Distributed Systems II

Software Architecture

Brae Webb & Richard Thomas

March 31, 2025

Distributed Systems Series

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

Distributed II *Complexities* of *stateful* systems.

Distributed III *Hard problems* in distributed systems.

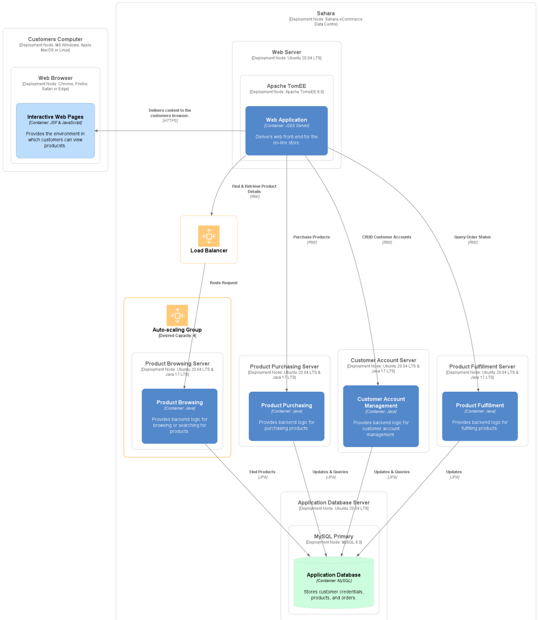
Distributed Systems Series

Distributed I Reliability and scalability of stateless systems.

Distributed II *Complexities* of *stateful* systems.

Distributed III Hard problems in distributed systems.

Previously in CSSE6400...



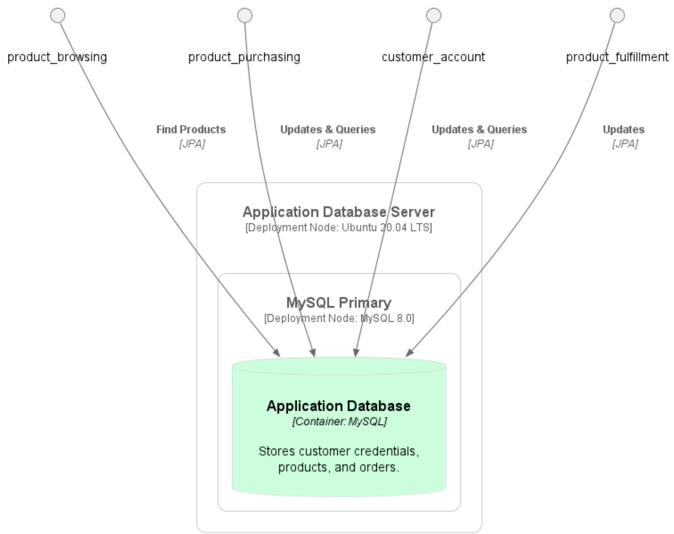
- We scaled a stateless service.
- Stateless: Services don't require persistent data *between* requests.
- Persistent state is saved in the database.
- This is normally easy to do.

Question

What is the *problem*?

The database

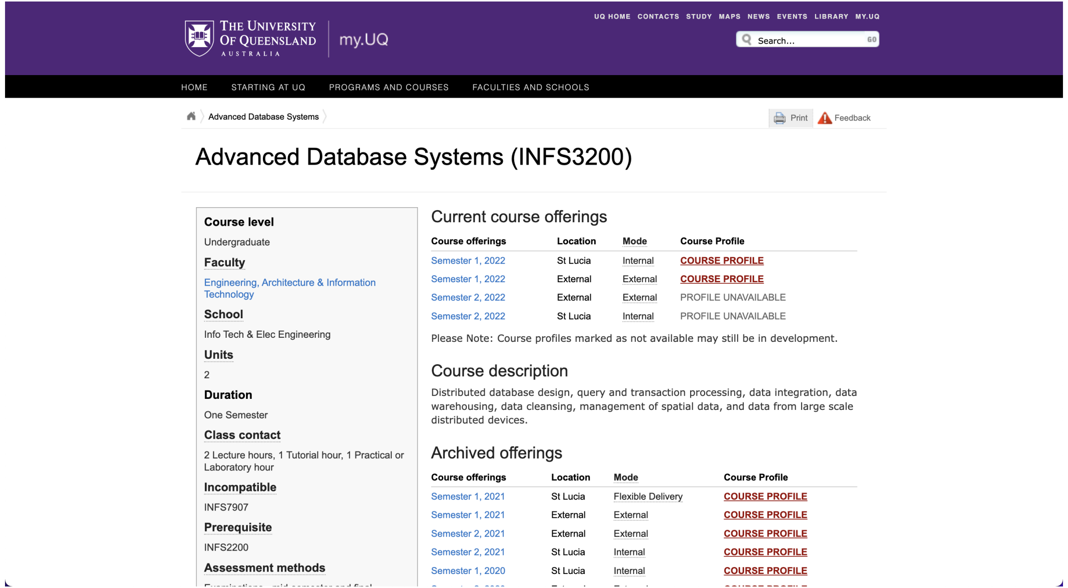
Database



- Database has state, persistent data.
- This is much harder to scale.

Disclaimer

This is *not* a database course.


THE UNIVERSITY OF QUEENSLAND AUSTRALIA | my.UQ

UQ HOME CONTACTS STUDY MAPS NEWS EVENTS LIBRARY MY.UQ

HOME STARTING AT UQ PROGRAMS AND COURSES FACULTIES AND SCHOOLS

Advanced Database Systems

Print Feedback

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

Current course offerings

Course offerings	Location	Mode	Course Profile
Semester 1, 2022	St Lucia	Internal	COURSE PROFILE
Semester 1, 2022	External	External	COURSE PROFILE
Semester 2, 2022	External	External	PROFILE UNAVAILABLE
Semester 2, 2022	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
Semester 1, 2021	St Lucia	Flexible Delivery	COURSE PROFILE
Semester 1, 2021	External	External	COURSE PROFILE
Semester 2, 2021	External	External	COURSE PROFILE
Semester 2, 2021	St Lucia	Internal	COURSE PROFILE
Semester 1, 2020	St Lucia	Internal	COURSE PROFILE

This is a database course.

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

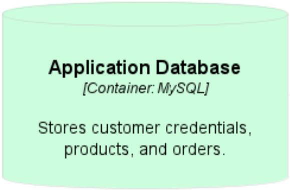
Question

What is *replication*?

Definition 0. Replication

Data copied across multiple different machines.

product_id	name	stock	price
1234	Nicholas Cage Reversable Pillow	10	$10.00
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 Reversable 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*.

- Scalability
- Reliability
- Performance

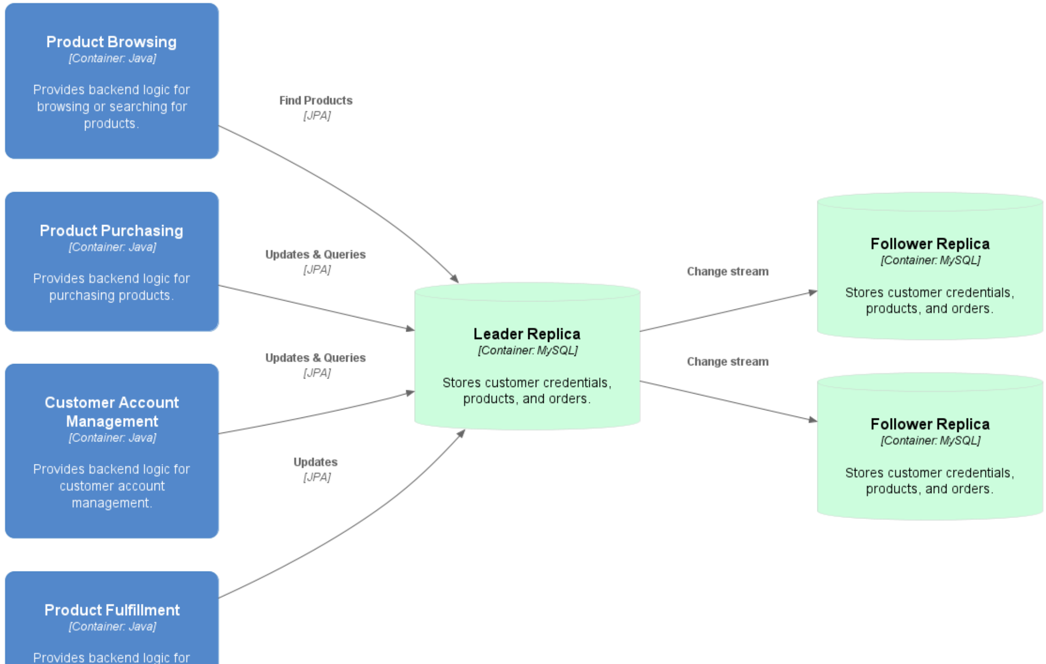
Question

How do we replicate our data?

- Easy without updates, just copy it.
- Updates, or writes, must *propagate* changes.

First approach

Leader-Follower Replication



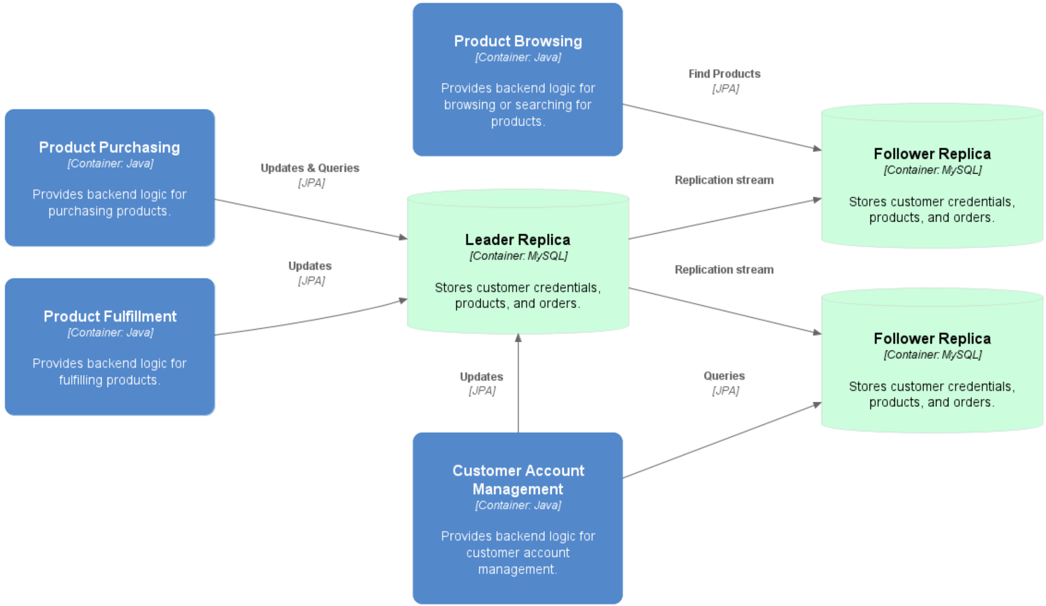
- Leader-Follower is the most common implementation.
- Multiple followers, only *one* leader.

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.

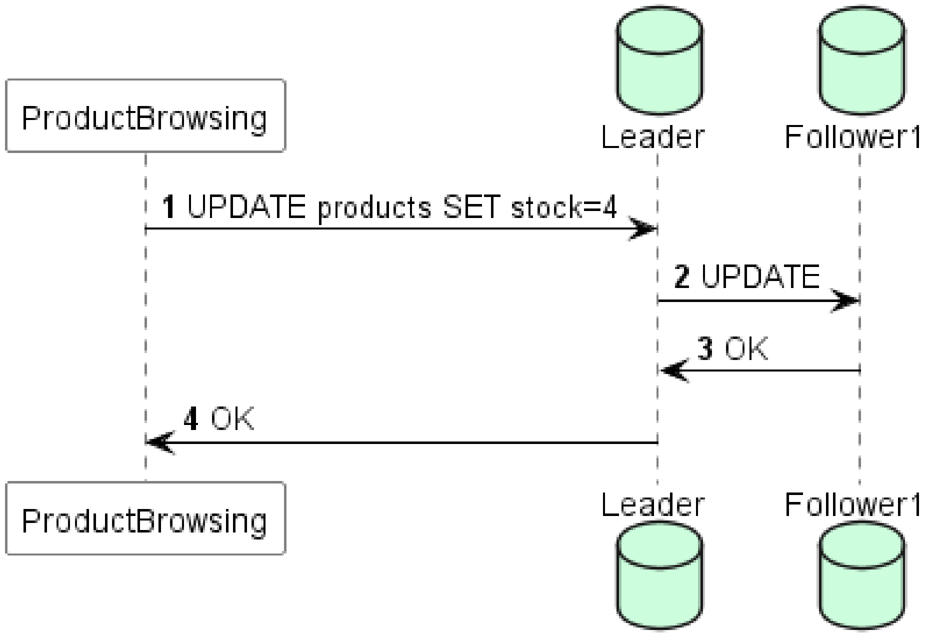
On read Any *replica* can be queried.



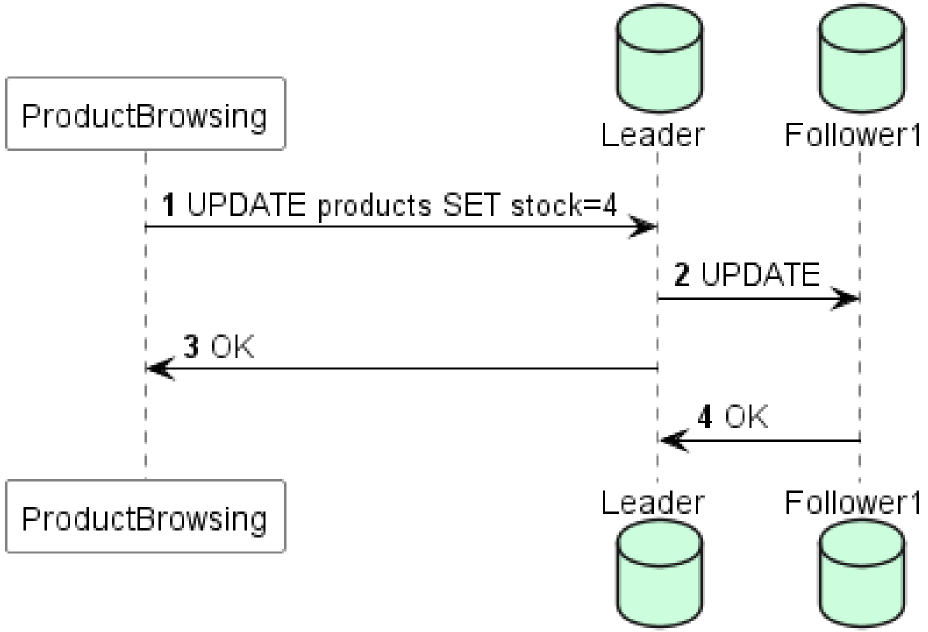
- Built-in to PostgreSQL, MySQL, MongoDB, RethinkDB, and Espresso.
- Can be added to Oracle and SQL Server.

Propagating changes

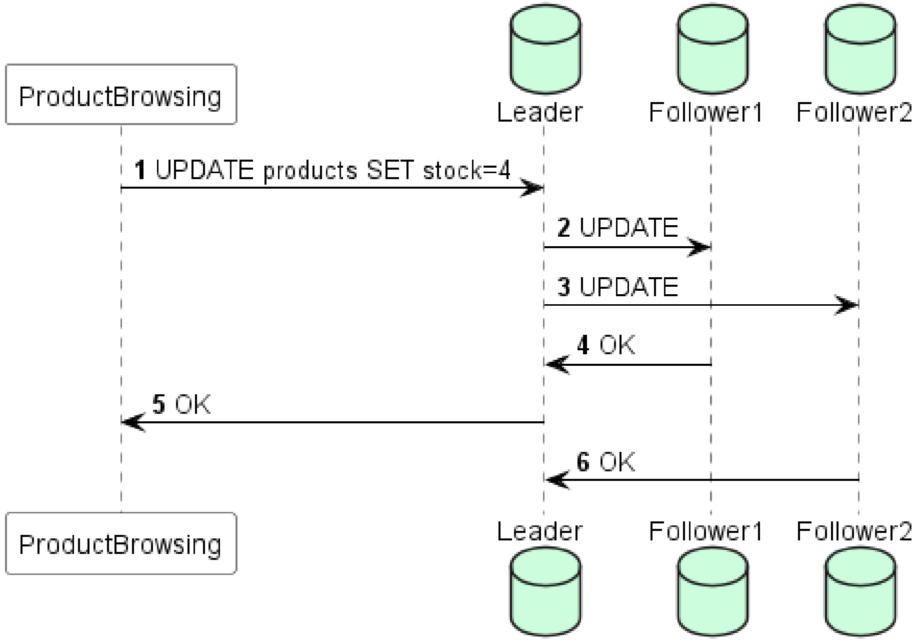
Synchronous vs. *Asynchronous*



Synchronous update.



Asynchronous update.



- What could go wrong here?
- *Follower1* can get out of sync with *Follower2*.
- Following material deals with *leader* or a *replica* going down.

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

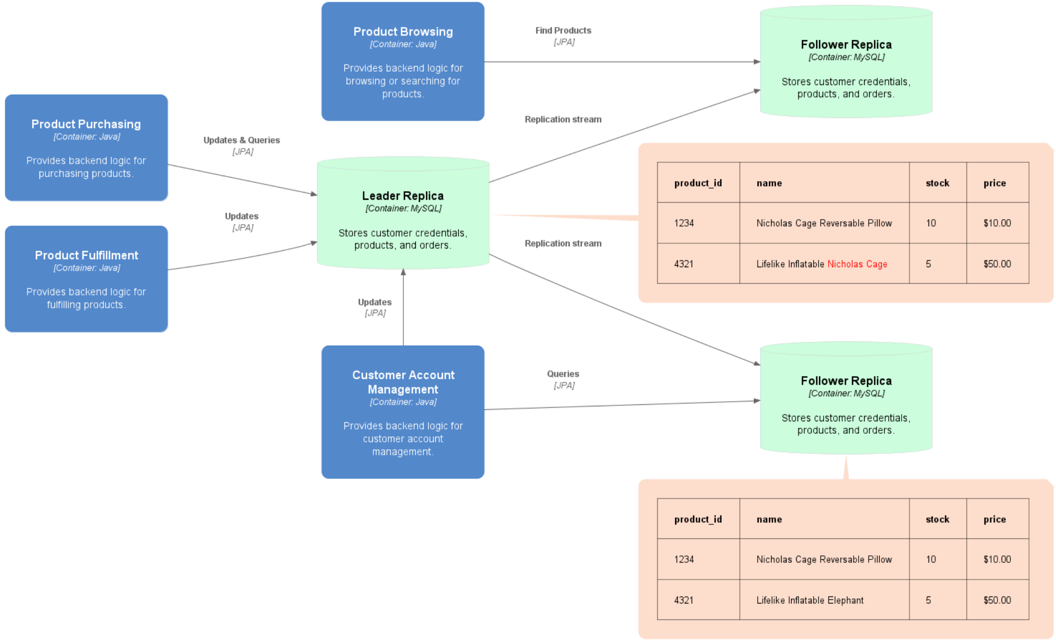
- Writes *don't* have to *wait* for propagation.
- If the leader goes down before propagating, the *write is lost*.

Asynchronous Propagation

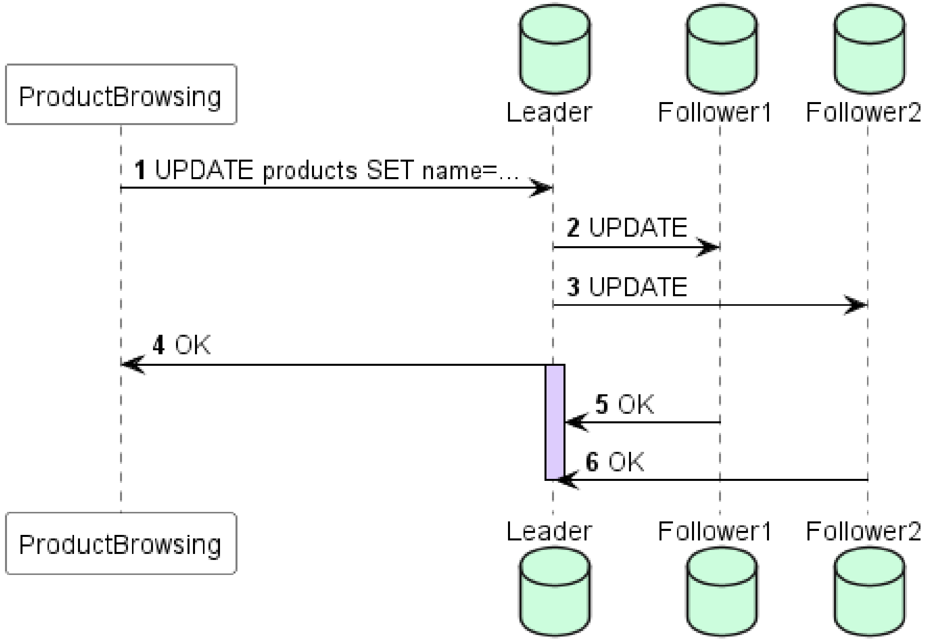
- Writes *don't* have to *wait* for propagation.
- If the leader goes down before propagating, the *write is lost*.
- Replicas can have out-dated or *stale* data.

Definition 0. Replication Lag

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



Replication Lag: Time it takes for the product name change to update across *all* followers.



The purple lifeline bar is *replication lag*.

Eventually, all replicas must become consistent

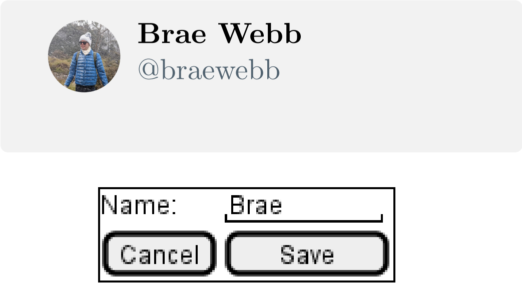
The system is *eventually consistent*.

- If writes stop for long enough.
- Eventually is intentionally *ambiguous*.

Problems?

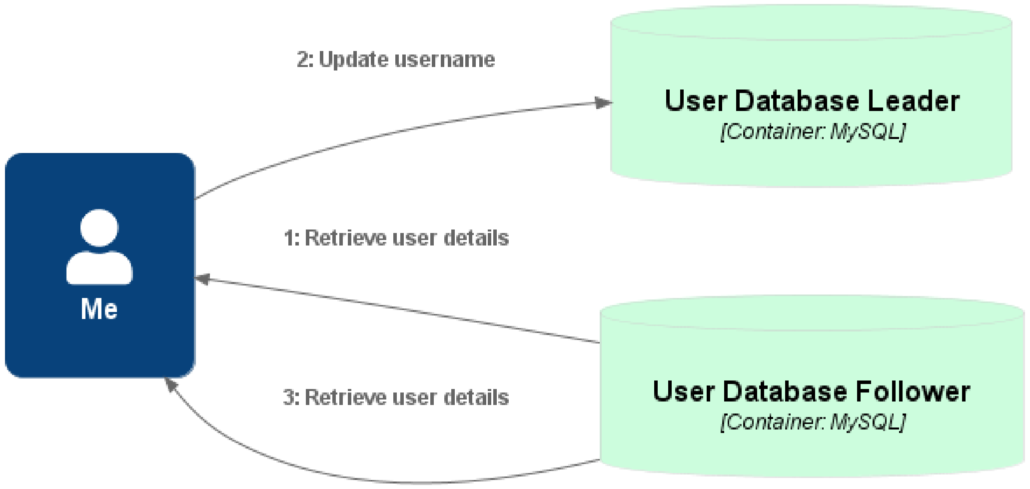
Brae Webb
@braewebb

1. Read user details.



1. Read user details.
2. Decide I don't like my name.
3. Update name.

1. Read user details.
2. Decide I don't like my name.
3. Update name.
4. Read user details.



- Typical interaction in simple Leader-Follower replication.
- Write is to *leader*.
- Read is from *follower*.
- *Replication lag* means reading a field immediately after updating it *may* lead to reading *stale* data.

Definition 0. Read-your-writes Consistency

Users always see the updates that *they have made*.

Doesn't care what other users see.

Brae Webb

@braewebb

My fist post

1. Misspell a tweet.

1. Misspell a tweet.
2. Correct spelling, and I see my *updated* tweet.

Brae Webb
@braewebb

My fist post

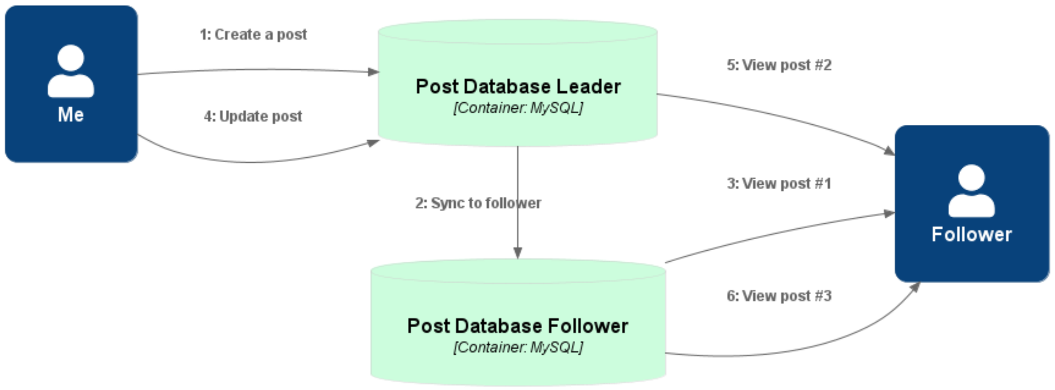
Brae Webb
@braewebb

My first post

Brae Webb
@braewebb

My fist post

1. Misspell a tweet.
2. Correct spelling, and I see my *updated* tweet.
3. Other users may still see *stale*, misspelt post.



- Go through each step in sequence.
- Step 6: 3rd view post, gets *old value*.

Definition 0. Monotonic Reads

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

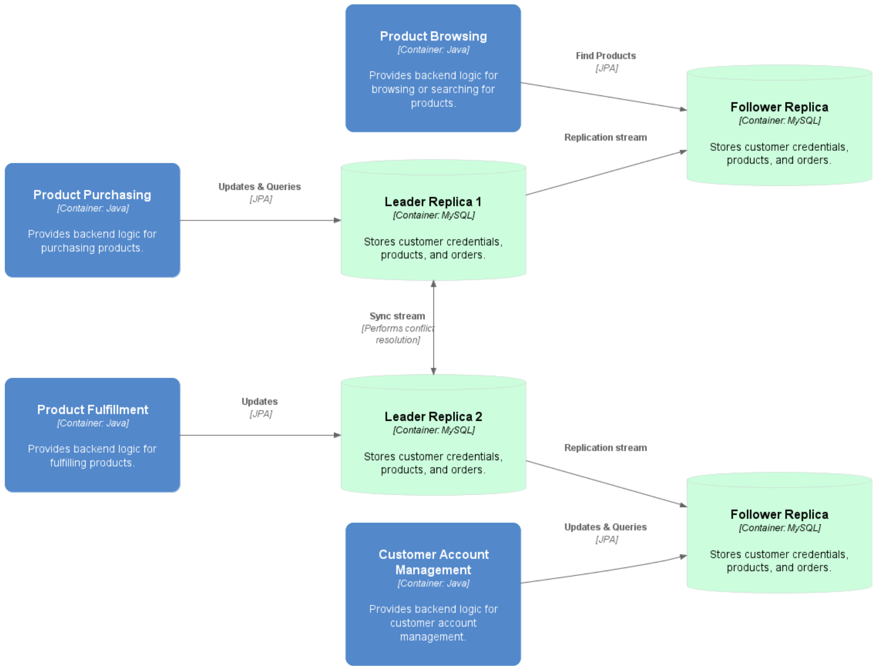
User doesn't travel back in time.

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



- Application can be partitioned to perform certain types of writes to a specific leader.
- Reads are from replicas, as with Leader-Follower 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*.

- Available via extensions in most databases, often not supported natively.
- Best to avoid where possible.
- Example: Globally distributed data centres.

Question

What might go wrong?

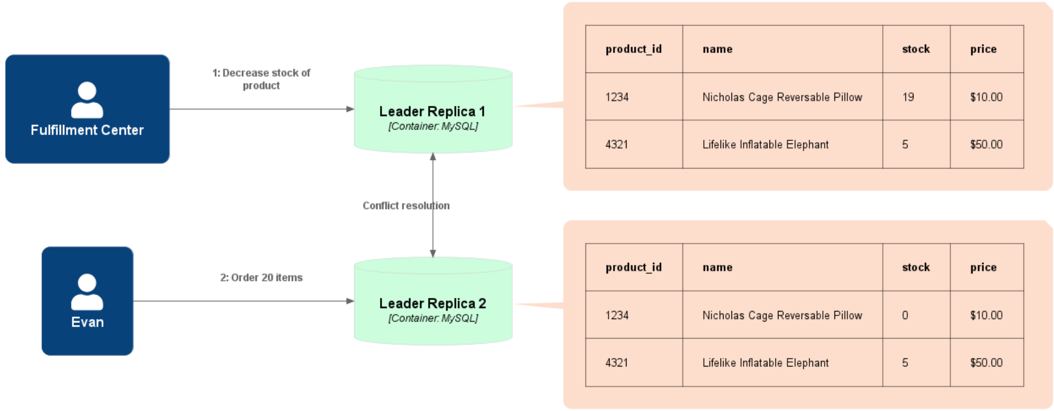
Question

What might go wrong?

Answer

Write conflicts

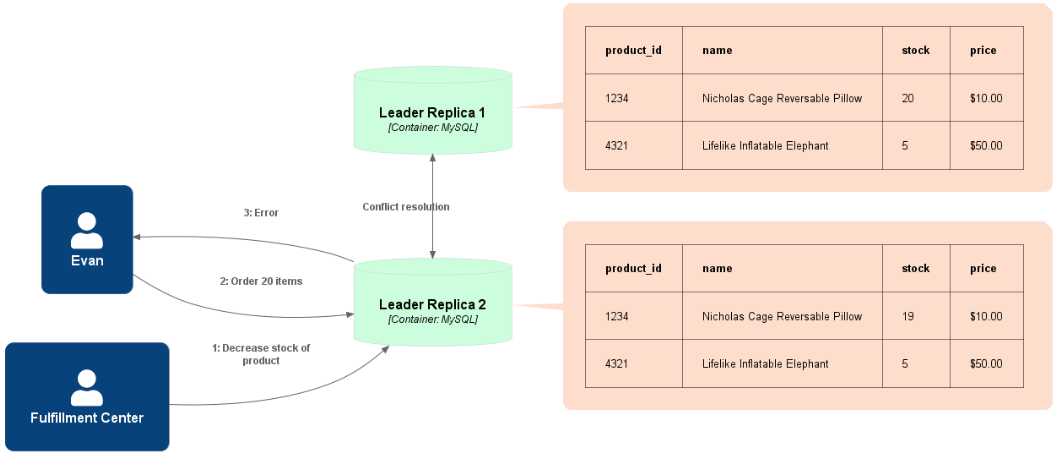
Conflict needs to be *resolved*.



1. Fulfilment centre finds faulty pillow and decreases inventory.
2. Customer orders 20 pillows, what they saw as the number available.
3. -1 Pillows?
4. How do we resolve this?

Where possible

Avoid write conflicts



product_id	name	stock	price
1234	Nicholas Cage Reversable Pillow	20	$10.00
4321	Lifelike Inflatable Elephant	5	$50.00

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

Requires application to ensure all writes to a field/table/shard are via the *same* leader.

Where impossible

Convergence

- Assign each *write* a unique ID.

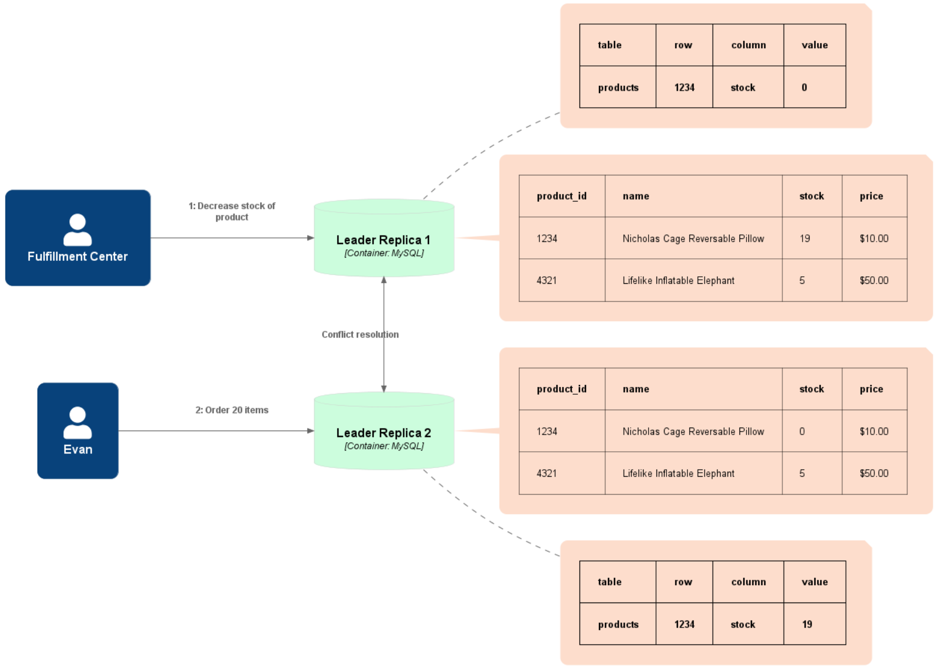
Convergence Strategies

- 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.

Yes, this can be *challenging*.



Fulfillment Center
1: Decrease stock of product
Leader Replica 1
[Container: MySQL]
Conflict resolution
Evan
2: Order 20 items
Leader Replica 2
[Container: MySQL]
table | row | column | value
products | 1234 | stock | 0
product_id | name | stock | price
1234 | Nicholas Cage Reversable Pillow | 19 | $10.00
4321 | Lifelike Inflatable Elephant | 5 | $50.00
product_id | name | stock | price
1234 | Nicholas Cage Reversable Pillow | 0 | $10.00
4321 | Lifelike Inflatable Elephant | 5 | $50.00
table | row | column | value
products | 1234 | stock | 19

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

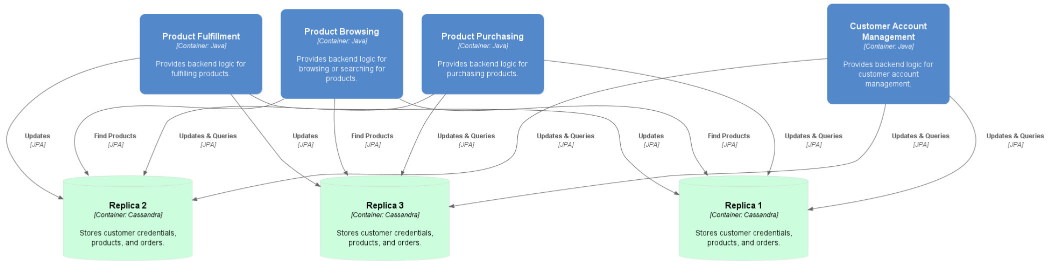
On Read When a conflict is next read, ask for a resolution.

- Bucardo allows a perl script for on write resolution.
- CouchDB prompts reads to resolve the conflict.

Third Approach

Leaderless Replication

- Early distributed databases were leaderless.
- Resurgance after Amazon created Dynamo.
- Dynamo is an internal service and *not* DynamoDB.
- Riak, Cassandra, and Voldemort are leaderless databases.

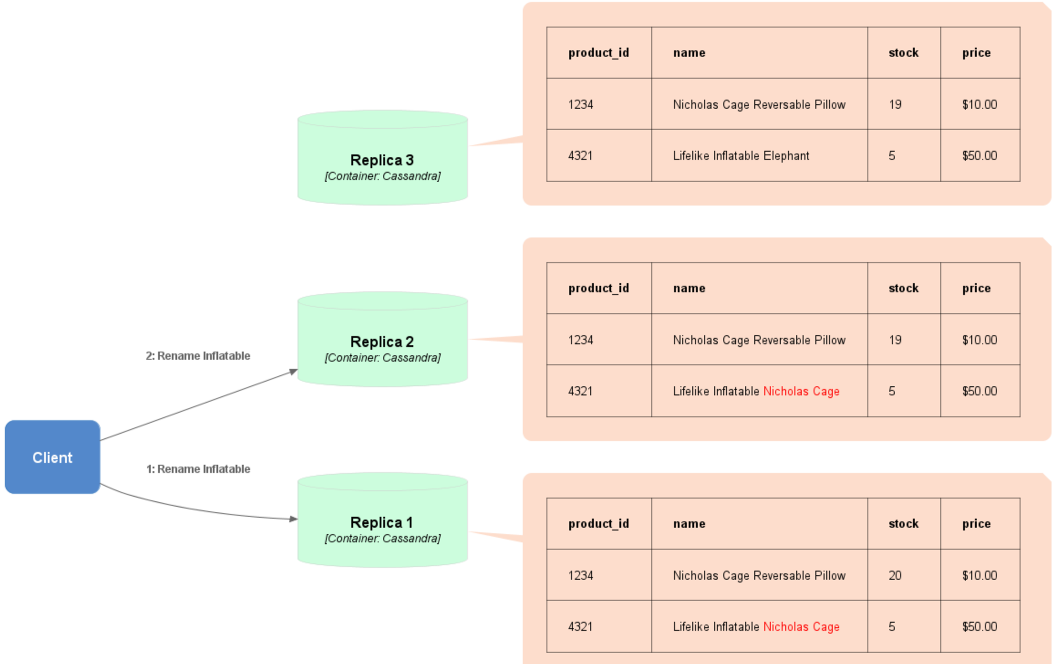


Reads and writes can be written to any node.

How do they work?

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

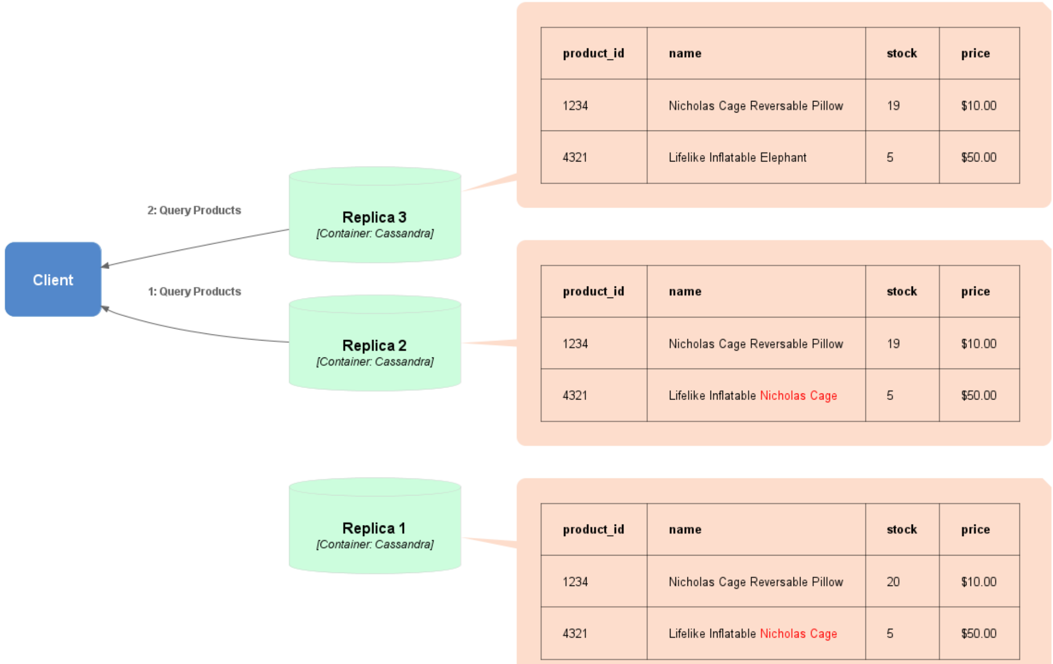
Leaderless Write



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

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

product_id	name	stock	price
1234	Nicholas Cage Reversable Pillow	20	$10.00
4321	Lifelike Inflatable Nicholas Cage	5	$50.00



Leaderless Read: At least one of the reads has the updated value.

How are changes propagated?

- Read Repair

1. Read Repair: Client detects stale data on read and writes updated data to that replica.

How are changes propagated?

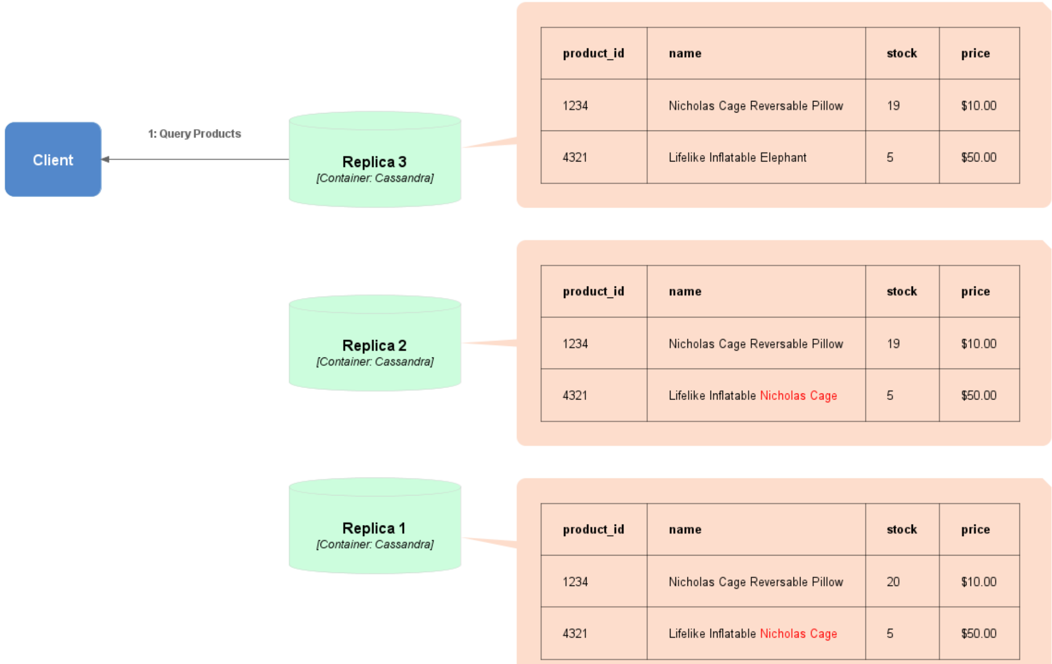
- Read Repair
- Anti-Entropy Process

1. Read Repair: Client detects stale data on read and writes updated data to that replica.

2. Anti-Entropy Process: Background process looks for stale or missing data and updates replicas.

Question

How do we know it's consistent?



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

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

product_id	name	stock	price
1234	Nicholas Cage Reversable Pillow	20	$10.00
4321	Lifelike Inflatable Nicholas Cage	5	$50.00

Reading from a single replica means we can't know if data is stale or inconsistent.

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

$$2 + 2 > 3$$

n total replicas

w amount of replicas to *write* to

r amount of replicas to *read* from

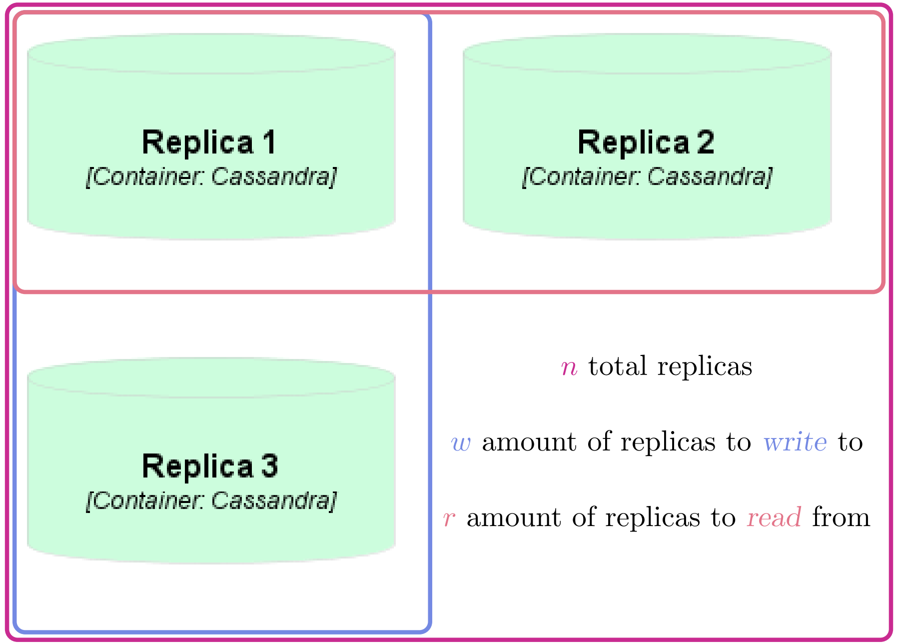
$$1 + 3 > 3$$

n total replicas

w amount of replicas to *write* to

r amount of replicas to *read* from

Nodes read from must overlap with the nodes written to.



- Graphical representation of previous equation.
- **Orange** inner group (*reads*), overlaps with **Blue** inner group (*writes*).
- Showing how reads overlap writes via a quorum.

Question

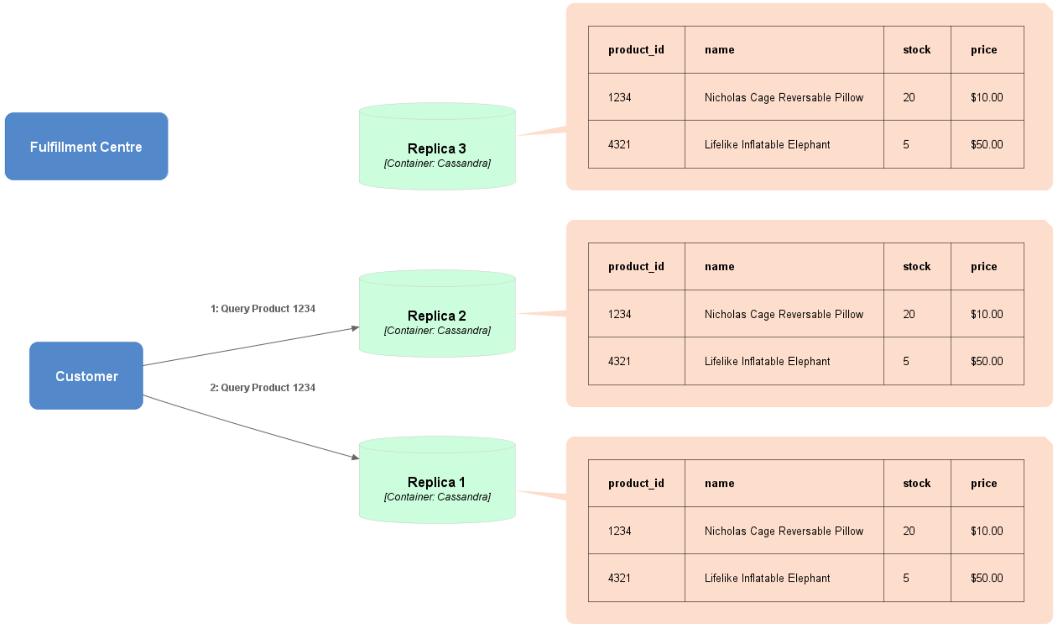
What about write conflicts?

Question

What about write conflicts?

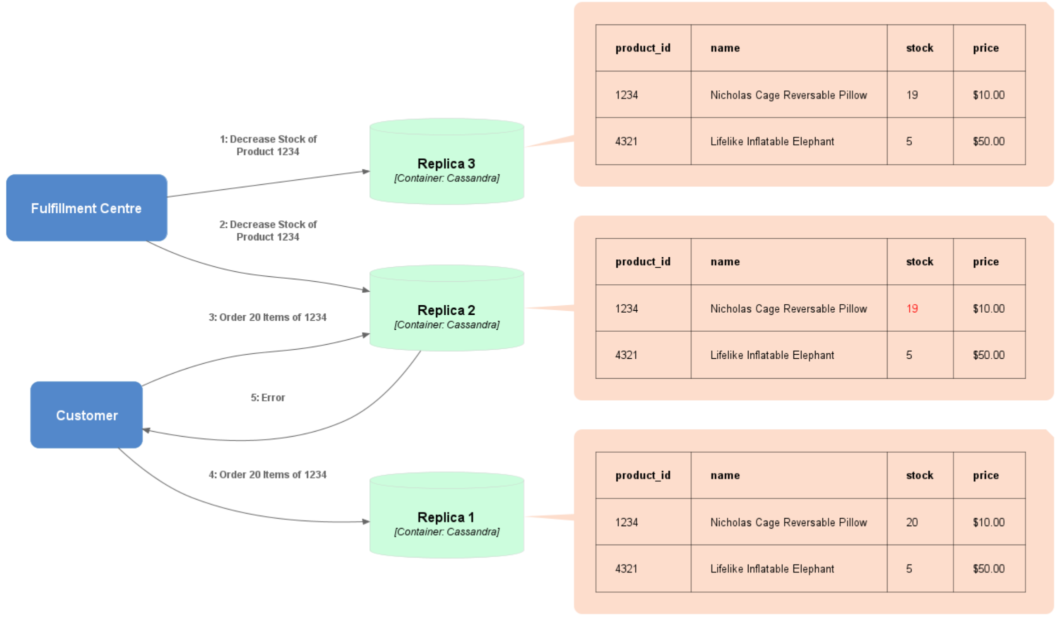
Answer

Same problem as with Multi-leader replication.



Same scenario as before:

1. Customer queries how many pillows are available.
2. Retrieves 20.



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

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

product_id	name	stock	price
1234	Nicholas Cage Reversable Pillow	20	$10.00
4321	Lifelike Inflatable Elephant	5	$50.00

Same scenario as before:

1. Fulfilment centre finds faulty pillow and decreases inventory.
2. Customer orders 20 pillows, what they saw as the number available.
3. -1 Pillows?
4. How do we resolve this?

Summary

- *Replication* copies data to multiple replicas.

Summary

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

Summary

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

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

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

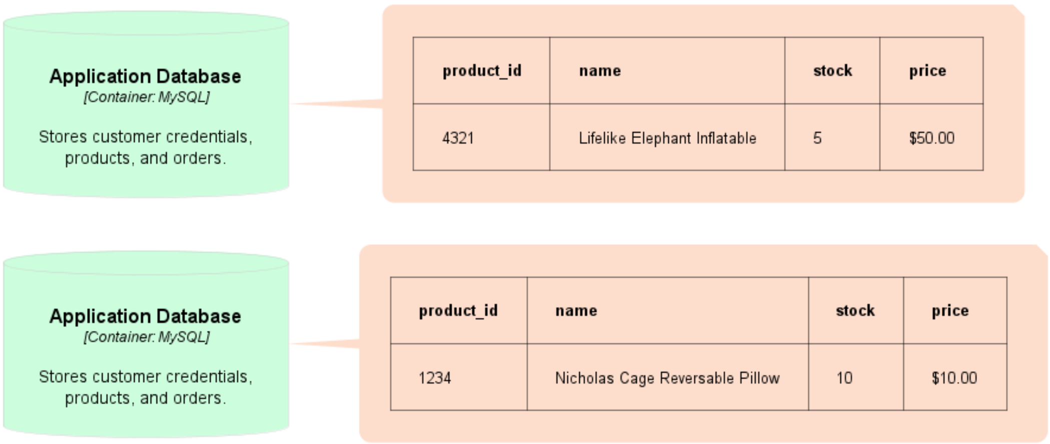
Answer

- Replication
- *Partitioning*
- Independent databases

Definition 0. Partitioning

Split the data of a system onto multiple nodes.
These nodes are *partitions*.

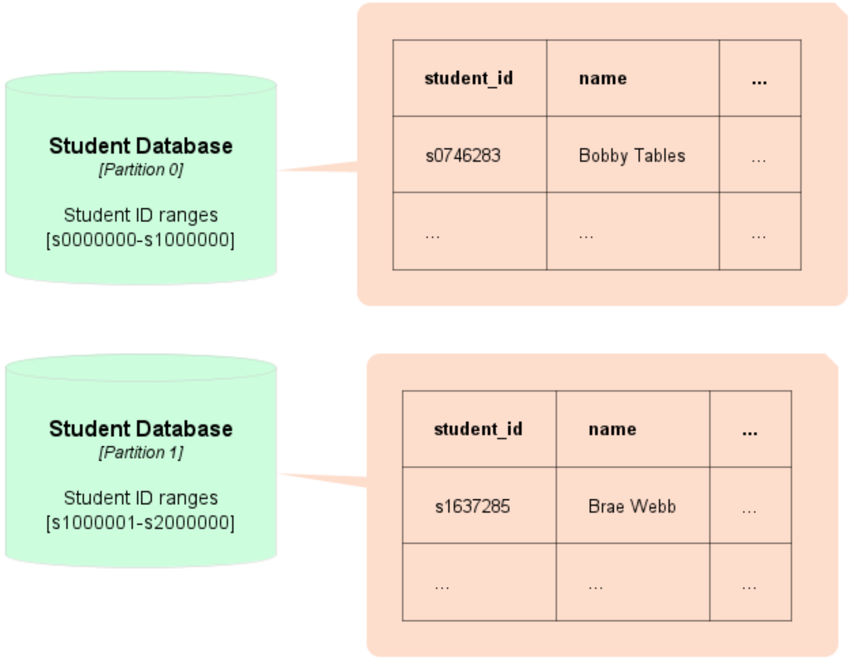
Also called shards, regions, tablets, etc.



- Pioneered in the 1980s.
- Allow scalability of large data, not just large load.
- Partitioning is normally combined with replication.

Question

How should we decide which data is stored where?



An example partitioning based on primary key, student ID.

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 which data is stored where?

Question

How should we decide which data is stored where?

Answer

Maximize spread of requests, avoiding *skewing*.

Question

Have we seen this before?

Question

Have we seen this before?

Answer

Hashing?

Hash tables hash entries to maximize the spread between buckets.

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.

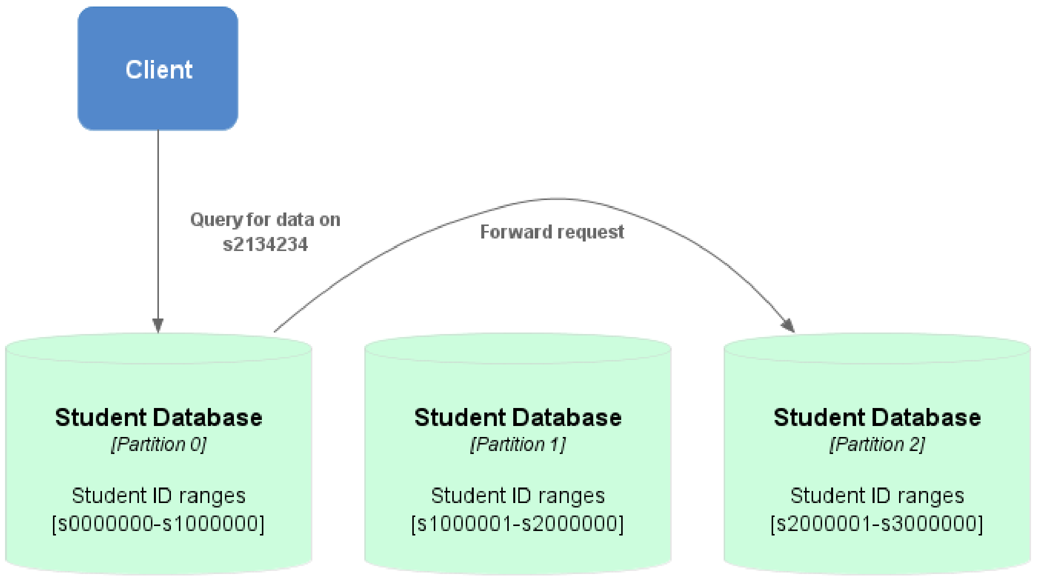
Question

How do we route queries?

Unlike stateless, only one node can process queries.

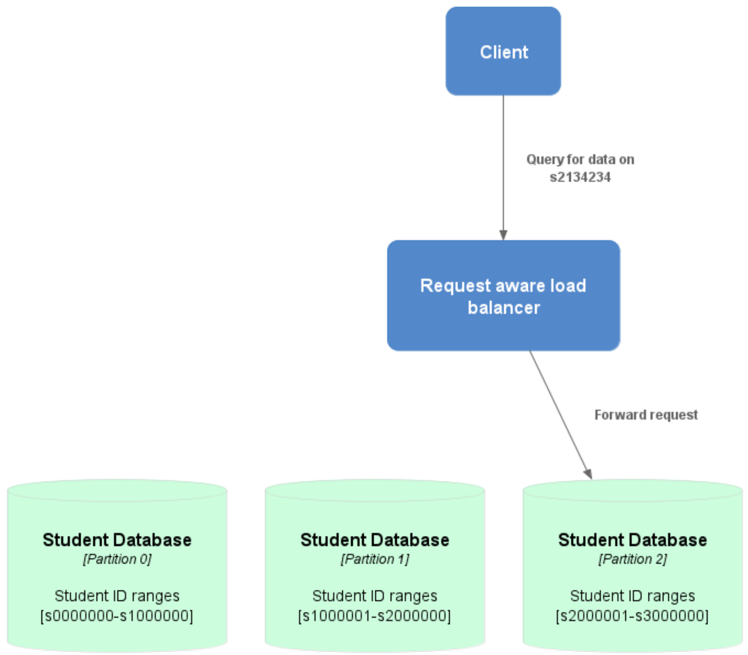
Query-insensitive Load Balancer

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


Client
Query for data on s2134234
Forward request
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]

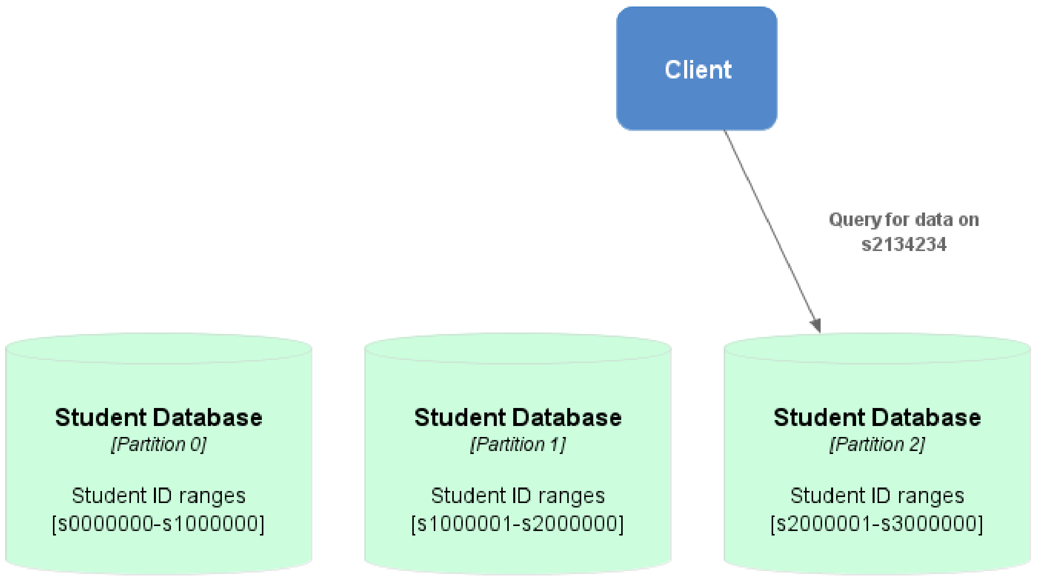
Query-sensitive Load Balancer

A load balancer which understands which queries should be forwarded to which node.


Client
Query for data on s2134234
Request aware load balancer
Forward request
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]

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]

- *Partitioning* splits data across multiple nodes.

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

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

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

- *Partitioning* splits data across multiple nodes.
- Requires a *consistent method* to chose 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

- 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