PROJECT T4

Redis Data Acceleration

High-Performance Caching Layer for MongoDB Atlas

FastAPI Redis Docker

Introduction

This project demonstrates how to integrate **Redis** as a high-performance caching layer to accelerate data retrieval from a persistent database (MongoDB Atlas).

Goal

Reduce query latency and lower the load on the primary database by serving hot data from RAM.

Strategies

Implementation of Cache-Aside, Write-Through, and advanced structures like Sorted Sets & Geospatial Indexes.

Outcome

Improved overall system scalability and response times (from ~50ms to ~2ms).

System Architecture

Hybrid Cloud Setup

> **Localhost + Docker:** The application logic, cache, and monitoring tools run in isolated containers.

> **Cloud Infrastructure:** MongoDB Atlas acts as the persistent storage layer via Internet/TLS.

> **Telemetry:** Prometheus scrapes metrics which are visualized in Grafana.

```
graph TD subgraph Cloud [☁️ Cloud] Mongo[(MongoDB Atlas)] end subgraph Docker [🐳 Local Docker] UI[Streamlit] --> API[FastAPI] API --> Redis[(Redis)] API --> Mongo Prom[Prometheus] -.-> API Graf[Grafana] -.-> Prom end linkStyle 2 stroke:red,stroke-width:2px;
```

Data Flow Logic

Read: Cache-Aside

The standard strategy for reading data.

1. **Check Redis:** Service queries the cache first.
2. **Hit:** Return data instantly (~2ms).
3. **Miss:** Query MongoDB Atlas (~50ms), return data to user, and *asynchronously* populate Redis for next time.

Write: Write-Through

Ensuring consistency between Cache and DB.

1. **Write to DB:** Update the "Source of Truth" (MongoDB).
2. **Update Redis:** Immediately update or invalidate the cache entry.
3. **Result:** Redis never serves outdated (stale) data to the user.

Data Model

Sample Mflix Database

Movies are stored as BSON documents in MongoDB. We map these to various Redis structures based on the use case.

Redis Strings: Full JSON caching.

Redis Hashes: Optimized partial object storage.

Redis ZSets: Leaderboards (Top Movies).

```
{
  "_id": "573a1390f29313caabcd4803",
  "title": "Winsor McCay",
  "genres": ["Animation", "Short"],
  "runtime": 7,
  "year": 1911,
  "imdb": {
    "rating": 7.7,
    "votes": 1034
  },
  "poster": "https://m.media-amazon.com/..."
}
```

BSON Document Structure

Hardware & Software Configuration

Hardware

- **Host:** Apple ARM M1
- **RAM:** 16 GB Unified Memory
- **Virtualization:** Docker Desktop

Core Stack

`fastapi`
`uvicorn`

`redis`
`pymongo`

`python-dotenv`

Tools

`prometheus`
`grafana`

`streamlit`
`plotly`

`locust`

Implementation Highlights

Connection Factory

Robust connection handling using Singleton pattern in `database.py`.

```
# Connection Setup
mongo_client = MongoClient(MONGO_URL)
db = mongo_client["sample_mflix"]

redis_client = redis.Redis(
    host=REDIS_HOST,
    port=REDIS_PORT,
    decode_responses=True
)
```

Optimized Pipelining

Reducing Round-Trip Time (RTT) by batching commands.

```
def get_top_movies_optimized(limit):
    # Get IDs from ZSET
    top_ids = redis.zrevrange("leaderboard", 0, limit)

    # PIPELINE: Fetch all details in 1 RTT
    pipe = redis_client.pipeline()
    for mid in top_ids:
        pipe.hgetall(f"movie:hash:{mid}")

    return pipe.execute()
```

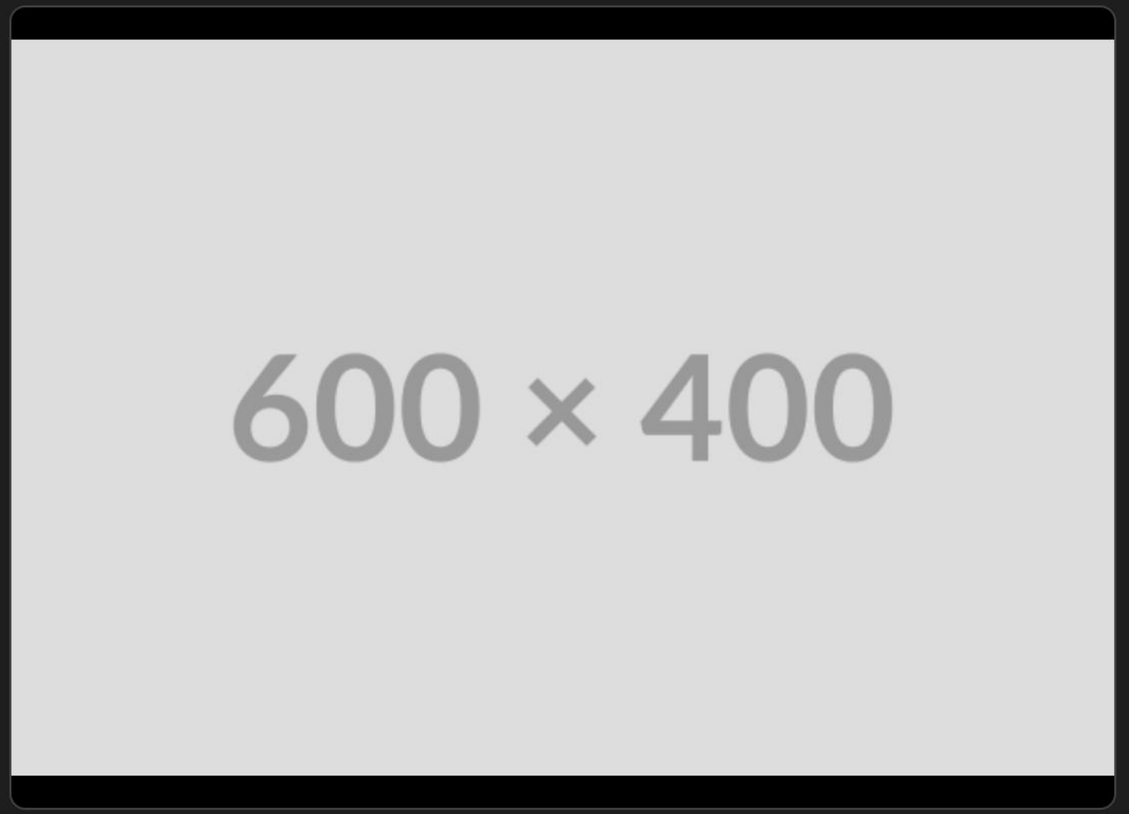
Performance Analysis

Locust Stress Test

We simulated high load (1 user peak, 120s runtime) to compare MongoDB vs Redis.

20x
Speed Increase

~0.3ms
Local Read Speed

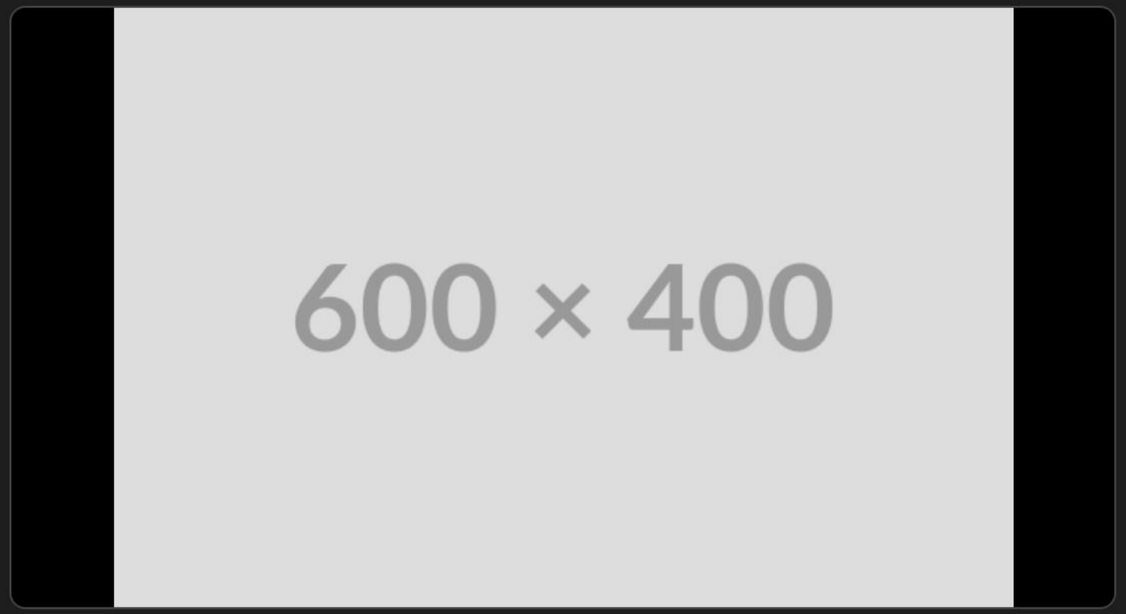


Advanced Metrics

Local Redis Speed

Achieving ~150x increase over raw Cloud DB.

Sets vs Hashes



Hashes are ~10x faster by fetching only needed fields.

Conclusion

The project successfully validates that integrating an in-memory layer drastically improves read performance. By using advanced structures like **Hashes** and techniques like **Pipelining**, we minimized network overhead and achieved scalable latency.

References: Redis.io Documentation • MongoDB Aggregation Framework • FastAPI Docs • Prometheus.io