



# Introduction to NoSQL Databases

CE384: Database Design  
Maryam Ramezani  
Sharif University of Technology  
[maryam.ramezani@sharif.edu](mailto:maryam.ramezani@sharif.edu)



# Introduction

# Parallel Databases and Data Stores

- Relational Databases – mainstay of business
- Web-based applications caused spikes
  - Especially true for public-facing e-Commerce sites
- Many application servers, one database
  - Easy to parallelize application servers to 1000s of servers, harder to parallelize databases to same scale
  - First solution: memcache or other caching mechanisms to reduce database access

# Scaling Up

- What if the dataset is huge, and very high number of transactions per second
- Use multiple servers to host database
- Parallel databases have been around for a while
  - But expensive, and designed for decision support not OLTP

# Scaling RDBMS – Master/Slave

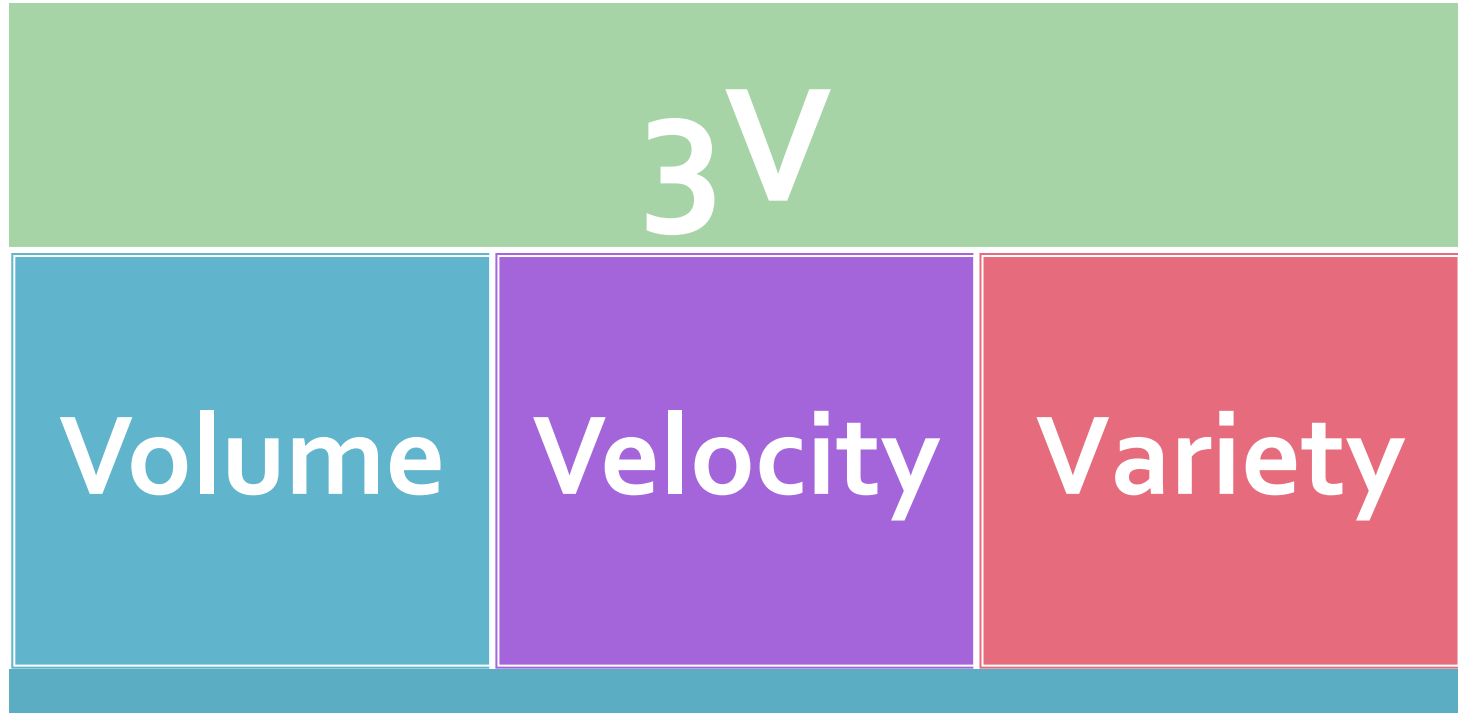
## ■ Master-Slave

- ❑ All writes are written to the master. All reads performed against the replicated slave databases
- ❑ Good for mostly read, very few update applications
- ❑ Critical reads may be incorrect as writes may not have been propagated down
- ❑ Large data sets can pose problems as master needs to duplicate data to slaves

# Scaling RDBMS - Partitioning

- Partitioning
  - Divide the database across many machines
    - E.g. hash or range partitioning
- Handled transparently by parallel databases
  - but they are expensive
- “Sharding”
  - Divide data amongst many cheap databases (MySQL/PostgreSQL)
  - Manage parallel access in the application
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware

# Big Data



# What is NoSQL?

- Stands for **Not Only SQL**
- Class of non-relational data storage systems
  - E.g. BigTable, Dynamo, PNUTS/Sherpa, ..
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties (will talk about the CAP theorem)
- Not a backlash/rebellion against RDBMS
- SQL is a rich query language that cannot be rivaled by the current list of NoSQL offerings



# Why Now?

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Explosion of storage needs in large web sites such as Google, Yahoo
  - ▣ Much of the data is not files
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Shift to dynamically-typed data with frequent schema changes
- Open-source community

# Distributed Key-Value Data Stores

- Distributed key-value data storage systems allow key-value pairs to be stored (and retrieved on key) in a massively parallel system
  - E.g. Google BigTable, Yahoo! Sherpa/PNUTS, Amazon Dynamo, ..
- Partitioning, high availability etc completely transparent to application
- Sharding systems and key-value stores don't support many relational features
  - No join operations (except within partition)
  - No referential integrity constraints across partitions
  - etc.

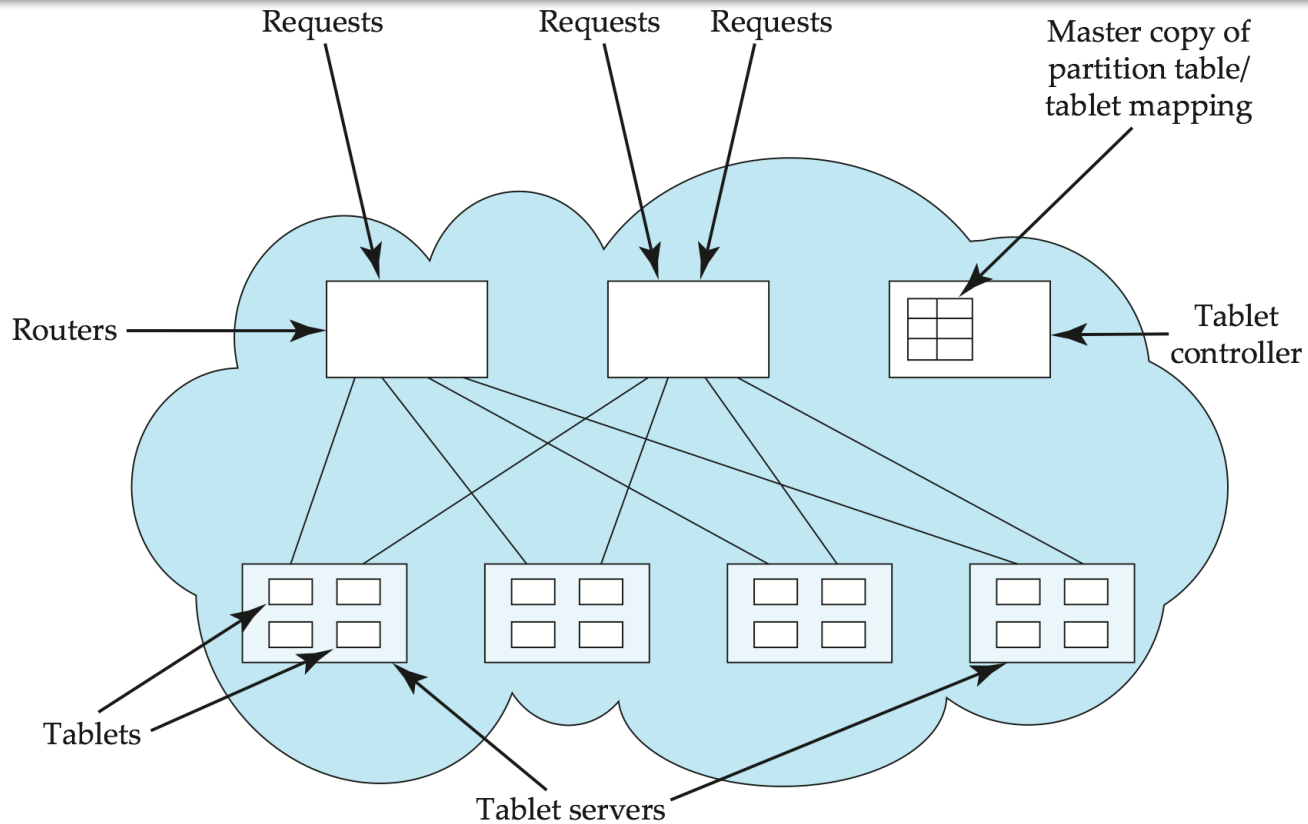
# Typical NoSQL API

- Basic API access:
  - `get(key)` -- Extract the value given a key
  - `put(key, value)` -- Create or update the value given its key
  - `delete(key)` -- Remove the key and its associated value
  - `execute(key, operation, parameters)` -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc).

# Flexible Data Model

ColumnFamily: Rockets		
Key	Value	
1	<b>Name</b>	<b>Value</b>
	name	Rocket-Powered Roller Skates
	toon	Ready, Set, Zoom
	inventoryQty	5
	brakes	false
2	<b>Name</b>	<b>Value</b>
	name	Little Giant Do-It-Yourself Rocket-Sled Kit
	toon	Beep Prepared
	inventoryQty	4
	brakes	false
3	<b>Name</b>	<b>Value</b>
	name	Acme Jet Propelled Unicycle
	toon	Hot Rod and Reel
	inventoryQty	1
	wheels	1

# PNUTS Data Storage Architecture



# CAP Theorem

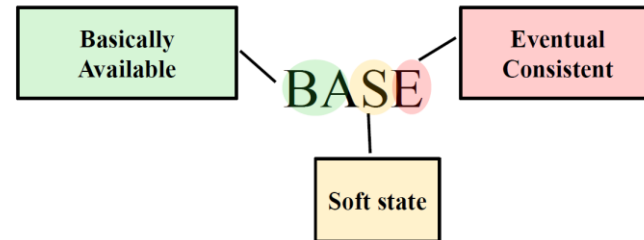
- Three properties of a system
  - Consistency (all copies have same value)
  - Availability (system can run even if parts have failed)
  - Partitions (network can break into two or more parts, each with active systems that can't talk to other parts)
- Brewer's CAP "Theorem": You can have at most two of these three properties for any system
- Very large systems will partition at some point
  - ➔ Choose one of consistency or availability
  - Traditional database choose consistency
  - Most Web applications choose availability
    - Except for specific parts such as order processing

# Availability

- Traditionally, thought of as the server/process available five 9's (99.999 %).
- However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.
  - Want a system that is resilient in the face of network disruption

# Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency), as opposed to ACID
  - ❑ Soft state: copies of a data item may be inconsistent
  - ❑ Eventually Consistent – copies becomes consistent at some later time if there are no more updates to that data item





# Common Advantages

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - When data is written, the latest version is on at least one node and then replicated to other nodes
  - Down nodes easily replaced
  - No single point of failure
- Easy to distribute
- Don't require a schema

# What does NoSQL Not Provide?

- Joins
- Group by
  - But PNUTS provides interesting materialized view approach to joins/aggregation.
- ACID transactions
- SQL
- Integration with applications that are based on SQL

# Should I be using NoSQL Databases?

- NoSQL Data storage systems makes sense for applications that need to deal with very very large semi-structured data
  - Log Analysis
  - Social Networking Feeds
- Most of us work on organizational databases, which are not that large and have low update/query rates
  - regular relational databases are THE correct solution for such applications

# Data Partitioning Algorithms

# Data Partitioning

- Simple hash based partitioning
- Range based partitioning :
- Consistent Hashing

# Simple hash-based partitioning

- General procedure for hash-based partitioning is for a given data element (a row or anything that is taken as an entity) , come up with a key and applying a hash function to map it to a number. Then do a modulo on hash function value and the number of servers. The resulting value would be in the range 1 to the number of servers added.
- $(\text{hash}(\text{key}) \% \text{number of servers}) = \text{a number (in the range of servers)}$
- Problem: Major problem with this simple hash-based partitioning is that when nodes come down, the data on that node needs to be moved around to other nodes in the cluster. Also, the count of nodes changes when nodes scale up or scale down and hence the re mapping of all keys is required, which involves huge data movement. Needless to say, it is most expensive operation to handle without making availability trait to suffer.

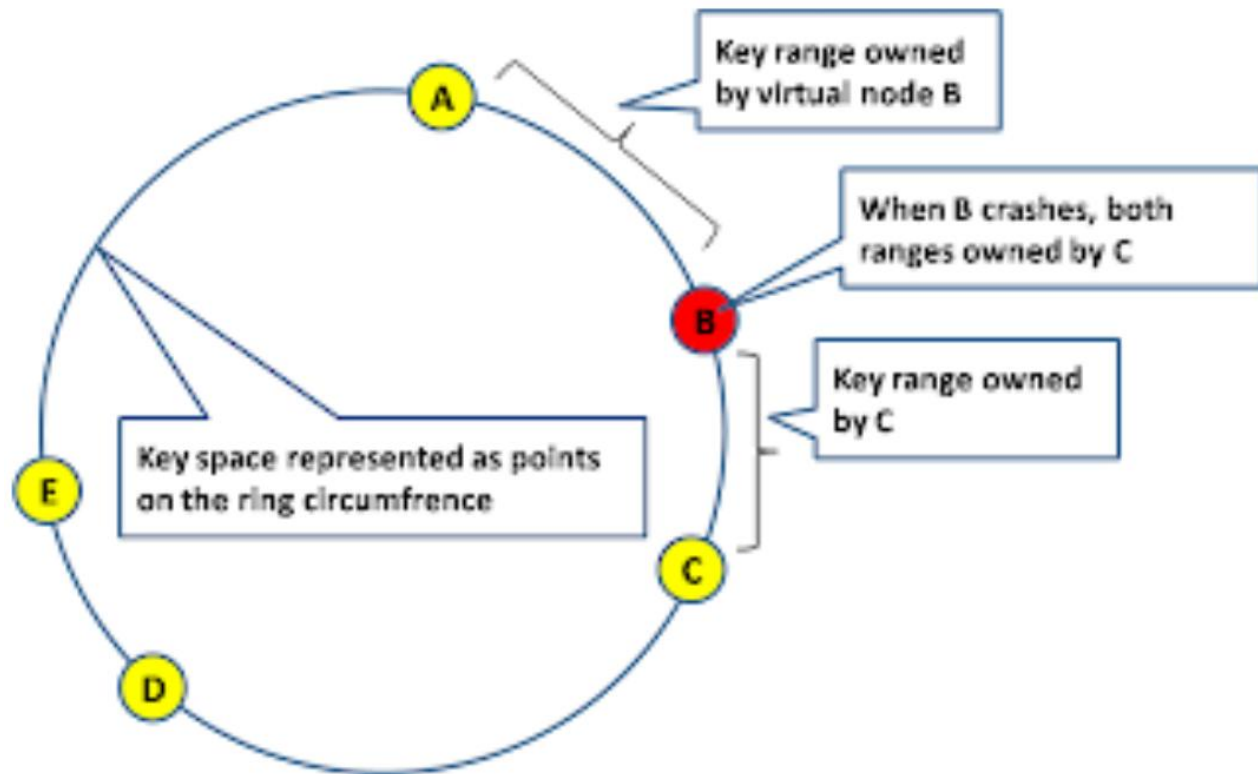
# Range based partitioning

- In this partitioning scheme, certain ranges are allocated to nodes, and if hash or any predefined function on the chosen key falls into a certain range, then corresponding server stores the data.
- Problem: In this partitioning technique there is possibility of data being skewed on some nodes vs the others. Again, the success of this depends on how well the function to decide the range is selected. It also exhibits the need to rebalance or re distribute data in the cluster in case of adding nodes and decommissioning nodes.

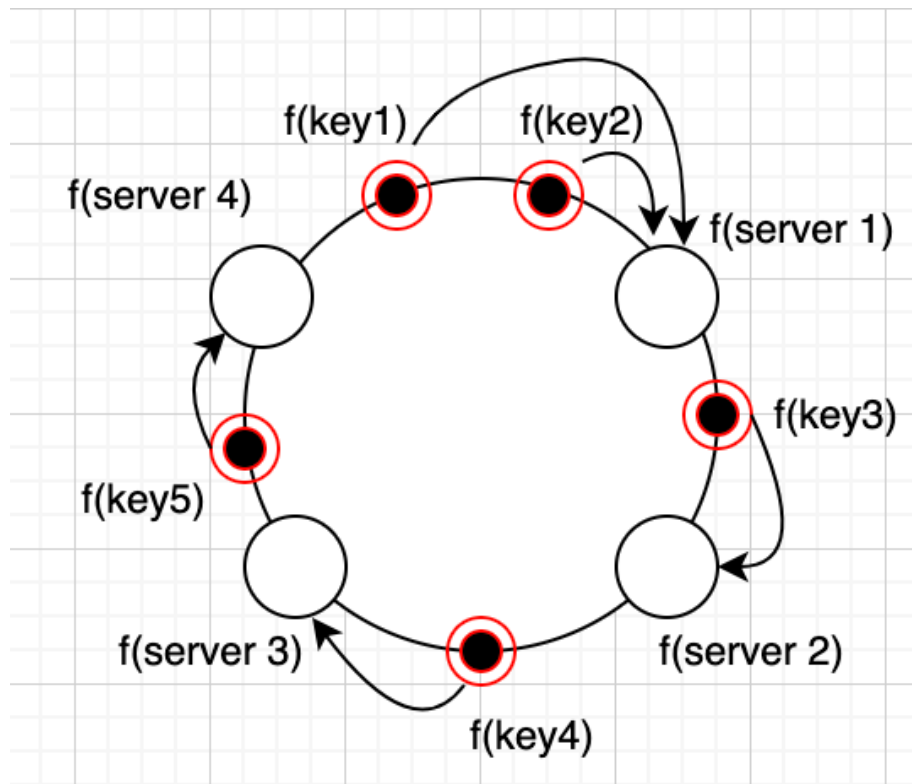
# Consistent Hashing



# Server lookup

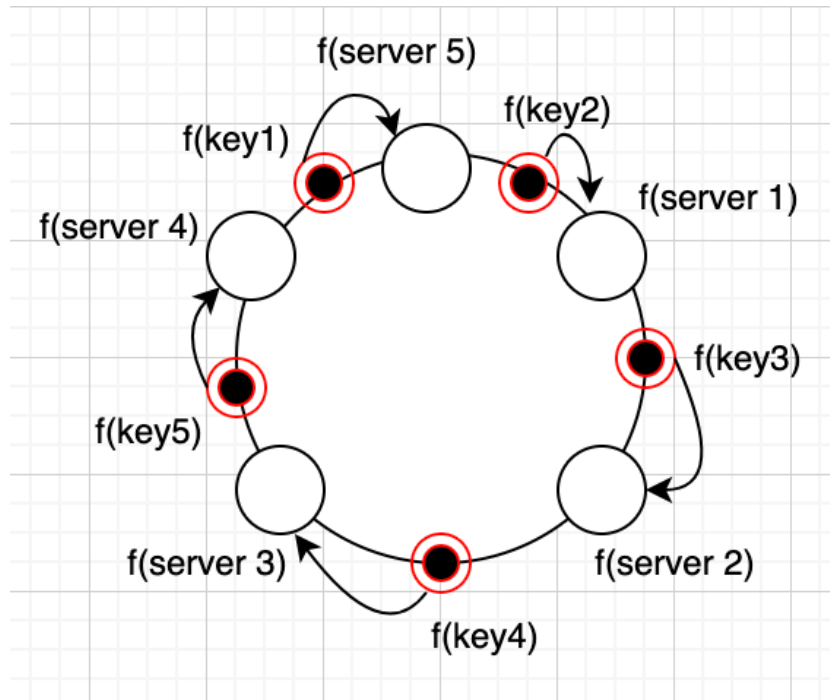


# Server lookup



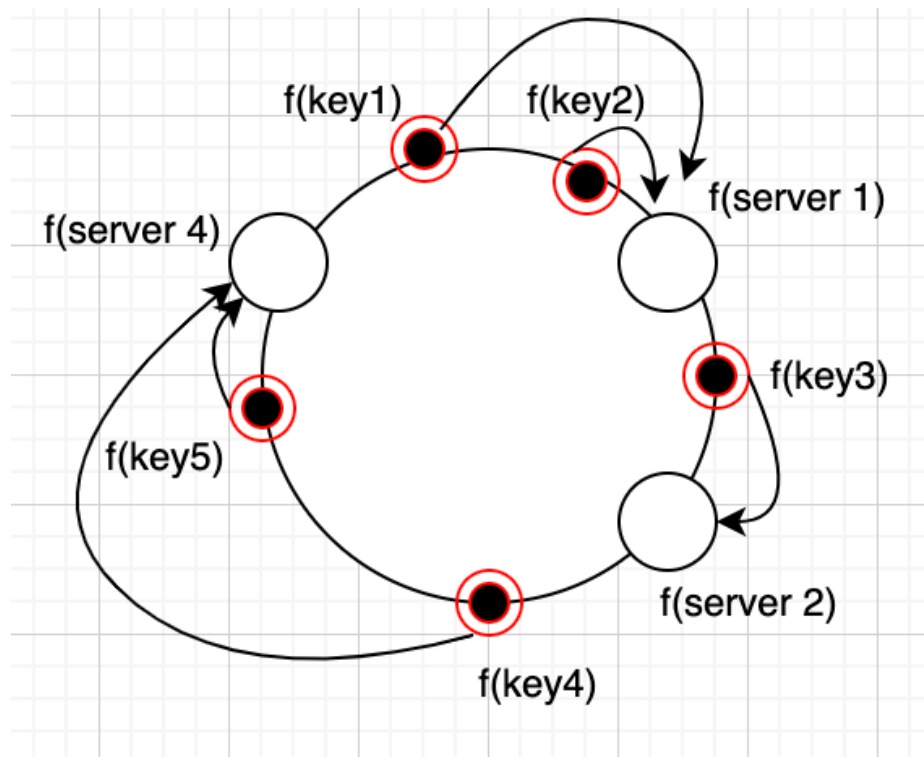
# Add a Server

- if server 5 is added, only keys between server 4 and server 5 are remapped. In this ex: only key1 is remapped from server 1 to server5



# Remove a Server

- if server 3 is removed, all the keys mapped to server3 will be remapped to next server in ring. In this ex: key4 will be remapped to server4

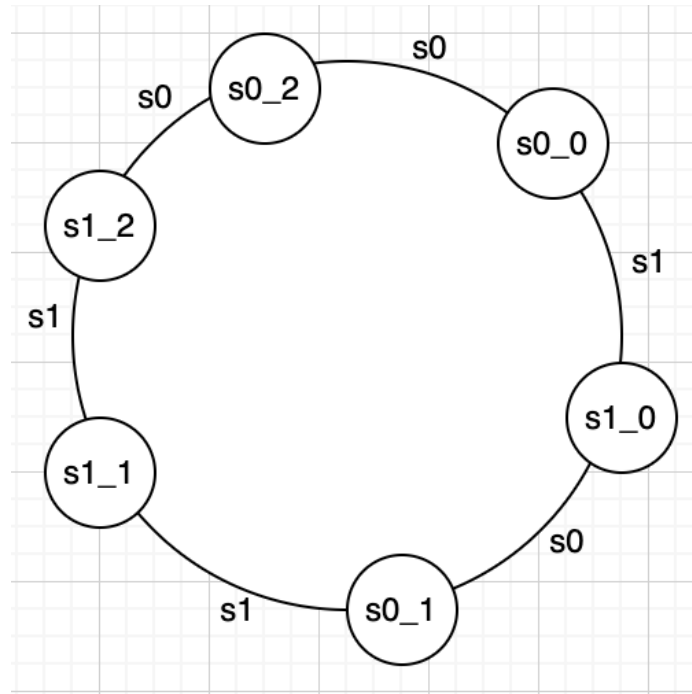


# Problem and Solution

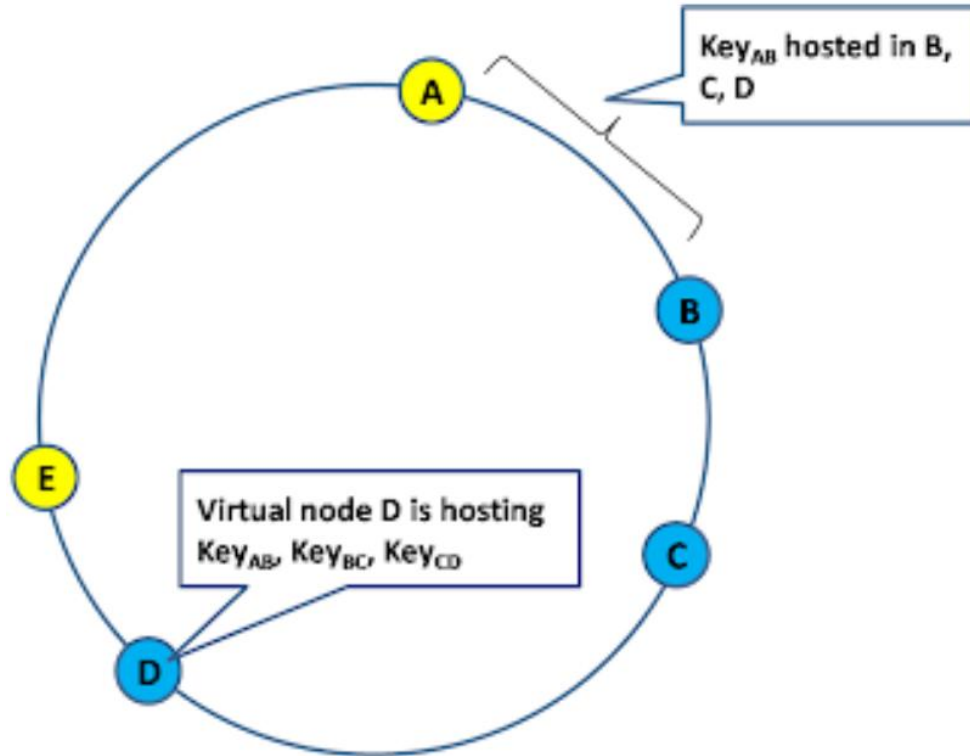
- Above approach works perfectly under the assumption that partition size is uniform and keys are uniformly distributed onto the servers. But in real world -
  - It is not possible to have uniform size of partitions on the ring. A partition is the hash space between two adjacent servers. It is possible to have very small/ very large partition between two servers.
  - It is possible to have non-uniform distribution of keys onto the ring. Nearly all the keys could be mapped to one server, while others don't get much keys.
- This problem is solved by virtual nodes.

# Virtual Nodes

- Virtual Nodes also referred as virtual replicas, is an extension to consistent hashing that aims to improve data distribution uniformly on all servers. Instead of mapping each physical node to a single point on ring, each server is represented by multiple virtual nodes on the ring. This results in more even distribution of keys across the nodes.



# Data Replication



# Benefits of consistent hashing

- Minimized keys are re-distributed when servers are added or removed
- Easier to scale horizontally since only  $k/n$  keys would be affected.
- Mitigate hotspot problem. Chances of excessively accessed keys arriving on same virtual node is very low.

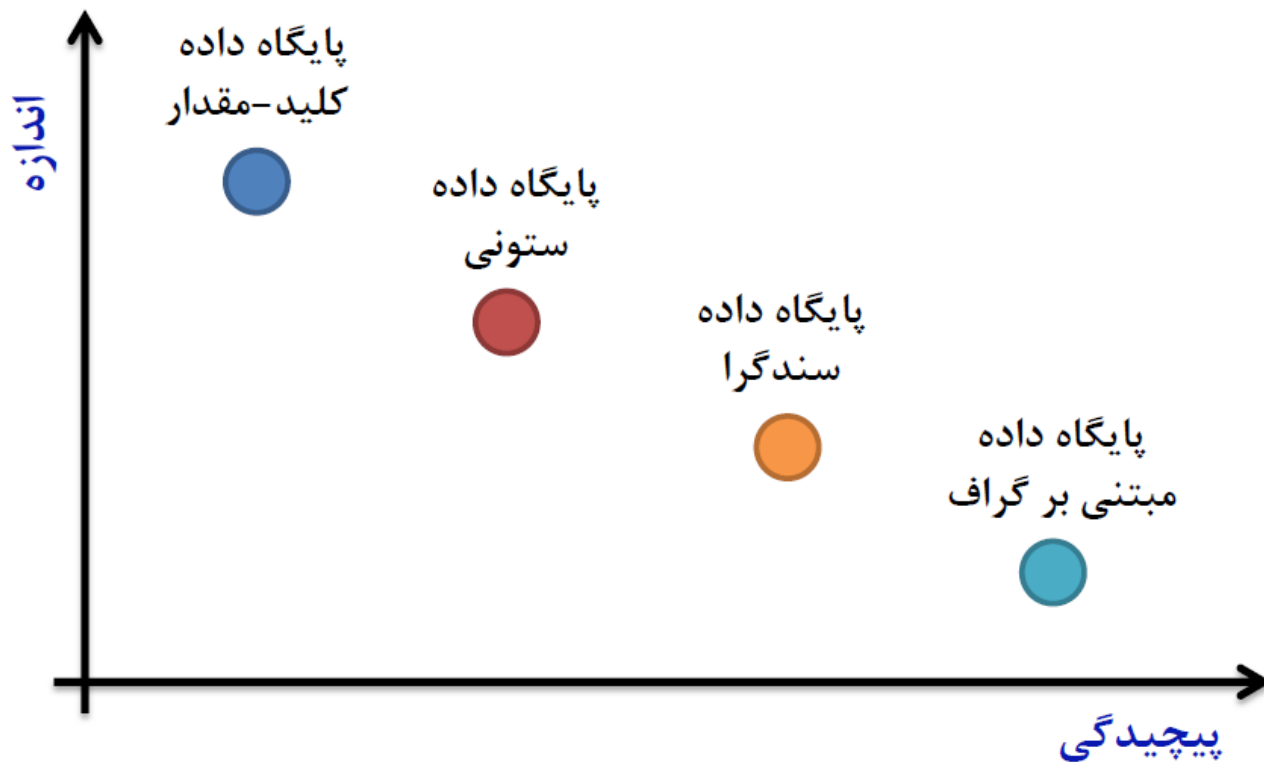


# NoSQL Types

- Key-Value Store
- Document Based
- Column Based
- Graph Based
- Vector Based


key-value		
Amazon DynamoDB (Beta)	ORACLE 11g BERKELEY DB	redis
column		
HBASE	riak	Cassandra
graph		
Neo4j the graph database	InfiniteGraph	sones
document		
CouchDB relax	mongoDB	terracore


# NoSQL DB




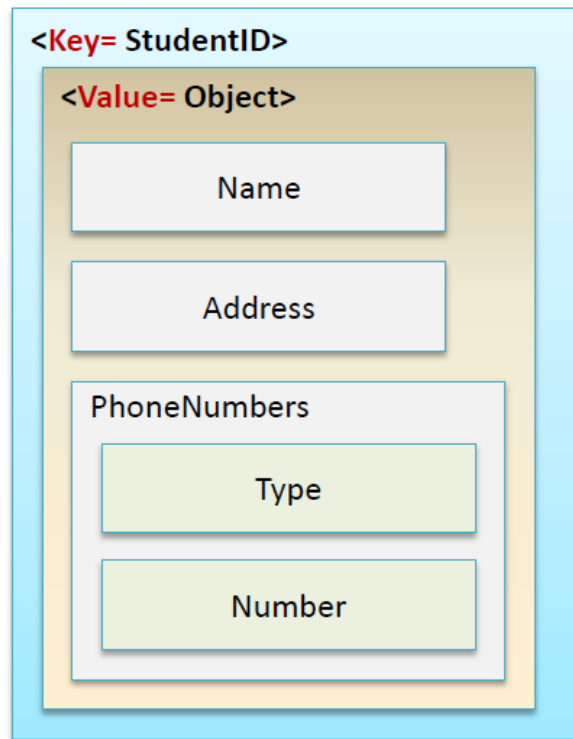
# Key-Value

- $(k, v)$
- $v$  is JSON or XML

$v := \text{store.get}(k)$  

$\text{store.put}(k, v)$  

$\text{store.delete}(k)$  



# Key-Value

## ✓ Advantages

- High speed
- Highly scalable
- Simple data model
- Supports horizontal distribution

## ✗ Disadvantages

- Many data structures cannot be modeled with key-value pairs
- Not suitable for complex queries with aggregation operators
- If data linking (joins) is needed, it must be handled at the application level

# Document Based

```
{
  "firstName": "Ali",
  "lastName": "Mohseni",
  "address": {
    "street": "21 2nd street",
    "city": "tehran"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "02177665544"
    },
    {
      "type": "mobile",
      "number": "091234567890"
    }
  ]
}
```

**JSON**

```
<?xml version="1.0">
<student>
  <firstName> Ali </firstName>
  <lastName> Mohseni </lastName>
  <address>
    <street> 21 2nd street </street>
    <city> tehran </city>
  </address>
  <phoneNumbers>
    <phone>
      <type> home </type>
      <number> 02177665544 </number>
    </phone>
    <phone>
      <type> mobile </type>
      <number> 91234567890 </number>
    </phone>
  </phoneNumbers>
</student>
```

**XML**