NoSQL & Key-Value Databases Notes

Distributed Databases and Concurrency Models

Pessimistic Concurrency (ACID)

- Focuses on "data safety" and assumes conflicts will occur
- Transactions protect themselves from other transactions
- Uses locking resources (both read and write locks) until a transaction is complete
- Similar to borrowing a book from a library - if you have it, no one else can use it
- Prioritizes data consistency and integrity

Optimistic Concurrency

- Does not obtain locks when reading or writing data
- Assumes conflicts are unlikely to occur
- Implements validation through timestamps and version numbers
- When a transaction is ready to commit, it checks if data has been modified by other transactions
- If a conflict is detected, the transaction can be rolled back and re-executed
- Works well in low-conflict systems (read-heavy workloads, analytical databases, backups)
- Less efficient in high-conflict systems due to the overhead of rolling back transactions

CAP Theorem

- States that distributed systems can have at most two of the following three properties:

  1. **Consistency**: Every user has an identical view of data at any moment
  2. **Availability**: System remains operational during failures
  3. **Partition Tolerance**: System can maintain operations when network partitions occur
- The three possible combinations:

  1. **CA (Consistency + Availability)**: Systems always respond with the latest data, but may not handle network partitions well
  2. **CP (Consistency + Partition Tolerance)**: If the system responds, it's with the latest data, but requests may be dropped during partitions
  3. **AP (Availability + Partition Tolerance)**: System always responds, but may not return the absolute latest data

- Traditional relational databases (PostgreSQL, MySQL) typically prioritize CA
- Many NoSQL systems (MongoDB, Redis) prioritize CP
- Some distributed systems (Cassandra, DynamoDB) prioritize AP

BASE - Alternative to ACID for Distributed Systems

- **Basically Available**: System guarantees availability but responses might be inconsistent or in a changing state
- **Soft State**: System state may change over time without input due to eventual consistency
- **Eventual Consistency**: System will eventually become consistent once all updates stop and are propagated

NoSQL Databases

- Term "NoSQL" originated in 1998 but has evolved to mean "Not Only SQL"
- Often refers to non-relational database systems
- Developed partly as a response to handling unstructured web-based data
- Allows for more flexible data models than traditional relational databases

Key-Value Databases

Characteristics

- **Simplicity**: Extremely simple data model compared to relational tables
- **Speed**: Usually deployed as in-memory databases
- **O(1) Operations**: Retrieving values by key is typically constant time
- **No Complex Queries**: No support for joins or complex queries
- **Scalability**: Horizontal scaling is simple by adding more nodes
- **Eventual Consistency**: In distributed environments, nodes will eventually converge on the same value

Use Cases

- **Data Science Applications**:
  - Storing EDA/experimentation results
  - Feature stores for low-latency ML model training and inference
  - Model monitoring metrics
- **Software Engineering Applications**:

- Session information storage
    - User profiles and preferences
    - Shopping cart data
    - Caching layer in front of slower disk-based databases

Redis (Remote Dictionary Server)

- Open-source, in-memory data structure store
- Primarily a key-value store but supports multiple data models
- Consistently ranked as the #1 key-value store according to db-engines.com
- Developed in 2009 in C++
- Can perform over 100,000 SET operations per second
- Supports durability through snapshots and append-only files
- Organized into 16 logical databases (numbered 0-15)

Redis Data Types

1. **Strings**
    - Simplest data type, maps a string key to a string value
    - Can store text, integers, serialized objects, binary arrays
    - Use cases: Caching, config settings, counters, rate limiting
    - Commands: SET, GET, INCR, DECR, INCRBY, DECRBY, SETNX
2. **Hashes**
    - Collection of field-value pairs under a single key
    - Maximum of 2^32-1 field-value pairs per hash
    - Use cases: Representing objects, session management, user tracking
    - Commands: HSET, HGET, HGETALL, HMGET, HINCRBY, HKEYS
3. **Lists**
    - Linked lists of string values
    - O(1) operations for insertion at front or end
    - Use cases: Stacks, queues, message passing, social media feeds, logging
    - Commands:
        - Queue operations: LPUSH, RPOP
        - Stack operations: LPUSH, LPOP
        - Other operations: LLEN, LRANGE
4. **Sets**
    - Unordered collections of unique strings
    - Support for powerful set operations (union, intersection, difference)
    - Use cases: Tracking unique items, access control, group membership

    - Commands: SADD, SISMEMBER, SCARD, SINTER, SDIFF, SREM, SRANDMEMBER
5. **JSON**
    - Full support for the JSON standard
    - Uses JSONPath syntax for navigation
    - Stored in binary as a tree structure for fast access to elements

Redis Setup

- Can be run locally via Docker
- Default port is 6379
- Requires security considerations in production:
    - Should not expose Redis port publicly
    - Should set authentication passwords
    - Should consider encryption for sensitive data