Distributed DBs and ACID - Pessimistic Concurrency:

ACID Transactions

- ACID is a set of properties that guarantee reliable processing of database transactions:
    - Atomicity: Each transaction is treated as a single, indivisible unit. Either all operations in the transaction succeed, or none of them do.
    - Consistency: Transactions take the database from one valid state to another, preserving the integrity constraints.
    - Isolation: Transactions are isolated from each other, ensuring that concurrently executing transactions do not interfere.
    - Durability: Once a transaction is committed, it's permanent and will survive system crashes.

Pessimistic Concurrency Control

- Pessimistic concurrency control is a strategy for managing concurrent access to database resources in a way that prevents conflicts.
    - The key idea behind pessimistic concurrency is that conflicts between transactions are likely to happen, so the system assumes that if something can go wrong, it probably will. As a result, it proactively prevents conflicts from happening during the execution of transactions.
    - In other words, it operates on the assumption that other transactions will interfere, so it uses techniques to lock resources to avoid conflicts.

How Does Pessimistic Concurrency Work?

- Locking Resources:
    - To avoid concurrent transactions from interfering with each other, the database will lock resources (e.g., rows, tables, or even entire databases) until the transaction is complete.
    - This ensures that no other transaction can modify the same data at the same time.
    - There are two types of locks used:
        1. Read Locks: Prevent other transactions from writing to the resource but allow them to read it.
        2. Write Locks: Prevent other transactions from reading or writing to the resource until the lock is released.
    - This creates a serialized execution of transactions, ensuring consistency and isolation but at the cost of performance and concurrency.

Write Lock Analogy

Analogy: Borrowing a Book from a Library

- Imagine you want to borrow a book from the library. When you take the book, no one else can borrow it until you return it.
- This is similar to a write lock: when a transaction locks a resource (e.g., a row in a database), no other transaction can access it (either for reading or writing) until the transaction is completed and the lock is released.

- Example:
    - Transaction A locks "Book A" (read/write lock).
    - Transaction B wants to read "Book A" but cannot until Transaction A finishes and releases the lock.

Why Pessimistic Concurrency?

- Data Safety:
    - The focus of pessimistic concurrency is on data safety and ensuring that transactions are executed correctly without interference.
    - By using locks, the system prevents conflicts and ensures that only one transaction can alter data at any given time, preventing inconsistencies or corruption.
- Use Cases:
    - Pessimistic concurrency control is ideal for environments where conflicts are highly likely or when data integrity is critical (e.g., financial systems, inventory systems, or applications with high-value transactions).

Challenges of Pessimistic Concurrency

- Deadlock: Since transactions hold locks, they can sometimes end up in a situation where two transactions are waiting on each other to release a resource, leading to a deadlock. The system must have a mechanism for detecting and resolving deadlocks, often by aborting one of the transactions to break the cycle.
- Performance Impact: Locking can significantly reduce throughput and increase response times, especially when there are many concurrent transactions. Since each transaction has to wait for others to release locks, this leads to inefficiencies in highly concurrent environments.
    - Additionally, the more data that is locked, the greater the chance of contention between transactions, further slowing down performance.

Optimistic Concurrency:

Overview

- In optimistic concurrency control, transactions do not acquire locks on data when reading or writing. Instead, the system assumes that conflicts will be rare, and thus transactions proceed under the assumption that no other transaction will interfere with their work.
- Optimistic because it works under the assumption that conflicts between transactions are unlikely to happen, and even if they do, it's not a problem. This approach minimizes contention for resources and allows for more concurrent transactions.

How Does Optimistic Concurrency Work?

1. Reading Data:
    - A transaction reads the data it needs to work with and also fetches a timestamp or version number associated with the data.
    - These are used to track when the data was last modified and are attached to each row in the database.
2. Making Changes:

    - The transaction proceeds with making changes to the data without locking it, assuming no one else is modifying the same data at the same time.
    - During this phase, other transactions can read and write to the same data without interference (because no locks are in place).
3. Checking for Conflicts:
    - Before committing the transaction, it checks if any of the data it worked with has been modified by another transaction since it was first read. This is done by comparing the timestamp or version number that was initially retrieved with the current state of the data.
    - If the data has been modified (i.e., if the timestamp/version number doesn't match), a conflict is detected, and the transaction may be rolled back and retried.
4. Commit:

- If no conflicts are found, the transaction proceeds to commit the changes, and the new timestamp/version number is updated in the database.

<u>Why Is It Optimistic?</u>

- Optimism comes from the assumption that conflicts are rare and that most transactions can proceed without interference. Rather than preemptively locking resources (as in pessimistic concurrency), the system relies on validation at commit time to detect conflicts.
- The idea is that, by not locking data during the transaction, you can achieve higher throughput and better concurrency, especially in systems with lower conflict rates.

<u>Types of Systems Where Optimistic Concurrency Works Well</u>

1. Low Conflict Systems
    - Backups: Systems performing operations like backups where few transactions occur concurrently.
    - Analytical Databases: In environments where data is mostly read-heavy and transactions involve more querying than updating, conflicts are less likely, and optimistic concurrency works effectively.
        - In these systems, conflicts are rare, and when they do occur, it's often acceptable to roll back and retry the transaction.
    - Benefits:
        - Higher throughput: Without the need for locks, more transactions can run concurrently, improving the performance of read-heavy systems.
        - Less contention: Since no locks are being placed, there is no waiting for locks to be released, so overall system performance can improve.

2. Read-heavy Systems
    - In read-heavy systems, most operations involve retrieving data rather than modifying it. Optimistic concurrency is a good fit for these environments because the likelihood of data contention (two transactions trying to modify the same data at the same time) is minimal.

        - Example: A news website where users are reading articles, but only a few are submitting comments or editing content. Most transactions are reads, so the likelihood of conflicts is low, and optimistic concurrency can ensure that the system scales efficiently.

Handling Conflicts in Optimistic Concurrency

- If two transactions try to modify the same data at the same time, optimistic concurrency detects this at the commit phase.
    - If a conflict is detected (i.e., the data has been modified by another transaction after it was read), the transaction is rolled back and the process is retried.
    - This rollback and retry mechanism means the system can handle conflicts without locking data, but it introduces an overhead because transactions need to be retried if conflicts are detected.

High Conflict Systems

- Challenges:
    - In high conflict systems, where many transactions are competing for the same data, the rollback and retry approach becomes less efficient.
    - If conflicts happen frequently, the system might end up spending a lot of time rolling back and retrying transactions, which can reduce performance.
    - In such cases, pessimistic concurrency (using locks to prevent conflicts before they happen) may be preferable, even though it might limit concurrency.
    - High conflict systems may require a more predictable and controlled approach to ensure consistency.

Comparison with Pessimistic Concurrency

- Pessimistic Concurrency (locks data to prevent conflicts before they happen) is more appropriate for high-conflict systems where data contention is frequent, and ensuring consistency is paramount.
- Optimistic Concurrency (checks for conflicts at commit time) is more suitable for low-conflict systems, where the overhead of managing locks is unnecessary and higher concurrency is desirable.

No SQL:

Origin of the Term "NoSQL"

- The term "NoSQL" was first used in 1998 by Carlo Strozzi to describe his lightweight, open-source relational database system that did not use SQL for querying.
- However, the modern understanding of NoSQL databases is quite different from Strozzi's original use of the term.

Modern Meaning: "Not Only SQL"

- Today, NoSQL is generally understood to mean "Not Only SQL", rather than strictly "No SQL" at all.

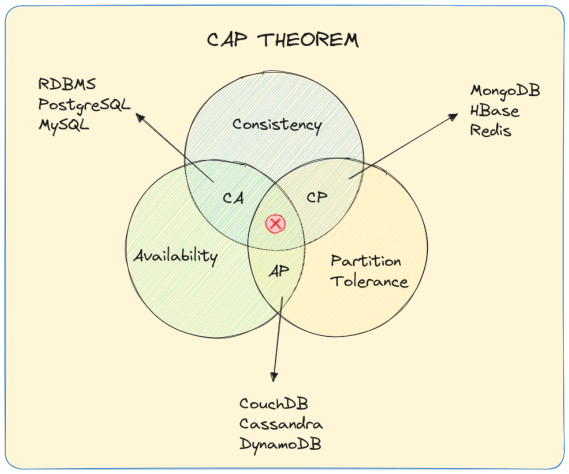
- While NoSQL databases often do not follow the traditional relational database model, some still allow for SQL-like querying or structured querying mechanisms.

Is NoSQL Always Non-Relational?
- NoSQL databases are often thought of as non-relational databases because they do not use the traditional table-based structure with strict schemas like relational databases (RDBMS).
- Instead, they allow for more flexible data models, including:
    - Key-Value Stores (e.g., Redis, DynamoDB)
    - Document Stores (e.g., MongoDB, CouchDB)
    - Column-Family Stores (e.g., Apache Cassandra, HBase)
    - Graph Databases (e.g., Neo4j, ArangoDB)

Why NoSQL?
- NoSQL databases were originally developed in response to the need for handling large-scale, web-based, and unstructured data that relational databases struggled with.
- The rise of Big Data, social media, and cloud computing created new challenges that required scalable, flexible, and high-performance database solutions that could handle:
    - Massive amounts of unstructured or semi-structured data (e.g., JSON, XML, multimedia content).
    - High-speed reads and writes (e.g., caching systems, real-time analytics).
    - Distributed and horizontally scalable architectures (e.g., global applications, cloud-based services).

Key Characteristics of NoSQL Databases
- Schema Flexibility: Unlike relational databases, NoSQL databases do not require predefined schemas. This makes it easier to evolve data models over time.
- Horizontal Scalability: Many NoSQL databases are designed to scale horizontally (adding more machines) rather than vertically (adding more power to a single machine).
- High Availability & Partition Tolerance: NoSQL databases often favor availability and partition tolerance over strict consistency, as per the CAP theorem.
- Optimized for Specific Use Cases: Rather than a one-size-fits-all approach, NoSQL databases specialize in different types of workloads (e.g., key-value stores for caching, document stores for flexible data representation, graph databases for connected data).

CAP Theorem Review



- The CAP theorem, proposed by Eric Brewer in 2000, states that a distributed database system can only guarantee two out of three of the following properties at any given time:
    1. Consistency (C) – Every user of the database has an identical view of the data at any given instant.
        - All nodes return the most recent version of the data.
        - No stale or conflicting versions exist.
        - If a write occurs, all subsequent reads must reflect that change immediately.
    2. Availability (A) – The database remains operational even in the event of failures.
        - Every request always receives a response (though it may not be the latest data).
        - The system does not go down even if some parts fail.
    3. Partition Tolerance (P) – The database can continue to function even if network failures create partitions that temporarily prevent some nodes from communicating.
        - Even if messages between nodes are delayed or lost, the system remains operational.
        - Network partitions are inevitable in distributed systems, so databases must decide how to handle them.

- Trade-Offs: Choosing Two Out of Three
    - Since it is impossible to achieve all three properties simultaneously, distributed databases must prioritize two based on their use case:

    1. Consistency + Availability (CA) → No Partition Tolerance
        - The system always returns the most up-to-date data and never serves stale reads.
        - It remains available under normal conditions but cannot function correctly if the network is partitioned (i.e., when nodes cannot communicate).
        - If a partition occurs, the system may refuse requests or enter a failure mode until connectivity is restored.

        - Example: Traditional relational databases (e.g., PostgreSQL, MySQL running on a single server).
            - A single-node database maintains consistency and availability but fails under network partitions.
2. <u>Consistency + Partition Tolerance (CP) → Reduced Availability</u>
    - The system always returns the most up-to-date data, even in the event of network failures.
    - However, if a partition occurs, some requests may be dropped or the system may become unavailable in order to maintain consistency.
    - This ensures data integrity at the cost of availability.
    - Example: HBase, Google Bigtable, MongoDB (with strong consistency settings)
        - Many banking systems prioritize CP because ensuring correct balances is more important than availability.
3. <u>Availability + Partition Tolerance (AP) → Eventual Consistency</u>
    - The system remains operational and responsive even when network partitions occur.
    - However, users may see slightly stale data due to eventual consistency.
    - Over time, updates propagate, and all nodes eventually reach a consistent state.
    - Example: DynamoDB, Cassandra, CouchDB, Riak
        - Content delivery networks (CDNs) prioritize availability and partition tolerance so that users always get content, even if it's not the absolute latest version.

<u>ACID Alternative for Distrib Systems - BASE</u>

- In distributed systems, strict ACID (Atomicity, Consistency, Isolation, Durability) guarantees can be difficult to maintain due to scalability constraints and the realities of network partitions (per the CAP theorem). To address this, many modern distributed databases adopt a more flexible approach known as BASE:
- BASE is an alternative consistency model designed to prioritize availability and scalability over strict consistency. It consists of three key principles:
    1. **Basically Available:**
        - The system guarantees availability (per the CAP theorem).
        - However, responses might indicate that data is incomplete, temporarily inconsistent, or even in a failure state due to ongoing updates.
        - The system "works most of the time", but data accuracy is not always immediate.
    2. **Soft State:**
        - The system's state can change over time, even without additional input.
        - This occurs due to eventual consistency mechanisms, such as background synchronization or replication processes.
        - Unlike ACID databases, BASE does not require immediate consistency across all replicas.

3. **Eventual Consistency:**
    - While data updates may not be immediately reflected across all nodes, the system guarantees that given enough time and no new updates, all replicas will converge to the same state.
    - All writes will eventually propagate across the distributed system, ensuring consistency in the long run.
    - This is a fundamental property of AP (Availability + Partition Tolerance) systems from the CAP theorem.

- Examples of BASE-Oriented Databases
    - NoSQL Databases (e.g., Apache Cassandra, DynamoDB, Riak)
    - Key-Value Stores (e.g., Redis, Amazon S3)
    - Document Stores (e.g., MongoDB, CouchDB)
    - Eventually Consistent Storage Systems (e.g., Amazon SimpleDB, Cosmos DB)
- When to Choose BASE over ACID - BASE is ideal for applications where:
    - High availability and scalability are more important than strict consistency
    - Data inconsistency is acceptable for short periods (e.g., social media feeds, recommendation systems)
    - The system can tolerate eventual consistency delays (e.g., analytics platforms, logging systems)
    - Horizontal scaling is needed to support massive workloads (e.g., global applications with distributed users)

    - *Example*: Imagine a social media platform like Twitter. If a user posts a tweet, their followers might not all see it immediately due to replication lag. However, within seconds or minutes, all servers will eventually update and reflect the post. This trade-off allows for high availability and global scalability, even if it means minor delays in data consistency.

<u>Key-Value Stores (3 Key Principles)</u>

- A key-value store is a non-relational database that follows a simple key = value structure. Each piece of data is stored as a unique key (identifier) paired with its corresponding value, making these databases highly efficient and scalable.

1. <u>Simplicity</u>
    - Minimalist Data Model: The key-value model is extremely simple compared to traditional relational databases (RDBMS), where data is stored in structured tables with predefined schemas.
    - Flat Data Structure: Unlike SQL databases that require tables, rows, and columns, key-value stores use a flat structure, making CRUD (Create, Read, Update, Delete) operations straightforward.
    - Schema-less: There are no strict rules on data format, allowing flexible data storage (e.g., JSON, XML, text, binary, etc.).

- Ideal Use Cases: Caching (storing frequently accessed data for quick retrieval), Session Management (storing user sessions in web apps), Configuration Storage (storing app settings and feature flags)

2. <u>Speed</u>
    - Optimized for Fast Lookups:
        - Key-value stores use hash tables or similar data structures under the hood.
        - This allows them to retrieve a value by its key in O(1) time complexity → extremely fast.
        - Unlike relational databases that require complex indexing and query optimization, key-value stores provide instant access to values.
    - In-Memory Performance:
        - Many key-value stores (e.g., Redis, Memcached) run entirely in memory, making reads and writes blazing fast.
        - Eliminates the overhead of disk-based storage systems.
    - No Complex Queries or Joins:
        - Key-value stores do not support SQL-like queries, foreign keys, or joins.
        - Why? Because these operations slow down performance, making the system less efficient for its primary use cases.
        - If querying and relationships are needed, other NoSQL models (e.g., document stores or column-family stores) are a better fit.
    - Example: Redis Cache - Retrieving a cached webpage or API response from Redis can be 10–100x faster than querying a relational database
    - Ideal Use Cases: Real-time applications (e.g., leaderboards, message queues), Caching API responses (e.g., storing computed results from expensive operations), Rate limiting (e.g., tracking API usage quotas)
3. <u>Scalability</u>
    - Designed for Horizontal Scaling:
        - Unlike relational databases, which scale vertically (adding more CPU/RAM to a single server), key-value stores scale horizontally (by distributing data across multiple nodes).
        - This makes them ideal for handling large-scale workloads with millions of concurrent users.
    - Eventual Consistency:
        - In a distributed key-value store, replicas of data exist across multiple servers.
        - The system guarantees eventual consistency, meaning that all nodes will converge to the same value over time, but not necessarily instantly.
        - Some systems allow users to trade off consistency for higher availability (per the CAP theorem)
    - Partitioning (Sharding)
        - Data can be easily partitioned across multiple servers by hashing the key (e.g., using consistent hashing).

        - This ensures that reads and writes remain fast and balanced across nodes.
    - Example: Amazon DynamoDB - Uses partitioning and replication to distribute billions of key-value pairs across a global infrastructure
    - Ideal Use Cases: Distributed caching layers, Global-scale applications, Internet of Things (IoT) data storage

KV DS Use Cases - Data Science

1. EDA & Experimentation Results Store
    - Store intermediate results from data preprocessing, exploratory data analysis (EDA), or feature engineering.
    - Allows for quick lookups of previous experiment results without needing to recompute expensive transformations.
    - Useful for A/B testing: store experiment metadata and results without polluting the production database.
    - Benefits: Reduces computation time by caching results, Provides an easy way to resume interrupted experiments, Eliminates unnecessary writes to a structured RDBMS
2. Feature Store:
    - Store frequently accessed ML features for low-latency retrieval during model training and inference.
    - Supports real-time feature lookups, which is crucial for applications like fraud detection and recommendation systems.
    - Benefits: Faster access to features without recomputation, Reduces latency in model training & serving, Simplifies feature versioning
3. Model Monitoring & Performance Tracking
    - Store key performance metrics for deployed models.
    - Track real-time accuracy, precision, recall, drift detection, etc.
    - Enables real-time alerting if model performance degrades.
    - Enables real-time monitoring and alerting, Fast retrieval of historical model performance, Supports model version tracking

KV DS Use Cases - Software Engineering

1. Storing Session Information
    - Everything about a user's current session (e.g., authentication, preferences, browsing history) can be stored in a single PUT operation and retrieved instantly.
    - Ideal for stateless web applications, where user sessions must persist across multiple requests.
    - Benefits: Fast, single-call retrieval, Reduces reliance on traditional session storage, Works across distributed systems
2. User Profiles & Preferences
    - Personalized user experiences can be enabled with a single GET operation.
    - Stores UI preferences, language settings, themes, and notification settings.
    - Eliminates the need for complex joins in relational databases.

    - Benefits: Fast lookup for user-specific settings, Simplifies UI personalization, Enables instant application of preferences across sessions
3. Shopping Cart Data
    - A shopping cart is tied to a user and must be accessible across browsers, devices, and sessions.
    - Key-value stores provide fast retrieval and eliminate session timeouts issues.
    - Benefits: Ensures shopping carts persist across devices, Improves checkout experience with faster load times, Eliminates need for complex SQL queries
4. Caching Layer (Speeding Up Database Queries)
    - Key-value stores like Redis and Memcached act as a caching layer in front of a disk-based database.
    - Frequently accessed queries can be stored for quick retrieval, reducing database load.
    - Benefits: Reduces load on primary database, Improves application response times, Supports scalability

Redis DB (Remote Directory Server)

- An open-source, in-memory database known for high speed and low latency.
- Primarily a Key-Value (KV) Store, but supports multiple data models beyond key-value pairs.
- Frequently used for caching, real-time analytics, and session management.

Model	Description	Use Cases
Key-Value	Simple string-based KV store	Caching, session storage
Lists	Ordered collections of strings	Message queues, logs
Sets	Unordered collections with unique elements	Tagging, leaderboards
Sorted Sets	Sets with a score for sorting	Ranking systems, recommendation engines
Hashes	Key-value pairs inside a single key	Storing user profiles, objects
Bitmaps	Efficiently store bits	User activity tracking
HyperLogLogs	Approximate cardinality estimation	Unique visitor counting
Streams	Time-ordered logs of events	Event sourcing, real-time analytics
Geospatial	Store and query location-based data	Location services, geofencing
JSON	Native JSON document store	NoSQL-like document storage
Vectors	Store embeddings for similarity search	AI, recommendation engines

Redis Performance & Ranking
Redis consistently ranks among the top key-value stores due to:

- Ultra-low latency (sub-millisecond response times), High throughput (millions of operations per second), Support for horizontal scaling and clustering, Atomic operations for data consistency, Replication and persistence for reliability,

Ranking of KV Stores (DB-Engines.com, March 2025)

- Redis. Amazon DynamoDB, etcd, RocksDB, Memcached

Redis

- In-Memory Database – Primarily operates in RAM for ultra-fast data access.
- Durability Options:
    1. Snapshotting (RDB) – Saves a snapshot of the dataset to disk at specified intervals.
    2. Append-Only File (AOF) – Logs every write operation to disk, allowing roll-forward recovery in case of failure.
- Developed in 2009 – Written in C++, designed for speed and efficiency.
- High Performance – Can handle 100,000+ SET operations per second.
- Rich Command Set – Supports various operations beyond simple key-value storage.
- Limitations:
    - No Complex Queries – Does not support SQL-like querying.
    - No Secondary Indexes – Data can only be accessed by its primary key.

Redis Data Types

Keys:

- Typically strings, but can be any binary sequence (e.g., numbers, encoded objects, or raw bytes).
- Keys should be short and meaningful to optimize memory usage and retrieval speed.
- Naming convention best practices:
- Use colons (:) as namespace separators (e.g., user:1001:profile).
- Keep key lengths small to improve lookup performance.

Values:

Data Type	Description	Use Cases
Strings	Basic key-value storage (binary-safe, up to 512MB)	Caching, counters, session storage
Lists	Linked lists with fast push/pop operations	Message queues, task scheduling
Sets	Unordered collection of **unique** elements	Tagging, leaderboards, social networks

Sorted Sets	Similar to sets, but each element has a **score** for sorting	Ranking systems, recommendation engines
Hashes	Key-value pairs inside a **single key**	User profiles, object storage
Geospatial Data	Stores latitude/longitude coordinates with radius queries	Location-based services, geofencing

<u>Redis Data Type Breakdown</u>

1. Strings
    - The most basic data type in Redis.
    - Supports string manipulation, bitwise operations, and numeric operations.
2. Lists (Linked Lists)
    - Ordered collection of string elements, allowing fast insertions/removals from both ends.
    - Used for queues, logs, and messaging systems.
3. Sets
    - Unordered collection of unique elements (no duplicates).
    - Useful for tagging, leaderboards, and social media followers/following.
4. Sorted Sets (ZSets)
    - Similar to Sets, but each element has a numeric score that determines sorting order.
    - Used for ranking systems, priority queues, and real-time leaderboards.
5. Hashes
    - Key-value pairs stored within a single key, reducing memory overhead.
    - Great for storing user profiles, objects, and configurations.
6. Geospatial Data
    - Stores latitude/longitude coordinates, enabling location-based queries.
    - Used in geofencing, ride-sharing apps, and proximity searches.

<u>Why Use Redis Data Types?</u>

- Optimized for speed – Most operations are O(1) or O(log N).
- Memory-efficient – Compact storage for large-scale applications.
- Versatile – Supports multiple models beyond just key-value storage.