

Chapter 6: Partitioning

Distributed Data Analytics
Partitioning

Thorsten Papenbrock
G-3.1.09, Campus III
Hasso Plattner Institut



Distributing Data

Replication vs. Partitioning

Replication

- Store copies of the same data on several nodes
- Introduces **redundancy**
- Improves scalability (parallel I/O; no memory scalability!)
- Improves availability (nodes can fully take the load of failed nodes)
- Improves latency (requests can be served by the closest node)

Partitioning

our focus now

- Store the data split in subsets (partitions) on several nodes
- Also known as **sharding**
- Improves scalability (some parallel I/O; **memory consumption**)
- Improves availability (node failures take out only parts of the data)
- Improves latency (place partitions close to where they are accessed most)

Distributed Data Analytics

Partitioning

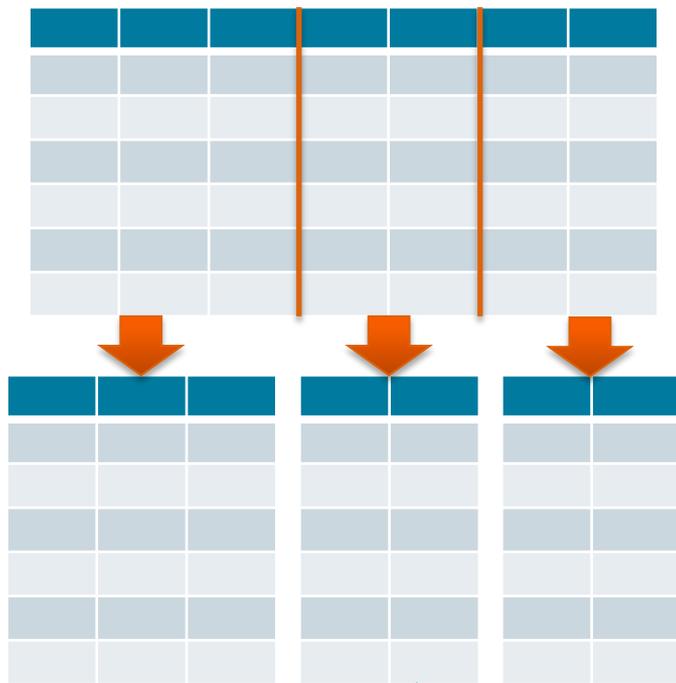
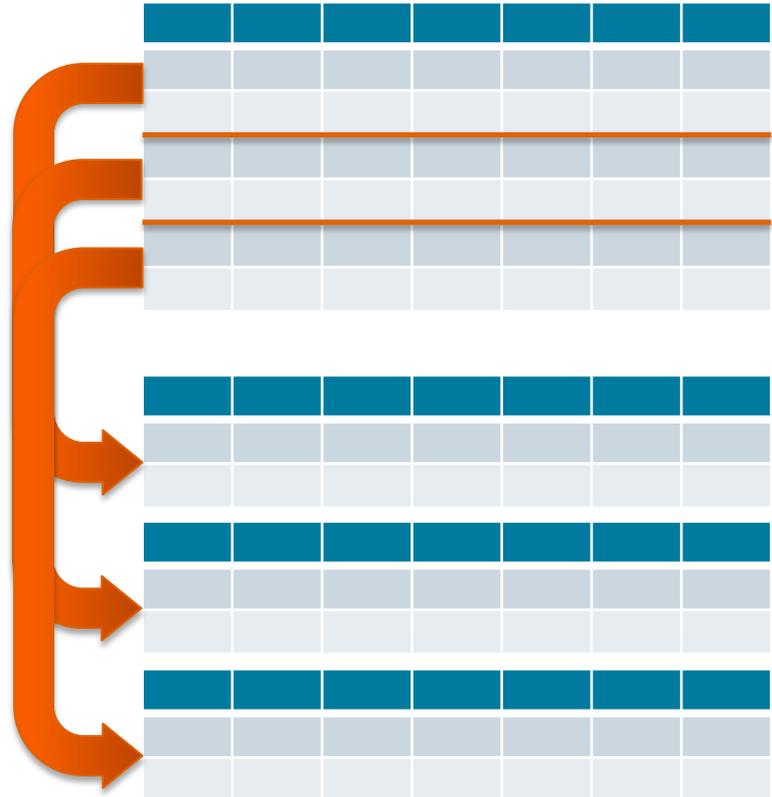
ThorstenPapenbrock
Slide 2

➤ Different mechanisms but usually used together

Distributing Data

Horizontal vs. Vertical Partitioning

Different dimensions but essentially the same partitioning strategies



Used in most distributed systems

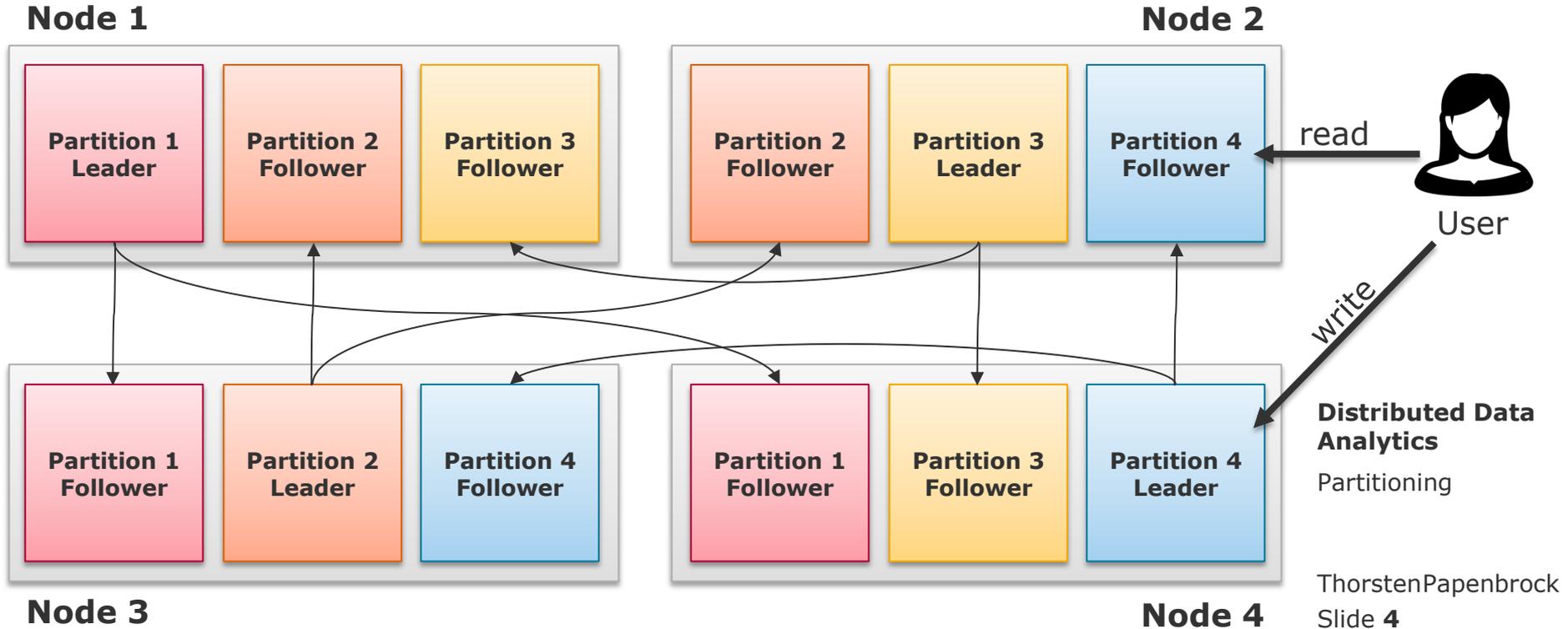
Used in column-based DBMSs

Distributed Data Analytics
Partitioning

ThorstenPapenbrock
Slide 3

Distributing Data

Replication and Partitioning



Synonymes

- **shard** (MongoDB, Elasticsearch, SolrCloud)
- **region** (HBase)
- **tablet** (Bigtable)
- **vnode** (Cassandra, Riak)
- **vBucket** (Couchbase)

Partitioning Algorithm

- **Each data item (record, row, document, ...) belongs to exactly one partition** (considering replicated partitions as same partitions)
- Algorithm task:
 - Given any data item, assign it to a partition
 - Keep partitions (possibly) balanced

Overview

Partitioning

Partitioning of Key-Value Data

- Partitioning by Key Range
- Partitioning by Hash of Key

Partitioning and Secondary Indexes

- Partitioning Secondary Indexes by Document
- Partitioning Secondary Indexes by Term

Rebalancing Partitions

- Strategies for Rebalancing
- Operations: Automatic or Manual Rebalancing

Request Routing

- Parallel Query Execution



Distributed Data Analytics
Partitioning

Overview

Partitioning

Partitioning of Key-Value Data

- Partitioning by Key Range
- Partitioning by Hash of Key

Partitioning and Secondary Indexes

- Partitioning Secondary Indexes by Document
- Partitioning Secondary Indexes by Term

Rebalancing Partitions

- Strategies for Rebalancing
- Operations: Automatic or Manual Rebalancing

Request Routing

- Parallel Query Execution



Distributed Data Analytics

Partitioning

Key-Value Data

- All data models:
 - relational (ID → record)
 - key-value (key → value)
 - column-family (row key → super column)
 - document (key → document)
 - graph (key → node/edge)

Different dimensions but similar techniques

Dimension

- **Horizontal partitioning**: distribution of rows, records, key-value pairs, ...
- **Vertical partitioning**: distribution of columns, super columns, value groups, ...

Distributed Data Analytics

Partitioning

Skew

- **Skew**: Situation where some partitions have more data/queries than others
- **Hot spots**: Partitions with disproportionately high load

ThorstenPapenbrock
Slide 8

Partitioning of Key-Value Data

Partitioning by Key Range

Range Partitioning

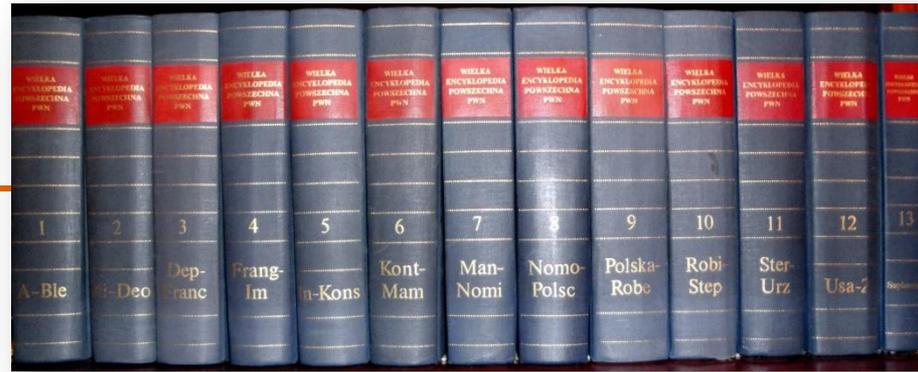
- Arrange keys in a continuous, sorted range
- Split this range into **partitions**:
 - also continuous and sorted
 - identified by min and max key value
 - not evenly spaced if key range is skewed:
 - e.g. as many words in [A,Ble] as in [Usa,Z]
 - implemented as (for instance) SSTables and LSM-Trees

Partition lookup for (new or existing) key

- Find partition where $min \leq key < max$ (**binary search**)

Properties

- Strength: **range queries**
- Weakness: **load skew** if certain key ranges are accessed more frequently than others



e.g. if a **timestamp** is the key, all inserts (and most reads) go to the partition with the newest entries

Partitioning of Key-Value Data

Partitioning by Hash of Key

Hash Partitioning

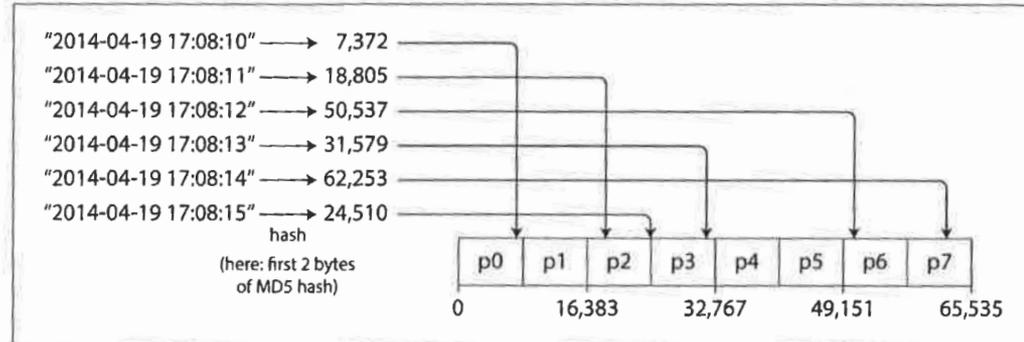
- Map the (skewed) range of keys to a uniformly distributed range of hashes
- Use equidistant range partitioning on the range of hashes
- **Hash function:**
 - calculates the key-to-hashes mapping (one-way-function)
 - skewed input, uniform output
 - e.g. MD5: a 128-bit hash function that maps arbitrary strings to numbers between 0 and $2^{128} - 1$

Why not simply $\text{hash} \% n$?

➤ Later!

Partition lookup for (new or existing) key

- Calculate hash
- Find partition where $\min \leq \text{key} < \max$ (**binary search**)

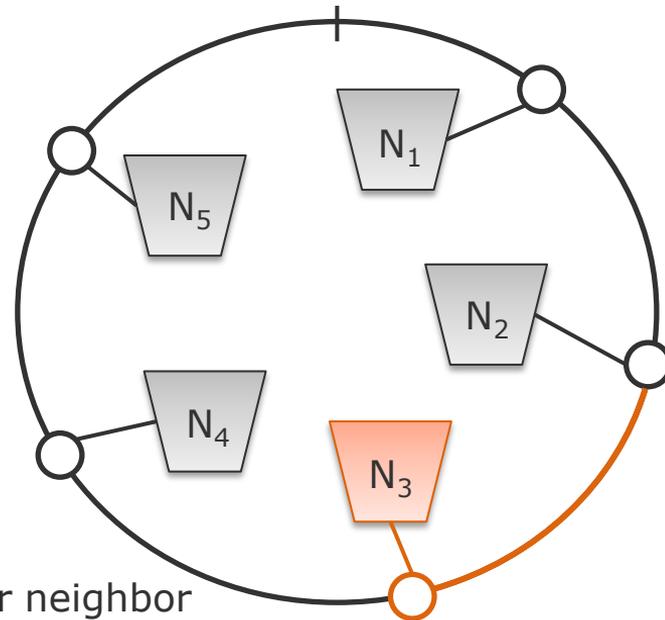


Partitioning of Key-Value Data

Partitioning by Hash of Key

Hash Partitioning

- Also known as **Consistent Hashing**
 - Range of keys is modeled as a ring
 - Nodes are hashed to positions on the ring
 - Each node N_i is responsible for all hashes k between its position i and the position j of its clockwise predecessor N_j with $j < k \leq i$
- If a node enters, it “steals” values from one node
- If a node leaves, it “leaves” all its values to its higher neighbor
 - Most assigned values stay untouched!



Partitioning by Hash of Key

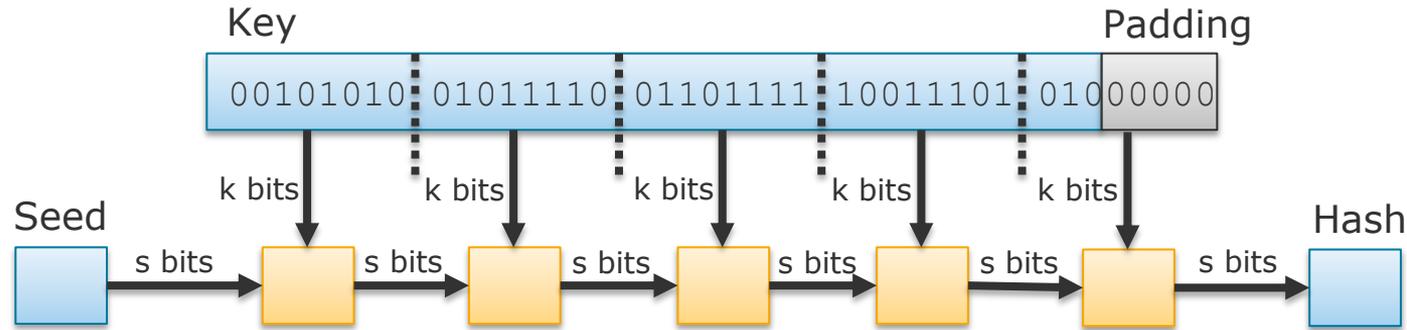
Hashing

- Use cases:

- Cryptography
- Checksums
- Partitioning

- Algorithm: (MD4, MD5, SHA-1, SHA-2, ...)

- Interpret key as bit-sequence
- Divide key into blocks of equal size k (e.g. $k = 64 * 8$ bit)
- Pad last block if it is too short
- For each block:
 - Combine the k block-bits with the s buffer bits (e.g. $s = 128$ bit) (first block starts with a standard seed sequence)
 - Combine algorithm uses some hashing-specific combination of bit-operations (AND, OR, bit-shifts, XOR, NOT, ...)



Merkle-Damgård construction:
A generic method to hash arbitrary-length inputs to fixed-length hashes

Partitioning of Key-Value Data

Partitioning by Hash of Key

Hashing

- Use cases:

- Cryptography
- Checksums
- Partitioning

- Algorithm: (MD4, MD5, SHA-1, SHA-2, ...)

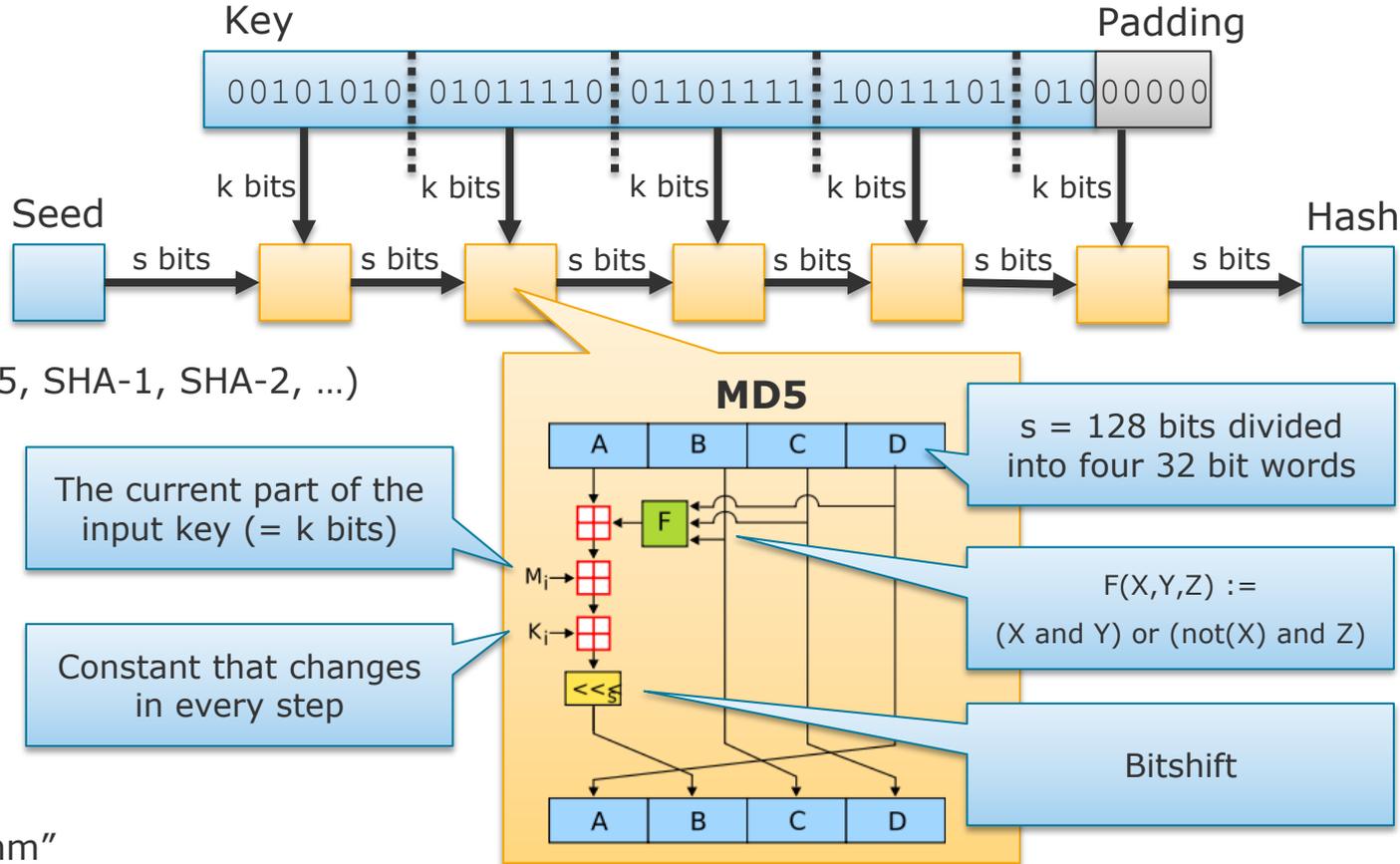
Weak for encryption, but good for partitioning

The current part of the input key (= k bits)

Constant that changes in every step

- MD5: "Message Digest 5"

- SHA-2: "Secure Hash Algorithm"



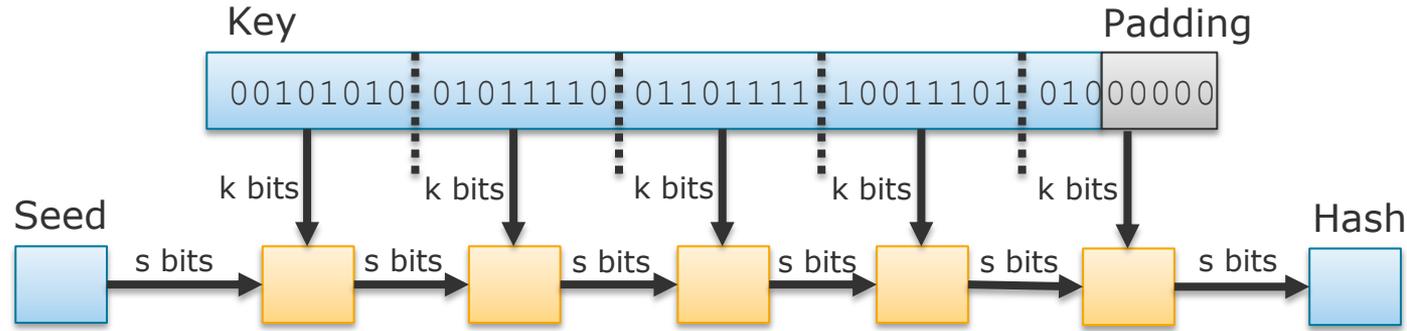
Partitioning of Key-Value Data

Partitioning by Hash of Key

Hashing

- Use cases:
 - Cryptography
 - Checksums
 - Partitioning

Example:



Input	Digest
Fox	DFCD 3454 BBEA 788A 751A 696C 24D9 7009 CA99 2D17
The red fox jumps over the blue dog	0086 46BB FB7D CBE2 823C ACC7 6CD1 90B1 EE6E 3ABC
The red fox jumps over the blue dog	8FD8 7558 7851 4F32 D1C6 76B1 79A9 0DA4 AEF6 4819
The red fox jumps over the blue dog	FCD3 7FDB 5AF2 C6FF 915F D401 C0A9 7D9A 46AF FB45
The red fox jumps over the blue dog	8ACA D682 D588 4C75 4BF4 1799 7D88 BCF8 92B9 6A6C

Digest (= Hash) with fix length independent of input length

Partitioning of Key-Value Data

- Partitioning by Key Range
- Partitioning by Hash of Key

Partitioning and Secondary Indexes

- **Partitioning Secondary Indexes by Document**
- **Partitioning Secondary Indexes by Term**

Rebalancing Partitions

- Strategies for Rebalancing
- Operations: Automatic or Manual Rebalancing

Request Routing

- Parallel Query Execution



Distributed Data Analytics
Partitioning

Secondary Index

- Any index (in addition to the primary key index) that ...
 - may not identify all records uniquely
 - cannot be implemented as a clustered index (sorting/grouping not possible)
- Used to ...
 - search for items with a certain value/property
 - accelerate frequent/complex queries
- **Does not map neatly to partitions and is larger than a clustered index**
 - Must be partitioned as well

Example: Indexes on color and maker of cars

```
CREATE INDEX idx_color_filter ON Cars (color);
```

```
CREATE INDEX idx_make_filter ON Cars (make);
```

**Distributed Data
Analytics**

Partitioning

ThorstenPapenbrock
Slide **16**

Partitioning and Secondary Indexes

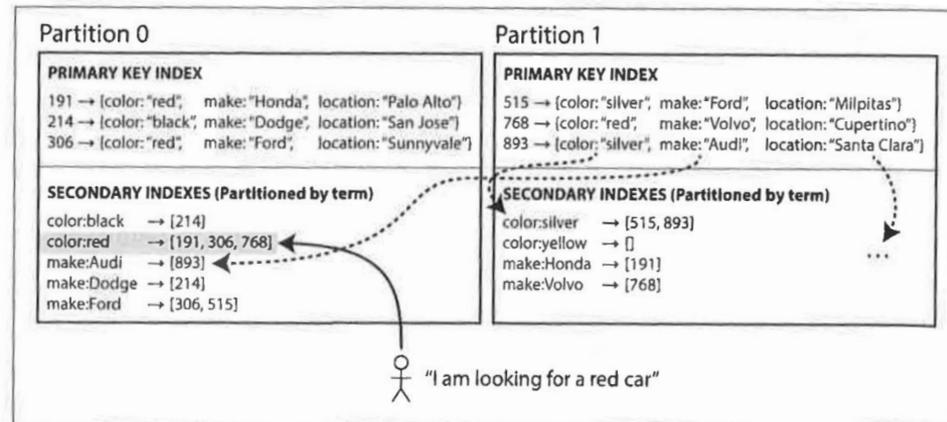
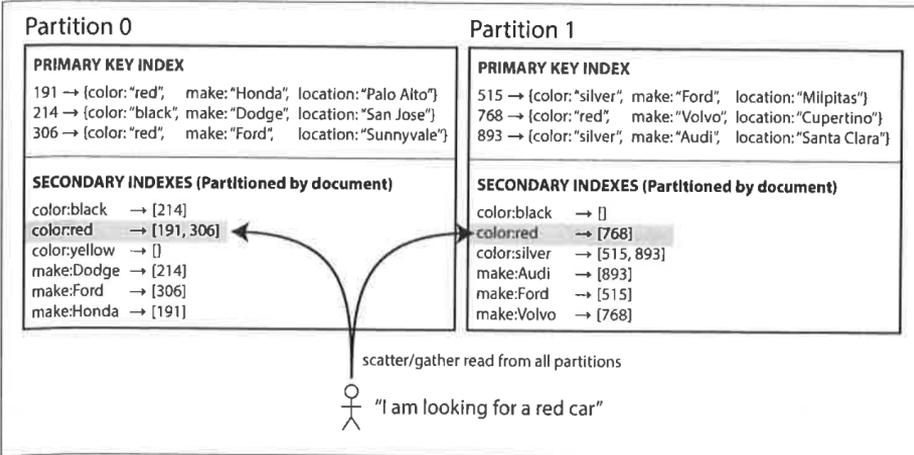
Partitioning Secondary Indexes ...

by Document: **Local Index**

- Every partition manages its own index with all pointers to local data items
 - Vertically partitioned index
- Insert/update/delete: performed locally
- Select: queries all partition indexes

by Term: **Global Index**

- Index entries are partitioned by their key independently from local data items
 - Horizontally partitioned index
- Insert/update/delete: require remote updates
- Select: queries only one partition index



by Document: **Local Index**

- Every partition manages its own index with all pointers to local data items
 - Vertically partitioned index
- **Insert/update/delete: performed locally**
- **Select: queries all partition indexes**



MongoDB

Riak

Cassandra

Elasticsearch

SolrCloud

VoltDB

OLTP view:

Secondary indexes must not cost much even if this makes them less effective.

OLAP view:

Write costs can be expensive because they are one-time efforts but reads must be efficient.

by Term: **Global Index**

- Index entries are partitioned by their key independently from local data items
 - Horizontally partitioned index
- **Insert/update/delete: require remote updates**
- **Select: queries only one partition index**



DynamoDB

Riak

Oracle Data Warehouse

Overview

Partitioning

Partitioning of Key-Value Data

- Partitioning by Key Range
- Partitioning by Hash of Key

Partitioning and Secondary Indexes

- Partitioning Secondary Indexes by Document
- Partitioning Secondary Indexes by Term

Rebalancing Partitions

- **Strategies for Rebalancing**
- **Operations: Automatic or Manual Rebalancing**

Request Routing

- Parallel Query Execution



Distributed Data Analytics

Partitioning

Rebalancing Partitions

Rebalancing

Things change:

- Query load → add more CPUs
- Data size → add more disks and RAM
- Nodes fail → other nodes need to take over
 - Require to move data around (**rebalancing**)!

Rebalancing requirements

- **Balanced result**: even data distribution after rebalancing
- **Downtime-less**: continue accepting reads/writes during rebalancing
- **Minimal data shift**: move no more data than necessary between nodes

How not to do it: $\text{hash} \% n$

- $\text{Hash} \% n$ results in numbers between 0 and $n-1$ to assign nodes with
- **BUT: if n changes, most hashes yield new node numbers, i.e., need to move!**
 - **Example:** $123456 \% 10 = 6$, $123456 \% 11 = 3$, $123456 \% 12 = 0$, ...

$\text{hash} \% n$ is still useful for e.g. **load balancing** or **fixed sized partitionings**, because lookup is in $O(1)$

Distributed Data Analytics

Partitioning

ThorstenPapenbrock

Slide 20

Rebalancing Partitions

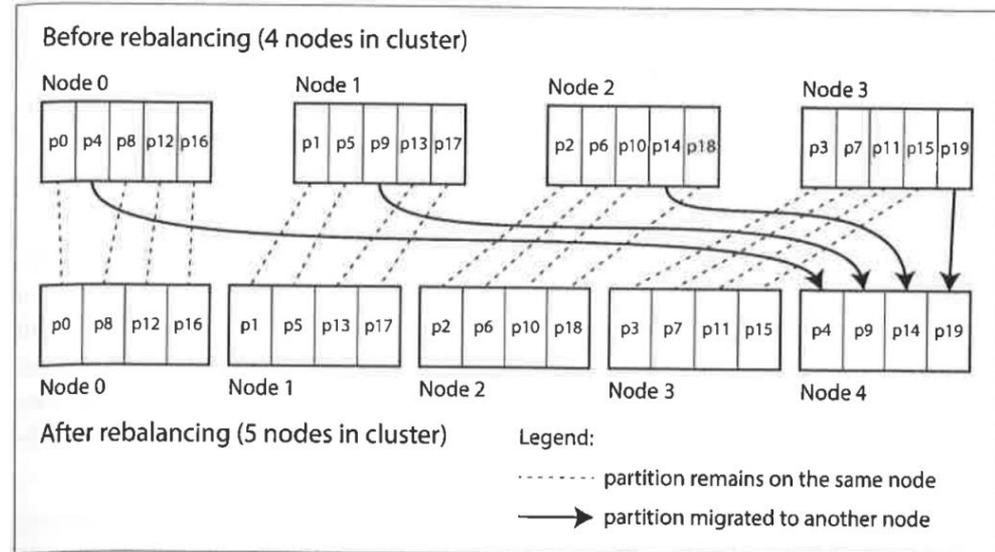
Fixed Number of Partitions

Idea

- Create many more **partitions p** than there are **nodes n**, i.e., several partitions per node
- Let new nodes “steal” partitions from all other nodes until distribution is even again
 - Key → partition mappings stay fix

Wait!

- We only moved the problem:
 - Partition → node mappings change!
- But: Partition → node mapping is ...
 - much smaller (say 1000 partitions)
 - usually fix in size (= #partitions)
 - Only a (partial) rewrite of a small data structure



Rebalancing Partitions

Fixed Number of Partitions

Idea

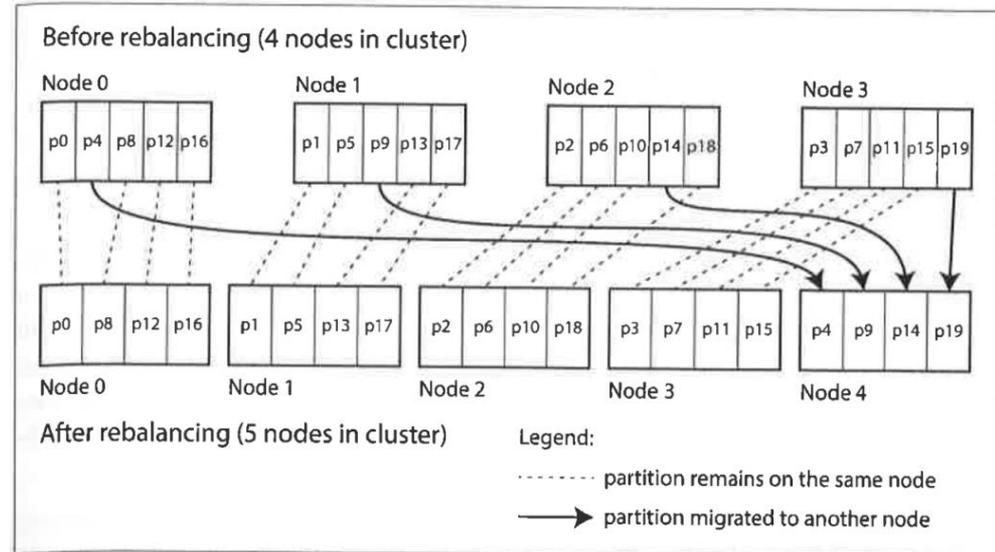
- Create many more **partitions p** than there are **nodes n**, i.e., several partitions per node
- Let new nodes “steal” partitions from all other nodes until distribution is even again
 - Key → partition mappings stay fix

Choosing p is difficult

- If p is too large (partitions small):
 - Expensive partition management
- If p is too small (partitions large):
 - Expensive rebalancing and recovery

Implementations

- Riak, Elasticsearch, Couchbase, Voldemort



Rebalancing Partitions

Dynamic Partitioning

Idea

- Create some initial number of partitions (e.g. $p = n$ for p partitions and n nodes)
- If a partition exceeds some **max size** threshold, **split** it.
- If a partition falls below some **min size** threshold, **merge** it.
 - Number of partitions proportional to dataset size

Works well for any partitioning that splits ranges (of keys or hashes).

Partition to node assignment

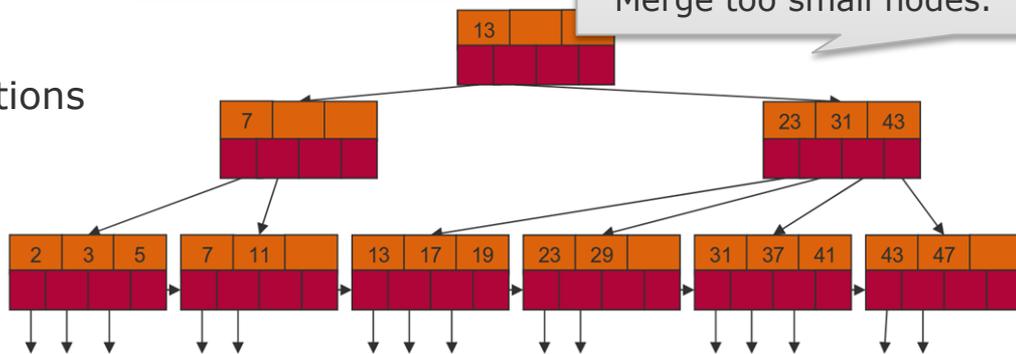
- Distribute partitions evenly between all nodes
- If new nodes enter, let them steal
 - Same as for fixed number of partitions

partition → node mappings that change!

Idea similar to **B-Trees**:
Split too large nodes.
Merge too small nodes.

Implementations

- Hbase, RethinkDB, MongoDB



Idea

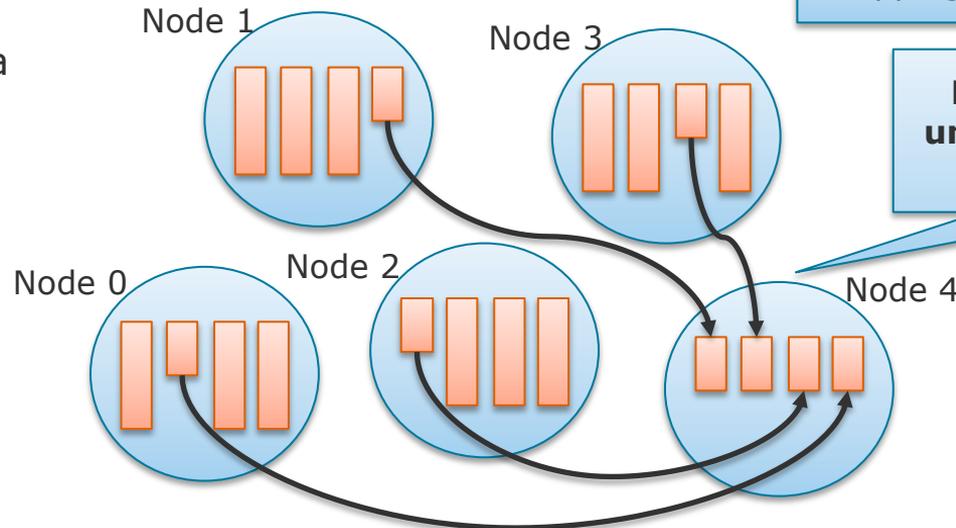
- Create a fixed number of **p partitions** on each of the **n nodes**
- Let new nodes fill their own p partitions by randomly splitting partitions on other nodes
 - Steal half of p partitions from other nodes

Works well for any partitioning that splits ranges (of keys or hashes).

partition → node mappings that change!

Implementations

- Cassandra, Ketma



Load may occasionally be **unbalanced** but is expected to even out over time

Overview

Partitioning

Partitioning of Key-Value Data

- Partitioning by Key Range
- Partitioning by Hash of Key

Partitioning and Secondary Indexes

- Partitioning Secondary Indexes by Document
- Partitioning Secondary Indexes by Term

Rebalancing Partitions

- Strategies for Rebalancing
- Operations: Automatic or Manual Rebalancing

Request Routing

- **Parallel Query Execution**



Distributed Data Analytics
Partitioning

Request Routing Partition Lookup

Partitions move between nodes regardless of the rebalancing strategy

Ask any node

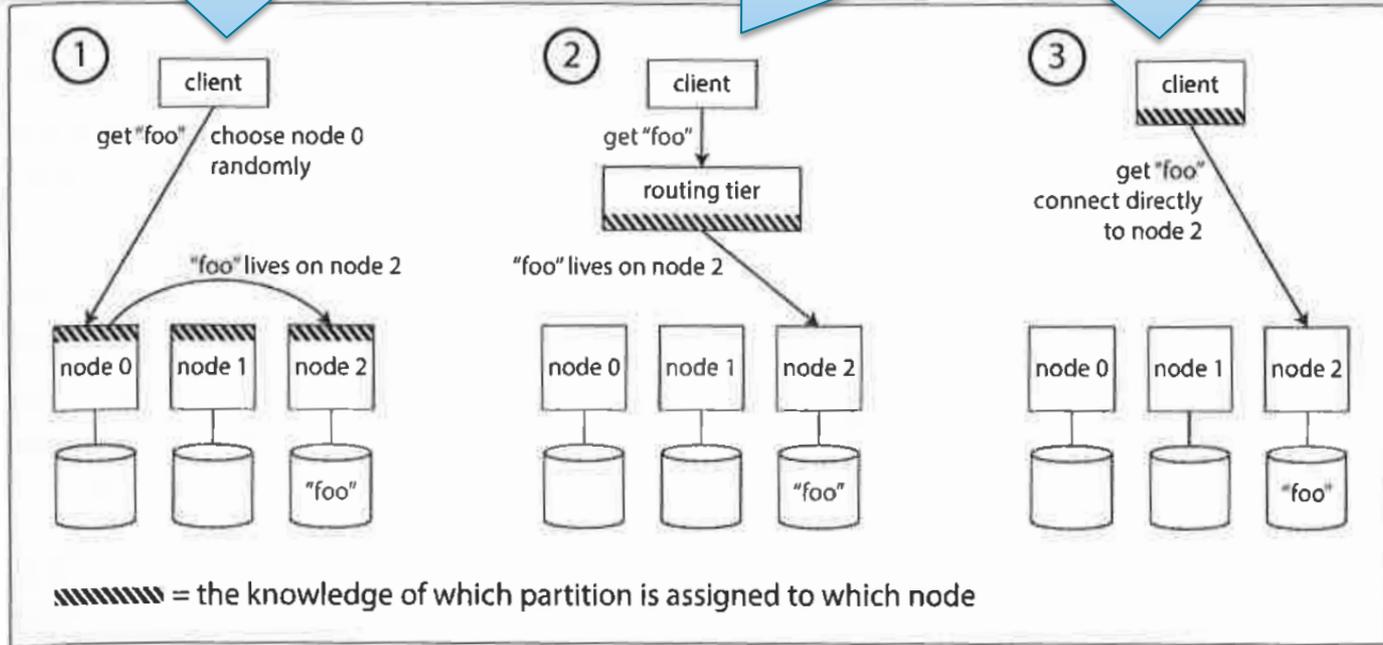
all nodes store a lookup table copy and can redirect queries

Ask a router

a routing tier stores the lookup table copy and can redirect queries

Ask node directly

all clients store a lookup table copy and can locate partitions



Request Routing Partition Lookup

Partitions move between nodes regardless of the rebalancing strategy

Ask any node

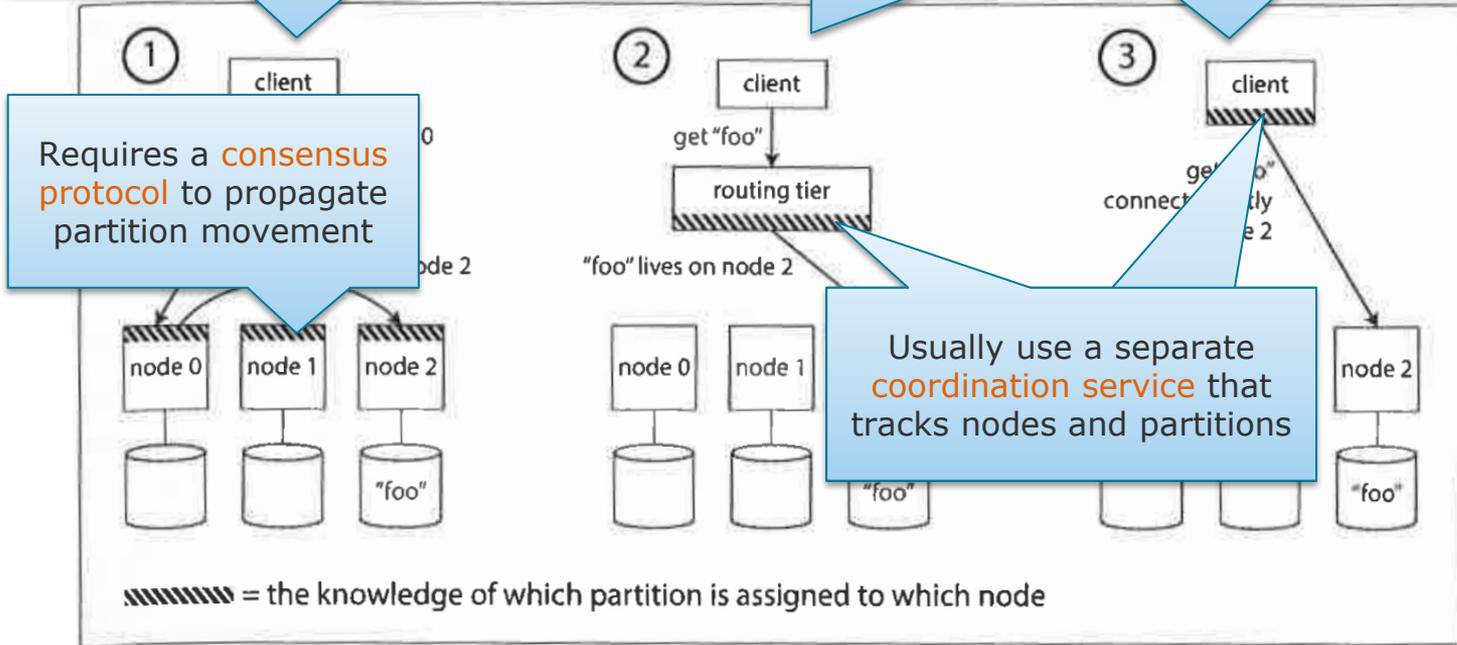
all nodes store a lookup table copy and can redirect queries

Ask a router

a routing tier stores the lookup table copy and can redirect queries

Ask node directly

all clients store a lookup table copy and can locate partitions



Request Routing

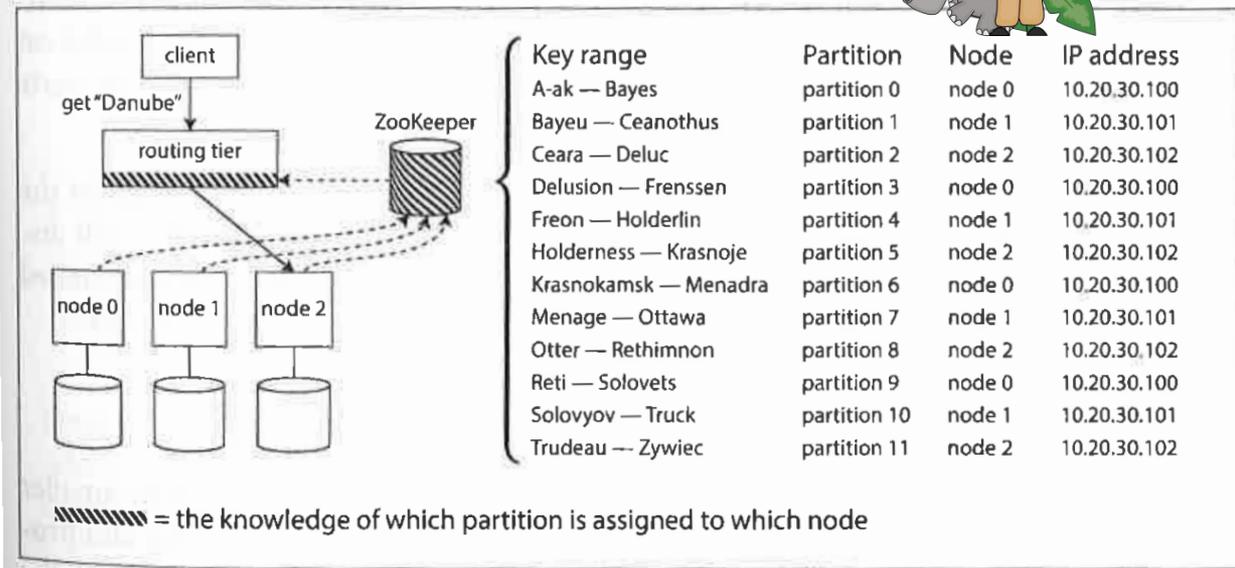
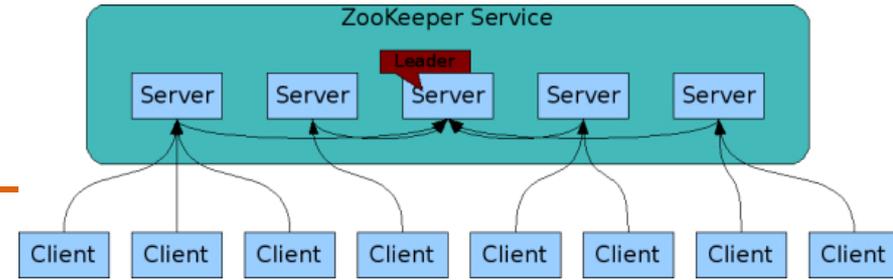
Partition Lookup: ZooKeeper

Apache ZooKeeper

- A **coordination service** for services in distributed systems
- Tracks and offers cluster metadata:
 - naming, localization, configuration, and synchronization of services
- Itself implemented as a distributed key-value store
- Leader-follower replication

Subscriber Model:

- Each client maintains a TCP connection
- Nodes send: heart beats and partition updates
- Router/client gets: partition addresses upon request



Request Routing

Partition Lookup: ZooKeeper

ZooKeeper users:

- Espresso, HBase, SolrCloud, Kafka, OpenStack Nova, Hadoop YARN ...

Many further **SOA and Cloud systems** that are no DBMSs!

More details on these features in the following session!

Features:

- **Service discovery** (e.g. find IP and port for a specific service)
- **Linearizable atomic operations** (e.g. atomic compare-and-set for implementing locks/leases)
- **Total ordering of operations** (e.g. generating monotonically increasing IDs for transactions)
- **Failure detection** (e.g. heartbeat failure detection to initiate leader elections)
- **Change notification** (e.g. notify clients about new/failed clients in the cluster)
- **Automatic cluster management** (e.g. leader election, partition re-balancing, ...)

