Report for Lab 2 : Chirp

205.2 - Beyond relational databases

Jeremy Duc

30 March 2025

Contents

1 Introduction and Motivation 3

1.1 Context 3

1.2 Project Objectives 3

1.3 Motivation 3

2 Requirements Clarification and Assumptions 4

2.1 Core Requirements 4

2.2 Assumptions 4

2.3 Design Decisions 4

3 Data Modeling as Key-Value 5

3.1 Key Design Principles 5

3.2 Data Models 5

3.2.1 User Model 5

3.2.2 Chirp Model 5

3.2.3 Timeline and Rankings 6

3.3 Data Relationships 6

4 Software Architecture and Functionalities 8

4.1 System Architecture 8

4.2 Core Components 8

4.2.1 Data Model (ChirpRedisModel) 8

4.2.2 Command-line Interface (ChirpApp) 9

4.2.3 Web Interface (Streamlit) 9

4.3 Data Flow 10

4.3.1 Posting a New Chirp 10

4.3.2 Retrieving Latest Chirps 10

4.4 Data Import Pipeline 11
4.5 Implemented Functionalities 11
4.5.1 Core Features 11
4.5.2 Additional Features 11

5 Testing **12**
5.1 Testing Strategy 12
5.2 Test Coverage 12

6 Conclusion and Future Work **13**
6.1 Achievements 13
6.2 Challenges and Limitations 13
6.3 Future Work 14
6.4 Lessons Learned 14
6.5 Conclusion 14

1 Introduction and Motivation

1.1 Context

This report presents the design, implementation, and evaluation of Chirp (Compact Hub for Instant Real-time Posting), a simplified Twitter clone developed as part of the "205.2 Beyond relational databases" course. The project demonstrates the application of key-value database concepts using Redis, moving beyond traditional relational database paradigms to explore alternative data modeling approaches.

1.2 Project Objectives

The primary objectives of this laboratory project were:

1. Learn to model a practical data-intensive application as a key-value database
2. Translate application requirements into implementation tasks and data modeling
3. Implement and interact with a key-value store from a programmatic perspective

1.3 Motivation

Social media platforms represent one of the most challenging use cases for database systems due to their high write throughput, complex data relationships, and need for real-time access. By implementing a Twitter-like service with Redis, this project provides hands-on experience with non-relational database patterns that can handle these requirements effectively.

Twitter (now X) serves as an excellent model for this exercise since it has well-defined core functionalities that lend themselves to key-value representation. The ability to post short messages, follow users, and retrieve timelines aligns well with the strengths of key-value stores like Redis, which excel at fast read/write operations and sorted collections.

2 Requirements Clarification and Assumptions

2.1 Core Requirements

Based on the laboratory instructions, the minimal requirements for the Chirp application were:

- **Following/followers**: Each user can have followers and follow other users
- **Chirps**: Users can post small text-only messages in English
- **Rankings**: The system must track and display various rankings:
  - Top 5 users with highest follower counts
  - Top 5 users with most chirps
  - List of 5 latest chirps

2.2 Assumptions

Several assumptions were made to guide the implementation:

1. **Unidirectional relationship model**: Following is unidirectional, meaning if user A follows user B, it doesn't imply that B follows A
2. **Limited timeline size**: To prevent memory exhaustion, the timeline will be capped at 100,000 entries
3. **No tweet deletion**: For simplicity, the initial version doesn't support chirp deletion
4. **Simple authentication**: User authentication is not implemented in this version
5. **English-only content**: As specified in the requirements, only English chirps are considered
6. **Engagement metrics**: Additional engagement metrics (likes, rechirps) were added to make the application more realistic

2.3 Design Decisions

Several key decisions guided the implementation strategy:

1. **Python as primary language**: Python was chosen for its excellent Redis library support and ability to rapidly prototype the application
2. **Separation of concerns**: The implementation strictly separates the data model, command-line interface, and web interface to ensure maintainability
3. **Streamlit for web interface**: Streamlit was selected for the web interface due to its simplicity and rapid development capabilities
4. **Redis as the sole database**: All data is stored in Redis with no secondary storage systems
5. **Unit testing**: Comprehensive unit tests were implemented to ensure reliability
6. **Import pipeline**: A data import pipeline was created to populate the system with realistic Twitter data

3 Data Modeling as Key-Value

3.1 Key Design Principles

The data model was designed around several key principles specific to key-value databases:

1. **Denormalization**: Data is intentionally duplicated where necessary to optimize read performance
2. **Composite keys**: Meaningful prefixes are used to organize related data
3. **Appropriate data structures**: Redis data types (hashes, sorted sets, lists, etc.) are selected based on access patterns
4. **Indexing for fast lookups**: Secondary indices are created for frequently queried attributes
5. **Score-based sorting**: Timestamps and counts are used as scores in sorted sets for efficient ranking

3.2 Data Models

3.2.1 User Model

Users are modeled using Redis hashes with the key pattern `users:{user_id}`:

```
# User hash example (users:123456789)
{
    "username": "testuser",
    "name": "Test User",
    "follower_count": 100,
    "following_count": 50,
    "chirp_count": 200,
    "created_at": "Mon Apr 01 12:00:00 +0000 2025",
    "profile_image": "https://example.com/image.jpg"
}
```

Listing 1: User data structure in Redis

For quick username lookups, a separate hash maps usernames to user IDs:

```
# Username index (usernames)
{
    "testuser": "123456789",
    "anotheruser": "987654321",
    ...
}
```

Listing 2: Username to user ID mapping

3.2.2 Chirp Model

Chirps (tweets) are stored as hashes with the key pattern `chirp:{chirp_id}`:

```
# Chirp hash example (chirp:987654321)
{
    "text": "This is a test chirp!",
    "user_id": "123456789",
    "username": "testuser",
    "created_at": "Mon Apr 01 12:30:00 +0000 2025",
    "lang": "en",
    "favorite_count": 10,
    "retweet_count": 5
}
```

Listing 3: Chirp data structure in Redis

3.2.3 Timeline and Rankings

Redis sorted sets are used for the timeline and user rankings:

```
# Global timeline (chirps:timeline)
# Format: chirp_id -> timestamp
# Sorted by timestamp for chronological order
{
    "987654321": 1712055000.0,
    "987654322": 1712055060.0,
    ...
}

# Top users by followers (users:top_followers)
# Format: user_id -> follower_count
# Sorted by follower count
{
    "123456789": 100,
    "987654321": 200,
    ...
}

# Top users by chirp count (users:top_posters)
# Format: user_id -> chirp_count
# Sorted by chirp count
{
    "123456789": 200,
    "987654321": 150,
    ...
}
```

Listing 4: Timeline and ranking data structures

3.3 Data Relationships

Relationships between entities are modeled through:

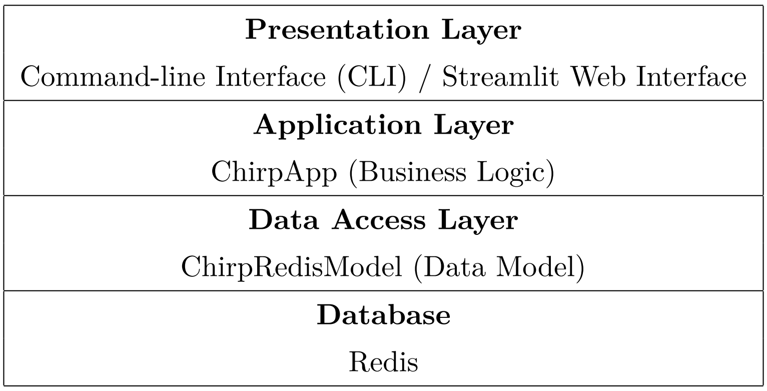
1. **Reference by ID**: Chirps contain user_id to establish ownership
2. **Denormalized fields**: Usernames are duplicated in chirp records for performance
3. **Counters**: Follower/following counts are maintained in user records

4. **Sorted sets**: Used to maintain relationships with additional metadata (timestamps, counts)

4 Software Architecture and Functionalities

4.1 System Architecture

The Chirp application is organized into a layered architecture:



Presentation Layer

Command-line Interface (CLI) / Streamlit Web Interface

Application Layer

ChirpApp (Business Logic)

Data Access Layer

ChirpRedisModel (Data Model)

Database

Redis

Figure 1: Architectural layers of the Chirp application

The project is structured as follows:

```
chirp-redis-keyvalue-lab/
|-- src/                    # Main source code
|   |-- app/                # Application code
|   |   |-- __init__.py
|   |   |-- chirp_app.py    # Command-line application
|   |   ‘-- streamlit_app.py # Web application
|   ‘-- models/             # Redis data models
|       |-- __init__.py
|       ‘-- redis_model.py  # Core Redis data model implementation
|-- scripts/                # Utility scripts
|   |-- import_data.py      # Data import script
|   |-- process_jsonl.py    # Data processing script
|   |-- reset_db.py         # Database reset script
|   |-- run_app.py          # Application launcher
|   ‘-- fix_engagement.py   # Script to add engagement metrics
|-- data/                   # Generated data
|   ‘-- processed/          # Processed data directory
‘-- tests/                  # Test suite
    |-- __init__.py
    |-- conftest.py
    |-- test_redis_model.py
    |-- test_import_data.py
    ‘-- test_streamlit_app.py
```

4.2 Core Components

4.2.1 Data Model (ChirpRedisModel)

The data model class encapsulates all interactions with Redis, providing an abstraction layer for the application logic:

```
class ChirpRedisModel:
    # Core methods
    def import_user(self, user_data): ...
    def import_chirp(self, chirp_data): ...
    def get_latest_chirps(self, count=5): ...
    def get_top_users_by_followers(self, count=5): ...
    def get_top_posters(self, count=5): ...
    def post_chirp(self, user_id, text): ...
    def like_chirp(self, chirp_id): ...
    def rechirp(self, chirp_id): ...
    def add_user(self, username, name, profile_image=''): ...
    def get_top_liked_chirps(self, count=5): ...
    def get_top_rechirped_chirps(self, count=5): ...
    def reset_db(self): ...
```

Listing 5: Key methods in the ChirpRedisModel class

4.2.2 Command-line Interface (ChirpApp)

The CLI provides an interactive interface to the Chirp functionality:

```
class ChirpApp:
    def __init__(self, host='localhost', port=6379, db=0): ...
    def display_welcome(self): ...
    def display_help(self): ...
    def format_chirp(self, chirp): ...
    def format_user(self, user): ...
    def display_latest_chirps(self): ...
    def display_top_followers(self): ...
    def display_top_posters(self): ...
    def post_new_chirp(self, username, text): ...
    def like_chirp(self, chirp_id): ...
    def rechirp(self, chirp_id): ...
    def add_new_user(self, username, name): ...
    def display_top_liked(self): ...
    def display_top_rechirped(self): ...
    def run(self): ...
```

Listing 6: ChirpApp class structure

4.2.3 Web Interface (Streamlit)

The Streamlit app provides a user-friendly web interface:

```
# Initialize the Redis model
@st.cache_resource
def get_model(): ...

# Main page components
st.title("Chirp")
st.subheader("Compact Hub for Instant Real-time Posting")

# Navigation
page = st.sidebar.radio("Go to", ["Home", "Post a Chirp", "Top Users", "About"])

# Page implementations (Home, Post a Chirp, Top Users, About)
if page == "Home":
    # Display latest chirps, most liked, most rechirped
    ...
elif page == "Post a Chirp":
    # Form to post new chirps or add users
    ...
elif page == "Top Users":
    # Display top users by followers and chirps
    ...
elif page == "About":
    # About page content
    ...
```

Listing 7: Streamlit app structure

4.3 Data Flow

4.3.1 Posting a New Chirp

The process of posting a new chirp involves:

1. The user submits text content via CLI or web interface
2. The application layer validates the input and calls the data model
3. The data model:
   - Generates a unique chirp ID based on timestamp
   - Creates a chirp hash with the text, user information, and metadata
   - Adds the chirp to the timeline sorted set with the timestamp as score
   - Increments the user's chirp count
   - Updates the top posters ranking
4. The application confirms the operation with feedback to the user

4.3.2 Retrieving Latest Chirps

When fetching the latest chirps:

1. The application requests latest chirps from the data model
2. The data model:
   - Queries the timeline sorted set for the most recent chirp IDs
   - Retrieves the full chirp data for each ID
   - Formats and returns the complete chirp objects
3. The application layer formats the chirps for display
4. The interface presents the formatted chirps to the user

4.4 Data Import Pipeline

A data import pipeline was developed to process Twitter data:

1. **processing.py** - Extracts English tweets from compressed JSON files
2. **import_data.py** - Filters and imports tweets into Redis
3. **fix_engagement.py** - Adds realistic engagement metrics

4.5 Implemented Functionalities

4.5.1 Core Features

- User profile management (creation, statistics)
- Posting new chirps (text-only messages)
- Viewing latest chirps timeline
- Viewing top users by followers
- Viewing top users by post count

4.5.2 Additional Features

- Engagement metrics (likes, rechirps)
- Top chirps by engagement (most liked, most rechirped)
- Web interface with Streamlit
- Data import capabilities
- Database reset functionality

5 Testing

5.1 Testing Strategy

A comprehensive testing approach was implemented with:

- Unit tests for the Redis model
- Import functionality tests
- Web interface tests using mocks

The testing framework uses:

- pytest as the test runner
- fakeredis for Redis mocking
- unittest.mock for additional mocking
- pytest-cov for coverage reporting

5.2 Test Coverage

The test suite covers:

- Core data model operations
- Data import functionality
- User and chirp management
- Timeline and ranking features
- Engagement functionality

6 Conclusion and Future Work

6.1 Achievements

This project successfully implemented:

- A comprehensive key-value data model for a social media application
- Efficient data structures for timeline, user management, and rankings
- Both command-line and web interfaces
- Data import functionality
- Additional engagement features

The implementation demonstrates:

- Effective use of Redis data structures (hashes, sorted sets)
- Appropriate denormalization strategies
- Performance optimization through careful key design
- Clean separation of concerns in architecture

6.2 Challenges and Limitations

Several challenges were encountered:

- Modeling relationships in a key-value store requires denormalization
- Managing sorted sets for rankings requires careful transaction management
- Limited options for complex queries compared to relational databases
- Memory consumption of denormalized data

Current limitations of the implementation:

- No support for media content (images, videos)
- Limited user authentication and security
- No support for hashtags or mentions
- No support for chirp deletion or editing
- Limited search capabilities

6.3 Future Work

Potential improvements for future iterations:

- Implementing a follow/unfollow mechanism
- Adding personalized user timelines
- Supporting hashtags and topic trending
- Implementing user mentions and notifications
- Adding search functionality
- Supporting chirp deletion and editing
- Improving security with authentication
- Implementing data sharding for scalability
- Adding media content support

6.4 Lessons Learned

This project provided valuable insights into:

- Non-relational data modeling approaches
- Performance implications of different Redis data structures
- Trade-offs between normalization and query performance
- Importance of key design in key-value databases
- Benefits and limitations of key-value stores for social media applications

6.5 Conclusion

The Chirp project demonstrates that key-value databases like Redis can effectively support core social media functionalities with excellent performance characteristics. The implementation successfully met all the laboratory requirements while providing additional features to enhance the application's realism and usability.

The denormalized data model and careful selection of Redis data structures enabled efficient operations for timeline retrieval, user rankings, and chirp management. This approach highlights the strengths of key-value databases in handling high-throughput, real-time social media scenarios, while also illustrating the design considerations necessary when moving beyond traditional relational database models.