

Key-Value Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Outline

- ❖ Key-value stores
 - General principles
- ❖ Riak, Redis
 - Characteristics
 - Main Features
 - Use Cases

Key-Value Databases

- ❖ Key value stores are the simplest of NOSQL types
 - consisting only of a unique key and a bucket containing any data you wish to store there.
- ❖ Key-value pairs
 - Key (id, identifier, primary key) – usually a string.
 - Value: can be anything (text, structure, image, etc.) – a black box for the database system.
- ❖ The content of the bucket can be literally anything
 - But unstructured or semi-structured data are the most common.
- ❖ The buckets can hold quite large entries including BLOBs (Basic Large Objects).
- ❖ KVs are row based systems designed for efficiency.

Key-Value Databases – Advantages

- ❖ Highly fault tolerant – always available.
- ❖ Schema-less offers easier upgrade path for changing data requirements
 - (Document stores provide even greater flexibility).
- ❖ Efficient at retrieving information about a particular object (bucket) with a minimum of disc operations.
- ❖ Very simple data model. Very fast to set up and deploy.
- ❖ Great at scaling horizontally across hundreds or thousands of servers.

Key-Value Databases – Advantages

- ❖ No requirement for SQL queries, indexes, triggers, stored procedures, temporary tables, forms, views, or the other technical overheads of RDBMS.
- ❖ Very high data ingest rates.
 - Favors write once, read many applications.
- ❖ Powerful offline reporting with very large data sets.
- ❖ Some vendors are offering advanced forms of KVs that approach the capabilities of document stores or column oriented stores.

Key-Value Databases – Disadvantages

- ❖ Not suitable for complex applications.
- ❖ Not efficient at updating records where only a portion of a bucket is to be updated.
- ❖ Not efficient at retrieving limited data from specific records.
 - For example, in an employee database returning only records of employees making between \$40K and \$60K.
- ❖ As the volume of data increases maintaining unique values as keys becomes more difficult
 - Some more complexity in generating character strings that will remain unique over a large set of keys.
- ❖ Generally needs to read all the records in a bucket or you may need to construct secondary indexes.

Key-Value Databases

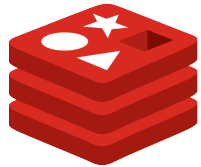
❖ **Suitable** use cases

- Session data, user profiles, user preferences, shopping carts, ...
- Create ever-growing datasets that are rarely accessed but grow over time. (Caching)
- Where write performance is your highest priority.

❖ **When not** to use

- Relationships among entities
- Queries requiring access to the content of the value part
- Set operations involving multiple key-value pairs

Key-Value Databases



redis



hazelcast



EHCACHE

EROSPIKE



SimpleDB

ORACLE®

BERKELEY DB



ArangoDB

Key Management

❖ How the keys should actually be designed?

❖ **Manually assigned keys**

- Real-world natural identifiers
- E.g. e-mail addresses, login names, ...

❖ **Automatically generated keys**

- Auto-increment integers
 - Not suitable in peer-to-peer architectures!
- More complex keys generated by algorithms
 - Keys composed from multiple components such as time stamps, cluster node identifiers, ...
 - Used in practice

Query Patterns

❖ Basic **CRUD** operations

- Only when a key is provided
- The knowledge of the keys is essential
- It might even be difficult for a particular database system to provide a list of all the available keys!

❖ **No searching by value**

- But we could instruct the database how to parse the values
- ... so that we can fetch the intended search criteria
- ... and store the references within index structures

❖ **Batch / sequential processing**

- MapReduce

Other Functionality

❖ **Expiration** of key-value pairs

- After a certain interval of time key-value pairs are automatically removed from the database
- Useful for user sessions, shopping carts etc.

❖ **Collections** of values

- We can store not only ordinary values, but also their collections such as ordered lists, unordered sets etc.

❖ **Links** between key-value pairs

- Values can mutually be interconnected via links
- These links can be traversed when querying

❖ Particular functionality depends on the store.

Riak Key-Value Store



RiakKV

- ❖ Developed by Basho Technologies
 - <http://basho.com/products/riak-kv/>
 - Implemented in Erlang
 - Initial release in 2009
 - Operating system: Linux, Mac OS X, ... (not Windows)
- ❖ Open source, incremental scalability, high availability, operational simplicity, decentralized design, automatic data distribution, advanced replication, fault tolerance, ...
- ❖ General-purpose, concurrent, garbage-collected programming language and runtime system

Data Model

- ❖ Instance (→ bucket types) → buckets → objects
- ❖ **Bucket** = collection of objects (logical, not physical collection)
 - Each object must have a unique key
 - Various properties are set at the level of buckets
 - E.g. default replication factor, read / write quora, ...
- ❖ **Object** = key-value pair
 - Key is a Unicode string
 - Value can be anything (text, binary object, image, ...)

Each object is also associated with metadata

 - E.g. its content type (text/plain, image/jpeg, ...),
 - and other internal metadata as well

Data Model

- ❖ How buckets, keys and values should be designed?
- ❖ Complex objects containing various kinds of data
 - E.g. one key-value pair holding information about all the actors and movies at the same time
- ❖ Buckets with different kinds of objects
 - E.g. distinct objects for actors and movies, but all in one bucket
 - Structured naming convention for keys might help
 - E.g. actor_trojan, movie_medvidek
- ❖ Separate buckets for different kinds of objects
 - E.g. one bucket for actors, one for movies

Riak Operations

❖ Basic CRUD operations

- Create: POST or PUT methods
 - Inserts a key-value pair into a given bucket
 - Key is specified manually, or will be generated automatically
- Read: GET method
 - Retrieves a key-value pair from a given bucket
- Update: PUT method
 - Updates a key-value pair in a given bucket
- Delete: DELETE method
 - Removes a key-value pair from a given bucket

❖ Extended functionality

- Links – relationships between objects and their traversal
- Search 2.0 – full-text queries accessing values of objects
- MapReduce

Riak Usage: API

❖ HTTP API

- All the user requests are submitted as HTTP requests with an appropriately selected method and specifically constructed URL, headers, and data

❖ Protocol Buffers API

❖ Erlang API

❖ Client libraries for a variety of programming languages

- Official: Java, Ruby, Python, C#, PHP, ...
- Community: C, C++, Haskell, Perl, Python, Scala, ...

Riak Usage: HTTP API

❖ cURL tool

- Allows to transfer data from / to a server using HTTP

❖ Options

- -X command, --request command
 - HTTP request method to be used (GET, ...)
- -d data, --data data
 - Data to be sent to the server (implies the POST method)
- -H header, --header header
 - Extra headers to be included when sending the request
- -i, --include
 - Include received headers when printing the response

Redis

(REmote DIctionary Service)



Redis Overview

❖ Redis

- **In-memory** key-value store
- Open source, master-slave replication architecture, sharding, high availability, various persistence levels, ...

❖ Developed by Redis Labs

❖ Implemented in C

❖ First release in 2009

❖ Available at <http://redis.io/>

Redis Overview

❖ Functionality

- Standard key-value store
- Support for structured values (e.g. lists, sets, ...)
- Time-to-live
- Transactions

❖ Redis is not just a plain key-value store, but a data structures server, supporting different kind of values.

❖ Real-world users

- Twitter, GitHub, Pinterest, StackOverflow, Flickr, ...

Data Model

❖ Structure

- Instance → databases → objects

❖ **Database** = collection of objects

- Databases do not have names, but integer identifiers

❖ **Object** = key-value pair

- Key is a string (i.e. any binary data)
- Values can be...
 - Atomic: string
 - Structured: list, set, ordered set, hash

Data Types

❖ String

- The only atomic data type
- May contain any binary data (e.g. string, integer counter, PNG image, ...)
- Maximal allowed size is 512 MB

❖ List

- Ordered collection of strings
- Elements should preferably be read / written at the head / tail

Data Types

❖ Set

- Unordered collection of strings
- Duplicate values are not allowed

❖ Sorted set

- Ordered collection of strings
- The order is given by a score (floating number value) associated with each element (from the smallest to the greatest score)

❖ Hash

- Associative map between string fields and string values
- Field names have to be mutually distinct

Interface

❖ Command line client

- redis-cli

❖ Two modes are available...

❖ Basic

- Commands are passed as standard command line arguments
 - E.g. redis-cli PING, redis-cli -n 16 DBSIZE
- Batch processing is possible as well
 - E.g. cat script.txt | redis-cli

❖ Interactive

- Users type database commands at the prompt redis-cli

❖ **RESP** (REdis Serialization Protocol)

Basic Commands

- ❖ **SET** key value
 - inserts / replaces a given string
- ❖ **GET** key
 - returns a given string
- ❖ **HELP** command
 - Provides basic information about Redis commands
- ❖ **CLEAR**
 - Clears the terminal screen
- ❖ **FLUSHDB**
 - Deletes all the keys of the currently selected database
- ❖ **BGSAVE**
 - Saves the current dataset (on background)

Strings Operations

❖ **STRLEN** key

- returns a string length

❖ **APPEND** key value

- appends a value at the end of a string

❖ **GETRANGE** key start end

- returns a substring Both the boundaries are considered to be inclusive
- Positions start at 0;
- Negative offsets for positions starting at the end

❖ **SETRANGE** key offset value

- replaces a substring
- Binary 0 are padded when the original string is not long enough

Counter Operations

- ❖ **INCR** key
 - Increments / decrements a value by 1
- ❖ **INCRBY** key increment
- ❖ **DECRBY** key increment
 - Increments / decrements a value by a given amount

Handling Keys

- ❖ **EXISTS** key
 - determines whether a key exists
- ❖ **KEYS** pattern
 - finds all the keys matching a pattern (*, ?, ...)
 - E.g. KEYS *
- ❖ **DEL** key ...
 - removes a given object / objects
- ❖ **RENAME** key newkey
 - changes the key of a given object
- ❖ **TYPE** key – determines the type of a given object
 - Types: integer, string, list, set, zset and hash

Volatile Keys

❖ Keys with limited time to live

- When a specified timeout elapses, a given object is removed
- Works with any data type

❖ **EXPIRE** key seconds

- Sets a timeout for a given object, i.e. makes the object volatile
- Can be called repeatedly to change the timeout

❖ **TTL** key

- Returns the remaining time to live for a key

❖ **PERSIST** key

- Removes the existing timeout

Complex Datatypes

- ❖ Redis' popularity comes mostly by supporting:
 - lists, hashes, sets, and sorted sets
- ❖ These collection can contain up to 2^{32} elements (more than 4 billion) per key.
- ❖ Commands follow a good pattern.
 - Set commands begin with S,
 - Hashes with H
 - Sorted sets with Z.
 - List commands generally start with either an L (for left) or an R (for right),
 - depending on the direction of the operation (such as LPUSH).

Lists

- ❖ **LPUSH** key value
- ❖ **RPUSH** key value
 - Adds a new element to the head / tail (Left / Right)
- ❖ **LINSERT** key BEFORE | AFTER pivot value
 - Inserts an element before / after another one
- ❖ **LPOP** key
- ❖ **RPOP** key
 - Removes and returns the first / last element (Left / Right)

Lists

❖ **INDEX** key index

- gets an element by its index
 - The first item is at position 0;

❖ **LRANGE** key start stop

- gets a range of elements

❖ **REM** key count value

- Removes a given number of matching elements from a list
 - Positive / negative = moving from head to tail / tail to head
 - 0 = all the items are removed

❖ **LEN** key

- gets the length of a list

Sets

- ❖ **SADD** key value ...
 - Adds an element / elements into a set
- ❖ **SREM** key value ...
 - Removes an element / elements from a set
- ❖ **SISMEMBER** key value
 - Determines whether a set contains a given element
- ❖ **SMEMBERS** key
 - gets all the elements of a set
- ❖ **SCARD** key
 - gets the number of elements in a set
- ❖ **SUNION / SINTER / SDIFF** key ...
 - Calculates and returns a set union / intersection / difference of two or more sets

Hashes

- ❖ **HSET** key field value
 - sets the value of a hash field
- ❖ **HGET** key field
 - gets the value of a hash field

Batch alternatives

- ❖ **HMSET** key field value
 - Sets values of multiple fields of a given hash
- ❖ **HMGET** key field ...
 - Gets values of multiple fields of a given hash

Hashes

- ❖ **HEXISTS** key field
 - determines whether a given field exists
- ❖ **HGETALL** key
 - gets all the fields and values
- ❖ **HKEYS** key
 - gets all the fields in a given hash
- ❖ **HVALS** key
 - gets all the values in a given hash
- ❖ **HDEL** key field
 - Removes a given field / fields from a hash
- ❖ **HLEN** key
 - returns the number of fields in a given hash

Sorted Sets

Basic operations

- ❖ **ZADD** key score value
 - Inserts one element / multiple elements into a sorted set
- ❖ **ZREM** key value ...
 - Removes one element / multiple elements from sorted set

Working with score

- ❖ **ZSCORE** key value
 - Gets the score associated with a given element
- ❖ **ZINCRBY** key increment value
 - Increments the score of a given element

Sorted Sets

Retrieval of elements

❖ **ZRANGE** key start stop

- Returns all the elements within a given range based on positions

❖ **ZRANGEBYSCORE** key min max

- Returns the elements within a given range based on scores

Other operations

❖ **ZCARD** key

- Gets the overall number of all elements

❖ **ZCOUNT** key min max

- Counts elements within a given range based on score

Geospatial field operations

- ❖ **GEOADD** key longitude latitude member ...
 - Adds the specified geospatial items (latitude, longitude, name) to the specified key.
- ❖ **GEODIST** key member1 member2 ...
 - Return the distance between two members.
- ❖ **GEOHASH** key member ...
 - Return Geohash string (compatible with geohash.org)
- ❖ **GEOPOS** key member ...
 - Return the positions (longitude,latitude) of all the specified members.
- ❖ **GEORADIUS** key longitude latitude radius ...
 - Return the members which are within the radius of the location.

References

- ❖ Commands
 - <http://redis.io/commands>
- ❖ Documentation
 - <http://redis.io/documentation>
- ❖ Data types
 - <http://redis.io/topics/data-types>