02/03/25, 02/05/25 - NoSQL & KV DBs

Distributed DBs and ACID - Pessimistic Concurrency

ACID transactions

- Focuses on “data safety”
- Considered a *pessimistic concurrency model* because it assumes one transaction has to protect itself from other transactions
    - It assumes that if something can go wrong, it will
- Conflicts are prevented by locking resources until a transaction is complete
    - Both read and write locks
    - Write lock analogy: borrowing a book from a library—if you have it, no one else can
    - Requires some overhead

Optimistic Concurrency

Transactions do not obtain locks on data when they read or write

Optimistic because it assumes conflicts are unlikely to occur

- Even if there is a conflict, everything will still be ok

How?

- Add last update timestamp and version number columns to every table
- Read them when changing
- Check at the end of transaction to see if any other transaction has caused them to be modified

Low Conflict Systems (backups, analytical dbs, etc.)

- Not updated often
- Read heavy systems
- Conflicts that arise can be handled by rolling back and re-running a transaction that notices a conflict
- Optimistic concurrency works well—allows for higher concurrency

High Conflict Systems

- A lot of updates
- Rolling back and rerunning transactions that encounter a conflict → less efficient
- So, a locking scheme (pessimistic model) might be preferable

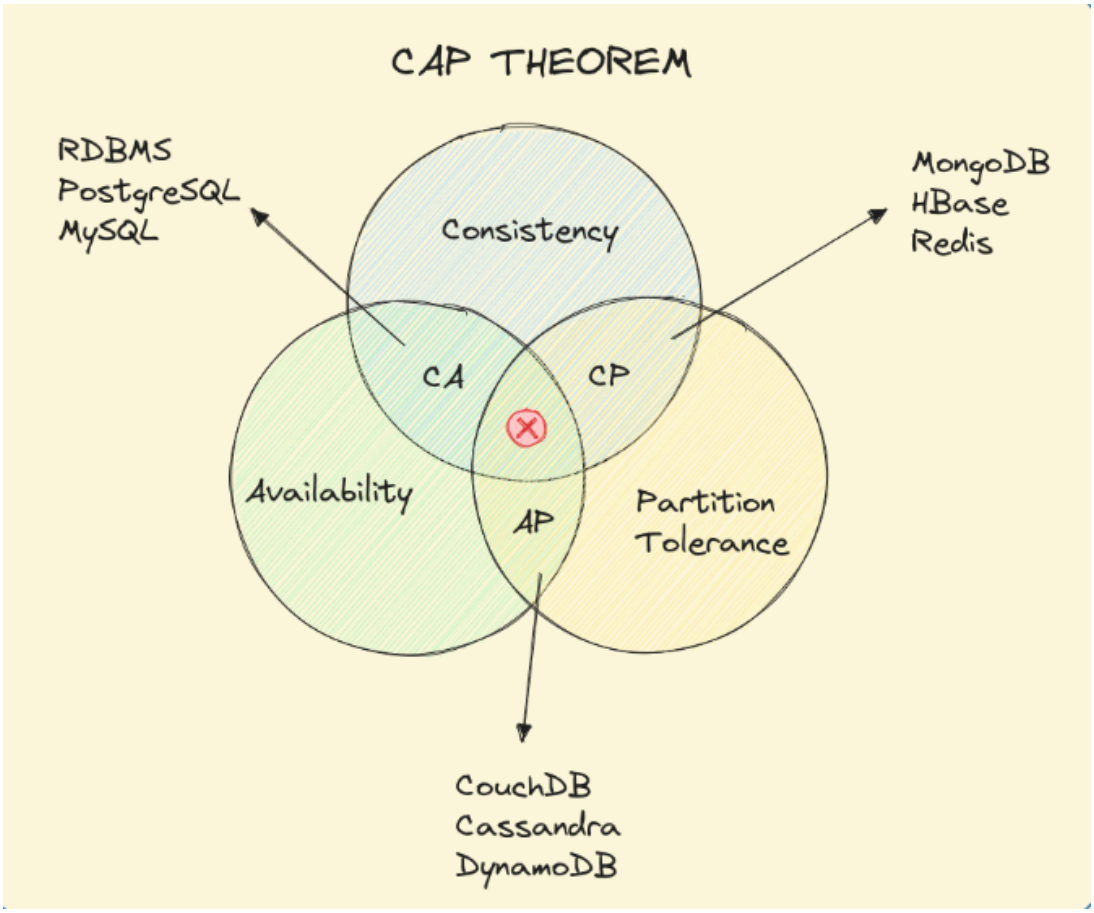
NoSQL

First used in 1998 by Carlo Strozzi to describe his relational database system that did not use SQL

More common, modern meaning is “Not Only SQL”

But, sometimes thought of as non-relational DBs

CAP Theorem



ACID Alternative for Distributive Systems - BASE

Basically Available

Guarantees the availability of the data (per CAP), but response can be “failure/unreliable” because the data is in an inconsistent or changing state

System appears to work most of the time

Soft State

The state of the system could change over time, even without input

Changes could be result of eventual consistency

- Data stores don’t have to be write-consistent

- Replicas don’t have to be mutually consistent

Eventual Consistency

The system will eventually become consistent

- All writes will eventually stop so all nodes/replicas can be updated

Categories of NoSQL DBs

Key-Value

Key-value stores are designed around:

- Simplicity
  - The data model is extremely simple
  - Comparatively, tables in a RDBMS are very complex
  - Lends itself to simple CRUD ops and API creation
- Speed
  - Usually deployed as in-memory DB
    - If the power goes out, the information is gone
  - Retrieving a value given its key is typically a O(1) operation because hash tables or similar data structures used under hood
  - No concept of complex queries or joins—they slow things down
- Scalability
  - Horizontal scaling is simple—add more nodes
  - Typically concerned with eventual consistency
    - In a distributed environment, the only guarantee is that all nodes will eventually converge on the same value

Use Cases

EDA/Experimentation Results Store

- Store intermediate results from data preprocessing and EDA
- Store experiment or testing (A/B) results without product database

Feature Store

- Store frequently accessed feature → low-latency retrieval for model training and prediction

Model Monitoring

- Store key metrics about performance of model, for example, in real-time inferencing

Storing Session Information

- Everything about the current session can be stored via a single PUT or POST and retrieved with a single GET—very fast

User Profiles and Preferences

- User info can be obtained with a single GET operation—language, time zone, product or UI preferences

Shopping Cart Data

Caching Layer

- In front of a disk-based database
- Data that is cached (already accessed) is more likely to be accessed again, so it is checked first

Redis (Remote Directory Server)

Open source, in-memory database

Sometimes called a data structure store

Primarily a KV store, but can be used with other models

Most popular KV database

Considered an in-memory database system, but

- Supports durability of data by:
    - Essentially saving snapshots to disk at specific intervals OR

    - Append-only file which is a journal of changes that can be used for roll-forward if there is a failure

Can be very fast

Rich collection of commands

Does not handle complex data

Data Types

Keys:

- Usually strings but can be any binary sequence

Values:

- Strings
- Lists (linked lists)
- Sets
- Sorted sets
- Hashes
- Geospatial data

Setting Up Redis in Docker

In Docker Desktop, search for Redis

Pull/Run the latest image

- Optional settings: add 6379 to Ports to expose that port so we can connect to it

Normally, you would not expose the Redis port for security reasons

- If you did this in a prod environment, major security hole
- We didn’t set a password

Connecting from DataGrip

Redis Database and Interaction

Redis provides 16 databases by default

- Numbered 0 to 15

Direct interaction with Redis is through a set of commands related to setting and getting k/v pairs (and variations)

Many language libraries available as well

Foundation Data Type - String

Sequence of bytes - text, serialized objects, bin arrays

Some Basic Commands

```
SET /path/to/resource 0
SET user:1 "John Doe"
GET /path/to/resource
EXISTS user:1
DEL user:1
KEYS user*

SELECT 5
// select a different database

SET someValue 0
INCR someValue       # increment by 1
INCRBY someValue 10  # increment by 10
DECR someValue       # decrement by 1
DECRBY someValue 5   # decrement by 5
# INCR parses the value as int and increments (or adds to value)

# only sets value to key if key does not already exist
SETNX key value
```

Hash Type

Value of KV entry is a collection of field-value pairs

Use Cases:

- Can be used to represent basic objects/structures
  - Number of field/value pairs per hash is 2^{32-1}
  - Practical limit: available system resources (e.g. memory)
- Session information management
- User/Event tracking (could include TTL)
- Active Session Tracking (all sessions under one hash key