# Sharding
# a.k.a. Partitioning

Data At Scale

# Sharding

- Distribute a large data set over several nodes
  - Known as shards in MongoDB, ElasticSearch, SolrCloud
  - Known as region in HBase
  - Tablet in big table
  - vnode in Cassandra and Riak
  - vBucket in Couchbase
  - Partition everywhere else

# Sharding

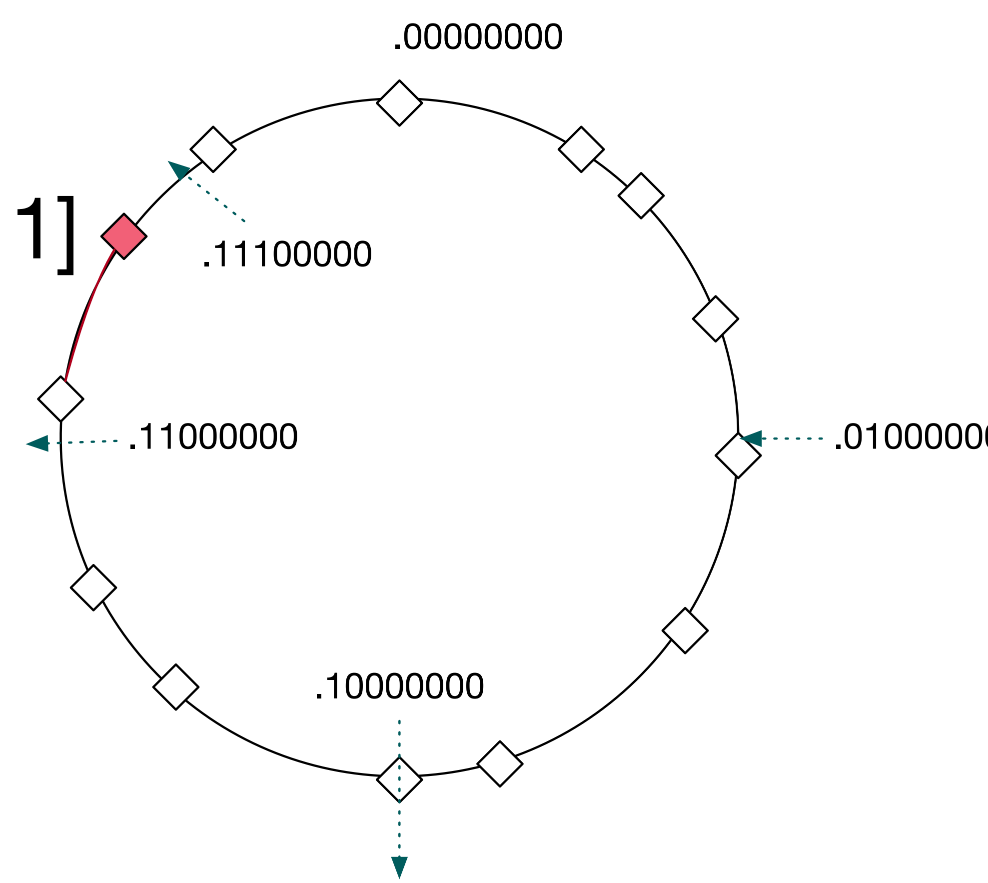
- Distributing relational databases
  - Strategy 1:  Assign tables to different nodes
  - Strategy 2: Horizontal partitioning of a table
    - Partitions the relations (rows)
  - Strategy 3: Vertical partitioning of a table
    - Partitions the columns
    - Usually repeats a column
    - Extreme form is columnar storage

# Sharding $\neq$ Replication

- Replication:
  - The same datum is stored at different nodes
- Partitioning:
  - Datum is only stored once, but different nodes store different data

# Partitioning by Key

- Linear Hashing
- Consistent Hashing
  - Keys are interpreted as binary numbers in [0, 1]
    - Arranged in a virtual circle
  - Nodes are given random ids in [0, 1]
  - Keys go to the next higher node

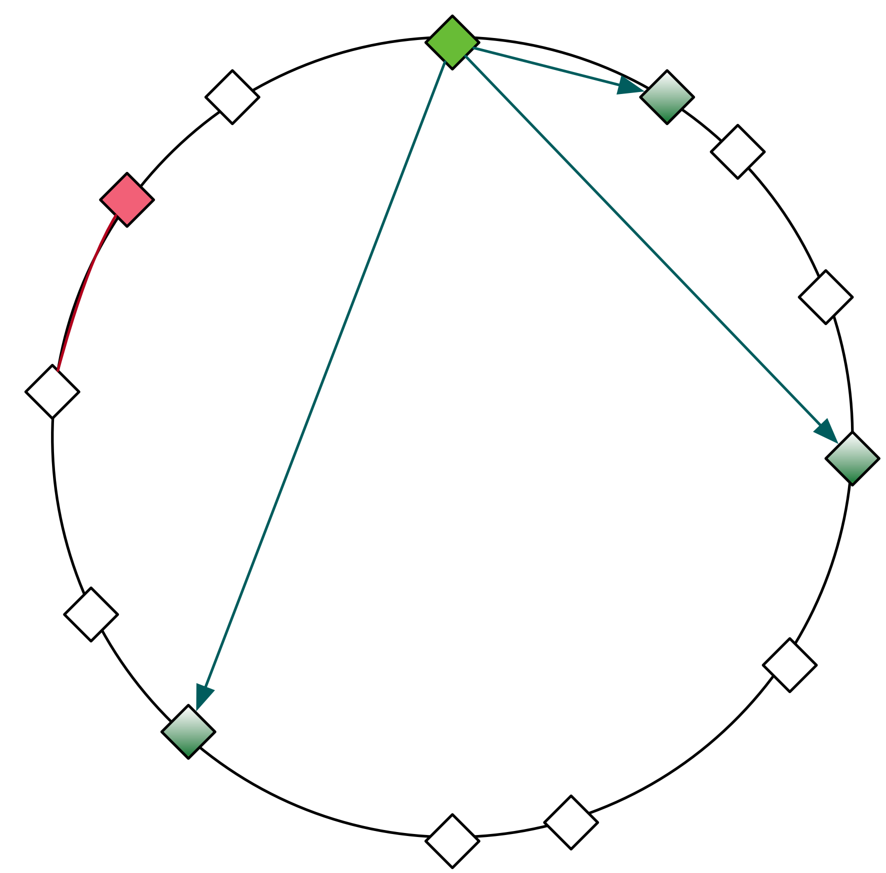


# Partitioning by key

- Distributed Hash Tables —Chord (2001)
  - P2P system using consistent hashing
    - Client looking for a record with a key
      - Needs to find one single node
      - Then Chord routing takes place

# Partitioning by key

- Each node knows its successor
  - This is enough for routing:
    - The request is sent to the next node until the node is the one responsible for the key
- To speed searches up, each node contains pointers to two other nodes further away
  - Route request to the furthest node with ID <= key



# Partitioning by key

- A new node is assigned a random key, splitting a random partition
    - Alternative: Broadcast from all nodes, finding the most overloaded one and give new node an id in the middle of the partition
- To allow updates: each node must know its predecessor as well
- Allow Stabilization process that updates finger table

- Chord with *n* nodes uses $O(\mathit{log}\ n)$ nodes to find a key

# Partitioning by key

- Hash function
  - Needs to be derived directly from key
    - E.g. Java Object.hashCode( ) and Ruby Object#hash depend on the system and the key
  - Needs to have good statistical properties
  - **Does usually not need to be cryptographically secure**

# Partitioning by key

- Ranges:  RP*, BigCloud, SDDS B*-tree

# Skewed Workloads

- No partitioning scheme known today deals well with hot keys
  - Records with a hot key overflow each node
  - Can artificially alter key to distribute over several nodes

# Secondary Indices

- Databases have long used a number of secondary indices in order to speed up access
- Some key value stores do not allow secondary indices
  - Hbase, Voldemort
- Others add them because of their usefulness
  - Riak

# Partitioning with Secondary Indices

- Partitioning Secondary Indices by Document
  - Use Document ID to partition documents
  - Each node maintains secondary indices only for its records
    - *Local secondary indices*
  - Using secondary indices means broadcasting to all nodes

# Partitioning with Secondary Indices

- Partitioning by Term
  - Create a <u>*global*</u> secondary index
  - Partition the global secondary index for fast access

# Rebalancing partitions

- Over time, databases tend to get bigger
  - Need to repartition
  - Bad Example: Using *key mod #nodes*
    - Almost all records move
  - LH* evolution (does not react to skewed workload)
  - Factorial number system

Thomas Schwarz, SJ, Ignacio Corderí, Darrell D.E. Long, Jehan-François Pâris: Simple, Exact Placement of Data in Containers, International Conference on Computing, Networking and Communications, Data Storage Technology and Applications Symposium, ICNC'13, San Diego, CA, January 28-31, 2013.

# Rebalancing partitions

- Create many partitions
  - Assign partitions to different nodes
    - Riak, Elasticsearch, Couchbase, Voldemort
- Dynamic Range Partitioning