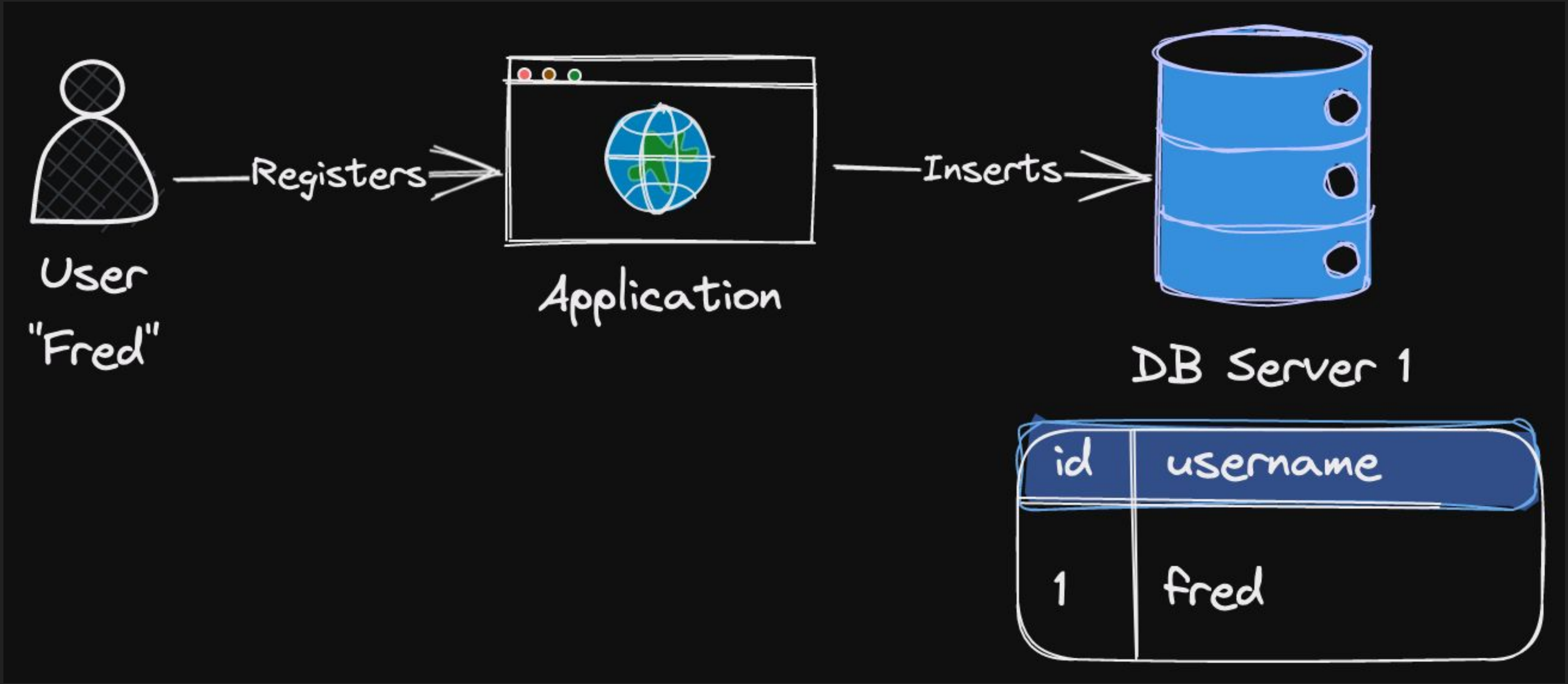
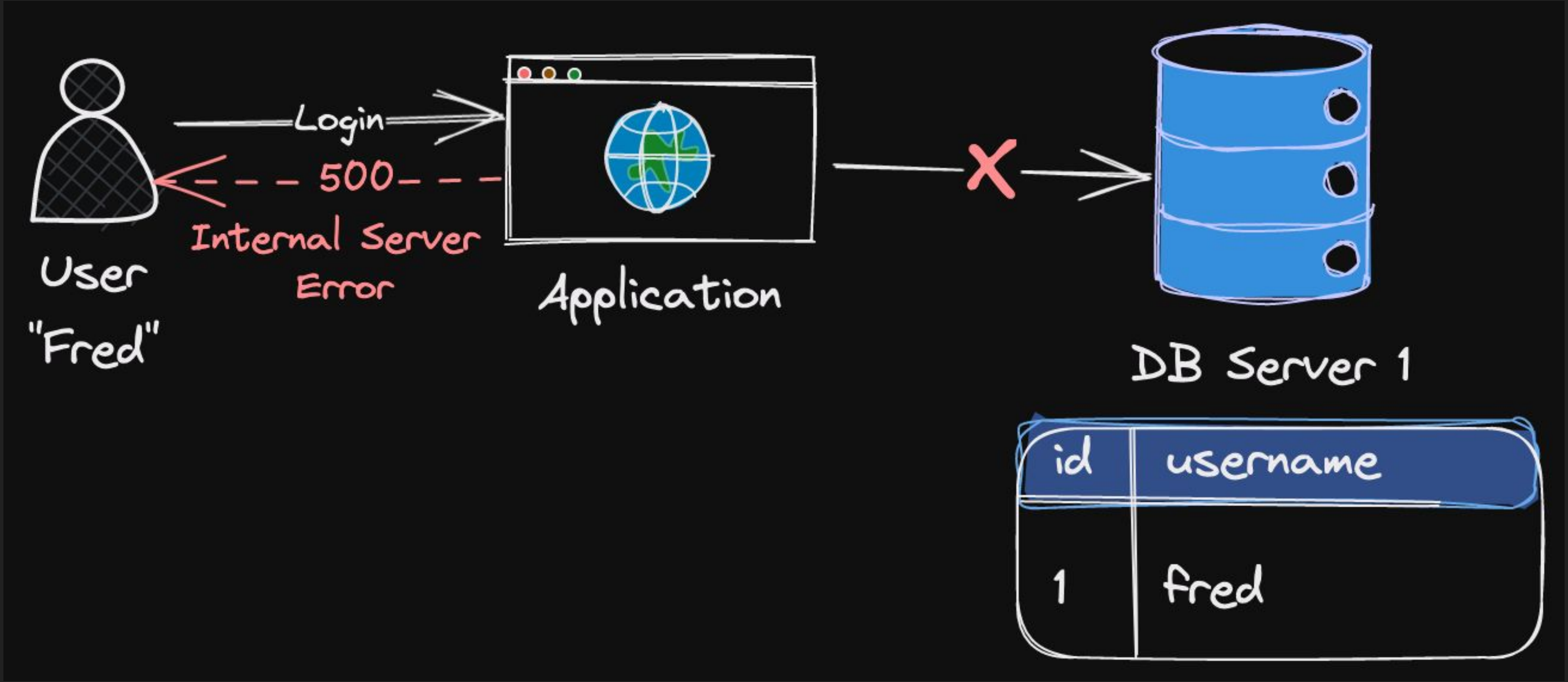
Database Replication

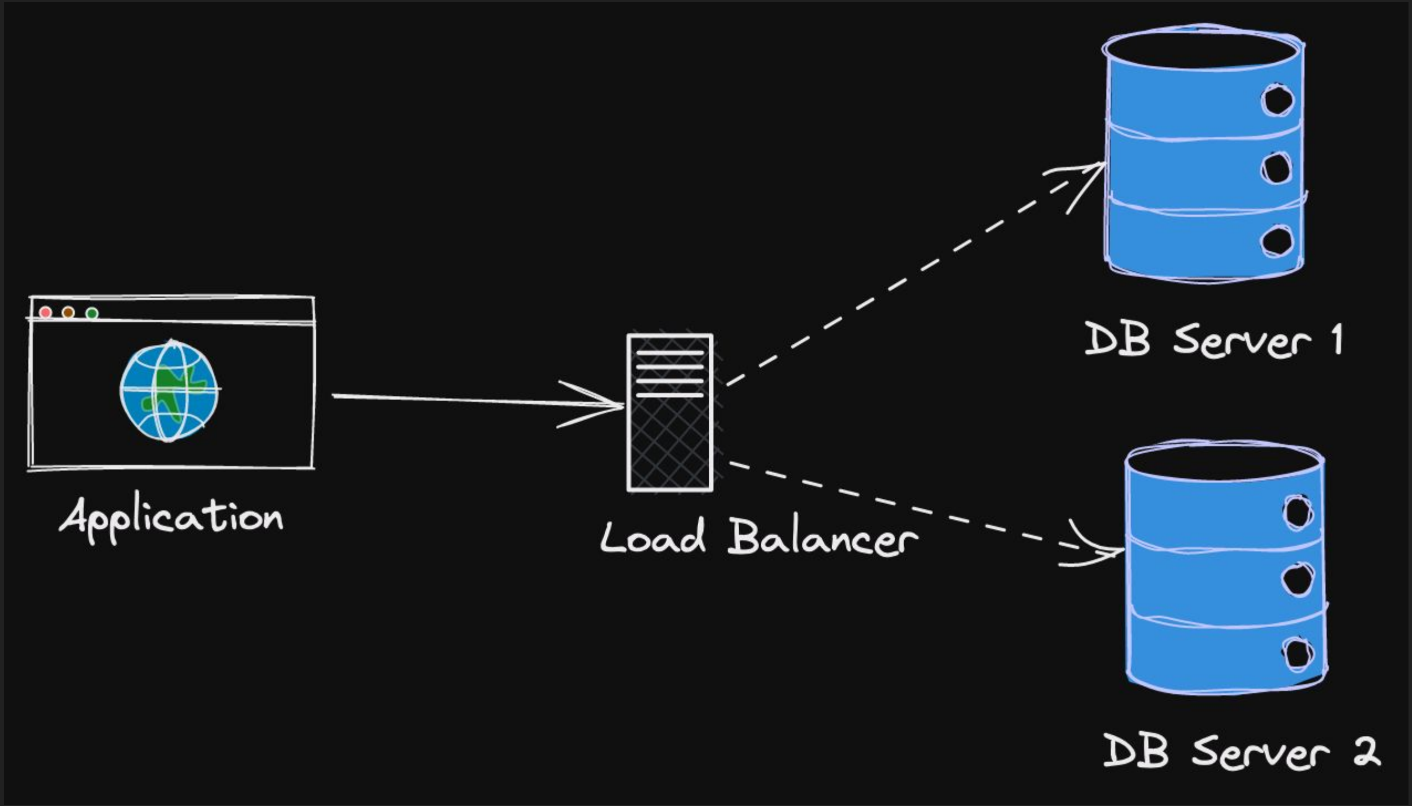
Single Database Server



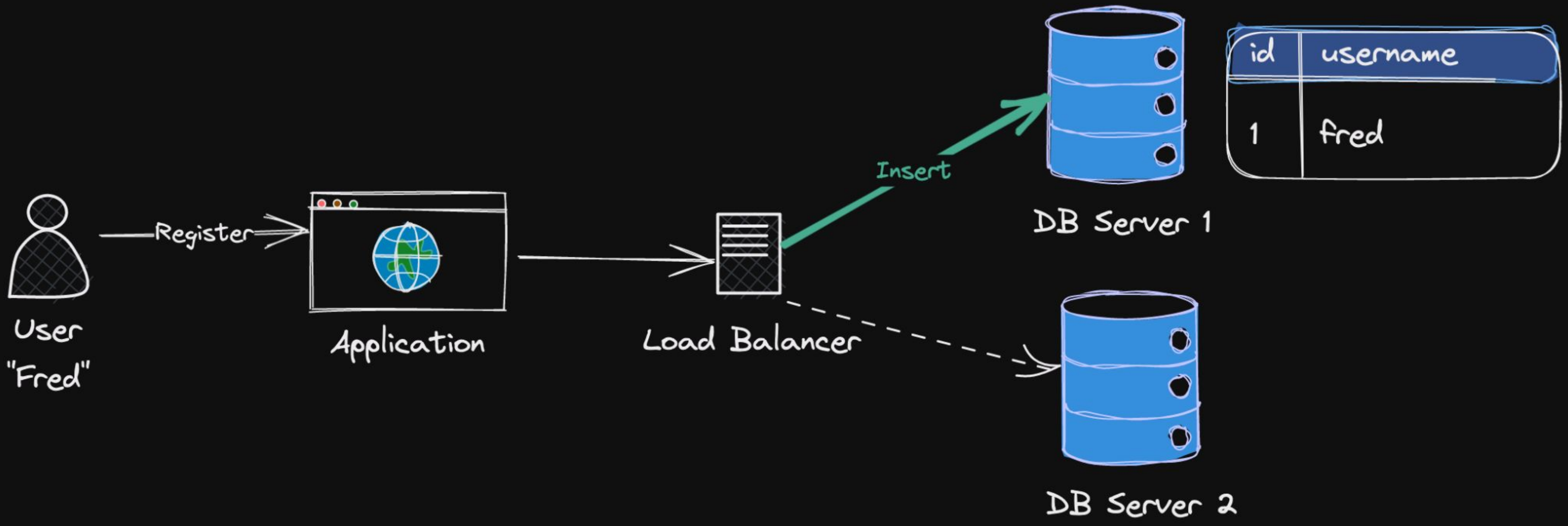
Single Database Server = Single Point of Failure



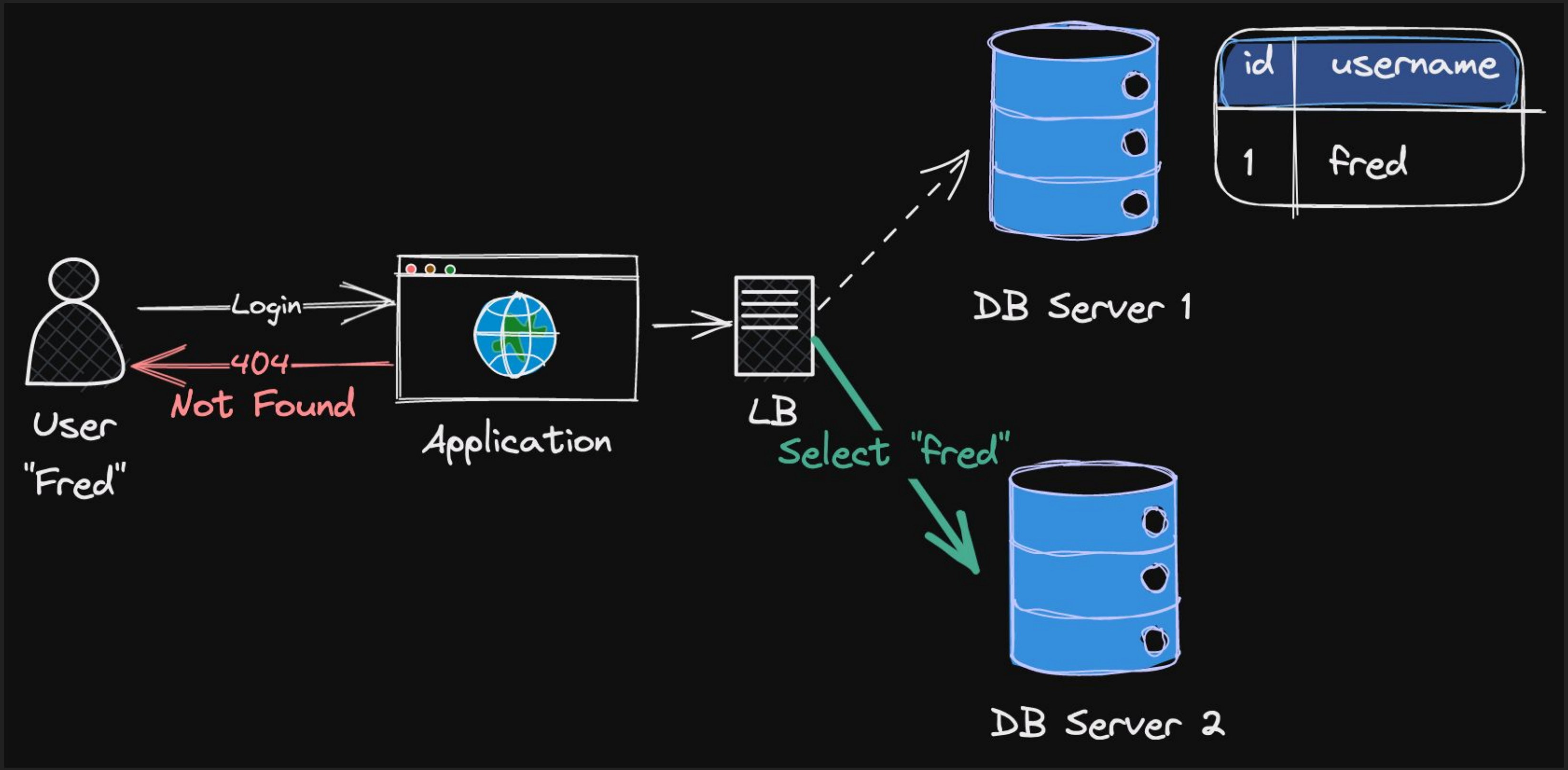
Multiple Database Instances



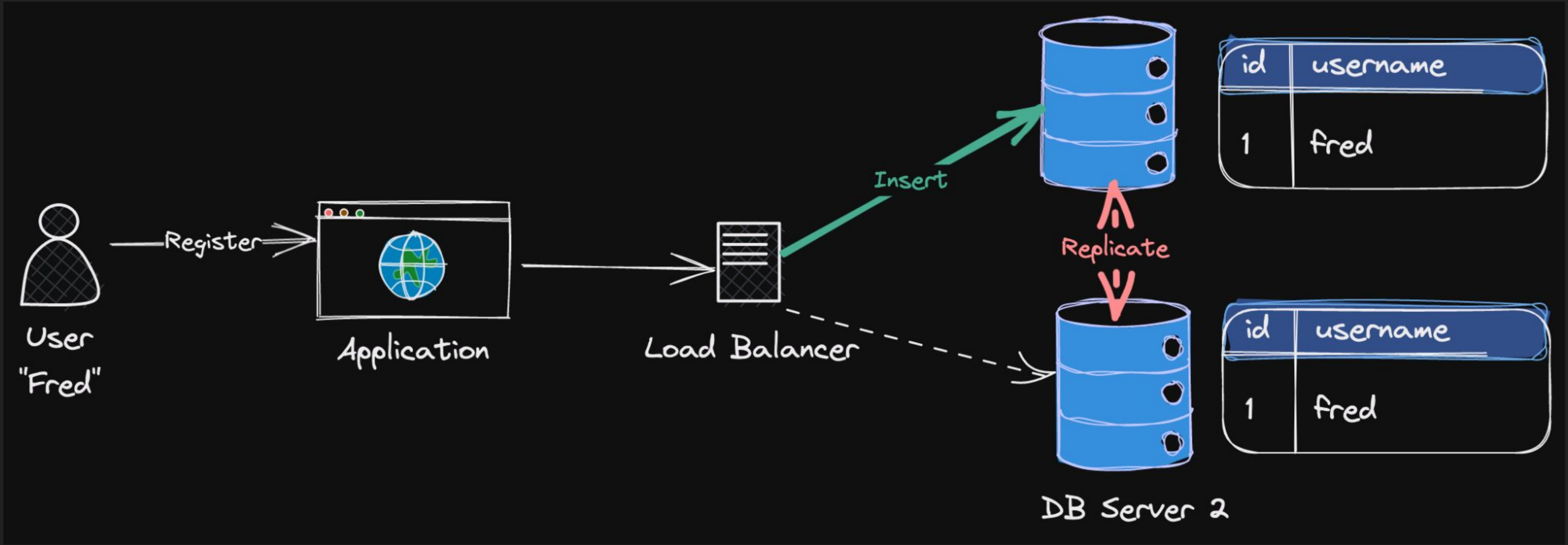
Multiple Database Instances



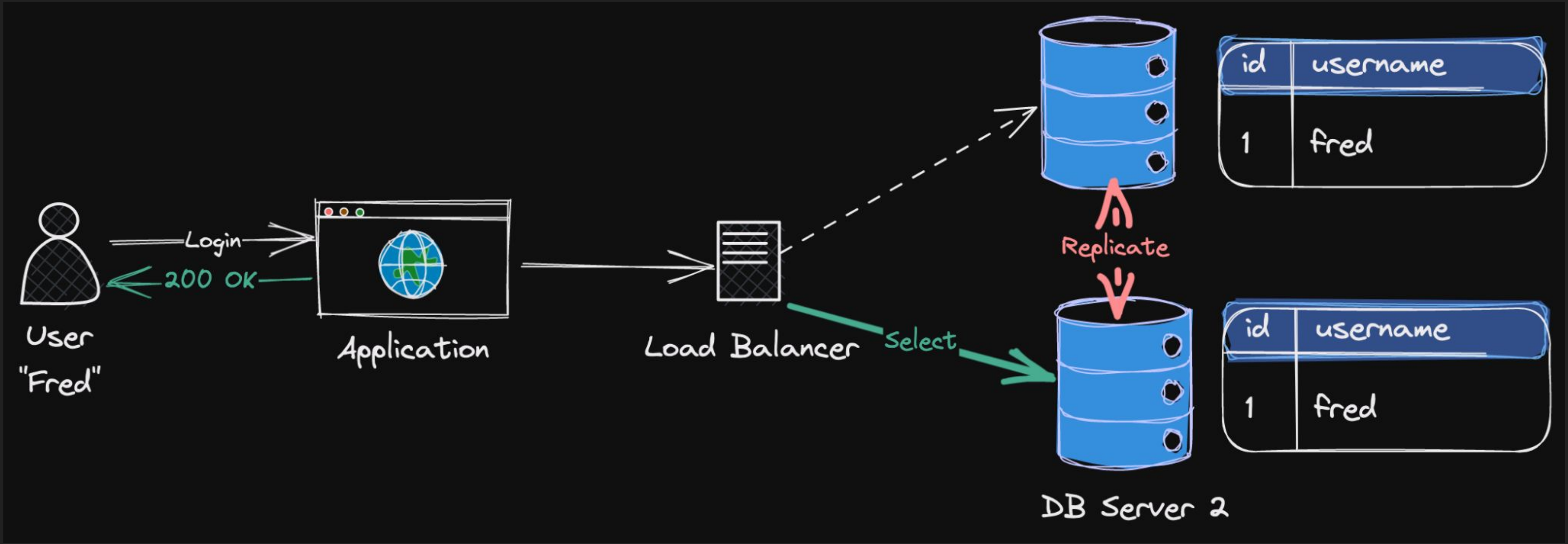
Multiple Database Instances



Database Replication



Database Replication



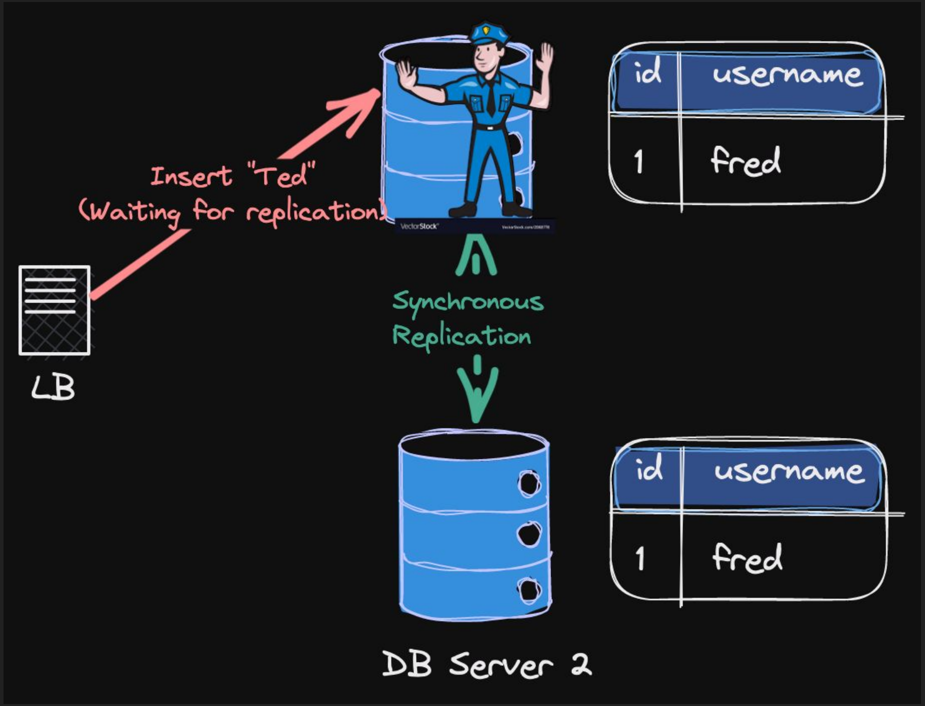
Database Replication

Copying data from a database on **one server** to a database on **another server**.

Agenda

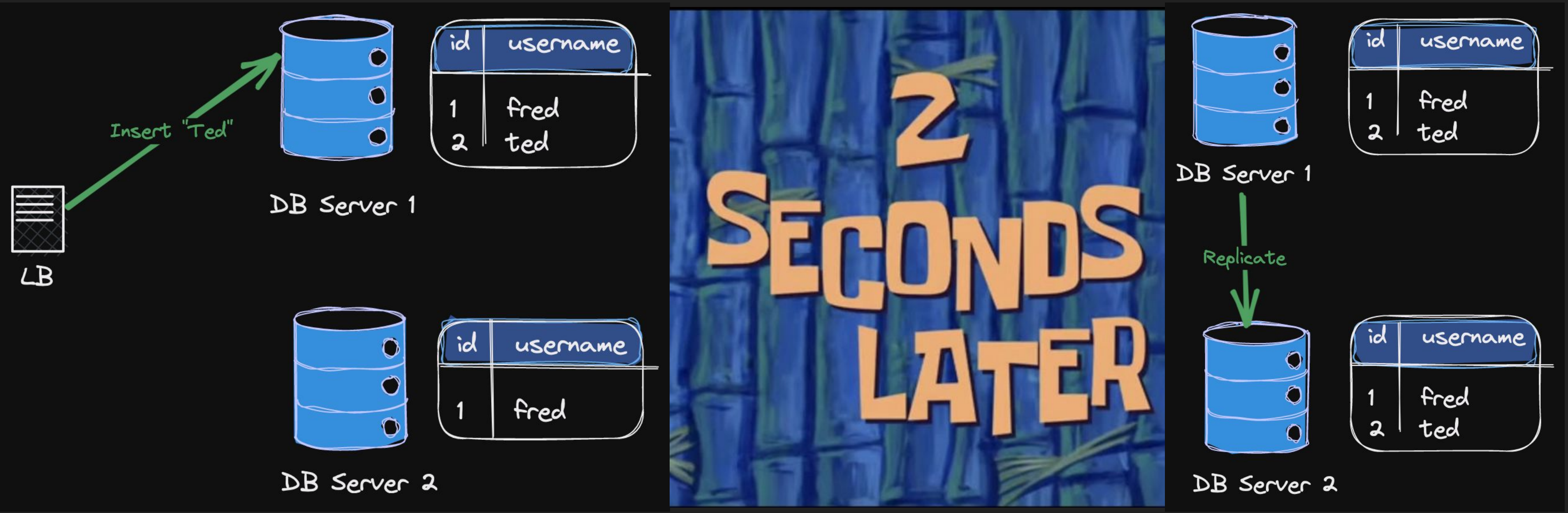
1. **Replication Modes**
    1.1. Synchronous
    1.2. Asynchronous
2. **The Replication Process**
3. **Replication Architecture**
    3.1. Primary-Secondary
    3.2. Multi-master
    3.3. Leaderless

Synchronous Replication

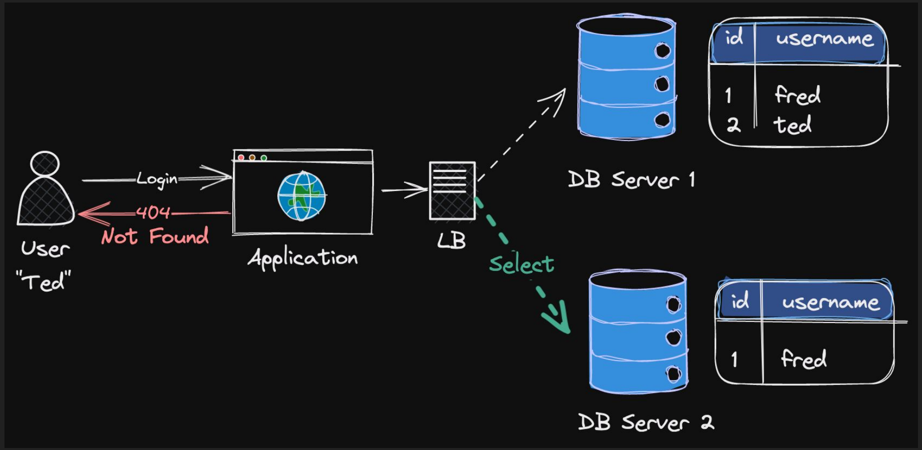


- Primary database writes the data locally
- **Waits for other notes** to confirm they've copied the data before returning success.
- **Consistent**: All nodes have the same data.
- **Unavailable**: Requests are put on hold while waiting for confirmation. Availability is reduced.

Asynchronous Replication

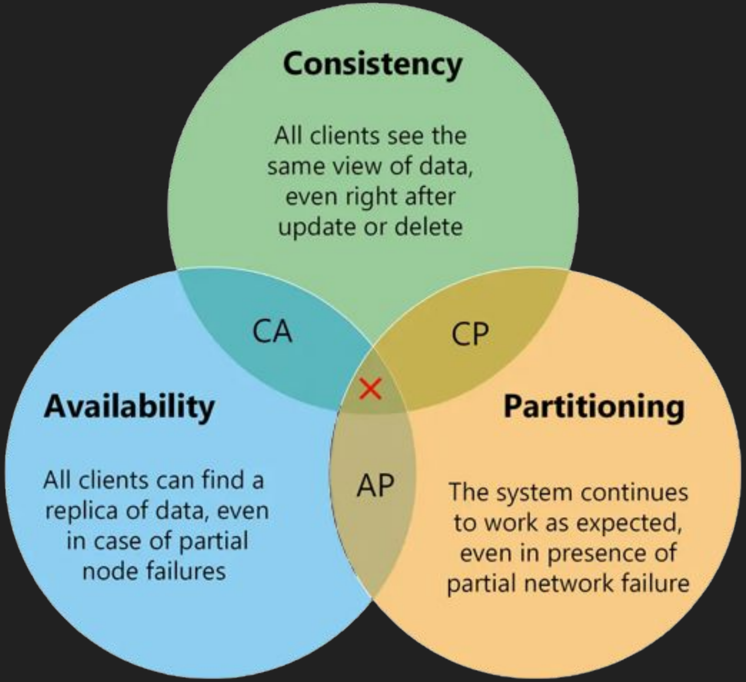


Asynchronous Replication



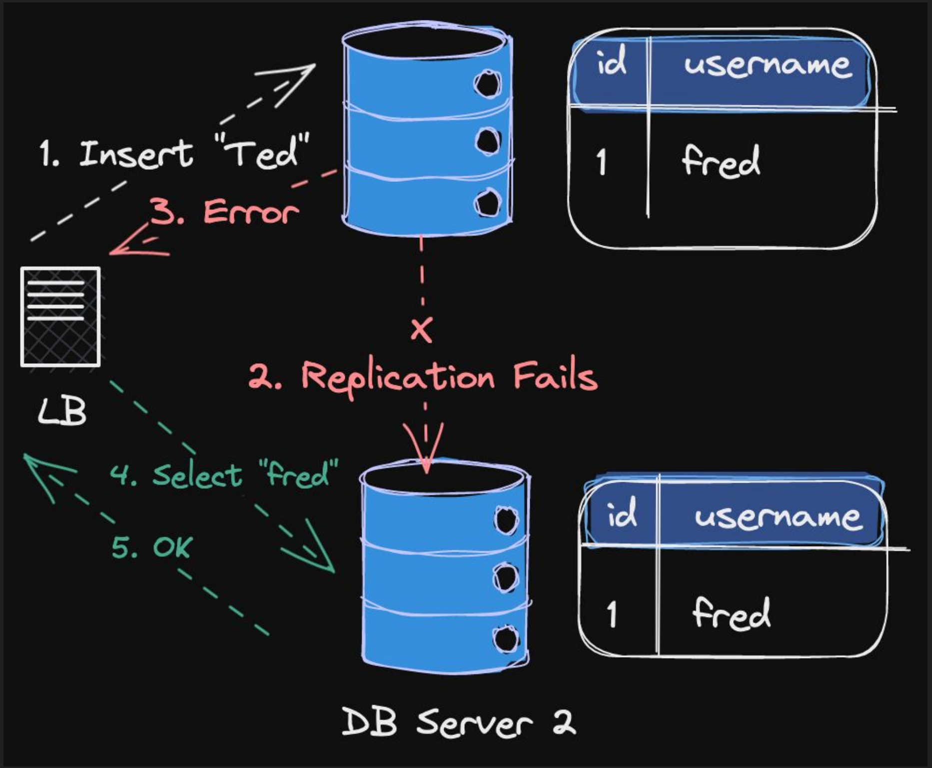
- Primary database writes the data locally and **sends success to client**.
- Replicas updated asynchronously, possibly after a set interval.
- **Available**: Servers available to process requests.
- **Inconsistent**: Database servers might not **immediately** have the same view of the data (Eventual Consistency)

CAP Theorem



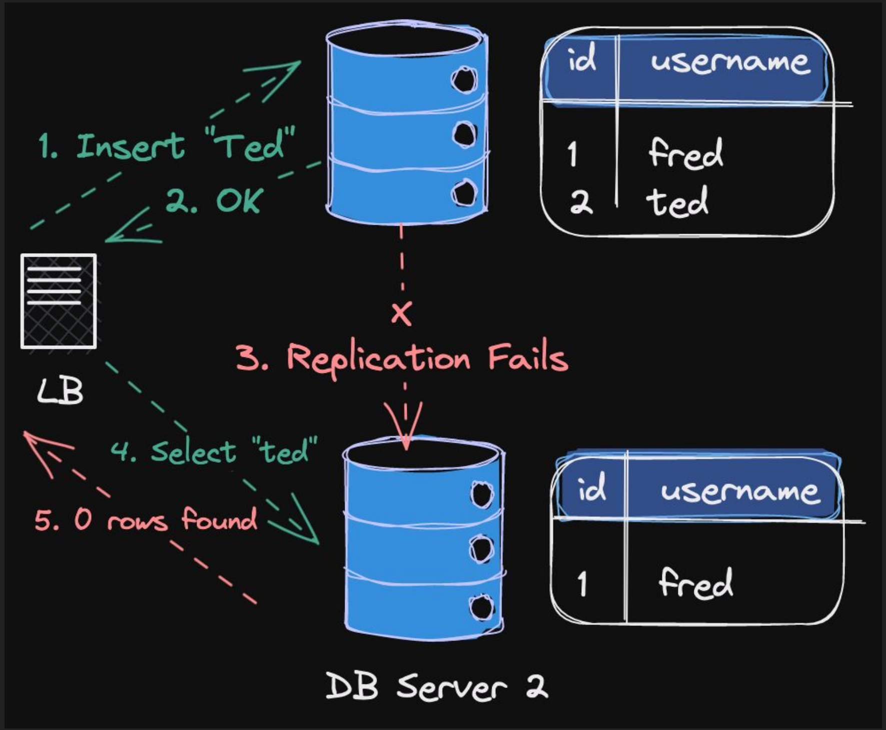
When the network in a distributed database breaks (Partitions), the servers can either be **Consistent OR Available, but not both.**

CAP Theorem: Synchronous Replication



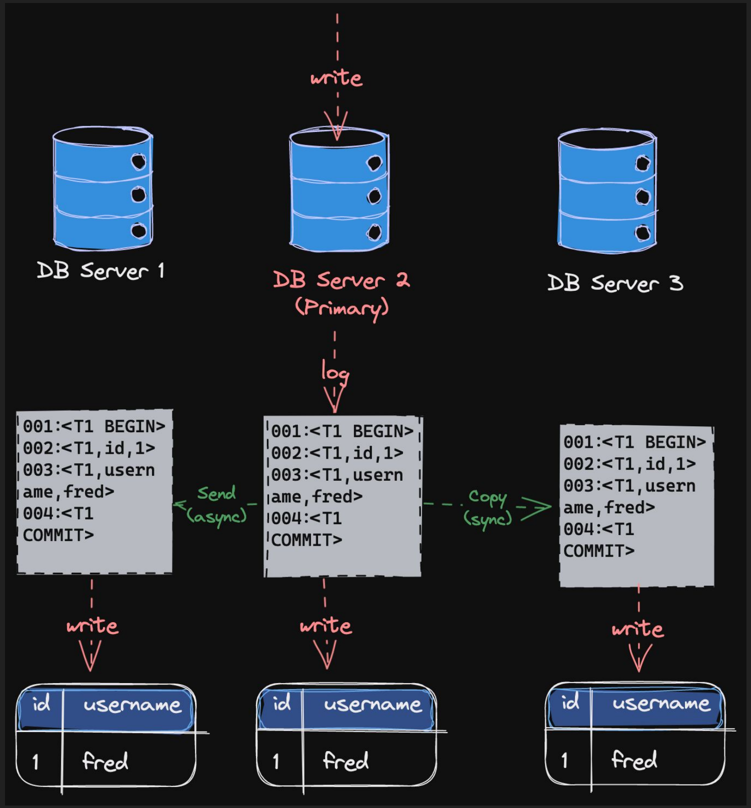
- With **synchronous replication**: all write requests fail. Cluster is not available.
  - Primary will not consider transaction a success until all replicas confirm copying the data.
  - Replicas can't confirm, because connection is broken.
  - After a timeout period, primary considers transaction a failure.
- Read requests return a consistent view of the data.

CAP Theorem: Asynchronous Replication



- With asynchronous replication, write requests succeed, but not replicated.
  - Cluster is available for write requests
  - Different servers have different views of data.

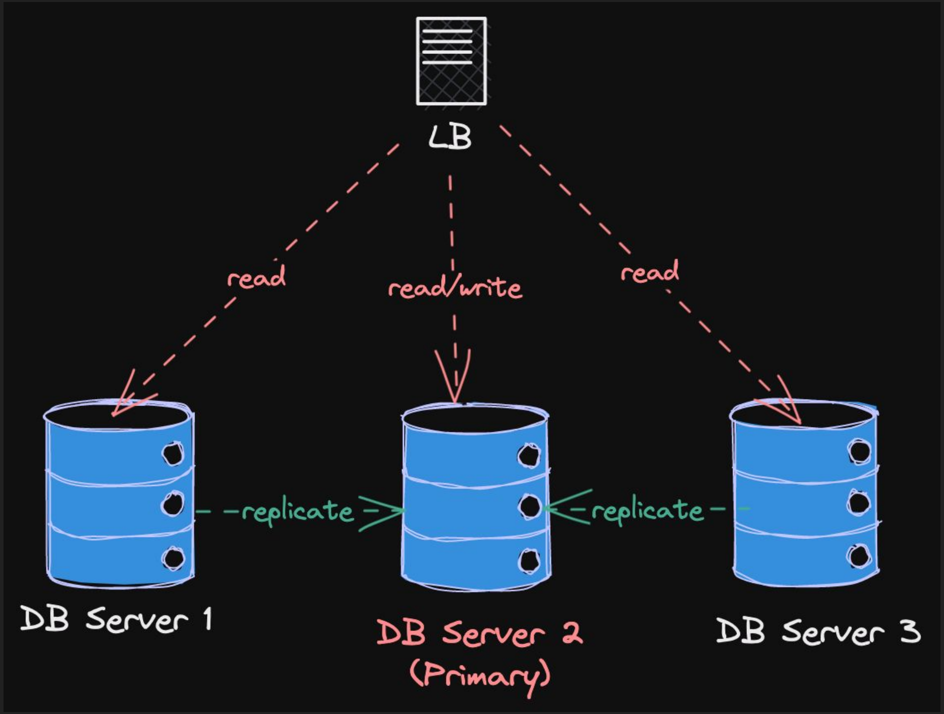
The Replication Process



- **Log Based Incremental Replication** is the most common way of replicating data.
- The primary node **writes changes to a log** file before applying them.
- The secondary node reads the changes from the primary's log file and applies them.
- Secondary maintains a pointer to the position where its read from a log.
- Changes are applied in the secondary in the same order as they were applied in the primary.
- Used by PostgreSQL, MySQL, MongoDB

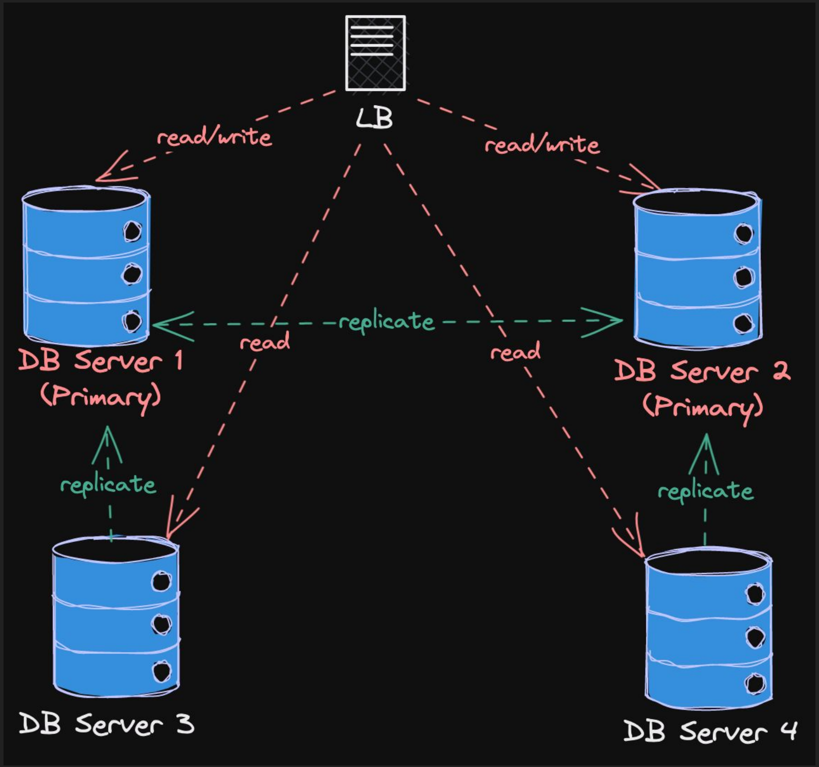
- For SQL databases, each DB has its own log file structure. Log file structure might change with versions. This can make migrations between versions and other SQL databases challenging.

Replication Strategies: Primary-Secondary



- One node is elected as the primary.
- Only the **primary handles write requests**.
- All nodes handle read requests.
- **Secondary nodes replicate** data from the primary node.
- Supported out-of-the-box by MySQL, PostgreSQL, MongoDB
- Secondary nodes reduces load on primary server.
- Suitable for read-intensive applications, not suitable for write-intensive applications e.g. a bank.

Replication Strategies: Mutli-Master



- Mutiple nodes selected as primary.
- Only the **primary handles write requests**.
- All nodes handle read requests.
- **Primary nodes send their writes to other primary nodes.**
- Secondary nodes replicate from one of the primary nodes.
- Can be added to PSQL, MariaDB, MongoDB using 3rd party tools.

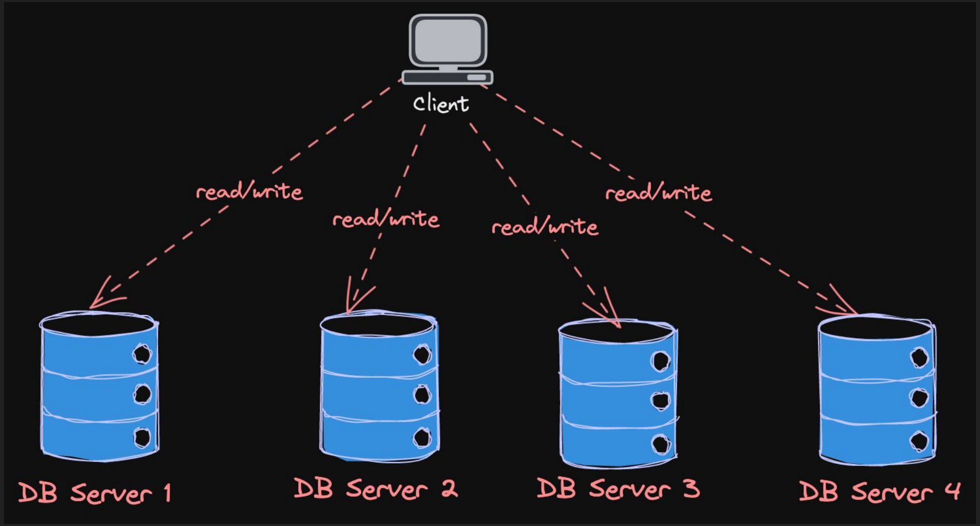
- Suitable for write-intensive applications.
- Can even be used for offline-first applications i.e. the database on the mobile device acts as a primary node.

- If network connection breaks between primary nodes, conflicts can occur.

Replication Strategies: Conflict Resolution Strategies

- **Routing:**
  - Load balancer routes all writes for a given record to the same primary node e.g. all writes for user “fred” go to DB Server 1.
  - Simple
  - Might result in uneven distribution of traffic between the two primary nodes (e.g. if one of the primary has more records that are updated more frequently.)
- **Last write wins**
  - A physical clock measures time in seconds.
  - It is difficult to synchronize physical clocks between two distributed nodes.
  - A logical clock measures events since a given point of time. Each insert or delete request is considered an event.
  - The last-write (in terms of the logical clock, rather than the physical clock) wins.

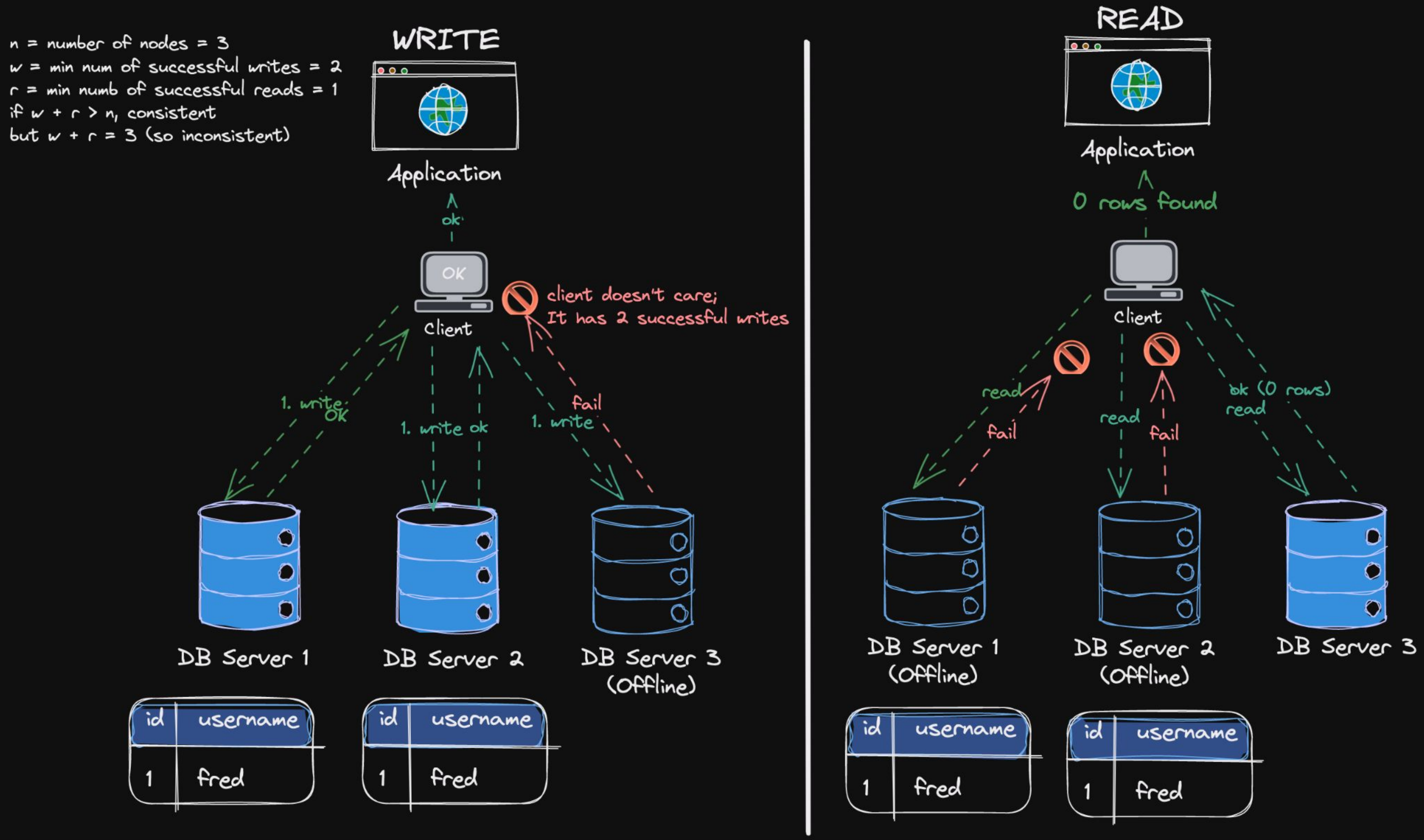
Replication Strategies: Leaderless

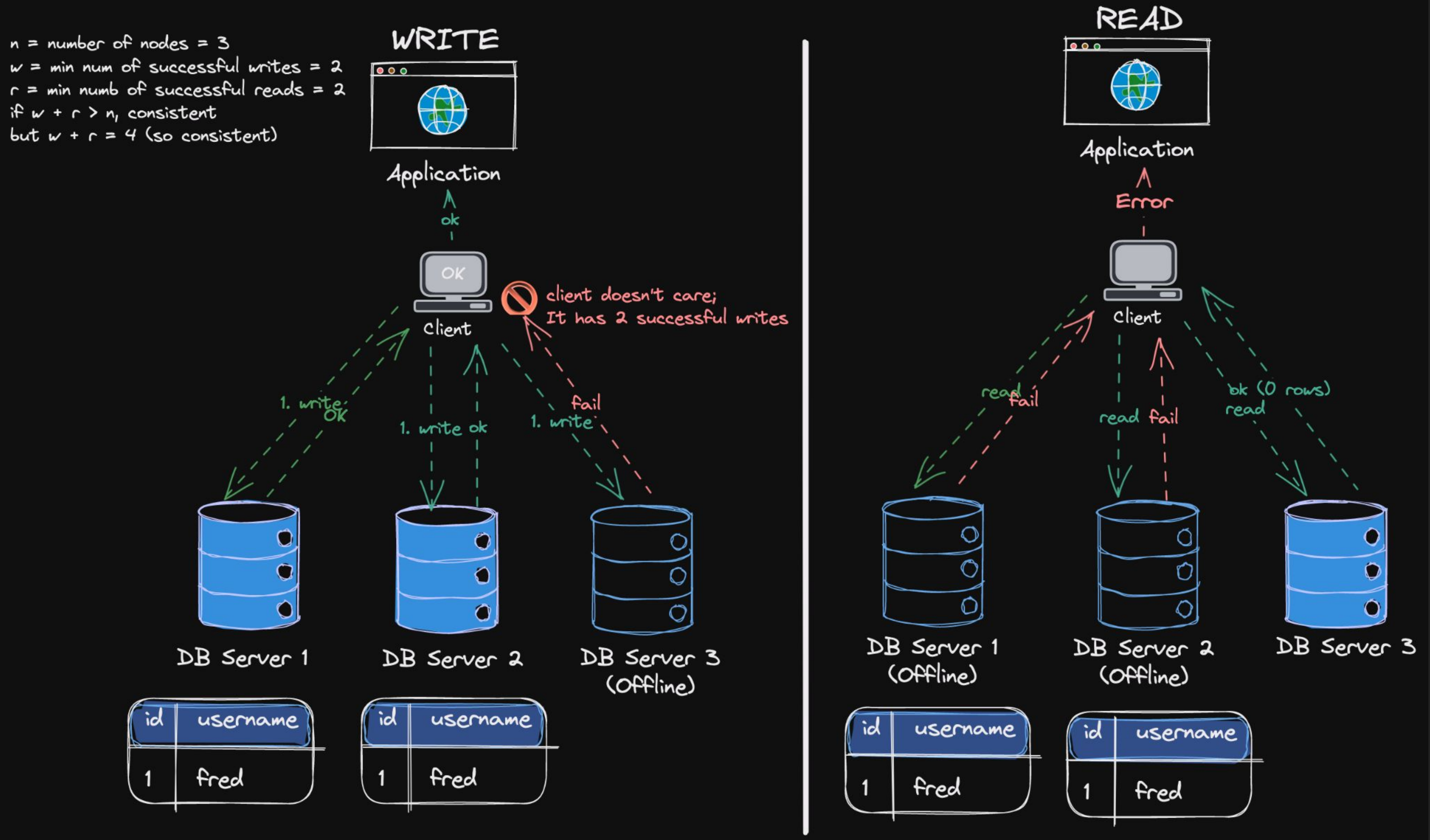


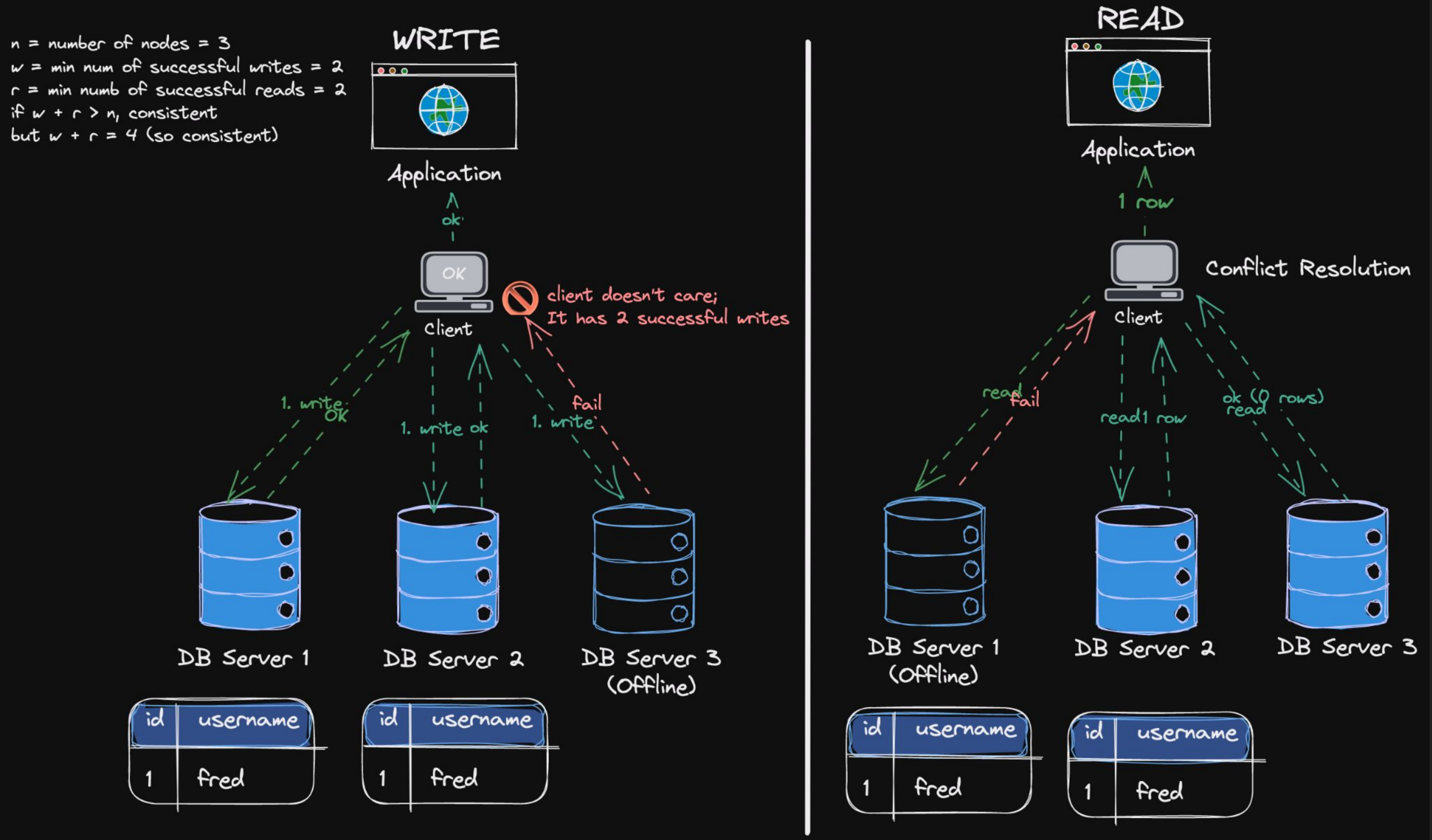
- Client sends read and write Requests are sent to **every single** database server.
- Database admin must configure:

w	Given a write request, How many nodes must the client wait to hear a success from?
r	Given a read request, How many nodes must the client wait to hear a success from?

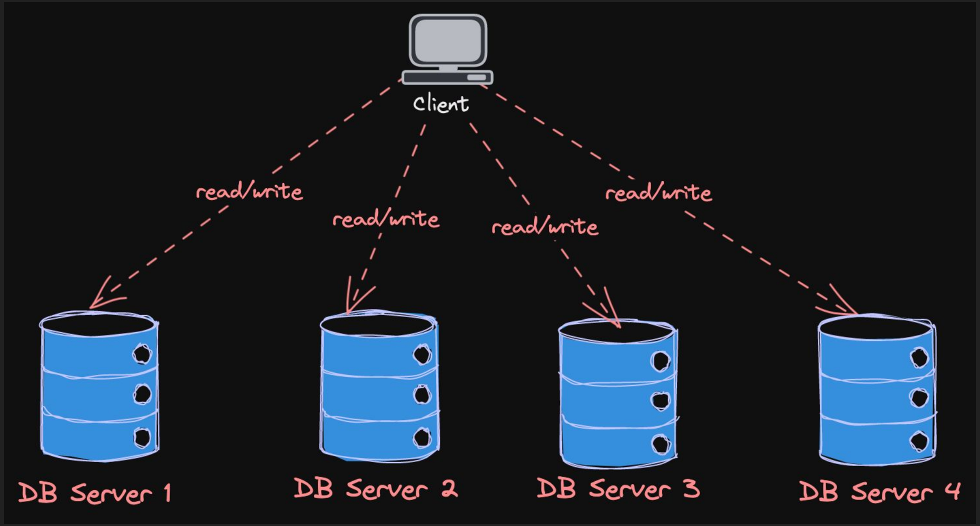
- If $w + r > n$ (n = number of nodes), then the **data will seem consistent**.
- This approach is called **Strong Quorums**.


n = number of nodes = 3
w = min num of successful writes = 2
r = min numb of successful reads = 1
if w + r > n, consistent
but w + r = 3 (so inconsistent)
WRITE
Application
ok
OK
Client
client doesn't care;
It has 2 successful writes
1. write
ok
1. write ok
fail
1. write
DB Server 1
DB Server 2
DB Server 3
(Offline)
id
username
1
fred
id
username
1
fred
READ
Application
0 rows found
Client
read
fail
read
fail
ok (0 rows)
read
DB Server 1
(Offline)
DB Server 2
(Offline)
DB Server 3
id
username
1
fred
id
username
1
fred


n = number of nodes = 3
w = min num of successful writes = 2
r = min numb of successful reads = 2
if w + r > n, consistent
but w + r = 4 (so consistent)
WRITE
Application
ok
OK
Client
client doesn't care;
It has 2 successful writes
1. write
OK
1. write ok
fail
1. write
DB Server 1
DB Server 2
DB Server 3
(Offline)
id
username
1
fred
id
username
1
fred
READ
Application
Error
Client
read
fail
read
fail
ok (0 rows)
read
DB Server 1
(Offline)
DB Server 2
(Offline)
DB Server 3
id
username
1
fred
id
username
1
fred


n = number of nodes = 3
w = min num of successful writes = 2
r = min numb of successful reads = 2
if w + r > n, consistent
but w + r = 4 (so consistent)
WRITE
Application
ok
OK
Client
client doesn't care;
It has 2 successful writes
1. write
ok
1. write ok
fail
1. write
DB Server 1
DB Server 2
DB Server 3
(Offline)
id
username
1
fred
id
username
1
fred
READ
Application
1 row
Conflict Resolution
Client
read
fail
read
1 row
ok (0 rows)
read
DB Server 1
(Offline)
DB Server 2
DB Server 3
id
username
1
fred
id
username
1
fred

Replication Strategies: Leaderless



- Used by **DynamoDB, Cassandra**.
- Conflicts resolved using last-write wins.

- **Configurable:** By changing the values of w and r, you can configure whether you want stronger consistency or higher availability.
- **Flexible**: Consistency and availability can be achieved even when a few nodes are offline.
- **Write Available**: All nodes are available for write requests.

- Minimum number of nodes configured for reads and write must be online.

References

- WAL Logging Explained (PostgreSQL)
- Log-based Incremental Replication (Other DBs)
- High Availability, Load Balancing, and Replication (PostgreSQL)
- Multi-master and master replication in MariaDB with JElastic
- Strong and Sloppy Quorums

Q & A

1. How is database replication done at a global scale?

- This is achieved using Cross Data Center Replication (XDCR)
- XDCR uses memory-to-memory data replication, so all writes are first saved in the memory and then put in a replication queue, which sends it over the network simultaneously through multiple threads.
- XDCR is typically done asynchronously.
- XDCR is bidirectional i.e. Cluster A replicates data from Cluster B and vice versa.
- Conflicts can be resolved using last-write-wins.
- XDCR improves disaster recovery capabilities, provides high availability and low latency to users
- Cloud providers offer solutions cross region replication e.g. AWS Cross Region Replication for RDS
- Reference

2. How are indices and schema changes replicated?

- This depends on the database and replication strategy.
- PostgreSQL, for example, supports logical and physical replication.

Logical Replication

- For PostgreSQL Logical Replication, schema changes and indexes are **NOT** replicated. The same schemas, tables must already exist in the secondary in order for replication to work.[1]
- Indexes must be recreated in the secondary node.
- Logical Replication is used in CDC (Change Data Capture)

Physical Replication

- For PostgreSQL, WAL's central concept is that changes to data files (where tables and indexes reside) must be written only after those changes have been logged[2]
- When the WAL file is replicated, the secondary receives table changes, data changes and indexes.
- The same changes are applied in the secondary.
- Further Reading

3. What happens when a WAL file is deleted before it can be replicated?

- As the DB is operating, blocks of data are first written serially and synchronously as WAL files, then some time later, usually a very short time later, written to the DB data files.
- At some point, depending on your configuration, the primary will remove or recycle the WAL files whose data has been committed to the DB.
- This is necessary to keep the primary's disk from filling up. However, these WAL files are also what streaming replicas read when they are replicating data from the primary.
- If the replica is able to keep up with the primary, removing these WAL files generally isn't an issue.
- If the replica falls behind or is disconnected from the primary for an extended period of time, the primary may have already removed or recycled the WAL file(s) that a replica needs (but see Streaming Replication Slots below). A replica can fall behind on a primary with a high write rate. How far the replica falls behind will depend on network bandwidth from the primary, as well as storage performance on the replica.
- To account for this possibility, we recommend keeping secondary copies of the WAL files in another location using a WAL archiving mechanism.
- WAL archiving and backups are typically used together since this then provides point-in-time recovery (PITR) where you can restore a backup to any specific point in time as long as you have the full WAL stream available between all backups.
- Reference

4. Using Strong Quorums, what if a record is deleted in 2 nodes but deletion fails in 1 node?

Context

- Suppose there are 3 nodes (n=3), we set min successful writes, w = 3 and min successful reads, r = 2. This means consistency w + r > n which means data should seem consistent.
- Let's say we get a write request. Record is written to all 3 nodes.
- Then we get a delete request, Record is deleted in 2 nodes, fails in the third.
- Then we get a read request. For first 2 nodes, the record doesn't exist and for the third node, the record is present. How is this conflict resolved?

Answer

- In Cassandra (which uses Strong quorums), when a record is deleted, a tombstone is left behind.
- In this scenario, the first 2 nodes return a tombstone; the third node says the record exists.
- Since the tombstone has a greater logical clock than the existent record, the database is able to determine that the record is actually deleted.
- Hints are used to propagate the deletion to nodes that didn't receive the update.
- Eventually tombstones are purged so that they don't take up too much space.
- Reference