Lecture 2

# Basic Principles: Scaling, Sharding & Replication


**Yuliia Prokop**
prokoyul@fel.cvut.cz

29. 9. 2025

Based on the presentation of Martin Svoboda (martin.svoboda@matfyz.cuni.cz)

**Czech Technical University in Prague**, Faculty of Electrical Engineering

# Lecture Outline

Different aspects of **data distribution**

- **Scaling**
  - Vertical vs. horizontal
- **Distribution** models
  - **Sharding**
  - **Replication**: master-slave vs. peer-to-peer architectures
- **CAP** properties
  - **Consistency**, **availability** and partition tolerance
  - ACID vs. **BASE guarantees**
- **Consistency**
  - Read and write quora

# Scalability

# Scalability

What is **scalability**?

- = **capability of a system to handle growing amounts of data and/or queries** without losing performance, or its potential to be enlarged in order to accommodate such a growth

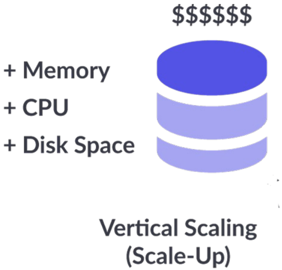
Two general approaches

- Vertical scaling
- Horizontal scaling

# Vertical Scalability

**Vertical scaling** (scaling up/down)

- = **adding resources to a <u>single node</u> in a system**
  - E.g. increasing the number of CPUs, extending system memory, using larger disk arrays, …
  - I.e. **larger and more powerful machines** are involved
- Traditional choice
  - In favor of **strong consistency**
  - Easy to implement and deploy
  - No issues caused by data distribution
  - …




Works well in many cases but …

# Vertical Scalability: Drawbacks

**Performance limits**

- **Even the most powerful machine has a limit**
- Moreover, everything works well…
  at least until we start approaching such limits

**Higher costs**

- The cost of expansion increases exponentially
  - In particular, it is **higher than the sum of costs of equivalent commodity hardware**

**Proactive provisioning**

- New projects / applications might evolve rapidly
- **Upfront budget is needed** when deploying new machines
- And so flexibility is seriously suppressed

# Vertical Scalability: Drawbacks

**Vendor lock-in**

- There are **only a few manufacturers** of large machines
- Customer is made dependent on a single vendor
  - Their products, services, but also implementation details, proprietary formats, interfaces, support, …
- I.e. it is difficult or impossible to switch to another vendor

**Deployment downtime**

- Inevitable downtime is often required when scaling up

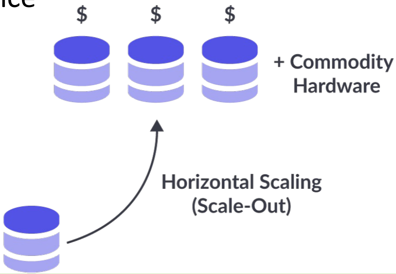
# Horizontal Scalability

**Horizontal scaling** (scaling out/in)

- = **adding more nodes to a system**
  - I.e. system is distributed across multiple nodes in a cluster
- Choice of many NoSQL systems

Advantages

- **Commodity hardware, cost effective**
- **Flexible** deployment and maintenance
- Often surpasses the vertical scaling
- Often no single point of failure
- …



# Horizontal Scalability: statistics

**Apple** Corporation uses Cassandra *(October, 2022)*:

- **thousands** of clusters
- about **300,000 nodes**
- about **100 Petabytes** of data

# Horizontal Scalability: Consequences

Significantly **increases complexity**

- Complexity of management, programming model, …

Introduces **new issues and problems**

- Data distribution
- Synchronization of nodes
- Data consistency
- Recovery from failures
- …

And there are also plenty of **false assumptions** …

# Horizontal Scalability: Fallacies

False assumptions

- Network is **reliable**
- **Latency** is zero
- **Bandwidth** is infinite
- Network is **secure**
- **Topology** does not change
- There is one **administrator**
- Network is **homogeneous**
- **Transport cost** is zero

Source: https://blogs.oracle.com/jag/resource/Fallacies.html

# Horizontal Scalability: Conclusion

⇒ a standalone node still might be a better option in certain cases

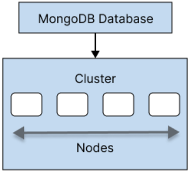
- E.g. for graph databases
  - Simply because it is difficult to split and distribute graphs
- In other words
  - **It can make sense to run even a NoSQL database system on a single node**
  - No distribution at all is the most preferred / simple scenario

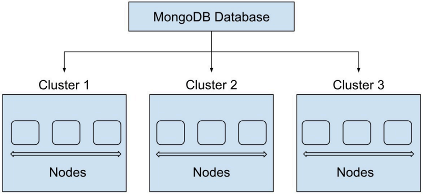
But in general, horizontal scaling really opens new possibilities

# Horizontal Scalability: Architecture

What is a **cluster**?

- = **a collection of mutually interconnected commodity nodes**
- Based on the **shared-nothing architecture**
  - Nodes do not share their CPUs, memory, or hard drives,
  - … Each node runs its operating system instance
  - **Nodes send messages to interact with each other**
- Nodes of a cluster can be heterogeneous
- Data, queries, calculations, requests, workload, …
  this is all <u>**distributed**</u> **among the nodes** within a cluster




https://www.geopits.com/blog/mongodb-replication-and-sharding.html

# Distribution Models

# Distribution Models

Generic techniques of **data distribution**

- **Sharding**
  - Idea: **different data on different nodes**
  - Motivation: increasing volume of data, increasing performance
- **Replication**
  - Idea: **the same data on different nodes**
  - Motivation: increasing performance, increasing fault tolerance

Both the techniques are <u>mutually orthogonal</u>

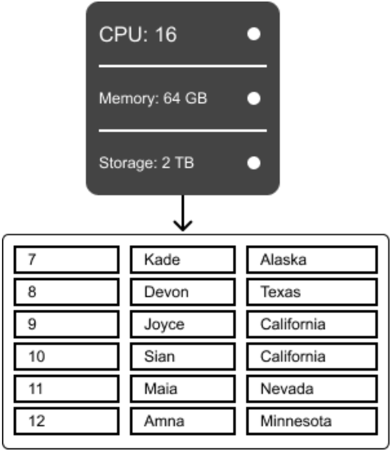
- I.e. we can use either of them, or combine them both

**Distribution model**

- = specific way how sharding and replication is implemented

NoSQL systems often offer **automatic sharding and replication**

# Sharding


CPU: 16
Memory: 64 GB
Storage: 2 TB
7
Kade
Alaska
8
Devon
Texas
9
Joyce
California
10
Sian
California
11
Maia
Nevada
12
Amna
Minnesota
CPU: 16
Memory: 64 GB
Storage: 2 TB
1
Faith
California
2
Max
Arizona
3
Pablo
Georgia
4
Dora
California
5
Philippa
Texas
6
Finnian
Alaska

# Sharding

**Sharding** (horizontal partitioning)

- **Placement of different data on different nodes**
  - What *different data* means? Usually *aggregates*
    - E.g. key-value pairs, documents, …
  - **Related pieces of data that are accessed together should also be kept together**
    - Specifically, operations involving data on multiple shards should be avoided (if possible)

The questions are…

- how to design aggregate structures?
- how to actually distribute these aggregates?

# Sharding vs Partitioning

## Partitioned Data

Database Instance

Partition 1

| ID | First Name | Last Name | Favorite Food |
|---|---|---|---|
| 1 | Bob | Smith | Bacon |
| 2 | Richard | Jones | Shrimp |
| 3 | Lori | Milton | Pizza |

Partition 2

| | | | |
|---|---|---|---|
| 4 | Sara | Simon | Brocolli |
| 5 | Aldene | Harris | Barbeque |

## Sharded Data

Sharded Database

Database Instance 1

Shard 1

| ID | First Name | Last Name | Favorite Food |
|---|---|---|---|
| 1 | Bob | Smith | Bacon |
| 2 | Richard | Jones | Shrimp |
| 3 | Lori | Milton | Pizza |

Database Instance 2

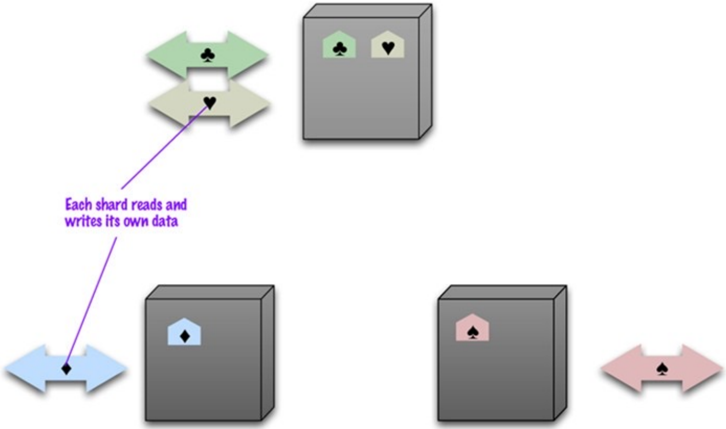
Shard 2

| ID | First Name | Last Name | Favorite Food |
|---|---|---|---|
| 4 | Sara | Simon | Brocolli |
| 5 | Aldene | Harris | Barbeque |

Notice in the partitioned architecture the data is divided into chunks, but the chunks live on a single database instance (node). In the sharded data example, each shard lives on a separate database instance (node).

https://www.pingcap.com/blog/database-sharding-defined/

# Sharding




Source: Sadalage, Pramod J. - Fowler, Martin: NoSQL Distilled. Pearson Education, Inc., 2013.

| id | name | city |
|---|---|---|
| 1 | Jonas | Prague |
| 2 | Hana | Brno |
| 3 | Kristian | Plzen |
| 4 | Ahmed | Prague |

Hash Function

| name | hash |
|---|---|
| Prague | 2 |
| Brno | 1 |
| Plzen | 1 |
| Prague | 2 |

**Shard 1**

| id | name | city |
|---|---|---|
| 2 | Hana | Brno |
| 3 | Kristian | Plzen |

**Shard 2**

| id | name | city |
|---|---|---|
| 1 | Jonas | Prague |
| 4 | Ahmed | Prague |

# Sharding

Objectives

- Achieve **uniform data distribution**
- Achieve **balanced workload** (read and write requests)
- Respect **physical locations**
  - E.g. different data centers for users around the world
- …

Unfortunately, these objectives…

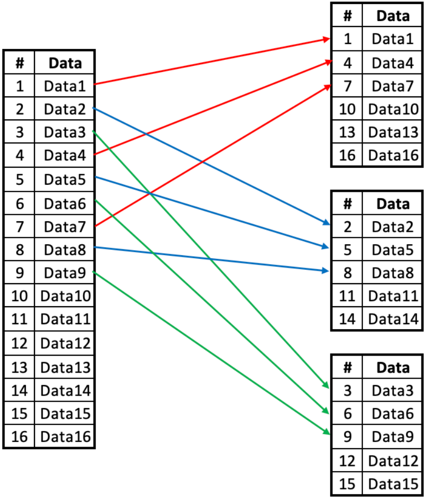
- may **mutually contradict each other**
- may **change in time**

So, how to actually determine shards for aggregates?

# Sharding

## Sharding strategies

- Based on <u>mapping structures</u>
  - Data is placed on shards in a ***random*** fashion
    - E.g. round-robin, …
  - Knowledge of the **mapping of individual aggregates to particular shards must then be maintained**
    - Thus usually maintained using a centralized index structures with all the disadvantages

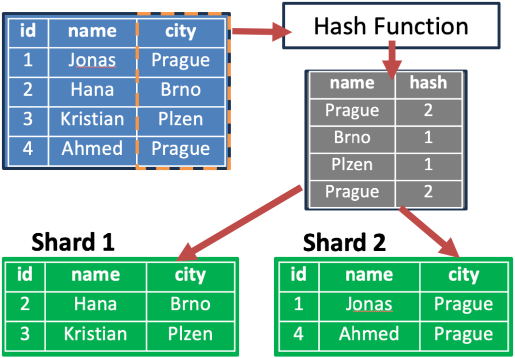


# Sharding

## Sharding strategies

- Based on general rules
  - **Each shard is responsible for storing certain data**

**Hash sharding**



# Sharding

## Sharding strategies

- Based on <u>general rules</u>
  - **Each shard is responsible for storing specific data**

### Range-based sharding

**Original data**

| | | |
|---|---|---|
| 1 | Faith | California |
| 2 | Max | Arizona |
| 3 | Pablo | Georgia |
| 4 | Dora | California |
| 5 | Philippa | Texas |
| 6 | Finnian | Alaska |
| 7 | Kade | Alaska |
| 8 | Devon | Texas |
| 9 | Joyce | California |
| 10 | Sian | California |
| 11 | Maia | Nevada |
| 12 | Amna | Minnesota |

**Shard 1: 1-6**

| | | |
|---|---|---|
| 1 | Faith | California |
| 2 | Max | Arizona |
| 3 | Pablo | Georgia |
| 4 | Dora | California |
| 5 | Philippa | Texas |
| 6 | Finnian | Alaska |

**Shard 2: 7-12**

| | | |
|---|---|---|
| 7 | Kade | Alaska |
| 8 | Devon | Texas |
| 9 | Joyce | California |
| 10 | Sian | California |
| 11 | Maia | Nevada |
| 12 | Amna | Minnesota |

# Sharding

### Hotspot

**Shard 1**

| id | name | city |
|---|---|---|
| 2 | Hana | Brno |
| 3 | Kristian | Plzen |
| 5 | Tomas | Brno |
| 6 | Ella | Brno |
| 7 | Matej | Plzen |
| 8 | Lenka | Brno |

**Shard 2**

| id | name | city |
|---|---|---|
| 1 | Jonas | Prague |
| 4 | Ahmed | Prague |

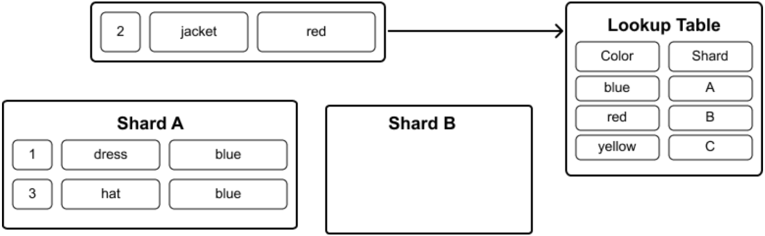
**Shard 3**

| id | name | city |
|---|---|---|
| 9 | Jakub | Ostrava |

# Sharding

**Sharding strategies based on general rules:**

- Directory-based sharding:
  - In an e-commerce system, a central directory server stores information about which shard contains each user's data. When requesting data, the system first consults the directory to determine the appropriate shard.

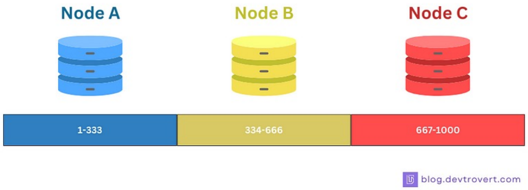


# Sharding

**Sharding strategies based on general rules:**

- Hash sharding : downsides
  - How to add server D and redistribute data between nodes? Node A (1–250), node B (251–500), node C (501–750), and node D (751–1000).

  A lot of the 1000 data items now need to be shifted to make room for Node D

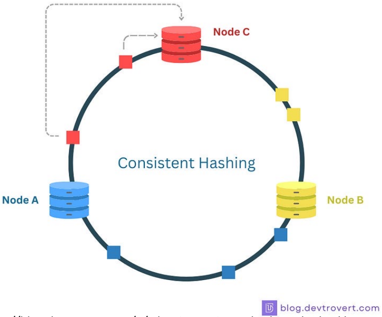


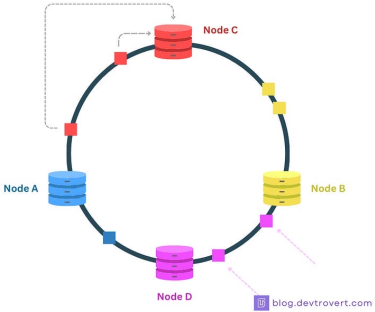

Source: https://blog.devtrovert.com/p/what-is-consistent-hashing-the-backbone

# Sharding

**Sharding strategies based on general rules:**

- Consistent hash sharding :
  - A distributed cache system uses a hash ring. Only a portion of the data is redistributed when adding or removing a node, minimizing data movement between shards.




Source: https://blog.devtrovert.com/p/what-is-consistent-hashing-the-backbone

# Sharding

**Sharding strategies based on general rules:**

- Entity/relationship-based sharding:
  - In a social network, a user and all related data (posts, friends, messages) are stored on a single shard. This ensures quick access to related information.
- Geography-based sharding :
  - A global video streaming service distributes content across servers in different geographical regions. Users in Europe are served by European servers, those in Asia by Asian servers, and so on.

# Sharding

**Sharding strategies based on general rules:**

- Functional sharding (by application **functionality** or modules):
  - Data is divided based on its functional purpose or logical affiliation with specific application parts. For example, in an extensive e-commerce system, separate shards can be created for user data, product data, order data, and so on.

- Time-based sharding:
  - Data is distributed across shards depending on the timestamp or date associated with each record.
  - Each shard contains data for a specific time (for example, a day, week, month, or year).

# Sharding

**Sharding strategies based on general rules:**

- Combined sharding (combination of multiple strategies):
  - A banking system uses geographic sharding to distribute data by region, and within each region, range sharding is applied based on account numbers.

# Sharding

**Why is sharding difficult**?

- Not only we need to be able to determine particular shards during write **requests**
  - I.e. when a new aggregate is about to be inserted
  - So that we can actually make a decision where it should be physically stored
- but also during read **requests**
  - I.e. when existing aggregate/s are about to be retrieved
  - So that we can actually **find and return them efficiently** (or detect they are missing)
  - And all that only based on the **search criteria** provided (e.g. key, id, …) unless all the nodes should be accessed

# Sharding

**Why is sharding** even more **difficult**?

- Structure of the **cluster may be changing**
  - Nodes can be added or removed
- Nodes may have **incomplete / obsolete cluster knowledge**
  - Nodes involved, their responsibilities, sharding rules, …
- Individual **nodes may be failing**
- **Network may be partitioned**
  - Messages may not be delivered even though sent

# Replication

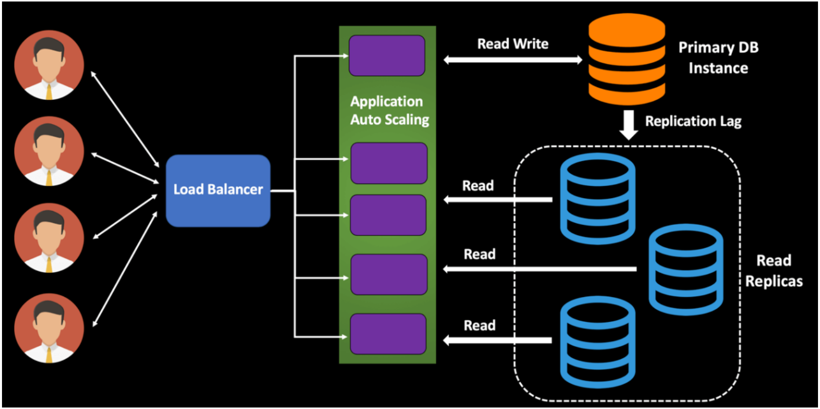
Replication

- **Placement of multiple copies of the same data (replicas) on different nodes**
- **Replication factor** = number of such copies

Two approaches

- **Master-slave architecture**
- **Peer-to-peer architecture**

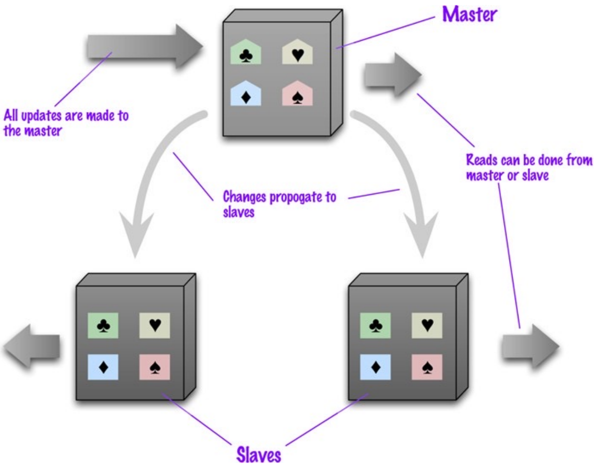
# Replication




Source: https://www.red-gate.com/simple-talk/databases/sql-server/performance-sql-server/designing-highly-scalable-database-architectures/

# Replication

## Master-Slave Architecture




Source: Sadalage, Pramod J. - Fowler, Martin: NoSQL Distilled. Pearson Education, Inc., 2013.

# Replication

## Master-Slave Architecture

Architecture

- **One node is primary** (master), **all the other secondary** (slave)
- Master node bears all the management responsibility
- All the nodes contain identical data

**Read requests can be handled by both the master or slaves**

- Suitable for read-intensive applications
  - More read requests to deal with → more slaves to deploy
- When the master fails, read operations can still be handled

# Replication

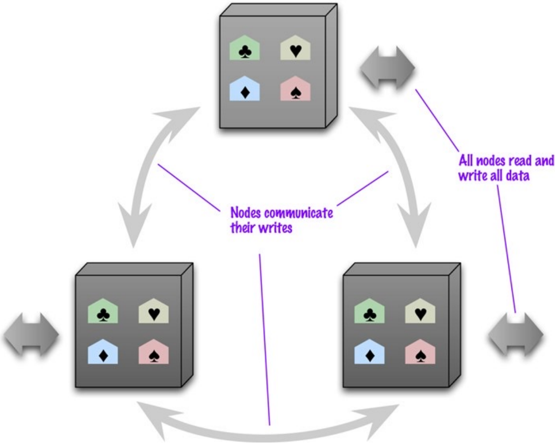
## Master-Slave Architecture

**Write requests can only be handled by the master**

- **Newly written replicas are propagated to all the slaves**
- Consistency issue
  - **At most, one write request is handled at a time.**
  - But the propagation still takes some time, during which obsolete reads might happen.
  - Hence certain **synchronization is required to avoid conflicts**
- In case of **master failure**, a new one needs to be appointed
  - **Manually** (user-defined) or **automatically** (cluster-elected)
  - Since the nodes are identical, appointments can be fast
- Master might therefore represent a **bottleneck** (because of the performance or failures)

# Replication

## Peer-to-Peer Architecture




Source: Sadalage, Pramod J. - Fowler, Martin: NoSQL Distilled. Pearson Education, Inc., 2013.

# Replication

## Peer-to-Peer Architecture

Architecture

- All the nodes have **equal roles and responsibilities**
- All the nodes contain identical data once again

**Both read and write requests can be handled by any node**

- No bottleneck, no single point of failure
- Both the operations scale well
  - More requests to deal with → more nodes to deploy
- Consistency issues
  - Unfortunately, **multiple write requests can be initiated independently and being executed at the same time**
  - Hence **synchronization is required to avoid conflicts**

# Consensus Protocols

Consensus protocols ensure data consistency among replicas.

They are essential for making unified decisions about the system's state, especially during failures and network partitions.

**Main Protocols:**

- Paxos
- Raft
- Viewstamped Replication
- Zookeeper’s Zab Protocol.

**Role of Consensus Protocols:**

- Ensuring data consistency across replicas.
- Leader or master election within the system.
- Coordinating data updates and managing node failures.

# Sharding and Replication

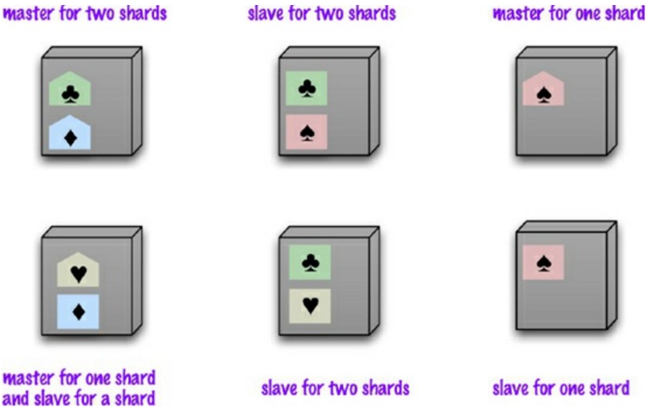
**Observations** with respect to the **replication**:

- **Does the replication factor really need to correspond to the number of nodes**?
  - No, replication factor of 3 will often be the right choice
  - Consequences
    - Nodes will no longer contain identical data
    - **Replica placement strategy** will be needed
- **Do all the replicas really need to be successfully written** when write requests are handled?
  - No, but consistency issues have to be tackled carefully

**Sharding and replication can be combined**… but how?

# Sharding and Replication

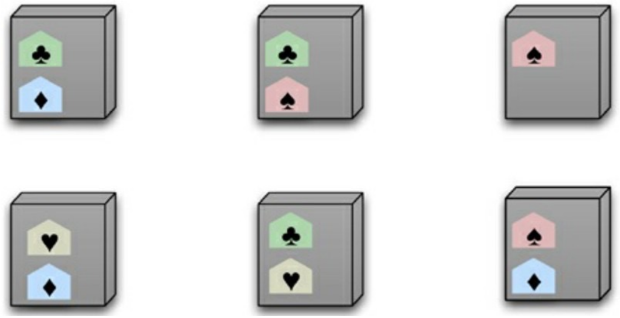
## Sharding and Master-Slave Replication




Source: Sadalage, Pramod J. - Fowler, Martin: NoSQL Distilled. Pearson Education, Inc., 2013.

# Sharding and Replication

## Sharding and Peer-to-Peer Replication



Source: Sadalage, Pramod J. - Fowler, Martin: NoSQL Distilled. Pearson Education, Inc., 2013.

# Sharding and Replication

## Sharding and Peer-to-Peer Replication

Storing information about data location:

- Cluster metadata
- Storage format (for example):

```
{
   "shard1": {
      "primary": "node1",
      "replicas": ["node2", "node3"],
      "keyRange": {"min": 0, "max": 1000}
   },
   "shard2": {
      "primary": "node4",
      "replicas": ["node5", "node6"],
      "keyRange": {"min": 1001, "max": 2000}
   }
}
```

# Sharding and Replication

Combinations of sharding and replication

- **Sharding + master-slave replication**
  - Multiple masters, each for different data
  - Roles of the nodes can overlap
    - Each node can be master for some data and/or slave for other
- **Sharding + peer-to-peer replication**
  - Basically placement of anything anywhere
    (although certain rules can still be applied)

# Sharding and Replication

Questions to figure out for any distribution model

- Can all the nodes serve both **read and write requests**?
- Which **replica placement strategy** is used?
- How the **mapping of replicas** is maintained?
- What level of **consistency and availability** is provided?
- What extent of **infrastructure knowledge** do the nodes have?
- …