MongoDB, Redis를 연계한 트래픽 처리

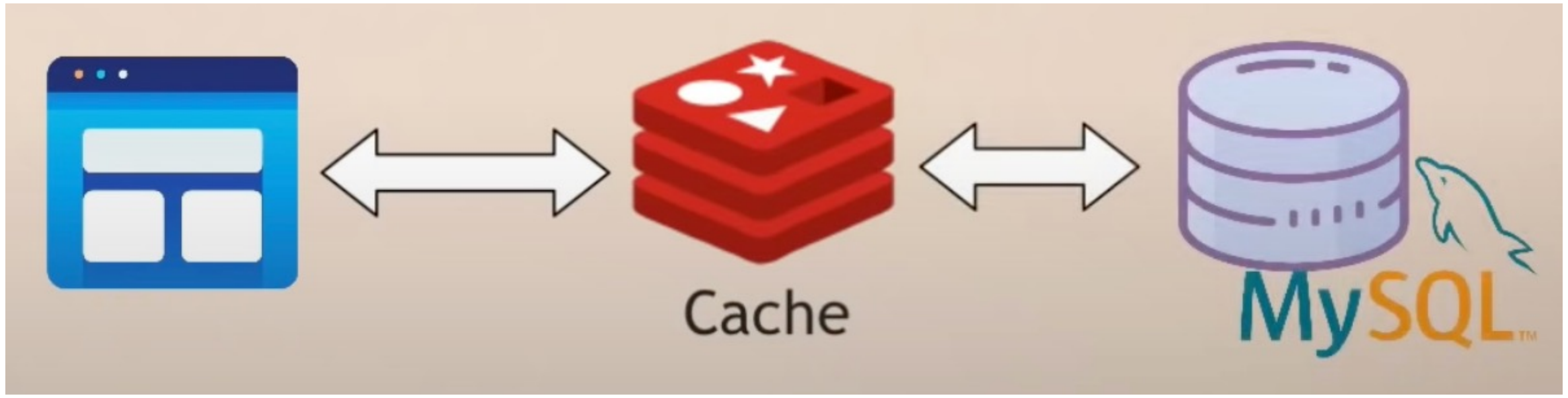
hwpark@wiken.co.kr

What is Redis?

- Redis stands for "REmote DIctionary Server," an open-source, in-memory data structure store used as a database, cache, and message broker.
- Known for its speed, reliability, and flexibility, Redis is optimized for high performance, providing a solution for many challenges of scalability and latency.

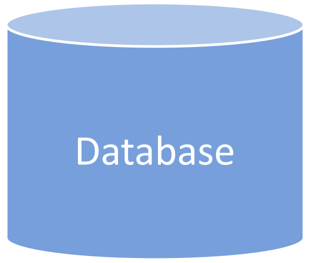
What is Redis?

- In-Memory Database – data stored on computer’s memory.
- Usage of Redis
  - Often used as cache to improve performance
  - Redis is a fully-fledged primary database
  - Persist multiple data formats



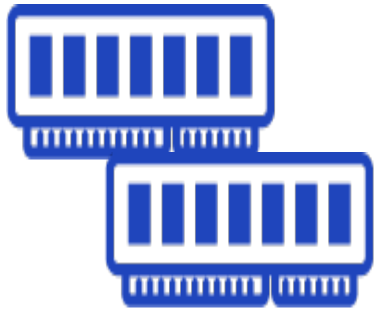
Redis as a Database

- Redis: An in-memory data structure store
  - Offers high-speed data retrieval and writing
  - Operates on an in-memory dataset
- Redis Persistence:
  - Provides persistence via snapshots and append-only files
  - Ensures data safety even during system restarts or outages
- Full-fledged Database:
  - Structured data storage using keys
  - Preserves data persistently to prevent loss



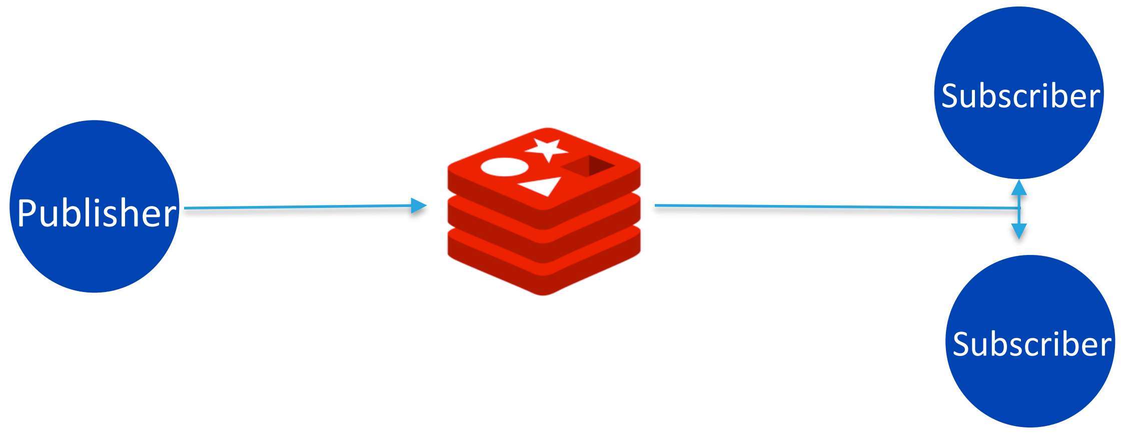
Redis as a Cache

- Redis: A high-performance cache
  - Provides temporary data storage for speedier data retrieval
  - Stores precomputed results or frequently accessed data in memory
- Performance Improvement:
  - Reduces need to access slower external data sources
  - Improves overall system performance



Redis as a Message broker

- Redis: A Pub/Sub Model User
  - Enables real-time communication between services
  - Ideal for chat, video streaming, or live tracking applications
- Swift and Efficient Message Exchanges:
  - Optimizes communication between system components
  - Leads to improved overall system efficacy



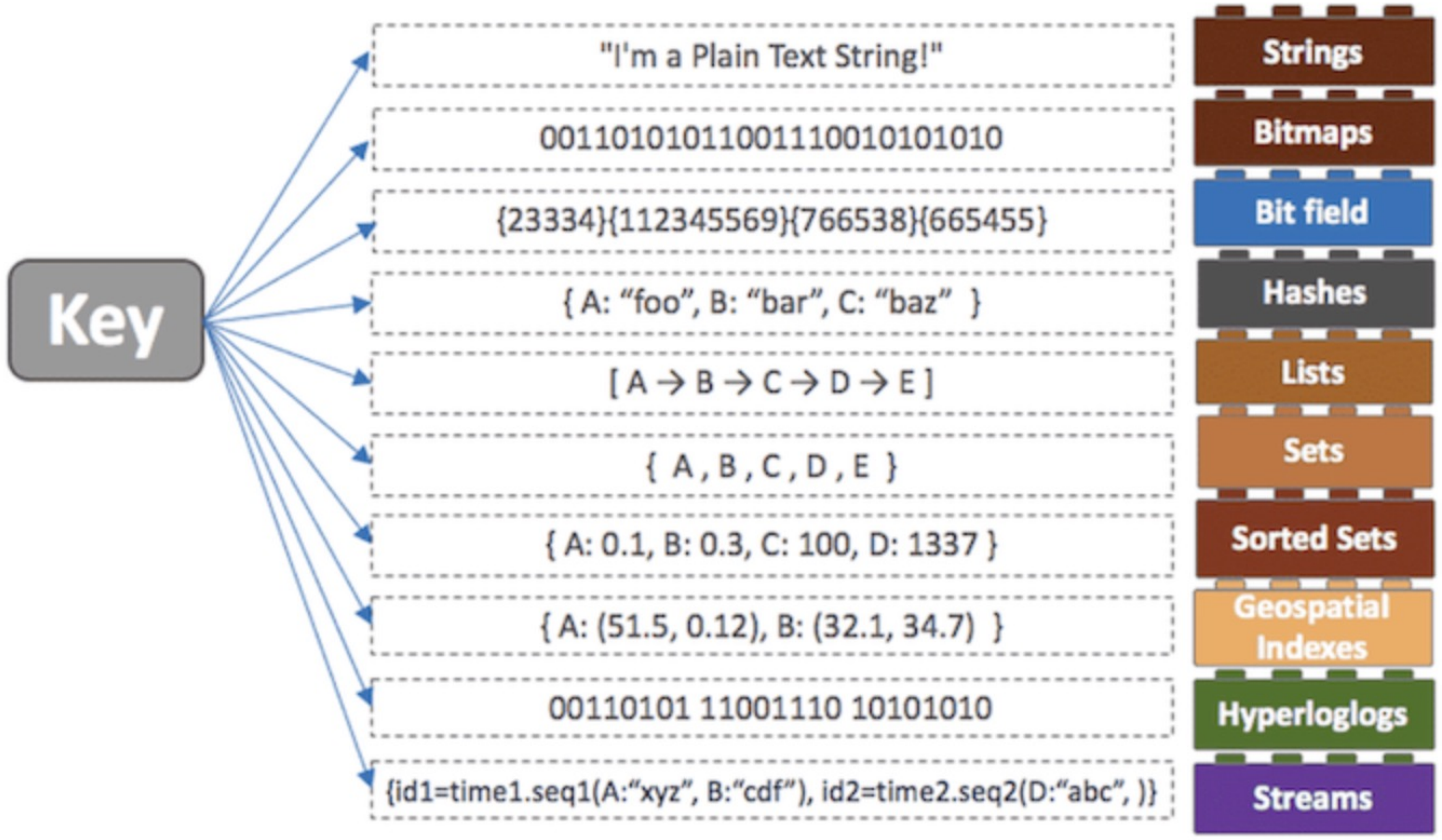
Redis Data Model

- Redis: A Key-Value Store
  - Dictionary model with keys mapped to values
  - Keys are binary-safe, can contain varied data types
  - Keys are unique and have a max size of 512MB
- Redis Value Types:
  - Complex types, setting Redis apart from other key-value stores
  - Supports strings, lists, sets, sorted sets, hashes, bitmaps, hyperloglogs, geospatial indexes
  - Varied data structures for greater flexibility and efficiency

Keys and Values in Redis

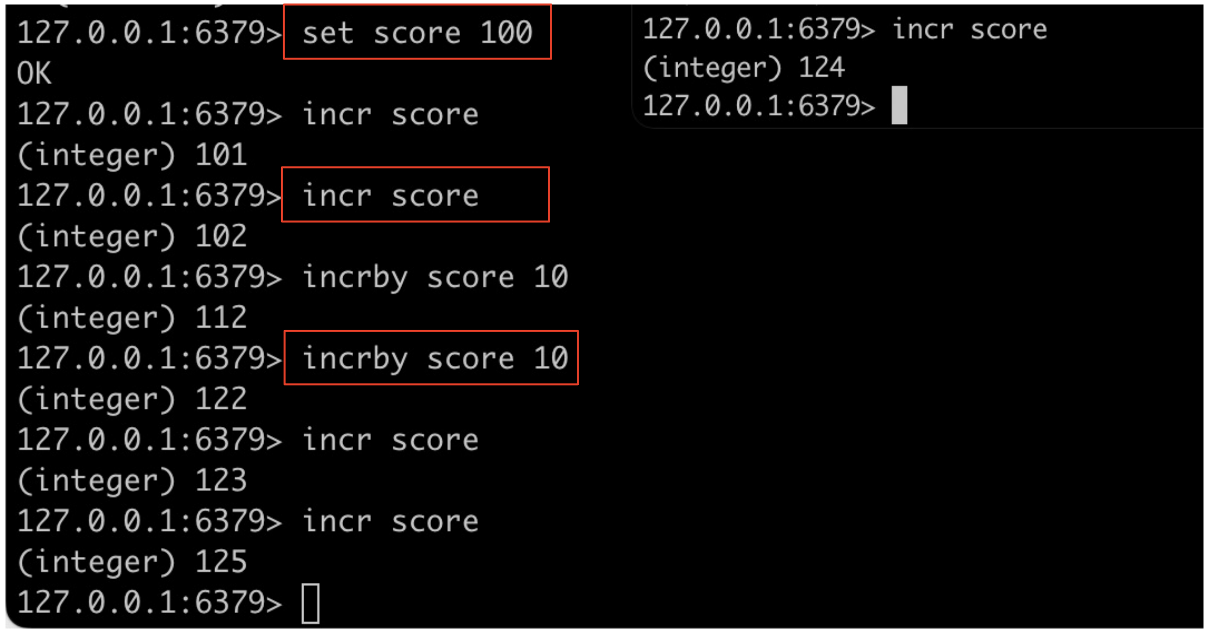
- Keys in Redis:
  - Names for values, used to store assigned objects
  - Association with values enables easy, fast data retrieval
- Value Handling:
  - Handles serialized data and complex types
  - In-memory storage allows for quick, efficient data operations
- Key Features:
  - Supports setting TTL (Time to Live) for keys
  - Automatic deletion of keys after set time
  - Useful feature for cache management

Redis Data Structures



Redis Strings

- **Strings: The Simplest Redis Data Structure**
  - Basic form of Redis values, binary-safe.
  - Can contain any kind of data up to 512MB.
  - Suitable for simple key-value pairs, counters, and serialized data.
  - Examples - counters

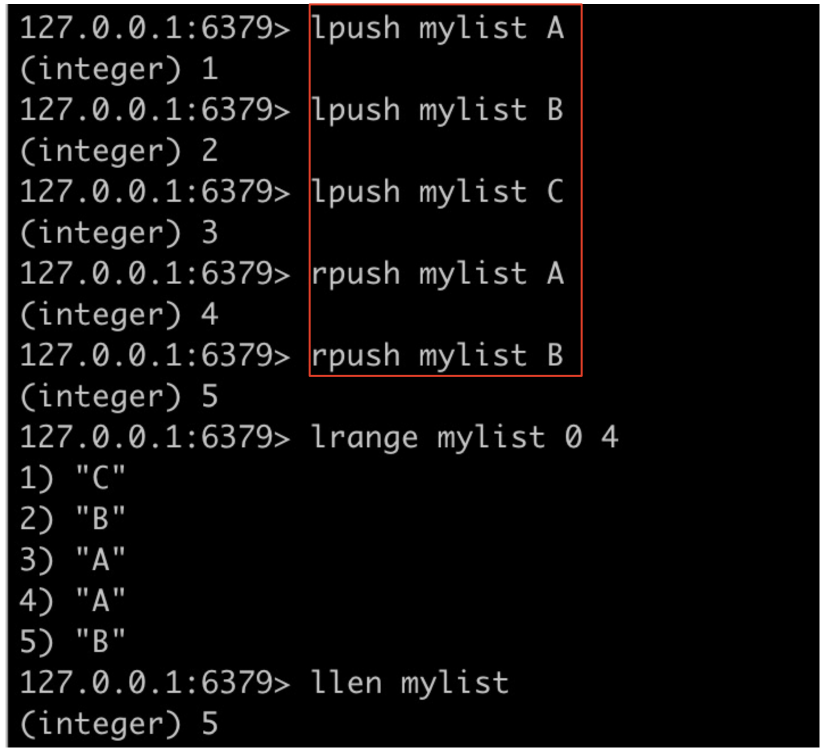


```
127.0.0.1:6379> set score 100
OK
127.0.0.1:6379> incr score
(integer) 101
127.0.0.1:6379> incr score
(integer) 102
127.0.0.1:6379> incrby score 10
(integer) 112
127.0.0.1:6379> incrby score 10
(integer) 122
127.0.0.1:6379> incr score
(integer) 123
127.0.0.1:6379> incr score
(integer) 125
127.0.0.1:6379>
```

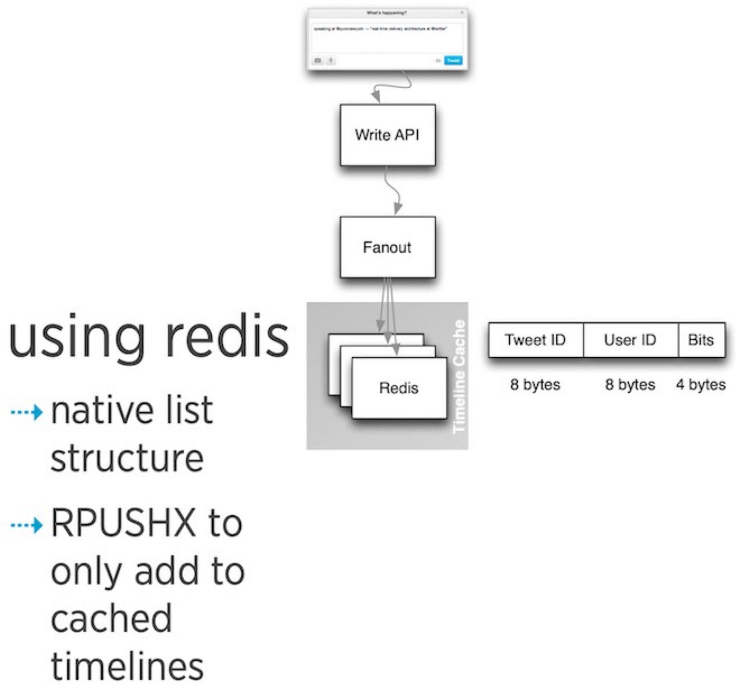
```
127.0.0.1:6379> incr score
(integer) 124
127.0.0.1:6379>
```

Redis Lists

- **Lists: Collections of String Elements**
  - Collections of string elements, ordered by insertion.
  - Ideal for implementing stacks or queues.
  - Supports push, pop operations and indexing.



```
127.0.0.1:6379> lpush mylist A
(integer) 1
127.0.0.1:6379> lpush mylist B
(integer) 2
127.0.0.1:6379> lpush mylist C
(integer) 3
127.0.0.1:6379> rpush mylist A
(integer) 4
127.0.0.1:6379> rpush mylist B
(integer) 5
127.0.0.1:6379> lrange mylist 0 4
1) "C"
2) "B"
3) "A"
4) "A"
5) "B"
127.0.0.1:6379> llen mylist
(integer) 5
```

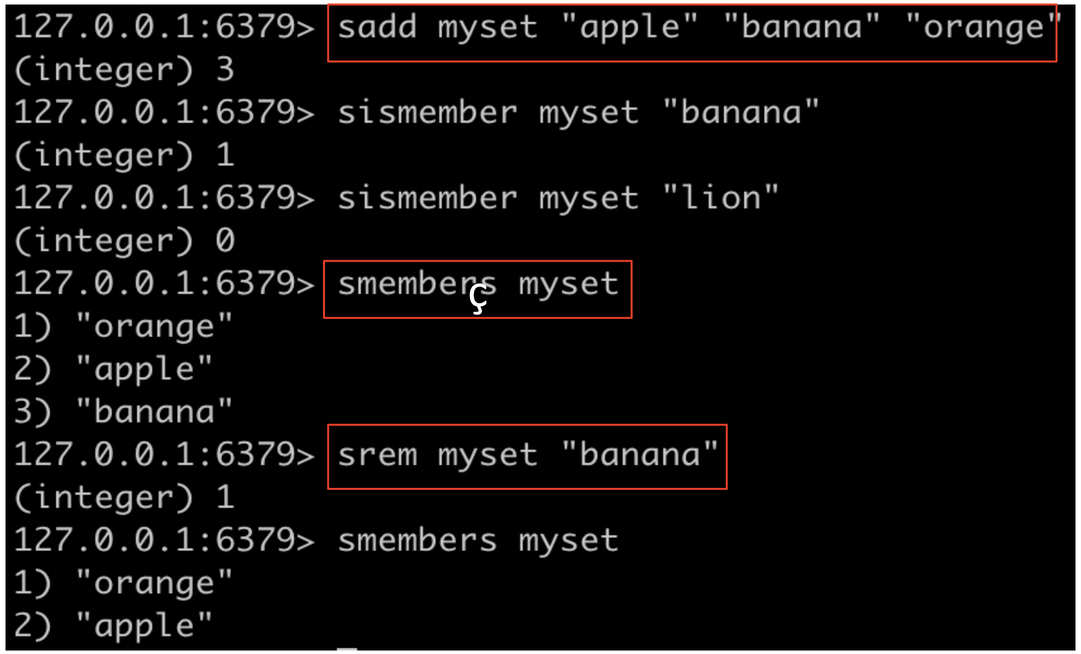


using redis

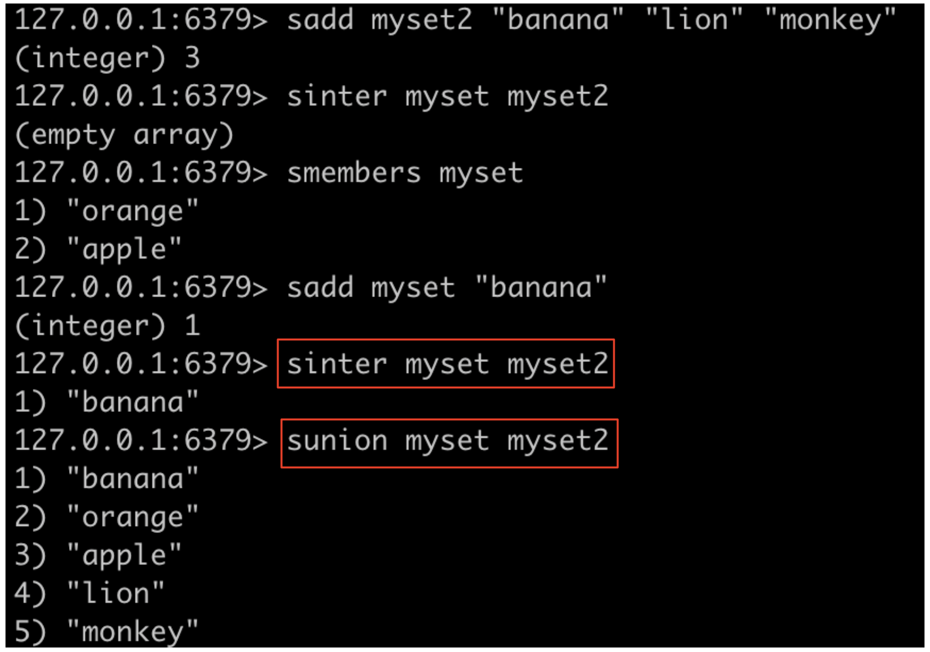
- native list structure
- RPUSHX to only add to cached timelines

Redis Sets

- **Sets: Unordered Collections of Strings**
  - Unordered collections of unique strings.
  - Supports adding, removing, and testing for existence of members.
  - Ideal for tracking unique items, set operations like intersection and union.



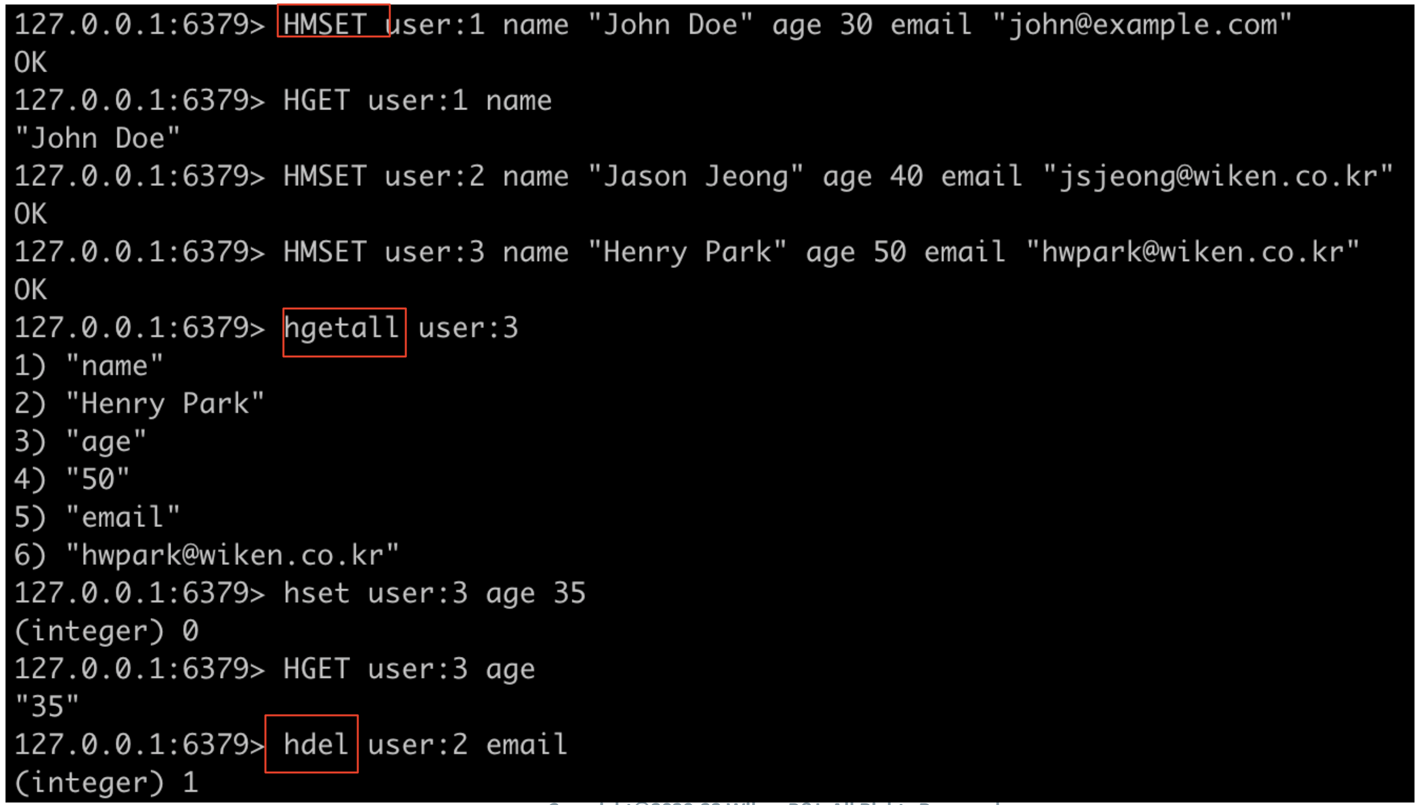
```
127.0.0.1:6379> sadd myset "apple" "banana" "orange"
(integer) 3
127.0.0.1:6379> sismember myset "banana"
(integer) 1
127.0.0.1:6379> sismember myset "lion"
(integer) 0
127.0.0.1:6379> smembers myset
1) "orange"
2) "apple"
3) "banana"
127.0.0.1:6379> srem myset "banana"
(integer) 1
127.0.0.1:6379> smembers myset
1) "orange"
2) "apple"
```



```
127.0.0.1:6379> sadd myset2 "banana" "lion" "monkey"
(integer) 3
127.0.0.1:6379> sinter myset myset2
(empty array)
127.0.0.1:6379> smembers myset
1) "orange"
2) "apple"
127.0.0.1:6379> sadd myset "banana"
(integer) 1
127.0.0.1:6379> sinter myset myset2
1) "banana"
127.0.0.1:6379> sunion myset myset2
1) "banana"
2) "orange"
3) "apple"
4) "lion"
5) "monkey"
```

Redis Hashes

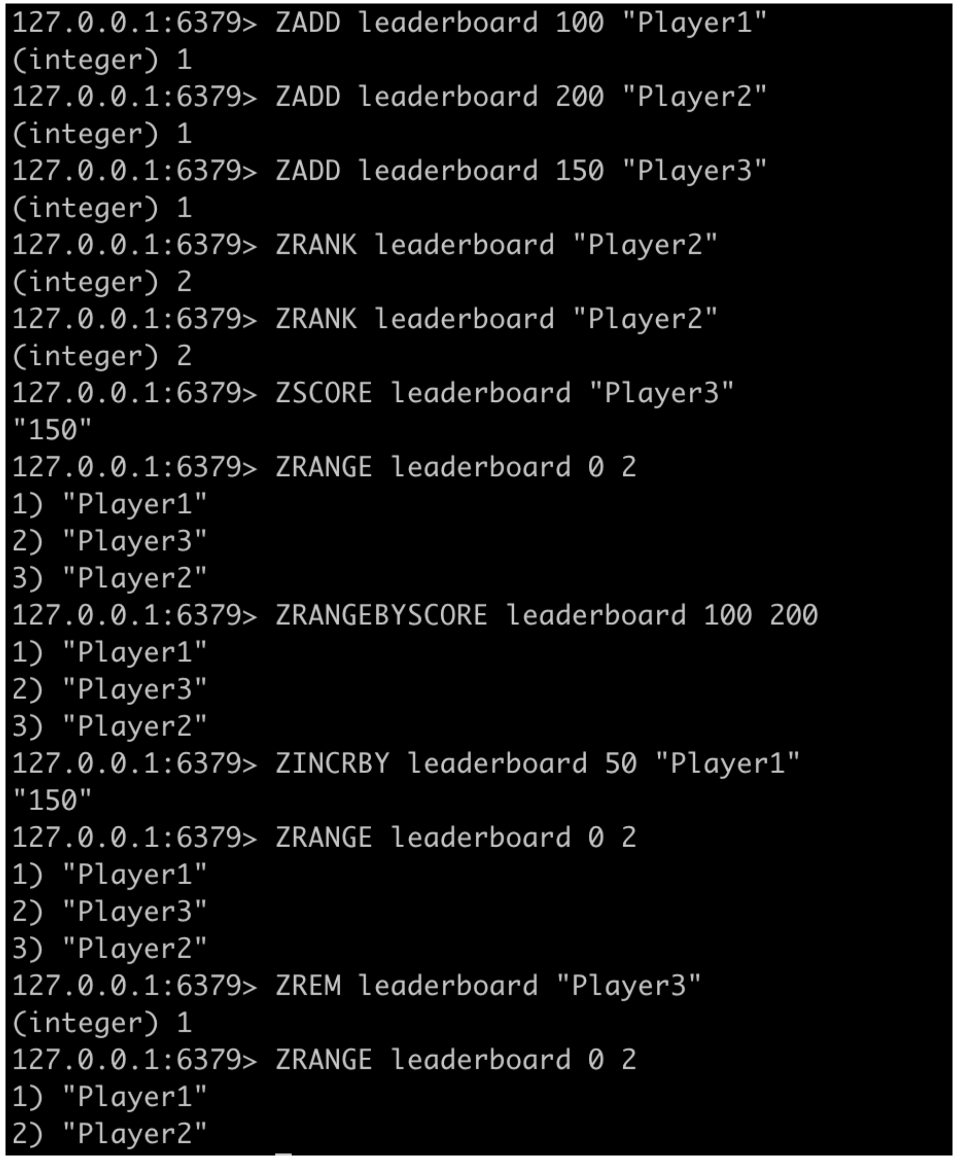
- **Hashes: Map between String Fields and Values**
  - Maps between string fields and string values.
  - Ideal for representing objects or data grouped together
  - Supports operations to set and get values, get all field-value pairs, etc.



```
127.0.0.1:6379> HMSET user:1 name "John Doe" age 30 email "john@example.com"
OK
127.0.0.1:6379> HGET user:1 name
"John Doe"
127.0.0.1:6379> HMSET user:2 name "Jason Jeong" age 40 email "jsjeong@wiken.co.kr"
OK
127.0.0.1:6379> HMSET user:3 name "Henry Park" age 50 email "hwpark@wiken.co.kr"
OK
127.0.0.1:6379> hgetall user:3
1) "name"
2) "Henry Park"
3) "age"
4) "50"
5) "email"
6) "hwpark@wiken.co.kr"
127.0.0.1:6379> hset user:3 age 35
(integer) 0
127.0.0.1:6379> HGET user:3 age
"35"
127.0.0.1:6379> hdel user:2 email
(integer) 1
```

Redis Sorted Sets

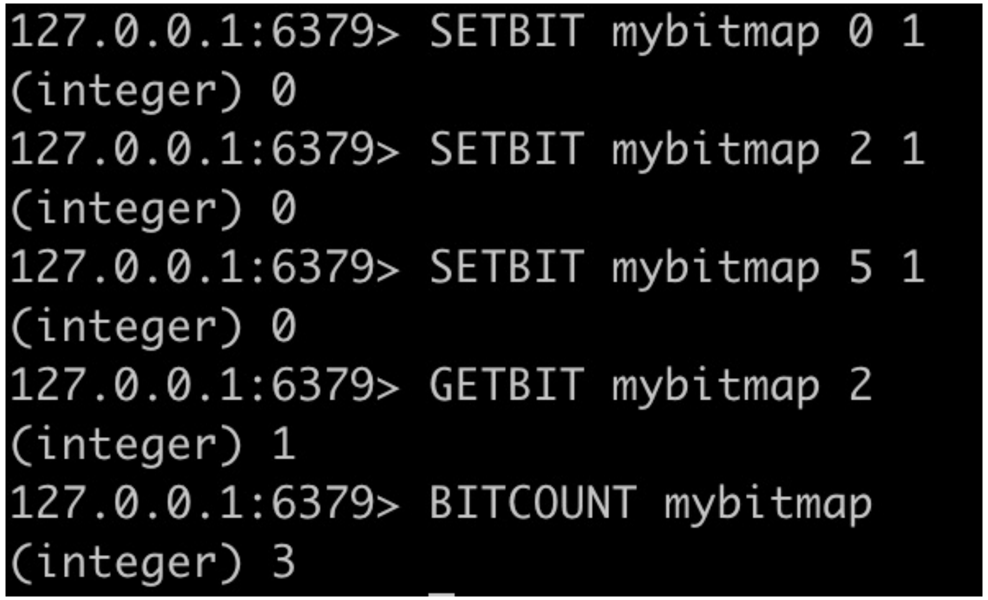
- **Sorted Sets: Strings with Associated Scores**
  - Similar to sets but every string element associated with a score.
  - Elements sorted by their scores.
  - Perfect for creating leaderboards, prioritized task lists, and more.



```
127.0.0.1:6379> ZADD leaderboard 100 "Player1"
(integer) 1
127.0.0.1:6379> ZADD leaderboard 200 "Player2"
(integer) 1
127.0.0.1:6379> ZADD leaderboard 150 "Player3"
(integer) 1
127.0.0.1:6379> ZRANK leaderboard "Player2"
(integer) 2
127.0.0.1:6379> ZRANK leaderboard "Player2"
(integer) 2
127.0.0.1:6379> ZSCORE leaderboard "Player3"
"150"
127.0.0.1:6379> ZRANGE leaderboard 0 2
1) "Player1"
2) "Player3"
3) "Player2"
127.0.0.1:6379> ZRANGEBYSCORE leaderboard 100 200
1) "Player1"
2) "Player3"
3) "Player2"
127.0.0.1:6379> ZINCRBY leaderboard 50 "Player1"
"150"
127.0.0.1:6379> ZRANGE leaderboard 0 2
1) "Player1"
2) "Player3"
3) "Player2"
127.0.0.1:6379> ZREM leaderboard "Player3"
(integer) 1
127.0.0.1:6379> ZRANGE leaderboard 0 2
1) "Player1"
2) "Player2"
```

Bitmaps in Redis

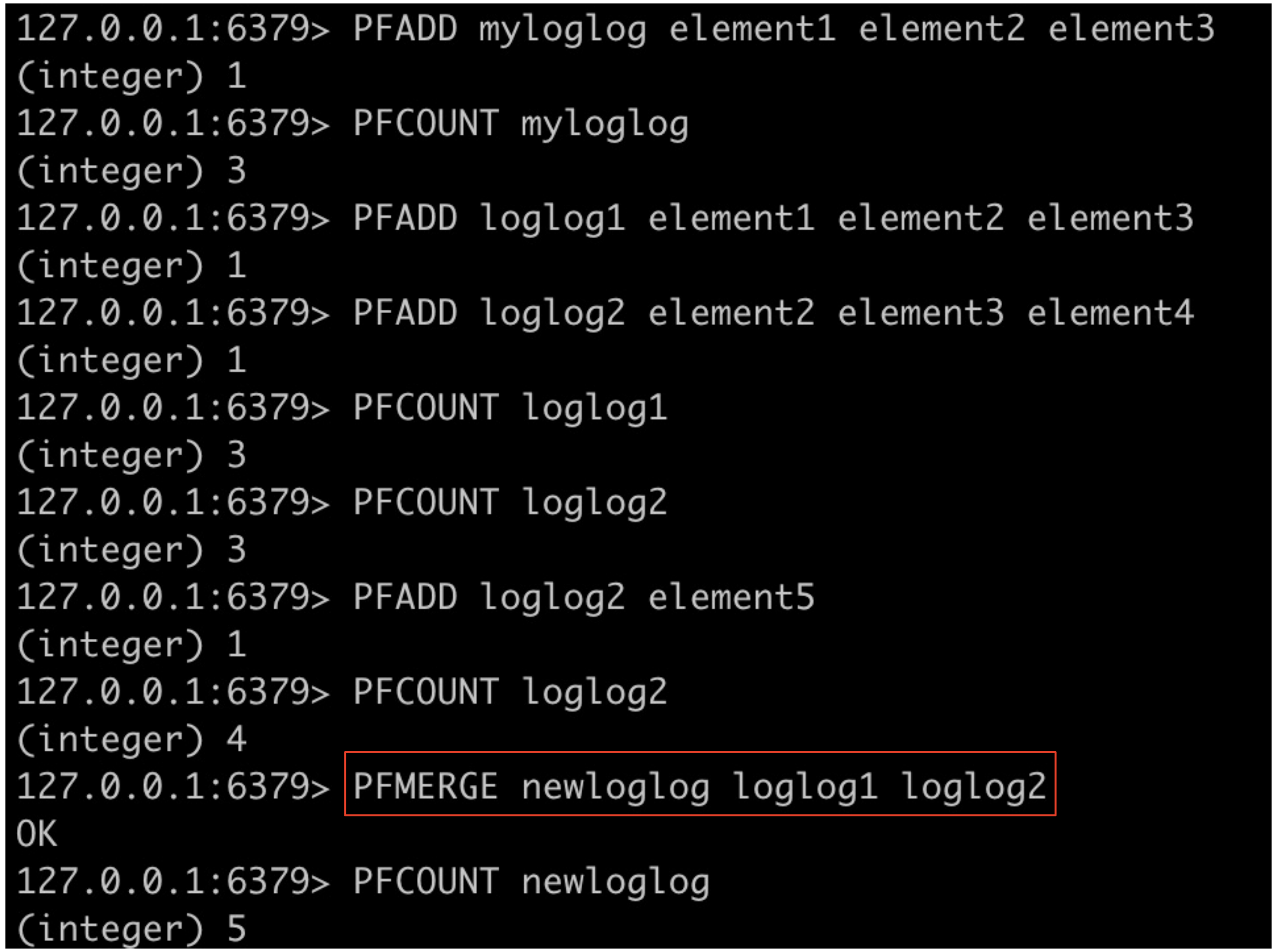
- **Bitmaps**:
  - A type of array where each element holds a bit value (0 or 1).
  - Ideal for tracking activities such as website visits, user logins, etc.
  - Efficient in storage and capable of performing bitwise operations.



```
127.0.0.1:6379> SETBIT mybitmap 0 1
(integer) 0
127.0.0.1:6379> SETBIT mybitmap 2 1
(integer) 0
127.0.0.1:6379> SETBIT mybitmap 5 1
(integer) 0
127.0.0.1:6379> GETBIT mybitmap 2
(integer) 1
127.0.0.1:6379> BITCOUNT mybitmap
(integer) 3
```

HyperLogLogs in Redis

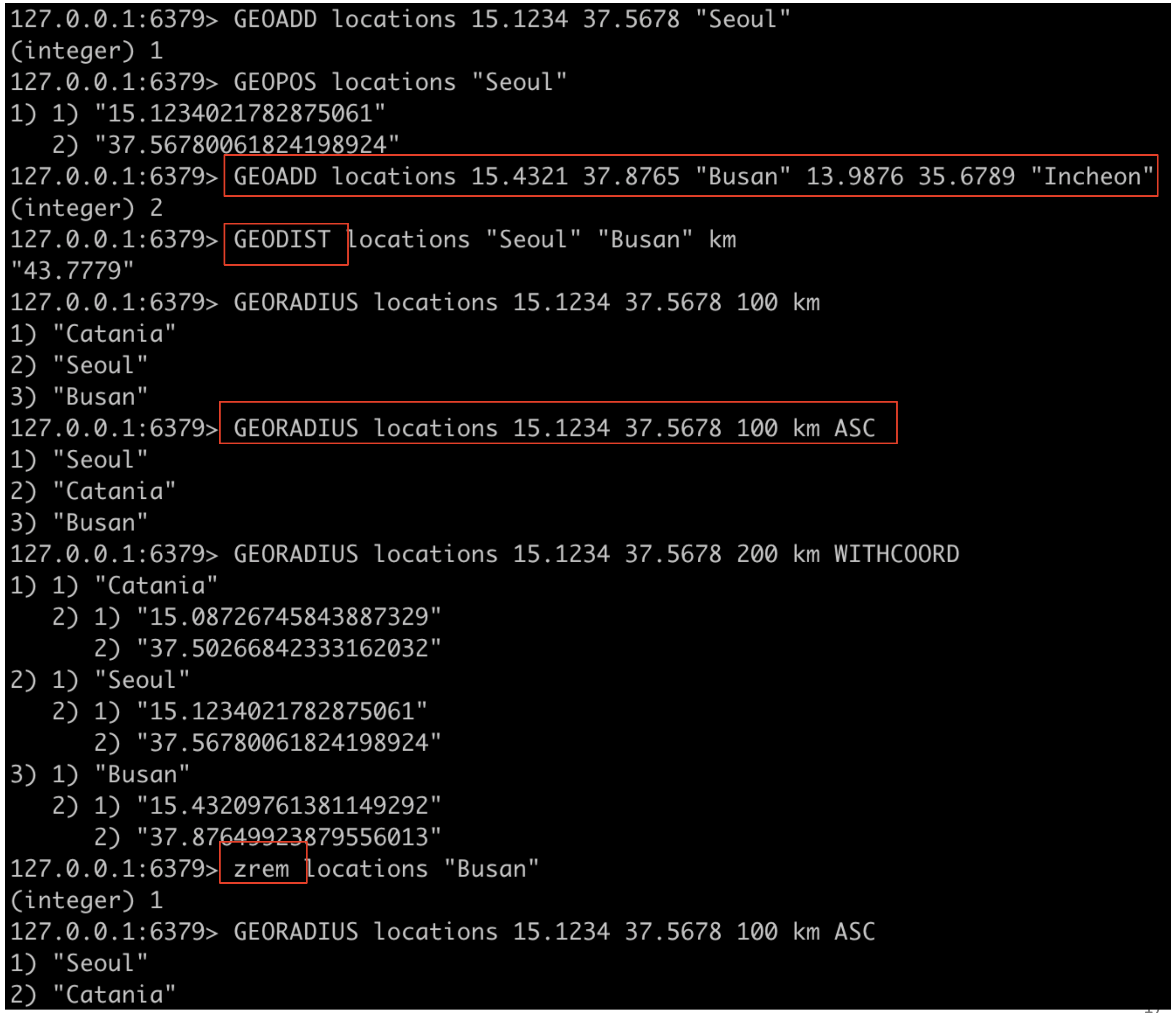
- **HyperLogLogs**:
  - Advanced probabilistic data structure.
  - Used to count unique elements in a set.
  - Especially useful for large data sets, as it uses constant storage space.



```
127.0.0.1:6379> PFADD myloglog element1 element2 element3
(integer) 1
127.0.0.1:6379> PFCOUNT myloglog
(integer) 3
127.0.0.1:6379> PFADD loglog1 element1 element2 element3
(integer) 1
127.0.0.1:6379> PFADD loglog2 element2 element3 element4
(integer) 1
127.0.0.1:6379> PFCOUNT loglog1
(integer) 3
127.0.0.1:6379> PFCOUNT loglog2
(integer) 3
127.0.0.1:6379> PFADD loglog2 element5
(integer) 1
127.0.0.1:6379> PFCOUNT loglog2
(integer) 4
127.0.0.1:6379> PFMERGE newloglog loglog1 loglog2
OK
127.0.0.1:6379> PFCOUNT newloglog
(integer) 5
```

Geospatial Indexes in Redis

- **Geospatial Indexes**:
  - Used to index geospatial data (longitude, latitude coordinates).
  - Supports queries such as the distance between two points and finding nearby locations.
  - Built on top of sorted sets, with the score being the geographic position.
  - Ideal for creating location-based services such as 'find nearby restaurants', 'track user location', etc.



```
127.0.0.1:6379> GEOADD locations 15.1234 37.5678 "Seoul"
(integer) 1
127.0.0.1:6379> GEOPOS locations "Seoul"
1) 1) "15.1234021782875061"
   2) "37.5678006182419892 4"
127.0.0.1:6379> GEOADD locations 15.4321 37.8765 "Busan" 13.9876 35.6789 "Incheon"
(integer) 2
127.0.0.1:6379> GEODIST locations "Seoul" "Busan" km
"43.7779"
127.0.0.1:6379> GEORADIUS locations 15.1234 37.5678 100 km
1) "Catania"
2) "Seoul"
3) "Busan"
127.0.0.1:6379> GEORADIUS locations 15.1234 37.5678 100 km ASC
1) "Seoul"
2) "Catania"
3) "Busan"
127.0.0.1:6379> GEORADIUS locations 15.1234 37.5678 200 km WITHCOORD
1) 1) "Catania"
   2) 1) "15.08726745843887329"
      2) "37.50266842333162032"
2) 1) "Seoul"
   2) 1) "15.1234021782875061"
      2) "37.5678006182419892 4"
3) 1) "Busan"
   2) 1) "15.43209761381149292"
      2) "37.87649923879556013"
127.0.0.1:6379> zrem locations "Busan"
(integer) 1
127.0.0.1:6379> GEORADIUS locations 15.1234 37.5678 100 km ASC
1) "Seoul"
2) "Catania"
```

Redis Caching

Introduction to Redis Caching

- Redis caching: A high-performance, in-memory data store
- Role of caching:
  - Improves application performance by reducing latency
  - Minimizes the load on the backend database
  - Enhances scalability and responsiveness
  - Optimizes resource utilization
  - Enables faster retrieval of frequently accessed data
  - Facilitates efficient handling of high traffic and concurrent requests
  - Supports real-time applications and dynamic content delivery
  - Mitigates the impact of network latency and data processing overhead
  - Provides a cost-effective solution for managing data access
  - Enables seamless integration with various caching patterns and strategies

Redis-backed Code for Caching

- Implementing Redis-backed code for caching frequently accessed data
- Benefits of using Redis for caching:
  - Faster data retrieval from memory compared to querying a database
  - Reduced network latency and improved application responsiveness
  - Offloads the load on the backend database, improving its performance
- Sample code snippets for caching MongoDB query results using Redis:
  - Connect to Redis server and configure caching settings
  - Check if data exists in Redis cache, if not, fetch from MongoDB and store in Redis
  - Retrieve data from Redis cache if available, otherwise query MongoDB and cache the result

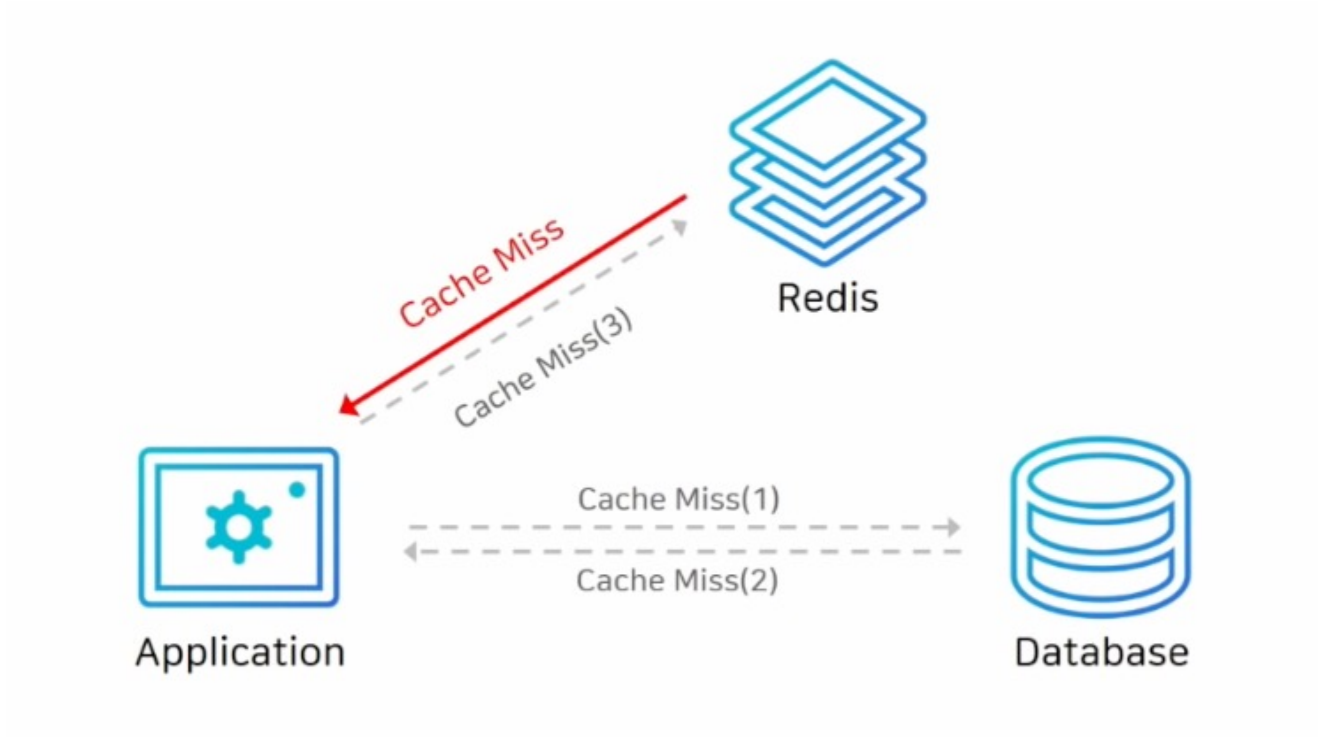
Redis Data Structures for Caching

- Overview of Redis data structures suitable for caching:
  - Strings: Store simple key-value pairs for single values or serialized objects
  - Hashes: Ideal for storing and retrieving complex data structures as key-value pairs
  - Sets: Efficient for storing and manipulating unique values

- Choosing the appropriate data structure based on caching requirements:
  - Strings for simple key-value caching
  - Hashes for caching structured data or multiple attributes
  - Sets for caching unique values or managing sets of related data
  - Consider the data size, complexity, and access patterns when selecting a data structure

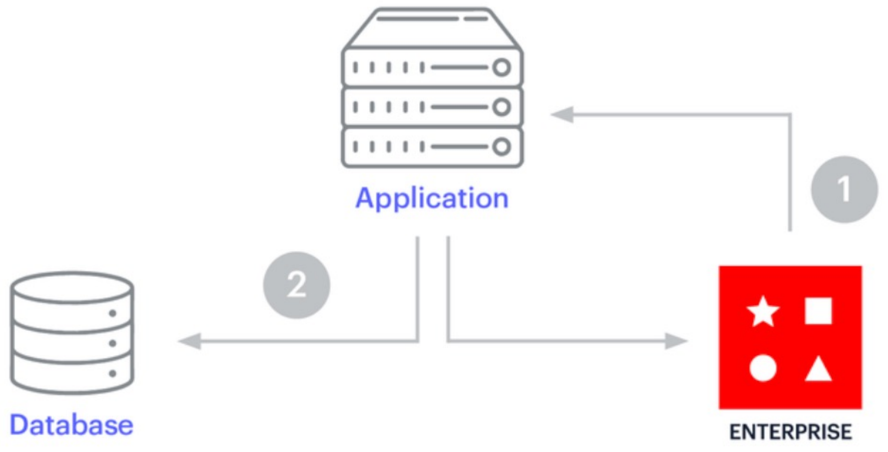
Caching Strategies

- Read
  - Look-Aside (Lazy Loading)



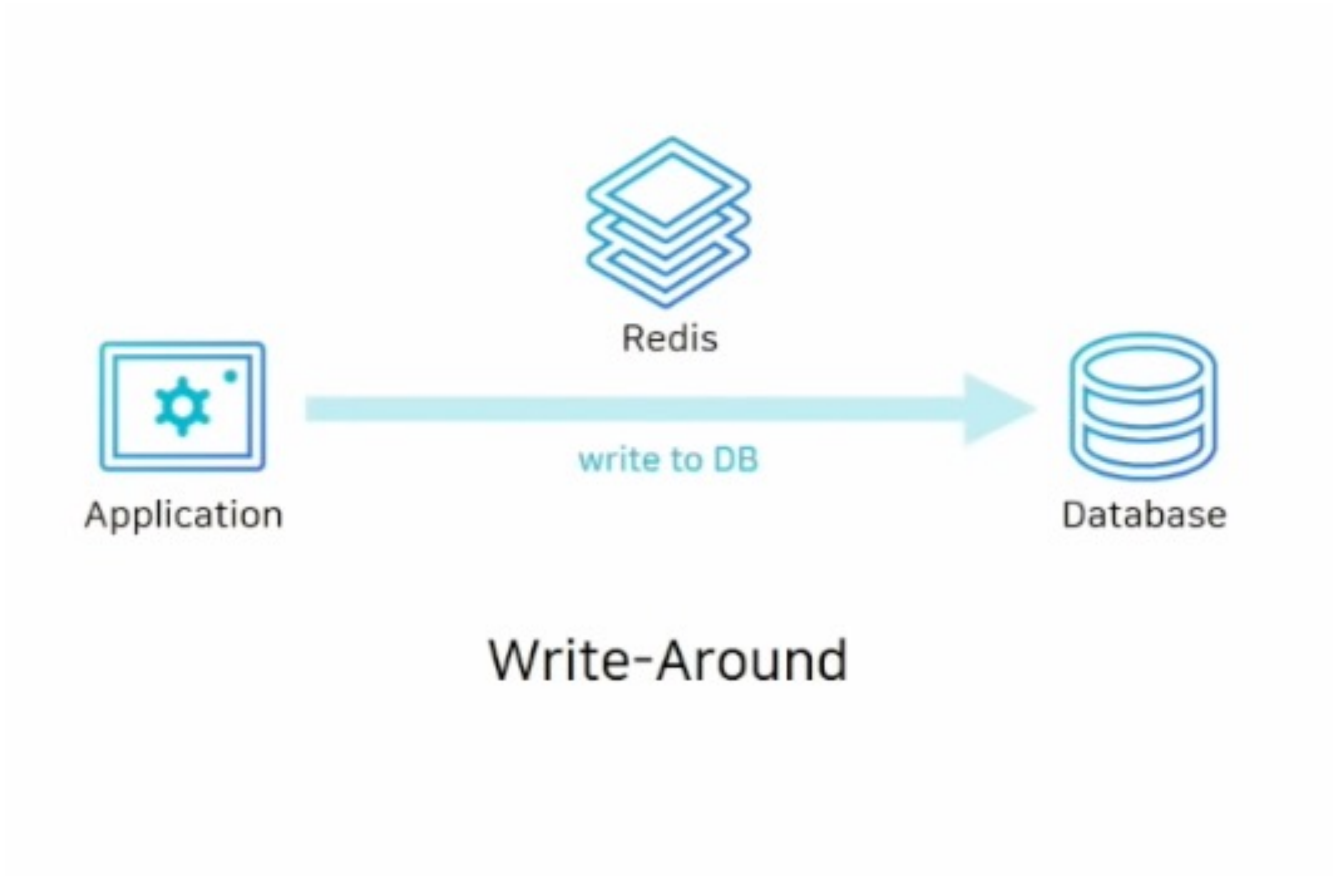
Caching Patterns with Redis

- Cache-Aside Pattern:
  - Application first checks if data exists in the cache
  - If data is found, it is returned to the application
  - If data is not found, application retrieves data from the source (e.g., database), stores it in the cache, and then returns it to the application



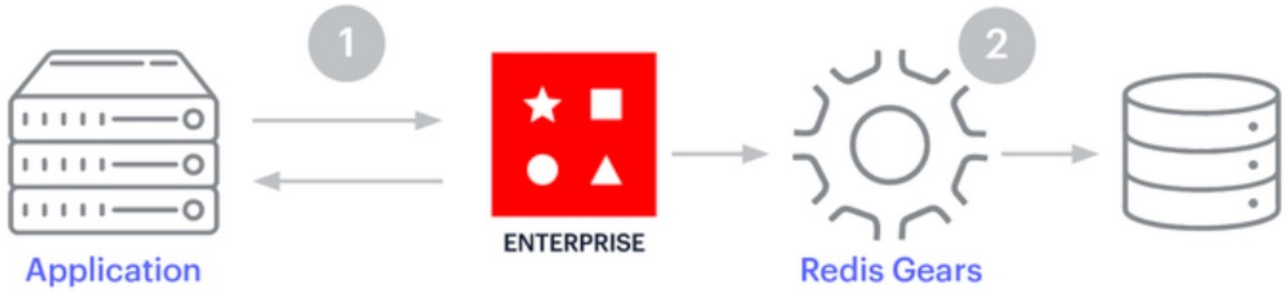
Caching Strategies

- Write
  - Write-Around



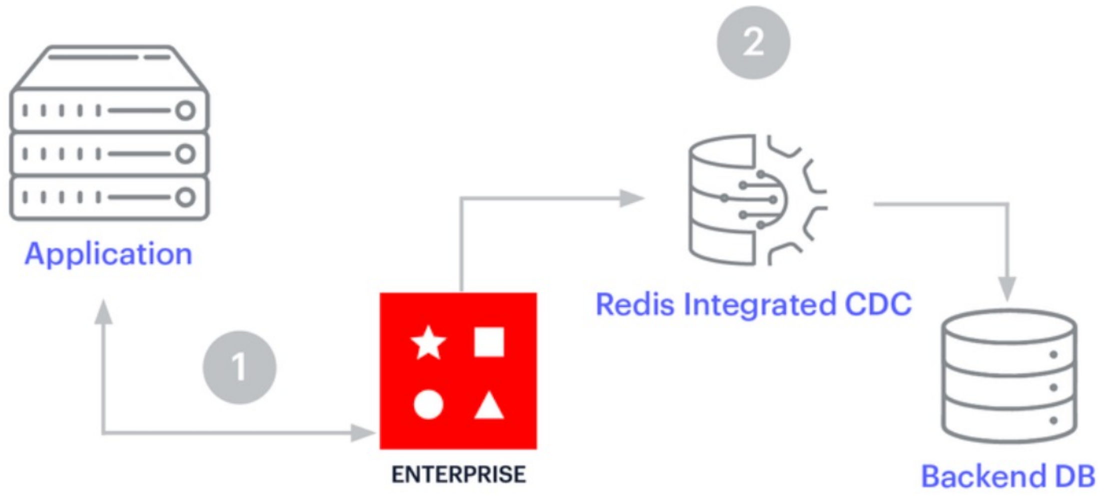
Caching Patterns with Redis

- Write-Through Pattern:
  - Application writes data to the cache and the underlying data store simultaneously
  - Data in the cache remains consistent with the underlying data store



Caching Patterns with Redis

- Write-Back Pattern:
  - Application writes data to the cache and asynchronously updates the underlying data store
  - Pros: Reduced write latency, improved application responsiveness
  - Cons: Potential data inconsistency between cache and data store in case of failure or system crash



Benefits of Redis Caching

- Improved Application Performance:
  - Faster data retrieval from in-memory cache
  - Reduced load on the backend database
  - Lower network latency, improved application responsiveness

- Scalability:
  - Redis supports distributed caching, allowing you to scale horizontally by adding more cache nodes
  - Provides high throughput and low latency even with large datasets

Benefits of Redis Caching

- Enhanced User Experience:
  - Reduced response times and improved page load speeds
  - Smoother user interactions and faster data access
- Cost Savings:
  - Caching reduces the need for expensive database queries, resulting in cost savings on database resources
- Flexible Caching Strategies:
  - Redis offers various caching patterns and data structures to accommodate different use cases and requirements

Cache Expiration and Eviction Policies

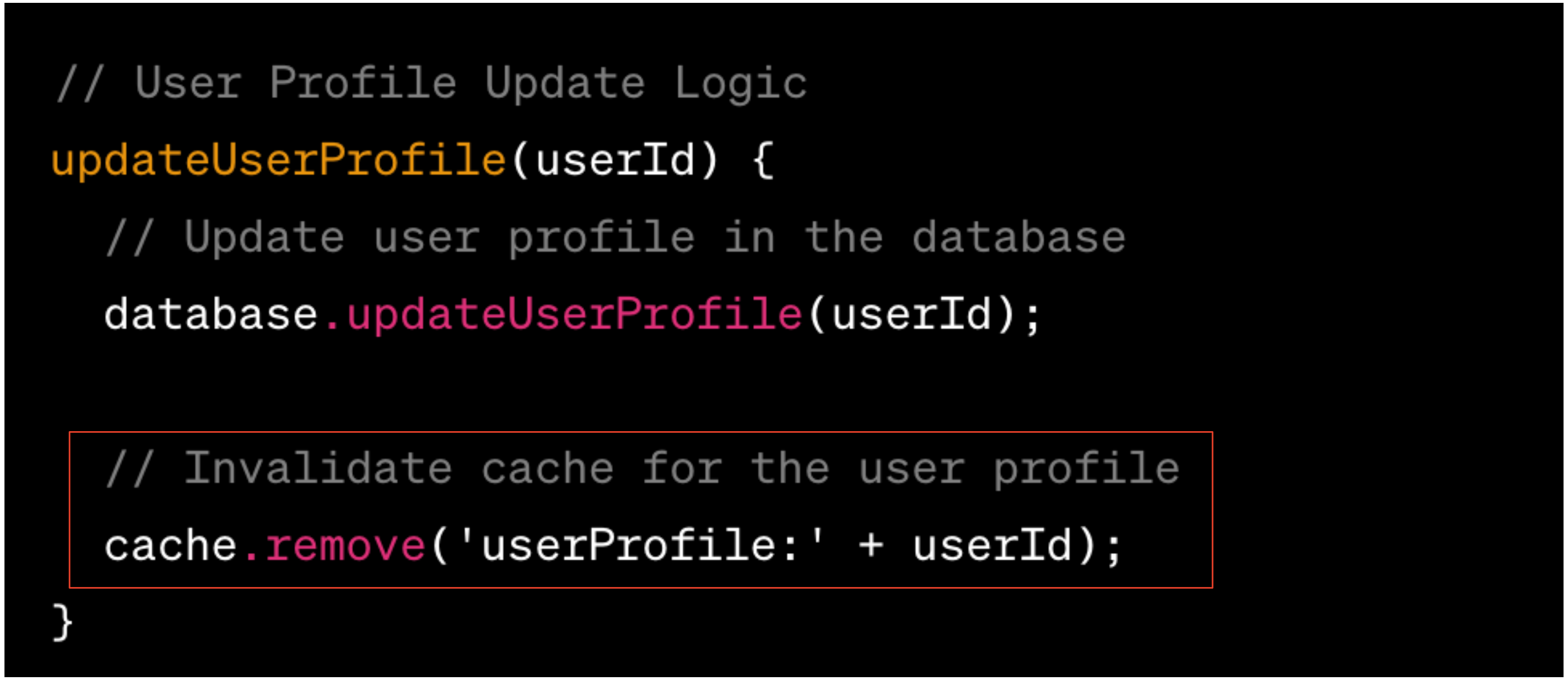
- Cache Expiration:
  - Redis allows setting expiration time for cache entries
  - Expiration time can be specified when storing the data in the cache
  - After the expiration time elapses, Redis automatically removes the entry from the cache
  - Useful for managing cache freshness and avoiding stale data
- Eviction Policies:
  - When the cache reaches its maximum capacity, Redis uses eviction policies to determine which entries to remove
  - Common eviction policies include LRU (Least Recently Used), LFU (Least Frequently Used), and Random
  - Eviction policies help maintain cache efficiency and make room for new data

Cache Expiration and Eviction Policies

- Least Recently Used (LRU):
  - Evicts the least recently accessed cache entries first
  - Suitable for scenarios where recently accessed data is more likely to be accessed again in the near future
- Least Frequently Used (LFU):
  - Evicts the least frequently accessed cache entries first
  - Suitable for scenarios where data with low access frequency can be removed to make space for frequently accessed data
- Random:
  - Evicts cache entries randomly without any specific criteria
  - Simple and easy to implement, but may not be optimal in terms of cache efficiency

Cache Invalidation Strategies

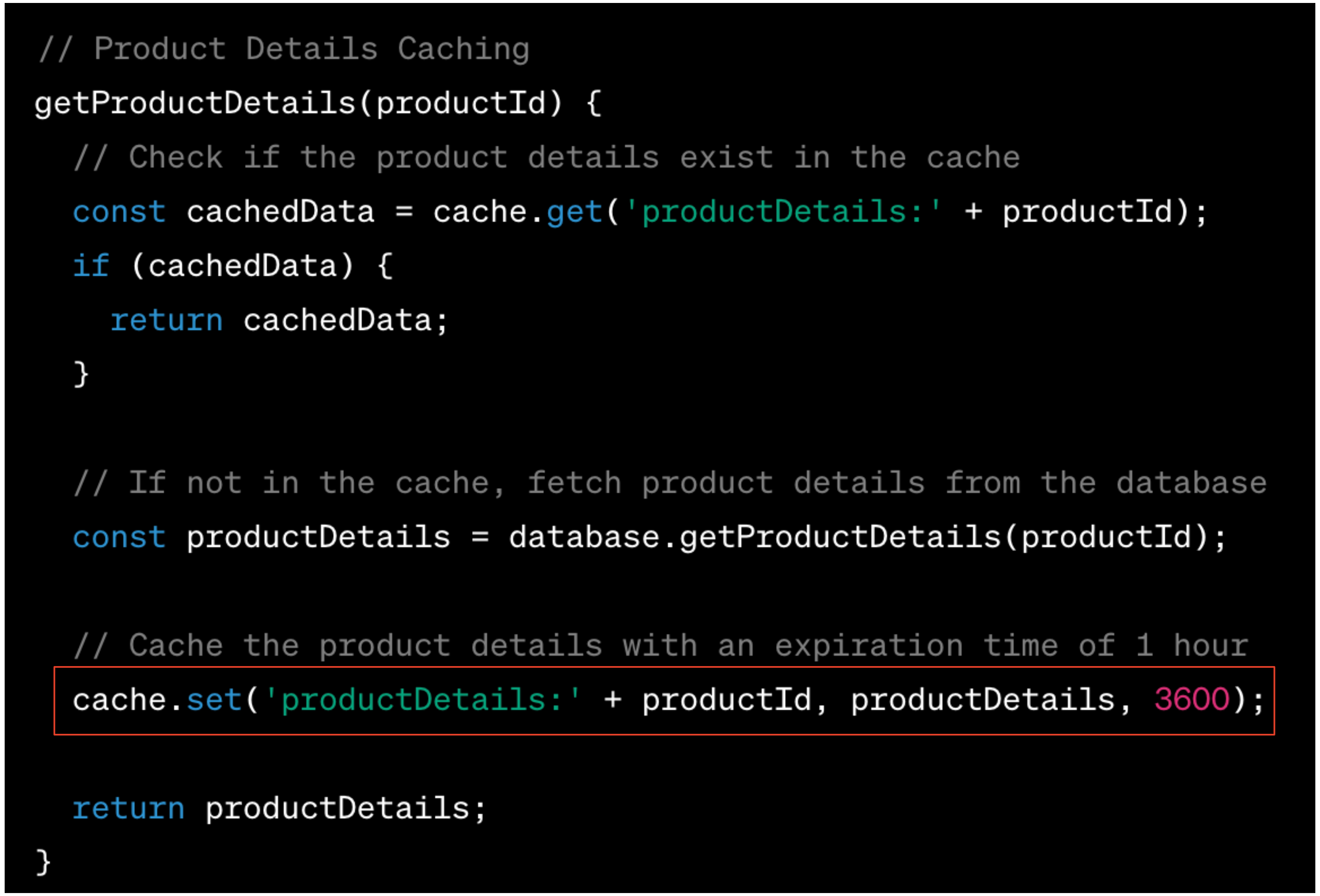
- Cache Invalidation:
  - Remove or update cached data when it becomes stale or invalid
  - Ensures up-to-date and accurate data for users



```
// User Profile Update Logic
updateUserProfile(userId) {
  // Update user profile in the database
  database.updateUserProfile(userId);

  // Invalidate cache for the user profile
  cache.remove('userProfile:' + userId);
}
```

Cache Invalidation Strategies

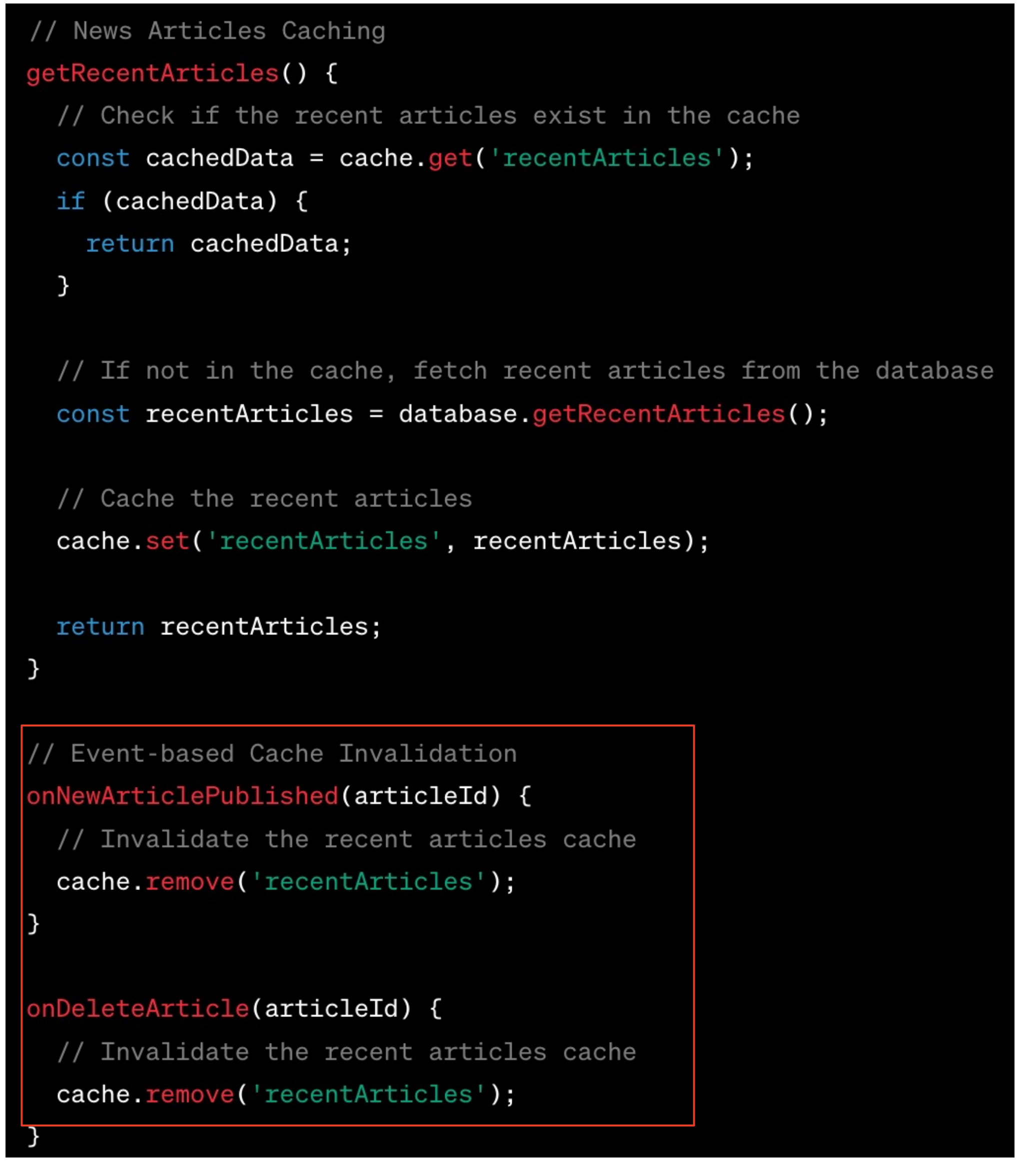
- Time-Based Invalidation:
  - Set fixed expiration time for cached data
  - Data becomes invalid after expiration, needs refreshing
  - Simple implementation but may serve stale data until expiration



```
// Product Details Caching
getProductDetails(productId) {
  // Check if the product details exist in the cache
  const cachedData = cache.get('productDetails:' + productId);
  if (cachedData) {
    return cachedData;
  }

  // If not in the cache, fetch product details from the database
  const productDetails = database.getProductDetails(productId);

  // Cache the product details with an expiration time of 1 hour
  cache.set('productDetails:' + productId, productDetails, 3600);

  return productDetails;
}
```

Cache Invalidation Strategies

- Event-Based Invalidation:
  - Invalidate cache based on data source events (updates, inserts, deletes)
  - Remove cache entry when corresponding event occurs
  - Keeps cache synchronized with data source



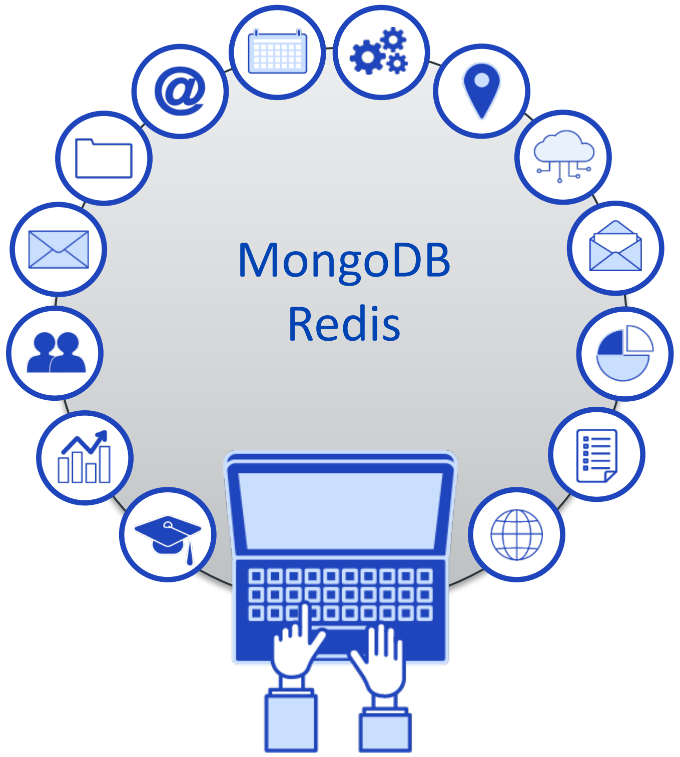
```
// News Articles Caching
getRecentArticles() {
  // Check if the recent articles exist in the cache
  const cachedData = cache.get('recentArticles');
  if (cachedData) {
    return cachedData;
  }

  // If not in the cache, fetch recent articles from the database
  const recentArticles = database.getRecentArticles();

  // Cache the recent articles
  cache.set('recentArticles', recentArticles);

  return recentArticles;
}

// Event-based Cache Invalidation
onNewArticlePublished(articleId) {
  // Invalidate the recent articles cache
  cache.remove('recentArticles');
}

onDeleteArticle(articleId) {
  // Invalidate the recent articles cache
  cache.remove('recentArticles');
}
```

Cache Invalidation Strategies

- Manual Invalidation:
  - Programmatically invalidate cache entries based on application logic
  - Fine-grained control over invalidation process
  - Requires careful tracking and management of changes
- Combination Strategies:
  - Use a mix of time-based and event-based invalidation
  - Time-based for general expiration, event-based for immediate updates
  - Balances freshness and efficiency
- Considerations for Cache Invalidation:
  - Choose strategy based on data volatility and application requirements
  - Understand trade-offs between cache freshness, performance, and complexity
  - Regularly evaluate and fine-tune invalidation for optimal cache utilization

Lab

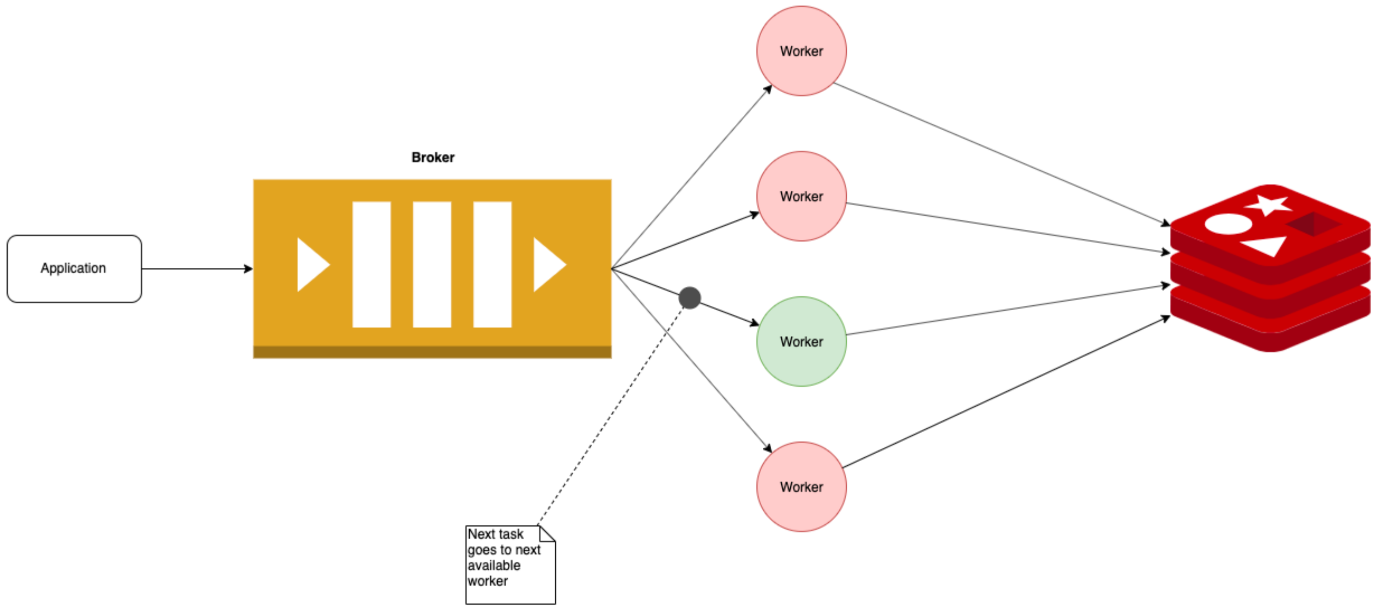
Redis Caching



Using Redis for Task Queues & Leaderboards

Distributed Task Queues

- Redis as a Distributed Task Queue:
  - Redis provides powerful features for building distributed task queues
  - It offers efficient data structures and operations for managing tasks and workers



Key Features of Redis Task Queues

- Atomic Operations:
  - Redis supports atomic operations, ensuring that task queue operations are performed as a single, indivisible unit.

- Blocking Operations:
  - Redis provides blocking list operations, allowing efficient task retrieval by blocking until a task is available.

- Priority Queue Support:
  - Redis' sorted sets enable the implementation of priority-based task queues, allowing tasks to be processed based on their priority.

Key Features of Redis Task Queues

- Delayed Task Execution:
  - Redis sorted sets combined with timestamps enable the scheduling of tasks for future execution, allowing for delayed task processing.
- Persistent Storage:
  - Redis offers persistence options such as RDB snapshots and AOF logs, ensuring that task queues can be restored after system restarts or failures.
- Scalability and High Availability:
  - Redis Cluster allows for horizontal scaling and distribution of task queues across multiple Redis instances, ensuring scalability and high availability.

Task Queue Security and Access Control

- Access Control:
  - Implement authentication and authorization mechanisms to secure access to Redis and the task queue
  - Use Redis ACL (Access Control List) or external authentication systems to restrict access based on roles and permissions
- Data Encryption:
  - Encrypt sensitive data such as task payloads or task results stored in Redis
  - Utilize Redis modules or external encryption mechanisms to ensure data confidentiality
- Rate Limiting:
  - Apply rate limiting techniques to prevent abuse or excessive task submissions
  - Redis can be used to enforce rate limits by tracking and throttling task queue operations
- Monitoring for Security Threats:
  - Regularly monitor Redis logs and employ security tools to detect and respond to potential security threats

Scalability and High Availability

- Horizontal Scaling:
  - Redis task queues can be horizontally scaled by adding more Redis instances or using Redis Cluster
  - Distribute the workload across multiple Redis nodes to handle increased task processing requirements
- High Availability:
  - Implement Redis replication and failover mechanisms to ensure high availability of the task queue
  - Use Redis Sentinel or Redis Cluster to maintain a fault-tolerant and highly available task queue infrastructure
- Load Balancing:
  - Employ load balancing techniques to evenly distribute task processing across multiple worker instances
  - Load balancing ensures efficient utilization of resources and prevents bottlenecks in task execution
- Monitoring and Performance Optimization:
  - Continuously monitor the performance and resource usage of the Redis task queue

Task Queue Patterns

- Simple Queue Pattern:
  - Tasks are added to the queue using the LPUSH command.
  - Workers fetch tasks from the queue using the BRPOP command, which blocks until a task is available.
  - Suitable for simple task processing where tasks are processed in the order they were added.

- Priority Queue Pattern:
  - Process tasks based on priority levels
  - Implement a priority queue using Redis Sorted Sets to ensure high-priority tasks are processed first

- Delayed Tasks Pattern:
  - Schedule tasks to be executed at a later time
  - Use Redis Sorted Sets or Redis Streams to store and manage delayed tasks

Task Queue Patterns

- Round-robin Queue Pattern:
  - Multiple queues are created, and tasks are distributed among these queues in a round-robin fashion.
  - Workers fetch tasks from the queues in a round-robin manner, ensuring fair distribution of tasks.
  - Suitable for load balancing and distributing tasks across multiple workers.

- Workflow Pattern:
  - Define and manage complex workflows involving multiple sequential or parallel tasks
  - Use Redis data structures, such as Redis Lists or Redis Hashes, to model and track the workflow stages

Fault Tolerance and Reliability

- Replication:
  - Use Redis replication to create replicas of the primary Redis instance for data redundancy and high availability
  - Ensure data durability and quick recovery in case of failures

- Data Persistence:
  - Choose the appropriate persistence strategy based on the trade-offs between performance and data safety
  - Use Redis persistence mechanisms like RDB snapshots or AOF logs for data persistence

Fault Tolerance and Reliability

- Backup and Disaster Recovery:
  - Implement backup and disaster recovery strategies to protect against data loss and enable quick recovery
  - Regularly backup Redis data and store backups in secure off-site locations
- Handling Failures:
  - Handle network failures, Redis crashes, or worker failures gracefully
  - Implement retry mechanisms, error handling, and fault-tolerant strategies in the application code
- Monitoring Failures:
  - Monitor Redis cluster health, replication status, and connectivity
  - Implement automated monitoring systems to detect failures and trigger alerts for prompt resolution

Error Handling and Retry Strategies

- Error Handling:
  - Implement error handling mechanisms to capture and handle exceptions or failures during task execution
  - Log errors for troubleshooting and analysis purposes
  - Define appropriate error handling strategies based on the specific use case and requirements
- Retry Strategies:
  - Configure retry mechanisms to handle failed tasks and retries for temporary errors
  - Define retry intervals and limits to avoid continuous retries and potential system overload
  - Implement exponential backoff strategies to gradually increase retry intervals
- Dead Letter Queue:
  - Set up a dead letter queue to capture failed tasks that have exceeded the retry limit
  - Analyze and investigate the causes of failures from the dead letter queue for resolution or further action

Advanced Features and Optimization

- Priority Queue:
  - Utilize Redis' Sorted Sets to implement a priority queue for task scheduling
  - Assign priority levels to tasks and process them in the order of their priority
- Delayed Queue:
  - Implement a delayed queue using Redis' Sorted Sets and the timestamp of task execution
  - Schedule tasks to be executed at a future time or after a specific delay
- Rate Limiting:
  - Use Redis' data structures and atomic operations to enforce rate limits on task processing
  - Prevent excessive task execution and control the flow of requests
- Pub/Sub Messaging:
  - Leverage Redis' Pub/Sub functionality for real-time communication between components of the task processing system
  - Publish messages to notify subscribers about task updates or events

Introduction to Redis for Leaderboards

- Redis as a Leaderboard Store:
  - Redis provides efficient data structures and operations for building high-performance leaderboards.
  - Leaderboards are commonly used in gaming, sports, and social applications to track and rank players or participants based on scores or other metrics.

Key Features of Redis Leaderboards

- Sorted Sets:
  - Redis provides the Sorted Set data structure for efficient leaderboard implementation.
  - Sorted Sets automatically sort elements based on their scores.
  - Scores can represent various metrics, such as points, scores, or rankings.
- Atomic Operations:
  - Redis supports atomic operations, ensuring consistent and reliable leaderboard updates.
  - Atomic operations enable concurrent access and modification of leaderboard data without conflicts.
  - Common atomic operations include ZADD (add/update scores), ZINCRBY (increment scores), and ZREMRANGEBYRANK (remove elements by rank).
- Range Queries:
  - Redis Sorted Sets allow performing range queries on leaderboards.
  - Range queries enable retrieving a specified number of elements within a specific score range.
  - Range queries facilitate leaderboard pagination and fetching top performers.

Key Features of Redis Leaderboards

- Real-time Updates:
  - Redis is known for its high-performance and real-time capabilities.
  - Leaderboards can be updated in real-time by integrating Redis with other systems, such as game servers or event-driven architectures.
  - Real-time updates provide an up-to-date leaderboard experience for users.
- Persistence Options:
  - Redis offers persistence options, such as RDB snapshots and AOF logs, ensuring leaderboard data durability.
  - Persistence mechanisms prevent data loss and allow leaderboard recovery in case of system failures.
- Scalability and Performance:
  - Redis is designed to handle high-throughput workloads, making it suitable for large-scale leaderboards.
  - Redis Cluster enables horizontal scaling by distributing data across multiple nodes.
  - In-memory storage and optimized data structures contribute to high-performance leaderboard operations.

Designing Leaderboards with Redis

- Factors to consider when designing a leaderboard:
  - Ranking Criteria:
    - Determine the criteria for ranking participants, such as scores, points, or ratings.
    - Define how the ranking algorithm should consider different metrics.
  - Update Frequency:
    - Determine how frequently leaderboard data needs to be updated.
    - Consider the frequency of updates to ensure real-time or near-real-time leaderboard updates.
  - Scalability:
    - Assess the expected number of participants and the potential growth of the leaderboard.
    - Plan for scalability by considering the distribution of leaderboard data across Redis nodes.

Designing Leaderboards with Redis

- Redis features for designing leaderboards:
  - Sorted Sets:
    - Utilize Redis Sorted Sets to store and manage leaderboard data.
    - Sorted Sets automatically sort elements based on their scores, providing an efficient ranking mechanism.
  - Pub/Sub (Publish/Subscribe):
    - Use Redis Pub/Sub for real-time updates and notifications.
    - Pub/Sub enables subscribers to receive leaderboard updates as soon as they occur, ensuring real-time interactivity.
  - Redis Cluster:
    - Leverage Redis Cluster for horizontal scalability and distribution of leaderboard data.
    - Redis Cluster allows distributing data across multiple nodes, accommodating large-scale leaderboards.

Real-time Leaderboards with Redis

- Real-time Updates:
  - Redis Sorted Sets allow you to update scores and rankings in real-time as new scores are received.
  - Implement event-driven mechanisms or pub/sub functionality to notify interested clients about leaderboard updates.
- Time-based Leaderboards:
  - Create time-based leaderboards, such as daily, weekly, or monthly leaderboards, by using appropriate score calculation and expiration techniques.
  - Allow players to compete within specific time intervals and track their progress over time.

Leaderboard Queries and Operations

- Retrieving Ranks:
  - Use Redis commands to retrieve a participant's rank, score, and other related information.
  - Leverage the Sorted Set operations to perform efficient range queries.

- Top N Queries:
  - Fetch the top N participants based on score or ranking using Redis commands like ZRANGE or ZREVRANGE.
  - Utilize options like WITHSCORES to retrieve scores along with participant details.

How Sorted Sets can be used for leaderboards

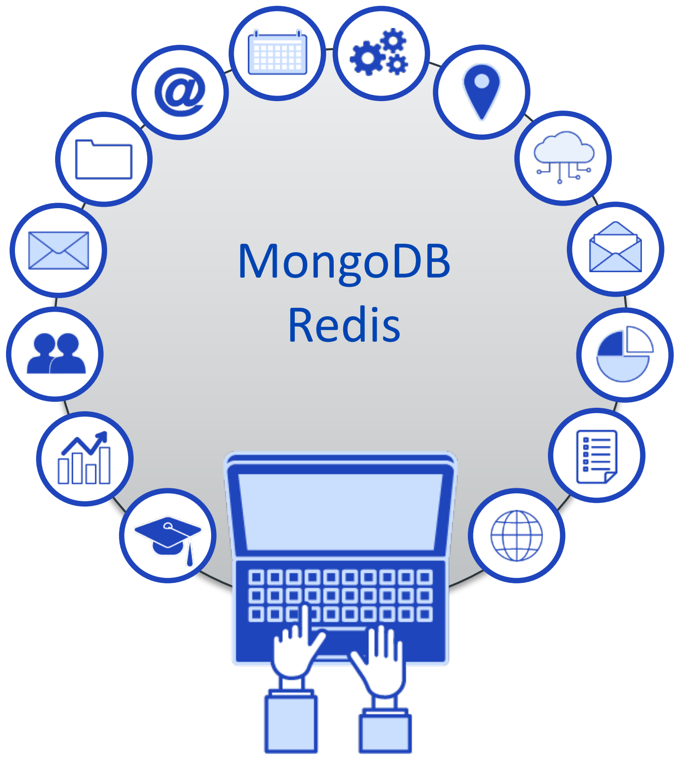
- Leaderboards require ranking participants based on their scores.
- Sorted Sets are ideal for leaderboards as they provide a natural way to store participants and their scores.
- The ZADD command is used to add participants and their scores to a Sorted Set.
- The ZRANK command allows you to find the rank of a participant based on their score.

Implementing Real-Time Leaderboards with Sorted Sets

- Step-by-step guide to implementing a real-time leaderboard using Sorted Sets:
  - Design the structure of the leaderboard using Redis Sorted Sets.
  - Use the ZADD command to add participants and their scores to the Sorted Set.
  - Use the ZRANGE command to retrieve the top participants based on their scores.
  - Update participant scores using the ZINCRBY command for real-time score adjustments.
  - Implement periodic leaderboard updates using a background process or scheduling mechanism.
  - Utilize Redis Pub/Sub to notify subscribers about leaderboard updates in real-time.

Lab

Redis Leaderboards



Redis Management

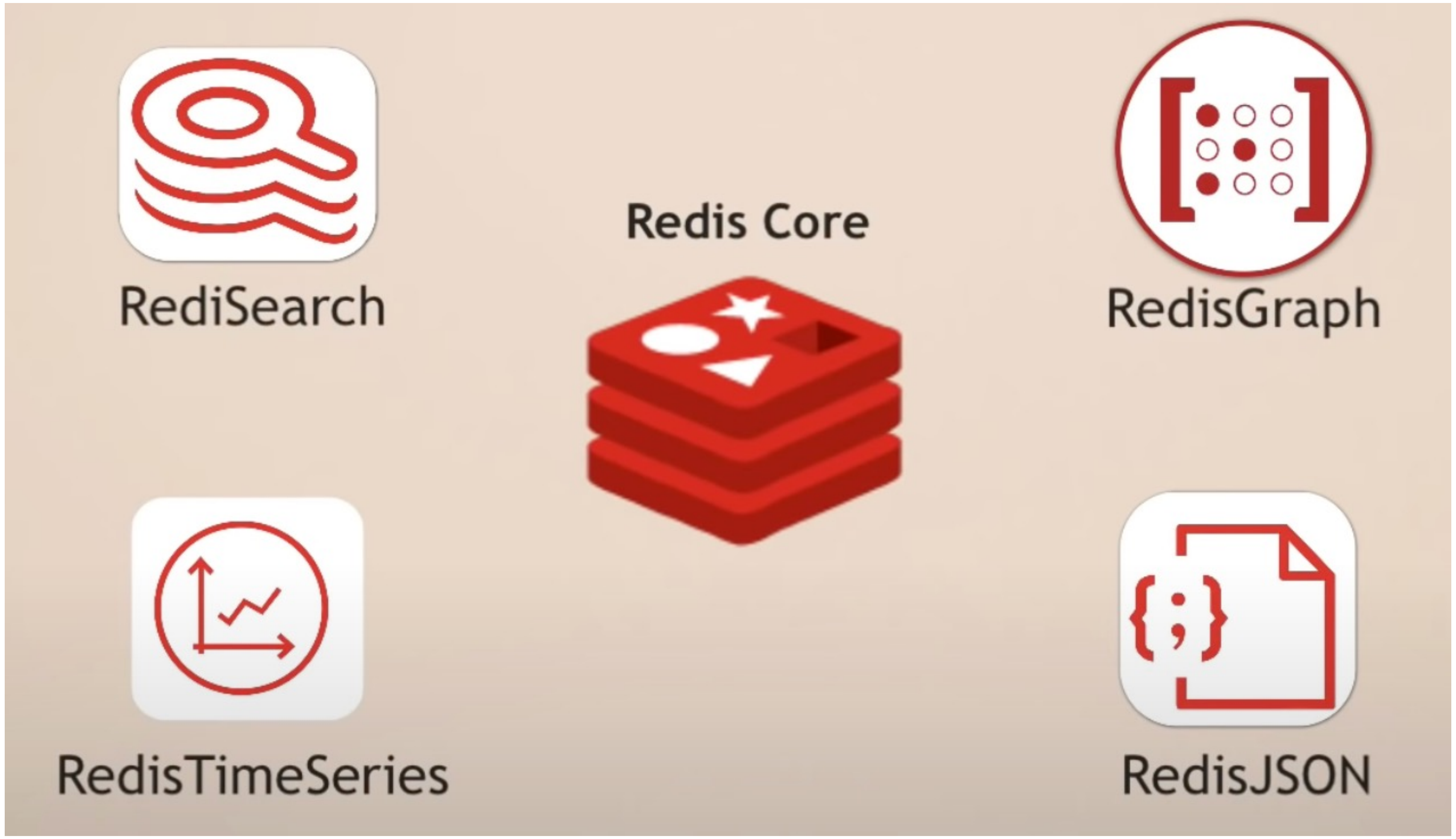
Data Persist

- RDB(Snapshotting):
  - Products single-file point-in-time snapshots of your dataset
  - Configure based on time or number of writes passed
  - Stored on disk

- AOF(Append Only File):
  - Logs every write operation continuously
  - When restarting Redis, it will re-play the AOF to rebuild the state
  - Much more durable

Extend with Redis Modules

- Redis Modules are add-on components that extend the functionality of Redis by introducing new data types, commands, and capabilities.



Integrating MongoDB and Redis

Use Cases for Integrating MongoDB and Redis

- Caching:
  - Discuss how Redis can be used as a caching layer for MongoDB to improve read performance and reduce database load.
  - Explore scenarios such as frequently accessed data, query results, or static data caching.
- Real-Time Analytics:
  - Explain how Redis can enhance real-time analytics by storing aggregated or pre-computed data from MongoDB, allowing for faster query responses.
- Session Management:
  - Illustrate how Redis can be used to manage session data, such as user authentication tokens, improving performance and scalability.

Why Use Redis as a Caching Layer

- **Improved Performance**: Redis stores data in-memory, enabling faster data retrieval compared to disk-based databases like MongoDB.
- **Reduced Database Load**: By caching frequently accessed data in Redis, the number of queries and requests to the underlying MongoDB database is reduced.
- **Efficient Data Retrieval**: Redis offers advanced data structures and operations that allow for efficient caching strategies. Features like key expiration, data eviction policies, and atomic operations contribute to optimized data retrieval and management.
- **Flexibility and Versatility**: Redis can be used as a versatile caching layer, supporting various data types and complex data structures. This makes it suitable for a wide range of caching use cases, including session caching, result caching, and object caching.
- **Seamless Integration**: Redis can be easily integrated with existing MongoDB deployments without requiring significant changes to the application code.

Redis as a Caching Layer for MongoDB Data

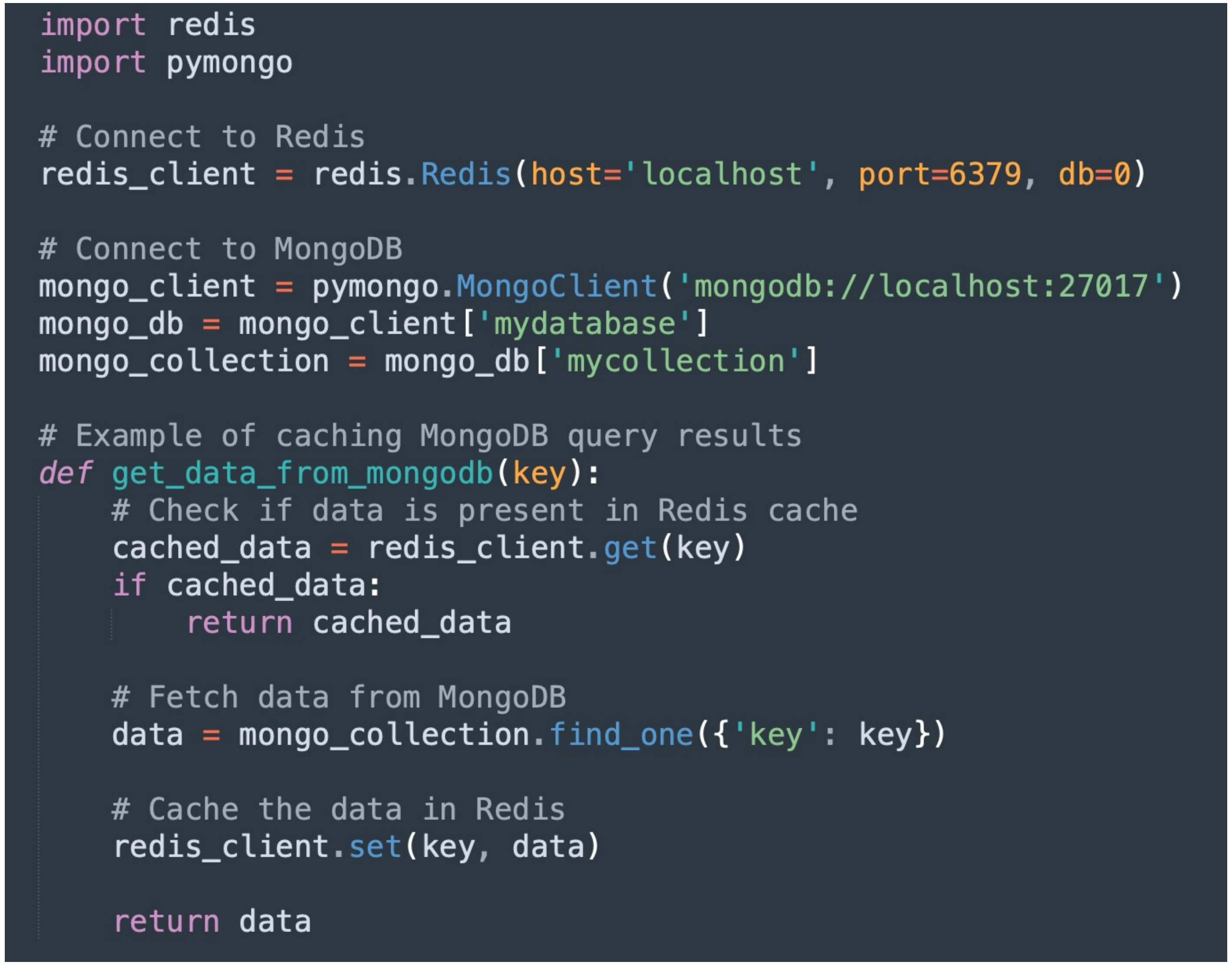
- Caching Strategies:
  - Explain different caching strategies, such as cache-aside and write-through, for integrating Redis as a caching layer for MongoDB.
  - Discuss the benefits and considerations of each strategy.

- Data Synchronization:
  - Describe techniques for keeping the data in Redis cache synchronized with MongoDB, ensuring data consistency and avoiding stale data.

Implementing Redis Cache for MongoDB

- Choose a Redis client library: Select a Redis client library compatible with your programming language and framework.
- Configure Redis connection: Set up the connection parameters to connect your application with the Redis server. This typically includes specifying the host, port, and authentication details.
- Cache MongoDB query results: Identify the queries or data that can benefit from caching and implement the logic to store the results in Redis.
- Implement caching strategies: Determine the appropriate caching strategy based on your application requirements.
- Ensure data consistency: Establish mechanisms to maintain data consistency between MongoDB and Redis.
- Monitor and optimize: Regularly monitor the cache usage, performance, and data consistency.

Example: Redis cache for MongoDB



```python
import redis
import pymongo

# Connect to Redis
redis_client = redis.Redis(host='localhost', port=6379, db=0)

# Connect to MongoDB
mongo_client = pymongo.MongoClient('mongodb://localhost:27017')
mongo_db = mongo_client['mydatabase']
mongo_collection = mongo_db['mycollection']

# Example of caching MongoDB query results
def get_data_from_mongodb(key):
    # Check if data is present in Redis cache
    cached_data = redis_client.get(key)
    if cached_data:
        return cached_data

    # Fetch data from MongoDB
    data = mongo_collection.find_one({'key': key})

    # Cache the data in Redis
    redis_client.set(key, data)

    return data
```

Using Change Streams in MongoDB to Update Redis Cache

- Change Streams Overview:
  - Provide an overview of MongoDB's change streams, which allow applications to receive real-time notifications of database changes.
  - Explain how change streams can be used to update the Redis cache when changes occur in MongoDB.

- Implementation Example:
  - Provide a code example demonstrating how to use change streams to capture MongoDB updates and propagate them to Redis.

MongoDB Change Streams

- MongoDB Change Streams: Change Streams provide a way to listen for changes in MongoDB collections in real-time.
- Utility of Change Streams: Change Streams enable applications to react to data changes and keep various components synchronized.
- Keeping Redis Cache Updated: Change Streams can be used to capture MongoDB data changes and update the Redis cache accordingly.
- Real-time Data Sync: With Change Streams, Redis cache stays in sync with MongoDB updates, ensuring data consistency.
- Triggering Cache Updates: Change Streams can be configured to trigger cache updates based on specific events or changes in the MongoDB data.
- Seamless Integration: Change Streams provide a seamless mechanism for integrating MongoDB and Redis, keeping the cache up-to-date.

Maintaining Consistency Between MongoDB and Redis

- Consistency Challenges:
  - Discuss the challenges of maintaining consistency between MongoDB and Redis, such as handling concurrent updates and handling cache evictions.

- Consistency Strategies:
  - Explore strategies for maintaining consistency, including write-through updates, using transactional operations, and employing proper error handling.
-

Implementing Change Streams for Cache Updates

- Step 1: Setting up the MongoDB Change Stream:
  - Enable Change Streams in MongoDB configuration.
- Step 2: Establishing the Connection to Redis:
  - Use a Redis client library to establish a connection to the Redis cache.
- Step 3: Handling Change Events:
  - Listen for change events emitted by the MongoDB Change Stream.
- Step 4: Updating the Redis Cache:
  - Determine the key in the Redis cache corresponding to the updated data.
- Step 5: Handling Errors and Ensuring Consistency:
  - Implement error handling mechanisms to handle connection issues or failures during cache updates.
- Step 6: Testing and Monitoring:
  - Validate the integration by testing various data change scenarios and observing cache updates.
- Step 7: Optimization and Scaling:
  - Monitor the performance of the change stream and cache updates.

Best Practices for Maintaining Consistency

- Data Modeling:
- Design a consistent data model that aligns with the requirements of both MongoDB and Redis.
- Ensure that the data model accommodates the data structures and capabilities of both databases.
- Define clear relationships and dependencies between MongoDB collections and Redis cache entries.

- Error Handling and Resilience:
- Implement robust error handling mechanisms to handle failures and prevent data inconsistencies.
- Use proper exception handling and logging techniques to capture and handle errors during data operations.
- Employ strategies like retries, circuit breakers, and fallback mechanisms to handle transient errors and maintain consistency.

- Synchronization and Eventual Consistency:
- Implement synchronization mechanisms to ensure eventual consistency between MongoDB and Redis.

Best Practices for Maintaining Consistency

- Utilize techniques such as change streams, event-driven architectures, or background jobs to propagate changes from MongoDB to Redis.
- Apply proper sequencing and ordering mechanisms to ensure the consistency of related data updates across both databases.
- . Monitoring and Auditing:
- Set up monitoring and auditing mechanisms to track the consistency between MongoDB and Redis.
- Monitor data replication and synchronization processes to identify and resolve any inconsistencies promptly.
- Implement proper logging and tracking of data operations to enable effective troubleshooting and auditing.
- Documentation and Communication:
- Maintain clear and up-to-date documentation on the integration between MongoDB and Redis.
- Document the data flow, synchronization processes, and any relevant configurations.
- Foster effective communication between development teams working on both databases to address any concerns or inconsistencies promptly.

Monitoring and Troubleshooting

- Monitoring:
  - Utilize monitoring tools and techniques to track the performance and health of MongoDB and Redis in the integrated setup.
  - Monitor key metrics such as response time, throughput, CPU and memory usage, and cache hit rate.
  - Set up alerts and notifications to proactively identify and address any potential issues or bottlenecks.

- Troubleshooting:
  - Identify common issues that may arise when integrating MongoDB and Redis and their respective troubleshooting approaches.
  - Perform detailed analysis and diagnosis to identify the root cause of performance or consistency-related problems.
  - Utilize logging, debugging, and profiling tools to gather insights and troubleshoot integration issues effectively.

Lab

MongoDB and Redis Integration