### **Data Replication Overview**

* **Why Replicate Data?**
  + **Scalability & High Throughput:** Handle growing data and read/write loads.
  + **Fault Tolerance & High Availability:** Maintain application uptime despite failures.
  + **Latency Reduction:** Serve users in different locations with low response times.
* **Challenges of Data Replication:**
  + **Consistency:** Updates must be propagated across distributed nodes.
  + **Application Complexity:** Distributed systems require additional logic for reads/writes.

### **Scaling Architectures**

* **Vertical Scaling:**
  + **Shared Memory:** Centralized server with some fault tolerance.
  + **Shared Disk:** Machines share a fast network but face scalability limits with high writes.
  + **Cost Considerations:** Cloud pricing for high-scale vertical machines can be expensive.
* **Horizontal Scaling (Shared Nothing):**
  + Each node has its own CPU, memory, and disk.
  + Coordination happens via application layer over the network.
  + Allows geographical distribution and use of commodity hardware.

### **Replication vs. Partitioning**

* **Replication:** Duplicates full datasets across nodes.
* **Partitioning:** Splits data, each partition storing a subset of the whole.

### **Replication Strategies**

1. **Single Leader Model:**
   * **Writes go to a single leader**, which replicates changes to followers.
   * Followers process instructions from the leader and serve read queries.
   * **Examples:** MySQL, PostgreSQL, Oracle, MongoDB, Kafka.
2. **Multiple Leader Model:**
   * Multiple leaders accept writes, requiring conflict resolution.
   * Used when applications span multiple data centers.
3. **Leaderless Model:**
   * Any node can accept writes, and data propagates via consensus mechanisms.
   * **Examples:** DynamoDB, Cassandra.

### **Replication Methods**

* **Statement-Based:** Sends raw SQL statements to replicas (error-prone).
* **Write-Ahead Log (WAL):** Logs changes at a byte level (storage-engine dependent).
* **Logical (Row-Based) Logs:** Tracks row-level changes, decoupled from storage engine.
* **Trigger-Based Replication:** Uses triggers to log changes but adds complexity.

### **Synchronous vs. Asynchronous Replication**

* **Synchronous:** Leader waits for follower confirmation (strong consistency but slower).
* **Asynchronous:** Leader does not wait (higher availability but potential data loss).

### **Handling Leader Failure**

* **Challenges:**
  + How to elect a new leader?
  + How to configure clients to switch leaders?
  + Avoiding "split-brain" situations where multiple leaders emerge.
* **Consensus Strategies:**
  + Choose a new leader based on replication state.
  + Use a controller node to manage failover.

### **Replication Lag & Consistency Guarantees**

* **Replication Lag:** Delay in propagating writes to all followers.
* **Read-After-Write Consistency:**
  + Clients should read **recently modified data** from the leader.
  + Can dynamically switch read behavior for "recently updated" data.
* **Monotonic Read Consistency:**
  + Ensures users **never see older data** after reading newer data.
* **Consistent Prefix Reads:**
  + Guarantees that writes appear **in the correct order** across nodes.

These notes should provide a solid foundation for your RAG-based study tool. Let me know if you need any refinements!

4o

##### **You said:**

05 - NoSQL Intro + KV DBs (1).pdf

PDF

please compile notes from this slideshow that I could add to an index for a rag that I will use to complete my exam for this class

###### **ChatGPT said:**

Here are the compiled notes from your **NoSQL & Key-Value Databases** slideshow, formatted for easy indexing in your RAG system:

### **ACID vs. BASE in Distributed Databases**

* **ACID (Pessimistic Concurrency)**
  + Focuses on **data safety** using locks to prevent conflicts.
  + Example: A **write lock** in a database is like borrowing a library book—others cannot modify it until it’s returned.
* **Optimistic Concurrency**
  + **No locks on data**; assumes conflicts are rare.
  + Uses **timestamps & version numbers** to detect conflicts before committing.
  + Works well for **low-conflict, read-heavy systems** (e.g., backups, analytics).

### **NoSQL Overview**

* **Origins**:
  + First used in **1998** for a relational DB without SQL.
  + Modern meaning: **"Not Only SQL"** (but not necessarily non-relational).
  + Created to handle **web-based unstructured data** at scale.

### **CAP Theorem (Consistency, Availability, Partition Tolerance)**

* **You can only achieve 2 out of 3:**
  + **Consistency + Availability (CA):** Latest data, but cannot handle network partitions.
  + **Consistency + Partition Tolerance (CP):** Latest data, but might drop requests.
  + **Availability + Partition Tolerance (AP):** Always available but might return outdated data.

### **BASE: The ACID Alternative for NoSQL**

* **Basically Available:** System is responsive but may return stale or missing data.
* **Soft State:** Data updates may change state over time (even without writes).
* **Eventual Consistency:** System eventually becomes consistent **after writes settle**.

### **NoSQL Database Categories**

1. **Key-Value Stores** (e.g., Redis, DynamoDB)
2. **Document Stores** (e.g., MongoDB, CouchDB)
3. **Column Stores** (e.g., Cassandra, Bigtable)
4. **Graph Databases** (e.g., Neo4j, ArangoDB)

### **Key-Value Databases**

* **Data Model:**
  + Stores **key-value pairs**, like a Python dictionary.
  + **CRUD operations** are extremely fast.
  + No complex queries or joins.
* **Advantages:**
  + **Speed**: Often in-memory, with O(1) retrieval via hash tables.
  + **Scalability**: Horizontal scaling ensures high availability.
  + **Eventual Consistency**: In distributed systems, all nodes **eventually converge** on the same value.

### **Use Cases for Key-Value Databases**

* **Data Science / ML**
  + **EDA & Experimentation:** Store preprocessing and experiment results.
  + **Feature Store:** Save frequently accessed features for ML models.
  + **Model Monitoring:** Track performance metrics in real-time.
* **Software Engineering**
  + **Session Storage:** Fast retrieval of user session data.
  + **User Profiles & Preferences:** Store user settings with single GET operations.
  + **Shopping Carts:** Maintain persistent cart data across sessions.
  + **Caching Layer:** Reduce DB load by caching frequent queries.

### **Redis: The Most Popular Key-Value Store**

* **Key Features:**
  + Open-source, **in-memory**, and **extremely fast** (>100,000 ops/sec).
  + Supports **multiple data structures** (Lists, Sets, Hashes, JSON, etc.).
  + Provides **snapshot-based** and **append-only** persistence methods.
  + **Downside:** No complex queries, joins, or secondary indexes.
* **Redis Data Types:**
  + **Strings** (basic key-value pairs).
  + **Lists** (linked lists, stacks, queues).
  + **Sets** (unordered collections, supports intersections/differences).
  + **Sorted Sets** (ordered collections).
  + **Hashes** (store structured objects).
  + **JSON** (supports tree-structured querying).

### **Basic Redis Commands**

**Setting & Retrieving Values:** redis  
SET user:1 "John Doe"

GET user:1

EXISTS user:1

DEL user:1

**Incrementing & Decrementing:** redis  
INCR counter

INCRBY counter 10

DECR counter

**Hashes (Structured Objects):** redis  
HSET bike:1 model "Demios" brand "Ergonom" price 1971

HGET bike:1 model

HGETALL bike:1

**Lists (Queues & Stacks):** redis  
LPUSH queue "task1"

RPUSH queue "task2"

LPOP queue

RPOP queue

**Sets (Unique Values & Relations):** redis  
SADD ds4300 "Mark"

SADD ds4300 "Sam"

SISMEMBER ds4300 "Mark"

* SCARD ds4300