Amazon's DynamoDB

Overview

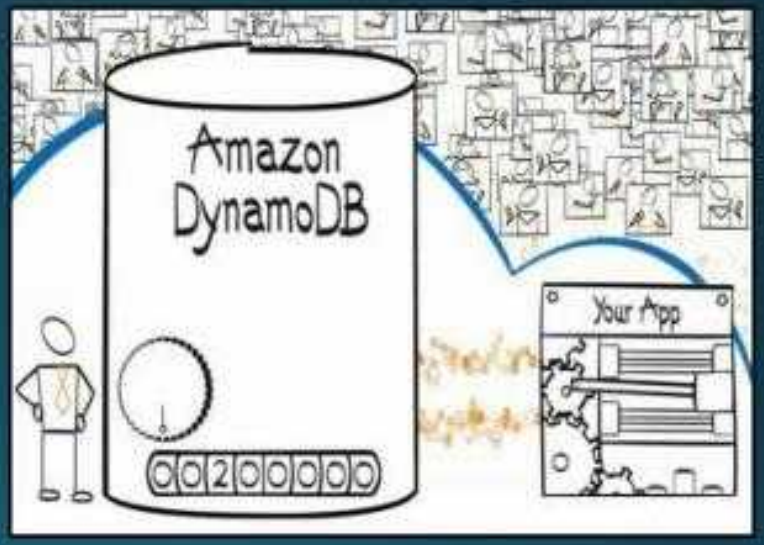
- DynamoDB Feature
- DynamoDB Data Model
- Voldemort Key-Value Distributed Data Store

Introduction to DynamoDB

- Fully managed NoSQL database service by Amazon
- Database type: Key-value stores
- Designed to address the core problems of database management, performance, scalability, and reliability

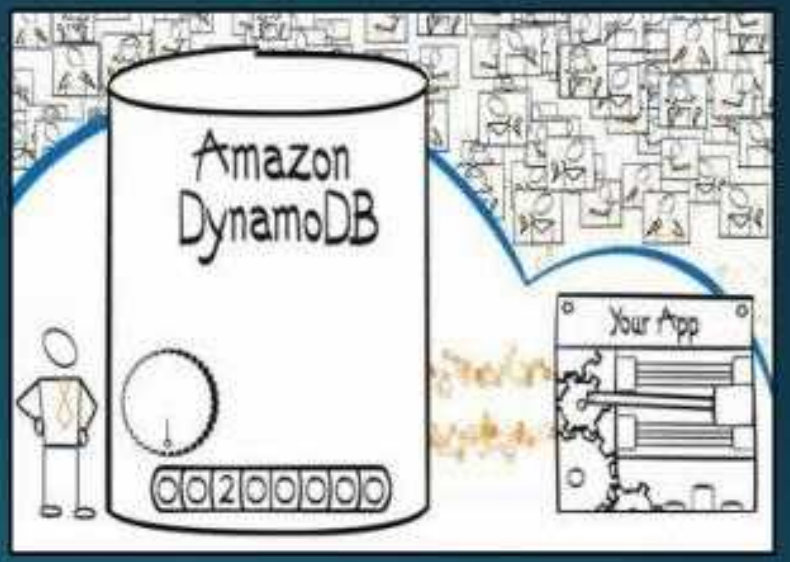
Features

- Scalable
  - Provisioned Throughput
  - Fully Distributed, Shared Nothing Architecture
- Fast Performance
  - Average service-side latencies < 10 ms
  - The service runs on Solid State Disks - consistent, fast latencies at any scale
- Easy Administration and Cost Effective
  - a fully managed service by Amazon
- Fault-tolerant
  - Synchronous replication across multiple zones in a region

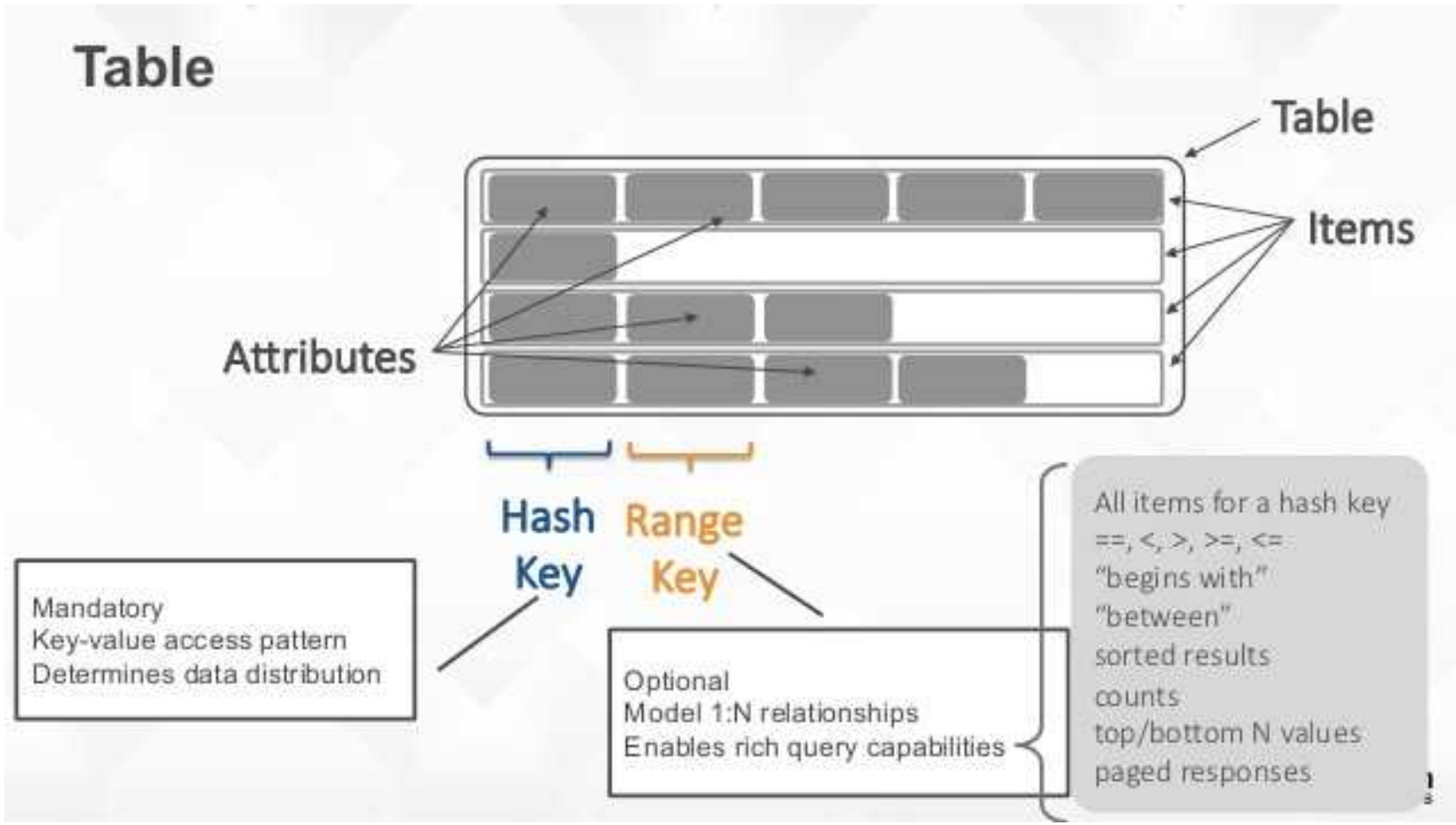


Features

- Flexible
  - Does not have a fixed schema
- Efficient Indexing
  - Every item identified by a primary key
- Strong consistency
  - Implemented with Atomic Counters
  - Disk-only writes
- Secure with Monitoring
  - AWS Identity and Access Management
  - CloudWatch for monitoring request throughput, latency and resource consumption
- Amazon Elastic MapReduce Integration
- Amazon Redshift Integration

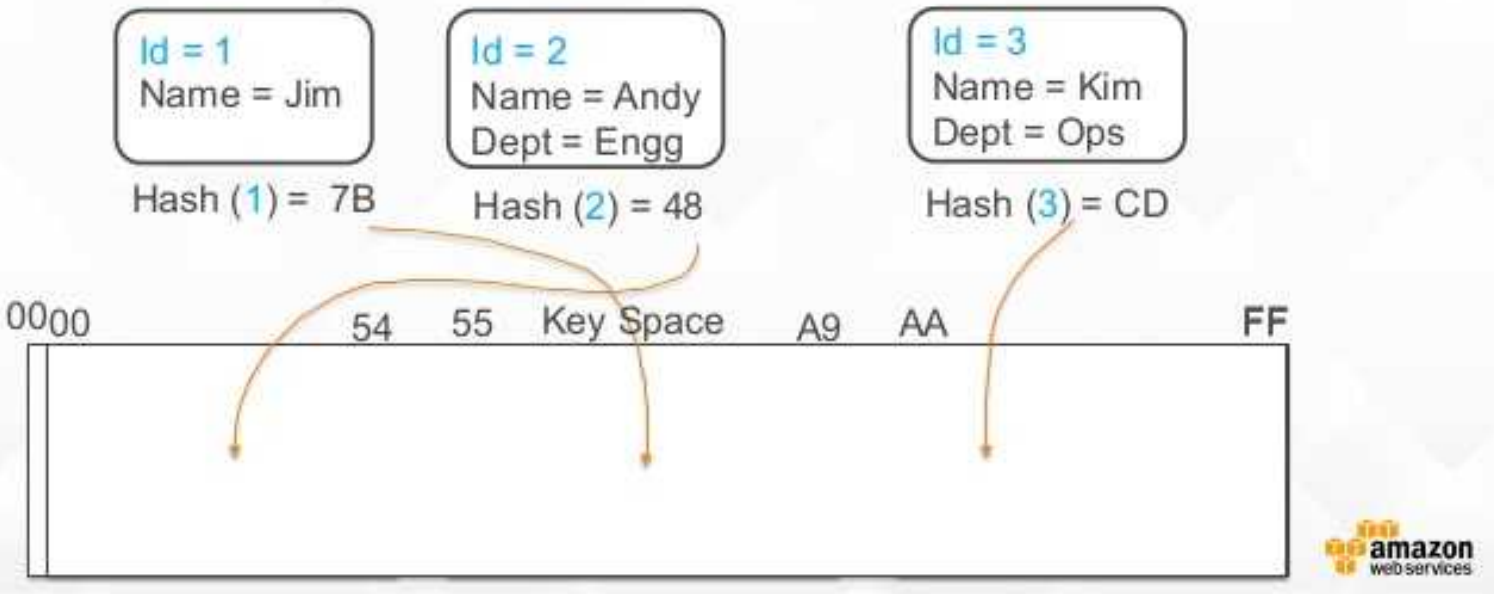


Table

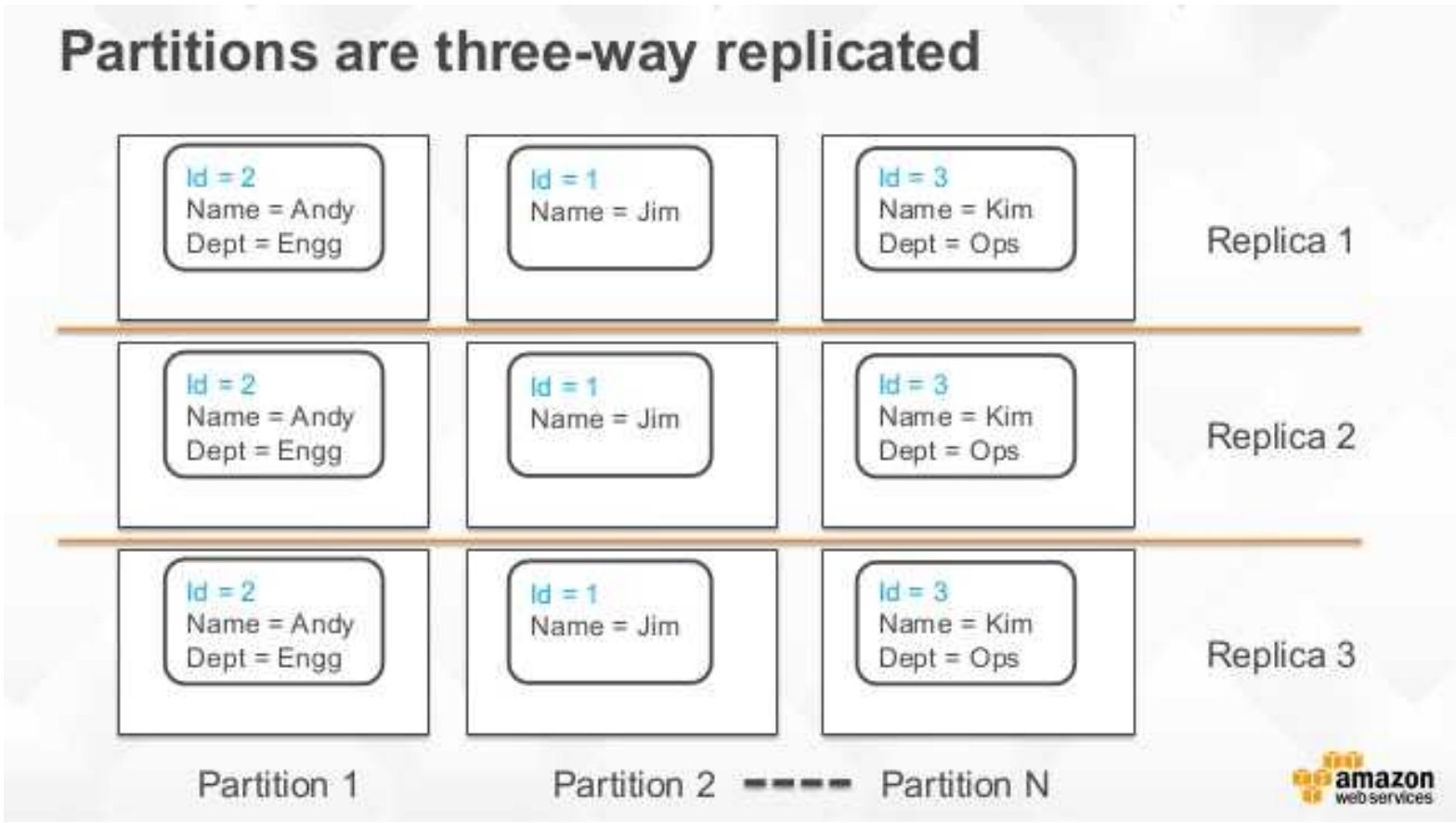


Hash table

- Hash key uniquely identifies an item
- Hash key is used for building an unordered hash index
- Table can be partitioned for scale

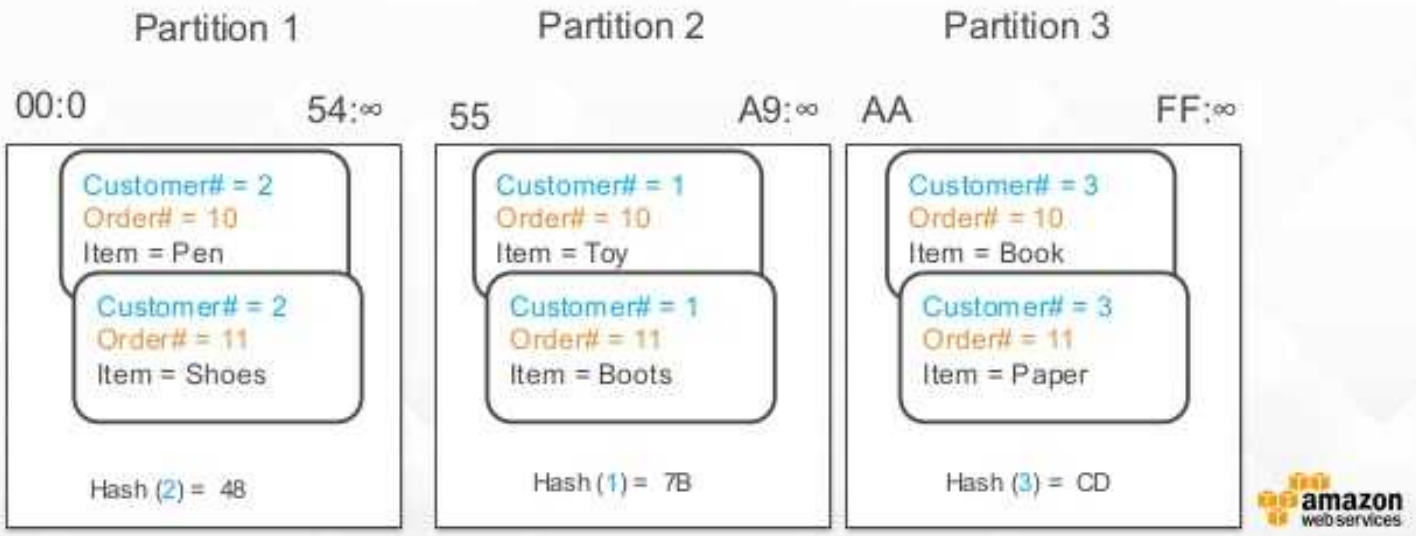


Partitions are three-way replicated

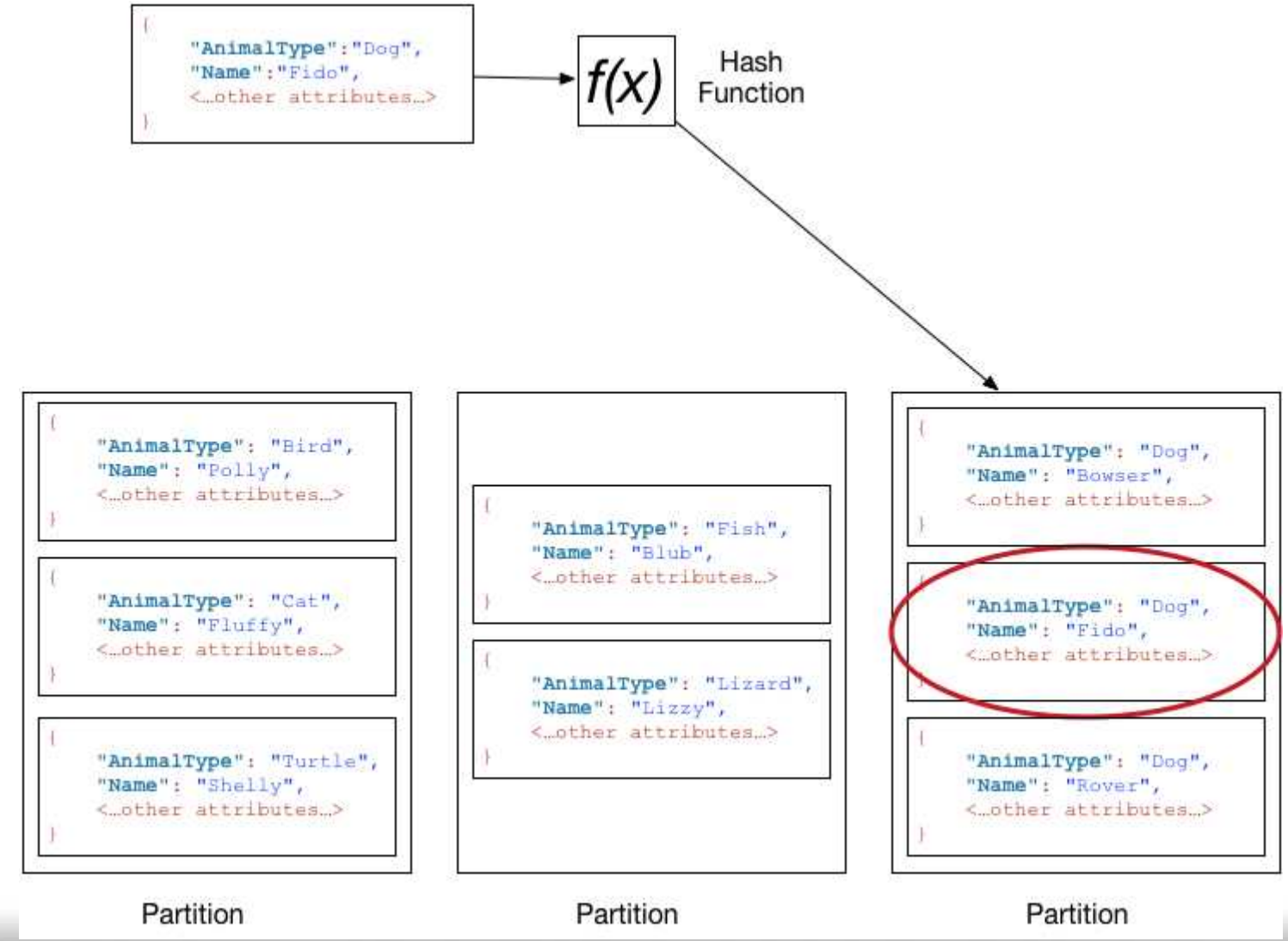


Hash-range table

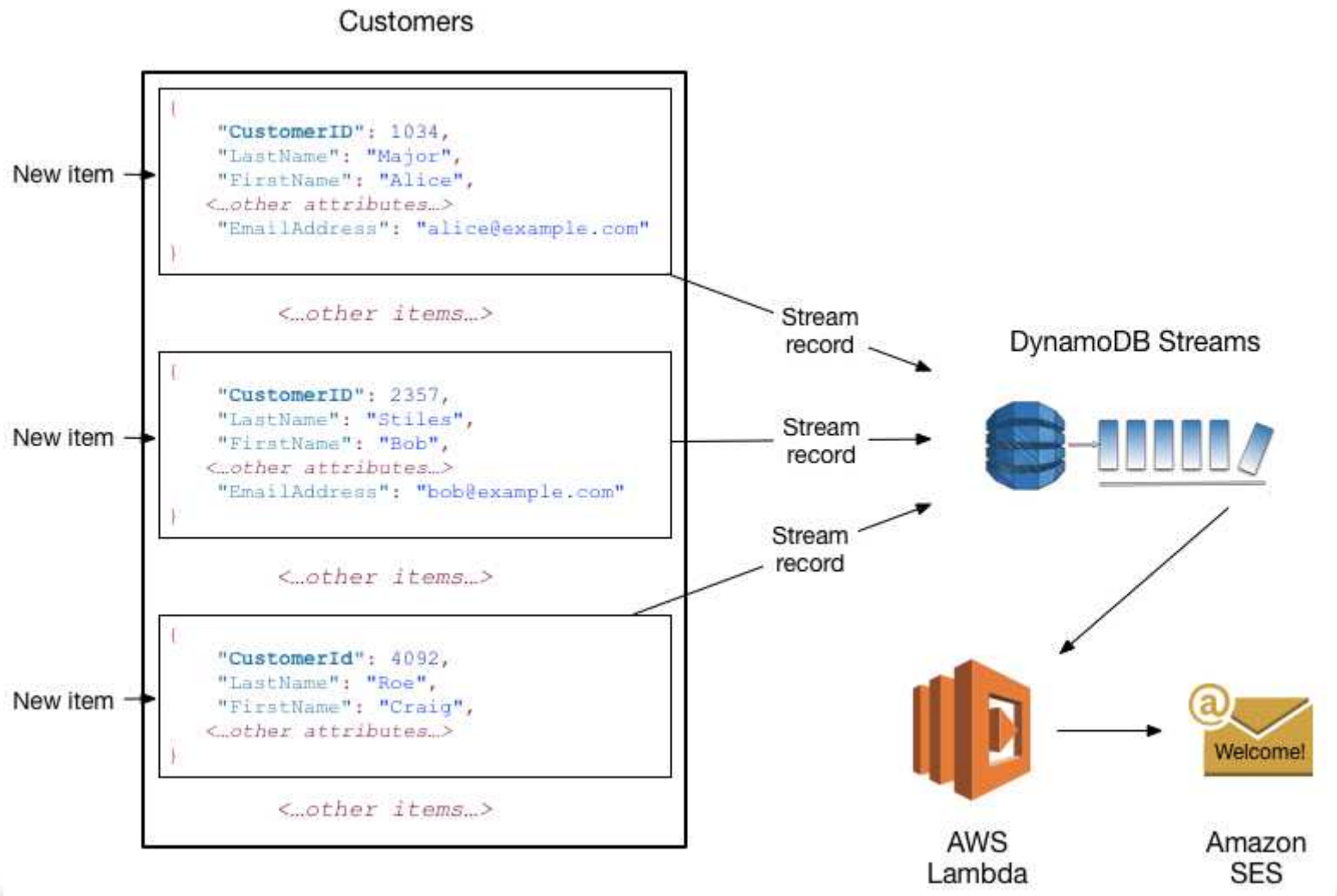
- Hash key and range key together uniquely identify an Item
- Within unordered hash index, data is sorted by the range key
- No limit on the number of items (∞) per hash key
  - Except if you have local secondary indexes



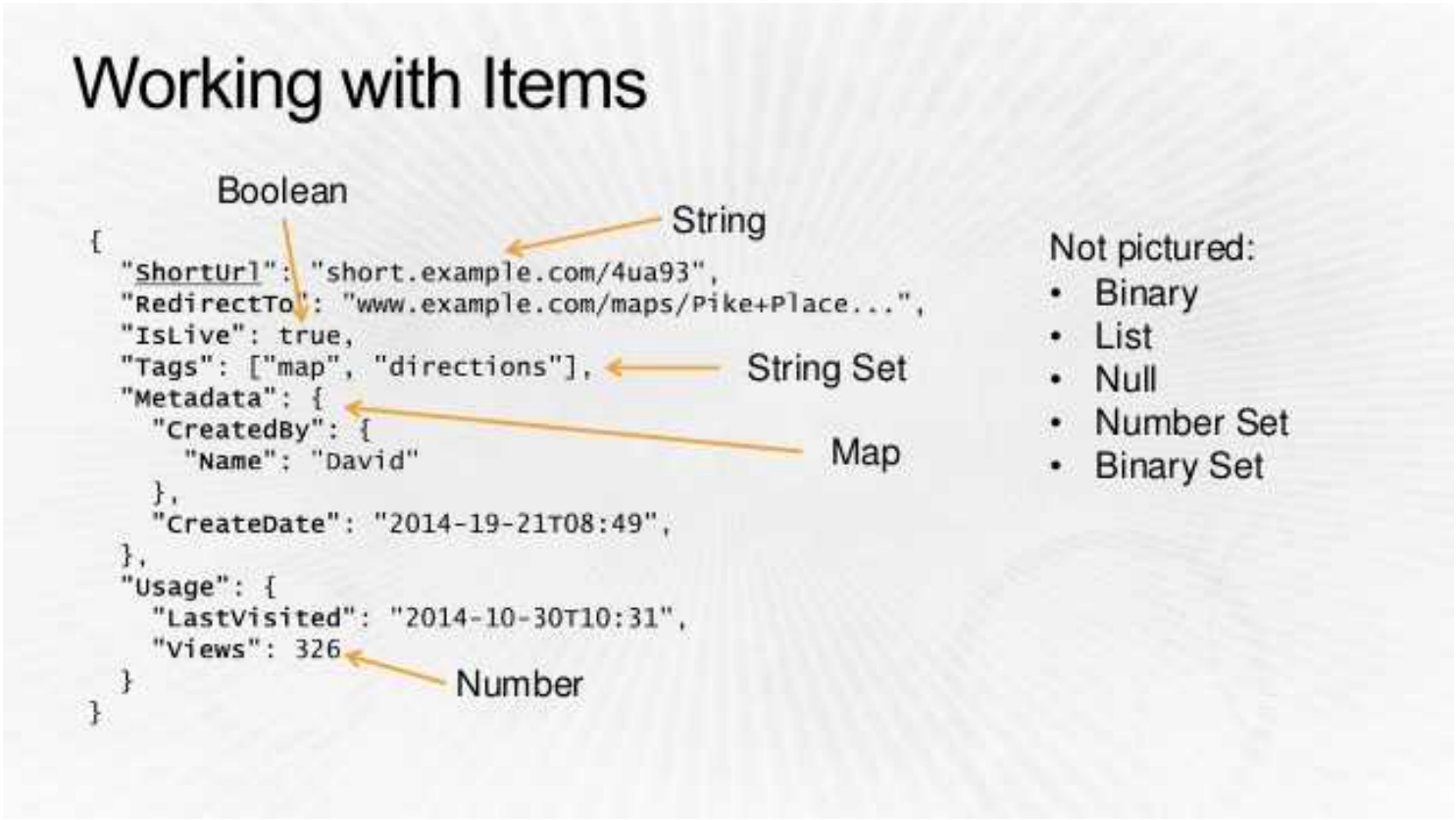
DynamoDB



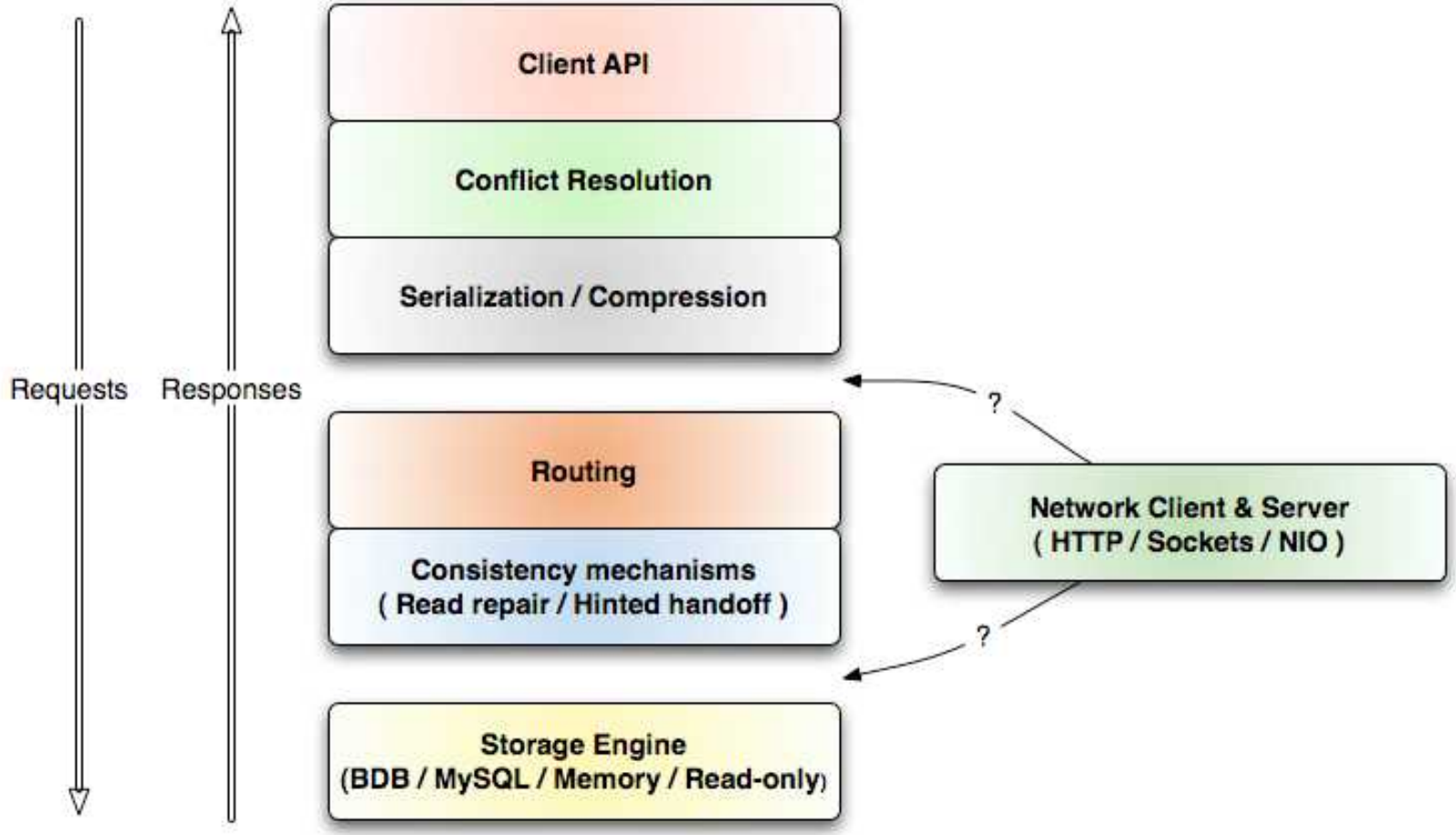
DynamoDB Components



DynamoDB



Logical Architecture



Voldemort Key-Value Distributed Data Store

- based on Amazon's DynamoDB
- high performance and horizontal scalability, as well as on providing replication for high availability and sharding for improving latency (response time) of read and write requests.
- to distribute the key-value pairs among the nodes of a distributed cluster - consistent hashing.

Voldemort Key-Value Distributed Data Store

Simple basic operations

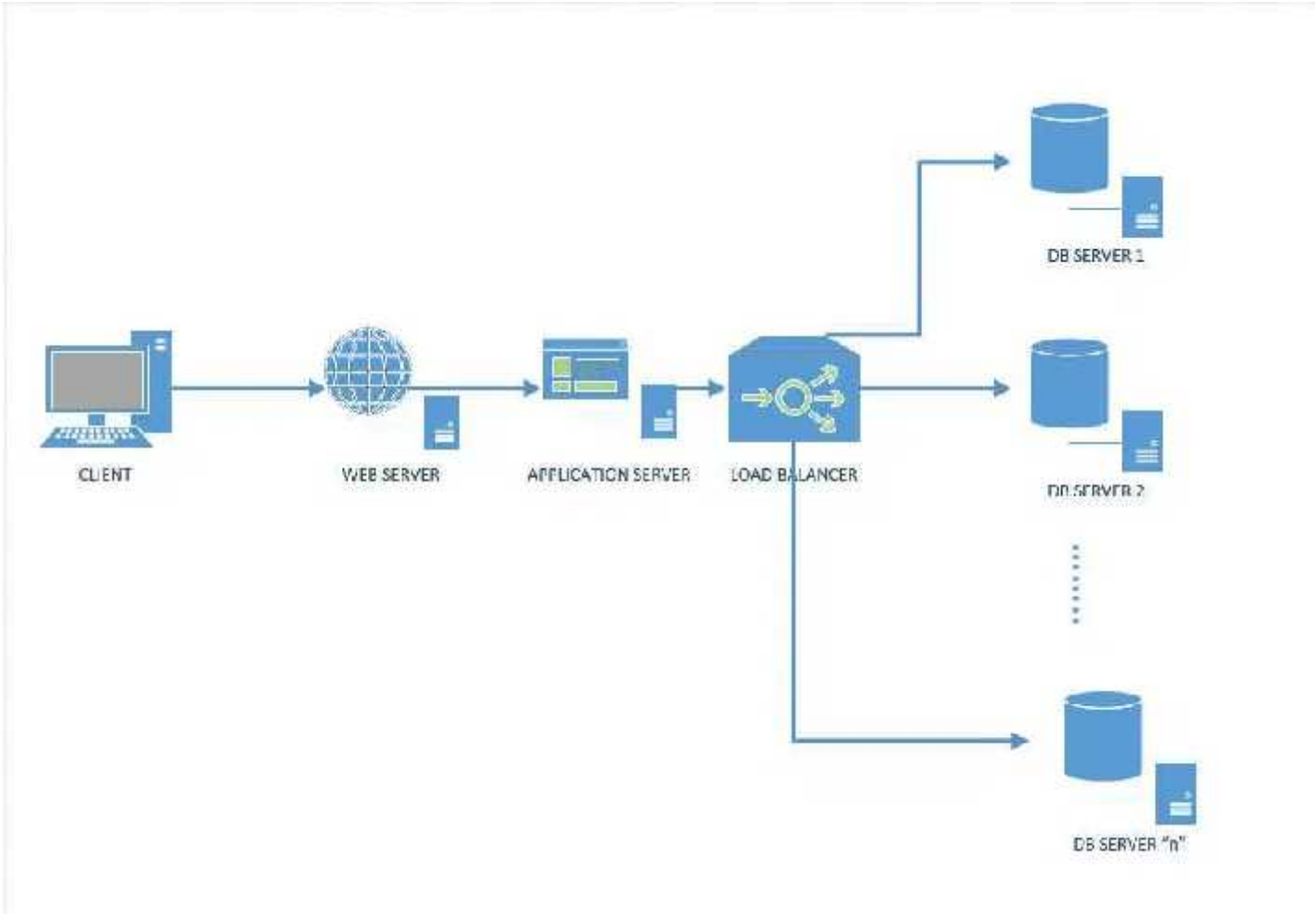
- A collection of (key, value) pairs is kept in a Voldemort store. Assume the store is called s.
- Basic interface for data storage and retrieval is very simple.
- Three operations: get, put, and delete.
  - s.put(k, v) inserts an item as a key-value pair with key k and value v.
  - s.delete(k) deletes the item whose key is k from the store, and
  - v = s.get(k) retrieves the value v associated with key k.
- At the basic storage level, both keys and values are arrays of bytes (strings).

Voldemort Key-Value Distributed Data Store

High-level formatted data values

- The values v in the (k, v) items can be specified in JSON (JavaScript Object Notation), and the system will convert between JSON and the internal storage format.
- Application provides the conversion (also known as serialization) between the user format and the storage format as a Serializer.
- The Serializer class must be provided by the user and will include operations
  - to convert the user format into a string of bytes for storage as a value,
  - to convert back a string (array of bytes) retrieved via s.get(k) into the user format.
- Voldemort has some built-in serializers for formats other than JSON.

Why do we need Consistent Hashing ?



Why do we need Consistent Hashing ?

Our goal is to design a database storage system such that:

- We should be able to distribute the incoming queries uniformly among the set of "n" database servers
- We should be able to dynamically add or remove a database server
- When we add/remove a database server, we need to move the minimal amount of data between the servers

Why do we need Consistent Hashing ?

A simple approach is as follows:

Generate a hash of the key from the incoming data :

" hashValue = HashFunction(Key)"

Figure out the server to send the data to by taking the modulo ("%") of the hashValue using the number of current db servers, n : "serverIndex = hashValue % n"

Let's walk through a simple example.

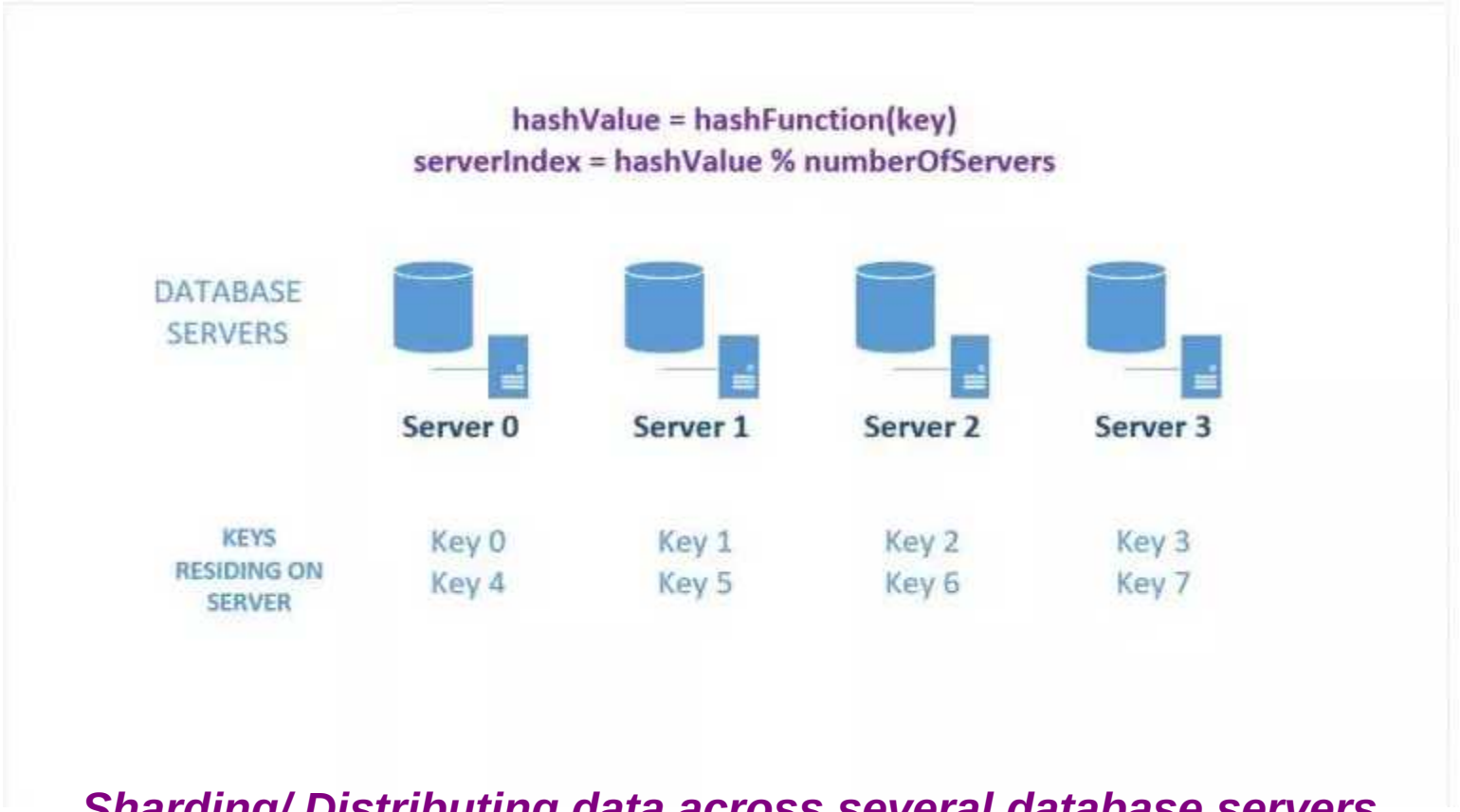
Imagine we have 4 database servers

Imagine our hashFunction returns a value from 0 to 7

We'll assume that "key0" when passed through our hashFunction, generates a hashvalue or 0, "key1" generates 1 and so on.

The serverIndex for "key0" is 0, "key1" is 1 and so on.

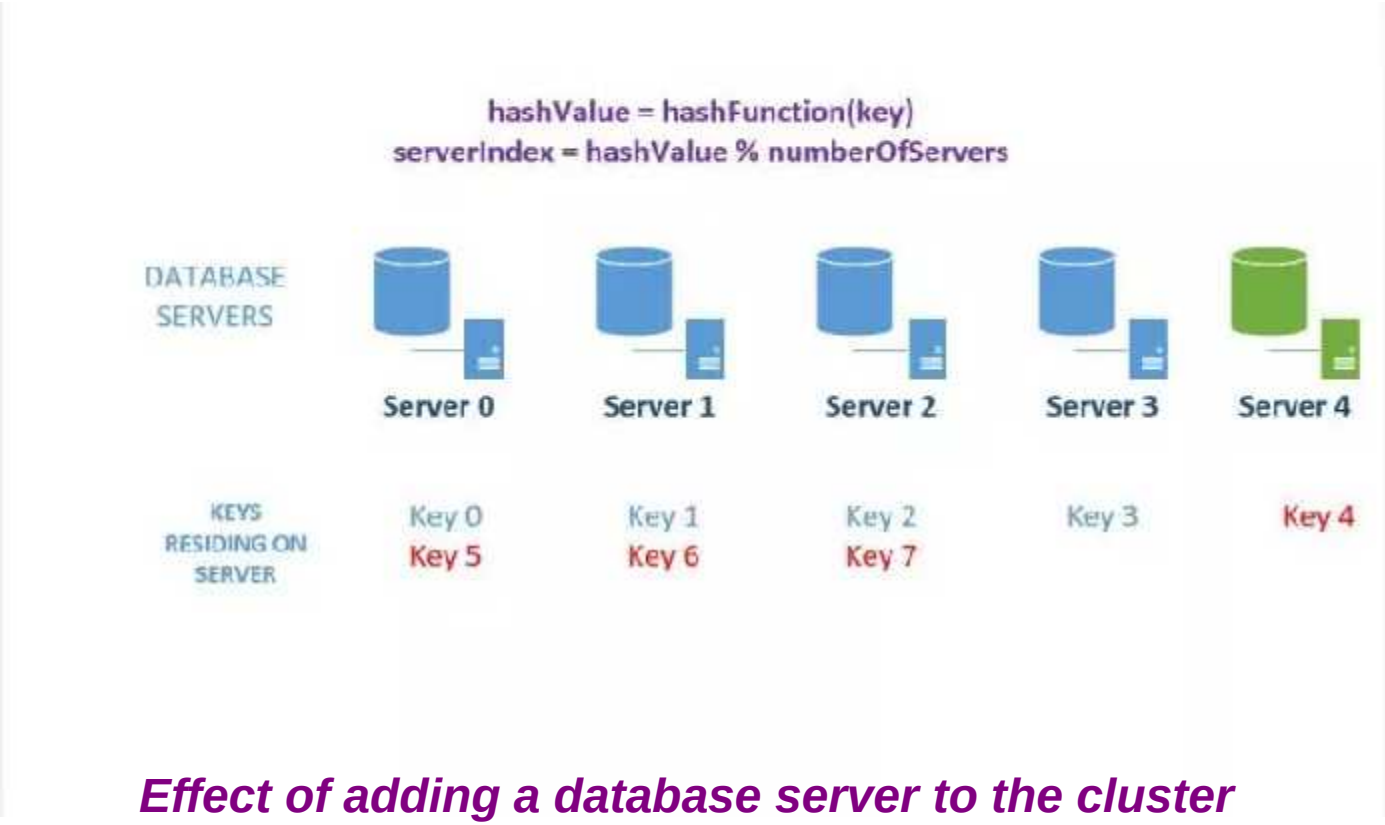
Why do we need Consistent Hashing ?



Sharding/ Distributing data across several database servers

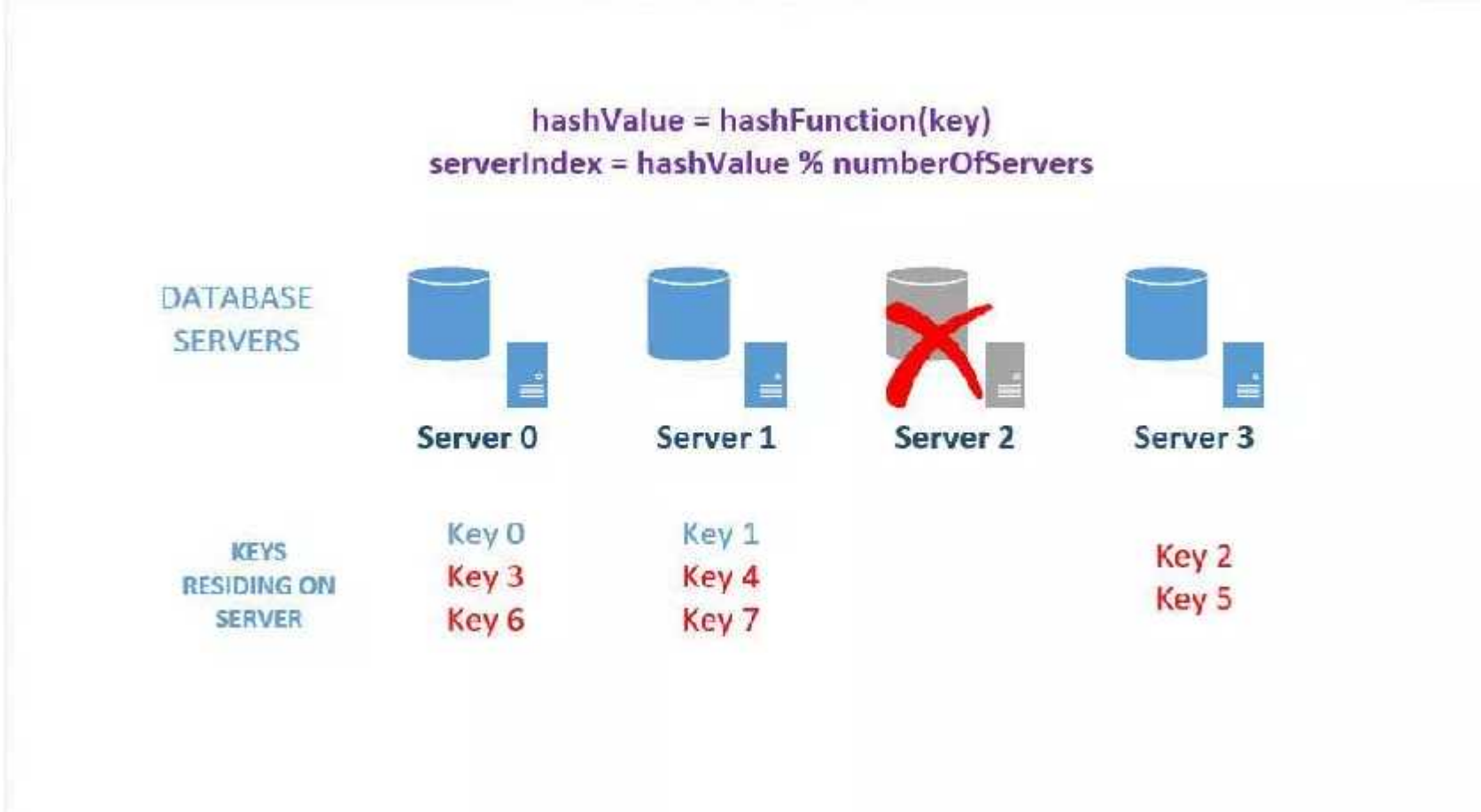
Two major drawbacks : Horizontal Scalability and Non-Uniform data distribution across servers.

Horizontal Scalability



Effect of adding a database server to the cluster

Horizontal Scalability



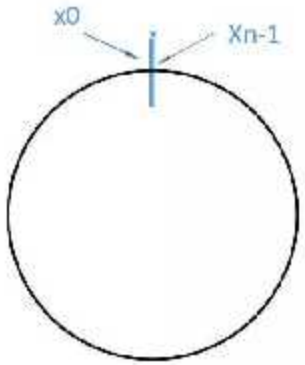
Effect of removing a server from the database cluster

How does Consistent Hashing Work ?

1. Creating the Hash Key Space: Consider we have a hash function that generates integer hash values in the range [0, 2^32-1)

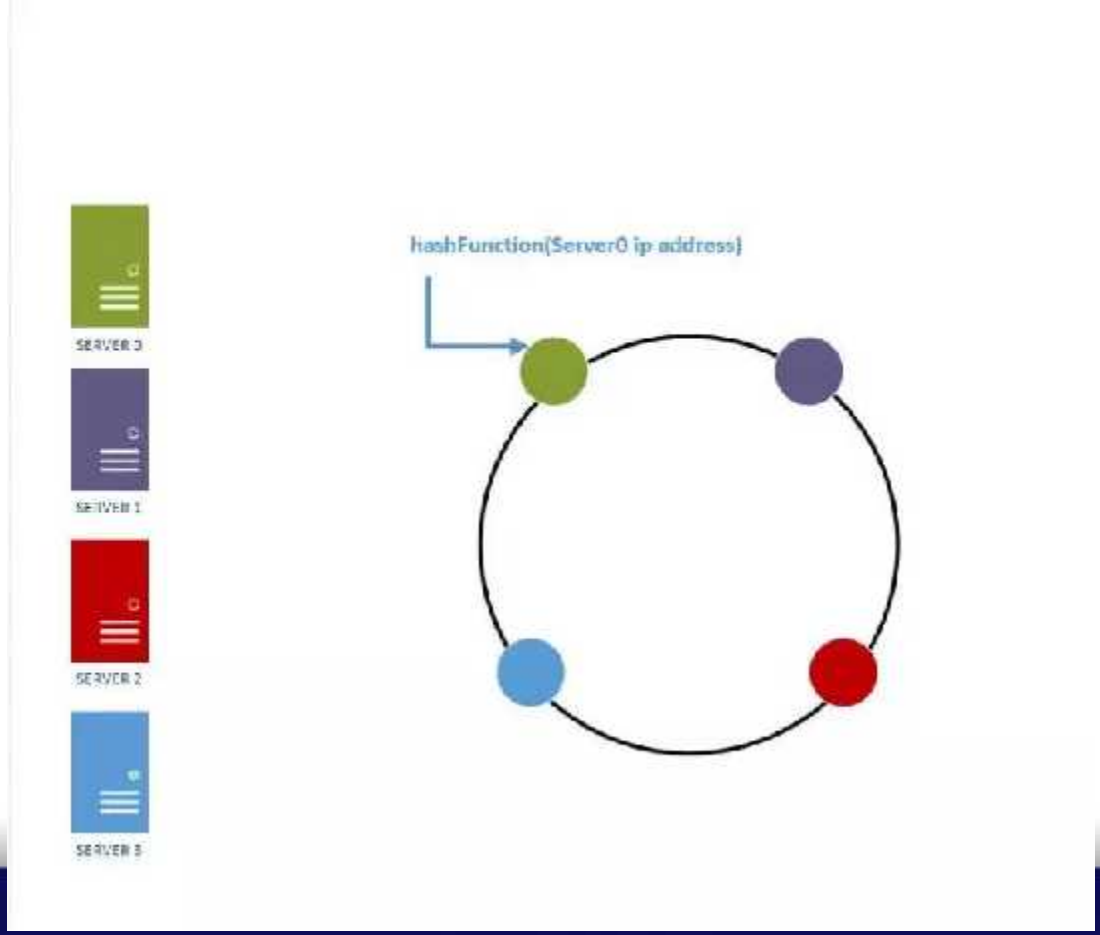
   We can represent this as an array of integers with 2^32 -1 slots. We'll call the first slot x0 and the last slot xn – 1. Figure out the server to send the data to by taking the modulo ("%") of the hashValue using the number of current db servers, n : "serverIndex = hashValue % n"

2. Representing the hashSpace as a Ring: Imagine that these integers generated in step # 2 is placed on a ring such that the last value wraps around.



How does Consistent Hashing Work ?

3. Placing DB Servers in Key Space (HashRing): Using the hash function, we map each db server to a specific place on the ring. For example, if we have 4 servers, we can use a hash of their IP addressed to map them to different integers using the hash function.



How does Consistent Hashing Work ?

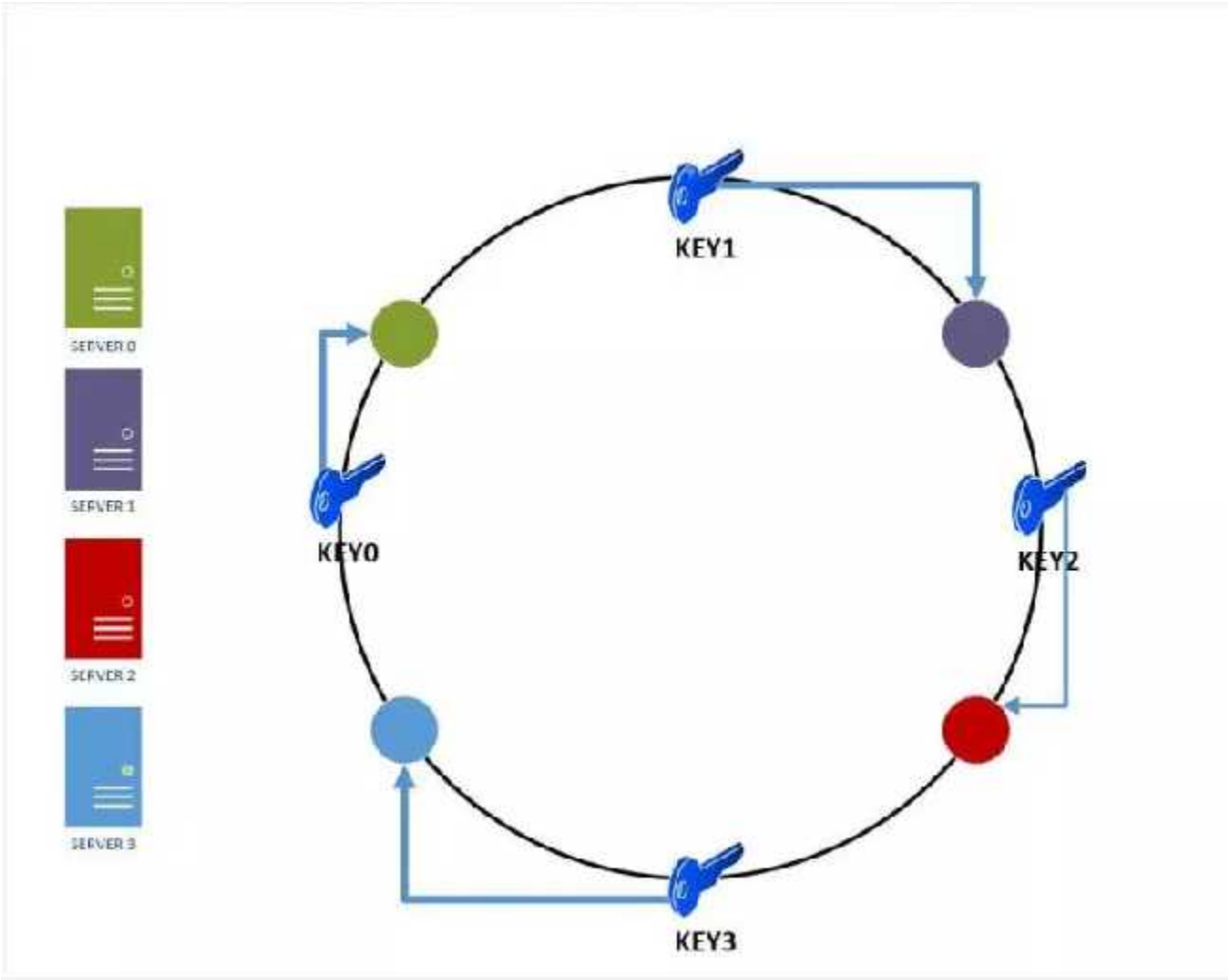
4. Determining Placement of Keys on Servers: To find which database server an incoming key resides on (either to insert it or query for it), we do the following:

   Run the key through the same hash function we used to determine db server placement on the ring.

   After hashing the key, we'll get an integer value which will be contained in the hash space, i.e., it can be mapped to some postion in the hash ring. There can be two cases:

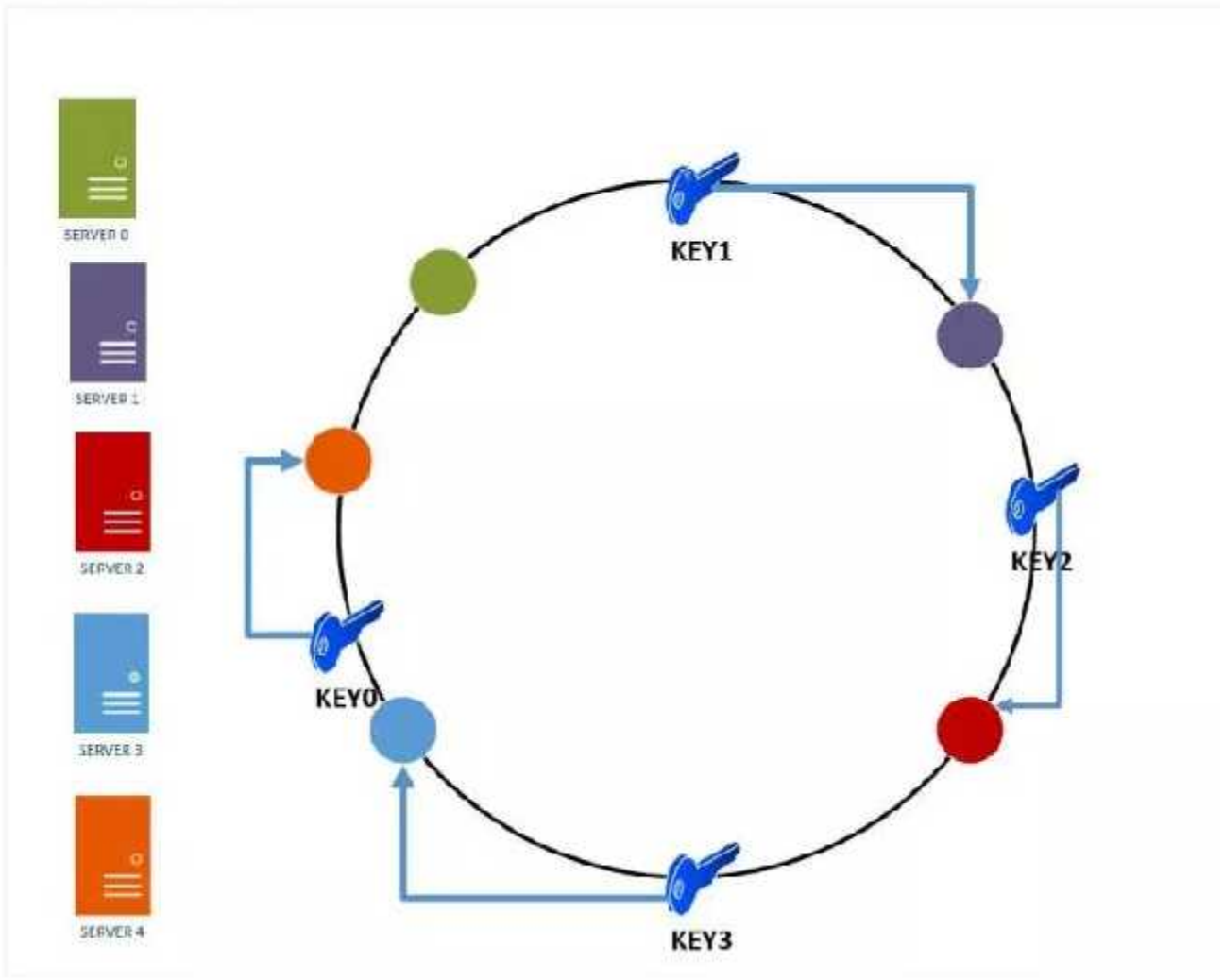
   - The hash value maps to a place on the ring which does not have a db server. In this case, we travel clockwise on the ring from the point where the key mapped to untill we find the first db server. Once we find the first db server travelling clockwise on the ring, we insert the key there. The same logic would apply while trying to find a key in the ring.
   - The hash value of the key maps directly onto the same hash vale of a db server – in which case we place it on that server.

How does Consistent Hashing Work ?

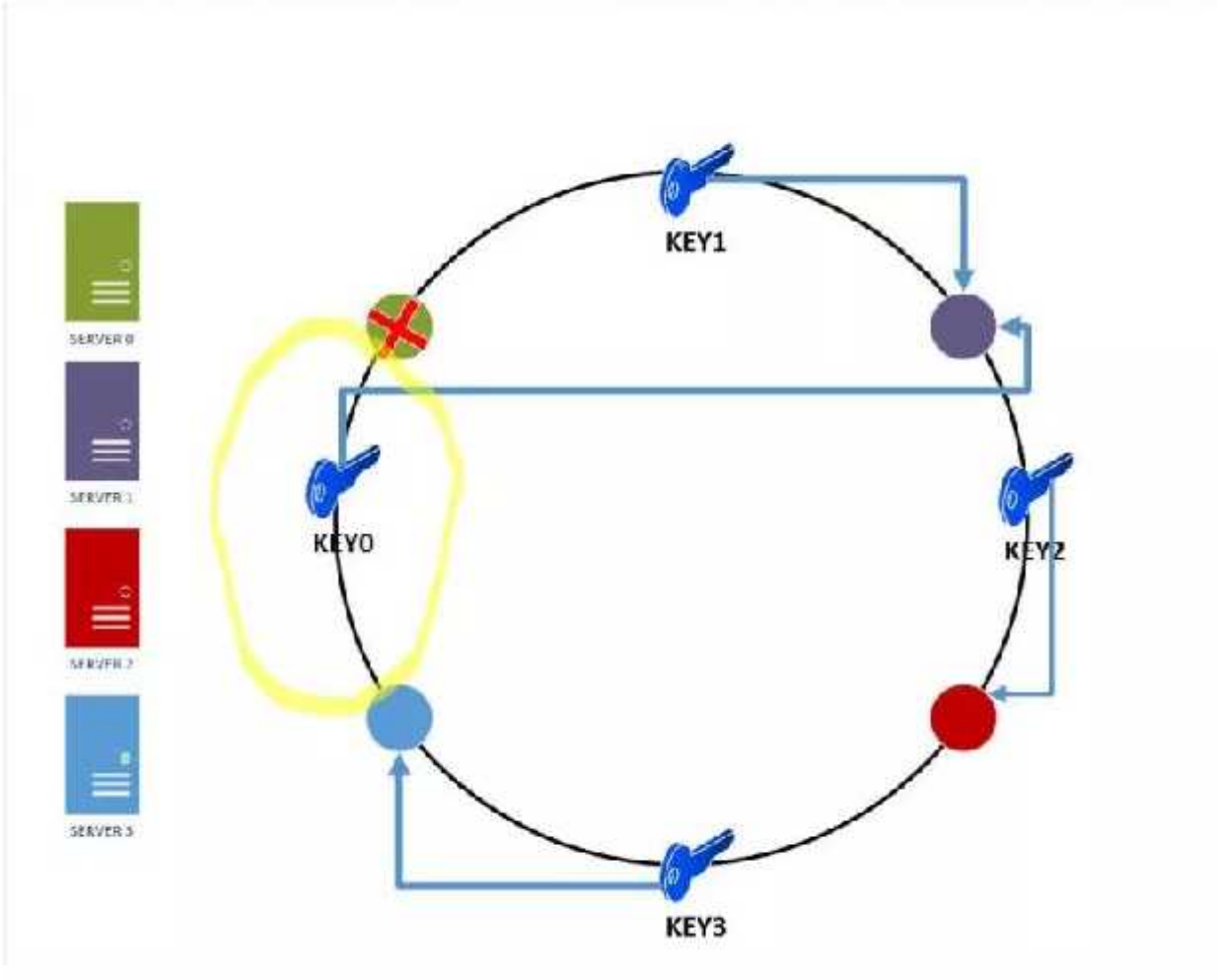


Key placements on database servers in a hash ring

5. Adding a server to the Ring



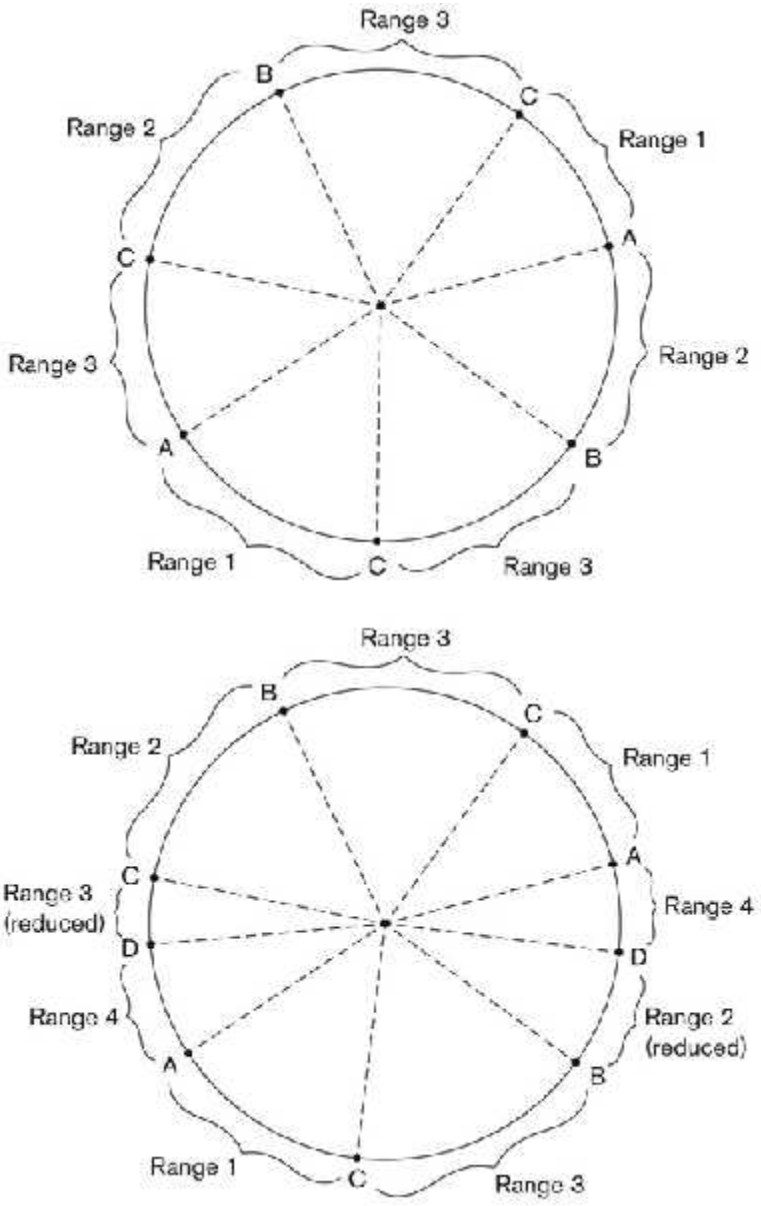
6. Removing a server from the ring



Example of consistent hashing.

(a) Ring having three nodes A, B, and C, with C having greater capacity. The h(K) values that map to the circle points in range 1 have their (k, v) items stored in node A, range 2 in node B, range 3 in node C.

(b) Adding a node D to the ring. Items in range 4 are moved to the node D from node B (range 2 is reduced) and node C (range 3 is reduced).



Consistency and versioning

- Voldemort uses a method similar to the one developed for DynamoDB for consistency in the presence of replicas.
- Basically, concurrent write operations are allowed by different processes so there could exist two or more different values associated with the same key at different nodes when items are replicated.
- Concurrent writes are allowed, but each write is associated with a vector clock value.
- Consistency is achieved when the item is read by using a technique known as versioning and read repair.
- When a read occurs, it is possible that different versions of the same value (associated with the same key) are read from different nodes. If the system can reconcile to a single final value, it will pass that value to the read; otherwise, more than one version can be passed back to the application, which will reconcile the various versions into one version based on the application semantics and give this reconciled value back to the nodes.

Thank you