Basic Principles

Big Data Management

Anis Ur Rahman
Faculty of Computer Science & Information Technology
University of Malaya

October 15, 2019

Borrowed from http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/

Lecture Outline

Different aspects of data distribution

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

Outline

1. Scalability
2. Distribution Models
3. CAP Theorem
4. Consistency

Scalability

What is scalability?

- the 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

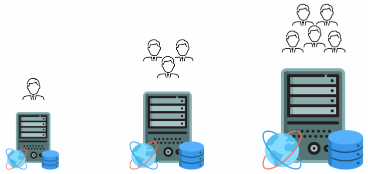
- Vertical scaling
- Horizontal scaling

Vertical Scalability

Vertical scaling (scaling up/down)

- **Adding** resources to a **single node** 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**
- Everything works well ... **until** start approaching the limit

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
- 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, ...
- e.e. it is **difficult** or **impossible** to **switch** to another vendor

Deployment downtime

- **Inevitable** downtime is often required when scaling up

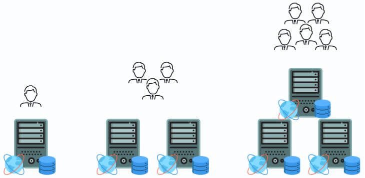
Horizontal Scalability

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: 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://www.red-gate.com/simple-talk/blogs/the-eight-fallacies-of-distributed-computing/`

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, hard drives, ...
  - Each node **runs its own** operating system instance
  - Nodes **send messages** to interact with each other
- Nodes of a cluster can be **heterogeneous**
- Data, queries, calculations, requests, workload, ... all **distributed** among the nodes within a cluster

Outline

1 Scalability

2 Distribution Models

3 CAP Theorem

4 Consistency

Distribution Models

Generic techniques of data distribution

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

Both the techniques are mutually orthogonal

- 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

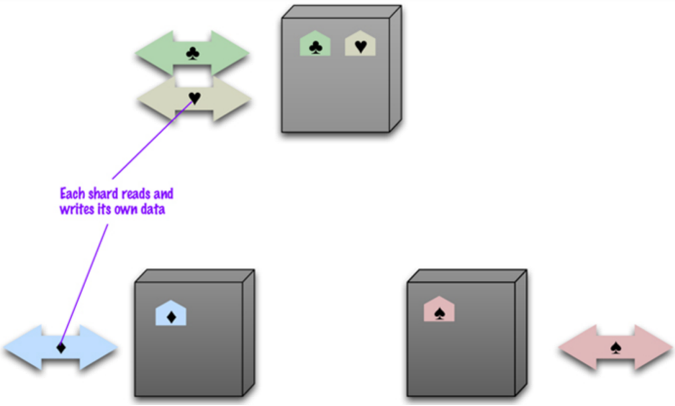
Sharding (horizontal partitioning)

- 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



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

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

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Vertical Partitions

VP1

CUSTOMER ID	FIRST NAME	LAST NAME
1	TAEKO	OHNUKI
2	O.V.	WRIGHT
3	SELDA	BAĞCAN
4	JIM	PEPPER

VP2

CUSTOMER ID	FAVORITE COLOR
1	BLUE
2	GREEN
3	PURPLE
4	AUBERGINE

Horizontal Partitions

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Source:

`https://www.digitalocean.com/community/tutorials/understanding-database-sharding`

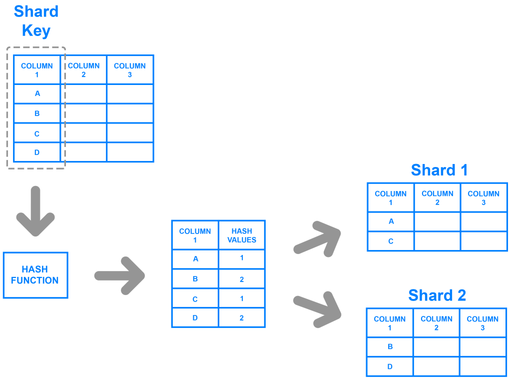
Sharding

Sharding strategies

- Based on **mapping structures**
    - 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
- Based on general rules
    - Each shard is responsible for **storing certain data**
    - Hash partitioning, range partitioning, ...

Sharding

Key-based sharding



Source:

https://www.digitalocean.com/community/tutorials/understanding-database-sharding

Sharding

Range-based sharding

PRODUCT	PRICE
WIDGET	$118
GIZMO	$88
TRINKET	$37
THINGAMAJIG	$18
DOODAD	$60
TCHOTCHKE	$999

($0-$49.99)

PRODUCT	PRICE
TRINKET	$37
THINGAMAJIG	$18

($50-$99.99)

PRODUCT	PRICE
GIZMO	$88
DOODAD	$60

($100+)

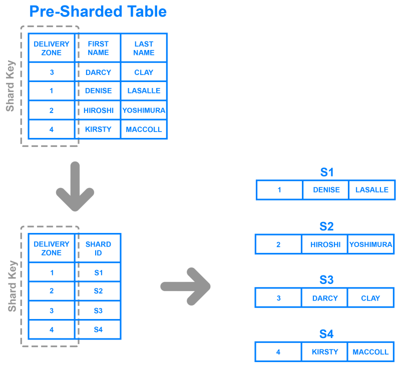
PRODUCT	PRICE
WIDGET	$118
TCHOTCHKE	$999

Source:

`https://www.digitalocean.com/community/tutorials/understanding-database-sharding`

Sharding

Directory-based sharding



Source:

https://www.digitalocean.com/community/tutorials/understanding-database-sharding

Sharding

Should I Shard?

- Amount of **application data** grows to exceed the storage capacity of a single database node.
- Volume of **writes** or **reads** to the database surpasses what a single node can handle,
  - resulting in slowed response times or timeouts.

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

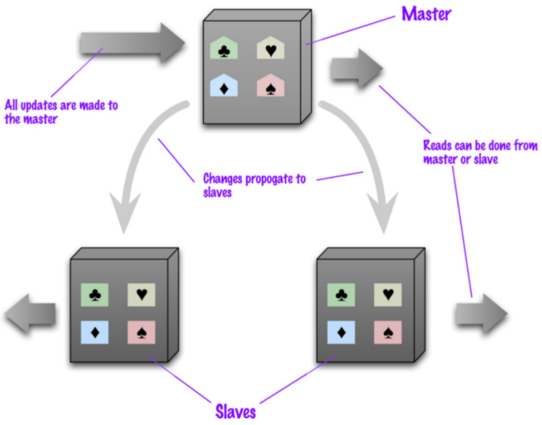
- Placement of **multiple copies** of the same data (replicas) on **different nodes**
- **Replication factor** = number of such copies

Two approaches

1. Master-slave architecture
2. Peer-to-peer architecture

Replication

Master-Slave Architecture



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

Replication

Master-Slave 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

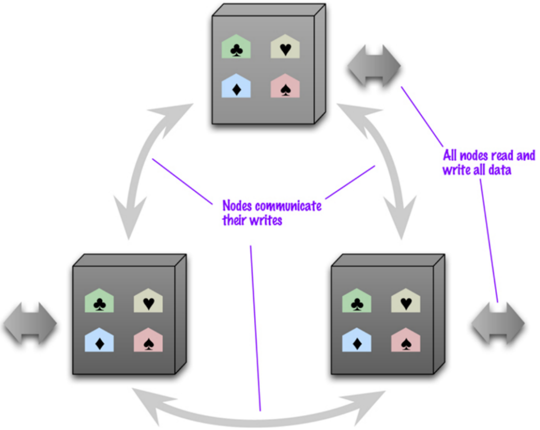
Master-Slave Architecture

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

- Newly written **replicas** are **propagated** to all the slaves
- **Consistency issue**
  - Luckily enough, **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**, appointment 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

- 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

Sharding and Replication

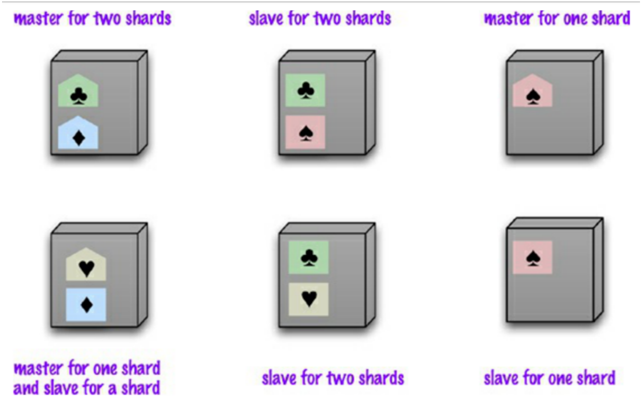
Observations with respect to the replication

- 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

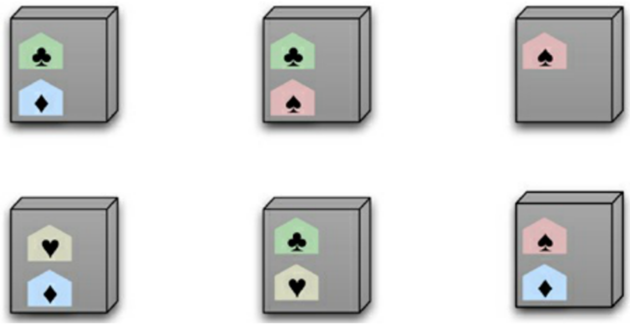
Sharding and Master-Slave Replication



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

Combinations of sharding and replication

1. **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**
2. **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?
- ...

Outline

1. Scalability
2. Distribution Models
3. CAP Theorem
4. Consistency

CAP Theorem

Assumptions

- **Distributed** system with sharding and replication
- Read and write operations on a **single aggregate only**

CAP properties

- Properties of a distributed system:
- Consistency, Availability, and Partition tolerance

CAP theorem

- It is **not possible** to have a distributed system that would **guarantee all** CAP properties at the same time.
- Only 2 of these 3 properties can be enforced.

But, what these properties actually mean?

CAP Properties

Consistency

- Read and write **operations** must be executed **atomically**
  - There must exist a total **order on all operations**
  - Each operation looks as if it was **completed at a single instant**
  - i.e. as if all operations were executed **sequentially** one by one on a single standalone node
- **Practical consequence.** after a write operation, all readers **see the same data**
  - Since any node can be used for handling of read requests, atomicity of write operations means that **changes must be propagated** to all the replicas
  - As we will see later on, other ways for such a strong consistency exist as well

CAP Properties

Availability

- If a node is working, it **must respond** to user requests
  - Every read or write request successfully received by a non-failing node in the system must result in a response,
  - i.e. their execution must not be rejected

Partition tolerance

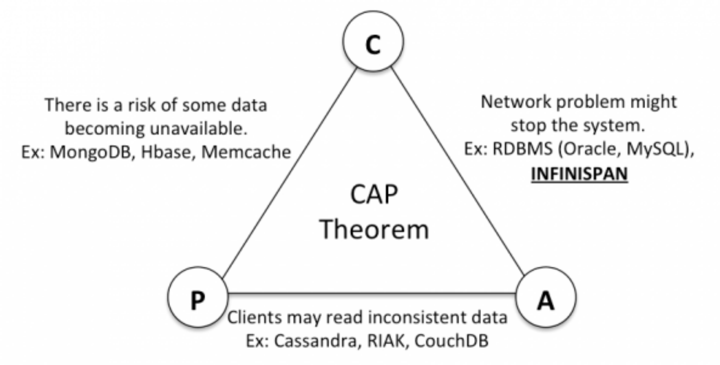
- System **continues to operate** even when two or more sets of **nodes get isolated**
- The network is **allowed to lose** arbitrarily many messages sent from one node to another
  - i.e. a connection failure **must not shut** the whole system down

CAP Theorem Consequences

If **at most two** properties can be **guaranteed** ...

1. **CA = consistency + availability**
    - Traditional **ACID** properties are **easy to achieve**
    - **Examples.**: RDBMS, Google BigTable
    - Any single-node system, but even clusters (at least in theory)
        - However, should the **network partition** happen, all the nodes must be **forced to stop** accepting user requests
2. **CP = consistency + partition tolerance**
    - **Other examples.** distributed locking
3. **AP = availability + partition tolerance**
    - New concept of **BASE** properties
    - **Examples.** Apache Cassandra, Apache CouchDB
    - **Other examples.** web caching, DNS

CAP Theorem Consequences



CAP Theorem Consequences

Partition tolerance is necessary in clusters

- Why? Because it is **difficult to detect** network failures
- Does it mean that only purely CP and AP systems are possible?
- No...

The real meaning of the CAP theorem:

- The real-world does not need to be just black and white
- **Partition tolerance** is a **must**, but we can **trade off** consistency versus availability
  - Just a little bit **relaxed consistency** can bring a lot of availability
  - Such trade-offs are not only possible, but often **works very well** in practice

CAP Theorem Example

You are asked to design a distributed cluster of 4 data nodes. Replication factor is 2 i.e. any data written in cluster must be written on 2 nodes; so when one goes down – second can serve the data. Now try to apply CAP theorem on this requirement.

In distributed system, two things may happen anytime i.e. node failure (hard disk crash) or network failure (connection between two nodes go down).

1. CP [Consistency/Partition Tolerance] Systems
2. AP [Availability/Partition Tolerance] Systems
3. CA [Consistency/Availability] Systems

ACID Properties

Traditional ACID properties

1. **Atomicity**
   - Partial execution of transactions is **not allowed** (all or nothing)
2. **Consistency**
   - Transactions bring the database from **one consistent (valid) state to another**
3. **Isolation**
   - Transactions executed in parallel **do not see uncommitted effects** of each other
4. **Durability**
   - Effects of **committed transactions** must remain **durable**

BASE Properties

New concept of BASE properties

1. **Basically Available**
   - The system **works** basically **all the time**
   - **Partial failures** can occur, but there are **no total system failures**
2. **Soft State**
   - The system is in **flux (unstable), non-deterministic state**
   - **Changes** occur all the time
3. **Eventual Consistency**
   - **Sooner or later** the system will be in **some consistent state**

BASE is just a vague term, no formal definition

- Proposed to illustrate design philosophies at the opposite ends of the consistency-availability spectrum

ACID and BASE

ACID

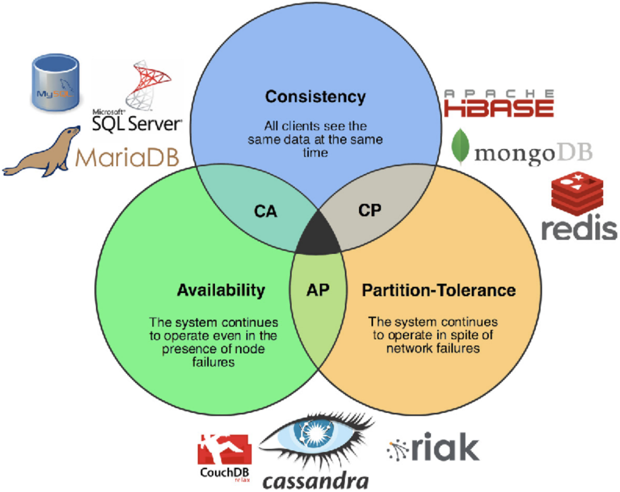
- Choose **consistency over availability**
- **Pessimistic** approach
- Implemented by **traditional** relational databases

BASE

- Choose **availability over consistency**
- **Optimistic** approach
- Common in **NoSQL databases**
- Allows levels of **scalability** that cannot be acquired with ACID

Current trend in NoSQL. strong consistency → eventual consistency

CAP Theorem Conclusion



Outline

1. Scalability
2. Distribution Models
3. CAP Theorem
4. Consistency

Consistency

Consistency in general...

- Consistency is the lack of contradiction in the database
- However, it has many facets ... e.g.
  - only assume **atomic operations** always **manipulating** just a **single aggregate**,
  - but **set operations** could also be considered etc.

Strong consistency is **achievable** even in clusters, but **eventual** consistency might often be **sufficient**

1. One minute obsolete article on a news portal does not matter
2. Even when an already unavailable hotel room is booked once again, the situation can still be figured out in the real world
3. ...

Consistency

Write consistency (update consistency)

- **Problem.** write-write conflict
  - Two or more **write requests** on the same aggregate are initiated concurrently
- **Context.** peer-to-peer architecture only
- **Issue.** lost update
- **Solution.**
  1. Pessimistic strategies
     - **Preventing** conflicts from occurring
     - Write locks, ...
  2. Optimistic strategies
     - Conflicts may occur, but are **detected** and **resolved** later on
     - Version stamps, vector clocks, ...

Consistency

Read consistency (replication consistency)

- **Problem.** read-write conflict
  - Write and read requests on the same aggregate are initiated concurrently
- **Context.** both master-slave and peer-to-peer architectures
- **Issue.** inconsistent read
- When not treated, inconsistency window will exist
  - Propagation of changes to all the replicas **takes some time**
  - Until this process is finished, inconsistent reads may happen
  - Even the initiator of the write request may read wrong data!
    - Session consistency/read-your-writes/sticky session

Strong Consistency

How many nodes need to be involved to get strong consistency?

- **Write quorum.** $W > N/2$
  - **Idea.** only one write request can get the **majority**
  - W = number of nodes successfully participating in the write
  - N = number of nodes involved in replication (replication factor)
- **Read quorum.** $R > N - W$
  - **Idea.** concurrent write requests cannot happen
  - R = number of nodes participating in the read
  - Should the retrieved replicas be mutually different, the newest version is resolved and then returned

When a quorum is not attained *rightarrow* the request cannot be handled

Strong Consistency

Examples for replication factor **N = 3**

- Write quorum **W = 3** and read quorum **R = 1**
  - All the replicas are always **updated**
  - Can **read** any one of them
- Write quorum **W = 2** and read quorum **R = 2**
  - Typical **configuration**, reasonable **trade-off**

Consequence

- Quora can be configured **to balance** read and write workload
  - The higher the **write quorum** is required lower the **read quorum** can then be required

Lecture Conclusion

There is a **wide range of options** influencing...

- **Scalability** – how well the entire system scales?
- **Availability** – when nodes may refuse to handle user requests?
- **Consistency** – what level of consistency is required?
- **Latency** – how long does it take to handle user requests?
- **Durability** – is the committed data written reliably?
- **Resilience** – can the data be recovered in case of failures?

it's good to know these properties and choose the right trade-off