

DS 4300

# Moving Beyond the Relational Model

Mark Fontenot, PhD  
Northeastern University

# Benefits of the Relational Model

- (Mostly) Standard Data Model and Query Language
  - ACID Compliance (more on this in a second)
    - Atomicity, Consistency, Isolation, Durability
  - Works well with highly structured data
  - Can handle large amounts of data
  - Well understood, lots of tooling, lots of experience
- \*Relational model works well for some datasets, not well for others

# Relational Database Performance

Many ways that a RDBMS increases efficiency:

- indexing (the topic we focused on)
- directly controlling storage
- column oriented storage vs row oriented storage \*(allows you to focus only on certain attributes (in cols))
- query optimization
- caching/prefetching \*(predict data you will need next (pull from hard drive into RAM))
- materialized views
- precompiled stored procedures \*(database functions)
- data replication and partitioning

# Transaction Processing

- **Transaction** - a sequence of one or more of the CRUD operations performed as a single, logical unit of work
  - Either the entire sequence succeeds (COMMIT)
  - OR the entire sequence fails (ROLLBACK or ABORT) \*(all or nothing)
- Help ensure
  - Data Integrity
  - Error Recovery
  - Concurrency Control
  - Reliable Data Storage
  - Simplified Error Handling

# ACID Properties

- **Atomicity**

- transaction is treated as an atomic unit - it is fully executed or no parts of it are executed

- **Consistency**

- a transaction takes a database from one consistent state to another consistent state \*(DB is never left in inconsistent state (ex: partially executed transaction))
- consistent state - all data meets integrity constraints

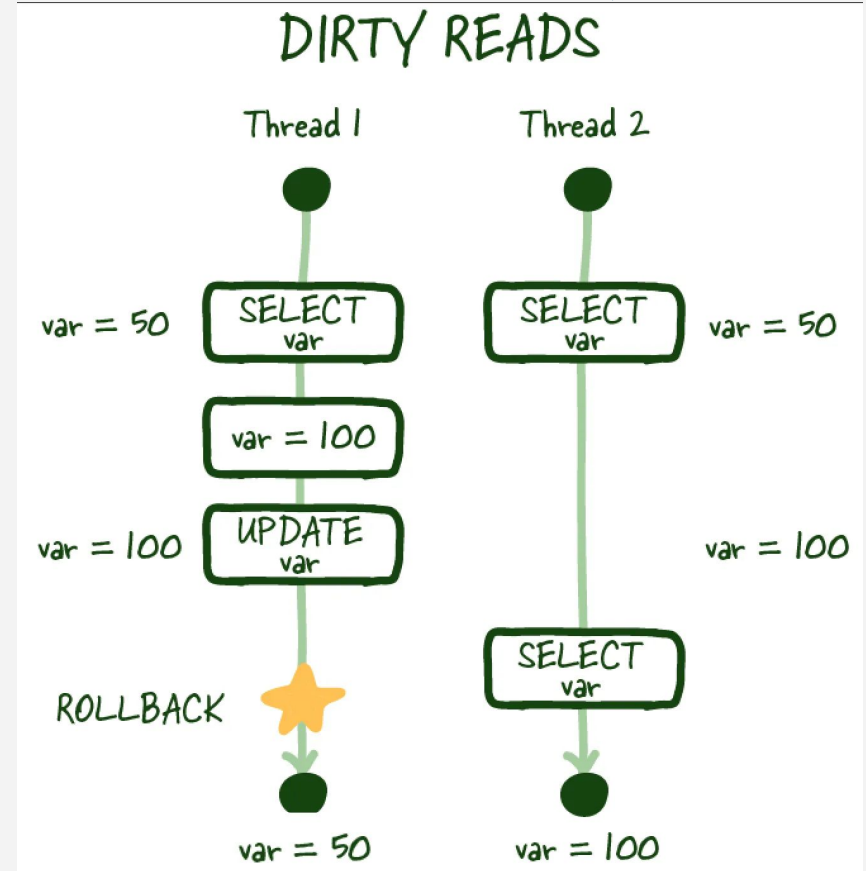
\*ACID stands for Atomicity, Consistency, Isolation, Durability

# \*Isolation is most varied in implementation **ACID Properties**

- **Isolation** \*(ensured through locking -> different databases lock at different levels of granularity (ex: row) -> different kinds of locking (read, writes, etc.))
  - Two transactions  $T_1$  and  $T_2$  are being executed at the same time but cannot affect each other
  - If both  $T_1$  and  $T_2$  are reading the data - no problem
  - If  $T_1$  is reading the same data that  $T_2$  may be writing, can result in: \*(how transactions can affect each other)
    - Dirty Read
    - Non-repeatable Read
    - Phantom Reads

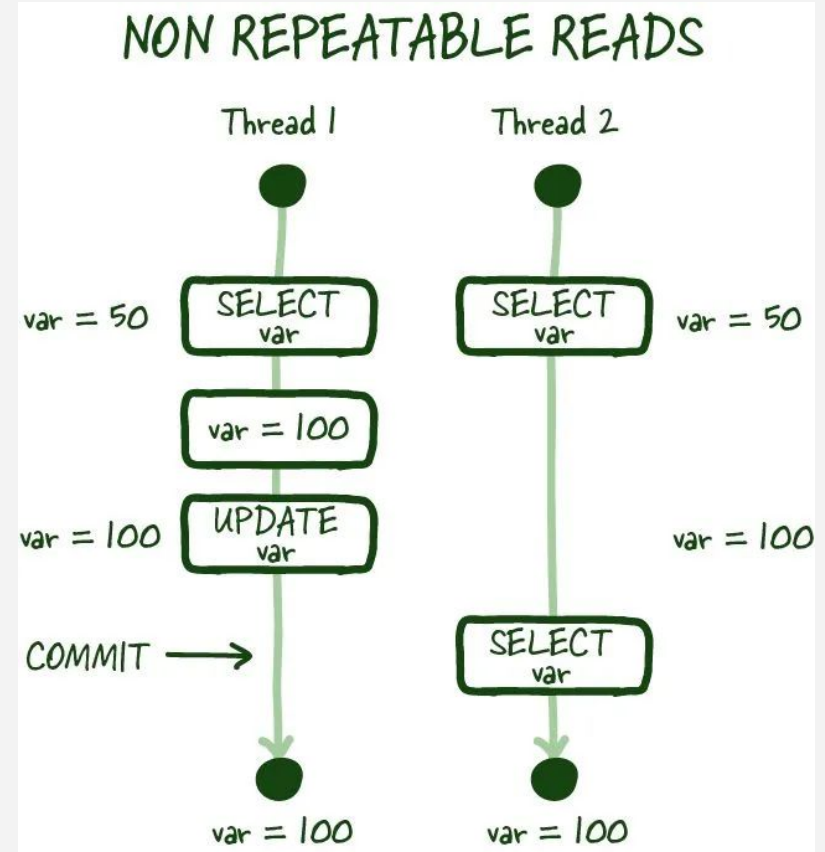
# Isolation: Dirty Read

**Dirty Read** - a transaction  $T_1$  is able to read a row that has been modified by another transaction  $T_2$  that hasn't yet executed a COMMIT



# Isolation: Non-Repeatable Read

**Non-repeatable Read** - two queries in a single transaction  $T_1$  execute a `SELECT` but get different values because another transaction  $T_2$  has changed data and **COMMITTED** \*(so can't commit same read)

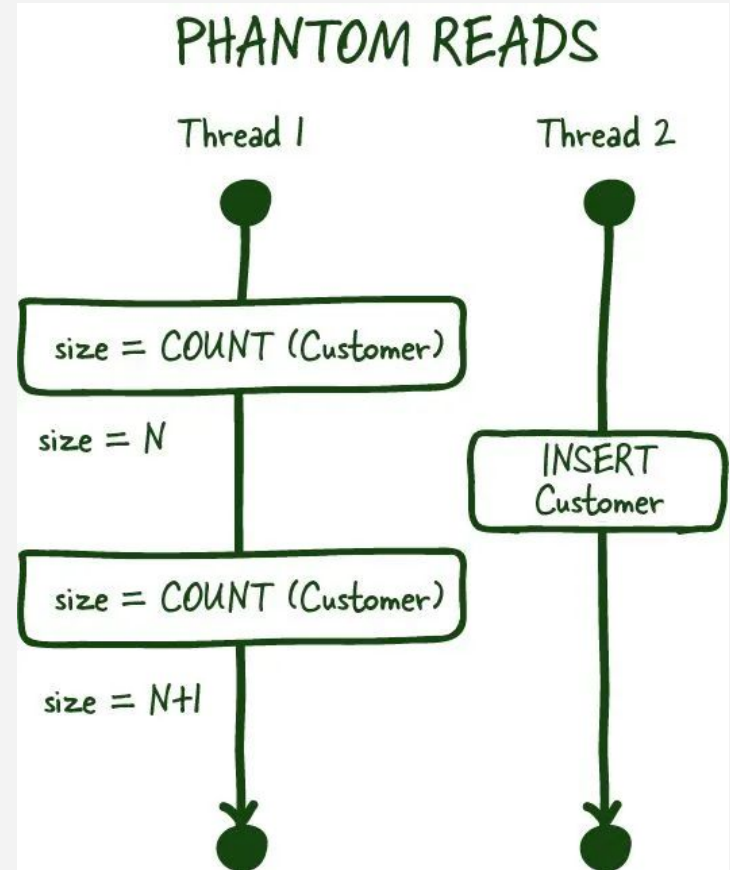




# Isolation: Phantom Reads

**Phantom Reads** - when a transaction  $T_1$  is running and another transaction  $T_2$  adds or deletes rows from the set  $T_1$  is using

\*during phantom read, a thread reads something I didn't insert in middle of my transaction



# Example Transaction - Transfer \$\$

```
DELIMITER //
```

```
CREATE PROCEDURE transfer(  
    IN sender_id INT,  
    IN receiver_id INT,  
    IN amount DECIMAL(10,2)  
)
```

```
BEGIN
```

```
    DECLARE rollback_message VARCHAR(255)  
        DEFAULT 'Transaction rolled back: Insufficient funds';  
    DECLARE commit_message VARCHAR(255)  
        DEFAULT 'Transaction committed successfully';
```

```
-- Start the transaction
```

```
START TRANSACTION;
```

```
-- Attempt to debit money from account 1
```

```
UPDATE accounts SET balance = balance - amount WHERE account_id = sender_id;
```

```
-- Attempt to credit money to account 2
```

```
UPDATE accounts SET balance = balance + amount WHERE account_id = receiver_id;
```

```
-- Continued Next Slide
```

# Example Transaction - Transfer \$\$

```
-- Continued from previous slide

-- Check if there are sufficient funds in account 1
-- Simulate a condition where there are insufficient funds
IF (SELECT balance FROM accounts WHERE account_id = sender_id) < 0 THEN
  -- Roll back the transaction if there are insufficient funds
  ROLLBACK;
  SIGNAL SQLSTATE '45000'      -- 45000 is unhandled, user-defined error
    SET MESSAGE_TEXT = rollback_message;
ELSE
  -- Log the transactions if there are sufficient funds
  INSERT INTO transactions (account_id, amount, transaction_type)
    VALUES (sender_id, -amount, 'WITHDRAWAL');
  INSERT INTO transactions (account_id, amount, transaction_type)
    VALUES (receiver_id, amount, 'DEPOSIT');

  -- Commit the transaction
  COMMIT;
  SELECT commit_message AS 'Result';
END IF;
END //
```

DELIMITER ;

# ACID Properties

- **Durability**

- Once a transaction is completed and committed successfully, its changes are permanent.
  - Even in the event of a system failure, committed transactions are preserved
- For more info on Transactions, see:
  - Kleppmann Book Chapter 7

# But ...

Relational Databases may not be the solution to all problems...

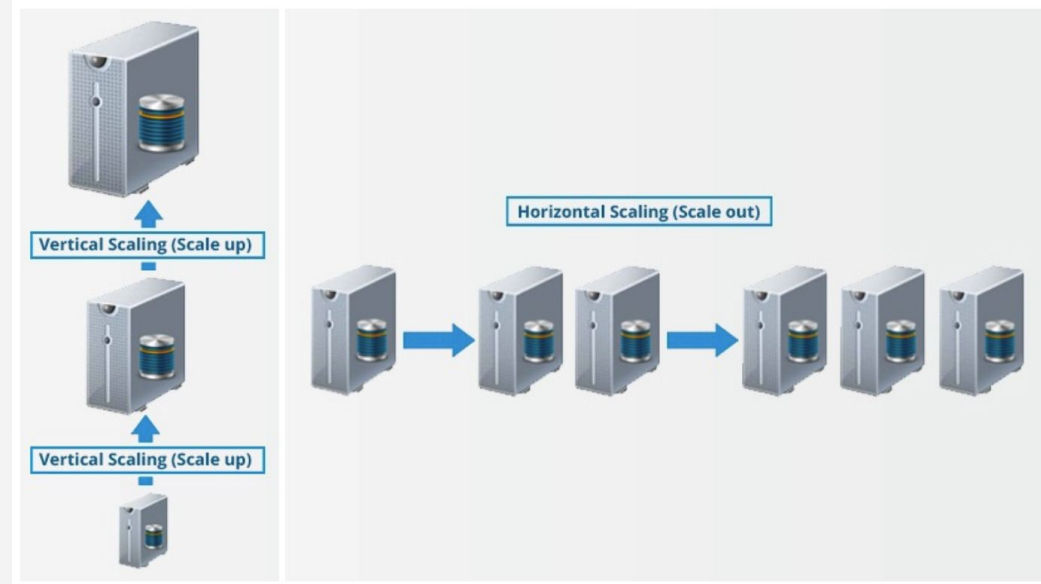
- sometimes, schemas evolve over time
- not all apps may need the full strength of ACID compliance
- joins can be expensive
- a lot of data is semi-structured or unstructured (JSON, XML, etc)
- Horizontal scaling presents challenges
- some apps need something more performant (real time, low latency systems) \*(joins, ACID compliance slows down RDBMS's)

# Scalability - Up or Out?

**Conventional Wisdom:** Scale vertically (up, with bigger, more powerful systems) until the demands of high-availability make it necessary to scale out with some type of distributed computing model \*(power is not always the answer but it is easiest)

**But why?** Scaling up is easier - no need to really modify your architecture. But there are practical and financial limits

**However:** There are modern systems that make horizontal scaling less problematic.



\*Vertical scaling: scale up, Horizontal scaling: scale out

## So what? Distributed Data when Scaling Out

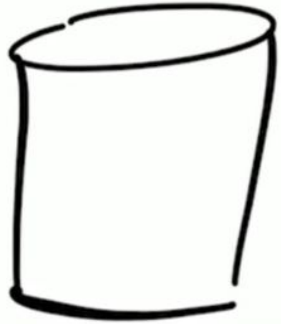
**A distributed system** is *“a collection of independent computers that appear to its users as one computer.”* -Andrew Tennenbaum

### Characteristics of Distributed Systems:

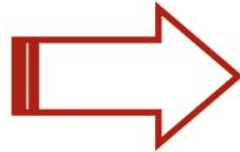
- computers operate concurrently
- computers fail independently
- no shared global clock

# Distributed Storage - 2 Directions

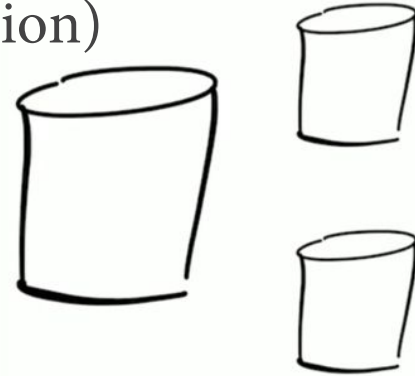
\*Replication: making multiple copies (also keeps data safe if main machine crashes, allows for geographic distribution)



Single  
Main  
Node

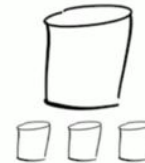


**Replication:**

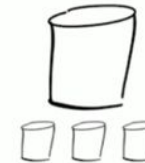


**Sharding:**

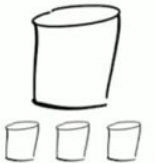
Aaron-  
Frances



Frances-  
Nancy



Nancy-  
Zed



\*Sharding: splitting by some attribute



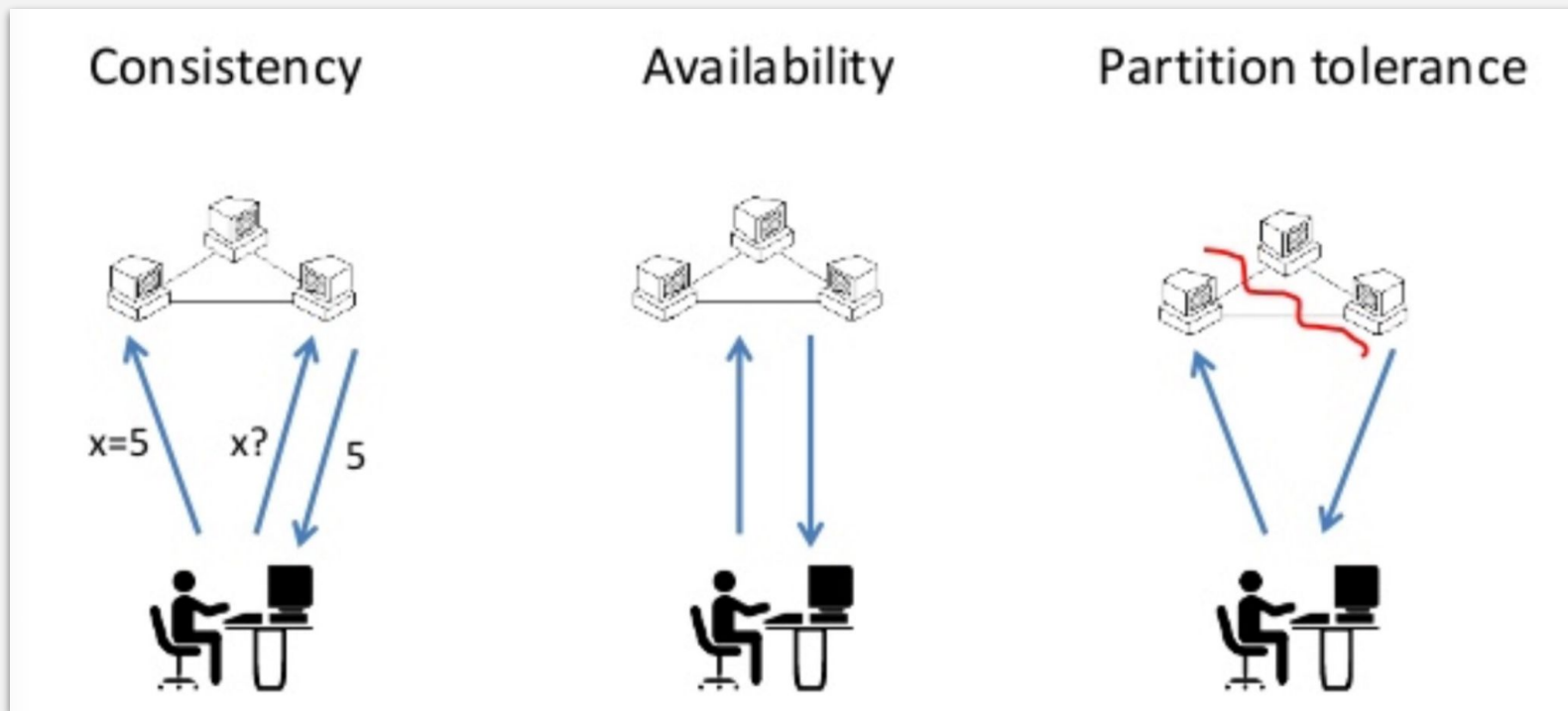
# Distributed Data Stores

- Data is stored on  $> 1$  node, typically replicated
  - i.e. each block of data is available on  $N$  nodes
- Distributed databases can be relational or non-relational
  - MySQL and PostgreSQL support replication and sharding
  - CockroachDB - new player on the scene
  - Many NoSQL systems support one or both models
- But remember: **Network partitioning is inevitable!**
  - network failures, system failures
  - Overall system needs to be **Partition Tolerant**
    - System can keep running even w/ network partition

\*CAP Theorem is used to evaluate one data system over another

# The CAP Theorem

\*Consistency in the CAP Theorem is not the same as consistency in ACID



\*CAP in CAP Theorem stands for Consistency, Availability, Partition Tolerance

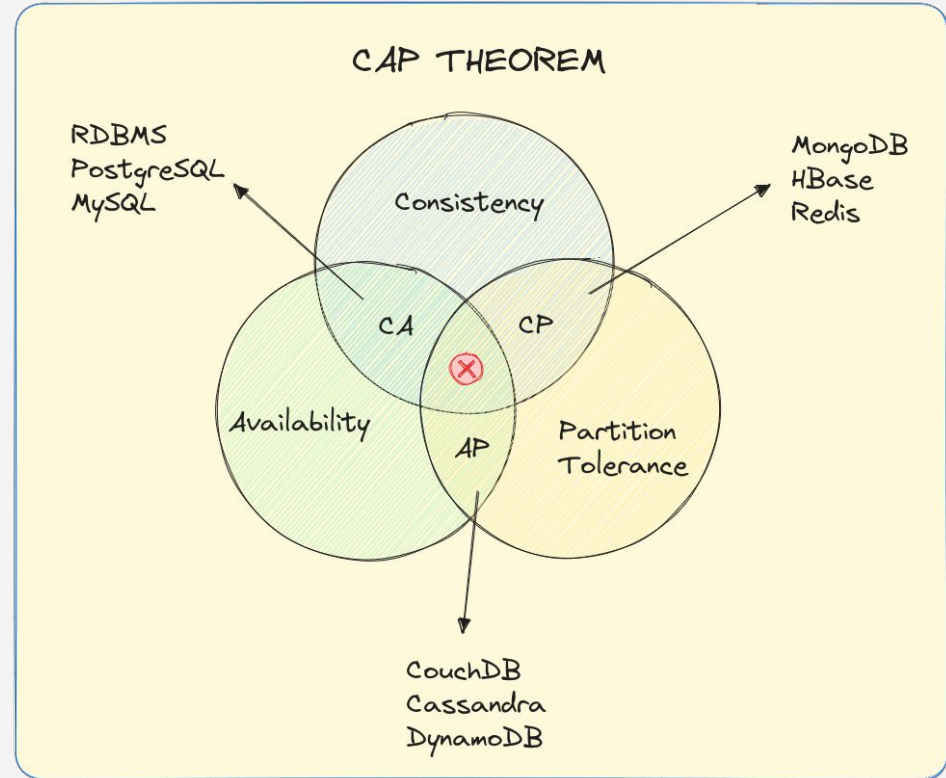
# The CAP Theorem

The **CAP Theorem** states that it is impossible for a distributed data store to *simultaneously* provide more than two out of the following three guarantees:

- **Consistency** - Every read receives the most recent write or error thrown
- **Availability** - Every request receives a (non-error) response - but no guarantee that the response contains the most recent write
- **Partition Tolerance** - The system can continue to operate despite arbitrary network issues.

# CAP Theorem - Database View

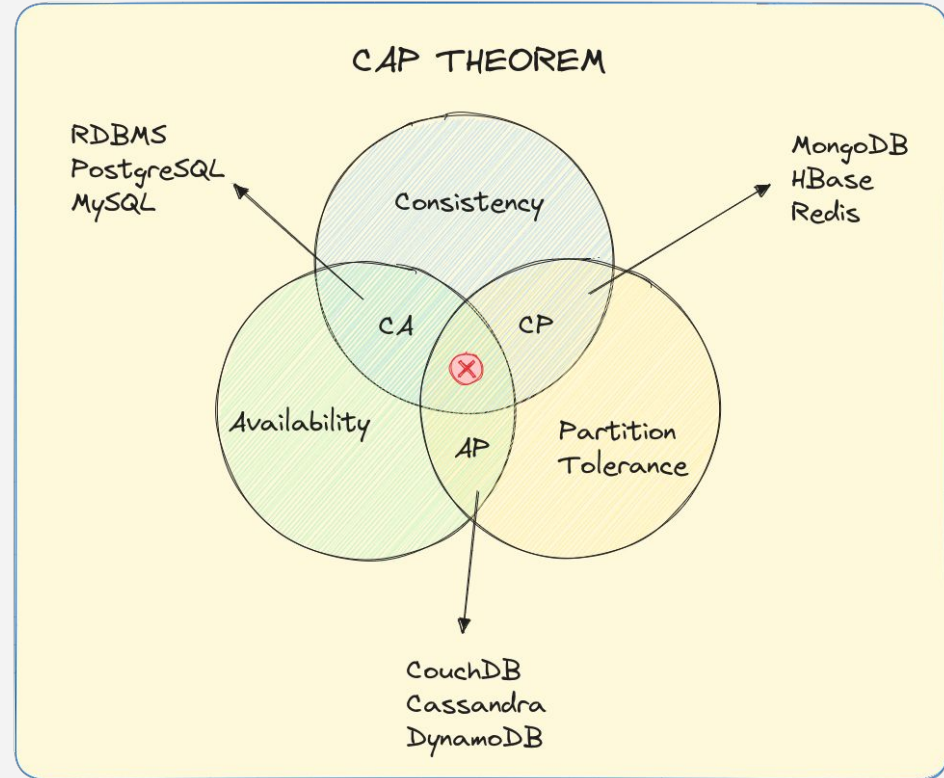
- **Consistency\***: Every user of the DB has an identical view of the data at any given instant
- **Availability**: In the event of a failure, the database remains operational
- **Partition Tolerance**: The database can maintain operations in the event of the network's failing between two segments of the distributed system



\* Note, the definition of Consistency in CAP is different from that of ACID.

# CAP Theorem - Database View

- **Consistency + Availability:** System always responds with the latest data \*(consistency) and every request gets a response \*(availability), but may not be able to deal with network issues
- **Consistency + Partition Tolerance:** If system responds with data \*(partition tolerance) from a distributed store, it is always the latest \*(consistency), else data request is dropped \*(partition tolerance).
- **Availability + Partition Tolerance:** System always sends are responds based on distributed store \*(availability), but may not be the absolute latest data. \*(partition tolerance)



# CAP in Reality

What it is really saying:

- If you cannot limit the number of faults, requests can be directed to any server, and you insist on serving every request, then you cannot possibly be consistent.

But it is interpreted as:

- You must always give up something: consistency, availability, or tolerance to failure. (partition tolerance)

??