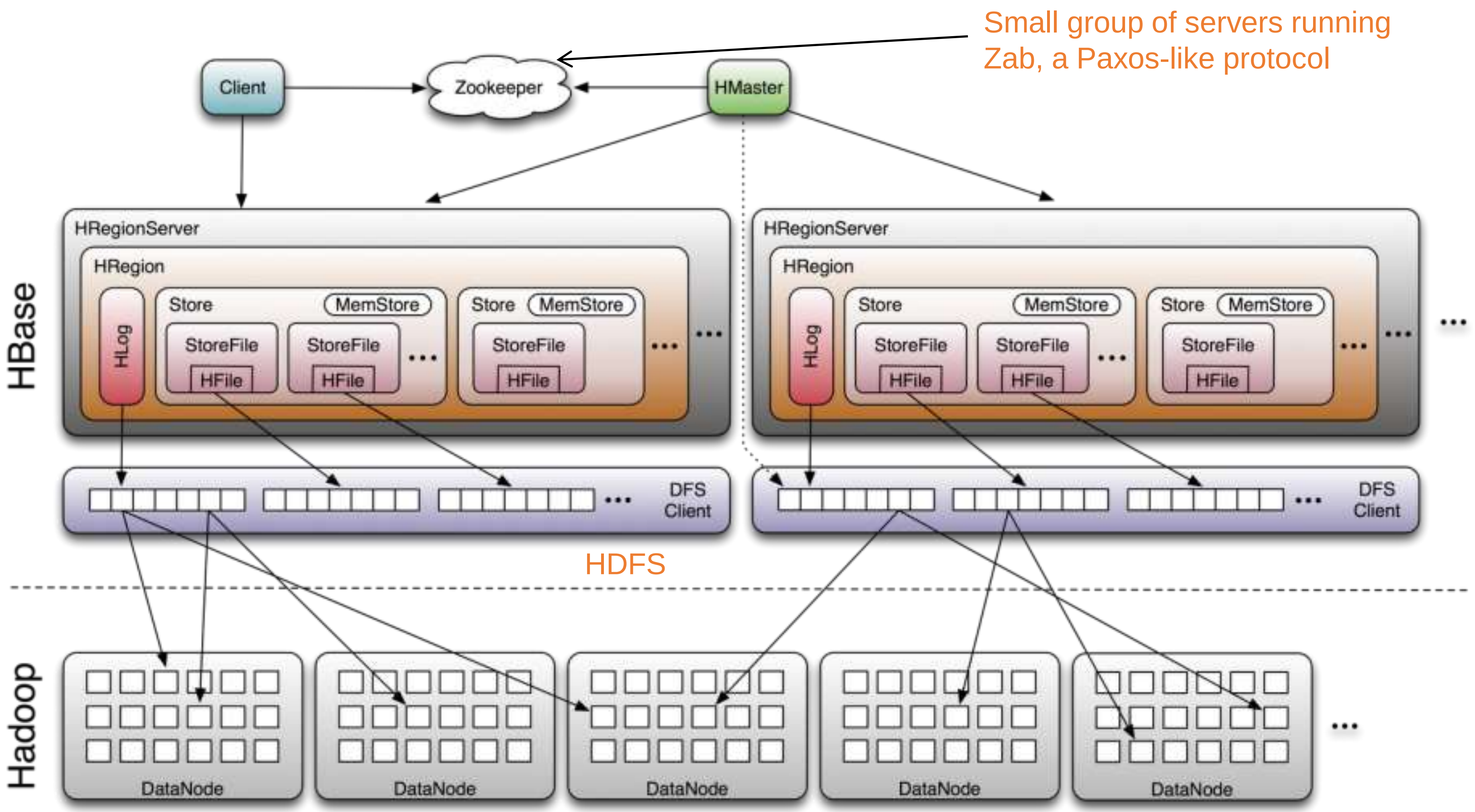
- Used to store different versions of data in a cell
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - *"Return most recent K values"*
  - *"Return all values in timestamp range (or all values)"*
- Column families can be marked w/ attributes:
  - *"Only retain most recent K values in a cell"*
  - *"Keep values until they are older than K seconds"*

HBase

- Google's BigTable was first "blob-based" storage system
- Yahoo! Open-sourced it → Hbase (2007)
- Major Apache project today
- Facebook uses HBase internally
- API
    - Get/Put(row)
    - Scan(row range, filter) – range queries
    - MultiPut



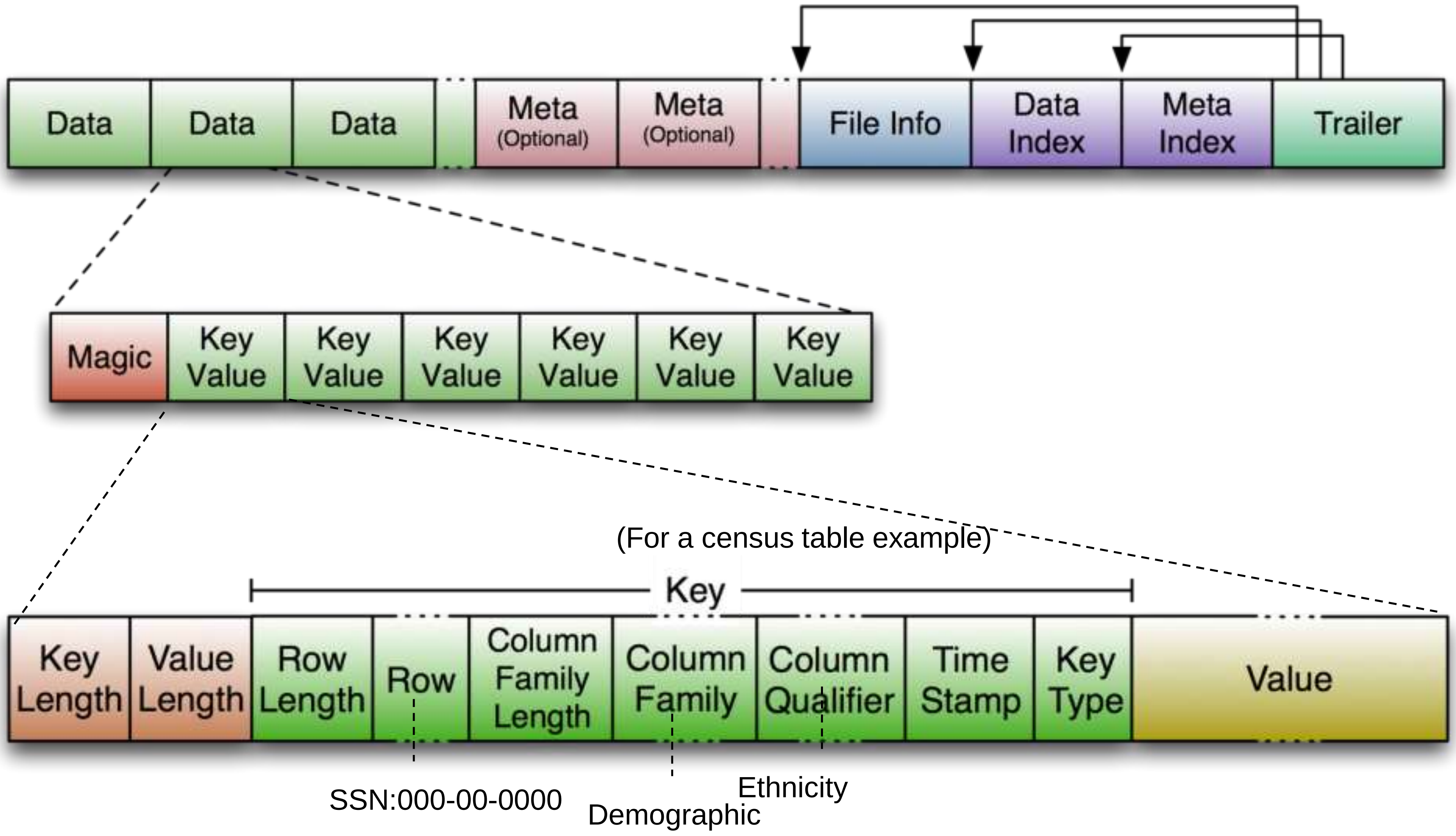
HBase Architecture



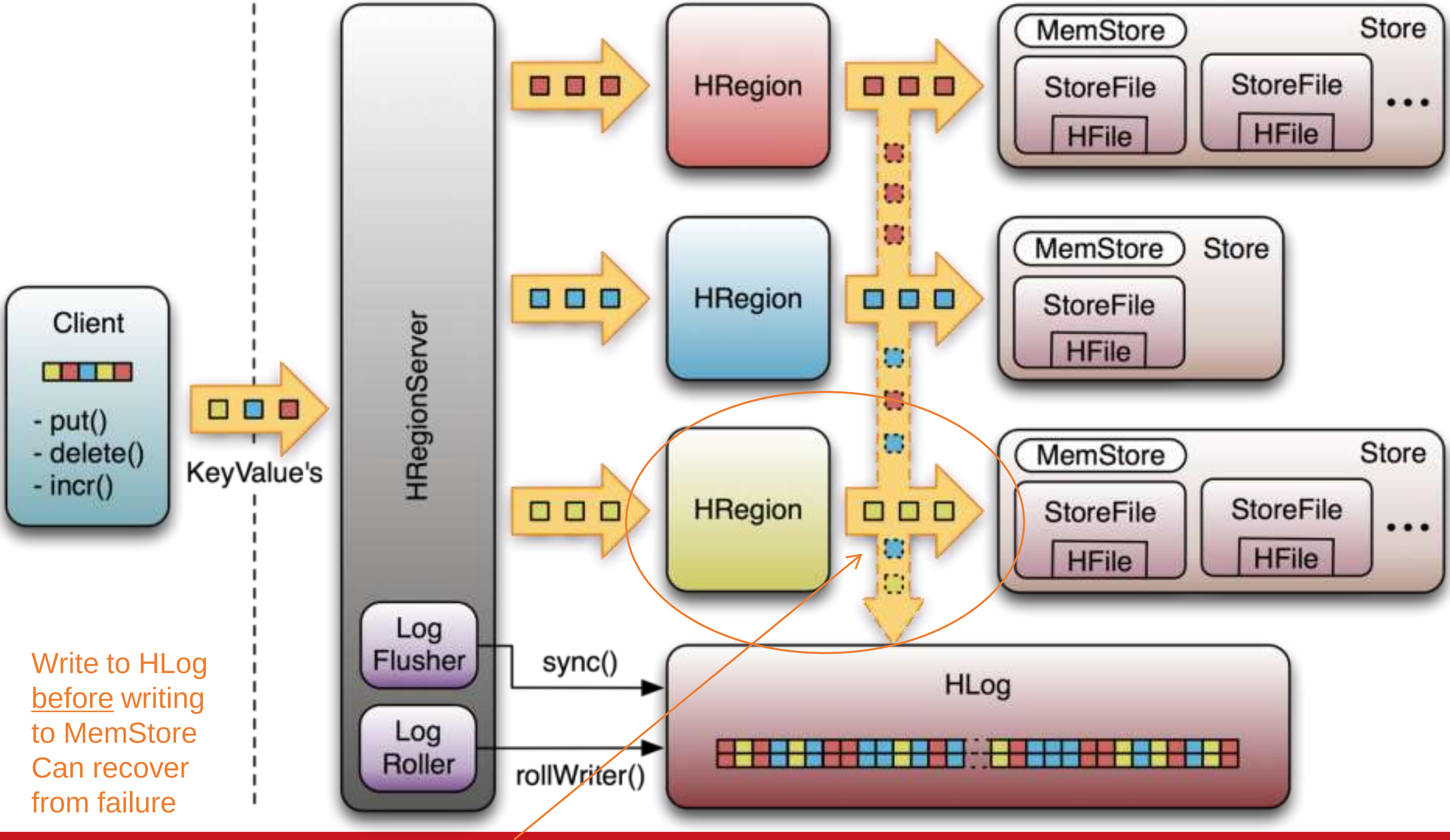
HBase Storage Hierarchy

- HBase Table
  - Split it into multiple regions: replicated across servers
    - One Store per ColumnFamily (subset of columns with similar query patterns) per region
      - Memstore for each Store: in-memory updates to Store; flushed to disk when full
        - StoreFiles for each store for each region: where the data lives
          - Blocks
- HFile
  - SSTable from Google's BigTable

HFile


Data
Data
Data
Meta (Optional)
Meta (Optional)
File Info
Data Index
Meta Index
Trailer
Magic
Key Value
Key Value
Key Value
Key Value
Key Value
Key Value
(For a census table example)
Key
Key Length
Value Length
Row Length
Row
Column Family Length
Column Family
Column Qualifier
Time Stamp
Key Type
Value
SSN:000-00-0000
Demographic
Ethnicity

Strong Consistency: HBase Write-Ahead Log

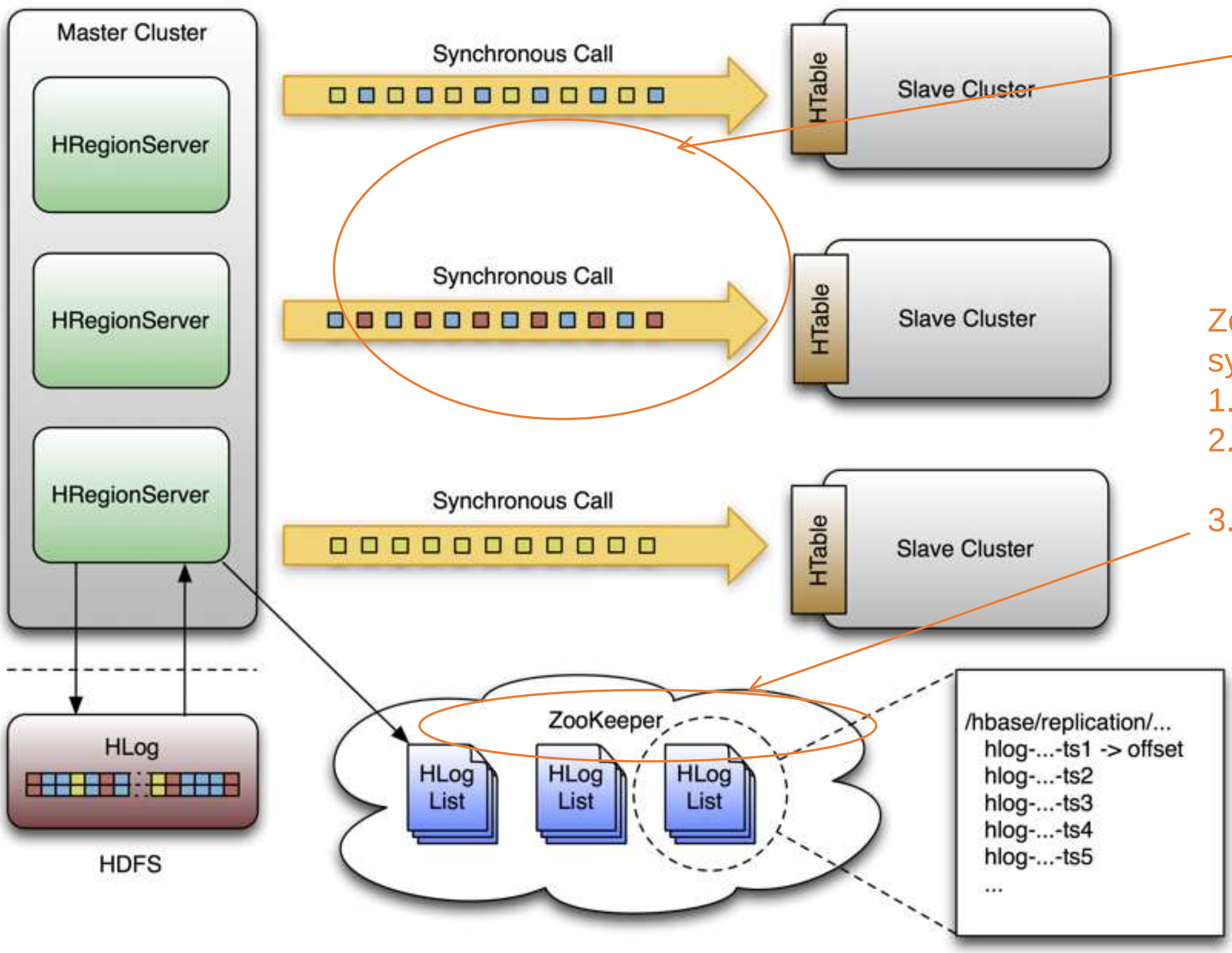


Write to HLog before writing to MemStore
Can recover from failure

Log Replay

- After recovery from failure, or upon bootup (HRegionServer/HMaster)
  - Replay any stale logs (use timestamps to find out where the database is w.r.t. the logs)
  - Replay: add edits to the MemStore
- Why one HLog per HRegionServer rather than per region?
  - Avoids many concurrent writes, which on the local file system may involve many disk seeks

Cross-data center replication



HLog

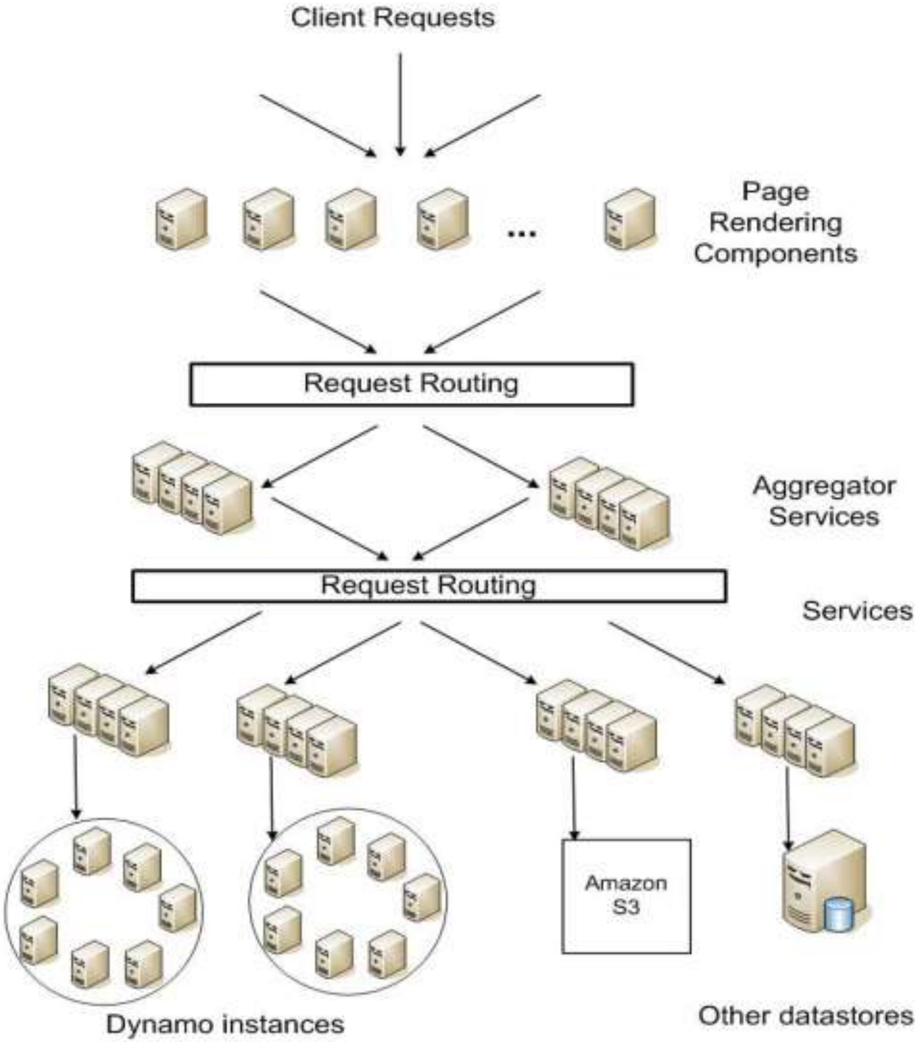
Zookeeper actually a file system for control information

1. /hbase/replication/state
2. /hbase/replication/peers
   /<peer cluster number>
3. /hbase/replication/rs/<hlog>

Dynamo: Amazon’s Highly Available Key-value Store

aws

Architecture



Dynamo: The big picture

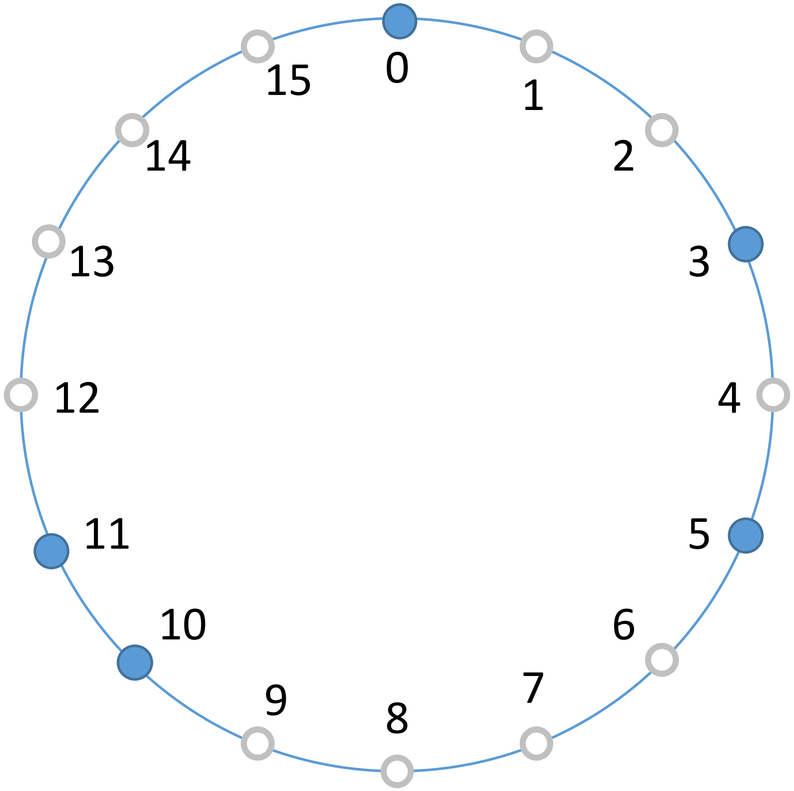
- Easy usage
- Load-balancing
- Replication
- Eventual consistency
- High availability
- Easy management
- Failure-detection
- Scalability

Easy usage: Interface

- get(key)
  - return single object or list of objects with conflicting version and context
- put(key, context, object)
  - store object and context under key
- Context encodes system meta-data, e.g. version number

Data Partitioning

- Based on consistent hashing
- Hash key and put on responsible node

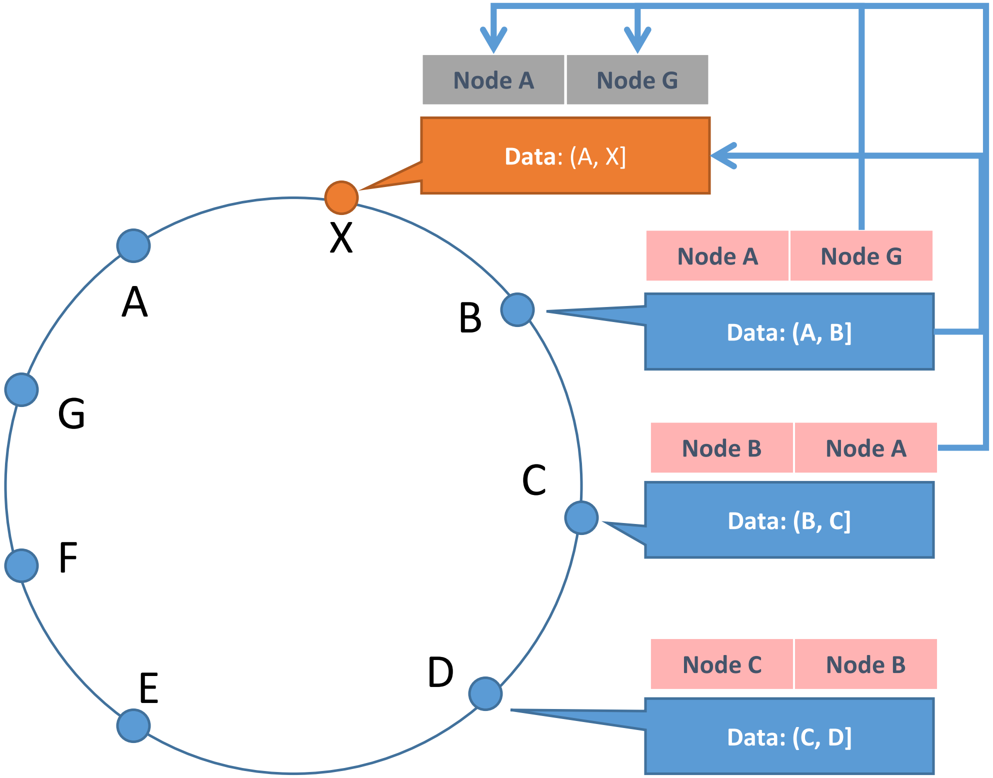


Load balancing

- Load
  - Storage bits
  - Popularity of the item
  - Processing required to serve the item
  - …
- Consistent hashing may lead to imbalance

Adding nodes

- A new node X added to system
- X is assigned key ranges w.r.t. its virtual servers
- For each key range, it transfers the data items



X=Data\(X,B)
Data=Data\(A,X)
Drop G

X=B\(X,B)
B=B\(A,X)
Drop A

Removing nodes

- Reallocation of keys is a reverse process of adding nodes

Implementation details

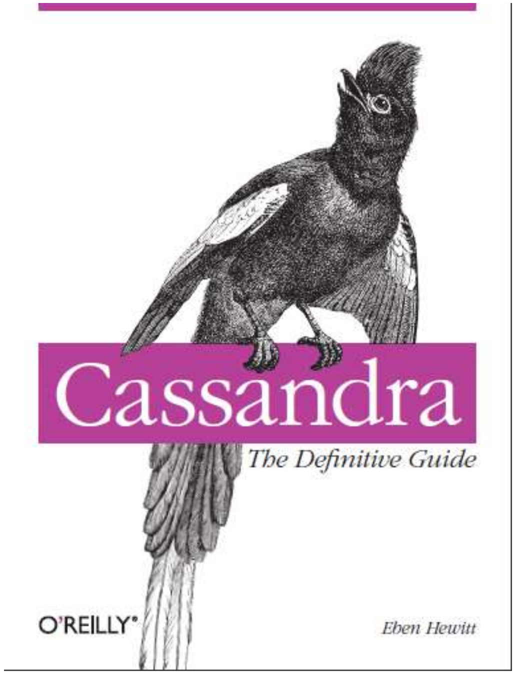
- Local persistence
  - BDS, MySQL, etc.
- Request coordination
  - Read operation
    - Create context
    - Syntactic reconciliation
    - Read repair
  - Write operation
    - Read-your-writes

Apache Cassandra

cassandra

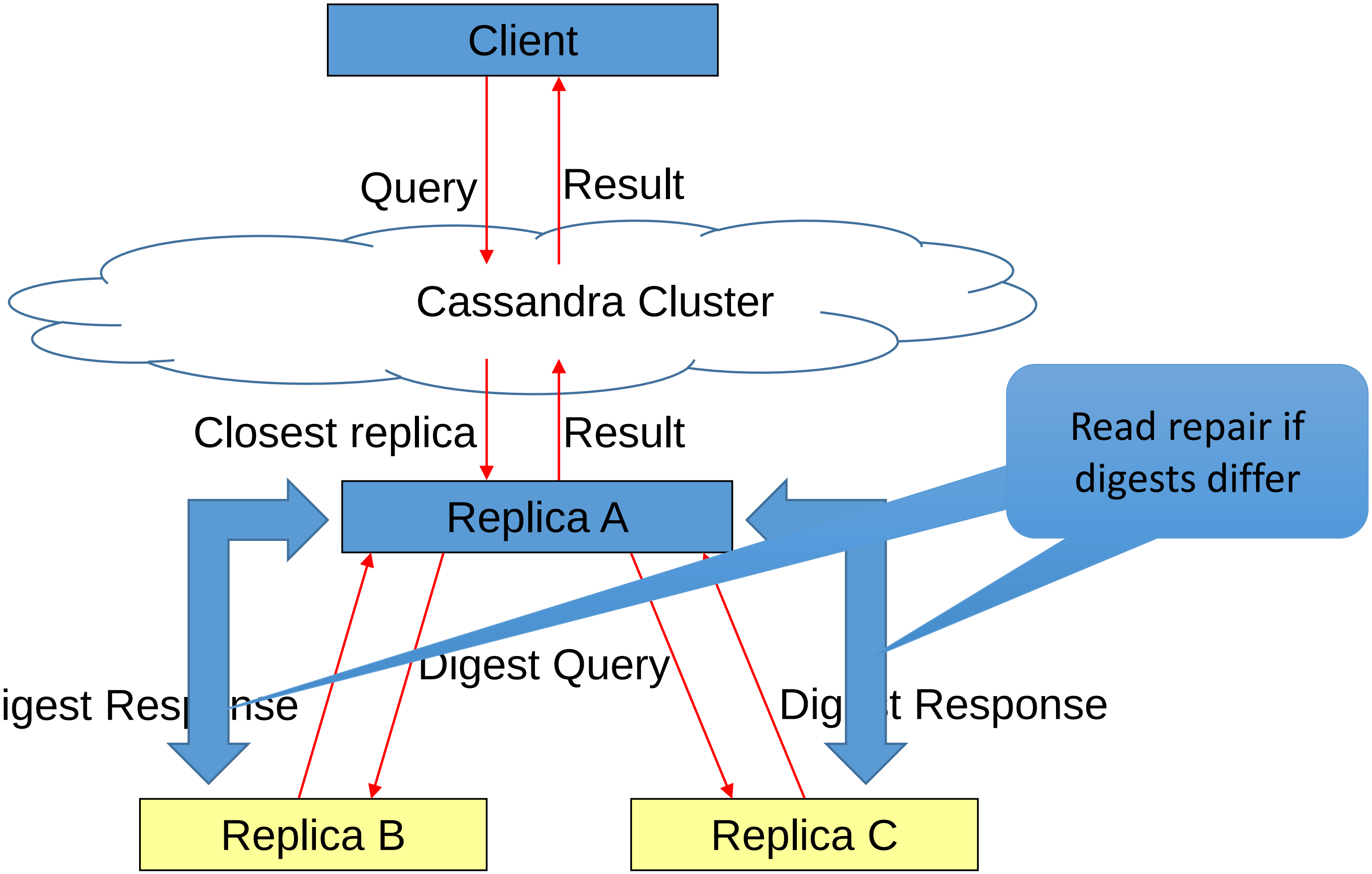
- Originally designed at Facebook (July 2008)
- Open-sourced
- Some of its myriad users:

Adobe
AppScale
IBM
Symantec.
twitter
ebaY
PBS KIDS
Spotify
hp
NETFLIX
ERICSSON



Cassandra
The Definitive Guide
O'REILLY
Eben Hewitt

Read operation



Read repair if digests differ

Facebook Inbox Search

- Cassandra developed to address this problem.
- 50+TB of user messages data in 150 node cluster on which Cassandra is tested.
- Search user index of all messages in 2 ways.
  - Term search : search by a key word
  - Interactions search : search by a user id

Latency Stat	Search Interactions	Term Search
Min	7.69 ms	7.78 ms
Median	15.69 ms	18.27 ms
Max	26.13 ms	44.41 ms

Facebook Inbox Search

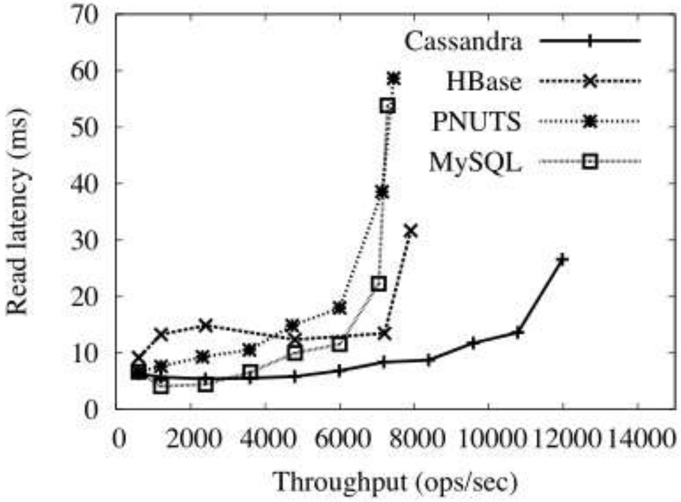
- MySQL > 50 GB Data
  Writes Average : ~300 ms
  Reads Average : ~350 ms

- Cassandra > 50 GB Data
  Writes Average : 0.12 ms
  Reads Average : 15 ms

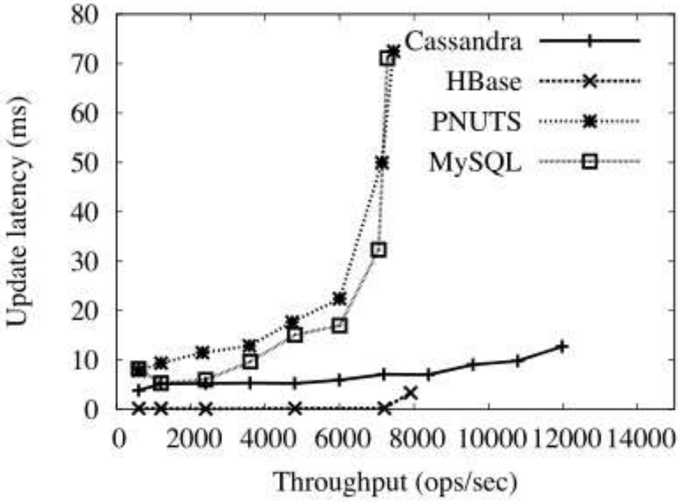
- Stats provided by Authors using facebook data.

Comparison using YCSB

- Cassandra, HBase and PNUTS were able to grow elastically while the workload was executing.
- PNUTS and Cassandra scaled well as the number of
- servers and workload increased proportionally. HBase’s performance was more erratic as the system scaled.

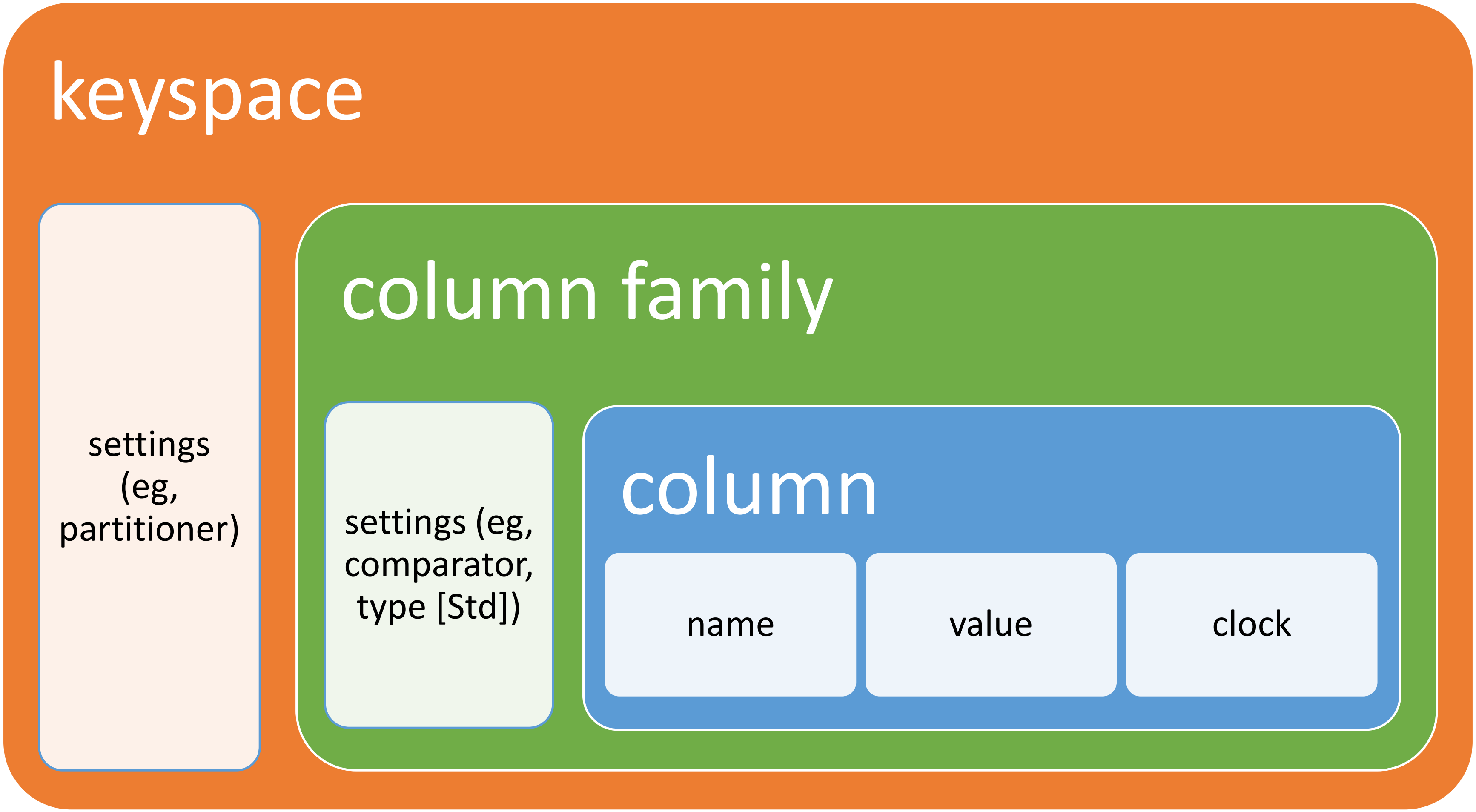


(a)



(b)

Structure



keyspace

settings (eg, partitioner)

column family

settings (eg, comparator, type [Std])

column

name

value

clock

Keyspace

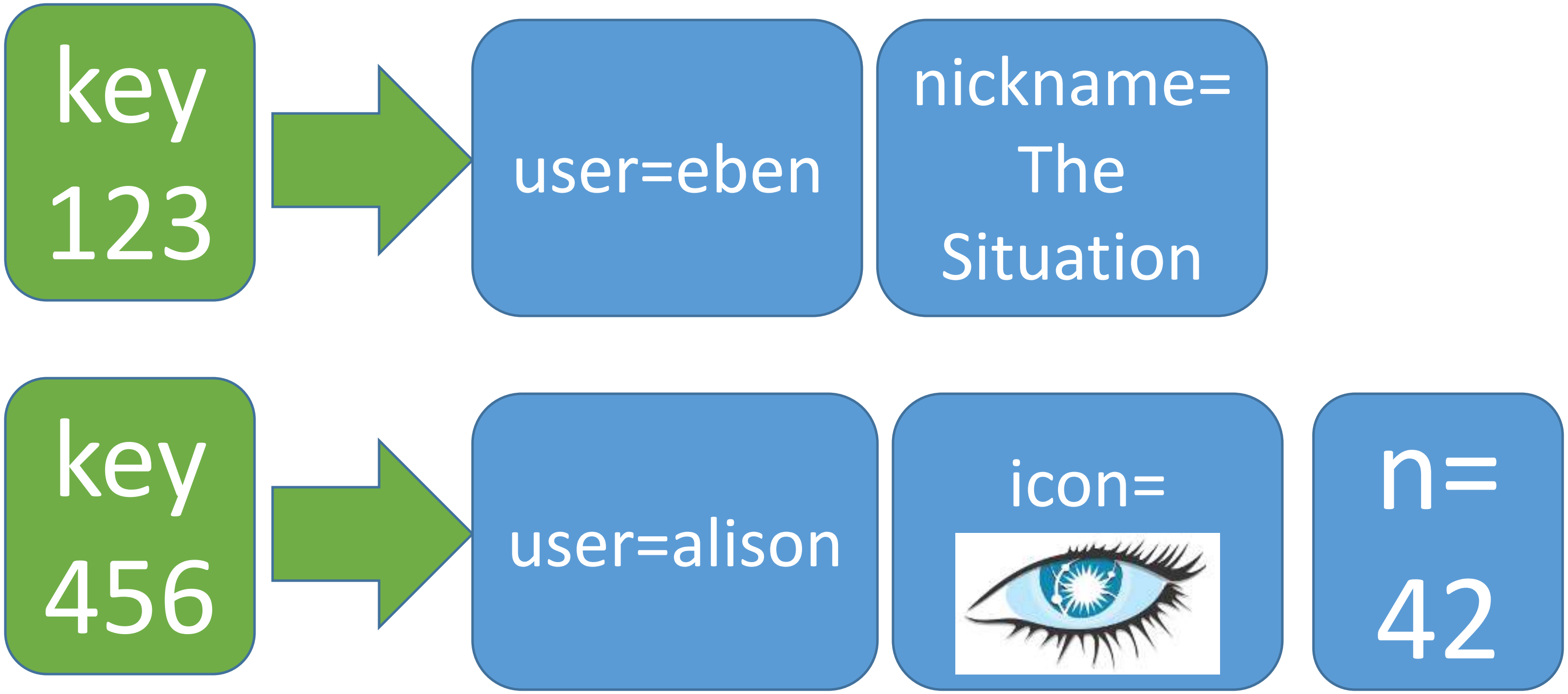
- ~= database
- typically one per application
- some settings are configurable only per keyspace

Column Family (CF)

- group records of *similar* kind
- not *same* kind, because CFs are sparse tables
- ex:
  - User
  - Address
  - Tweet
  - PointOfInterest
  - HotelRoom

Column Family (CF)

cassandra



key 123

user=eben

nickname= The Situation

key 456

user=alison

icon=

n= 42

JSON(JavaScript Object Notation)-like notation

User {

123 : { email: alison@foo.com,

icon:  },

456 : { email: eben@bar.com,

location: The Danger Zone}

}

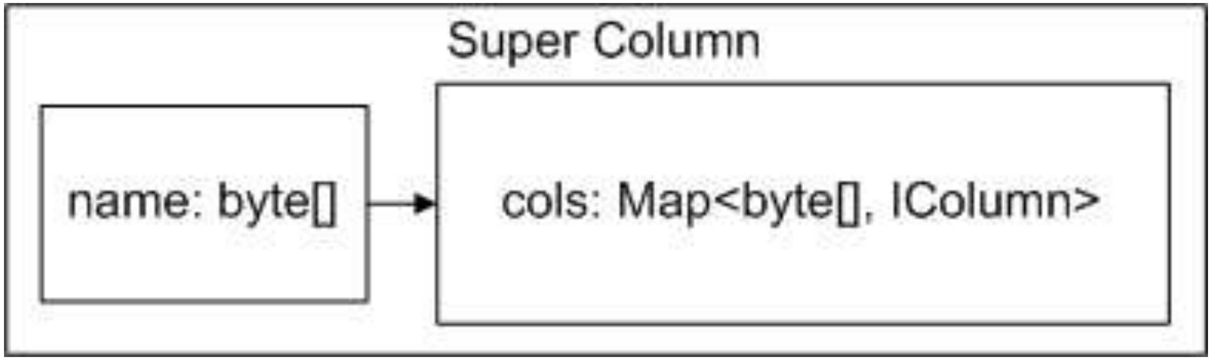
A column has 3 parts

cassandra

1. name
    - byte[]
    - determines sort order
    - used in queries
    - indexed
2. value
    - byte[]
    - *you don’t query on column values*
3. timestamp
    - long (clock)
    - last write wins conflict resolution

super column

cassandra



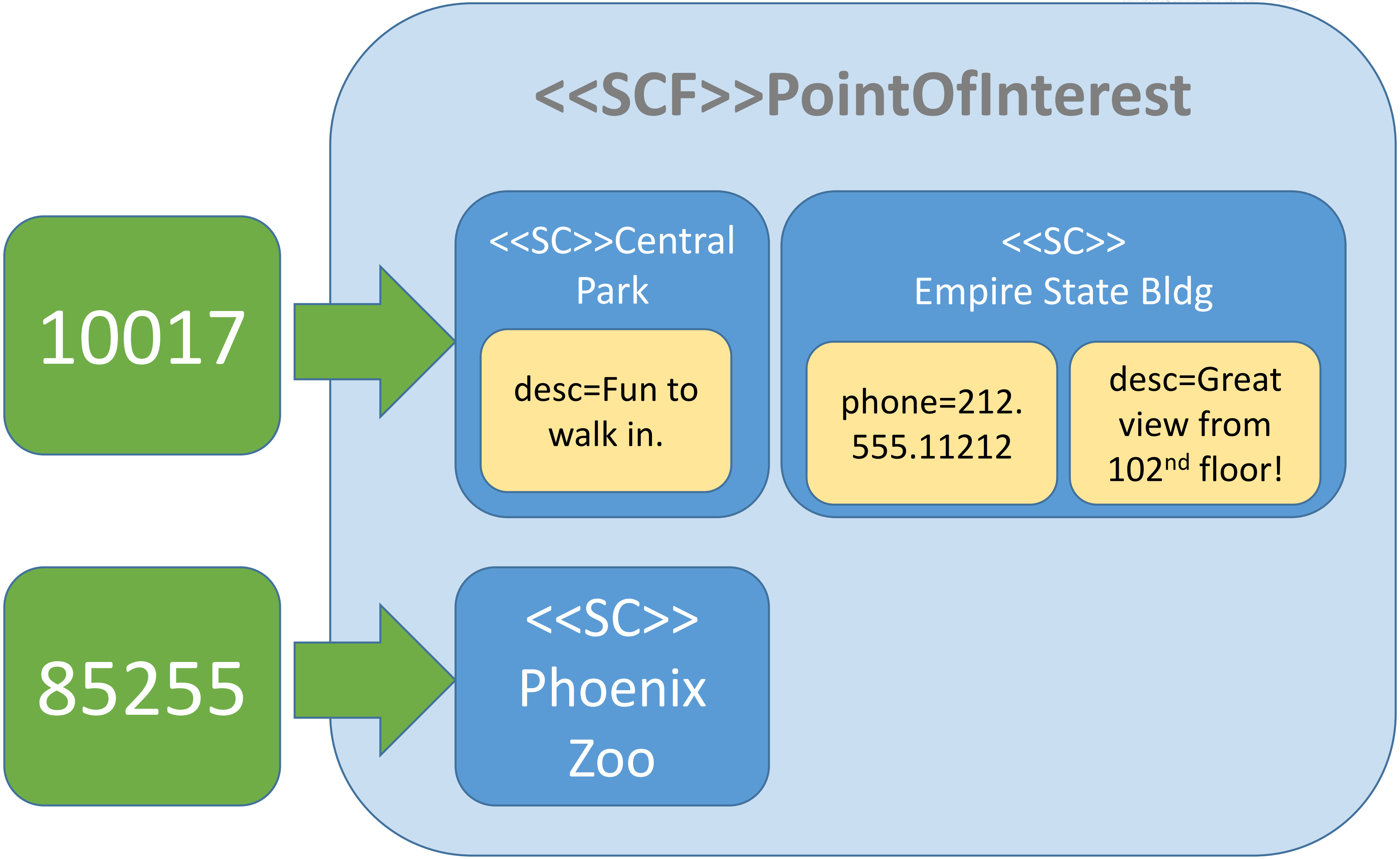
Super Column

name: byte[]

cols: Map<byte[], IColumn>

super columns group columns under a common name

super column family



<<SCF>>PointOfInterest

10017

<<SC>>Central Park

desc=Fun to walk in.

<<SC>>
Empire State Bldg

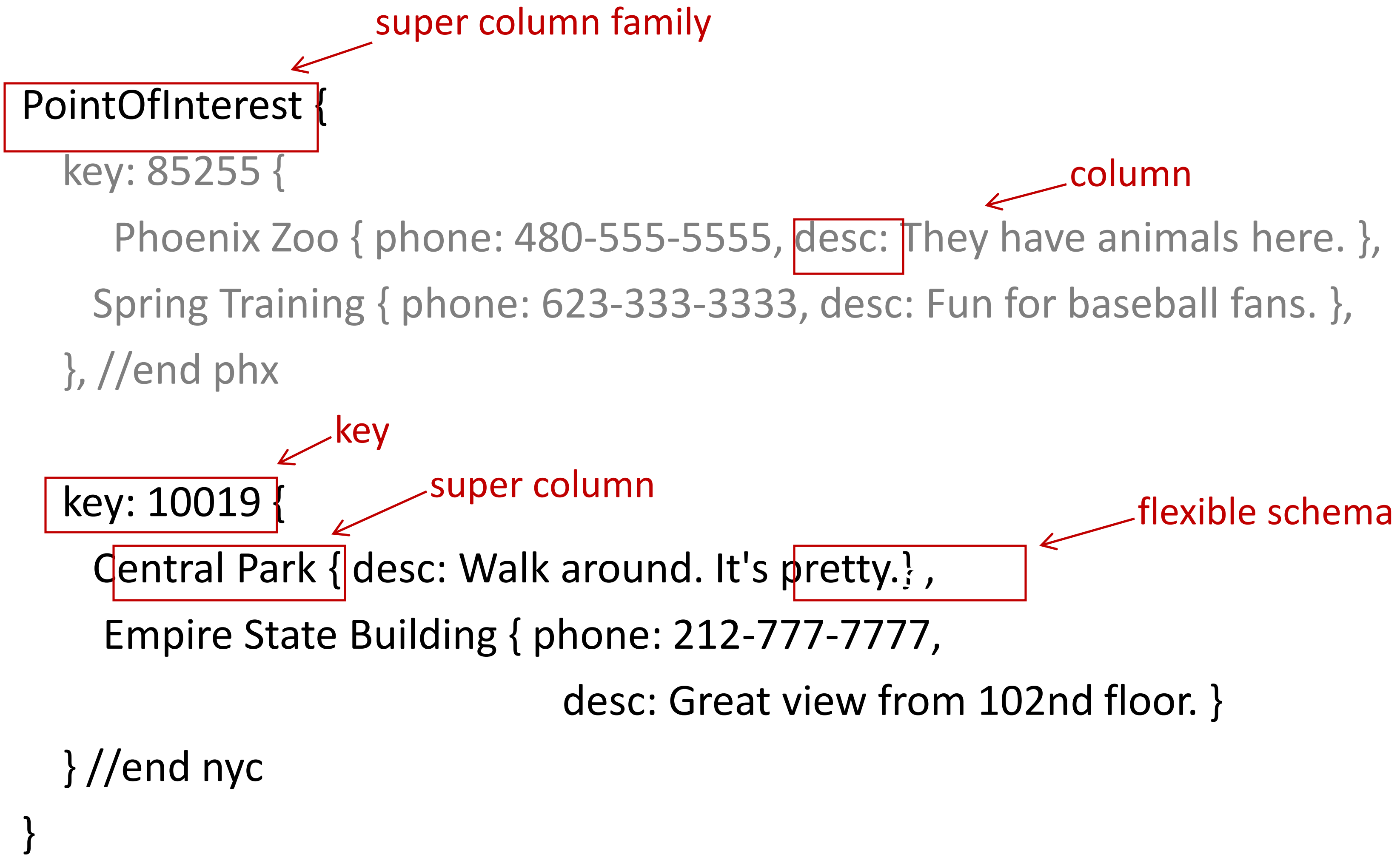
phone=212.555.11212

desc=Great view from 102nd floor!

85255

<<SC>>
Phoenix Zoo

super column family



super column family

column

key

super column

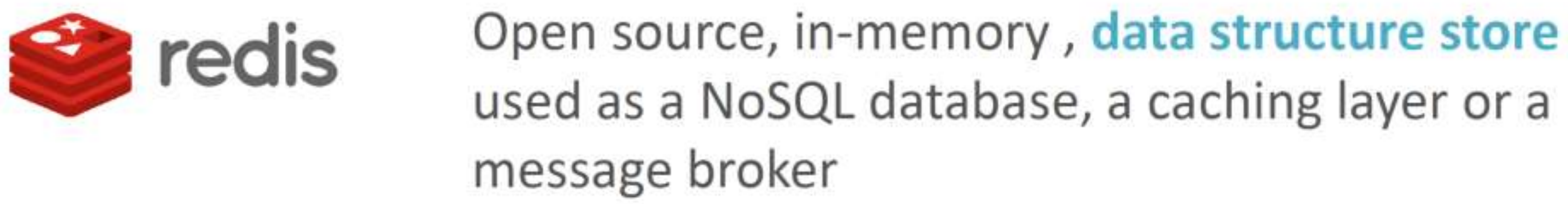
flexible schema

```
PointOfInterest {
    key: 85255 {
        Phoenix Zoo { phone: 480-555-5555, desc: They have animals here. },
      Spring Training { phone: 623-333-3333, desc: Fun for baseball fans. },
    }, //end phx

    key: 10019 {
      Central Park { desc: Walk around. It's pretty.} ,
       Empire State Building { phone: 212-777-7777,
                                              desc: Great view from 102nd floor. }
    } //end nyc
}
```

What is Redis

- an in-memory key-value store, with persistence
- open source, written in C
- “can handle up to 2^{32} keys, and was tested in practice to handle at least 250 million of keys per instance.” http://redis.io/topics/faq
- History
  - REmote DIctionary Server, released in Mar. 2009



Open source, in-memory , data structure store used as a NoSQL database, a caching layer or a message broker

Redis Tops Database Popularity Ranking

redis

G2 CROWD	........#1 NoSQL in User Satisfaction and Market Presence
stackshare	........#1 NoSQL among Top 10 Data Stores
DATADOG	........#1 database on Docker
ClusterHQ DevOps.com	#1 NoSQL database deployed in containers
DB-ENGINES	.........#1 in growth among top 3 NoSQL databases
UpScored	.........#1 database in skill demand

Redis: the cloud native database

redis

GAME CHANGER

A new database concept based on **Data Structures & Lua scripting**

Data structures used by developers like “Lego” building blocks

HIGH PERFORMANCE

World’s fastest & most powerful database:
1.5M ops/sec @ <1msec with a single cloud instance

DYNAMIC COMMUNITY

Largest open source community among the NoSQL databases.
Compatible with most programming languages and environments.

Strings
Sets
Sorted Sets
Lists
Hashes
Hyperlog-logs
Bitmaps
Geospatial Indexes

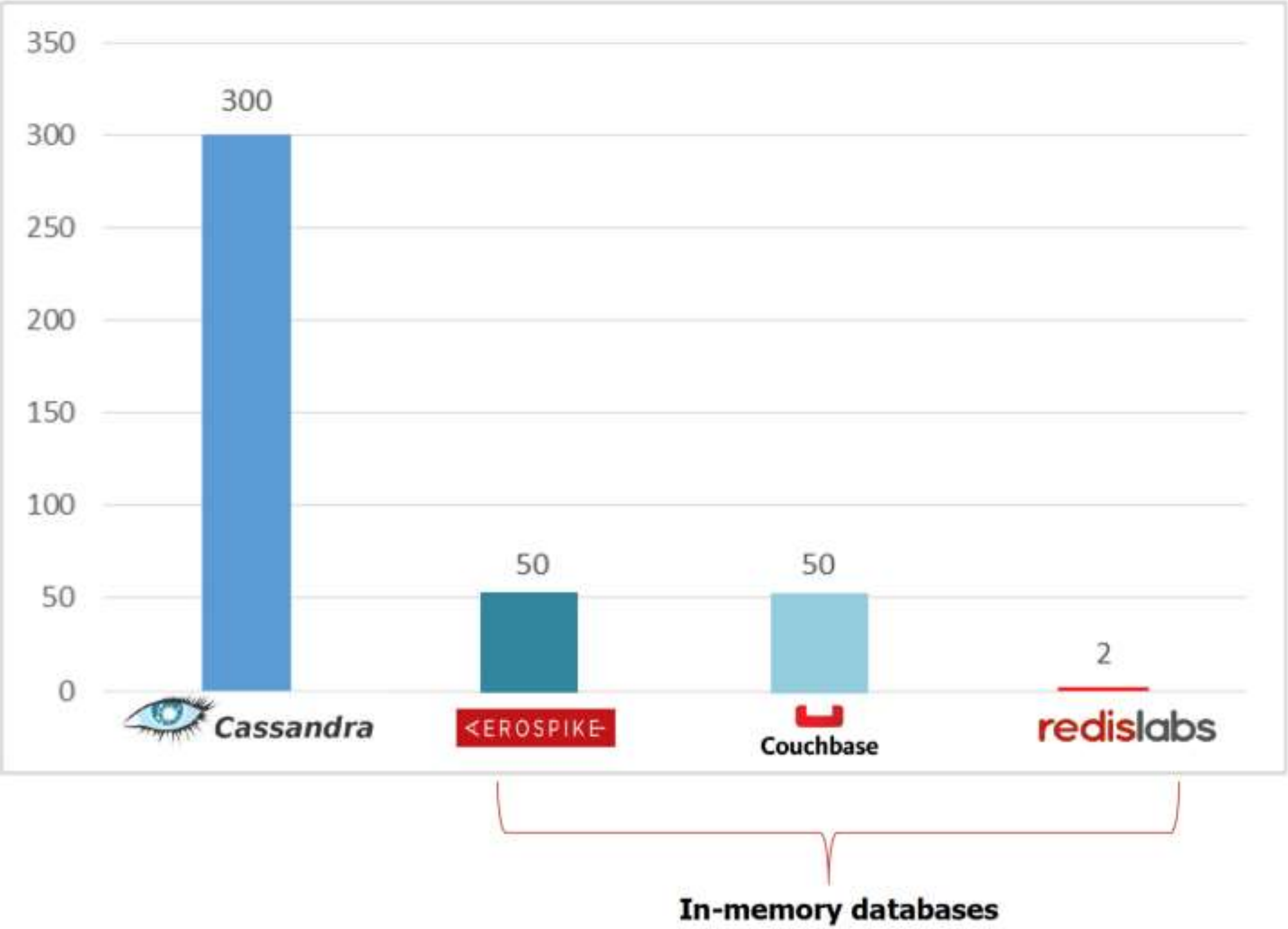
Redis: offered the cloud service over IaaS and PaaS

4 Clouds, 45 data centers across the world

10 partner channels

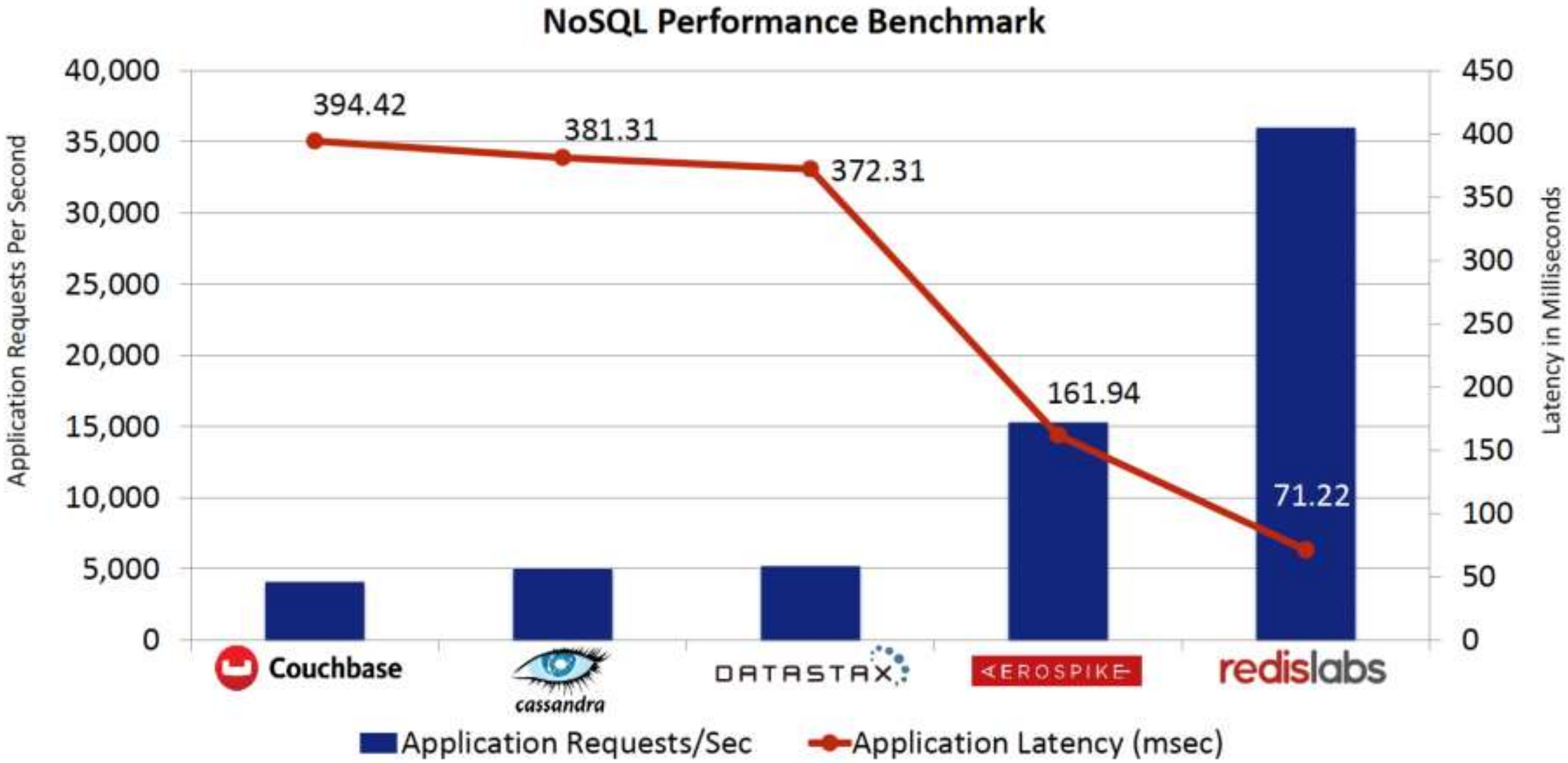
How many servers to get 1M writes/sec?

redis

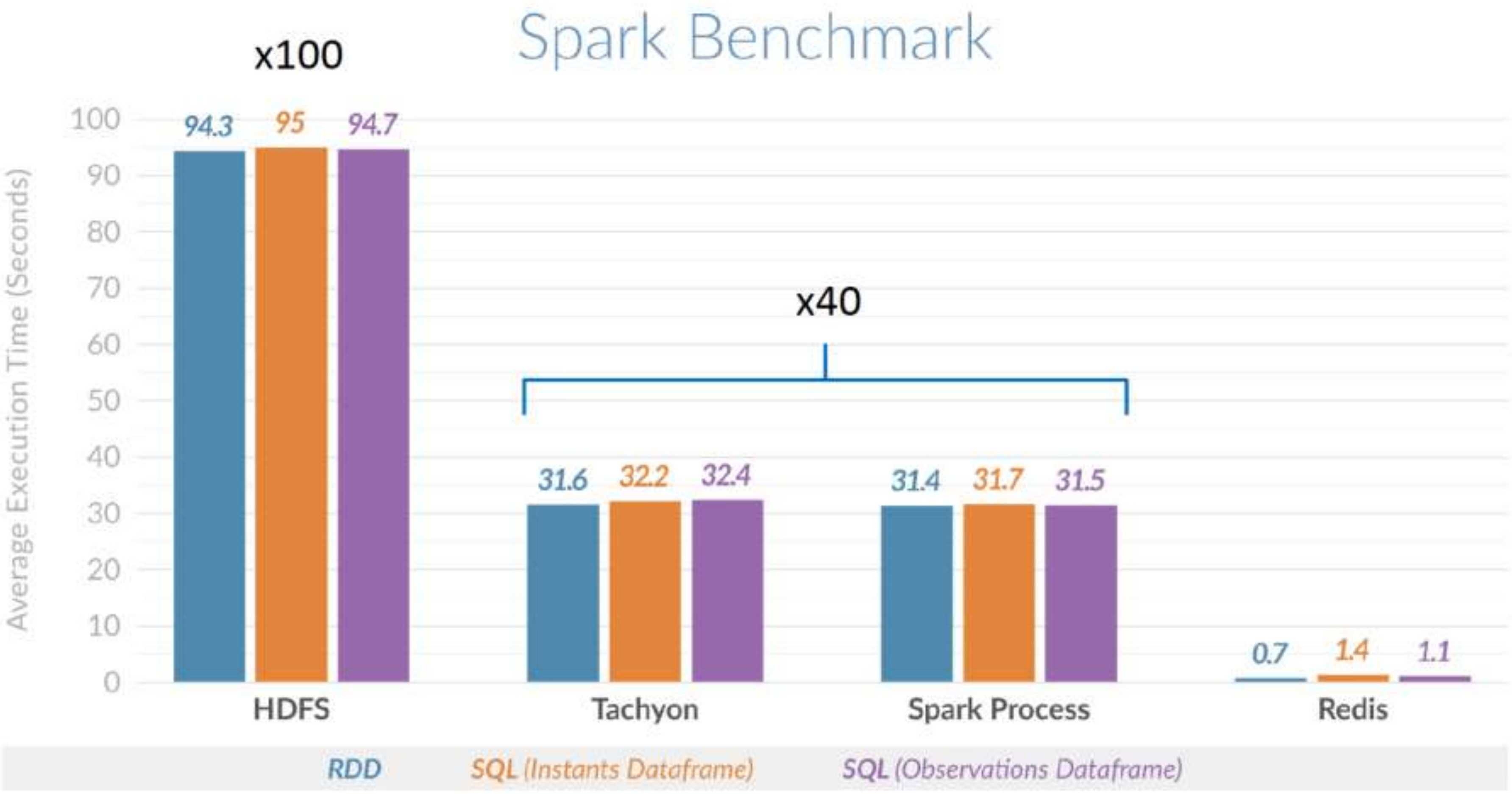


Real world write intensive app

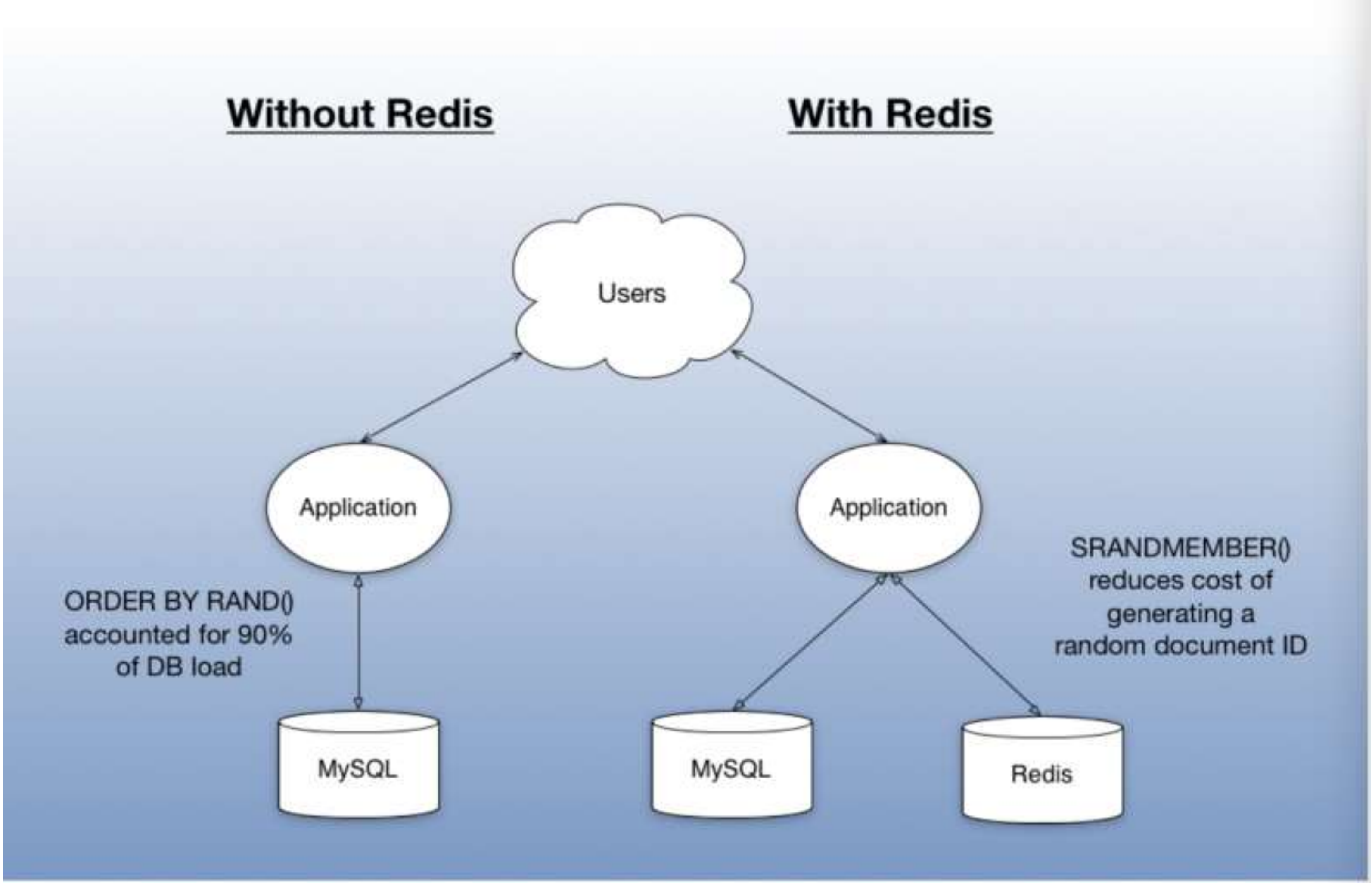
redis



Spark with Redis



How to use Redis?

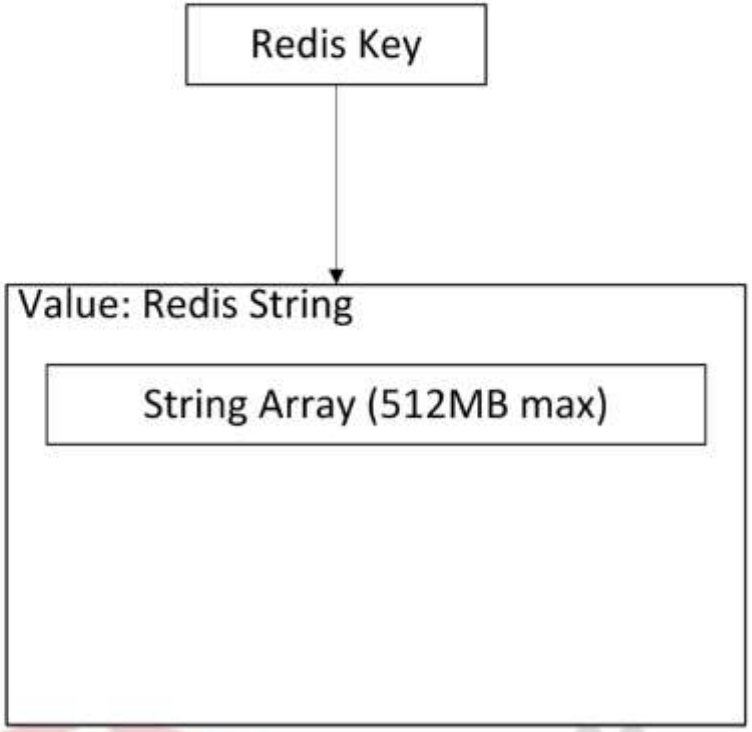


ORDER BY RAND() accounted for 90% of DB load

SRANDMEMBER() reduces cost of generating a random document ID

Logical Data Model (1)

- Data Model
- Key
  - Printable ASCII
- Value
  - Primitives
    - **Strings**
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - Sorted Sets



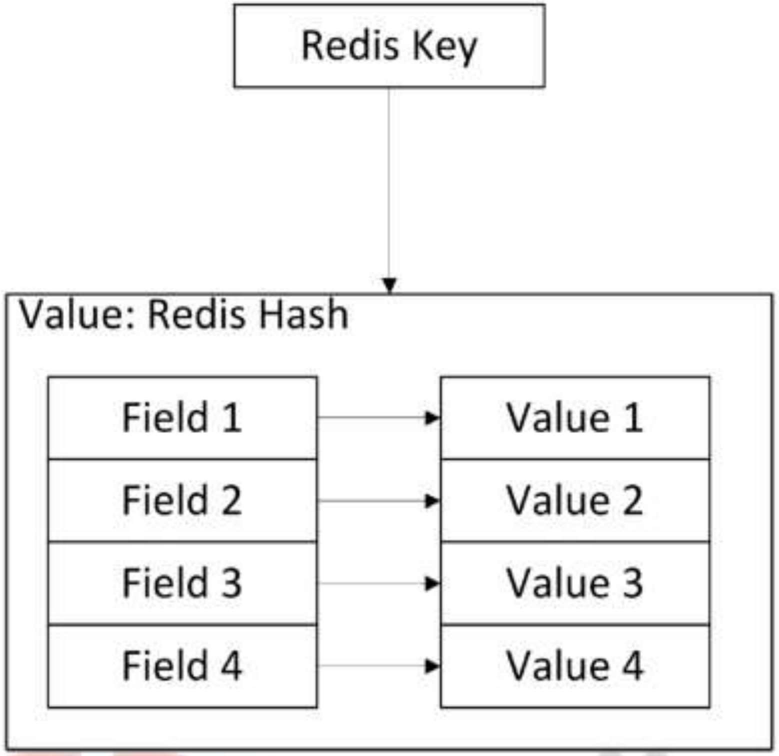
Redis Key

Value: Redis String

String Array (512MB max)

Logical Data Model (2)

- Data Model
- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - **Hashes**
    - Lists
    - Sets
    - Sorted Sets



Redis Key

Value: Redis Hash

Field 1 → Value 1

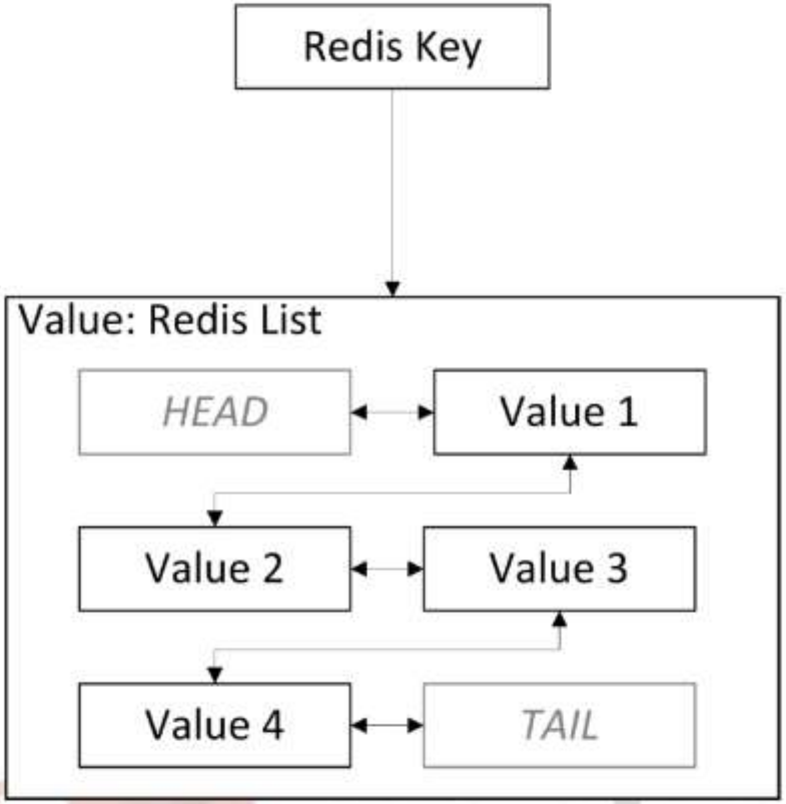
Field 2 → Value 2

Field 3 → Value 3

Field 4 → Value 4

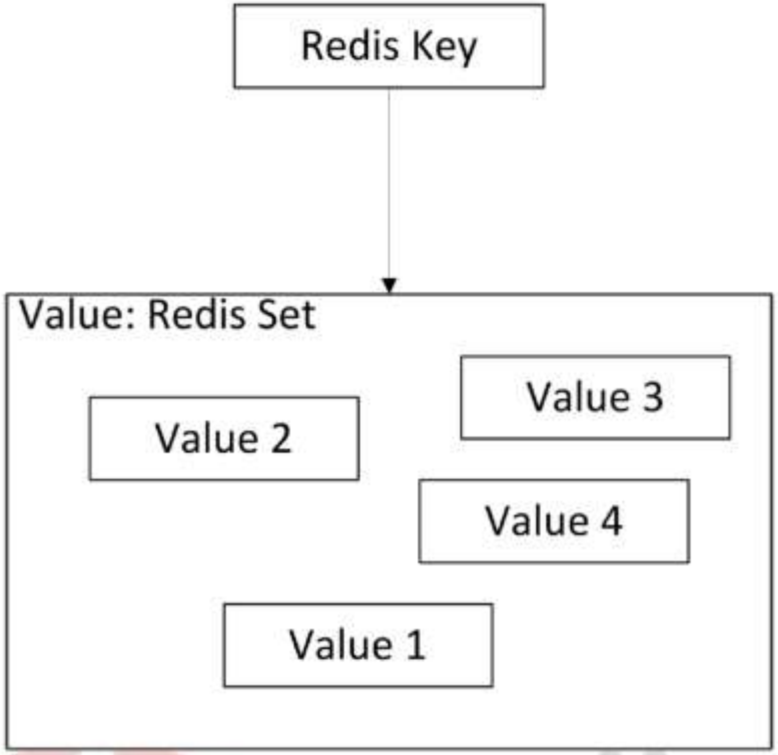
Logical Data Model (3)

- Data Model
- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - **Lists**
    - Sets
    - Sorted Sets



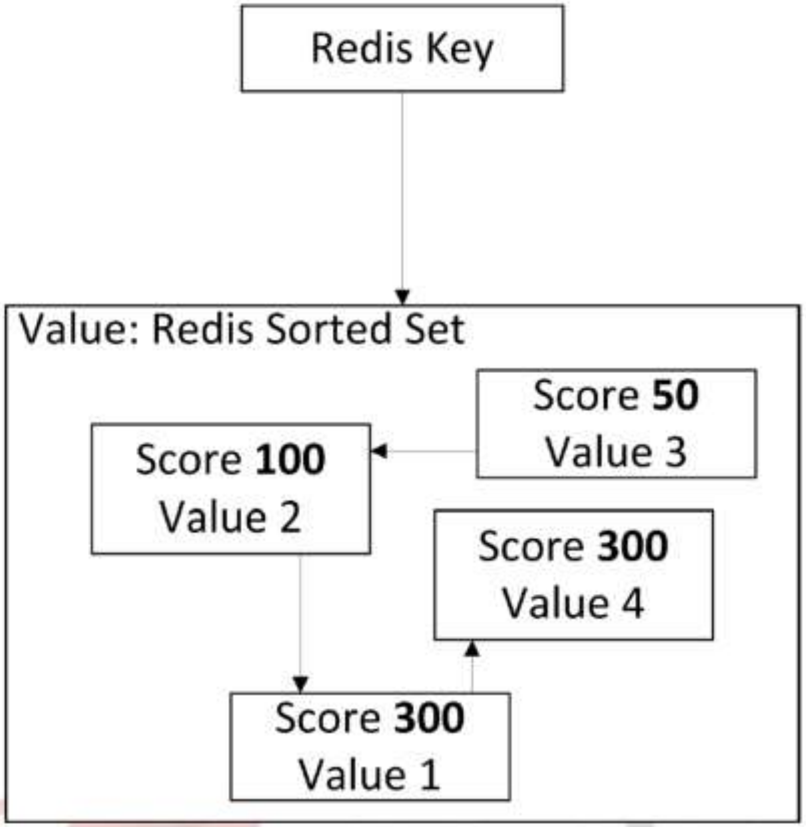
Logical Data Model (4)

- Data Model
- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - **Sets**
    - Sorted Sets



Logical Data Model (5)

- Data Model
- Key
  - Printable ASCII
- Value
  - Primitives
    - Strings
  - Containers (of strings)
    - Hashes
    - Lists
    - Sets
    - **Sorted Sets**



Redis Key

Value: Redis Sorted Set

Score **50**
Value 3

Score **100**
Value 2

Score **300**
Value 4

Score **300**
Value 1

Shopping Cart Example

redis

Relational Model

carts

CartID	User
1	james
2	chris
3	james

cart_lines

Cart	Product	Qty
1	28	1
1	372	2
2	15	1
2	160	5
2	201	7

```
UPDATE cart_lines
SET    Qty = Qty + 2
WHERE  Cart=1 AND Product=28
```

Redis Model

```
set  carts_james ( 1 3 )
set  carts_chris ( 2 )
hash cart_1 {
    user        : "james"
    product_28 : 1
    product_372: 2
}
hash cart_2 {
    user        : "chris"
    product_15 : 1
    product_160: 5
    product_201: 7
}
```

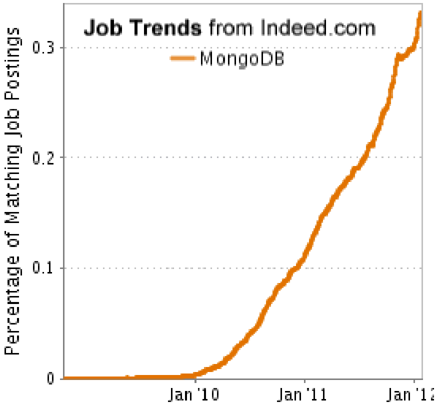
```
HINCRBY cart_1 product_28 2
```

MongoDB

- Developed by 10gen in Feb. 2009
- It is a NoSQL database
- A document-oriented database
- Open Source, Cost Effective

MongoDB

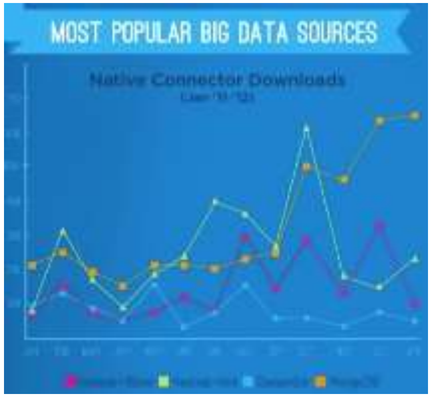
#2 ON INDEED'S FASTEST GROWING JOBS



Top Job Trends

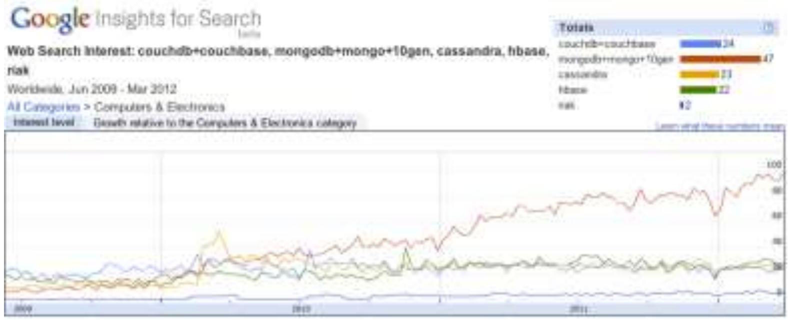
1. HTML5
2. **MongoDB**
3. iOS
4. Android
5. Mobile app
6. Puppet
7. Hadoop
8. jQuery
9. PaaS
10. Social Media

JASPERSOFT BIGDATA INDEX



Demand for MongoDB, the document-oriented NoSQL database, saw the biggest spike with over 200% growth in 2011.

GOOGLE SEARCHES



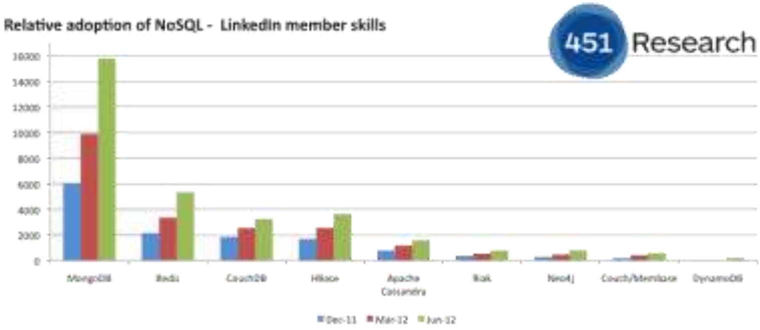
Web Search Interest: couchdb+couchbase, mongodb+mongo+10gen, cassandra, hbase, riak

Worldwide, Jun 2009 - Mar 2012

All Categories > Computers & Electronics

Growth relative to the Computers & Electronics category

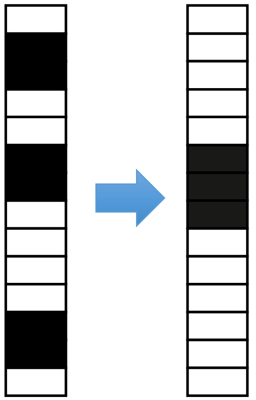
451 GROUP
"MONGODB INCREASING ITS DOMINANCE"



MongoDB is fast and scalable

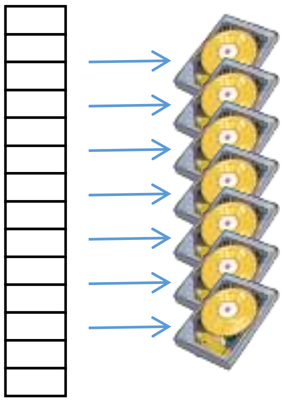
mongoDB

Better data locality

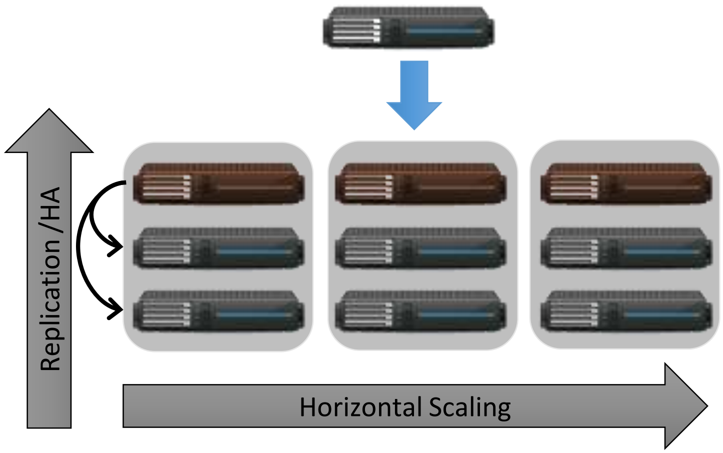


Relational MongoDB

In-Memory
Caching



Distributed Architecture



MongoDB is

mongoDB

General Purpose
- Rich data model
- Full featured indexes
- Sophisticated query language

Easy to Use
- Easy mapping to object oriented code
- Native language drivers in all popular languages
- Simple to setup and manage

Fast & Scalable
- Operates at in-memory speed wherever possible
- Auto-sharding built in
- Dynamically add / remove capacity with no downtime

Why MongoDB?

- All the modern applications deals with huge data.
- Development with ease is possible with mongoDB.
- Flexibility in deployment.
- Rich Queries.
- Older database systems may not be compatible with the design.

And it’s a document oriented storage: Data is stored in the form of JSON Style.

Why MongoDB?

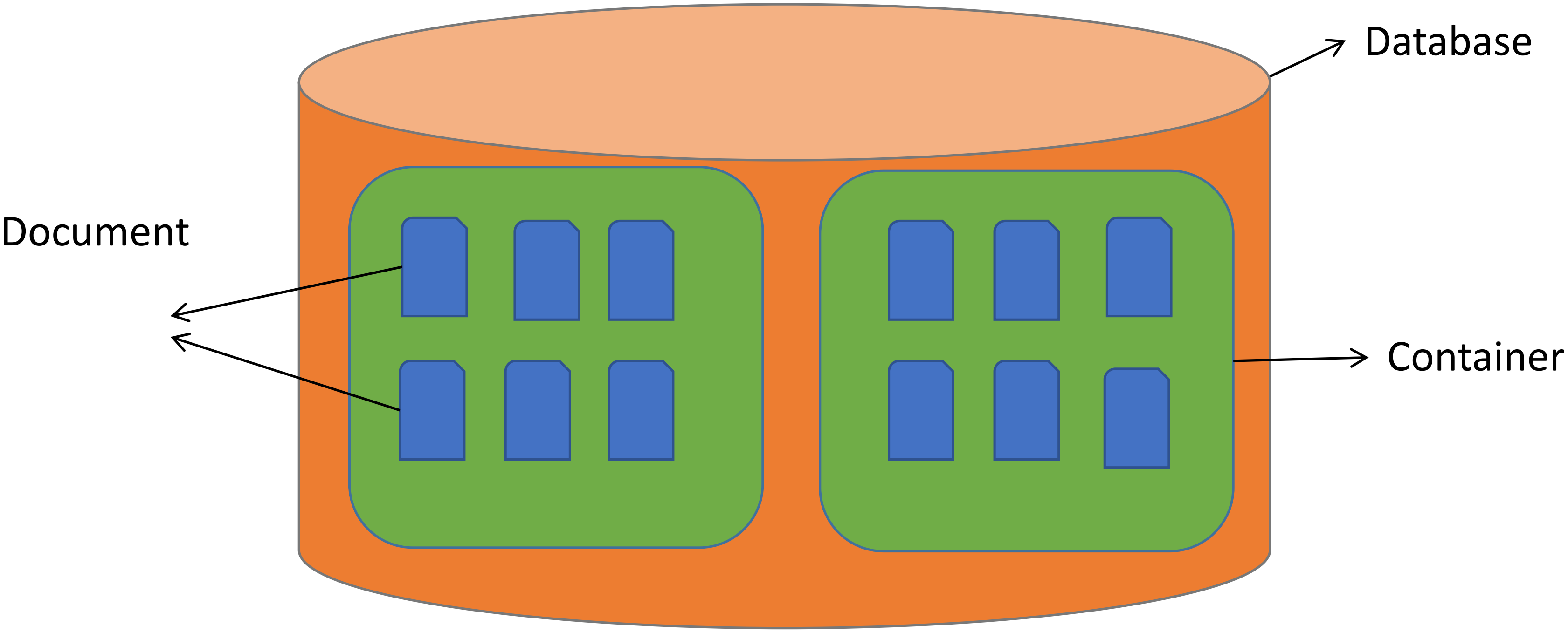
- All the modern applications deals with huge data.
- Development with ease is possible with mongoDB.
- Flexibility in deployment.
- Rich Queries.
- Older database systems may not be compatible with the design.

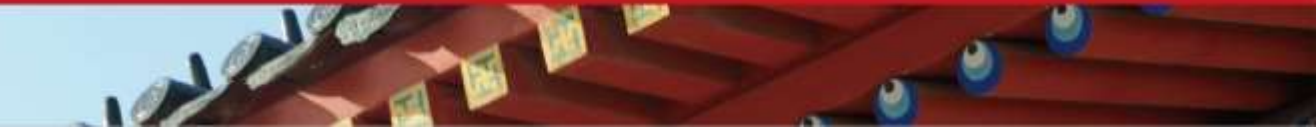
And it's a document oriented storage: Data is stored in the form of JSON Style.

MongoDB Architecture

mongoDB

Architecture :





Document (JSON) Structure

mongoDB

- The document has simple structure and very easy to understand the content
- JSON(JavaScript Object Notation) is smaller, faster and lightweight compared to XML.
- For data delivery between servers and browsers, JSON is a better choice
- Easy in parsing, processing, validating in all languages
- JSON can be mapped more easily into object oriented system.

```
• [
• {
•       "Name": "Tom",
•       "Age": 30,
•       "Role": "Student",
•       "University": "CU",
   }
  {
•       "Name": “Sam",
•       "Age": 32,
•       "Role": "Student",
•       "University": “OU",
  }
]
```

Differences between XML and JSON

XML	JSON
It is a markup language.	It is a way of representing objects.
This is more verbose than JSON.	This format uses less words.
It is used to describe the structured data.	It is used to describe unstructured data which include arrays.
JavaScript functions like eval(), parse() doesn't work here.	When eval method is applied to JSON it returns the described object.

Example:
<car> <company>Volkswagen</company>
<name>Vento</name>
<price>800000</price> </car>
	{
	"company": Volkswagen,
	"name": "Vento",
	"price": 800000
	}

Why JSON?

- JSON is faster and easier than XML when you are using it in AJAX web applications:
- **Steps involved in exchanging data from web server to browser involves:**

Using XML

1. Fetch an XML document from web server.
2. Use the XML DOM to loop through the document.
3. Extract values and store in variables.
4. It also involves type conversions.

Using JSON

1. Fetch a JSON string.
2. Parse the JSON string using eval() or parse() JavaScript functions.

The insert() Method

- To insert data into MongoDB collection, you need to use MongoDB's **insert()** or **save()** method.

- The basic syntax of **insert()** command is as follows –

“db.COLLECTION_NAME.insert(document)”

```
db.StudentRecord.insert (

{
   "Name": "Tom",
   "Age": 30,
   "Role": "Student",
   "University": "CU",
    },
  {
   "Name": “Sam",
   "Age": 22,
   "Role": "Student",
   "University": “OU",
  }


)
```

The find() Method

- To query data from MongoDB collection, you need to use MongoDB's **find()** method.

- The basic syntax of **find()** method is as follows

  “db.COLLECTION_NAME.find()”

- find() method will display all the documents in a non-structured way.

- To display the results in a formatted way, you can use **pretty()** method.

  “db.mycol.find().pretty() “

db.StudentRecord .find().pretty()

The remove() Method

- MongoDB's **remove()** method is used to remove a document from the collection. remove() method accepts two parameters. One is deletion criteria and second is justOne flag.
- **deletion criteria** − (Optional) deletion criteria according to documents will be removed.
- **justOne** − (Optional) if set to true or 1, then remove only one document.
- Syntax
- **db.COLLECTION_NAME.remove(DELLETION_CRITTERIA)**

mongoDB

Remove based on DELETION_CRITERIA

db.StudentRecord.remove({" Name": "Tom})

Remove Only One:-Removes first record

db.StudentRecord.remove(DELETION_CRITERIA,1)

Remove all Records

db.StudentRecord.remove()

MongoDB is easy to use

mongoDB

MySQL

```
START TRANSACTION;
INSERT INTO contacts VALUES
   (NULL, ‘joeblow’);
INSERT INTO contact_emails VALUES
  ( NULL, ”joe@blow.com”,
     LAST_INSERT_ID() ),
  ( NULL, “joseph@blow.com”,
     LAST_INSERT_ID() );
COMMIT;
```

MongoDB

```
db.contacts.save( {
    userName: “joeblow”,
    emailAddresses: [
      “joe@blow.com”,
      “joseph@blow.com” ] } );
```

Schema Free

- MongoDB does not need any pre-defined data schema
- Every document could have different data!

{name: “will”,
eyes: “blue”,
birthplace: “NY”,
aliases: [“bill”, “la ciacco”],
loc: [32.7, 63.4],
boss: ”ben”}

{name: “jeff”,
eyes: “blue”,
loc: [40.7, 73.4],
boss: “ben”}

{name: “ben”,
hat: ”yes”}

{name: “brendan”,
aliases: [“el diablo”]}

{name: “matt”,
pizza: “DiGiorno”,
height: 72,
loc: [44.6, 71.3]}

Thank you!