

Comprehensive Lecture Notes: Redis

Architecture, Data Models, and Distributed Systems Context

Based on Course Material & Technical Enrichment

Contents

1	Introduction to Redis	2
1.1	Definition and Origins	2
1.2	Core Philosophy	2
2	Logical Data Model	2
2.1	1. Primitives: Strings	2
2.2	2. Containers (Data Structures)	2
2.2.1	Hashes	2
2.2.2	Lists	3
2.2.3	Sets	3
2.2.4	Sorted Sets (ZSets)	3
3	System Architecture	3
3.1	Single-Threaded Event Loop	3
3.2	Memory Management	4
4	Data Persistence	4
4.1	RDB (Redis Database File) - Snapshots	4
4.2	AOF (Append Only File)	4
5	Replication and Distributed Consistency	4
5.1	Master-Slave Architecture	4
5.2	CAP Theorem Analysis	5
6	Transactions	5
6.1	Commands	5
7	Real-World Case Studies	5
7.1	1. The Guardian (Document Review System)	5
7.2	2. StackOverflow (Multi-Level Caching)	5
8	Example: Shopping Cart Design	6
8.1	Relational Model (SQL)	6
8.2	Redis Model (NoSQL)	6

1 Introduction to Redis

1.1 Definition and Origins

Redis stands for **REmote DIctionary Server**. It is an open-source, in-memory data structure store used as a database, cache, and message broker.

- **Origin:** The project was started in early 2009 by Salvatore Sanfilippo (antirez).
- **Motivation:** Sanfilippo was developing a real-time web analytics platform called *LLOOGG*. He discovered that traditional relational databases (like MySQL) could not handle the write throughput required for real-time analytics efficiently.
- **Evolution:** In March 2010, VMware hired Sanfilippo to work on Redis full-time. It remains BSD licensed.

1.2 Core Philosophy

Redis is often referred to as a "**Data Structure Server**". Unlike a simple key-value store (like Memcached) where values are just opaque blobs of string, Redis supports complex data structures (lists, sets, hashes) as values. This allows developers to run atomic operations on these structures server-side, reducing network overhead and application complexity.

2 Logical Data Model

Redis operates as a key-value store where keys are strings (printable ASCII is standard), and values can be one of several data types.

2.1 1. Primitives: Strings

The most basic value type.

- **Capacity:** A string value can be up to 512 MB.
- **Usage:** Caching HTML fragments, user sessions, or simple counters.
- **Complexity:** Basic operations like `GET` and `SET` are $O(1)$.

2.2 2. Containers (Data Structures)

Redis supports collections of strings organized in various ways.

2.2.1 Hashes

Maps between string fields and string values (like a Python dictionary or Java HashMap).

- **Ideal for:** Representing objects (e.g., a User object with fields `name`, `age`, `password`).
- **Operations:**
 - `HSET key field value`: Set a field ($O(1)$).
 - `HMGET key field1 field2`: Get multiple fields ($O(N)$ where N is the number of fields retrieved).

2.2.2 Lists

Collections of string elements sorted by insertion order. Physically implemented as **Doubly Linked Lists**.

- **Performance Characteristics:**
 - Fast adding/removing from head or tail ($O(1)$).
 - Slow access by index ($O(N)$).
- **Use Cases:** Message queues, recent activity logs (e.g., Twitter timelines).
- **Commands:** LPUSH (push left), RPUSH (push right), LPOP (pop left).

2.2.3 Sets

Unordered collections of unique strings.

- **Properties:** No duplicate members.
- **Operations ($O(1)$):** SADD (add), SREM (remove), SISMEMBER (check existence).
- **Set Algebra ($O(N)$):** Redis can perform server-side unions (SUNION), intersections (SINTER), and differences (SDIFF).

2.2.4 Sorted Sets (ZSets)

Similar to Sets but every member is associated with a floating-point score. Members are sorted by score.

- **Implementation:** Uses a dual data structure consisting of a Hash Table (for uniqueness) and a **Skip List** (for sorting).
- **Complexity:** Add/Remove/Update is $O(\log N)$.
- **Use Cases:** Leaderboards, priority queues.
- **Commands:** ZADD, ZRANGE (get top N items), ZRANK.

3 System Architecture

3.1 Single-Threaded Event Loop

One of the most defining characteristics of Redis is its architecture.

- **Single-Threaded:** Redis uses a single thread for processing commands. This eliminates the need for context switching and locks, which are expensive in multi-threaded environments.
- **Implication:** Redis operations must be fast. A single slow command (like a set intersection on millions of items) blocks all other clients.
- **Event Loop:** It uses I/O Multiplexing (typically via `epoll` on Linux or `kqueue` on BSD) to handle thousands of concurrent client connections efficiently.

3.2 Memory Management

Redis resides in main memory (RAM).

- **Eviction Policies:** When memory is full, Redis can evict keys based on policies like LRU (Least Recently Used), LFU (Least Frequently Used), or TTL (Time To Live).
- **Virtual Memory (Historical Note):** Early versions of Redis attempted to implement application-level Virtual Memory to swap "cold" values to disk. This was complex and deprecated in favor of OS-level paging or simply adding more RAM/Sharding.

4 Data Persistence

Although Redis is an in-memory store, it offers persistence to ensure data survives server restarts.

4.1 RDB (Redis Database File) - Snapshots

Performs point-in-time snapshots of the dataset at specified intervals.

- **Mechanism:** The main process calls `fork()`. The child process writes the data to a file on disk while the parent continues serving clients. This utilizes the OS's **Copy-on-Write (CoW)** memory mechanism.
- **Configuration Example:** `save 60 1000` (Save if 60 seconds have passed AND at least 1000 keys have changed).
- **Pros:** Compact file, fast restart.
- **Cons:** Data loss occurs for the period between the last snapshot and a crash.

4.2 AOF (Append Only File)

Logs every write operation received by the server.

- **Mechanism:** Operations are appended to a log file. On restart, Redis replays the log to reconstruct the dataset.
- **Fsync Policies:**
 - **always:** Safe but slow (fsync every write).
 - **everysec:** Default. Good balance of speed and safety (max 1 second data loss).
 - **no:** Relies on OS to flush buffers (fastest, riskiest).
- **Compaction:** Redis supports `BGREWRITEAOF` to compress the log (e.g., turning 100 increments into a single "SET" command).

5 Replication and Distributed Consistency

5.1 Master-Slave Architecture

Redis uses a primary-replica (Master-Slave) topology.

- **Master:** Handles all **Writes**.
- **Slave:** Connects to the Master and replicates data. Slaves are typically Read-Only.
- **Replication Flow:** 1. Slave sends `SYNC`. 2. Master starts background save (RDB) and buffers new writes. 3. Master sends RDB to Slave. 4. Slave loads RDB. 5. Master sends buffered writes and continues streaming new commands (asynchronous replication).

5.2 CAP Theorem Analysis

In the context of the CAP theorem (Consistency, Availability, Partition Tolerance):

- **Consistency:** Redis replication is asynchronous. Therefore, it provides **Eventual Consistency**. There is a window where a slave may serve stale data, or a write acknowledged by the master is lost if the master crashes before propagating to slaves.
- **Partition Tolerance:** A major weakness in basic Redis setups. If the network partitions, the system requires logic (like Redis Sentinel or manual intervention) to promote a slave to master.
- **Availability:** High availability is achieved by adding slaves. If the master fails, the system inhibits writes (sacrificing availability for write operations) until a new master is elected.

6 Transactions

Redis transactions ensure that a group of commands is executed sequentially and in isolation.

6.1 Commands

- **MULTI:** Starts the transaction block.
- **EXEC:** Executes all queued commands.
- **DISCARD:** Aborts the transaction.
- **WATCH:** Implements Optimistic Locking (Check-and-Set). It monitors a key; if that key changes before **EXEC** is called, the transaction fails.

Note: Redis transactions do not support rollback in the traditional SQL sense. If a command fails during execution, the rest are still processed.

7 Real-World Case Studies

7.1 1. The Guardian (Document Review System)

- **Problem:** Needed to assign documents randomly to MP (Members of Parliament) expense reviewers.
- **Old Solution:** MySQL `ORDER BY RAND()`. This sort operation accounted for 90% of the database load.
- **Redis Solution:** Used Redis Sets and the `SRANDMEMBER` command.
- **Result:** Random extraction became an $O(1)$ operation, drastically reducing DB load.

7.2 2. StackOverflow (Multi-Level Caching)

- **L1 (Local Cache):** Caches user sessions and view counts on the local web server.
- **L2 (Site Cache):** Caches hot questions and user acceptance rates for a specific site (e.g., ServerFault vs. StackOverflow).
- **L3 (Global Cache):** Shared data among all sites (e.g., global inboxes).
- **Performance:** A dedicated Redis machine handled hundreds of thousands of operations with negligible CPU usage (0%) and low memory footprint.

8 Example: Shopping Cart Design

This example illustrates the difference between Relational modeling and Redis modeling.

8.1 Relational Model (SQL)

Requires two tables: `carts` and `cart_lines`. To add an item, you must perform an `UPDATE` query with a `WHERE` clause joining the tables.

```
UPDATE cart_lines SET Qty = Qty + 2
WHERE Cart=1 AND Product=28
```

8.2 Redis Model (NoSQL)

Uses a **Hash** data structure where the Key is the User/Cart ID, Fields are Product IDs, and Values are Quantities.

```
HINCRBY cart:james product:28 2
```

Advantages:

- No schema migration required to add items.
- Atomic increment operation (`HINCRBY`) handles concurrency automatically.
- Data for one user is physically stored together (locality of reference).