Redis + Python Notes

Introduction to Redis and Redis-py

Redis (Remote Dictionary Server) is an open-source, in-memory data structure store that can be used as a database, cache, message broker, and queue. It supports various data structures such as strings, hashes, lists, sets, and more.

Redis-py is the official Python client library for Redis. It provides a Python interface to interact with Redis servers and supports all Redis commands. The library is maintained by Redis, Inc. (formerly Redis Labs) and is available on GitHub.

Setting Up Redis with Python

Install the client library:

```
pip install redis
```

Connecting to Redis:

```
import redis
redis_client = redis.Redis(
    host='localhost',
    port=6379,
    db=2,
    decode_responses=True
)
```

Key connection parameters:

- `host`: The Redis server address (localhost or 127.0.0.1 for local development)
- `port`: Usually 6379 by default
- `db`: Redis supports multiple logical databases (0-15)
- `decode_responses`: When True, converts byte responses to strings automatically

Working with Redis Data Types

Strings

Redis strings can store text, integers, or binary data. They're useful for simple key-value pairs, counters, and caching.

```
# Set and get a simple string
redis_client.set('user:name', 'Alice')
name = redis_client.get('user:name')

# Counters
redis_client.set('page:views', 0)
redis_client.incr('page:views')  # Increment by 1
redis_client.incrby('page:views', 10)  # Increment by 10

# Multiple operations
redis_client.mset({'key1': 'val1', 'key2': 'val2', 'key3': 'val3'})
values = redis_client.mget('key1', 'key2', 'key3')
```

Key string commands:

- `setex()`: Set with expiration time
- `setnx()`: Set only if key doesn't exist
- `getdel()`: Get value and delete key
- `append()`: Append to existing string
- `strlen()`: Get string length

Lists

Redis lists are linked lists that allow for quick operations at both ends. They're perfect for queues, recent activity lists, and message processing.

```
# Create a list
redis_client.rpush('tasks', 'task1', 'task2', 'task3')

# Get all elements
tasks = redis_client.lrange('tasks', 0, -1)  # 0 to -1 means all elements

# Pop elements from either end
first_task = redis_client.lpop('tasks')  # First in, first out (queue)
last_task = redis_client.rpop('tasks')   # Last in, first out (stack)
```

Key list commands:

- `lpush()`: Add to the beginning

- `rpush()`: Add to the end
- `lpop()`: Remove from beginning
- `rpop()`: Remove from end
- `llen()`: Get list length
- `lrem()`: Remove specific elements
- `lpos()`: Find position of an element

Hashes

Redis hashes store field-value pairs under a single key, making them ideal for representing objects or records.

```
# Create a hash
redis_client.hset('user:123', mapping={
    'username': 'alice',
    'email': 'alice@example.com',
    'visits': 10
})

# Get all fields and values
user_data = redis_client.hgetall('user:123')

# Get specific fields
email = redis_client.hget('user:123', 'email')

# Get all keys in the hash
fields = redis_client.hkeys('user:123')
```

Key hash commands:

- `hset()`: Set field(s) in a hash
- `hget()`: Get a field value
- `hgetall()`: Get all fields and values
- `hdel()`: Delete specific fields
- `hexists()`: Check if field exists
- `hlen()`: Get number of fields

Advanced Redis Concepts

Redis Pipelines

Pipelines allow bundling multiple commands to reduce network round trips, improving performance for batch operations.

```
# Create a pipeline
pipe = redis_client.pipeline()

# Queue commands (not executed yet)
for i in range(5):
    pipe.set(f"key:{i}", f"value:{i}")

# Execute all commands in a single request
results = pipe.execute()

# Chain commands
results = pipe.get("key:0").get("key:1").get("key:2").execute()
```

Benefits of pipelines:

- Reduced network overhead
- Atomic execution of commands
- Better performance for batch operations

Redis in Data Science and Machine Learning

Redis serves critical roles in ML systems:

1. **Feature Store**: Redis is excellent for online feature stores in ML systems where low latency (<10ms) is crucial for real-time prediction services.

2. **Inference Store**: Storing pre-computed features or model results for quick lookups during inference.

3. **Caching Layer**: Caching frequently accessed data or model predictions to reduce computation load.

4. **Model Registry**: Tracking model versions, parameters, and metadata.

5. **Streaming Data Processing**: Using Redis streams for real-time data processing pipelines.

The feature store architecture typically consists of:

- Offline Store: For batch processing and training (often using databases like Snowflake)

- Online Store: For real-time serving (where Redis excels)
- Feature Registry: For tracking feature definitions and metadata

Redis Best Practices

1. **Key Naming Conventions**:
    - Use colon `:` for namespacing (e.g., `user:123:profile`)
    - Be consistent with naming patterns
    - Consider including data type in keys for clarity
2. **Memory Management**:
    - Set appropriate expiration times (TTL) for ephemeral data
    - Use `SCAN` instead of `KEYS` for large databases
    - Monitor memory usage with `INFO memory`
3. **Performance Optimization**:
    - Use pipelining for bulk operations
    - Consider using connection pooling for concurrent applications
    - Use Redis data structures that best match your access patterns
4. **Security**:
    - Set strong passwords
    - Limit network exposure
    - Use Redis ACLs (Access Control Lists) for fine-grained permissions
5. **Data Persistence**:
    - Configure RDB snapshots or AOF logs based on your durability needs
    - Implement proper backup strategies

Common Redis Use Cases

1. **Caching**: Temporary storage to speed up repeated data access
2. **Session Management**: Storing user session data
3. **Rate Limiting**: Controlling API requests frequency
4. **Leaderboards and Counters**: Using sorted sets for rankings
5. **Pub/Sub Messaging**: For implementing messaging patterns
6. **Task Queues**: Basic job processing and distribution
7. **Real-time Analytics**: Tracking metrics with low latency
8. **Geospatial Applications**: Using Redis geo commands