NoSQL & Key-Value Databases Overview

Distributed Databases and ACID
Distributed databases often implement concurrency control to maintain data integrity and consistency.

Pessimistic Concurrency
- **Definition**: Transactions lock data to prevent other transactions from conflicting, assuming conflicts will occur.
- **Purpose**: Ensures data safety.
- **Implementation**: Locks resources until transaction completion (both read/write locks).
- **Analogy**: Borrowing a book from a library; if one person has it, others cannot.

Optimistic Concurrency
- **Definition**: Assumes conflicts rarely occur, transactions proceed without locking data.
- **Implementation**: Adds timestamp/version columns, verifies data consistency at transaction end.
- **Applicability**:
  - **Low-conflict systems** (analytics databases, backup systems) use optimistic concurrency for higher efficiency.
  - **High-conflict systems** are better with pessimistic concurrency due to frequent rollbacks.

Introduction to NoSQL
- **Historical Note**: The term "NoSQL" initially described relational databases without SQL, coined in 1998 by Carlo Strozzi.
- **Modern Interpretation**: Typically means "Not Only SQL" or non-relational databases.
- **Usage Context**: Designed for processing unstructured web-based data efficiently.

CAP Theorem
The CAP theorem highlights a fundamental limitation in distributed database systems, stating that only two out of three properties can simultaneously be achieved:
- **Consistency**: Every user sees identical data.
- **Availability**: Database remains operational despite failures.
- **Partition Tolerance**: Operates despite network partitions.

CAP Combinations:
- **Consistency + Availability (CA)**: Responses always reflect the most recent data but may fail during network partitions.

- **Consistency + Partition Tolerance (CP)**: Guarantees latest data or no response at all.
- **Availability + Partition Tolerance (AP)**: Guarantees responses but potentially outdated data.

ACID Alternative: BASE (for distributed systems)
- **Basically Available**:
  - Data availability prioritized over strict consistency.
  - Occasional inconsistencies or "unreliable" states permitted.
- **Soft State**:
  - System state may change without direct input, driven by eventual consistency.
  - Data stores aren't immediately write-consistent.
- **Eventual Consistency**:
  - All system replicas will eventually converge, even if temporary inconsistencies occur.

Key-Value Databases
- **Data Model**: Simple key-value pairs.
- **Design principles**:
  - **Simplicity**: Straightforward CRUD operations.
  - **Speed**: Often O(1) retrieval due to hash-based implementations.
  - **Scalability**: Easy horizontal scaling and eventual consistency guarantee.

Key-Value Database Use Cases:
- **Data Science**: Experiment results, feature stores, and model monitoring metrics.
- **Software Engineering**: Session management, user profiles, shopping carts, caching layers.

Redis Database
- **Definition**: Redis (Remote Directory Server) is an open-source, in-memory database and key-value store.
- **Performance**: Over 100,000 operations per second possible.
- **Capabilities**: Supports durability via snapshots or append-only logs, developed initially in C++ (2009).
- **Limitations**: Only supports key-based lookups, no complex queries or secondary indexing.

Redis Data Types:
- **Keys**: Usually strings or binary sequences.
- **Values**: Strings, lists, sets, sorted sets, hashes, and geospatial data.

Redis Interaction
- **Databases**: Offers 16 default databases, numbered 0-15.

- **Commands**: `SET`, `GET`, `DEL`, `EXISTS`, `KEYS`, `SELECT`.
- **Atomic Increment/Decrement**: `INCR`, `INCRBY`, `DECR`, `DECRBY`.
- **Conditional Sets**: `SETNX` only sets values if a key does not exist.

Redis Data Structures and Commands

Hashes
- Collection of field-value pairs for object-like structures.
- Commands: `HSET`, `HGET`, `HGETALL`, `HMGET`, `HINCRBY`.
- Use cases: Session tracking, event tracking, structured user data.

Lists
- Linked lists storing ordered string values.
- Operations: Implement stacks (`LPUSH`, `LPOP`) and queues (`LPUSH`, `RPOP`).
- Other operations: `LLEN`, `LRANGE`.
- Use cases: Queues, message passing, logging, batch processing.

Sets
- Unordered collection of unique elements.
- Commands: `SADD`, `SISMEMBER`, `SCARD`, `SINTER`, `SDIFF`, `SREM`, `SRANDMEMBER`.
- Use cases: Tracking unique items, access control lists, social networking structures.

Redis JSON Type
- Fully supports JSON standards using JSONPath.
- Fast navigation and storage via internal binary tree structures.
- Efficient access to sub-elements.

Redis in Docker
- Installation and setup easily via Docker Desktop.
- Default Redis port is **6379**.
- Important Security Note: Avoid exposing Redis port directly to prevent vulnerabilities, especially in production environments.

Connecting to Redis from Tools
- Example: DataGrip.
- Steps: Create new Redis Data Source, ensure port **6379** is correct, test connection.

This expanded explanation details each component from your slides, enhancing clarity and providing thorough coverage of NoSQL databases, key-value stores, Redis, concurrency strategies, and the CAP theorem.