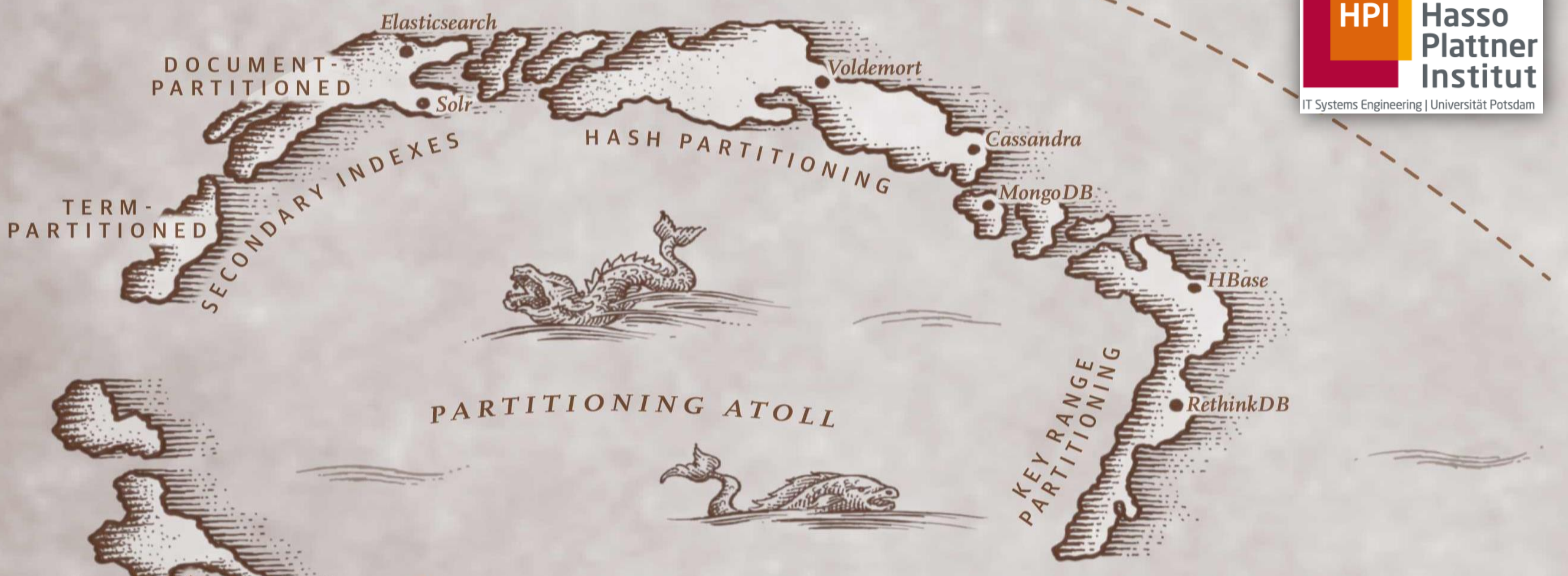


Distributed Data Management
Partitioning

Thorsten Papenbrock

F-2.04, Campus II

Hasso Plattner Institut

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/underutilised node)

Partitioning

our focus now

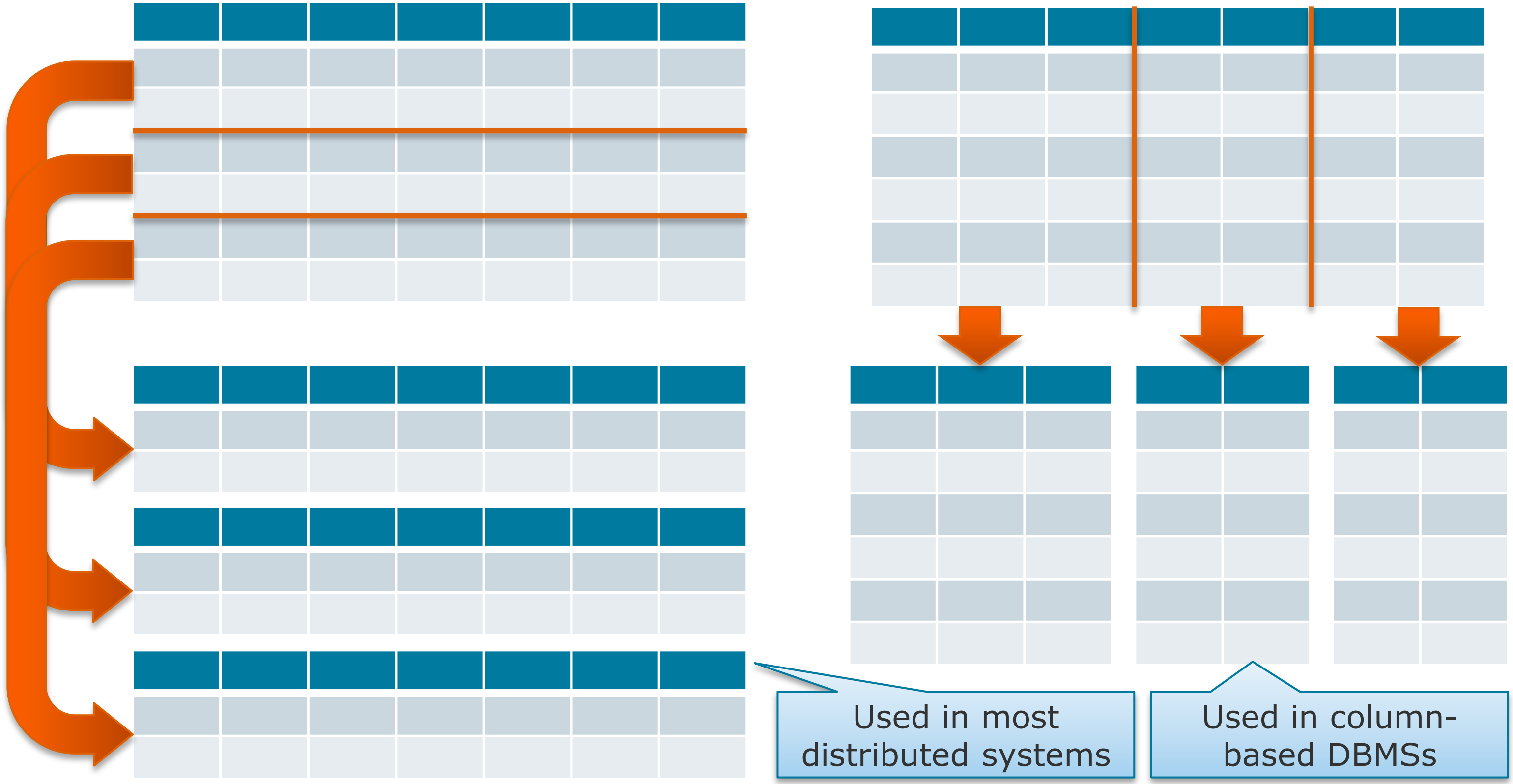
- Store the data split in subsets (partitions) on several nodes
- Also known as sharding
- Improves scalability (some parallel I/O; **memory consumption**)
- Improves availability (node failures take out only parts of the data)
- Improves latency (place partitions close to where they are accessed most)

Different mechanisms but usually used together

Distributing Data

Horizontal vs. Vertical Partitioning

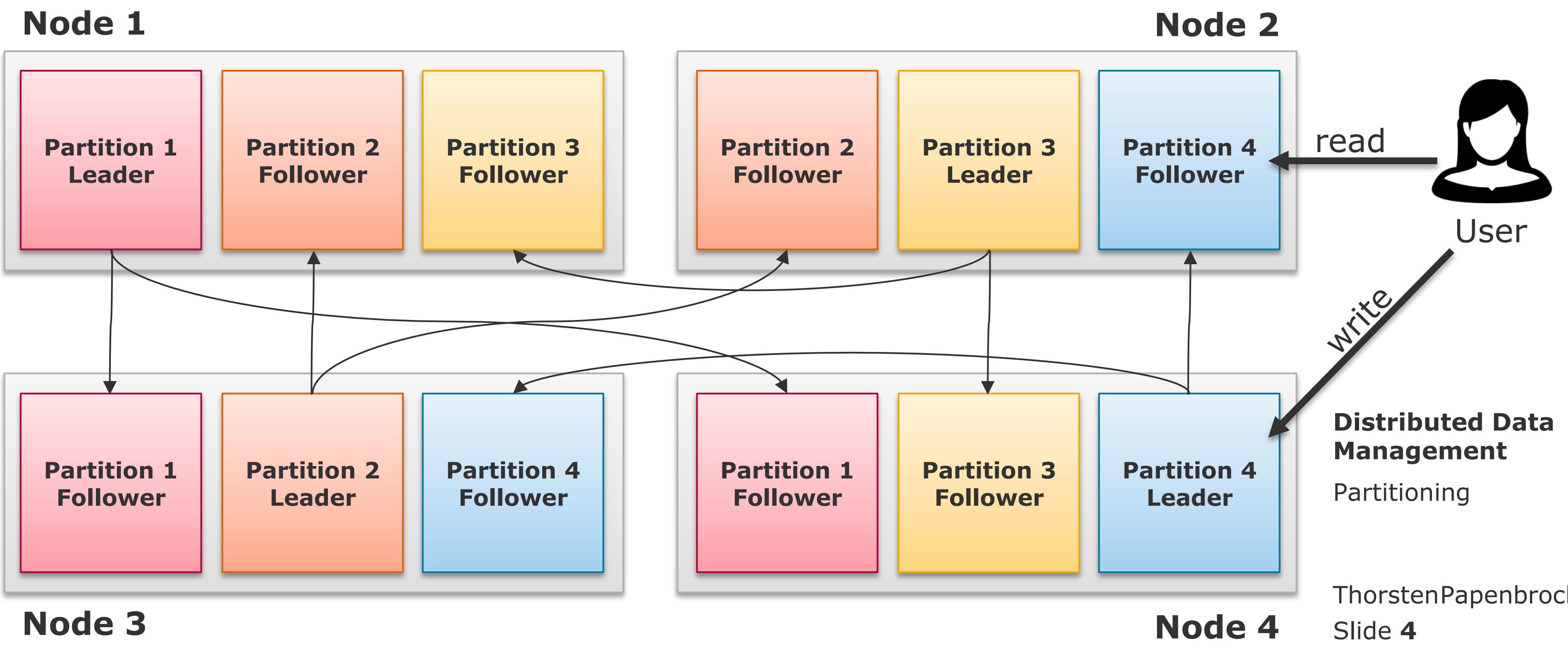
Different dimensions but essentially the same partitioning strategies



Used in most distributed systems

Used in column-based DBMSs

Replication and Partitioning



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 tasks:
  1. Given any data item, assign it to a partition.
  2. Keep partitions (possibly) balanced.

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

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

Concepts

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

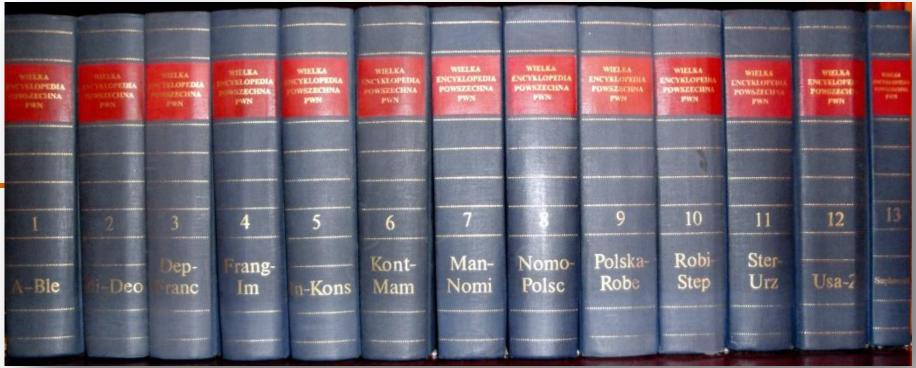
Dimension

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

Unbalancing issues

- Size/Load Skew: Some partitions have more data/queries than others.
  - Hot spots: Partitions that have disproportionately high load.

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 by Hash of Key

Hash Partitioning

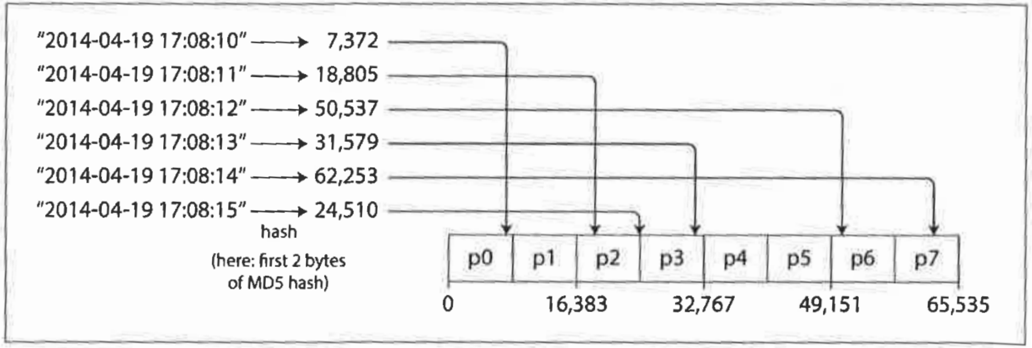
- Map the (skewed) range of keys to a uniformly distributed range of hashes.
- Use equidistant range partitioning on the range of hashes.
- Hash function:
  - calculates the key-to-hashes mapping (one-way-function)
  - skewed input, uniform output
  - e.g. MD5: a 128-bit hash function that maps arbitrary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

Why not simply `hash % n`?

➢ Later!

Partition lookup for (new or existing) key

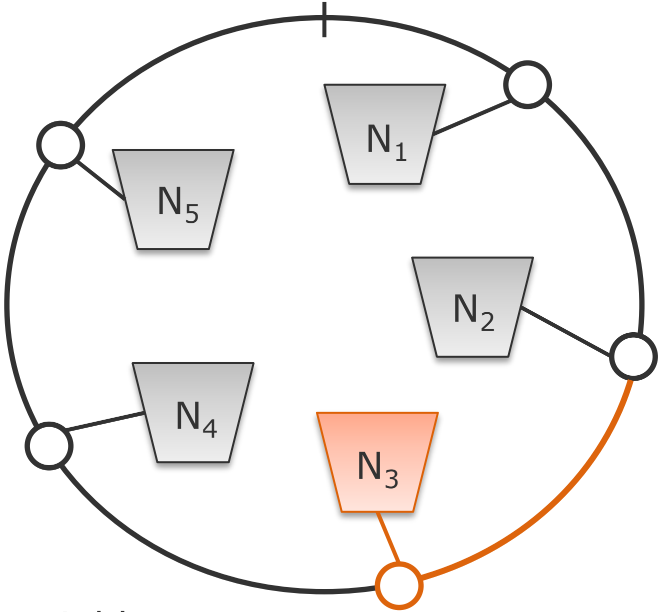
- Calculate hash
- Find partition where $min \leq key < max$ (binary search)



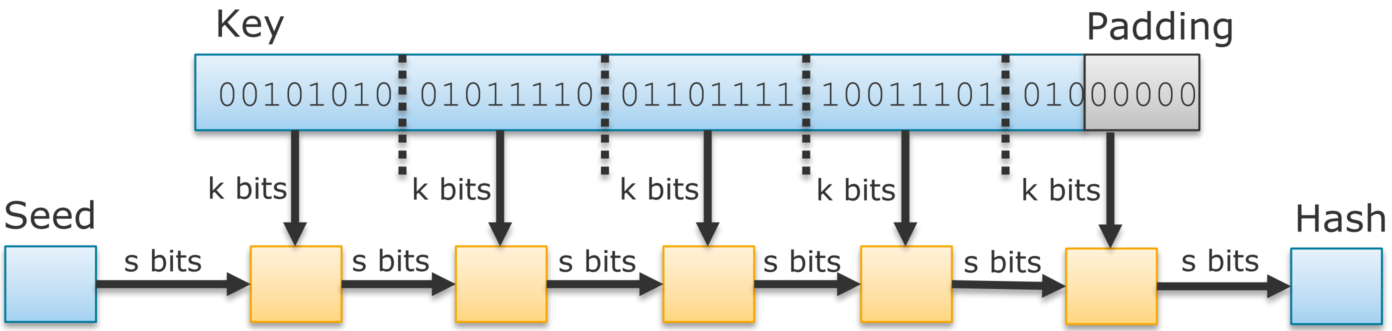
Partitioning by Hash of Key

Hash Partitioning

- Consistent Hashing
  - ➢ Used to keep partition-to-node assignments stable
  - 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.
    - ➢ Partition sizes my be unbalanced.



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.

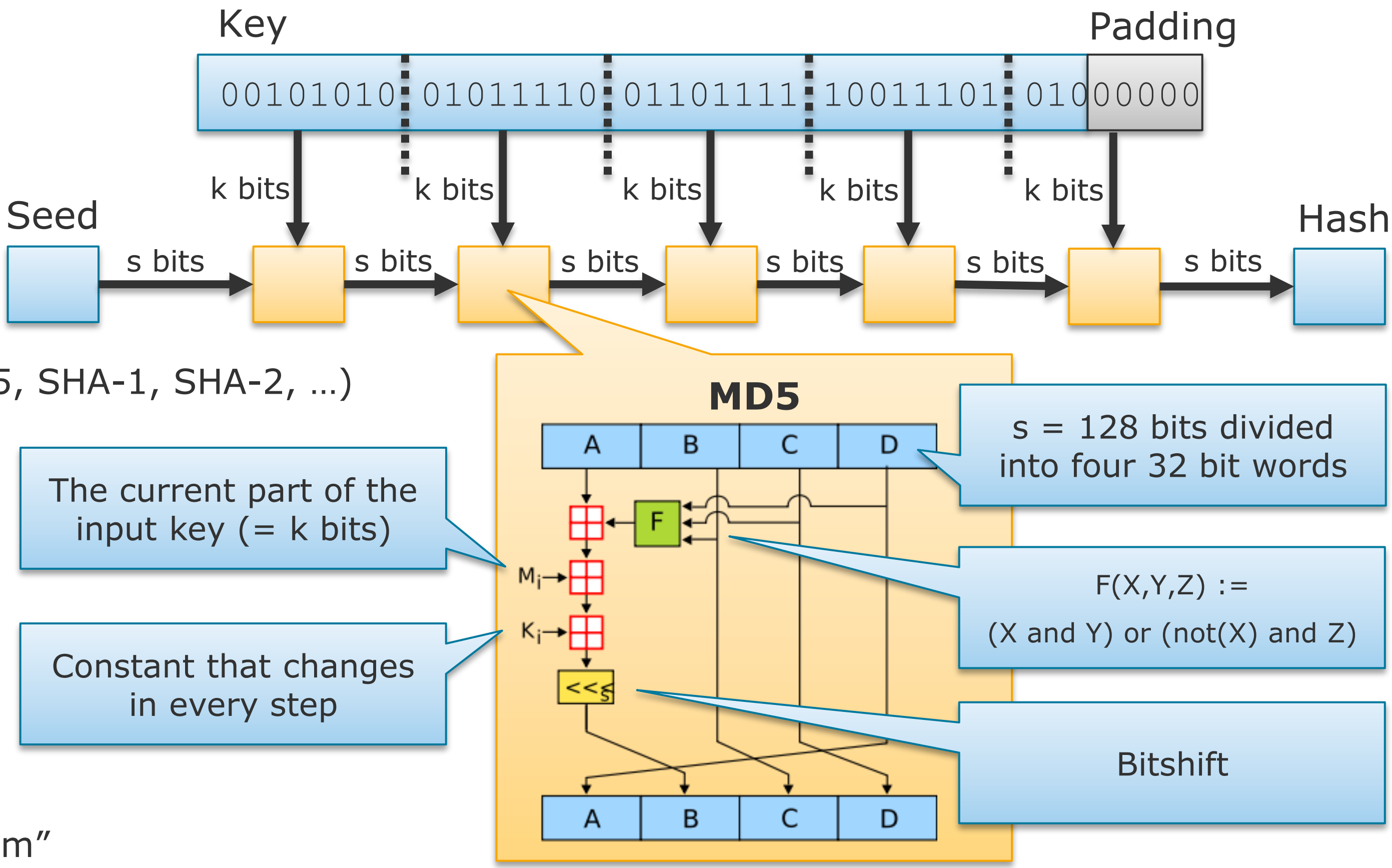
Partitioning by Hash of Key

Hashing

- Use cases:
  - Cryptography
  - Checksums
  - Partitioning
- Algorithm: (MD4, MD5, SHA-1, SHA-2, …)

Weak for encryption, but good for partitioning

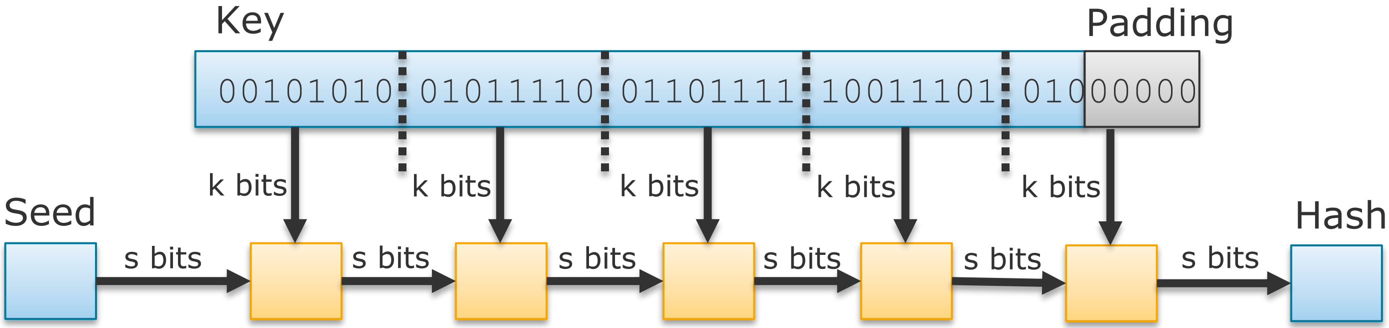
- MD5: "Message Digest 5"
- SHA-2: "Secure Hash Algorithm"

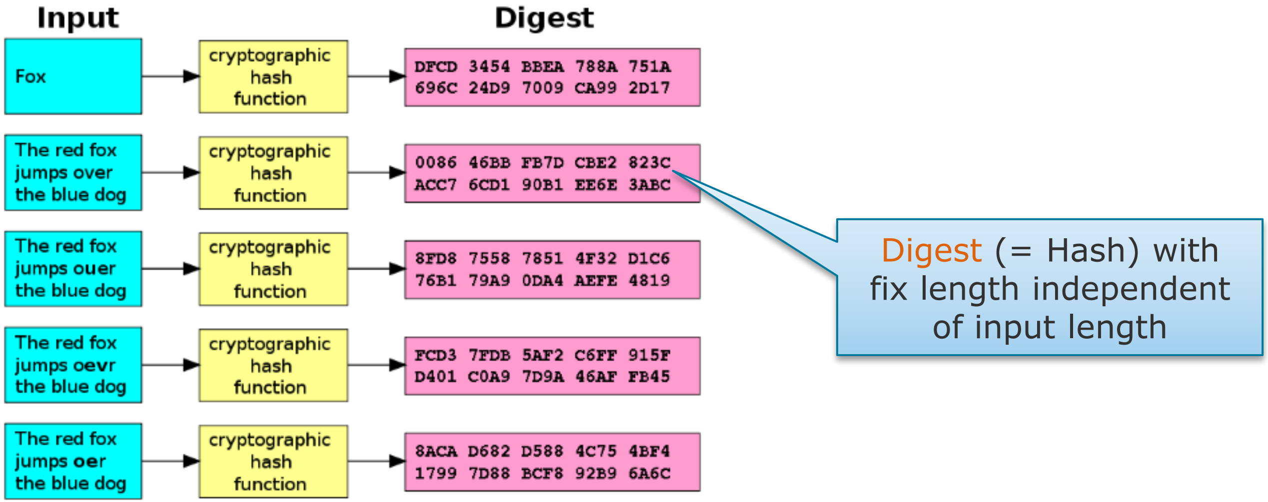


Partitioning by Hash of Key

Hashing

- **Use cases:**
  - Cryptography
  - Checksums
  - Partitioning
- **Example:**





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

Secondary Indexes

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);

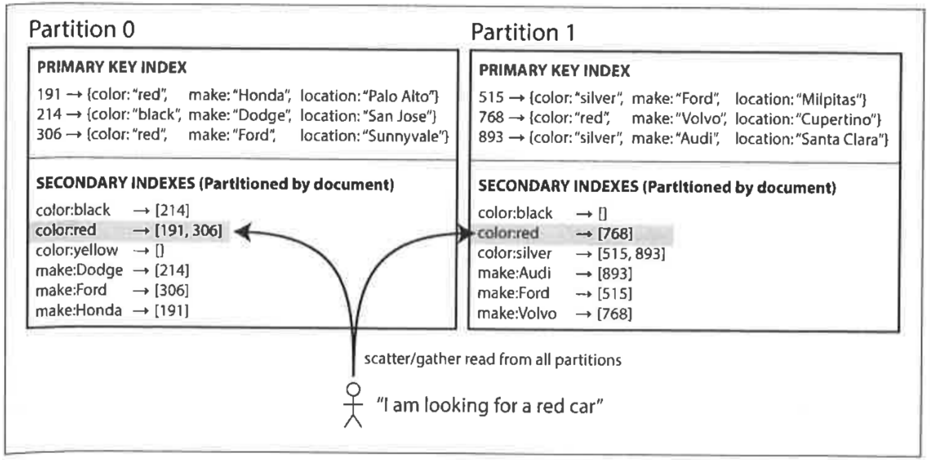
CREATE INDEX idx_make_filter **ON** Cars (make);

Partitioning and Secondary Indexes

Partitioning Secondary Indexes …

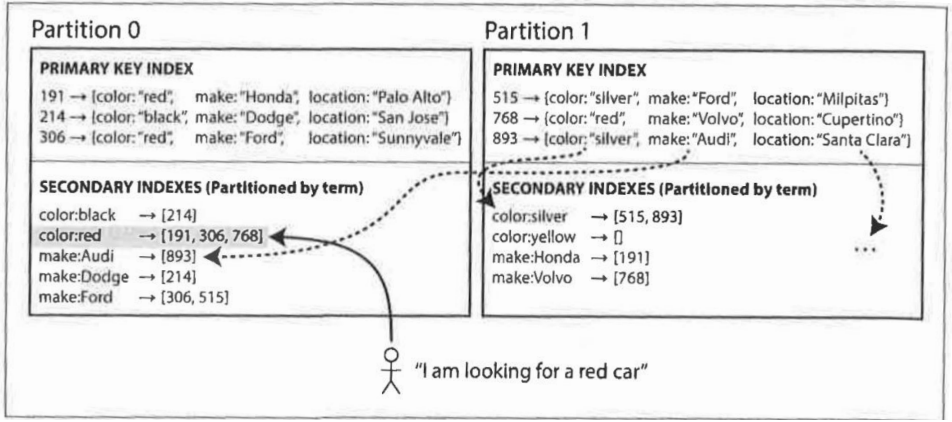
by Document: **Local Index**

- 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



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

MongoDB

Riak

Cassandra

Elasticsearch

SolrCloud

VoltDB

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

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.

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

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: `hash % n`

- `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`, …

`hash % n` is still useful for e.g. load balancing or fixed partitionings, because lookup is in O(1)

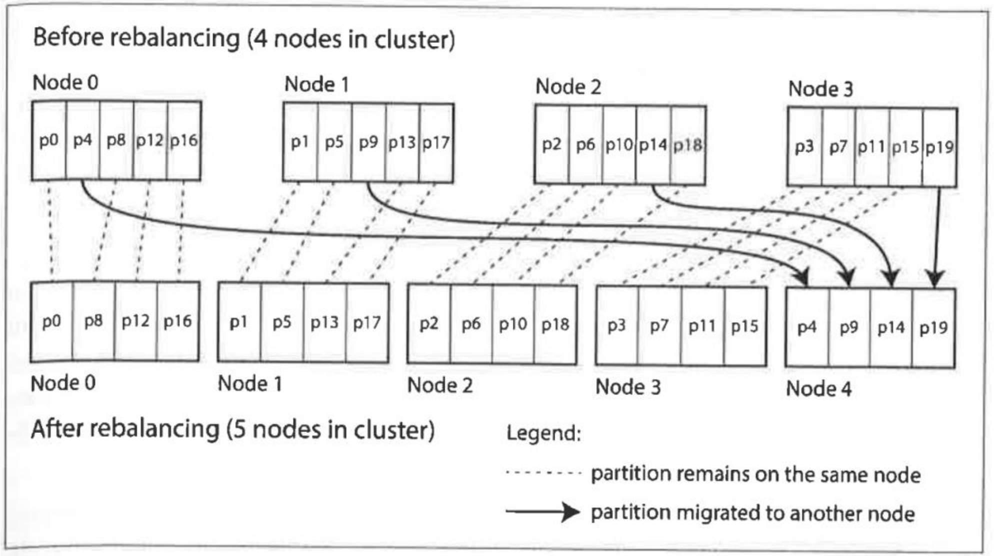
Fixed Number of Partitions

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



Fixed Number of Partitions

Idea

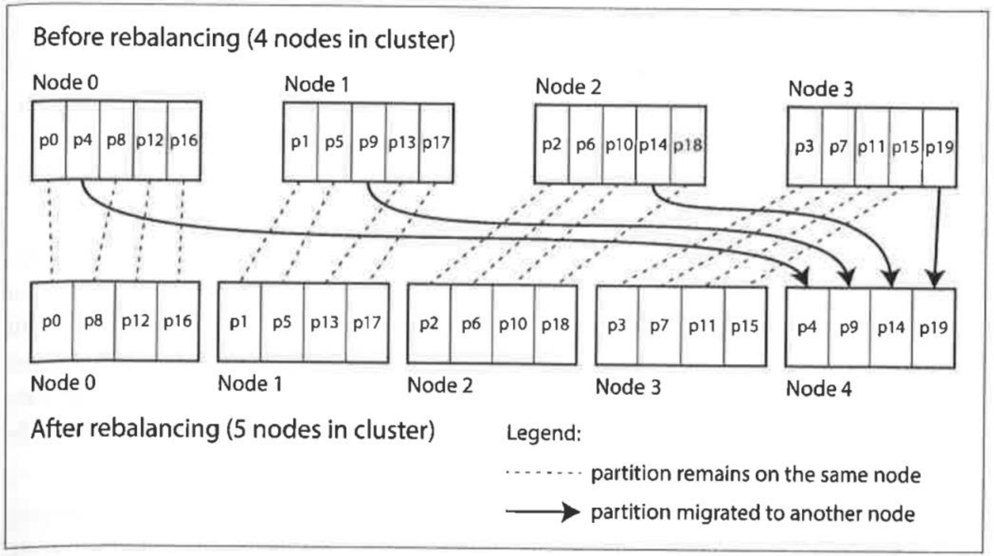
- Create many more partitions p than there are nodes n, i.e., several partitions per node.
- Let new nodes “steal” partitions from all other nodes until distribution is even again.
  - Key → partition mappings stay fix

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).

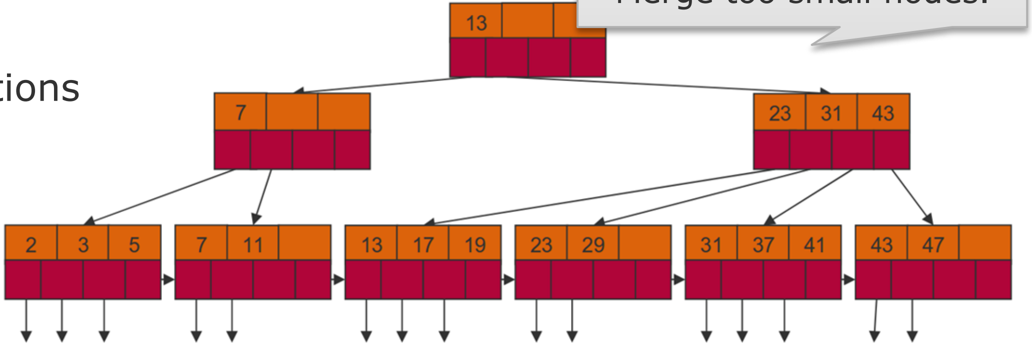
Partition to node assignment

- 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

Fixed Number of Partitions per Node

HPI Hasso Plattner Institut

Idea

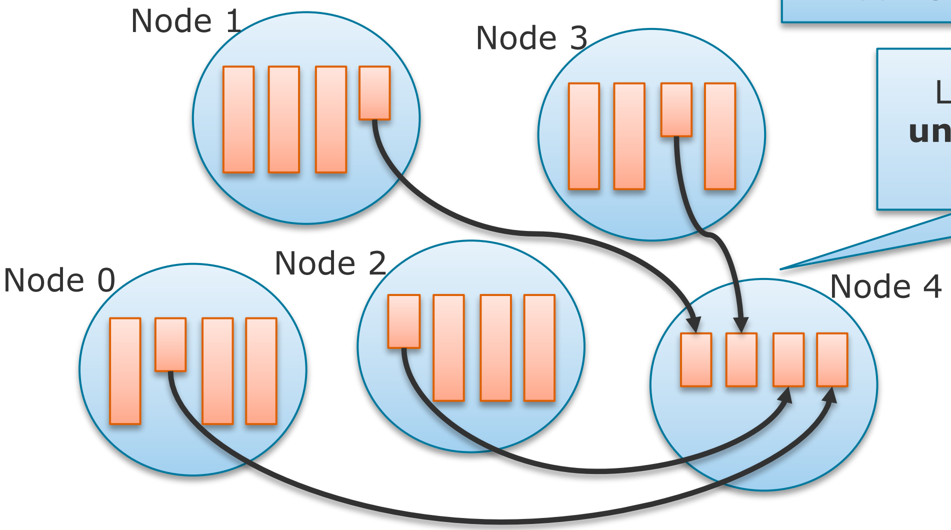
- Create a fix number of p partitions on each of the n nodes.
- Let new nodes fill their own p partitions by randomly splitting partitions on other nodes.
  - Steal half of p partitions from other nodes.

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**

Partition Lookup

Partitions move between nodes regardless of the rebalancing strategy

Ask any node

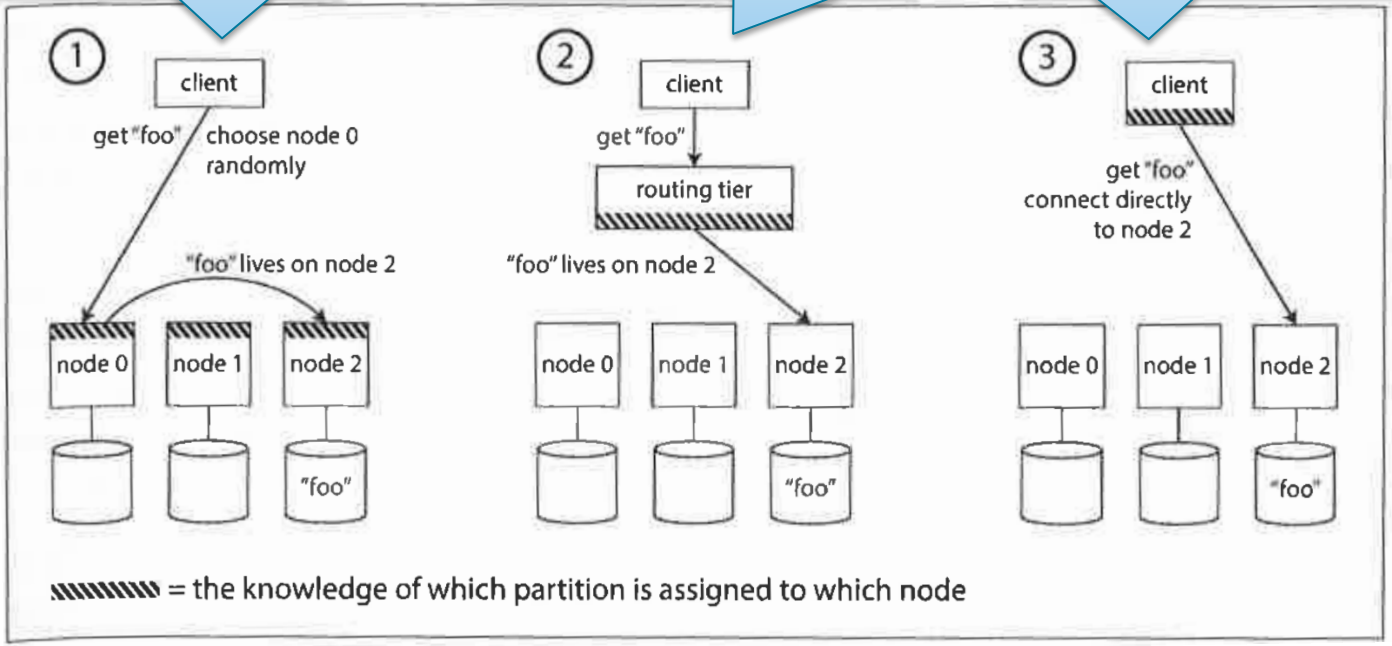
All nodes store a lookup table copy and can redirect queries.

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

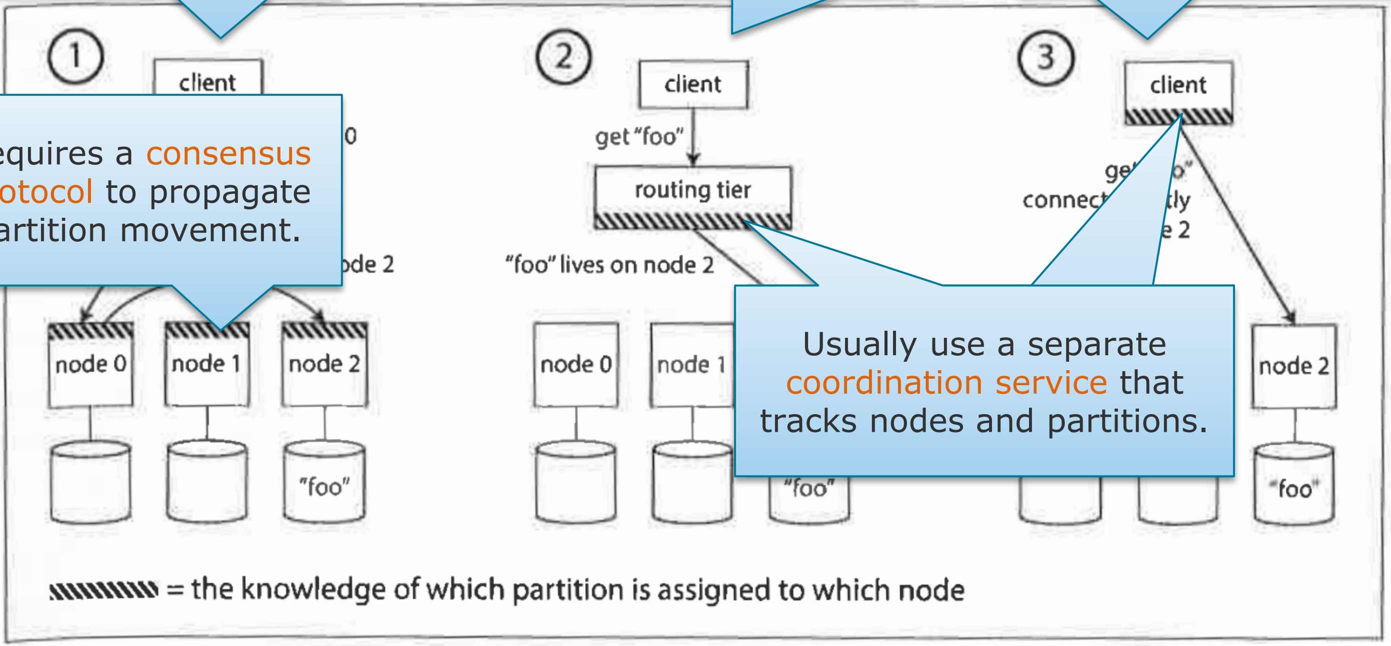
All nodes store a lookup table copy and can redirect queries.

Ask a router

A routing tier stores the lookup table copy and can redirect queries.

Ask node directly

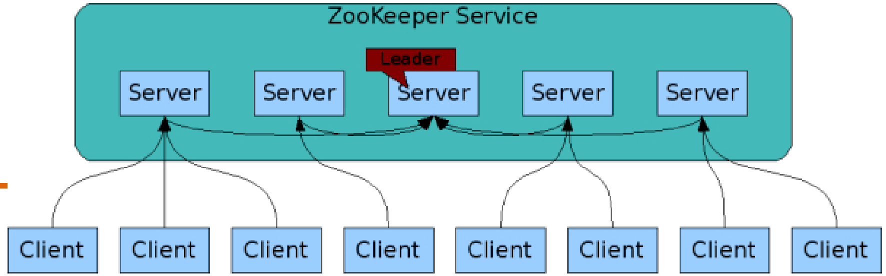
All clients store a lookup table copy and can locate partitions.



Requires a consensus protocol to propagate partition movement.

Usually use a separate coordination service that tracks nodes and partitions.

Partition Lookup: ZooKeeper

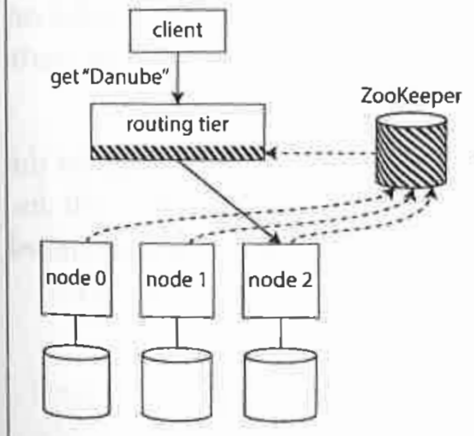


Apache ZooKeeper

- 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:

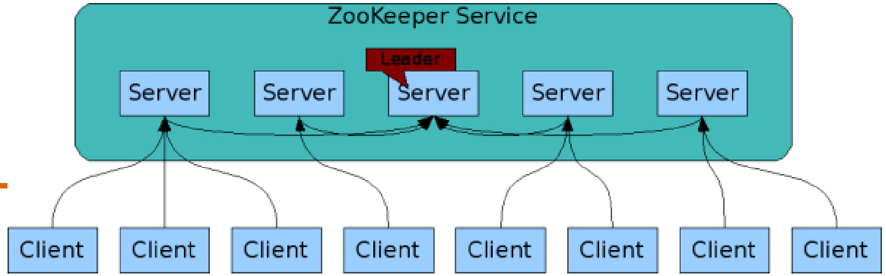
- Each router/client maintains a TCP connection.
- Nodes send heart beats and partition updates.
- Router/clients get partition addresses upon request.



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

= the knowledge of which partition is assigned to which node

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 sessions!

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, …)

Check yourself

The consistent hashing method as described on slide 11 has a number of shortcomings. To overcome those issues, real-world implementations often introduce additional virtual nodes for each physical node in the system.

1) Can you name three different shortcomings?

   Hint: Think of assumptions that might not hold in practice.

2) How could virtual nodes solve those issues?

Chapter 6: Partitioning