Data storage in distributed systems - part I

Piotr Jaczewski
RTB House

What will this lecture be about?

In this part of the course we will consider the topic of data storage technologies for data intensive distributed systems.

We will cover the topics of:

- The aspects of running a traditional RDBMS in a distributed system.
- The common problems regarding concurrency and transactions using RDBMS.

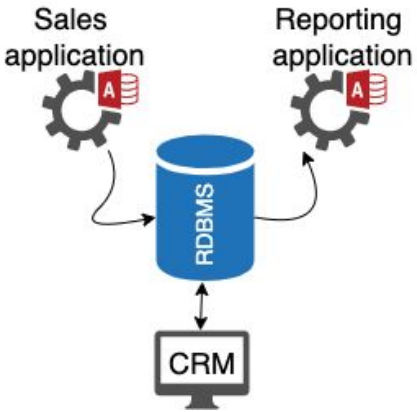
RDBMS

Why still use RDBMS?

- Well established flexible data model.
- Stable and battle-proven.
- Powerful Structured Query Language.
- Full support for ACID transactions.
- Auxiliary utilities which try to fix common problems like impedance mismatch (different representation of data between memory and database).
- Well documented.

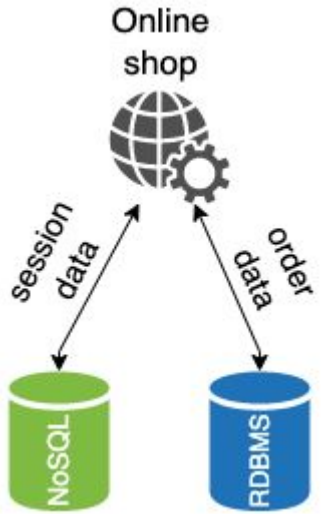
RDBMS as an “Integration Database”

- The SQL databases are frequently serving as integration layer between various applications.
- Consistent means of communication between distinct teams.
- Complex data structures that may cripple the development flexibility and performance of applications.
- Application specific data storages?



Polyglot Persistence

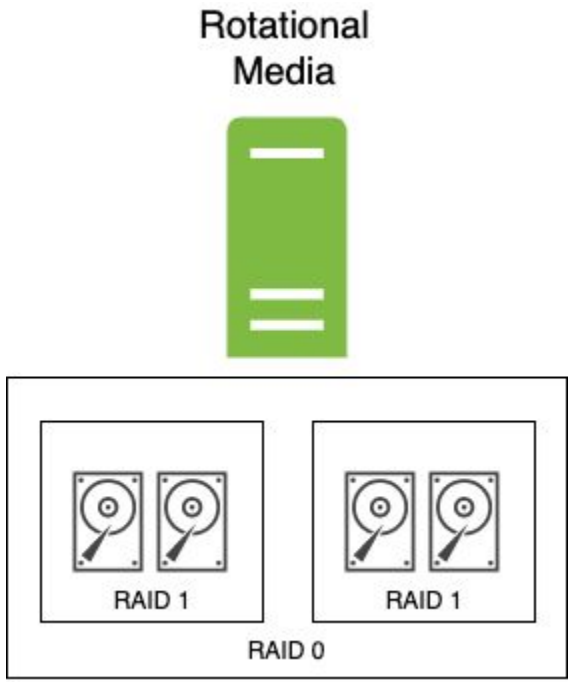
- Even in RDBMS realm the requirements of OLAP and OLTP systems are different.
- Transactional and analytical data are forced into the same schema.
- Some data does not need to conform to the same rigorous backup/recovery strategies as other systems (ie. user session data).
- Picking the right storage for each individual data requirement may be more productive than trying to fit everything into a single database.
- The hybrid approach to data storage - “polyglot persistence”.



RDBMS vs NoSQL

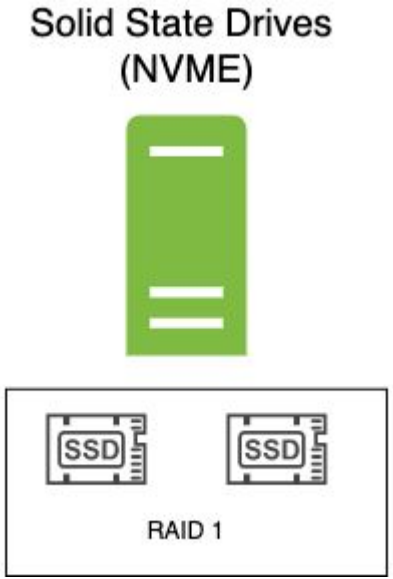
Before focusing on NoSQL in the next lecture we will discuss the practical aspect of working with relational databases in a distributed environment.

Hardware Architecture for RDBMS



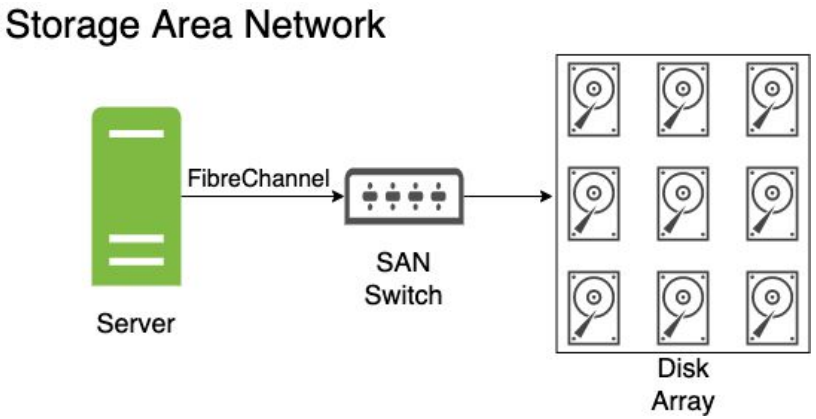
- With rotational media in DAS (Direct Access Storage) configuration the latency of the storage device is the performance bottleneck in RDBMS applications. To mitigate use RAID 10 (1 + 0) configuration.
- In this configuration the striped data is additionally mirrored for better fault tolerance.
- RAID capabilities can be either provided by system software or dedicated hardware controllers.

Hardware Architecture for RDBMS



- With solid state media in DAS (Direct Access Storage) configuration, the devices connected directly to PCI Express bus are no longer the performance bottleneck.
- In this scenario the usual configuration is to use RAID 1 for drive redundancy.

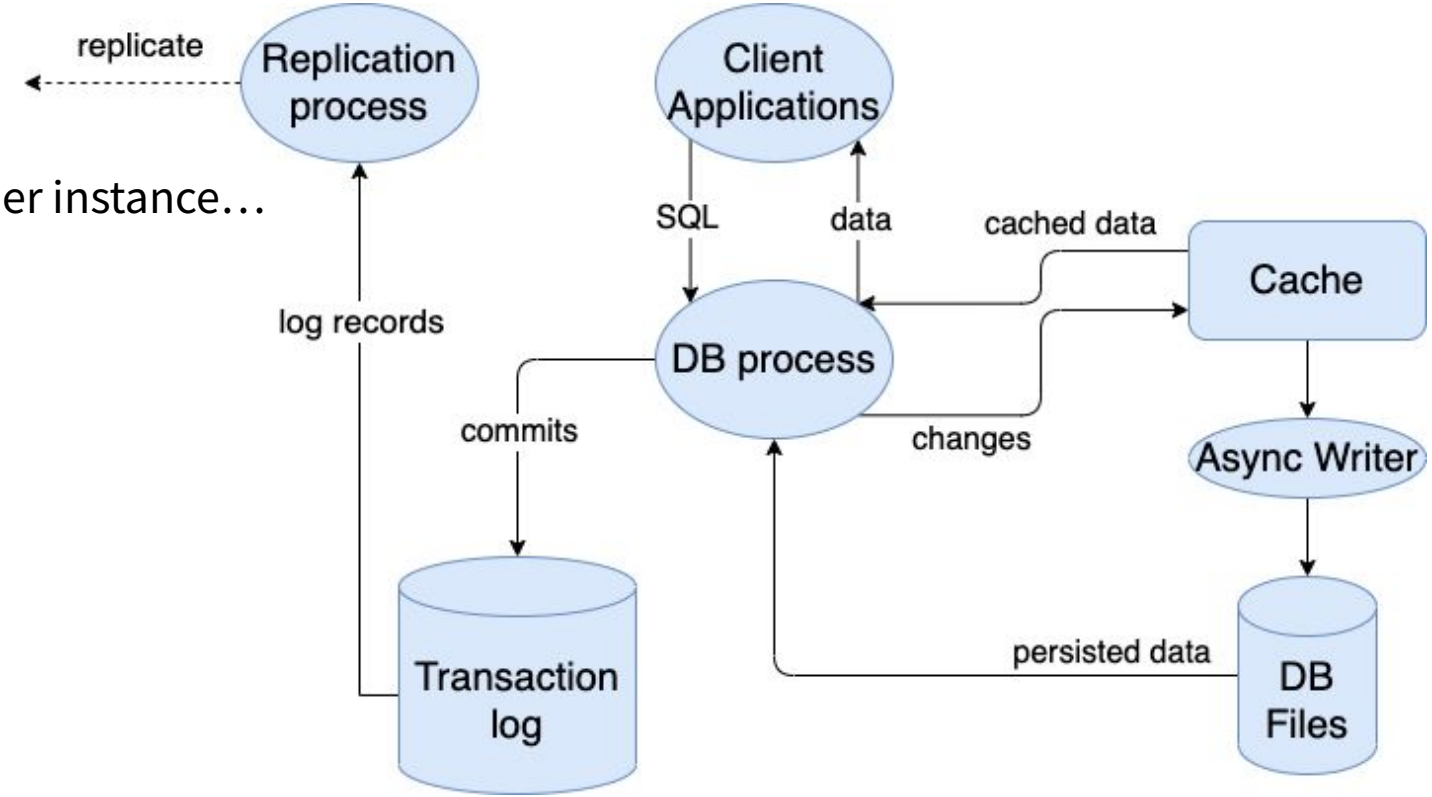
Hardware Architecture for RDBMS



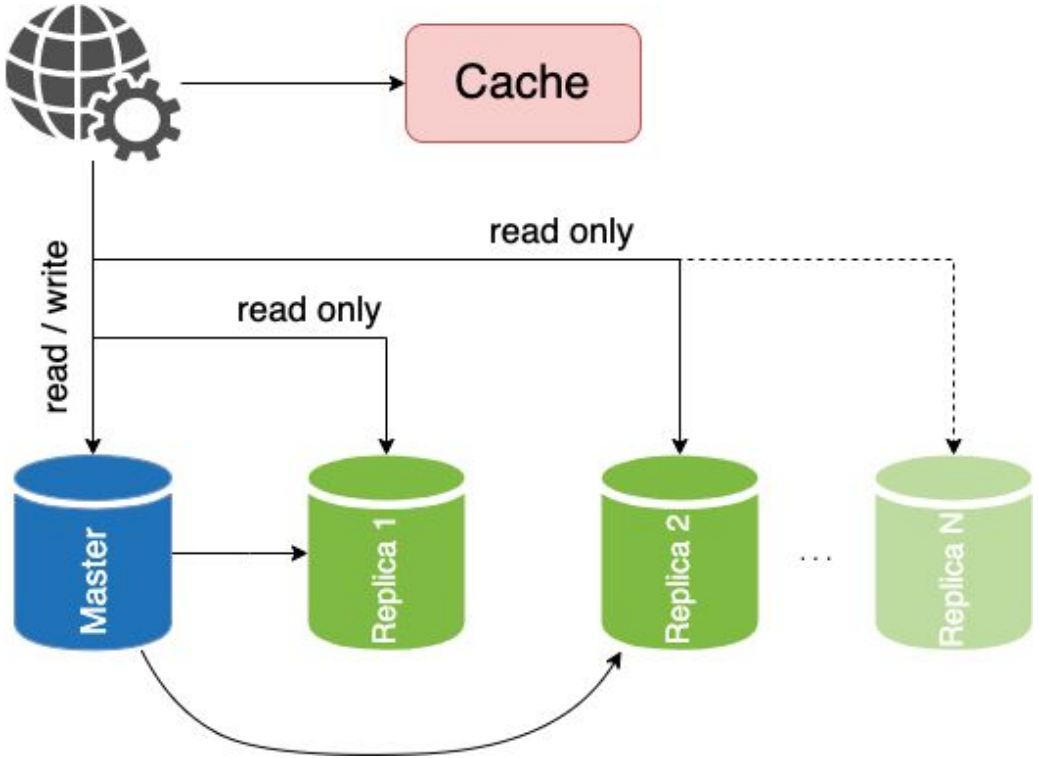
- The data storage for database can be also accessed via SAN
- Not as efficient as directly accessed storage.
- Many vendors offer custom proprietary solutions.
- Offers more flexibility regarding storage management.

Typical RDBMS Process

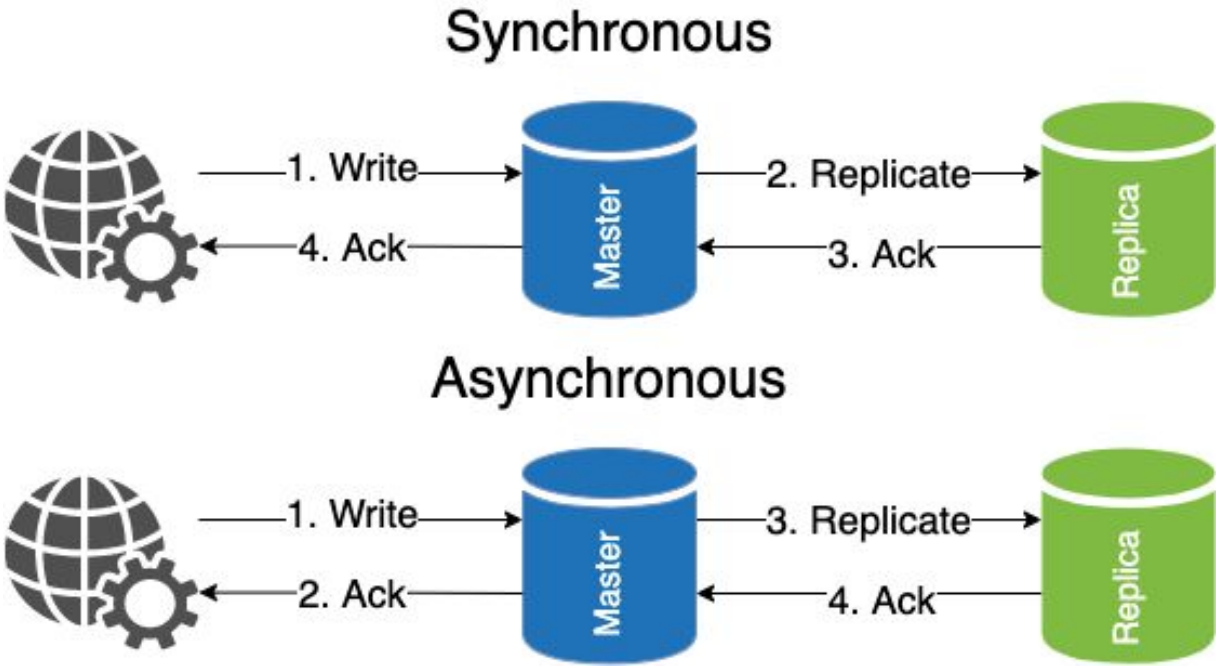
To another instance…



Typical Architecture Involving RDBMS



RDBMS Replication



RDBMS Replication

An (incomplete) selection of possible methods:

- **Log Shipping** - copying completed WAL segments, better resource utilisation, higher risk of reduced durability (lost changes).
- **Log Streaming** - sending individual WAL entries over an open connection, higher resource utilisation, lesser risk of reduced durability.
- **SQL replication** - possible inconsistent results between replica nodes (ie NOW() function yields different results).
- **Logical replication -** changes are published in a defined message format and are applied by subscriber nodes.
- **File System (Block Device) Replication** - all changes to a file system are mirrored to a file system residing on another computer.

Replication Burden

Replication adds additional burden to primary node with addition of follower nodes.

Cascading replication - follower nodes cascade changes to other followers.

Custom solution - when replicating between DC's:

- Allow only aggregates in database - denormalization with JSON columns.
- Force applications that mutate data to post aggregated changes on queue system.
- Feed "followers" on target DC's from queues (Kafka).
- Run checkers to fix consistency issues .
- The performance of replication lag consumption is scalable (more change consumers).

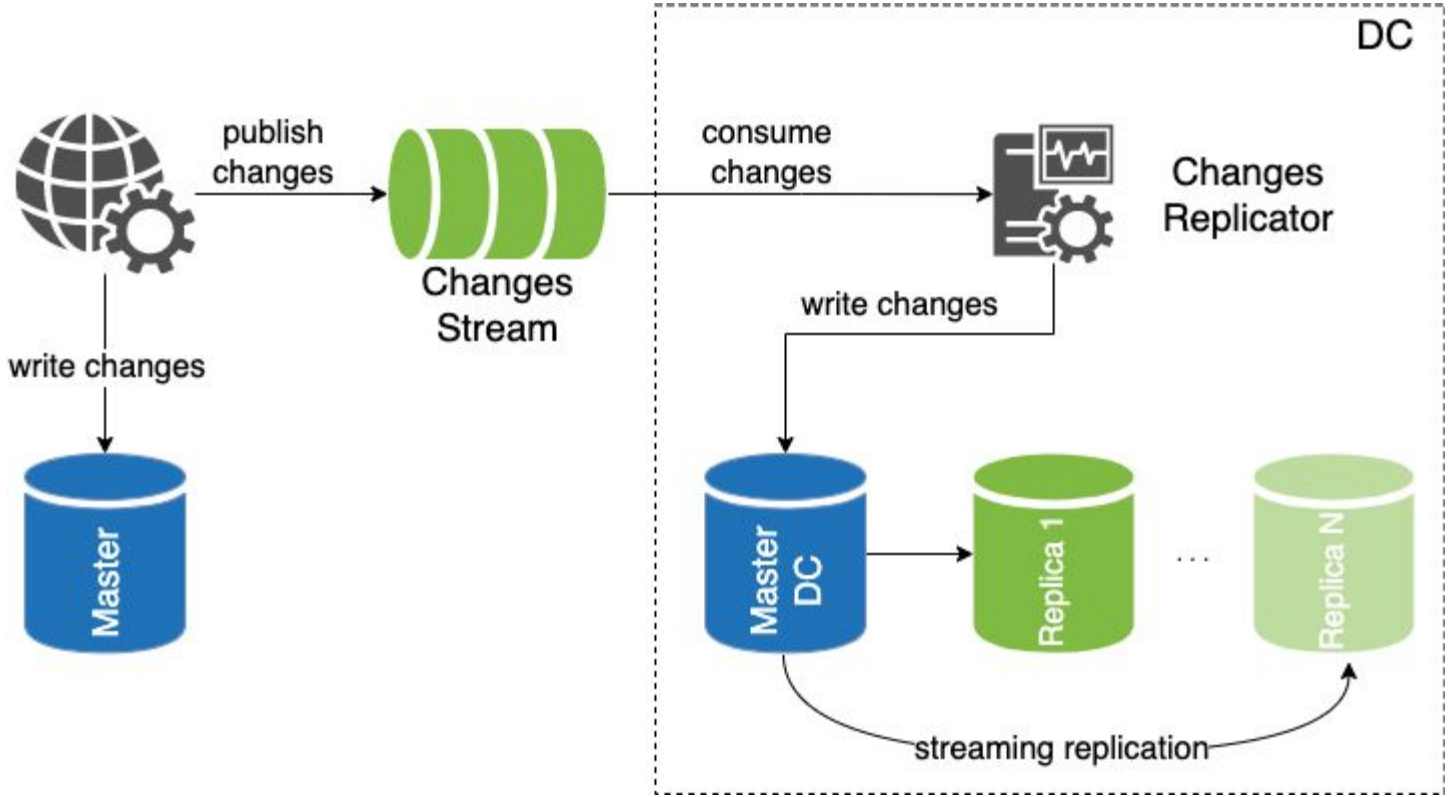
Custom Cross-DC Replication - Data Model

Table: rtb.jobs_json		
id PK (bigint)	**data (jsonb)**	**changed_at (timestamp)**

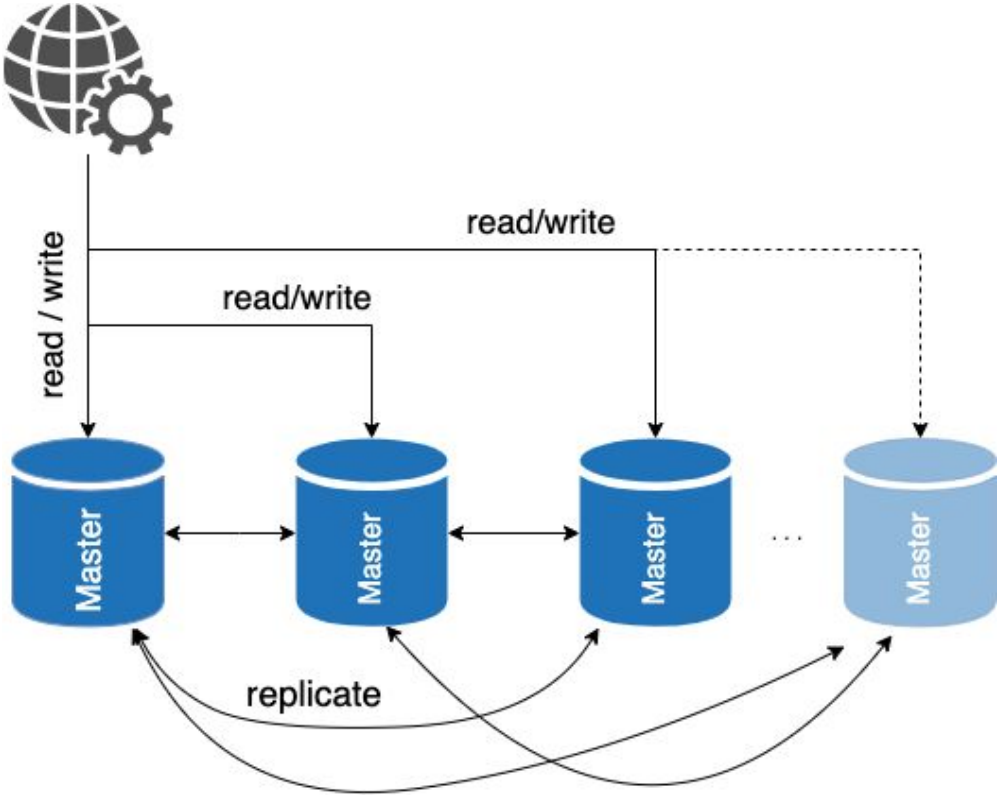
1	<pre>{"id": 1, "author": "core",
	"status": {
	"CORE": "DONE"
	} …</pre>
		2022-01-09 08:25:00.516576

2	<pre>{"id": 2, "author": "core",
	"status": {
	"CORE": "IN_PROGRESS"
	} …</pre>
		2022-01-10 11:32:28.717892

Custom Cross-DC Replication - Architecture



Multi-Master Replication



- Adds complexity.
- Unsupported natively by RDBMSes.
- Inevitable conflicts with non-trivial methods of resolution.

Load Balancing and Connection Pooling

SELECT queries on RDBMS may be distributed among several replicas for load balancing reasons, whereas write queries are usually distributed to master node.

Establishing connections to database can be expensive so it is a good idea to set a pool of connections and reuse them.

Both load balancing and connection pooling can be achieved in two ways:

- Internally in an instance of the application.
- By a centralized component.

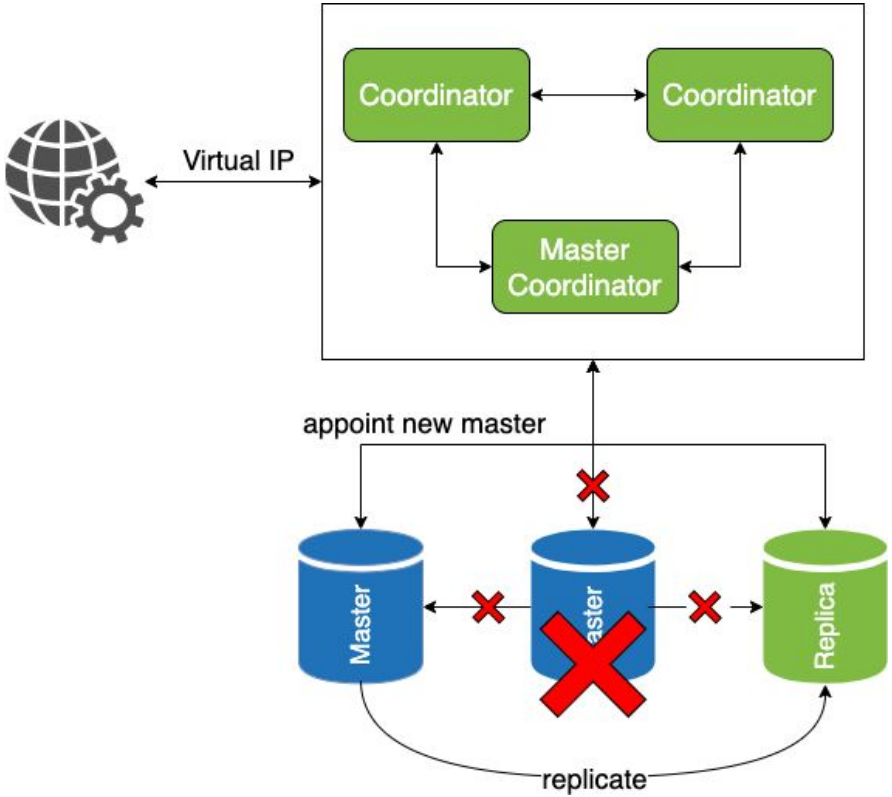
RDBMS Failover

When the primary database fails and needs to be replaced by on of the replica nodes.

- **Manual** - the new master has to be promoted manually by system's operator. This will mean usually partial unavailability for a RDBMS.

- **Automatic** - the database cluster manager will automatically promote new master from the set of available replicas.

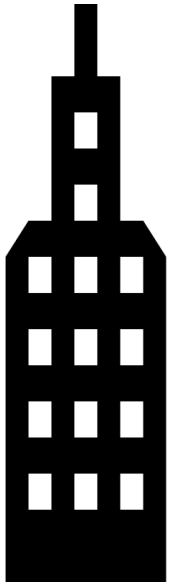
RDBMS Failover



RDBMS Vertical Scaling

How to scale the RDBMS vertically with losing as little availability as possible?

- CPU, RAM - these are hot swappable elements in high-end server equipment.
- Increasing disk capacity - in some SAN solutions this can be done with zero downtime by adding/replacing disks.
- On commodity hardware usually requires short downtime or partial unavailability (read only mode).

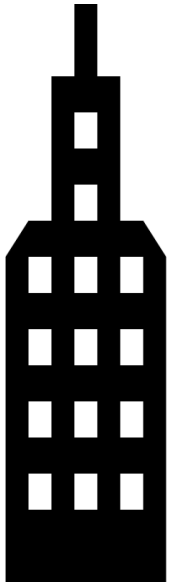


RDBMS Vertical Scaling

In case of commodity servers the exemplary strategies to scale capacity can be used:

By using Logical Volumes:

1. Add new disk to the system and expand the current logical volume and resize the filesystem online.
2. Replace the existing logical volume:
   - Add new disk and create new logical volume.
   - Synchronize the data between the old and new logical volumes.
   - Disable database for a short moment and replace the volumes.
   - Re-enable the database.

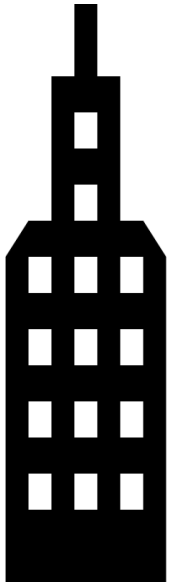


RDBMS Vertical Scaling

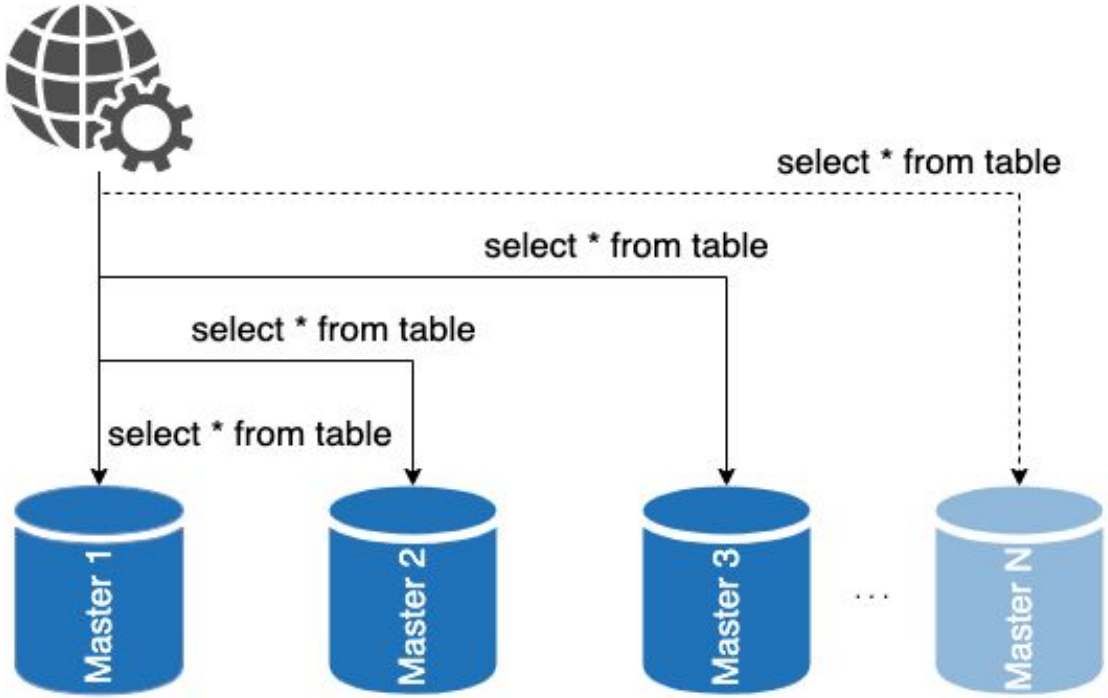
In case of commodity servers the exemplary strategies to scale capacity can be used:

By adding a new replica:

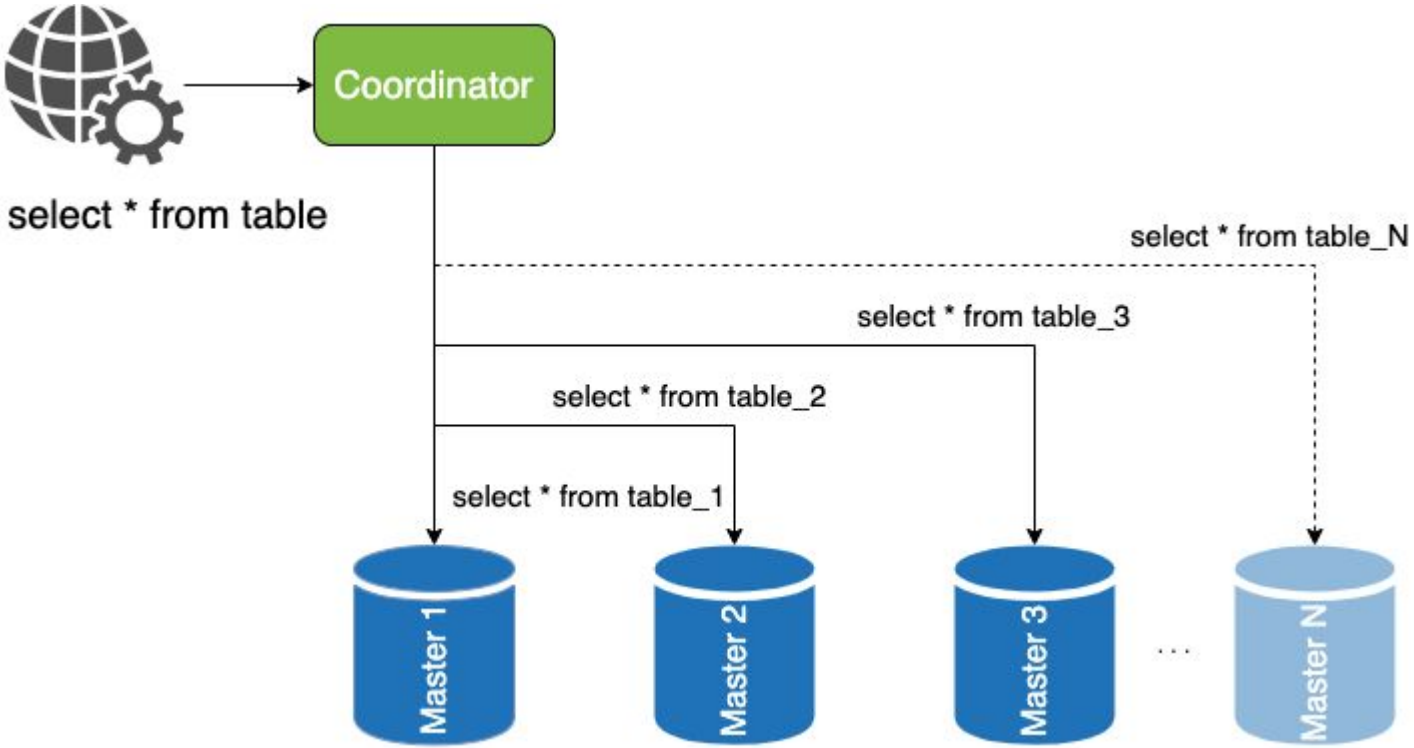
- Add new machine with desired new capacity (and other specification).
- Configure it as a replica and let it synchronize with the master.
- Wait for the replica to fully synchronize and promote it to leadership.



RDBMS Horizontal Scaling (Sharding)



RDBMS Horizontal Scaling (Sharding)



RDBMS Horizontal Scaling (Sharding)

Range based - each shard allocates rows with keys in a specified range of sharding key values:

- Range read queries can be satisfied from a single shard.
- Writes may be bottlenecked due to operations on one shard.
- Tendention to introduce hot spots.

Hash based - rows are distributed according to some hashing function applied to the sharding key:

- Range read queries involve all shards.
- Writes are not limited by a single shard node.
- Even distribution of records.

RDBMS Horizontal Scaling (Sharding)

- It’s hard to perform cross shard joins. The usual way of mitigating this issue is to denormalize the data so that queries can be performed on a single database.

- The addition of the new shard (ie. due to the depletion of storage capacity) is complex. The sharding function has to be modified and data has to be moved around servers. Consistent hashing technique may be used instead.

- Celebrity problem - some shards may be unevenly overwhelmed by read-write operations and may need further partitioning.

Transactions

The main advantage of using single node relational databases is the fact that they fully support the concept of a transaction.

- **A**tomicity - operations within transaction either completely succeed or abort.
- **C**onsistency - operations within transaction do not violate invariants or data integrity.
- **I**solation - different transactions do not interfere with each other.
- **D**urability - the data has been written to disk (including WAL).

For a distributed system the isolation part is the most interesting one.

Isolation Levels - Read Committed

Default isolation level in many database implementations.

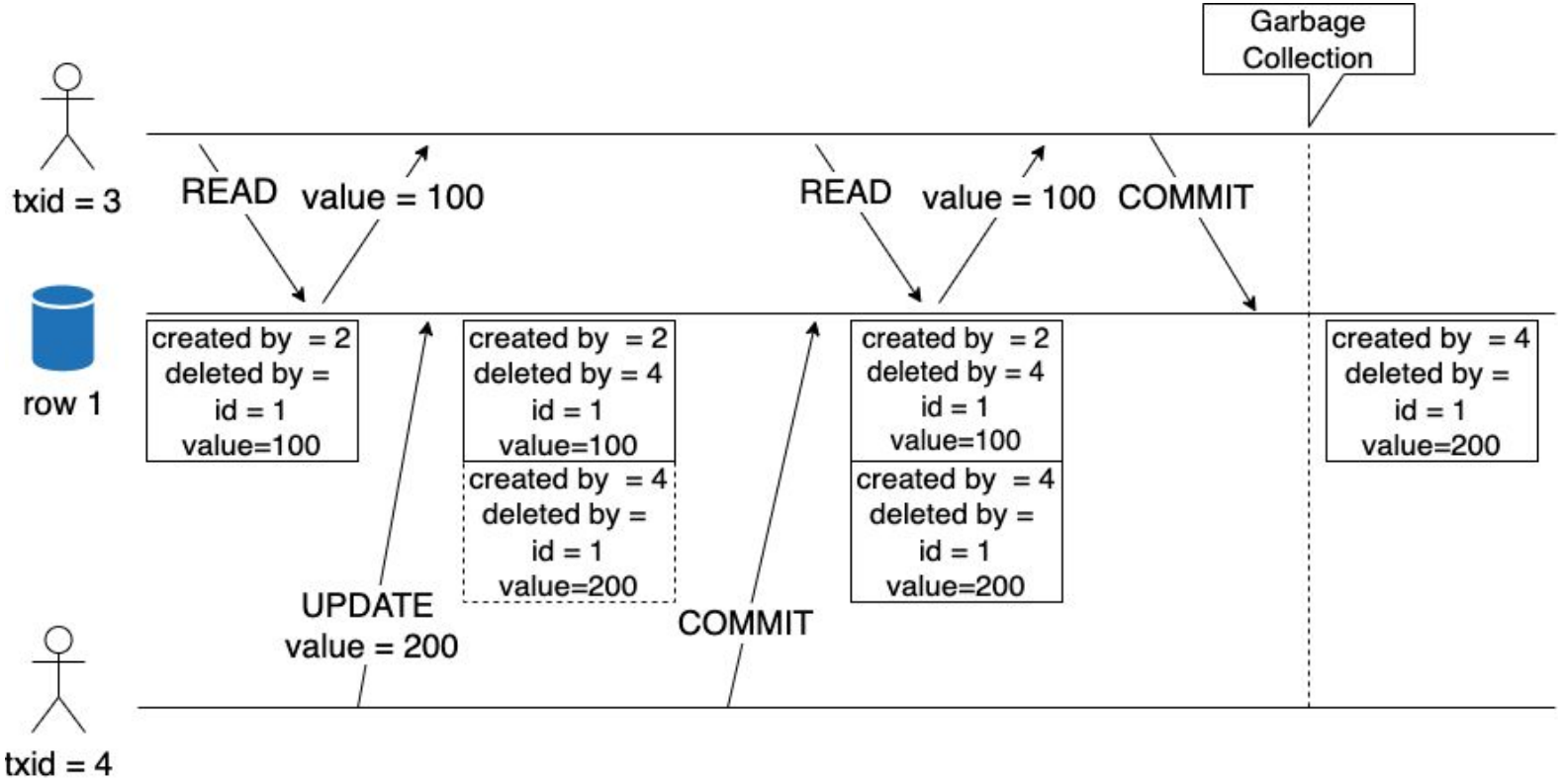
- **Dirty writes** - are not possible, records are locked on writes by concurrent transactions, locks are released upon commit.
- **Dirty reads** - are not possible, the concurrent transactions hold their changes in memory until committed.
- **Non-repeatable reads** - within transaction are possible, once the concurrent transactions commit, their values are visible.
- **Read skew** - transaction may see an inconsistent state due to non-repeatable reads. Backups, integrity checks suffer.

Isolation Levels - Repeatable Read

Same features as read committed but guarantees no read skew. Use for long running analytical queries and backups.

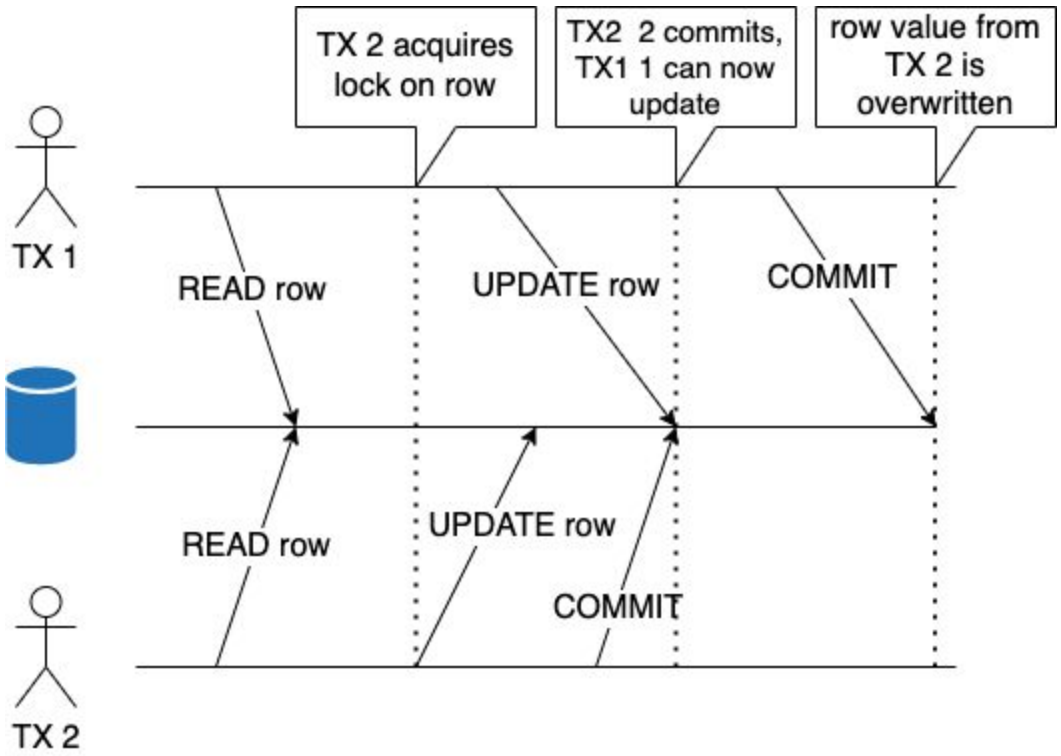
Snapshot isolation - readers never block writers and writers never block readers, implemented using MVCC (Multiversion Concurrency Control) techniques.

Multiversion Concurrency Control



Lost Updates

Application read-modify-write cycles may unintentionally overwrite result values of previous transactions.



Lost Updates - Prevention

- **Atomic operations:**

```
UPDATE counter SET value = value + 1 WHERE name='counter1';
```

- **Compare and set operation** (might not work with repeatable read isolation level):

```
UPDATE counter SET value = 6 WHERE name='counter1' AND value = 5;
```

- **Automatic detection** - transaction manager detects a lost update and aborts the transaction in a repeatable read isolation level.
- **Explicit locking:**

```
SELECT value FROM counter WHERE name='counter1' FOR UPDATE;
UPDATE counter SET value = 6 WHERE name='counter1' ;
```

Write Skew

Anomaly that can happen when two transactions read the same objects concurrently and then update/insert some of the objects. This may violate the invariant condition deduced from the read.

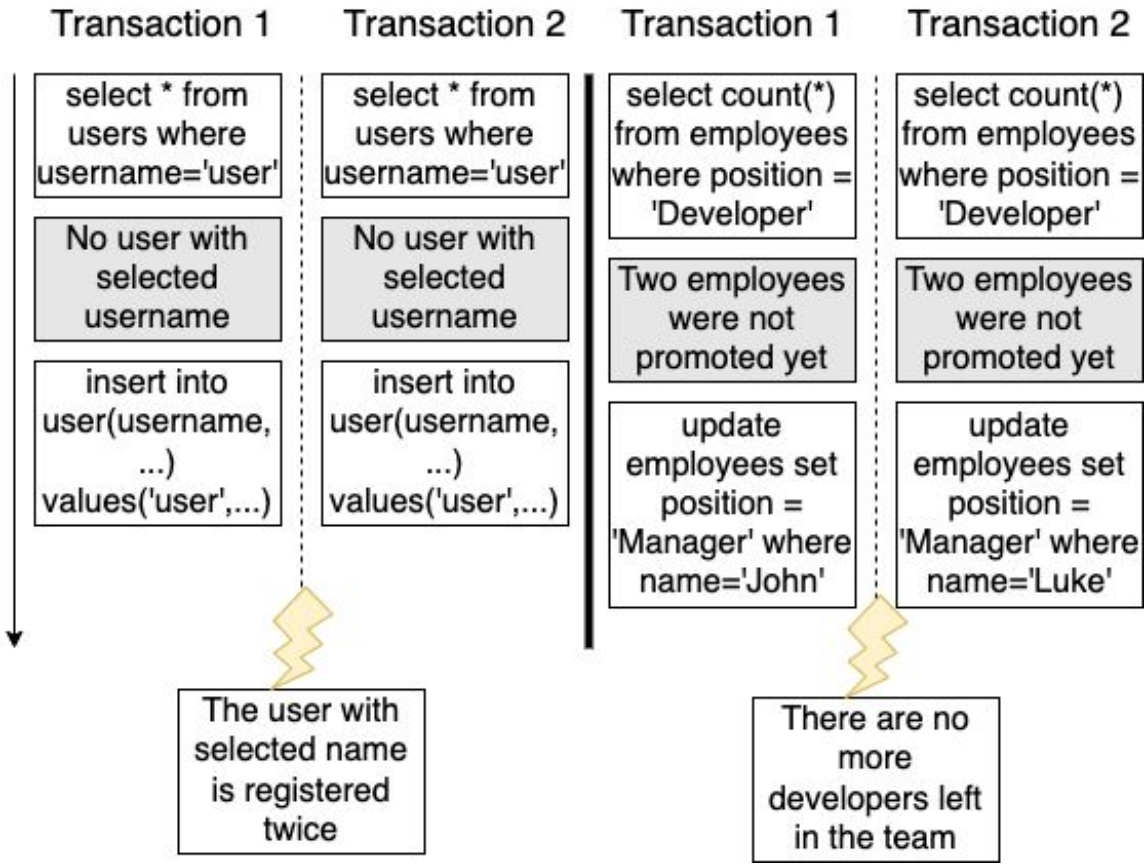
1. Select query checks whether some requirement is satisfied by searching for rows that match some condition.
2. Based on the result the application decides to write data and commit.
3. The write may change the entry condition in another running concurrent transaction, thus making that transaction illegal.

Phantoms - the write in one transaction changes search query results in another transaction.

Write Skew

How to avoid write skew?

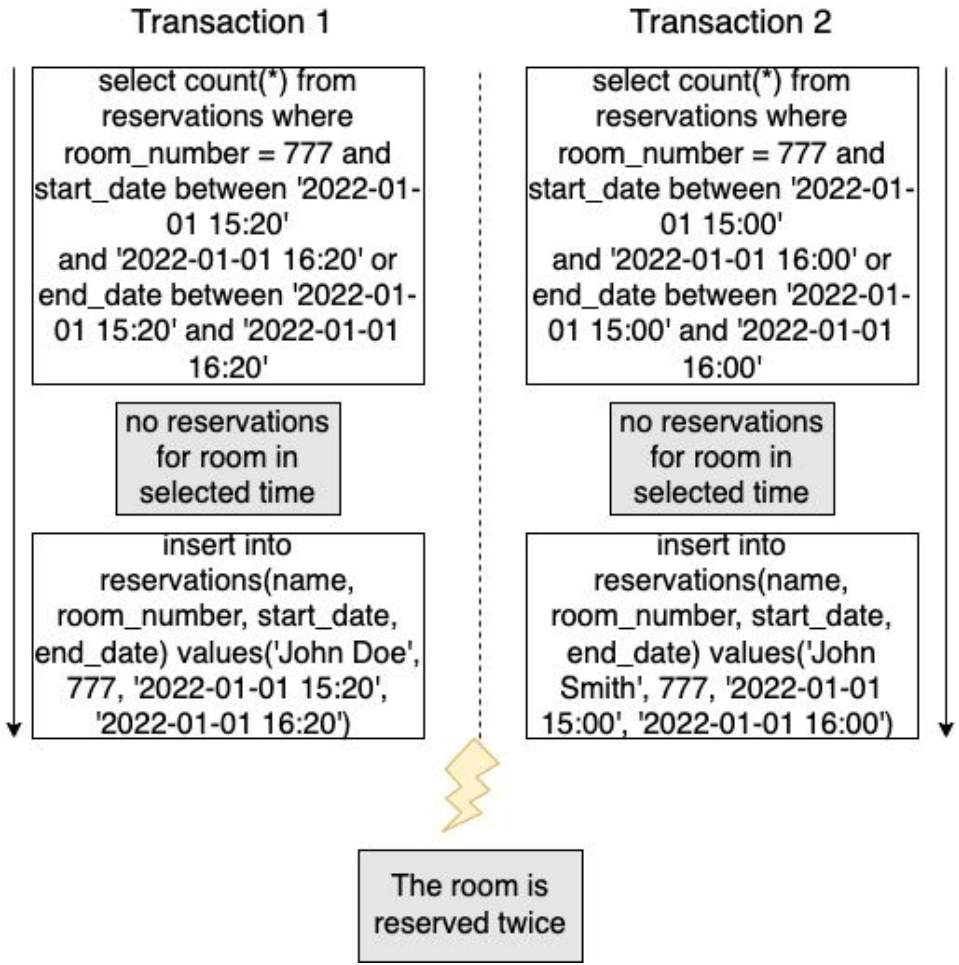
- Switch to serializable isolation level.
- Use database constraints.
- Use explicit locking on rows that the transaction depends on.



Write Skew

How to avoid write skew?

- Materialize conflicts - when there is no object to attach the locks, some artificial data must be introduced to perform the locking.



Isolation Levels - Serializable

Transactions may run in parallel but the overall result is as if they were running serially without any concurrency.

Ways to implement serializable isolation:

- **True serial execution** - no concurrency at all, transactions are executed in a single thread.

- **Two-Phase Locking** - shared and exclusive mode locks (phantoms are prevented by predicate and index locks), possible deadlocks, low performance:
  SELECT … FOR SHARE
  SELECT … FOR UPDATE

Isolation Levels - Serializable

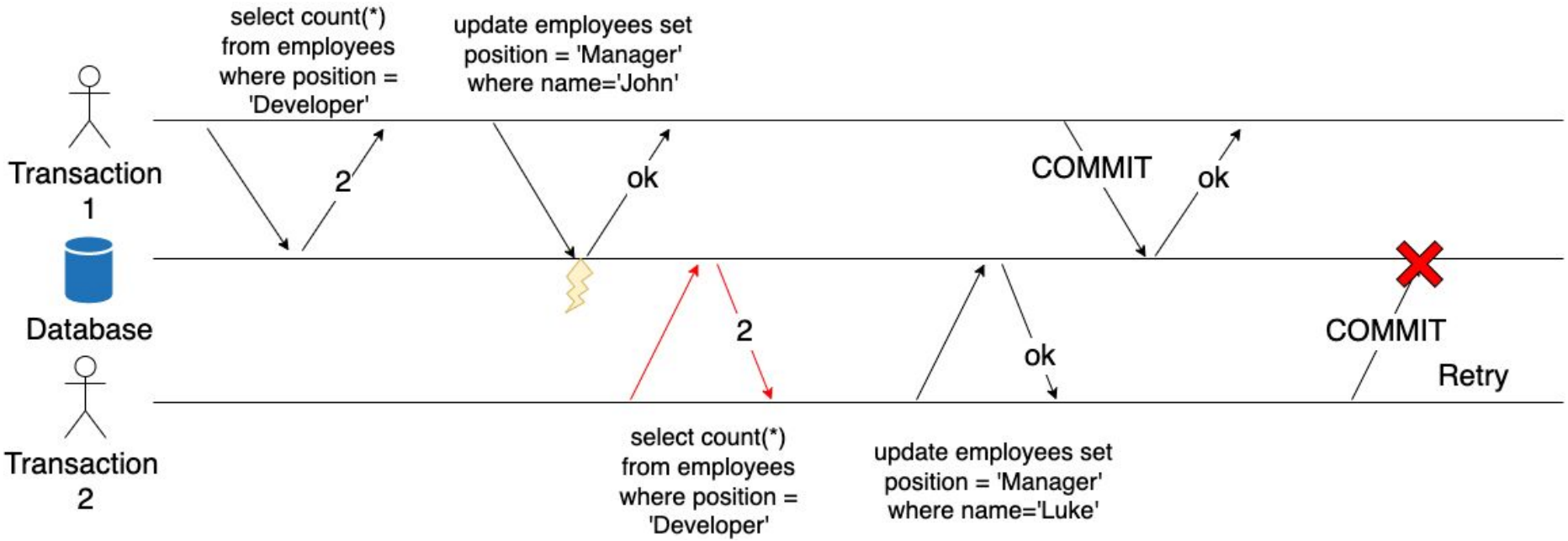
Ways to implement serializable isolation:

- **Serializable Snapshot Isolation -** “optimistic” concurrency control technique. No exclusive locks, conflict detections upon commit. Failed transactions must be re-executed.

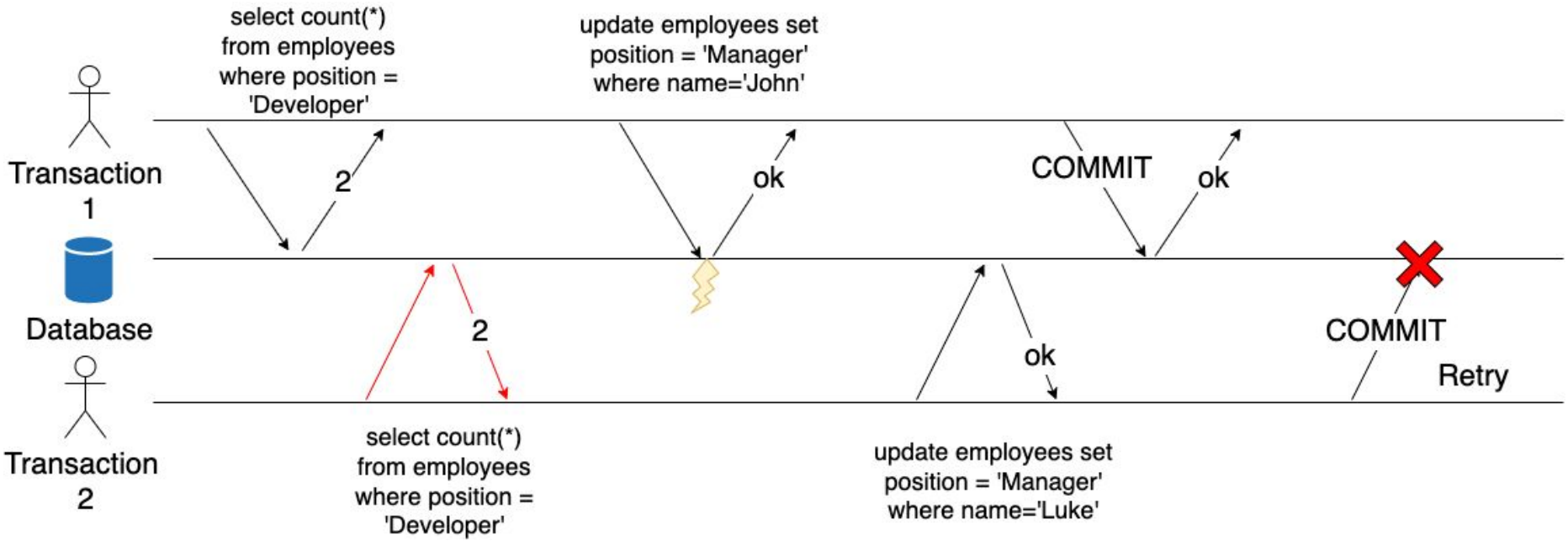
Conflict detection:

- Stale reads - tracking of concurrent and committed transactions which made writes modifying the read premise of a current transaction.
- Writes affecting prior reads - tracking of transactions which made writes modifying prior premise reads and were committed.

Serializable Snapshot Isolation

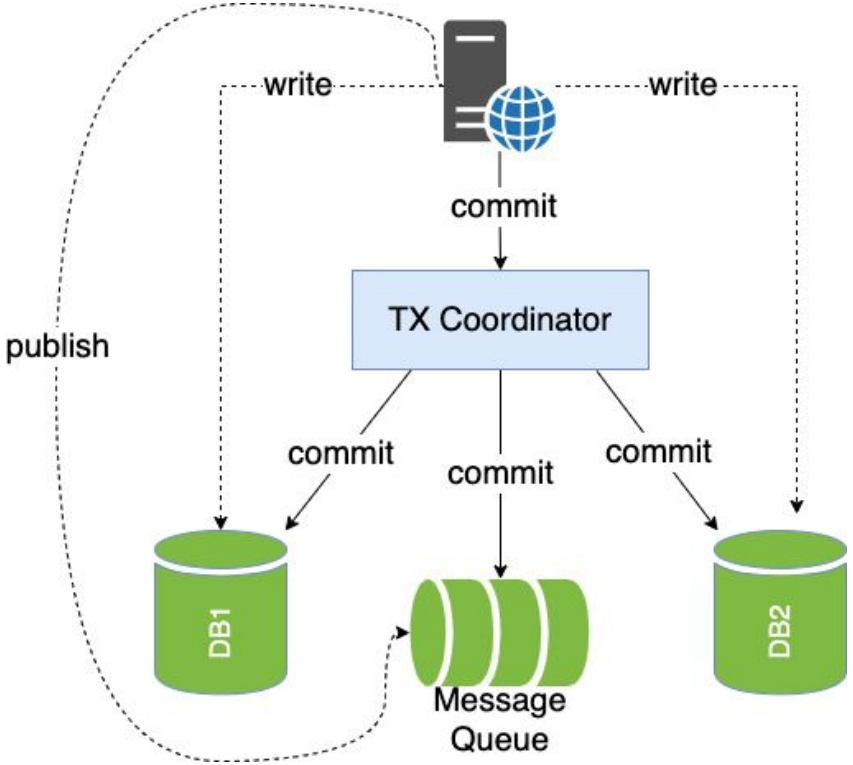


Serializable Snapshot Isolation

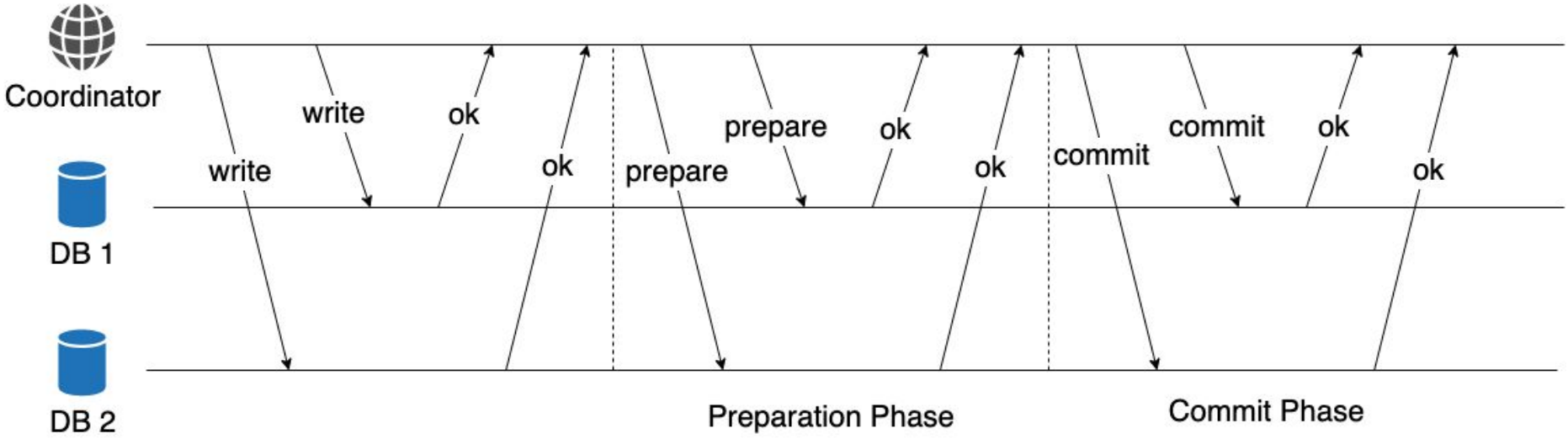


Distributed Transactions

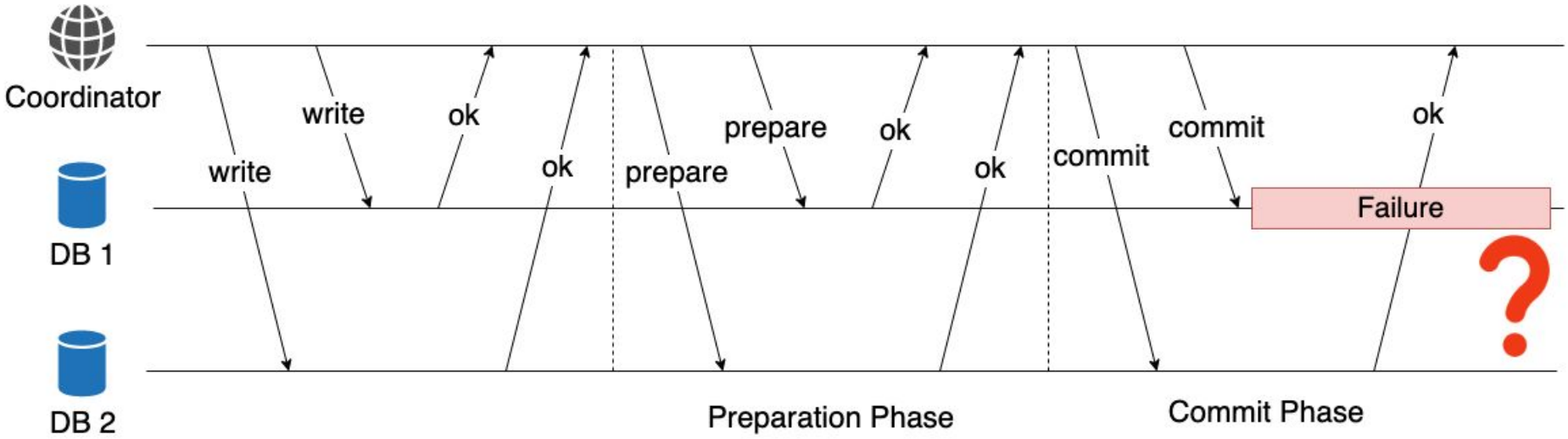
- Distributed transactions may include 2 or more databases.
- The distributed transactions are usually based around 2 Phase Commit algorithm.
- Transaction is governed by a Coordinator, which can be either external or internal to the application.
- The XA standard can be also implemented by Message Queues and other types of external storage.



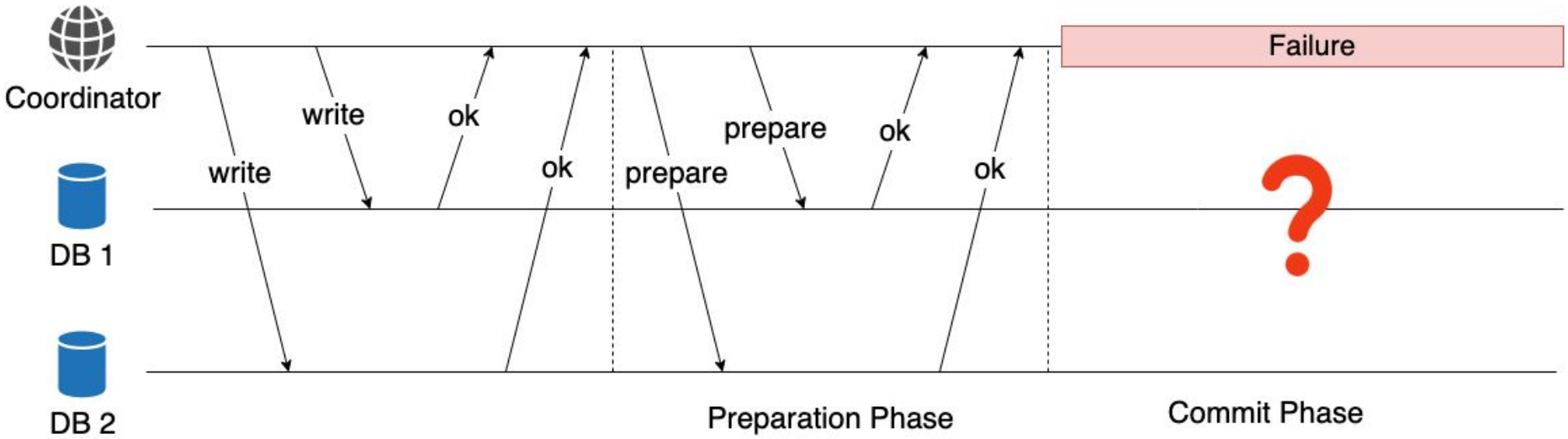
Distributed Transactions - 2 Phase Commit



Distributed Transactions - 2 Phase Commit



Distributed Transactions - 2 Phase Commit



Distributed Transactions - 2 Phase Commit

- 2 Phase Commit in the worst case of coordinator failure assumes the intervention of a system administrator.
- Coordinator is usually a single point of failure.
- If coordinator is embedded within an application then application must persist the coordinator logs in order to recover failed transaction after a crash.
- Perhaps the better option is to use compensating transactions?

Summary

We have discussed:

- The aspects of replication, load balancing, high availability of RDBMS.
- Weak transaction isolation levels, methods of achieving serializability.
- Distributed transactions.