Redis

CST 363

What is Redis?

REmote **DI**ctionary **S**erver

- An open-source, in-memory, single-threaded "data structure store"
  - allows for the storage and retrieval of data structures, such as strings, hashes, lists, sets, and more
  - speedy – micro-seconds vs. the milli-seconds we’re used to with disk-based stores.
- Known for its high performance, scalability, and versatility
- Virtually every production system uses Redis (or an equivalent) for caching, sessions, rate-limiting, real-time messaging, and lightweight queues.
  - A staple of "database adjacencies."

History

- 2009 - Salvatore Sanfilippo, an Italian developer, found that MySQL could not provide the necessary performance for real-time web analytics
- Went from a small personal project from Sanfilippo, to being an industry standard for in-memory data storage.

Comparison to SQL

Aspect	SQL Databases	Redis
Data Model	Relational model with structured tables	NoSQL key-value store with support for multiple data structures (e.g., lists, sets, hashes, streams)
Storage	Disk-based storage with caching layers for performance	In-memory storage for speed, with optional disk persistence (RDB, AOF)
Query Language	Uses SQL (Structured Query Language)	Uses its own command set (e.g., GET, SET, HGETALL); no SQL
Use Cases	Structured data, complex queries, transactional integrity, relational models	Caching, real-time analytics, messaging, fast-access scenarios

Where Traditional RDBMSs Struggle (1 of 5)

Hot-Read Caching

- A small subset of data ("hot keys" or documents) is read far more often (10x - 100x) than the rest.
- Every cache-miss triggers planner, optimizer, disk I/O (~1–3 ms each)

With Redis:

- `GET user:1234` always sub-millisecond
- Built-in "Least Recently Used" (LRU) eviction, no extra query planning

Where Traditional RDBMSs Struggle (2 of 5)

High-Frequency Counters & Concurrency

- Atomic `UPDATE … SET counter = counter + 1` at very high queries per second (QPS)
- Row-level locks slow things down under heavy load.
- WAL/journaling thrash + autovacuum bloat slow sustained writes

With Redis:

- `INCR page:view:5678` is lock-free & `O(1)`

Where Traditional RDBMSs Struggle (3 of 5)

Ephemeral State & Session Management

- Short-lived data (sessions, feature-flags, temp locks) that expires on its own.
- Frequent `INSERT` / `DELETE` churn, index maintenance
- Table bloat → expensive `VACUUM`, fragmentation

With Redis:

- Keys with `TTL` auto-expire; no background `GC`
- In-memory list, set, hash structures for fast lookups

Where Traditional RDBMSs Struggle (4 of 5)

Event Streaming & Pub/Sub

- On-disk queues or tailable cursors lack true ordering & replay
- Notifications don’t scale to many consumers
- No built-in consumer groups or at-least-once delivery guarantees

With Redis: Redis Streams + consumer-groups for ordered, replayable log

Where Traditional RDBMSs Struggle (5 of 5)

Heavy Analytical & Batch Workloads

- Large scans, complex joins or aggregations, real-time analytics.
- Long-running queries lock resources, contend with OLTP traffic, and can overwhelm primary nodes.

With Redis:

- Can use RedisTimeSeries or Lua scripts for near-real-time stats

Core Data Structures

Redis supports five different data structures: strings, hashes, lists, sets and ordered sets – regardless of the type, a value is accessed by a key.

- **String**: arbitrary byte values
- **Hash**: maps of fields → values
- **List**: ordered collections
- **Set**: unique, unordered collections
- **Sorted Set**: scored, ordered collections

Data type	Canonical commands	Typical use
String	`SET/GET/INCRBY`	Counters, flags
Hash	`HSET/HGETALL`	User profiles
List	`RPUSH/LRANGE`	Log buffers, task queues
Set	`SADD/SMEMBERS`	Unique tags
Sorted Set	`ZADD/ZRANGE`	Leaderboards, feeds
Streams	`XADD/XREAD`	Append-only event logs

Memory-Oriented Caveats (1 / 2)

> Redis is fast **because** everything lives in RAM—here's how to keep it that way.

Lever	Why it matters	Quick tips
`maxmemory`	Hard ceiling for RAM use	Size for worst-case + headroom • Monitor with `INFO MEMORY`
Eviction policy	Decides *which* keys disappear when full	`allkeys-lru` (smart cache) • `volatile-ttl` (expire-only) • Benchmark with `redis-benchmark --lru`
Big-key anti-pattern	5 MB key blocks event loop, slows replicas & AOF rewrite	Shard large hashes/lists • Prefer many small keys

Memory-Oriented Caveats (2/2)

Lever	Why it matters	Quick tips
Memory-efficient structures	Same task, less RAM	Bitmaps for flags • HyperLogLog for cardinality • Bloom filters for “probable existence”
Diagnostics	Spot trouble early	`MEMORY USAGE <key>` • `MEMORY DOCTOR` • `MEMORY STATS`

Rule of thumb: *If a key can’t be read in ‹1 ms or fits on one screen, it’s probably too big—break it up or pick a leaner structure.*

Hands-on: Basic Commands

Access the Redis command line interface (redis-cli) with:

```
docker exec -it my-redis redis-cli
```

1. **Strings & counters**

```
SET page_views 0
INCRBY page_views 42
GET page_views    # 42
```

2. **Lists**

```
RPUSH recent_log "user123 signup" "user123 click"
LRANGE recent_log 0 -1
```

Hands-on: Basic Commands (cont.)

3. **Sets & membership**

```
SADD online_users u1 u2 u3
SCARD online_users
SISMEMBER online_users u2
```

4. **Pub/Sub teaser**

```
SUBSCRIBE classroom
# in another shell
PUBLISH classroom "Hello CST 363"
```

Redis Persistence & Durability

> **"Isn't everything lost on reboot?"**

Mechanism	What it does	Write frequency	Disk footprint	Worst-case data loss*
RDB snapshot	Forks the process and saves a compressed dump file (`.rdb`)	On a schedule (e.g., `save 900 1` , `save 300 10`)	Small, binary & compressed	All writes since last snapshot
AOF (Append-Only File)	Appends every write to a log and replays it on restart	`appendfsync always` | `everysec` | `no`	Larger, plaintext stream	0 s with `always` ; ≤ 1 s with `everysec`

*"Data loss" here means writes not yet persisted if the server crashes.

Hybrid best-practice

```
appendonly yes          # enable AOF
appendfsync everysec    # fsync once per second
save 3600 1             # RDB snapshot every hour
```

Blend near-zero data loss (AOF) with compact hourly snapshots (RDB) for quick restarts & cheaper off-box backups.

Redis + PostgreSQL ❤️‍🔥

- **Read-through cache**
- Flask route hits Redis → miss triggers Postgres query → result cached with TTL.
- **Write-behind / event sourcing**
    - App writes to Redis **Stream**; separate worker persists to Postgres asynchronously.
- **Real-time counters**
    - `INCR` in Redis, nightly ETL to Postgres for durable analytics.
- **Pub/Sub fan-out** for web-socket notifications while Postgres stays source of truth.
    - Publisher sends one message to a channel
    - Redis instantly "fans out" (copies) that message to every active subscriber on that channel.

NBA Play-by-Play App

Real-time streaming + durable storage

- "Pretend live" multi-game replay of NBA play-by-play
- Accelerated or real-time playback via `SIM_SPEED`
- **Redis** for low-latency fan-out & caching
- **Postgres** for durable event logging
- Browser dashboard via Flask + Server-Sent Events

Data Ingestion & Scheduling

1. **Read CSV** (`pbp2023.csv`) into Pandas
2. **Filter** to a few games on a certain date (here 2023-03-24)
3. **Compute** `elapsed_seconds` per event:

   ```
   elapsed = minutes*60 + seconds + periods_before*period_length
   ```

4. **Seed a min-heap** with each game's first event

   ```
   heap = [(elapsed0, game_id, idx0, events0), ... ]
   ```

Real-time Replay Loop

- **Pop** next event from heap → `(sim_t, gid, idx, evts)`
- **Sleep** until `(sim_t / SIM_SPEED)` matches wall-clock
- **Publish** payload → Redis pub/sub + cache
- **Batch** payload → Postgres `INSERT ... executemany`
- **Push** next event of that game back onto heap
- Repeat until heap is empty

Redis: Real-time Layer

- **Pub/Sub**
  - `r.publish("pbp_live", payload_json)`
  - Flask SSE subscribes → pushes to browsers
- **Cache**
  - `r.set("game:{id}:latest", payload_json, ex=120)`
  - Quick lookup of current score/state

Postgres: Durable Log

- Table `pbp(ts timestamptz, game_id text, … )`
- Batched writes (`200` events per COMMIT)
- Index on `(game_id, ts)` for fast history queries
- Enables audit, replay, analytics after the demo

Flask + SSE Front-End

- `/` → renders Bootstrap dashboard
- `/events` → SSE endpoint using `redis.pubsub().listen()`
- Browser JS:

```
const es = new EventSource("/events");
es.onmessage = ev ⇒ { … }
```

- Updates:
  - Scoreboard cards per game
  - Live feed list (last 100 plays)

Why This Architecture?

Concern	Tool	Role
Real-time updates	Redis	Pub/Sub & in-memory cache
Durable persistence	Postgres	Batched event store
Simple streaming	SSE	Server-Sent Events → browser `EventSource`
Merge multiple feeds	Heap	Time-ordered “live” event scheduling

Possible Enhancements

- **Scale fan-out**: Redis Streams, Kafka, or WebSockets
- **Async I/O**: FastAPI/Quart + async Redis client
- **Historical API**: add `/history?game_id=` querying Postgres
- **Schema enhancements**: numeric time fields, PKs, constraints
- **Simplify demo**: flatten+sort or `heapq.merge` for static CSV

Redis Docs

Check out the Redis website for documentation and tutorials!