

# Distributed Systems II

## *Software Architecture*

Brae Webb & Richard Thomas & Guangdong Bai

March 30, 2026

## *Distributed Systems Series*

Distributed I    *Reliability* and *scalability* of  
*stateless* systems

Distributed II    *Complexities* of *stateful*  
systems

Distributed III    *Hard problems* in distributed  
systems

## *Distributed Systems Series*

Distributed I Reliability and scalability of  
stateless systems

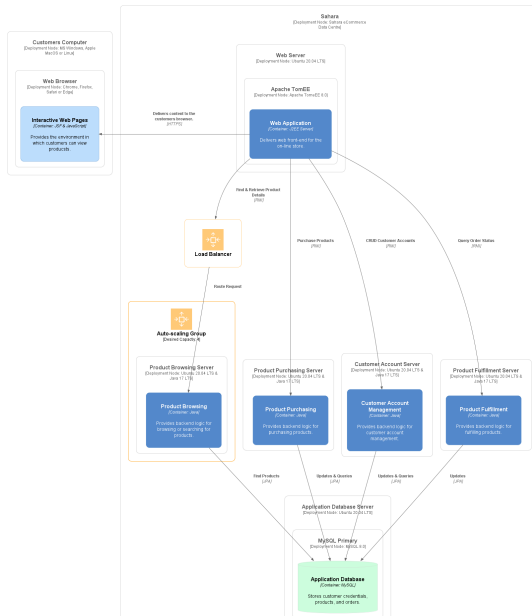
Distributed II *Complexities* of *stateful*  
systems

Distributed III Hard problems in distributed  
systems

*Previously in Distributed I: Benefits...*

- Improved *reliability*
- Improved *scalability*
- Improved *latency*

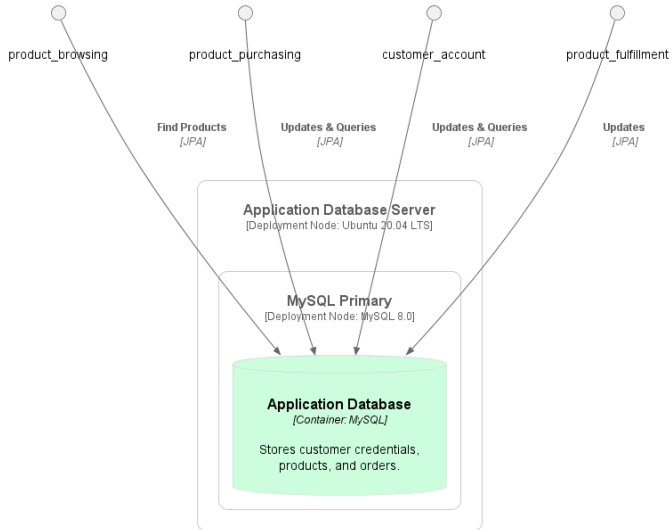
# Previously in Distributed I...



*Question*

What is the *problem*?

# Database



## *Stateless vs. Stateful Systems*

Stateless Does *not* utilise *persistent data*. Or: each request is independent.

Stateful Does utilise *persistent data*. Or: the server or service remembers and uses data from previous interactions.



*Disclaimer*

This is *not* a database course

## Advanced Database Systems (INFS3200)

### Course level

Undergraduate

### Faculty

[Engineering, Architecture & Information  
Technology](#)

### School

Info Tech & Elec Engineering

### Units

2

### Duration

One Semester

### Class contact

2 Lecture hours, 1 Tutorial hour, 1 Practical or Laboratory hour

### Incompatible

INFS7907

### Prerequisite

INFS2200

### Assessment methods

Examinations and coursework

### Current course offerings

Course offerings	Location	Mode	Course Profile
<a href="#">Semester 1, 2022</a>	St Lucia	Internal	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 1, 2022</a>	External	External	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 2, 2022</a>	External	External	PROFILE UNAVAILABLE
<a href="#">Semester 2, 2022</a>	St Lucia	Internal	PROFILE UNAVAILABLE

Please Note: Course profiles marked as not available may still be in development.

### Course description

Distributed database design, query and transaction processing, data integration, data warehousing, data cleansing, management of spatial data, and data from large scale distributed devices.

### Archived offerings

Course offerings	Location	Mode	Course Profile
<a href="#">Semester 1, 2021</a>	St Lucia	Flexible Delivery	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 1, 2021</a>	External	External	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 2, 2021</a>	External	External	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 2, 2021</a>	St Lucia	Internal	<a href="#">COURSE PROFILE</a>
<a href="#">Semester 1, 2020</a>	St Lucia	Internal	<a href="#">COURSE PROFILE</a>

*Question*

How do we fix database scaling issues?

*Question*

How do we fix database scaling issues?

*Answer*

- Replication

*Question*

How do we fix database scaling issues?

*Answer*

- Replication
- Partitioning

*Question*

How do we fix database scaling issues?

*Answer*

- Replication
- Partitioning
- Independent databases

*Question*

How do we fix database scaling issues?

*Answer*

- *Replication*
- Partitioning
- Independent databases

*Question*

What is *replication*?



*Definition 0.* Replication

Data copied across multiple different machines.



product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	10	\$10.00
4321	Lifelike Elephant Inflatable	5	\$50.00



product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	10	\$10.00
4321	Lifelike Elephant Inflatable	5	\$50.00

*Definition 0.* Replica

Database node which stores a copy of the data.

*Question*

What are the advantages of *replication*?

*Question*

What are the advantages of *replication*?

*Answer*

- *Scale* our database to cope with higher loads.

### *Question*

What are the advantages of *replication*?

### *Answer*

- *Scale* our database to cope with higher loads.
- Provide *fault tolerance* from a single instance failure.

### *Question*

What are the advantages of *replication*?

### *Answer*

- *Scale* our database to cope with higher loads.
- Provide *fault tolerance* from a single instance failure.
- Locate instances *closer to end-users*.

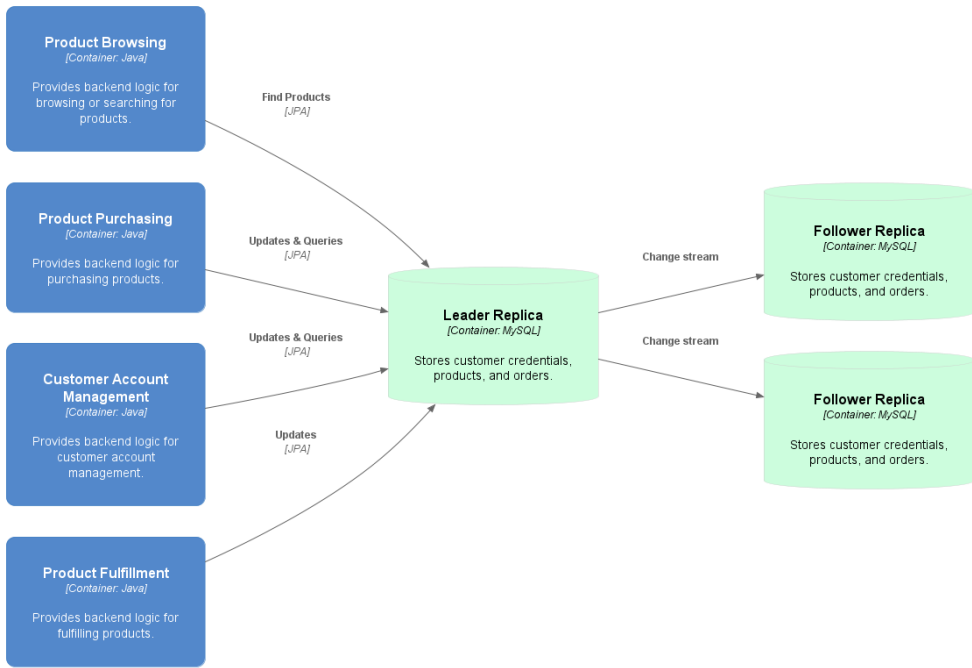
*Question*

How do we replicate our data?



*First Approach*

## Leader-Follower Replication



### *Definition 0.* Leader-based Replication

one node (the leader) handles all write operations, and multiple other nodes (followers) replicate the data and handle read operations.

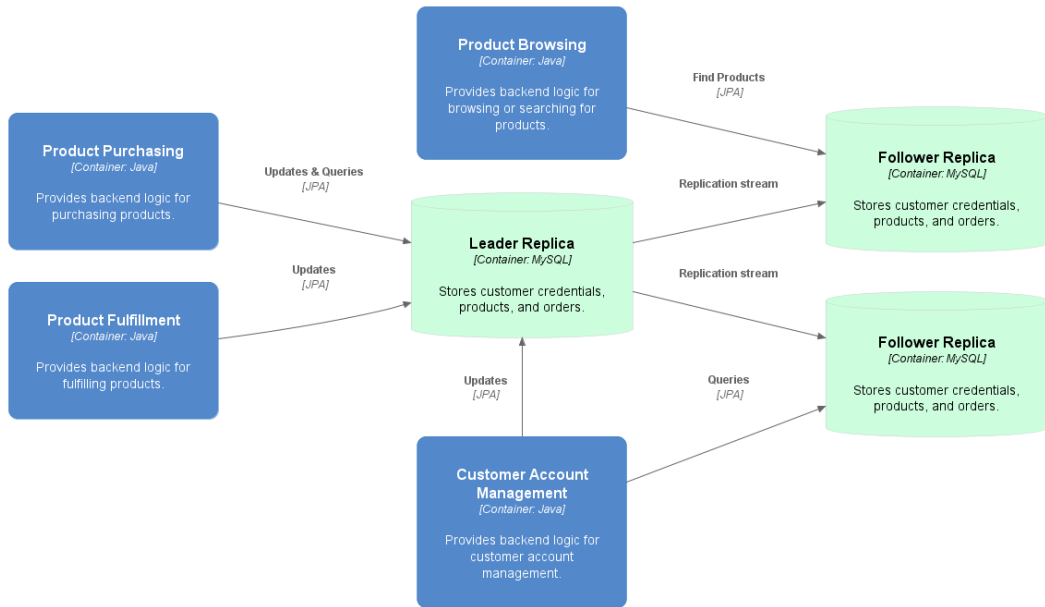
## *Leader-based Replication*

On write Writes sent to *leader*, change is propagated via change stream.

## *Leader-based Replication*

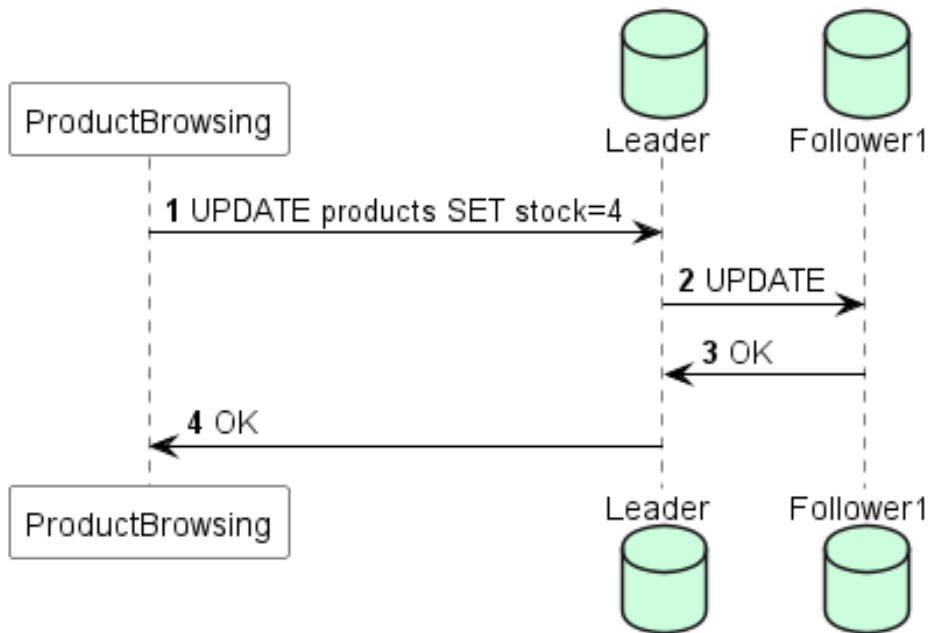
On write Writes sent to *leader*, change is propagated via change stream.

On read Any *replica* can be queried.

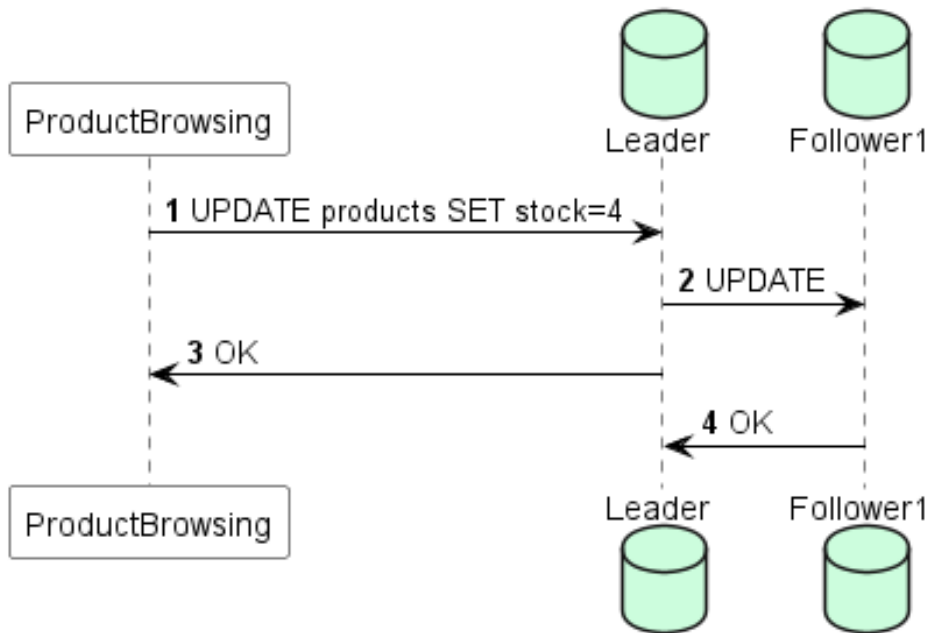


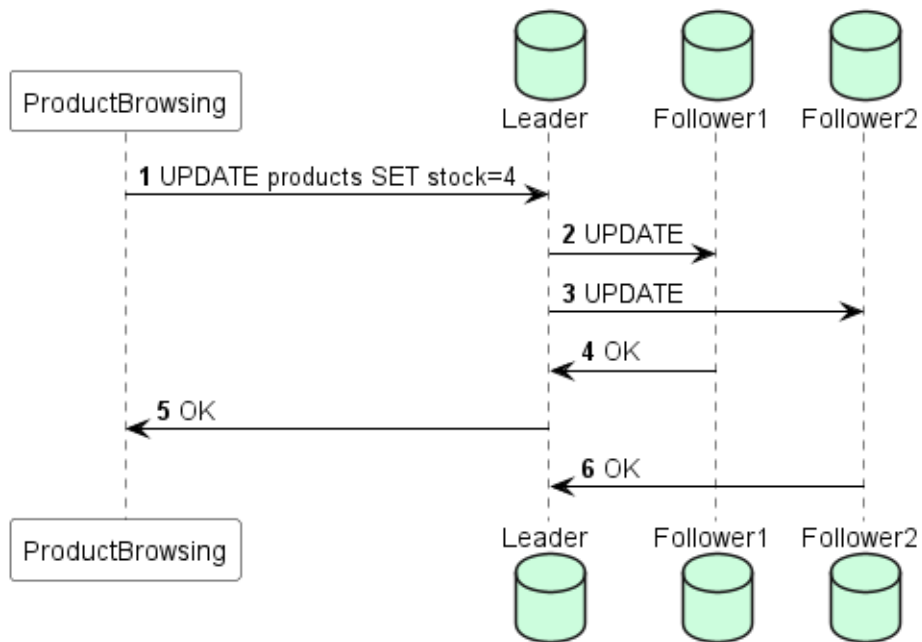
*Propagating Changes*

*Synchronous* vs. *Asynchronous*









## *Synchronous Propagation*

- Writes must propagate to *all followers* before being successful.

## *Synchronous Propagation*

- Writes must propagate to *all followers* before being successful.
- *Any* replica goes down, *all* replicas are un-writeable.

## *Synchronous Propagation*

- Writes must propagate to *all followers* before being successful.
- *Any* replica goes down, *all* replicas are un-writeable.
- Writes must *wait* for propagation to *all* replicas.

## *Asynchronous Propagation*

- Writes *don't* have to *wait* for propagation.

## *Asynchronous Propagation*

- Writes *don't* have to *wait* for propagation.
- If the leader goes down before propagating, the *write is lost*.

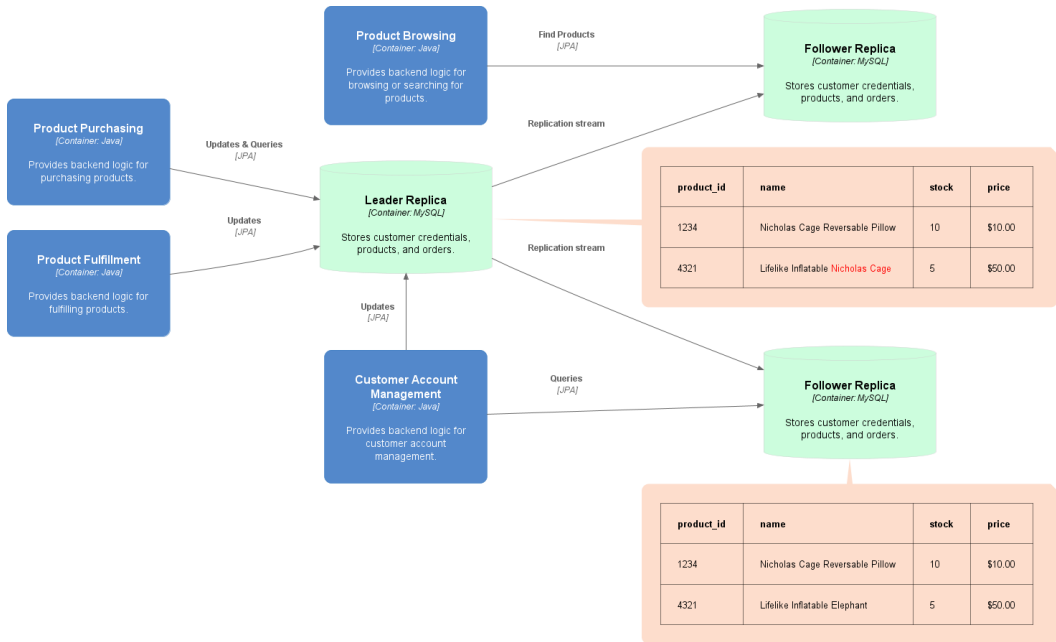
## *Asynchronous Propagation*

- Writes *don't* have to *wait* for propagation.
- If the leader goes down before propagating, the *write is lost*.
- Replicas can have out-dated or *stale* data.



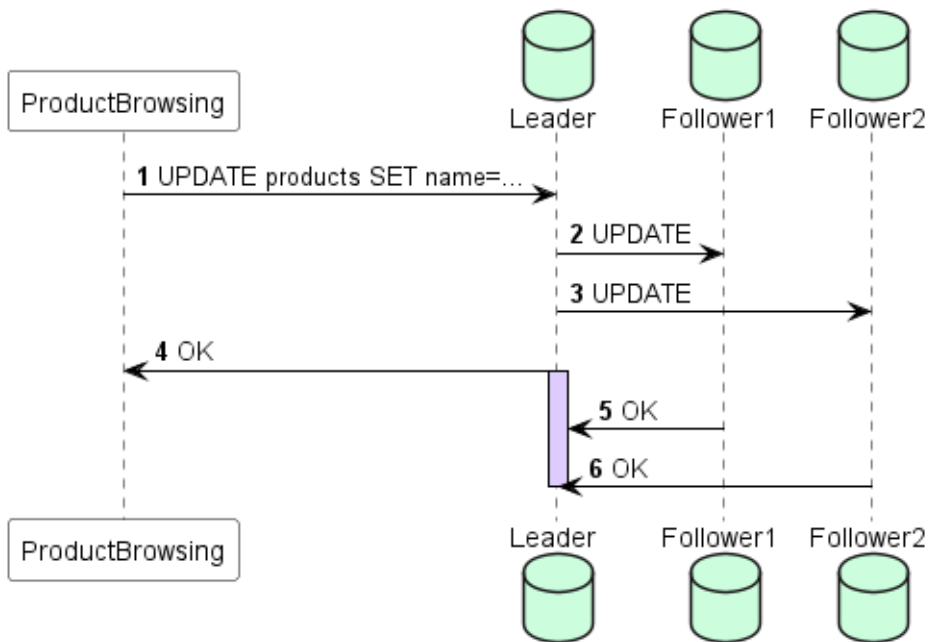
*Definition 0.* Stale data

Outdated or inconsistent data that does not reflect the latest updates, mainly due to replication delays, caching, or network issues in distributed systems.



*Definition 0.* Replication Lag

The time taken for replicas to update *stale* data.



*Eventually, all replicas must become consistent*

The system is *eventually consistent*.

*Eventual Consistency*

Sufficient? Problems?



**Brae Webb**  
@braewebb



**Brae Webb**

@braewebb

Name:	<input type="text" value="Brae"/>
<input type="button" value="Cancel"/>	<input type="button" value="Save"/>





**Brae Webb**

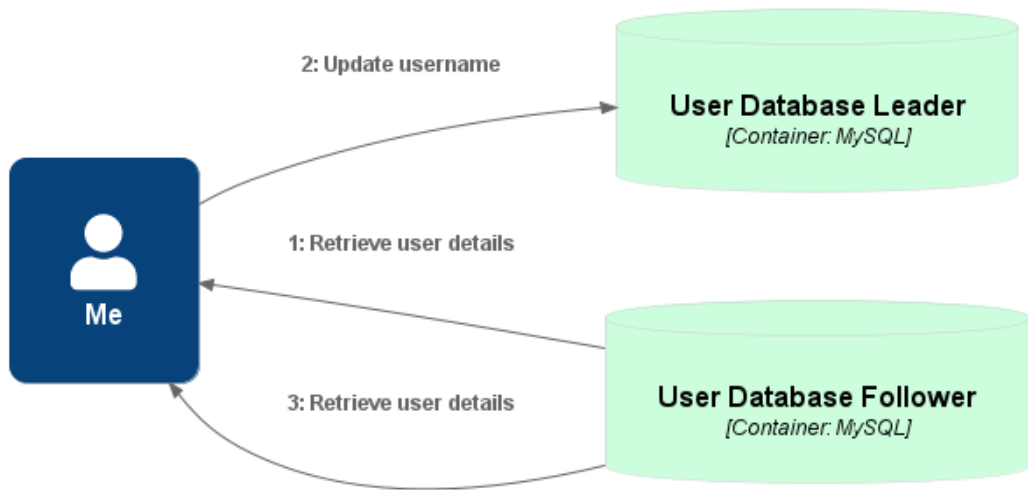
@braewebb

Name:	<input type="text" value="Brae"/>
<input type="button" value="Cancel"/>	<input type="button" value="Save"/>



**Brae Webb**

@braewebb



*Definition 0.* Read-your-writes Consistency

Users always see the updates that *they have made* (even though others see stale data).



**Brae Webb**

@braewebb

My fist post



**Brae Webb**

@braewebb

My fist post



**Brae Webb**

@braewebb

My first post



**Brae Webb**

@braewebb

My fist post



**Brae Webb**

@braewebb

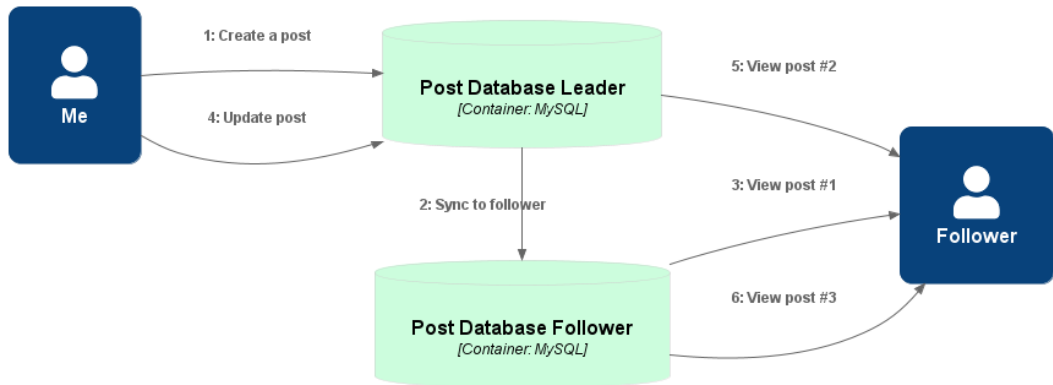
My first post



**Brae Webb**

@braewebb

My fist post



*Definition 0.* Monotonic Reads

Once a user reads an updated value, they don't later see the old value.



*Definition 0.* Causal Consistency

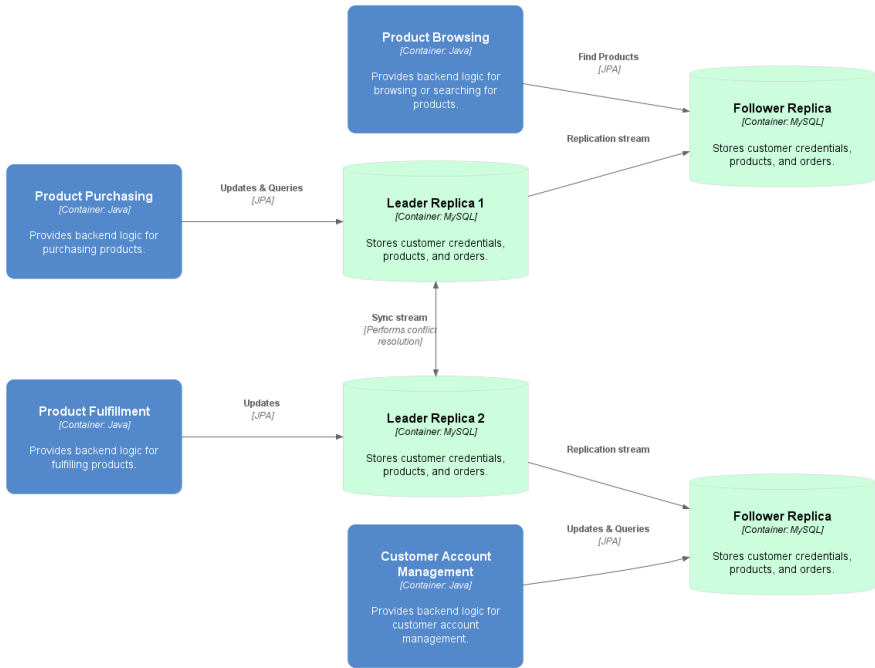
Causally related updates appear in order (e.g., comments appear under the right post).

## Summary

- Leader-follower databases allow *reads to scale* more effectively.
- Asynchronous propagation weakens consistency to *eventually consistent*.
- Leader-follower databases still have a *leader write bottle-neck*.

*Second approach*

Multi-leader Replication



## *Why multi-leader?*

- If you have multiple leaders, you can write to any, allowing *writes to scale*.

## *Why multi-leader?*

- If you have multiple leaders, you can write to any, allowing *writes to scale*.
- A leader going down doesn't prevent writes, giving *better fault-tolerance*.

*Question*

What might go wrong?

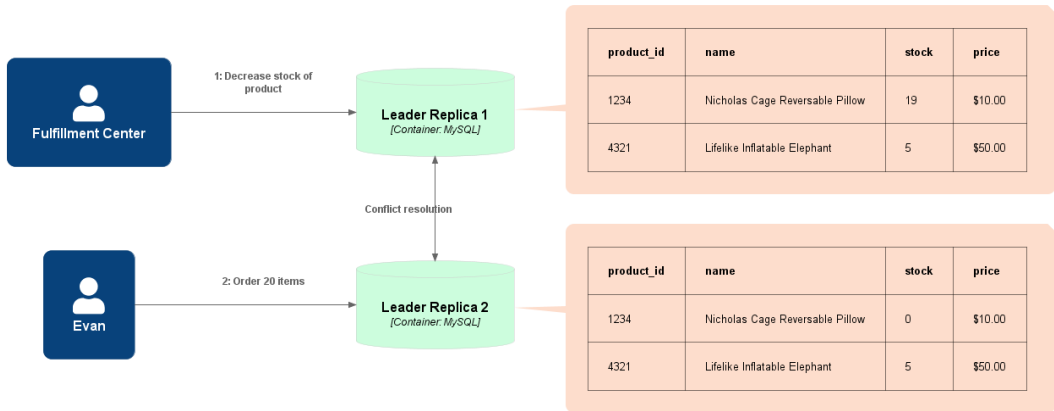
*Question*

What might go wrong?

*Answer*

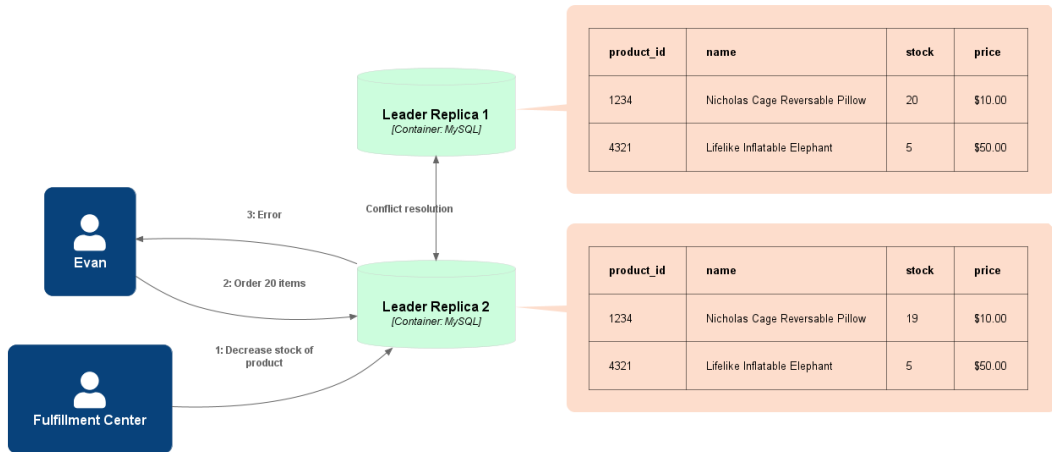
Write conflicts





*Where possible*

Avoid write conflicts



*Where impossible*  
Convergence

## *Convergence Strategies*

- Assign each *write* a unique ID

## *Convergence Strategies*

- Assign each *write* a unique ID
- Assign each *leader replica* a unique ID

## *Convergence Strategies*

- Assign each *write* a unique ID
- Assign each *leader replica* a unique ID
- Custom resolution logic



1: Decrease stock of product



table	row	column	value
products	1234	stock	0



2: Order 20 items



Conflict resolution

product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	19	\$10.00
4321	Lifelike Inflatable Elephant	5	\$50.00

product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	0	\$10.00
4321	Lifelike Inflatable Elephant	5	\$50.00

table	row	column	value
products	1234	stock	19



## *Resolving Conflicts*

**On Write** When a conflict is first noticed, take proactive resolution action.

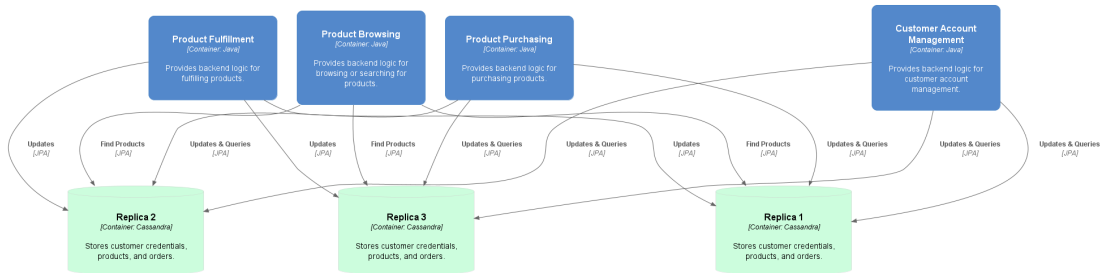
Example: Last Write Wins (LWW)  
in DynamoDB.

**On Read** Stores all conflicting versions on write. When a conflict is next read, ask for a resolution.

Example: *git pull*

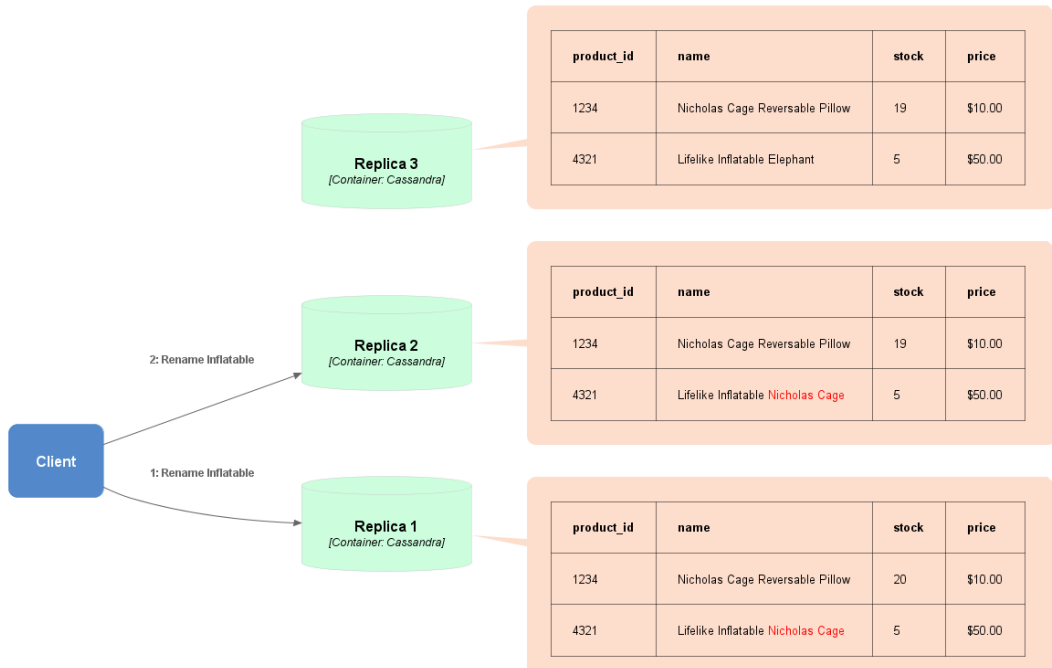
*Third Approach*

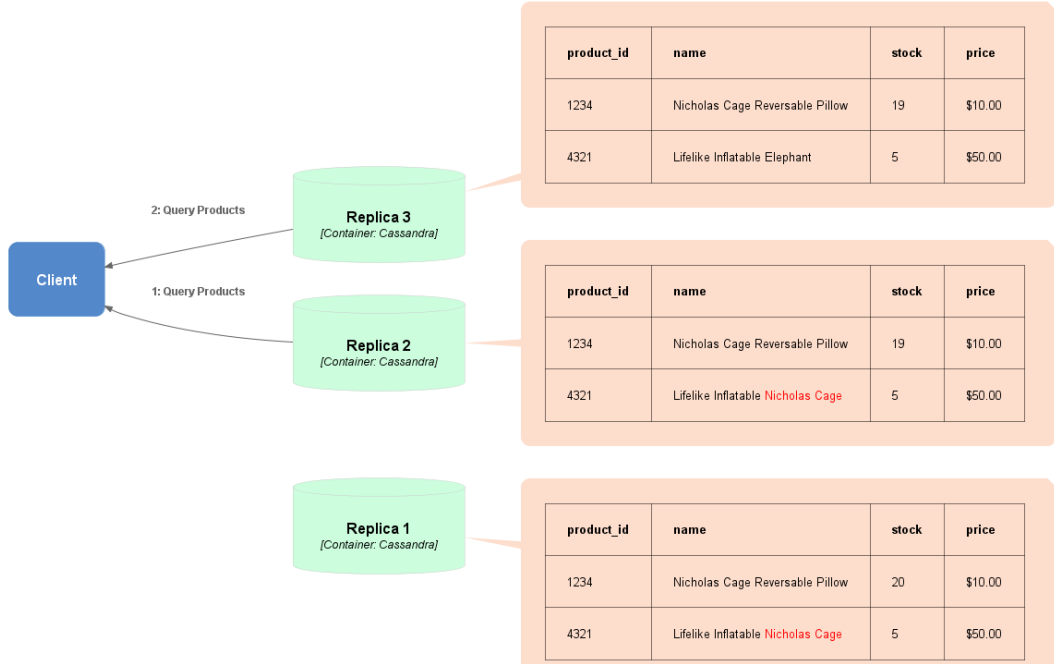
Leaderless Replication



*How do they work?*

Each read/write is sent to *multiple* replicas.





*How are changes propagated?*

- Read Repair

*How are changes propagated?*

- Read Repair
- Anti-Entropy Process



*Question*

How do we know it's consistent?



1: Query Products



product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	19	\$10.00
4321	Lifelike Inflatable Elephant	5	\$50.00



product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	19	\$10.00
4321	Lifelike Inflatable <b>Nicholas Cage</b>	5	\$50.00



product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	20	\$10.00
4321	Lifelike Inflatable <b>Nicholas Cage</b>	5	\$50.00

*Question*

How do we know it's consistent?

*Question*

How do we know it's consistent?

*Answer*

Quorum Reads and Writes

## Quorum Consistency

$$w + r > n$$

$n$  total replicas

$w$  amount of replicas to *write* to

$r$  amount of replicas to *read* from

## Quorum Consistency

$$2 + 2 > 3$$

$n$  total replicas

$w$  amount of replicas to *write* to

$r$  amount of replicas to *read* from

## Quorum Consistency

$$1 + 3 > 3$$

$n$  total replicas

$w$  amount of replicas to *write* to

$r$  amount of replicas to *read* from



$n$  total replicas

$w$  amount of replicas to *write* to

$r$  amount of replicas to *read* from



*Question*

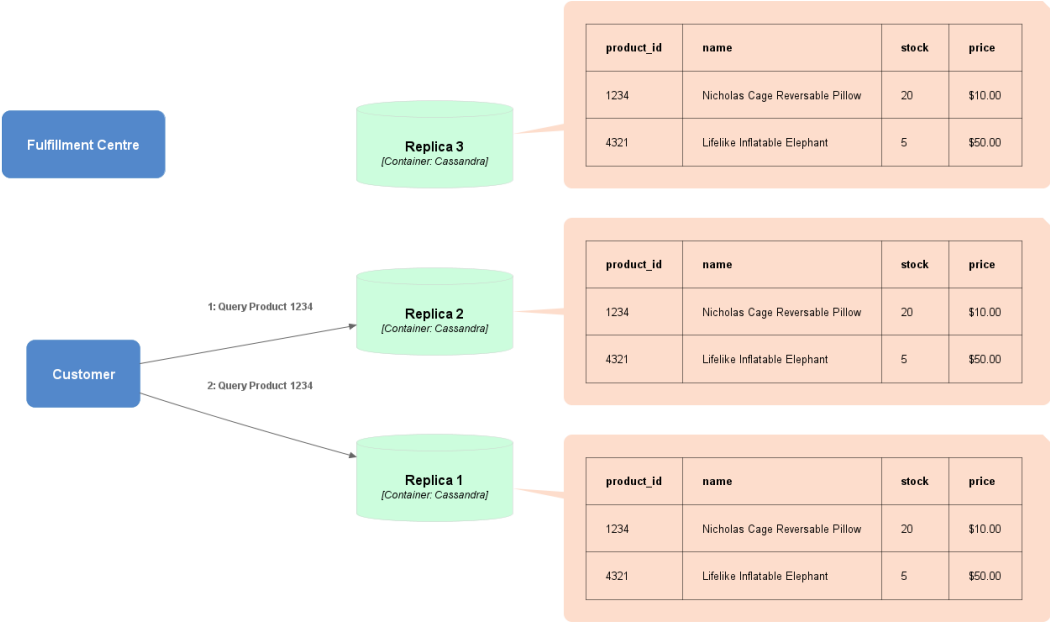
What about write conflicts?

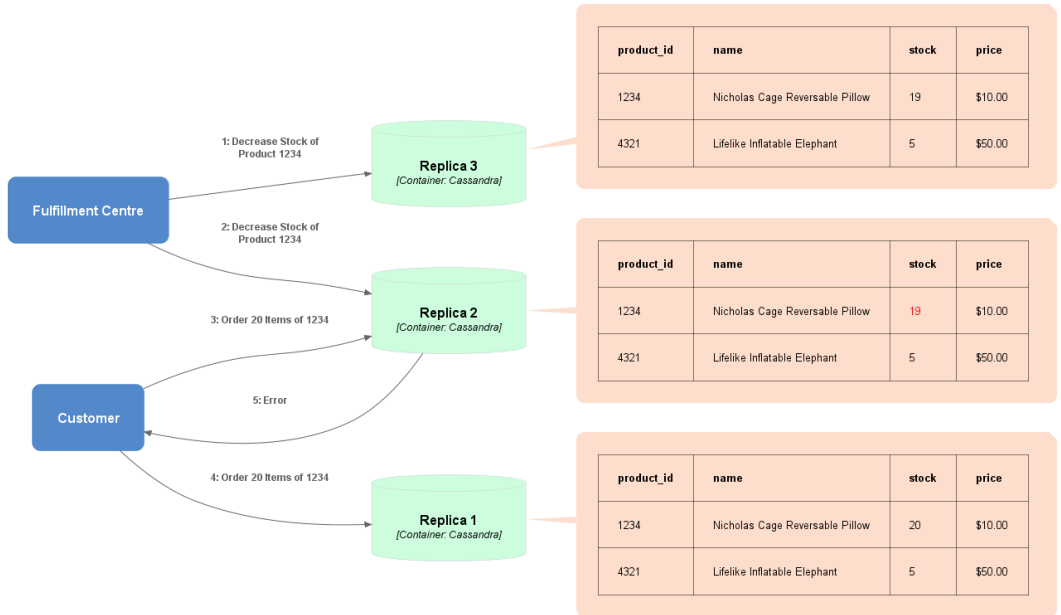
*Question*

What about write conflicts?

*Answer*

Same problem as with Multi-leader replication.





## *Summary*

- *Replication* copies data to multiple replicas.

## *Summary*

- *Replication* copies data to multiple replicas.
- *Leader-based* replication is most common and simplest.

## *Summary*

- *Replication* copies data to multiple replicas.
- *Leader-based* replication is most common and simplest.
- Replication introduces *eventual consistency*.

## Summary

- *Replication* copies data to multiple replicas.
- *Leader-based* replication is most common and simplest.
- Replication introduces *eventual consistency*.
- *Multi-leader* replication scales writes as well as reads but introduces *write conflicts*.



## Summary

- *Replication* copies data to multiple replicas.
- *Leader-based* replication is most common and simplest.
- Replication introduces *eventual consistency*.
- *Multi-leader* replication scales writes as well as reads but introduces *write conflicts*.
- *Leaderless* replication is another approach which keeps the problems of multi-leader.

*Question*

How do we fix database scaling issues?

*Question*

How do we fix database scaling issues?

*Answer*

- *Replication*
- Partitioning
- Independent databases

*Question*

How do we fix database scaling issues?

*Answer*

- Replication
- *Partitioning*
- Independent databases

*Definition 0.* Partitioning

Split the data of a system onto multiple nodes.

These nodes are *partitions*.

### Application Database

[Container: MySQL]

Stores customer credentials,  
products, and orders.

product_id	name	stock	price
4321	Lifelike Elephant Inflatable	5	\$50.00

### Application Database

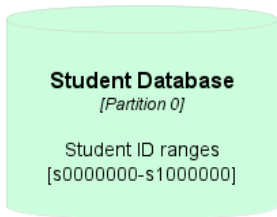
[Container: MySQL]

Stores customer credentials,  
products, and orders.

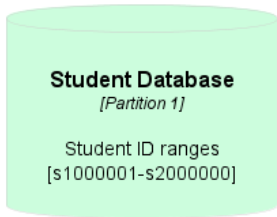
product_id	name	stock	price
1234	Nicholas Cage Reversible Pillow	10	\$10.00

*Question*

How should we decide which data is stored where?



student_id	name	...
s0746283	Bobby Tables	...
...	...	...



student_id	name	...
s1637285	Brae Webb	...
...	...	...



*Question*

What is the problem with this?

*Question*

What is the problem with this?

*Answer*

Over time some partitions become inactive, while others receive almost all load.

*Question*

How should we decide where data is stored?

*Question*

How should we decide where data is stored?

*Answer*

Maximize spread of requests, avoiding *skewing*.

*Question*

Have we seen this before?

*Question*

Have we seen this before?

*Answer*

Hashing?

*Question*

What is the problem with this?

*Question*

What is the problem with this?

*Answer*

Range queries are inefficient, i.e. get all students between s4444444 and s4565656.

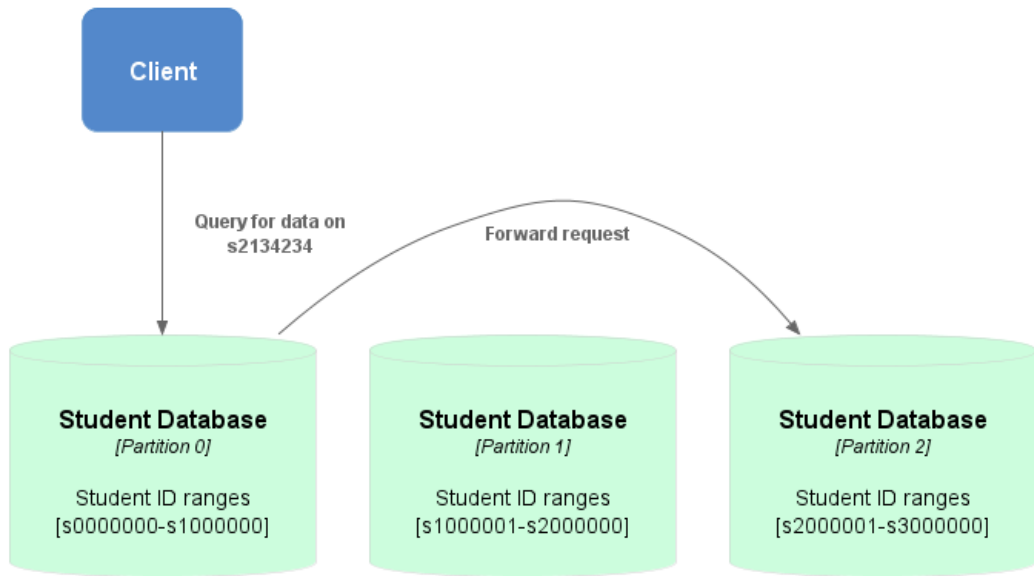


*Question*

How do we route queries?

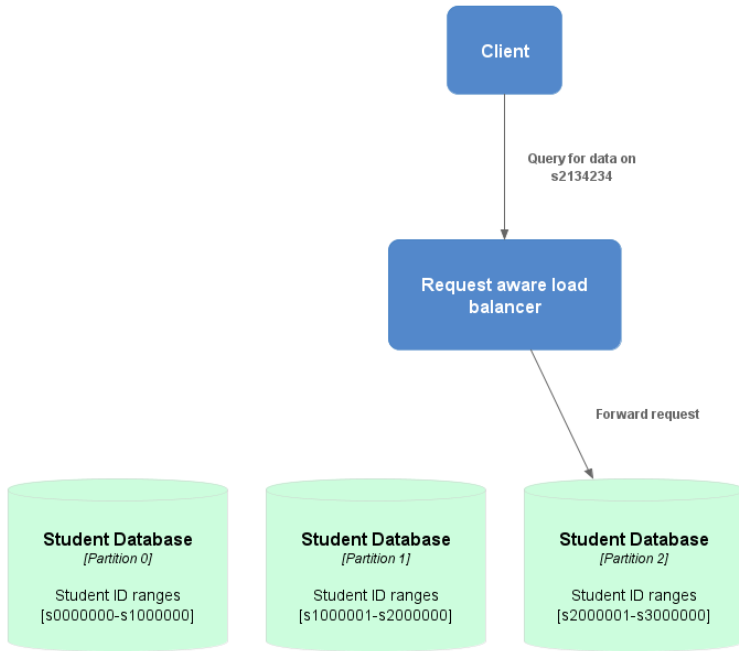
### *Query-Insensitive Load Balancer*

Randomly route to any node, responsibility of the node to re-route to the correct node.



## *Query-Sensitive Load Balancer*

Load balancer understands which queries should be forwarded to which node.



### *Client-aware Queries*

Place the responsibility on clients to choose the correct node.

Client

Query for data on  
s2134234

**Student Database**

*[Partition 0]*

Student ID ranges  
[s0000000-s1000000]

**Student Database**

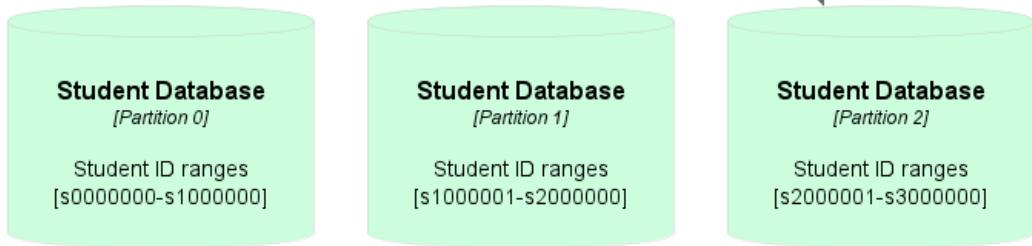
*[Partition 1]*

Student ID ranges  
[s1000001-s2000000]

**Student Database**

*[Partition 2]*

Student ID ranges  
[s2000001-s3000000]



## *Summary*

- *Partitioning* splits data across multiple nodes.



## Summary

- *Partitioning* splits data across multiple nodes.
- Requires a *consistent method* to choose appropriate node.

## Summary

- *Partitioning* splits data across multiple nodes.
- Requires a *consistent method* to choose appropriate node.
- Partitioning by *primary key* can create *skewing*.

## Summary

- *Partitioning* splits data across multiple nodes.
- Requires a *consistent method* to choose appropriate node.
- Partitioning by *primary key* can create *skewing*.
- Partitioning by *hash* makes range queries less efficient.

## Summary

- *Partitioning* splits data across multiple nodes.
- Requires a *consistent method* to choose appropriate node.
- Partitioning by *primary key* can create *skewing*.
- Partitioning by *hash* makes range queries less efficient.
- Three approaches to *routing requests*.

## *Disclaimer*

We have ignored the *hard* parts of replication.

*Question*

How do we fix database scaling issues?

*Question*

How do we fix database scaling issues?

*Answer*

- Replication
- *Partitioning*
- Independent databases

*Question*

How do we fix database scaling issues?

*Answer*

- Replication
- Partitioning
- *Independent databases*



## *Summary*

- Replications

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless
  - Eventual consistency

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless
  - Eventual consistency
  - Write conflicts

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless
  - Eventual consistency
  - Write conflicts
- Partitioning

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless
  - Eventual consistency
  - Write conflicts
- Partitioning
  - Consistent method to pick nodes for data

## *Summary*

- Replications
  - Leader-based, multi-leader, and leaderless
  - Eventual consistency
  - Write conflicts
- Partitioning
  - Consistent method to pick nodes for data
  - Avoiding skewing