Module 3 – Moving Beyond the Relational Model

- Benefits of the relational model
    - Relational databases use SQL which is a widely adopted standard
    - ACID compliance ensures transactions are processed reliably
    - Relational models are well-suited for structured data with clear schemas
    - Can handle large data volumes (scalability)
    - There is extensive tooling and expertise available
- Ways a relational database can increase efficiency:
    - Indexing: Speeds up searches by organizing data.
    - Storage Control: Optimizes data layout for efficiency.
    - Column vs. Row Storage: Column-oriented storage benefits analytics; row-oriented storage benefits transactional systems.
    - Query Optimization: Uses execution plans to improve query speed.
    - Caching & Prefetching: Reduces repeated database calls.
    - Materialized Views: Stores query results for fast access.
    - Precompiled Stored Procedures: Speeds up frequently used operations.
    - Data Replication & Partitioning: Improves availability and load balancing.
- Transaction processing – a transaction is a sequence of one or more of the CRUD operations performed as a single, logical unit of work
    - Commit – the entire sequence succeeds (save all changes if successful)
    - Rollback / Abort – the entire sequence fails (revert changes if failure occurs)
    - Transactions help ensure:
        - Data integrity – prevents incomplete updates
        - Error recovery – enables system restoration
        - Concurrency control – manages simultaneous transactions
        - Reliable data storage – ensures durability
        - Simplified error handling – manifest failures systematically
- ACID properties
    - Atomicity – a transaction is all or nothing, it either completes fully or doesn't happen at all
    - Consistency – ensures database transitions from one valid state to another
        - Ex: if a transaction transfers money, it must debit one account and credit another without violating integrity constraints
    - Isolation – transactions execute independently, preventing interference
        - Three common issues arise without proper isolation:
            - Dirty Read – a transaction reads uncommitted changes from another transaction
            - Non-Repeatable Read – a second read during the same transaction returns different data because another transaction modified it

            - Phantom Read – a transaction sees new rows added by another transaction before it commits.
    - Durability – once a transaction is committed, it remains so, even after a system failure
- Example of a transaction (SQL Procedure) check the code in slides 10-11
    - Implement a money transfer – begin transaction, deduct from sender and credit receiver, check for sufficient funds, rolls back if insufficient funds exist, commits changes if all checks pass
- Challenges of a relational database
    - Schema evolution – sometimes schemas evolve over time and can be difficult to manage
    - Not all apps may need the full strength of ACID compliance
    - Expensive joins – complex queries can be computationally expensive
    - Semi-structured/unstructured data – JSON, XML, etc., don't fit well into rigid schemas
    - Scaling issues – horizontal scaling (across multiple machines) is harder than vertical scaling (upgrading a single machine)
    - Performance needs – some applications require lower latency than traditional relational databases can offer
- Scalability approaches
    - vertical scaling (scaling up) – increase power of a single machine, which is simpler but has physical and financial limits
    - horizontal scaling (scaling out) – uses multiple machines (distributed systems), which is more complex but allows handling larger loads. There are modern systems that make horizontal scaling less problematic
- Distributed systems – a system of independent computers appearing as one computer
    - Key features of a distributed system: computers operate concurrently, computers can fail independently, and there is no shared global clock (?)
    - Distributed databases
        - Data is stored on multiple nodes, often replicated.
        - MySQL, PostgreSQL support replication/sharding
        - NoSQL databases are often distributed, supporting one or both models
        - Key challenge – network failures are inevitable (network partitioning)
        - Distributed databases can be relational or non-relational
- The CAP theorem — a distributed database cannot guarantee all three simultaneously
    - Consistency – every read gets the most recent write or an error
    - Availability – every request gets a response (even if not the latest)

    - Partition tolerance – the system continues functioning despite network failures
- CAP theorem - DB view
    - Consistency*: Every user of the DB has an identical view of the data at any given instant (*note that the definition of consistency in CAP is different from that of ACID)
    - 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
    - Real world database trade-offs
        - CP (Consistency + Partition Tolerance): Prioritizes data correctness but may sacrifice availability during failures, includes MongoDB, HBase, Redis
        - AP (Availability + Partition Tolerance): Always responds but may return outdated data, includes CouchDB, Cassandra, DynamoDB
        - CA (Consistency + Availability): Works only if network partitions never happen (which is unrealistic), includes RDBMS, PostgreSQL, MySQL
    - If a system must always respond, it cannot guarantee consistency if partitions exist, a trade-off is always necessary
- Relational databases offer ACID properties, structured data handling, and strong performance tuning
- Distributed databases help with scalability but face CAP theorem limitations
- Choosing the right database depends on application needs (performance, consistency, scalability)

Module 4 – Replicating Data

- Benefits of replicating (distributing) data
    - Scalability & High Throughput – handles growing data volumes or increasing read/write loads beyond a single machine's capacity
    - Fault Tolerance & High Availability – ensures application uptime even if some machines fail
    - Latency Reduction – improves response times by placing data closer to users worldwide
- Challenges of distributed data
    - Consistency issues – updates must propagate across multiple nodes
    - Increased application complexity – applications often need to handle distributed read/ writes
- Scaling strategies
    - Vertical Scaling (Scaling Up)

        - Shared memory architecture – a single, centralized server with fault tolerance via hot-swappable components
        - Shared disk architecture – multiple machines share a fast storage system, but write contention limits scalability
    - Horizontal Scaling (Scaling Out)
        - Shared nothing architecture – each node has its own CPU, memory, and disk
        - Nodes communicate via a network and are often geographically distributed
        - Uses commodity hardware (cheaper than scaling up)
- Replication vs. Partitioning
    - Replication duplicates data across multiple nodes
    - Partitioning (sharding): divides data into subsets across different nodes
- Common replication strategies
    - Distributed databases typically use one of these replication models:
        - Single Leader Model (most common)
            - All writes go to a single leader.
            - Leader sends replication info to followers.
            - Clients can read from leader or followers.
        - Multiple Leader Model
            - Multiple nodes can accept writes.
            - Each leader synchronizes changes across the system.
            - More complex, but better for distributed workloads.
        - Leaderless Model
            - No designated leader
            - All nodes can accept reads and writes.
            - Data consistency is maintained via quorum-based reads and writes.
- Leader based replication is most common and widely used in both relational and NoSQL databases
    - Relational: MySQL, PostgreSQL, Oracle, SQL Server.
    - NoSQL: MongoDB, RethinkDB, LinkedIn's Espresso.
    - Messaging Systems: Kafka, RabbitMQ.
- How is replication data transmitted to followers? Methods below
    - Statement-Based Replication
        - Sends SQL statements (INSERT, UPDATE, DELETE) to replicas.
        - Issues: error prone due to non-deterministic functions (NOW()), triggers, and concurrency problems.
    - Write-Ahead Log (WAL) Replication
        - Logs every byte-level change in the database.

        - Issue: Requires all nodes to use the same storage engine.
    - Logical (Row-Based) Log Replication
        - Stores row changes (before/after state) in a transaction log.
        - Benefit: Decoupled from storage engine, easier to interpret
    - Trigger-Based Replication
        - Logs changes into a separate table when triggers fire.
        - Benefit: Customizable for application needs.
        - Issue: Can be error-prone
- Synchronous vs. asynchronous replication
    - Synchronous Replication
        - Leader waits for followers to confirm before committing changes.
        - Guarantees strong consistency but increases latency.
    - Asynchronous Replication
        - Leader does not wait for confirmation from followers.
        - Improves availability and performance but can lead to data inconsistencies
- What happens when the leader fails? Challenges below
    - Electing a new leader
        - Use a consensus strategy (e.g., node with the most recent updates).
        - Use a dedicated controller node to assign leadership.
    - Client redirection
        - Update clients to write to the new leader.
    - Handling data loss (if using asynchronous replication):
        - New leader may not have all writes.
        - Options:
            - Recover missing writes.
            - Accept some data loss.
    - Avoiding "Split Brain" scenarios
        - If the old leader recovers, there must be a way to prevent multiple leaders from conflicting.
    - Failure detection
        - Determining the optimal timeout before electing a new leader
- Replication lag – the time it takes for updates on the leader to be reflected on all the followers
    - Synchronous replication: Replication lag causes writes to be slower and the system to be more brittle as num followers increases.
    - Asynchronous replication: We maintain availability but at the cost of delayed or eventual consistency. This delay is called the inconsistency window.
- Read-After-Write Consistency
    - Scenario: A user adds a comment on Reddit but does not see it immediately.

    - Solution 1: Always read the most recent data from the leader.
    - Solution 2: Temporarily switch to reading from the leader for "recently updated" data.
    - Challenge:
        - Followers are geographically close to users, but now we must route reads to a distant leader for consistency.
    - Implementation (as in scenario):
        - Method 1: Modifiable data (from the client's perspective) is always read from the leader.
        - Method 2: Dynamically switch to reading from leader for "recently updated" data.
            - For example, have a policy that all requests within one minute of last update come from leader.
- Monotonic read consistency
    - Issue – if a user reads data from different followers, they might see older data after newer data. Anomalies occur when a user reads values out of order from multiple followers.
    - Solution – ensure that once a user reads newer data, they will never read older data. Monotonic read consistency ensures that when a user makes multiple reads, they will not read older data after previously reading newer data.
- Consistent Prefix Reads
    - Issue – when different partitions replicate at different rates, data may be read out of order. There is no global write consistency
    - Solution – Consistent Prefix Reads Guarantee ensures that data is always read in the order it was written.
- Summary
    - Data replication improves scalability, fault tolerance, and latency but introduces consistency challenges.
    - Common replication models:
        - Single Leader (Master-Slave)
        - Multiple Leader
        - Leaderless
    - Replication methods vary in complexity and performance trade-offs.
    - Synchronous vs. Asynchronous Replication:
        - Synchronous ensures consistency but increases latency.
        - Asynchronous is faster but may lead to inconsistencies.
    - Leader failure introduces major challenges, requiring leader election, data reconciliation, and client redirection.

    - Replication lag affects read consistency, and various strategies like Read-After-Write and Monotonic Reads help mitigate issues
    - Consistent Prefix Reads ensure ordered data retrieval across partitions.

Module 5 – NoSQL & KV DBs

- Database Concurrency: Handling Multiple Transactions
    - ACID Transactions & Pessimistic Concurrency
        - ACID (Atomicity, Consistency, Isolation, Durability): Ensures transactions are processed reliably.
        - Uses locks to prevent conflicts.
            - Example: Borrowing a library book. If you have it, no one else can until you return it.
    - Optimistic Concurrency
        - Assumes conflicts are rare, so transactions don't lock data.
        - Instead, they check at the end of a transaction whether another process has modified the data.
        - Works well in systems with many reads but few writes, like backups or analytical databases
- Origin of noSQL databases was meant for non-SQL databases (in 1998) but now it means Not Only SQL, which are flexible, scalable databases that don't necessarily use SQL. they are designed to handle unstructured, web-scale data efficiently
- CAP Theorem Review (Trade-offs in Distributed Databases)
    - You can only have two out of three:
        - Consistency: Every user sees the same, latest data.
        - Availability: The system is always up and running.
        - Partition Tolerance: The system keeps working even if some network connections fail.
    - Real-World CAP Trade-offs
        - Consistency + Availability: Always provides fresh data but struggles during network failures.
        - Consistency + Partition Tolerance: Ensures accurate data but may reject requests when failures occur.
        - Availability + Partition Tolerance: Keeps running even during failures but might return slightly outdated data.
- Alternative to ACID: BASE Model (Used in NoSQL) – instead of strict consistency, NoSQL databases use BASE:
    - Basically Available: System remains operational, but some data may be temporarily missing.
    - Soft State: Data updates over time, even without new inputs.

    - Eventual Consistency: Data will become consistent eventually, but not always immediately.
- Types of NoSQL Databases
    - Key-Value Stores (Simplest, fastest)
        - Stores data as key = value pairs.
        - Examples: Redis, DynamoDB
    - Document Stores
        - Stores JSON-like documents.
        - Examples: MongoDB, CouchDB
    - Column-Family Stores
        - Optimized for large-scale data storage.
        - Examples: Cassandra, HBase
    - Graph Databases
        - Used for relationships between data points.
        - Examples: Neo4j, ArangoDB
- Key-Value (KV) Databases Explained
    - Simpler than relational databases (No complex tables).
    - Super fast because they use in-memory storage and hash tables.
    - No complex queries—just quick data retrieval by key.
    - Key-Value Store Benefits
        - Simplicity: Easy to use.
        - Speed: Fetching a value by key is super fast (O(1) time complexity).
        - Scalability: Horizontally scalable—just add more servers
- Common use cases for Key-Value Databases
    - Experiment Data Storage (store test results, machine learning data, store intermediate results from data preprocessing and EDA store experiment or testing (A/B) results w/o prod db).
    - Feature Store (store precomputed ML model features, store frequently accessed feature, low-latency retrieval for model training and prediction)
    - Model monitoring (store key metrics about the performance of the model, for example in real-time inferencing)
    - Session Management (store user sessions for quick access, everything about the current session can be stored via a single PUT or POST and retrieved with a single GET and very fast).
    - User Profiles & Preferences (quick lookup of user settings, User info could be obtained with a single GET operation… language, TZ, product or UI preferences).
    - Shopping Carts (fast storage of items in a cart, Cart data is tied to the user needs to be available across browsers, machines, sessions).

    - Caching (speed up web requests by temporarily storing frequently accessed data)
- Introduction to Redis (Popular Key-Value Database) and sometimes called a data structure store
    - Open-source, in-memory database (fast and lightweight).
    - Can be used as:
        - Key-Value store
        - Graph Database
        - Full-text search
        - Time-series database
    - Fast operations: Handles 100,000+ writes per second!
    - Persistence options:
        - Takes snapshots at intervals.
        - Maintains an append-only log for crash recovery
    - Does not handle complex data and no secondary indexes, only supports lookup by key
- Redis Data Types
    - Keys: Usually strings, but can be any binary sequence.
    - Values:
        - Strings (Basic text or binary values).
        - Lists (Ordered collections, useful for queues, chat messages).
        - Sets (Unique, unordered elements).
        - Sorted Sets (Like sets but with a ranking system).
        - Hashes (Field-value pairs, similar to objects).
        - Geospatial Data (Store and query location-based data)
- Setting Up Redis
    - Using Docker
        - Download and run the Redis image.
        - Expose port 6379 (default Redis port).
        - Avoid exposing it publicly for security reasons.
    - Using DataGrip
        - File > New > Data Source > Redis
        - Connect using port 6379.
        - Test the connection.-
- Basic Redis Commands - command and description:
    - SET key value — Store a key-value pair
    - GET key — Retrieve the value for a key
    - DEL key — Delete a key
    - EXISTS key — Check if a key exists
    - EXISTS key — Check if a key exists (returns 1 if exists, 0 otherwise)

    - EXPIRE key seconds — Set a key to expire after a set time
    - TTL key — Check how much time is left before a key expires
    - FLUSHDB — Deletes all keys in the current Redis database
    - FLUSHALL — Deletes all keys in all Redis databases
    - Example:
        - SET username "JohnDoe"
        - GET username # Output: "JohnDoe"
        - EXPIRE username 60 # Key expires in 60 seconds
        - TTL username # Check remaining time before expiration
- Working with numbers in redis
    - INCR key — Increase an integer value by 1
    - DECR key — Decrease an integer value by 1
    - INCRBY key amount — Increase by a specific amount
    - DECRBY key amount — Decrease by a specific amount
    - Example:
        - SET counter 10
        - INCR counter # counter = 11
        - INCRBY counter 5 # counter = 16
        - DECR counter # counter = 15
        - DECRBY counter 3 # counter = 12
- Working with Hashes (Key-Value Store for Objects) - a 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)
    - Basic hash commands:
        - HSET key field value — Set a field inside a hash
        - HGET key field — Retrieve a specific field's value
        - HGETALL key — Get all fields and values in a hash
        - HMGET key field1 field2 — Get multiple field values
        - HINCRBY key field amount — increase a numerical field by a certain amount
        - Example:
            - HSET user:1 name "Alice" age 30 city "Boston"
            - HGET user:1 name # Output: "Alice"

            - HGETALL user:1 # Output: name: "Alice", age: 30, city: "Boston"
            - HINCRBY user:1 age 2 # Age is now 32
- Linked lists crash course
    - Sequential data structure of linked nodes (instead of contiguously allocated memory)
    - Each node points to the next element of the list (except the last one - points to nil/null)
    - O(1) to insert new value at front or insert new value at end
- List Data Type (Queues & Stacks) - value of KV Pair is linked lists of string values
    - A List in Redis is like an array or queue. It stores ordered elements
    - Use Cases
        - Implementing message queues (e.g., order processing)
        - queue management & message passing queues (producer/consumer model)
        - logging systems (easy to keep in chronological order)
        - build social media streams/feeds
        - batch processing by queueing up a set of tasks to be executed sequentially at a later time
        - Managing task lists (e.g., batch jobs).
        - Storing chat messages (e.g., live messaging apps).
    - Queue (FIFO) Commands - command and description:
        - LPUSH queue value — Add to the start of the queue
        - RPOP queue — Remove from the end of the queue
    - Stack (LIFO) Commands - command and description
        - LPUSH stack value — Add to the top of the stack
        - LPOP stack — Remove from the top of the stack
    - A List in Redis is like an array or queue. It stores ordered elements.
    - Commands for lists
        - LPUSH key value — Add to the front of the list (left push)
        - RPUSH key value — Add to the end of the list (right push)
        - LPOP key — Remove and return the first element (FIFO)
        - RPOP key — Remove and return the last element (LIFO)
        - LLEN key — Get the length of the list
        - LRANGE key start stop — Get a subset of the list
    - Example:
        - LPUSH todo "Task 1"
        - LPUSH todo "Task 2"
        - RPUSH todo "Task 3"
        - LRANGE todo 0 -1 # Output: ["Task 2", "Task 1", "Task 3"]

        - LPOP todo # Output: "Task 2" (removed)
- Sets (Unique Collections)
    - Use Cases
        - Tracking unique website visitors.
        - Managing user groups (e.g., students enrolled in a course).
        - Access control lists (who has permission to see something).
    - Set Commands - command and description
        - SADD key value — Add an item to the set
        - SREM key value — Remove an item from the set
        - SISMEMBER key value — Check if an item exists in the set
        - SCARD key — Get the number of items in the set
        - SMEMBERS key — Get all items in the set
        - Example:
            - SADD students "Alice"
            - SADD students "Bob"
            - SADD students "Charlie"
            - SADD students "Alice" # Will not be added again (duplicate)
            - SMEMBERS students # Output: ["Alice", "Bob", "Charlie"]
            - SISMEMBER students "Bob" # Output: 1 (true)
            - SCARD students # Output: 3 (size of set)
- Sorted sets (ranking system): A Sorted Set is like a regular Set, but each item has a score. This is useful for leaderboards, rankings, and priority queues.
    - Commands for sorted sets
        - ZADD key score value — Add a value with a ranking (score)
        - ZRANGE key start stop WITHSCORES — Get values in order (lowest to highest)
        - ZREVRANGE key start stop WITHSCORES — Get values in reverse order (highest to lowest)
        - ZSCORE key value — Get the score of a specific value
    - Example:
        - ZADD leaderboard 100 "Alice"
        - ZADD leaderboard 200 "Bob"
        - ZADD leaderboard 150 "Charlie"
        - ZRANGE leaderboard 0 -1 WITHSCORES # Output: ["Alice", 100, "Charlie", 150, "Bob", 200]
        - ZREVRANGE leaderboard 0 -1 WITHSCORES # Output: ["Bob", 200, "Charlie", 150, "Alice", 100]
- JSON data in redis allows you to stored structured data
    - Commands for json in redis
        - JSON.SET key . value — Store a JSON object

        - JSON.GET key — Retrieve a JSON object
        - JSON.GET key path — Retrieve a specific field in JSON
    - Example:
        - JSON.SET user:1 . '{"name": "Alice", "age": 30, "city": "Boston"}'
        - JSON.GET user:1 # Output: {"name": "Alice", "age": 30, "city": "Boston"}
        - JSON.GET user:1 .name # Output: "Alice"
- Summary:
    - Hashes store structured objects like user profiles.
    - Lists work as queues and stacks.
    - Sets store unique items like friend lists.
    - Sorted Sets keep ranked items like leaderboards.
    - JSON support allows for structured storage.

Module 6 – Redis and Python

- After installation of redis, connect to a redis server with the following code
    - import redis
    - redis_client = redis.Redis(
    - host='localhost', # Or 127.0.0.1 if using Docker
    - port=6379, # Default Redis port
    - db=2, # Redis has 16 databases (0-15)
    - decode_responses=True # Converts bytes to strings automatically
    -)
- Basic string operations in redis with python
    - # Set a key-value pair
    - redis_client.set('clickCount:/abc', 0)
    -
    - # Get value of a key
    - val = redis_client.get('clickCount:/abc')
    -
    - # Increment the value
    - redis_client.incr('clickCount:/abc')
    -
    - # Get updated value
    - ret_val = redis_client.get('clickCount:/abc')
    - print(f'Click count = {ret_val}')
- Multiple values at once
    - # Set multiple key-value pairs at once
    - redis_client.mset({
    - 'key1': 'val1',

    - 'key2': 'val2',
    - 'key3': 'val3'
    - })
    -
    - # Get multiple values at once
    - print(redis_client.mget('key1', 'key2', 'key3'))
    - # Output: ['val1', 'val2', 'val3']
- List commands in redis with python
    - # Create a list and add elements
    - redis_client.rpush('names', 'mark', 'sam', 'nick')
    -
    - # Get all values from the list
    - print(redis_client.lrange('names', 0, -1))
    - # Output: ['mark', 'sam', 'nick']
- More list commands
    - rpush(key, value1, value2, ...) — Add values to the end of the list
    - lpush(key, value1, value2, ...) — Add values to the start of the list
    - lpop(key) — Remove and return the first element
    - rpop(key) — Remove and return the last element
    - lrange(key, start, stop) — Get values from a list (similar to slicing in Python)
    - llen(key) — Get the number of elements in the list
- Hash commands in redis with python
    - # Store a user profile in a Redis hash
    - redis_client.hset('user-session:123', mapping={
    - 'first': 'Sam',
    - 'last': 'Uelle',
    - 'company': 'Redis',
    - 'age': 30
    - })
    -
    - # Retrieve all fields
    - print(redis_client.hgetall('user-session:123'))
    - # Output: {'first': 'Sam', 'last': 'Uelle', 'company': 'Redis', 'age': '30'}
- More hash commands
    - hset(key, field, value) — Set a field in a hash
    - hget(key, field) — Get a field's value
    - hgetall(key) — Get all fields and values
    - hdel(key, field) — Delete a field
    - hlen(key) — Get the number of fields

- Using redis pipelines for efficiency; normally, sending multiple commands to Redis one at a time can be slow. Pipelines allow batch processing, reducing network overhead and are ideal for bulk inserts or batch operations
    - r = redis.Redis(decode_responses=True)
    - pipe = r.pipeline()
    -
    - # Batch insert multiple keys at once
    - for i in range(5):
    - pipe.set(f"seat:{i}", f"#{i}")
    -
    - # Execute all at once
    - set_5_result = pipe.execute()
    - print(set_5_result) # Output: [True, True, True, True, True]
    -
    - # Retrieve multiple keys in a batch
    - pipe = r.pipeline()
    - get_3_result = pipe.get("seat:0").get("seat:3").get("seat:4").execute()
    - print(get_3_result) # Output: ['#0', '#3', '#4']
- Redis as a feature store for machine learning
    - Example: storing user engagement data for machine learning
        - redis_client.hset('user:1001', mapping={
        - 'click_rate': 0.75,
        - 'session_time': 180, # seconds
        - 'purchase_likelihood': 0.92
        - })
        -
        - # Fast retrieval during ML inference
        - user_data = redis_client.hgetall('user:1001')
        - print(user_data)
        - # Output: {'click_rate': '0.75', 'session_time': '180', 'purchase_likelihood': '0.92'}
    - Why use redis for machine learning?
        - Fast Reads/Writes: In-memory storage ensures low latency.
        - Scalability: Redis can store millions of ML feature sets.
        - Simple API: Works well with real-time AI models