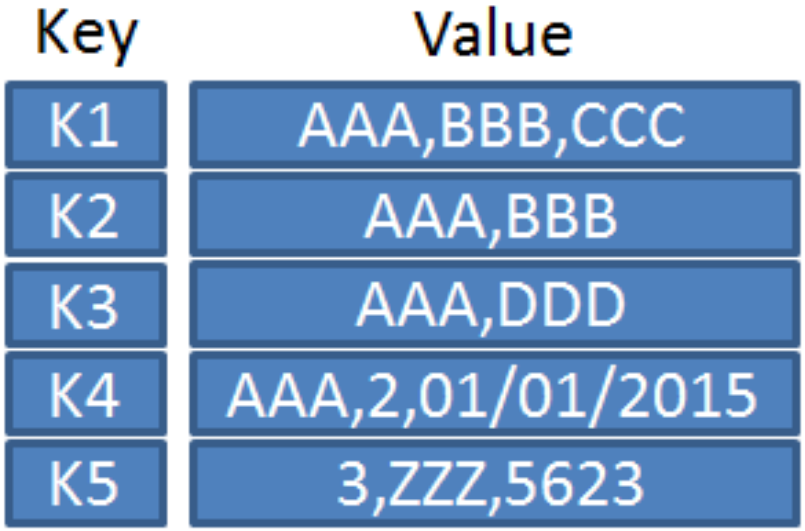
Types of NoSQL Databases – Key-value stores

Chapter 3

HO
GENT

Key-value Stores

- Key-value based database store data as (key, value) pairs
  - Keys are unique. Sometimes keys are composed of multiple fields to yield a unique key.
  - The value contains an uninterpreted value that may have any length and content. Because the value is not interpreted by the DBMS you can therefore write any data in it. The values of different records can have a completely different structure. There is no database scheme that specifies how the values should be structured.
  - Aka Hash map, or hash tabl or dictionary



Key-value Stores

- Key-value based database stores data as (key, value) pairs
  - Hash map, or hash table or dictionary
  - The basic idea is you have a key. You go to the database and tell: "Grab me the value of the key in the database." The database knows absolutely nothing about what's in that value. It could be a single number, it could be some complex document, it could be an image. The database doesn't know and doesn't care. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled.
  - https://try.redis.io/

Example of key value DB: Redis

- Try it yourself at: https://try.redis.io

Key-value Stores: Java example (in memory)

```
import java.util.HashMap;
import java.util.Map;
public class KeyValueStoreExample {
            public static void main(String...  args) {
                        // Keep track of age based on name
                        Map<String, Integer> age_by_name = new HashMap<>();

                        // Store some entries
                        age_by_name.put("wilfried", 34);
                        age_by_name.put("seppe", 30);
                        age_by_name.put("bart", 46);
                        age_by_name.put("jeanne", 19);

                        // Get an entry
                        int age_of_wilfried = age_by_name.get("wilfried");
                        System.out.println("Wilfried's age: " + age_of_wilfried);

                        // Keys are unique
                        age_by_name.put("seppe", 50); // Overrides previous entry
            }
}
```

Key-value Stores

- Keys (e.g., “bart”, “seppe”) are hashed by means of a so-called **hash function**
  - A hash function takes an arbitrary value of arbitrary size and maps it to a key with a fixed size, which is called the hash value.
  - Each hash can be mapped to a space in computer memory. This ensures fast retrieval based on the key.

Key
wilfried
seppe
bart
jeanne

Hash	Key
01	(wilfried,34)
03	(seppe,30)
07	(bart,46)
08	(jeanne,19)

Key-value Stores

- NoSQL databases (like key value stores) are built with horizontal scalability support in mind
- Distribute hash table over different locations
- Assume we need to spread our hashes over three servers
  - Hash every key (e.g. “wilfried”, “seppe”) to a server identifier or index by the formula
    - index(hash) = mod(hash, nrServers) + 1

Key-value Stores

Example (3 servers):

Key
wilfried
seppe
bart
jeanne

Hash function

Hash	Index (= mod 3 + 1)
01	2
03	1
07	2
08	3

Server #1	Entries
03	(seppe,30)

Server #2	Entries
01	(wilfried,34)
07	(bart,46)

Server #3	Entries
08	(jeanne,19)

= Sharding: distributes different data across multiple servers, so each server acts as a single source for a subset of data
(no redundant storage of data so far!)

Key-value Stores

- Example: Memcached
  - Implements a distributed memory-driven hash table (i.e. a key-value store), which is put in front of a traditional database (e.g. MS SQL Server) to speed up queries by caching recently accessed objects in RAM
  - = Caching solution
  - Next slides: same example with 3 serves and 4 entries.

Key-value Stores: MemCached in Java

```
import java.util.ArrayList;
import java.util.List;
import net.spy.memcached.AddrUtil;
import net.spy.memcached.MemcachedClient;

public class MemCachedExample {
  public static void main(String[] args) throws Exception {
    List<String> serverList = new ArrayList<String>() {
      {
        this.add("memcachedserver1.servers:11211");
        this.add("memcachedserver2.servers:11211");
        this.add("memcachedserver3.servers:11211");
      }
    };
```

Key-value Stores : MemCached in Java (cont'd)

```
MemcachedClient memcachedClient = new MemcachedClient(
       AddrUtil.getAddresses(serverList));

   // ADD adds an entry and does nothing if the key already exists
   // Think of it as an INSERT
   // The second parameter (0) indicates the expiration - 0 means no expiry
   memcachedClient.add("marc", 0, 34);
   memcachedClient.add("seppe", 0, 32);
   memcachedClient.add("bart", 0, 66);
   memcachedClient.add("jeanne", 0, 19);


   // SET sets an entry regardless of whether it exists
   // Think of it as an UPDATE-OR-INSERT
   memcachedClient.add("marc", 0, 1111); // <- ADD will have no effect
   memcachedClient.set("jeanne", 0, 12); // <- But SET will
```

Key-value Stores : MemCached in Java (cont'd)

```
// REPLACE replaces an entry and does nothing if the key does not exist
 // Think of it as an UPDATE
 memcachedClient.replace("not_existing_name", 0, 12); // <- Will have no effect
 memcachedClient.replace("jeanne", 0, 10);

 // DELETE deletes an entry, similar to an SQL DELETE statement
 memcachedClient.delete("seppe");

 // GET retrieves an entry
 Integer age_of_marc = (Integer) memcachedClient.get("marc");
 Integer age_of_short_lived = (Integer) memcachedClient.get("short_lived_name");
 Integer age_of_not_existing = (Integer) memcachedClient.get("not_existing_name");
 Integer age_of_seppe = (Integer) memcachedClient.get("seppe");
 System.out.println("Age of Marc: " + age_of_marc);
 System.out.println("Age of Seppe (deleted): " + age_of_seppe);
 System.out.println("Age of not existing name: " + age_of_not_existing);
 System.out.println("Age of short lived name (expired): " + age_of_short_lived);

 memcachedClient.shutdown();

 }
}
```

Key-Value Stores: example

- Example
  - Assume there is an **art museum** that has a mobile application by which **registered visitors** can give their **opinion about paintings** during their tour in the museum. The app registers **every minute automatically where the visitor is located** and gives information about the paintings on this location. The visitor subsequently can give his opinion.
  - The app generates a lot of data that must be written to the database quickly, so the developer has chosen to use a key-value database ‘Visitors opinions’.

HO
GENT

Key-Value Stores: example

- Example

VisitorID, timestamp	Value
V1, 15/1:14h00	‘Room 1’
V1, 15/1:14h01	‘Room 1, not nice, crowded’
V1, 15/1:14h01	‘Room 1, 2/10’
V2, 15/1:14h02	‘Room 1, Rembrandt is amazing’
V1, 15/1:14h03	‘Room 1, not having a good time’
V2, 15/1:14h03	‘Room 1, 9/10’
V1, 15/1:14h04	‘Room 2, this is more like it, 7/10’
V2, 15/1:14h04	‘Room 1, really nice in here’
V3, 15/1:14h04	‘Room 1, crowded’

HO
GENT

Key-Value Stores: example

- Example
  - There is a composite key that consists of VisitorID and timestamp. The values are rather simple. This will make the processing of the values easier. If the visitor doesn't do anything, only the location is registered during the visit. This can bring new insights for a future organisation of the museum.
  - A few possible instructions
    - get(V2, 15/1:14h02) → what was registered for visitor V2 on 15/1 at 14h02
    - put(V3, 15/1:14h05, 'Room 1, Wauw!') → add a new record
    - delete(V2, 15/1:14h02) → remove a record
  - There is no support for complex (SQL like) queries. If you want to know the number of visitors in Room 1 on 15/1 between 14h00 and 14h59, you have to write code in the app to calculate this.

HO GENT

NoSQL clusters

Cluster = group of servers working together as a single logic server

- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying

NoSQL clusters

- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying

Request Coordination

- In many NoSQL implementations (e.g. Cassandra, Google's BigTable, Amazon's DynamoDB) all nodes implement the same functionality
- There is no *master* node, because this creates a single point of failure.
- Are all able to perform the role of request coordinator
- → Need for membership protocol that ensures
  - Dissemination (nodes are aware of each other)
    - Based on periodic, pairwise communication
  - Failure detection

NoSQL clusters

- Request Coordination
- **Consistent Hashing**
- Replication and Redundancy
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying

Consistent Hashing

- **Consistent hashing** schemes are often used, which avoid having to remap each key to a new node when nodes are added or removed
- Suppose we have a situation where 10 keys are distributed over 3 servers ($n = 3$) with the following hash function
  – $h(key) = key$ modulo n
  (or h(key) = key modulo n + 1 as in the above example)

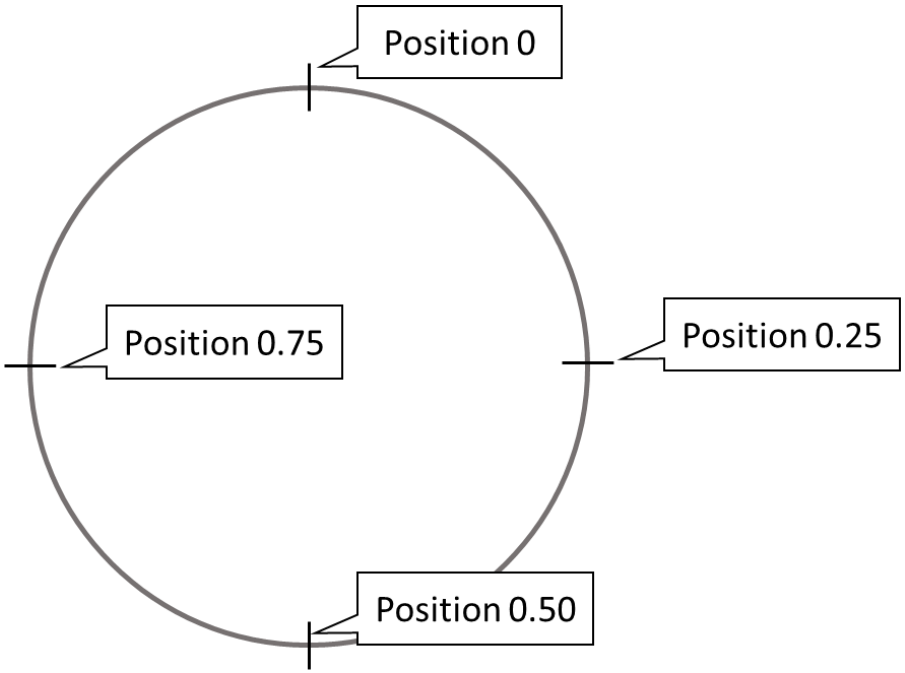
Consistent Hashing

	n		
key	3	2	4
0	0	0	0
1	1	1	1
2	2	0	2
3	0	1	3
4	1	0	0
5	2	1	1
6	0	0	2
7	1	1	3
8	2	0	0
9	0	1	1

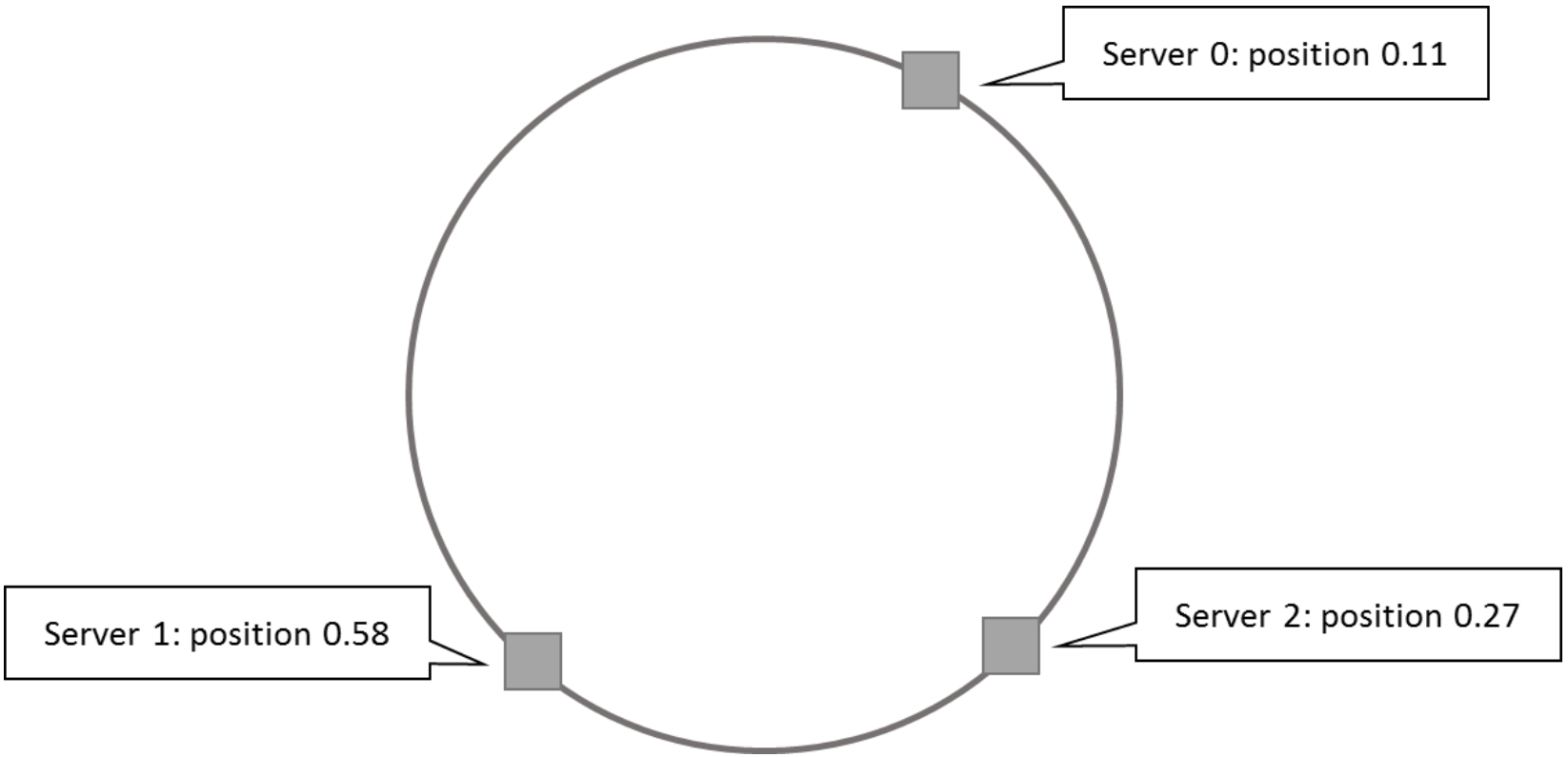
Effect of removing (n = 2) or adding (n = 4) a server: most pairs have to be moved to another server.

Solution: Consistent Hashing

- At the core of a consistent hashing setup is a so called **“ring”-topology**, which is basically a representation of the number range [0,1[:



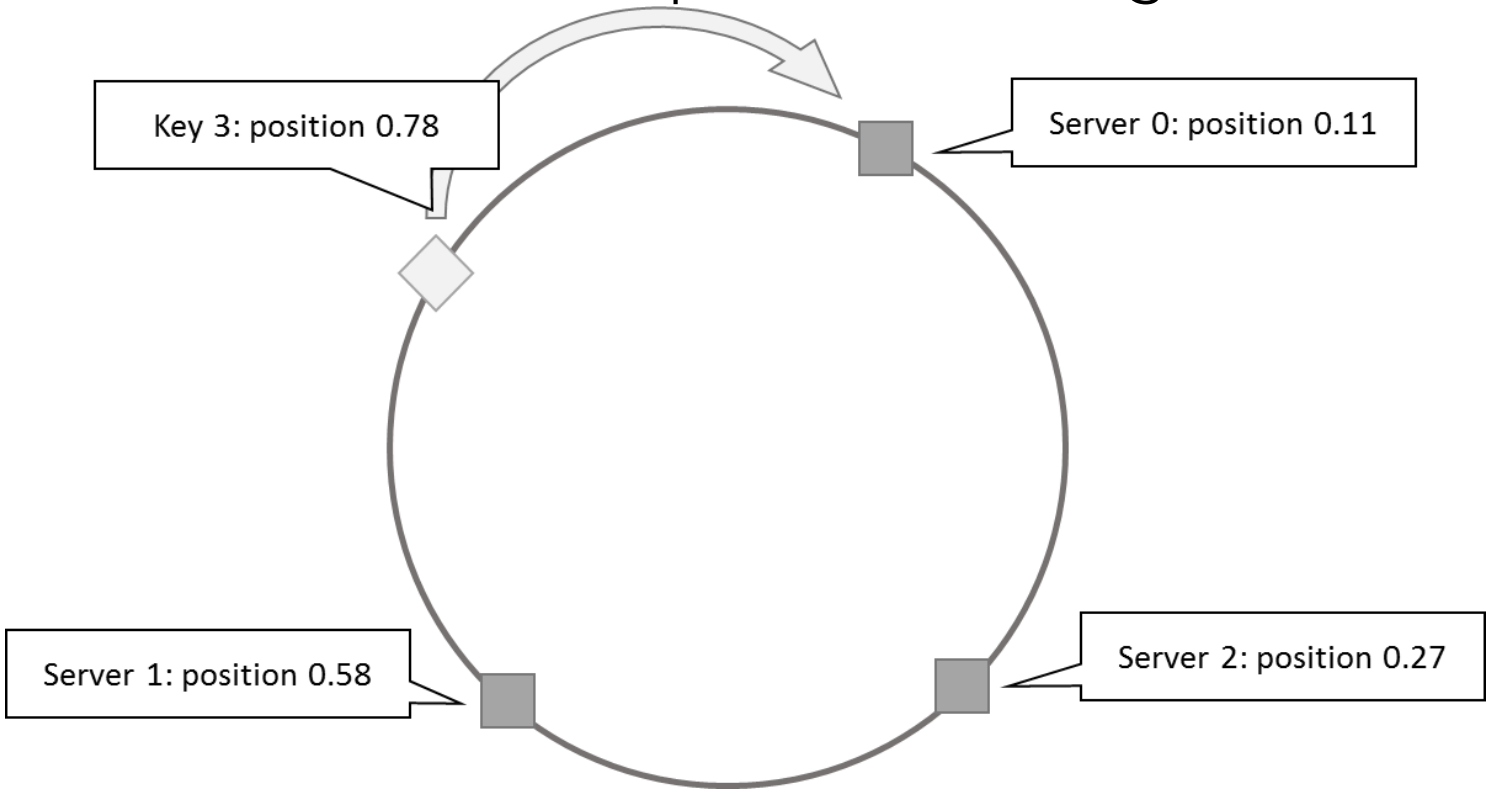
Consistent Hashing



All servers (three servers with identifiers 0, 1, 2) are hashed to place them in a position on this ring. 0.11, 0.27 and 0.58 are examples of hash values for 0, 2 and 1

Consistent Hashing

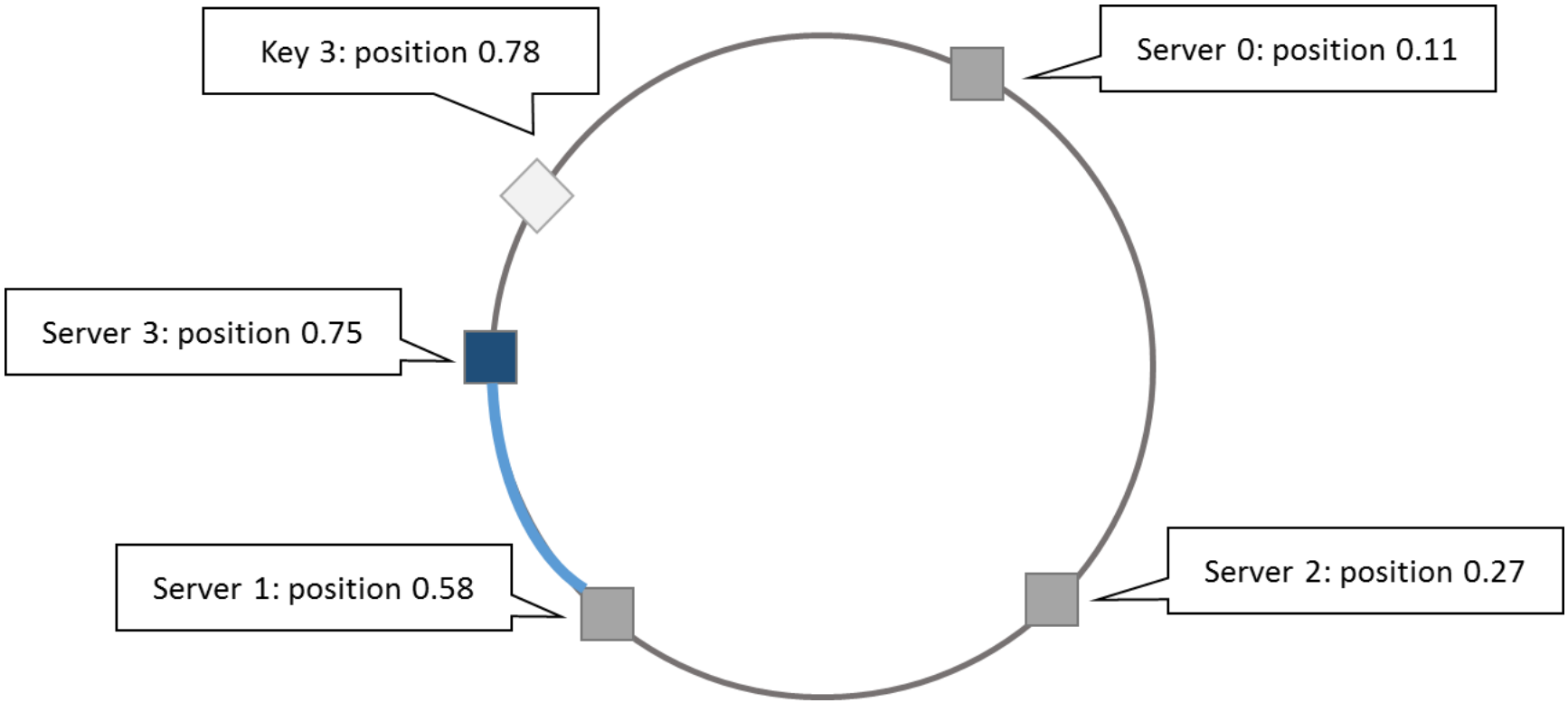
- Hash each key to a position on the ring, and store the actual key-value pair on the first server that appears clockwise of the hashed point on the ring



Consistent Hashing

- Because of the uniformity property of a “good” hash function, roughly $1/n$ of key-value pairs will end up being stored on each server
- Most of the key-value pairs will remain unaffected in case a machine is added or removed as illustrated on the next slide.

Consistent Hashing: add a machine



When Server 3 is added only the keys in the blue zone have to be moved from Server 0 to Server 3.

NoSQL clusters

- Request Coordination
- Consistent Hashing
- **Replication and Redundancy**
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying

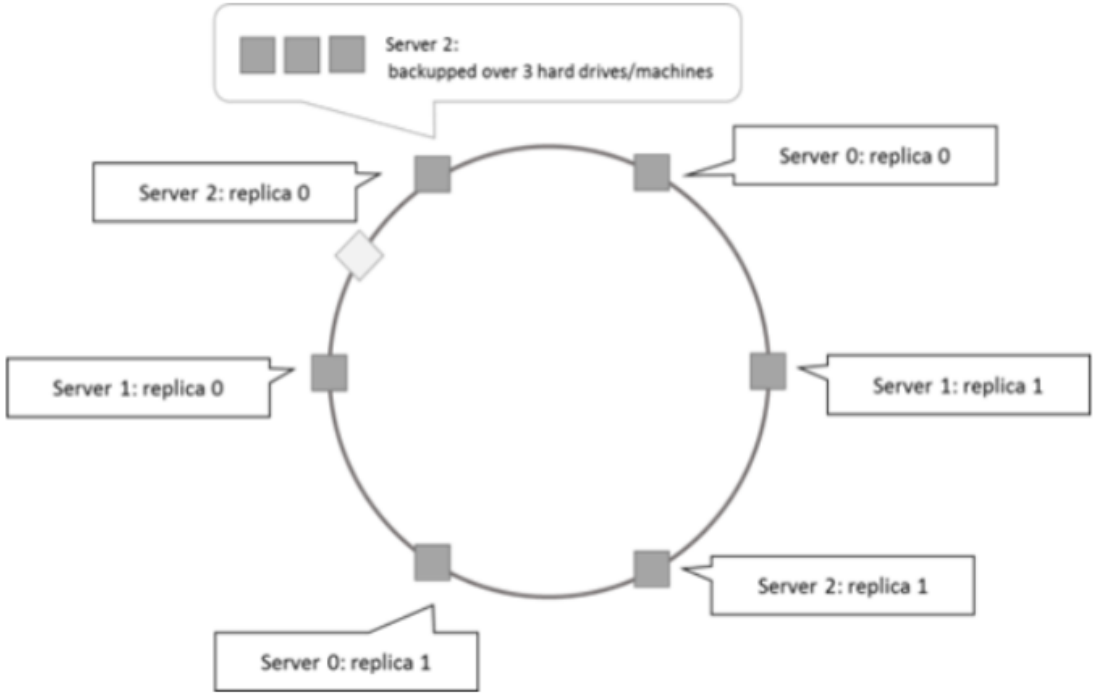
Replication and Redundancy

- Problems with consistent hashing:
  - If 2 servers end up being mapped close to one another, one of these nodes will end up with few keys to store
  - In case a server is added, all of the keys moved to this new node originate from just one other server
- Instead of mapping a **server** *s* to a single point on our ring, we map it to multiple positions, called **replicas**
- For each physical server *s*, we hence end up with *r* (the number of replicas) points on the ring
- Note: each of the replicas still represents the same physical instance (≠ redundancy)
  - Virtual nodes

Replication and Redundancy

- It is also possible to set up a full redundancy scheme where each node itself corresponds to multiple physical machines each storing a fully redundant copy of the data:

- replica: virtual node to tackle consistent hashing problems
- backup: physical copy



NoSQL clusters

- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- **Eventual Consistency**
- Stabilization
- Integrity Constraints and Querying

Eventual Consistency

- Membership protocol does not guarantee that every node is aware of every other node *at all times* (focus on availability, not on consistency)
  – It will reach a consistent state over time
- State of the network might not be perfectly consistent at any moment in time, though will become eventually consistent at a future point in time
- Many NoSQL databases guarantee so called **eventual consistency**

Eventual Consistency

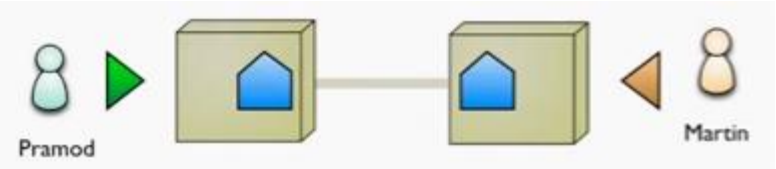
- Most NoSQL databases follow the **BASE** principle
  - Basically available, Soft state, Eventual consistency
  - As opposed to ACID (Atomicity, Consistency, Isolation, Durability) in Relational DMBS.
- **CAP theorem** states that a distributed computer system cannot guarantee the following three properties at the same time:
  - Consistency (all nodes see the same data at the same time)
  - Availability (guarantees that every request receives a response indicating a success or failure result)
  - Partition tolerance (the system continues to work even if nodes go down or are added).

Eventual Consistency

- Most NoSQL databases sacrifice the consistency part of CAP in their setup, instead striving for eventual consistency (*business must go on at all time*)
- The full BASE acronym stands for:
  - **B**asically **a**vailable: NoSQL databases adhere to the availability guarantee of the CAP theorem
  - **S**oft state: the system can change over time, even without receiving input (since nodes continue to update each other)
  - **E**ventual consistency: the system will become consistent over time but might not be consistent at a particular moment

Consistency and availability

- As soon as you replicate data a new class of consistency problems starts coming in.
- This will be illustrated with a following example.
- Pramod (in India) and Martin (in the US) want to book the same hotel room
- They send out a booking request
- They send their requests to local processing nodes

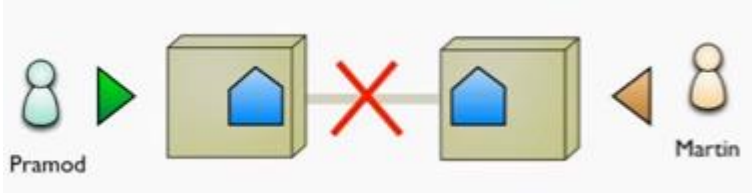


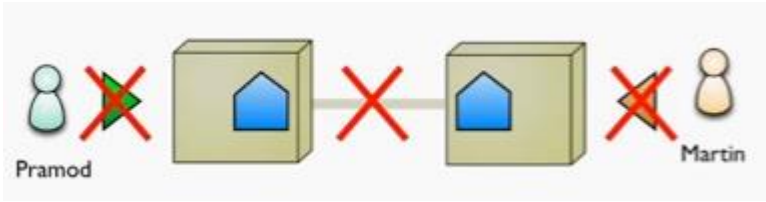
- The processing nodes need to communicate ensuring only one of both can book the hotel room

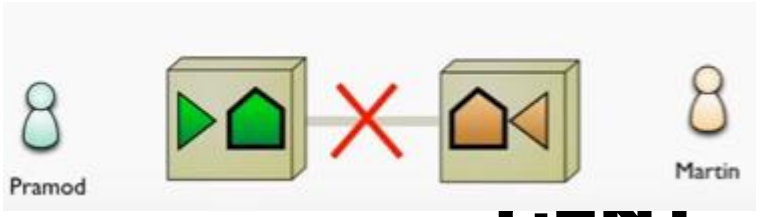
HO
GENT

Consistency and availability

- Again they both want to book a hotel room, **but the communication line has gone down.** The two nodes cannot communicate.
- They send out their requests
- Alternative 1
  The system tells the users the communication lines have gone down, so they can't take the hotel bookings at the moment
- Alternative 2
  The system accepts both bookings and the hotel room gets double booked







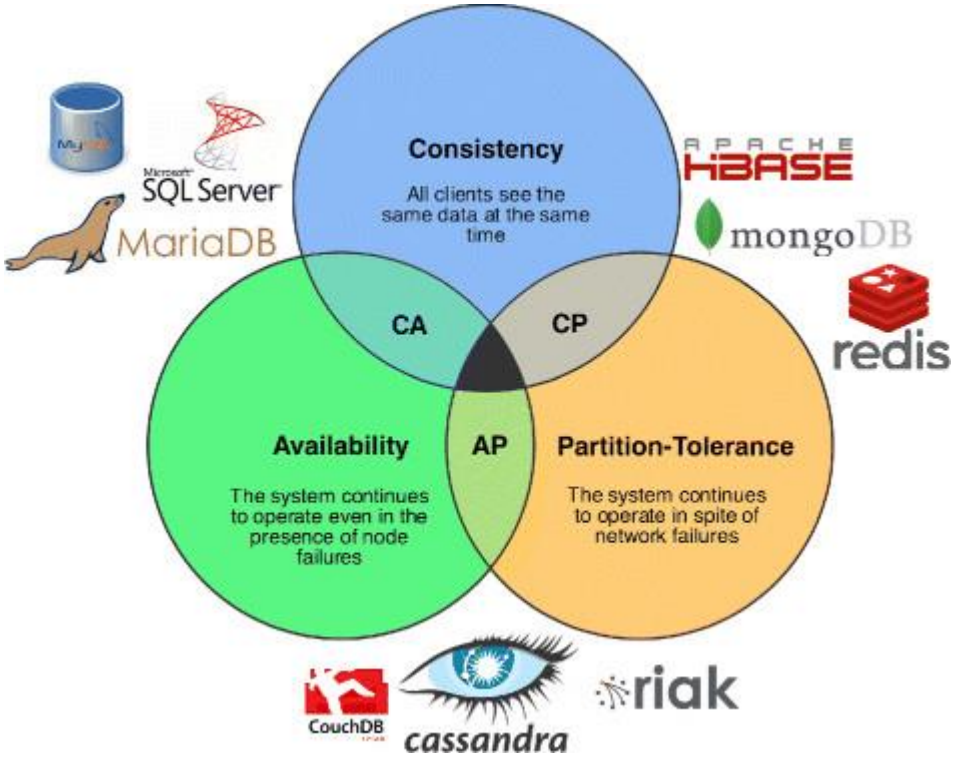
Consistency and availability

- This illustrates a **choice** between consistency (“I’m not going to do anything while the communication lines are down”) and availability (“I’m going to keep on going but at the risk of introducing an inconsistent behavior”)
- This is a business choice (IT can’t decide on this).

HO
GENT

CAP theorem

- As soon as you have a distributed system, you only can pick 2 out of 3
- This isn't a single binary choice. You can trade off levels of consistency and availability



HO GENT

NoSQL clusters

- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- Eventual Consistency
- **Stabilization**
- Integrity Constraints and Querying

Stabilization

- The operation which repartitions hashes over nodes in case nodes are added or removed is called **stabilization**
- If a consistent hashing scheme being applied, the number of fluctuations in the hash-node mappings will be minimized.

NoSQL clusters

- Request Coordination
- Consistent Hashing
- Replication and Redundancy
- Eventual Consistency
- Stabilization
- Integrity Constraints and Querying

Integrity Constraints and Querying

- **Key value stores** represent a very diverse gamut of systems
- Full blown DBMSs versus caches
- Only limited query facilities are offered
  - E.g. put and set
- Limited to no means to enforce structural constraints
  - DBMS remains agnostic to the internal structure
- No relationships, referential integrity constraints or database schema, can be defined.
- The database knows nothing about the value: it could be a single number, a complex document, an image, … The value is a blob that the data store just stores, whithout caring or knowing what's inside
- They always use primary-key access => great performance