FAKULTA ELEKTROTECHNICKÁ

České vysoké učení technické v Praze

B4M36DS2 – Database Systems 2

Lecture 5 – Key-Value stores: Redis. Data types

23. 10. 2023

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram **@Yulia_Prokop**

Based on **Martin Svoboda**'s materials (https://www.ksi.mff.cuni.cz/~svoboda/courses/211-B4M36DS2/)

https://cw.fel.cvut.cz/b231/courses/b3b36prg/start

Outline

Key-value stores

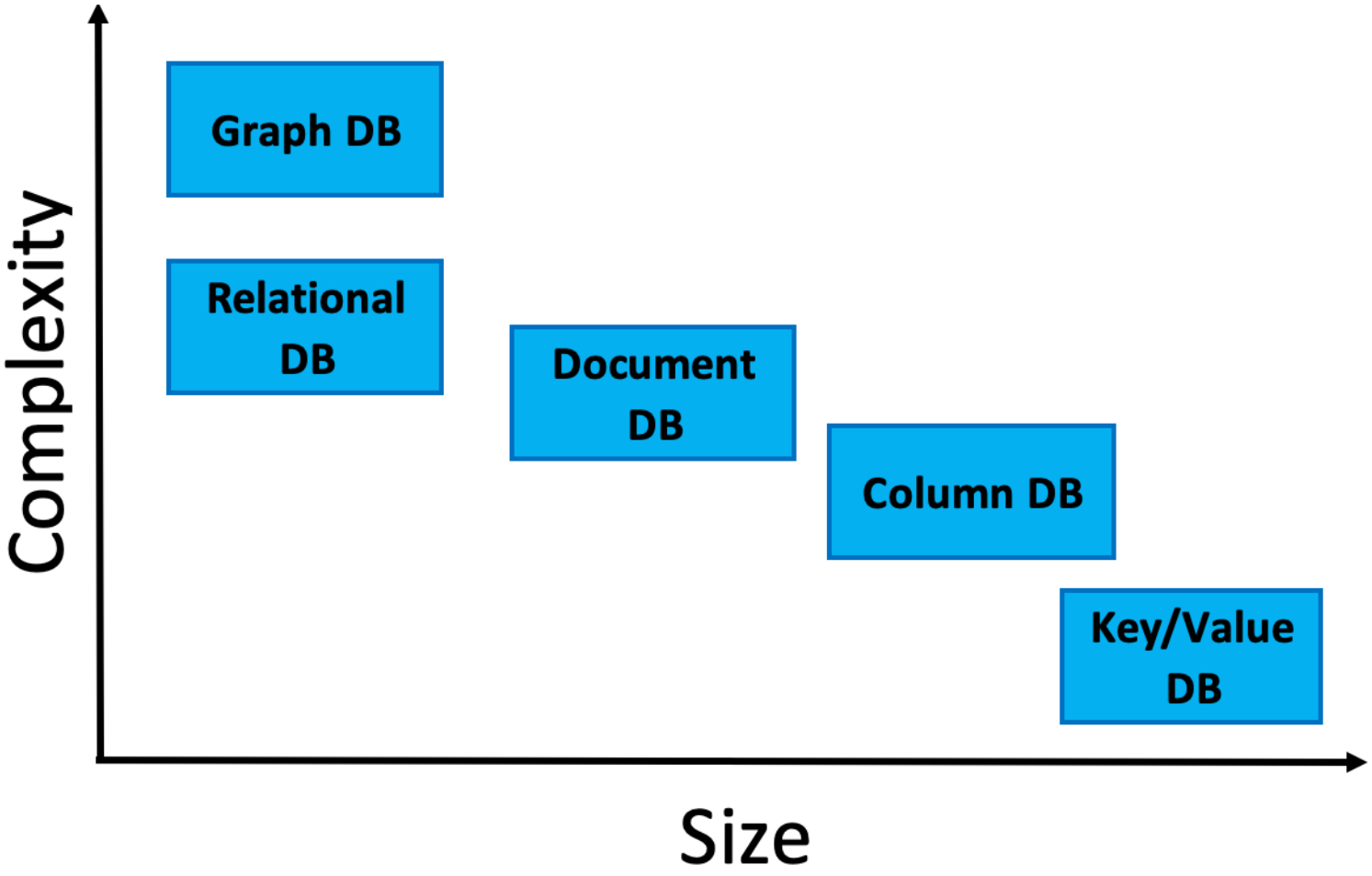
- Introduction

Redis

- Data model
- Keys
- Data Types
- Basic commands and operations
- Examples

Key-value databases

Size / Complexity of data stores



Key-Value Stores

- **Key-value** storage systems store large numbers (billions or even more) of small (KB-MB) sized records
- Records are **partitioned** across multiple machines and
- Queries are routed by the system to appropriate machine
- Records are also **replicated** across multiple machines, to ensure availability even if a machine fails
  - Key-value stores ensure that updates are applied to all replicas, to ensure that their values are **consistent**

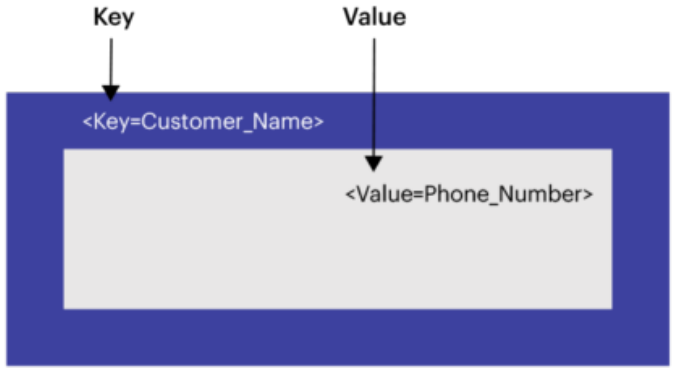
Key-Value Stores

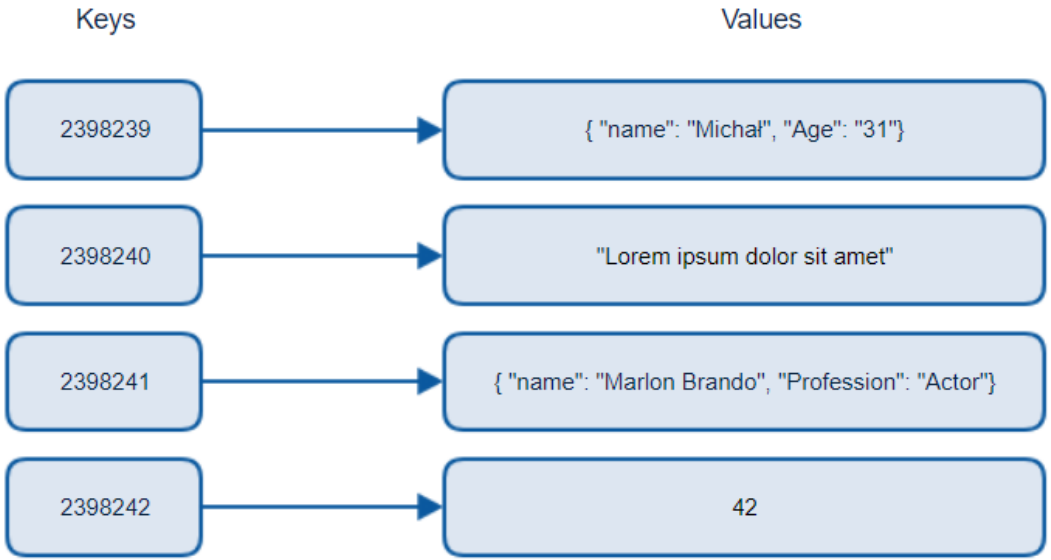
- Key-value stores may store
  - **uninterpreted bytes** with an associated key
    - E.g., Amazon S3, Amazon Dynamo
  - **Wide-table** (can have arbitrarily many attribute names) with associated key
    - Google BigTable, Apache Cassandra, Apache Hbase, Amazon DynamoDB
    - Allows some operations (e.g., filtering) to execute on storage node
  - JSON
    - MongoDB, CouchDB (document model)
- **Document stores** store semi-structured data, typically JSON
- Some key-value stores support multiple versions of data, with timestamps/version numbers

Key-Value Stores

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334





Source: https://redis.com, https://www.michalbialecki.com

Key-Value Stores

Data model

- The most simple NoSQL database type
  - Works as a simple hash table (mapping)
- **Key-value pairs**
  - **Key** (id, identifier, primary key)
  - **Value:** binary object, black box for the database system

Query patterns

- Create, update or remove value for a given key
- **Get value** for a given key

Characteristics

- Simple model ⇒ **great performance, easily scaled, …**
- Simple model ⇒ **not for complex queries nor complex data**

Key Management

How the keys should actually be designed?

- **Real-world** identifiers
  - E.g. e-mail addresses, login names, …
- **Automatically generated** values
  - Auto-increment integers
    - Not suitable in peer-to-peer architectures!
  - Complex keys
    - Multiple components / combinations of time stamps, cluster node identifiers, …
    - Used in practice instead

Prefixes describing entity types are often used as well

- E.g. movie_medvidek, movie_223123, …

Query Patterns

Basic **CRUD operations**

- Only when a key is provided
- ⇒ 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!

Accessing the contents of the value part is not possible in general

- But we could instruct the database how to **parse the values**
- … so that we can **index** them based on certain **search criteria**

Batch / sequential processing

- **MapReduce**

Other Functionality

Expiration of key-value pairs

- Objects are **automatically removed** from the database **after a certain interval of time**
- Useful for user sessions, shopping carts etc.

Links between key-value pairs

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

Collections of values

- Not only ordinary values can be stored, but also their collections (e.g. **ordered lists**, **unordered sets**, …)

Particular functionality always depends on the store we use!

When to use a key-value database

- Handling Large Volume of Small and Continuous Reads and Writes
- Storing Basic Information
- Applications with Infrequent Updates and Simple Queries
- When your application needs to handle lots of small continuous reads and writes, that may be volatile

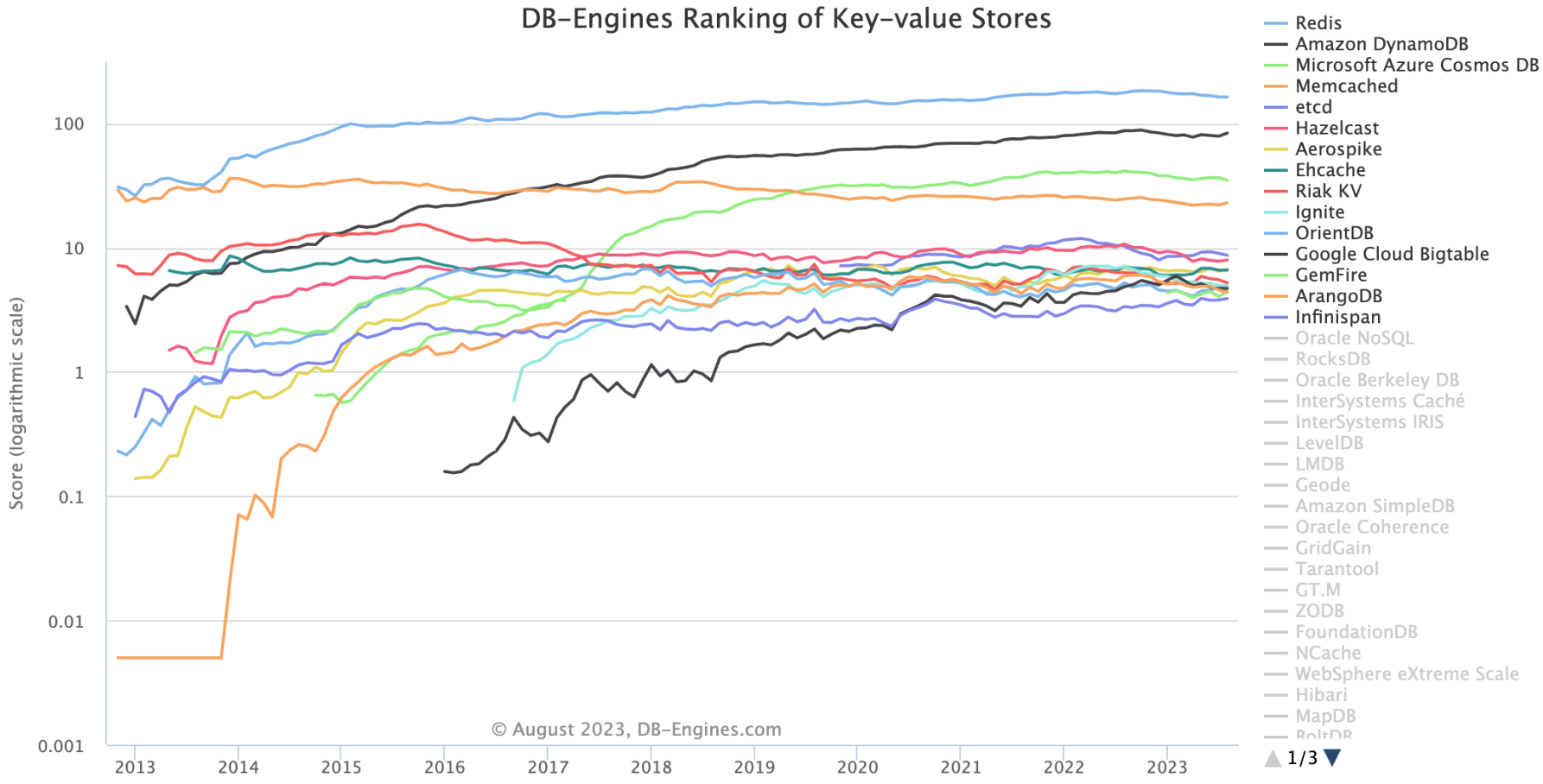
Use cases

- Session Management on a Large Scale
- Using Cache to Accelerate Application Responses
- Storing Personal Data on Specific Users
- Product Recommendations and Personalized Lists
- Managing Player Sessions in Massive Multiplayer Online Games

Limitations of Key-Value Stores

- Key-value stores are not optimized and require a parser for multiple values. They are only optimized for a single key and value.
- These kinds of stores cannot filter out the value fields.
- Key-value stores are not optimized for lookup. Lookup will scan the whole collection, which will affect performance.
- Key-value stores are not compatible with SQL.
- Key-value stores do not allow you to rollback if a key store fails.
- There is no standard query language as opposed to SQL.

Key-value Stores ranking



Source: https://db-engines.com

Redis

Redis

- **In-memory data structure store**

  Can be used as Database, Cache, Message broker

- Open-source software: http://redis.io/
- Developed by **Redis Labs**
- Implemented in **C**
- First release in 2009
- Redis is really fast: 100,000+ read / writes per second

WHY Redis? Who uses Redis?

- Very flexible
- Very fast
- No schemas, column names
- Rich Datatype Support
- Caching & Disk persistence

Redis functionality

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

  Twitter, GitHub, Pinterest, StackOverflow, Flicker, …

Data Model

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**, **sorted set**, **hash**
    - **Key** and **value** are binary-safe, up to 512Mb

Redis keys

- **Keys** are binary safe – it is possible to use any binary sequence as a key
- The empty string is also a valid key
- Too long keys are not a good idea
- Too short keys are often also not a good idea ("**u:1000:pwd**" versus "**user:1000:password**")
- A nice idea is to use some kind of schema, like: "**object-type:id:field**"

Redis keys commands

- **SET** key value [EX seconds]

  Sets the string value of a key

- **GET** key

  Returns the string value of a key

- **EXISTS** key [key …]

  Checks whether a key exists

- **TYPE** [key]

  Returns the type of a key

- **DEL** key [key …]

  Deletes a key

Keys – Examples

EXISTS mykey
(integer) 1
DEL mykey
(integer) 1
EXISTS mykey
(integer) 0

SET a hello
OK
GET a
"hello"
RENAME a ahoj
OK
GET a
(nil)

122

EXISTS a
(integer) 0
GET ahoj
"hello"

TYPE ahoj
string
SET x 120
OK
TYPE x
string

Volatile Objects

Keys with limited time to live

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

- ✓ Temporary lockdowns
- ✓ Temporary subscriptions
- ✓ Burnable bonuses
- ✓ SMS timeout (not more often than in a minute)
- ✓ Verification by code (within 1 minute).

Volatile Objects

- **Set:**
  - **EXPIRE** key seconds
    - Sets a timeout for a given object, i.e. makes the object volatile.
  - **EXPIREAT** key timestamp
  - **PEXPIRE** key milliseconds
  - **PEXPIREAT** key milliseconds-timestamp
- **Examine:**
  - **TTL** key
    - Returns the remaining time to live for a key that has a timeout
  - **PTTL** key
- **Remove:**
  - **PERSIST** key
    - Removes the existing timeout, i.e. makes the object persistent

Redis: Volatile Objects examples

SET counter 100
OK

GET counter
"100"

EXPIRE counter 10
(integer) 1

EXISTS counter
(integer) 1

EXISTS counter
(integer) 0

SET key some-value
OK

EXPIRE key 20
(integer) 1

GET key
"some-value"

GET key
"some-value"

GET key
(nil)

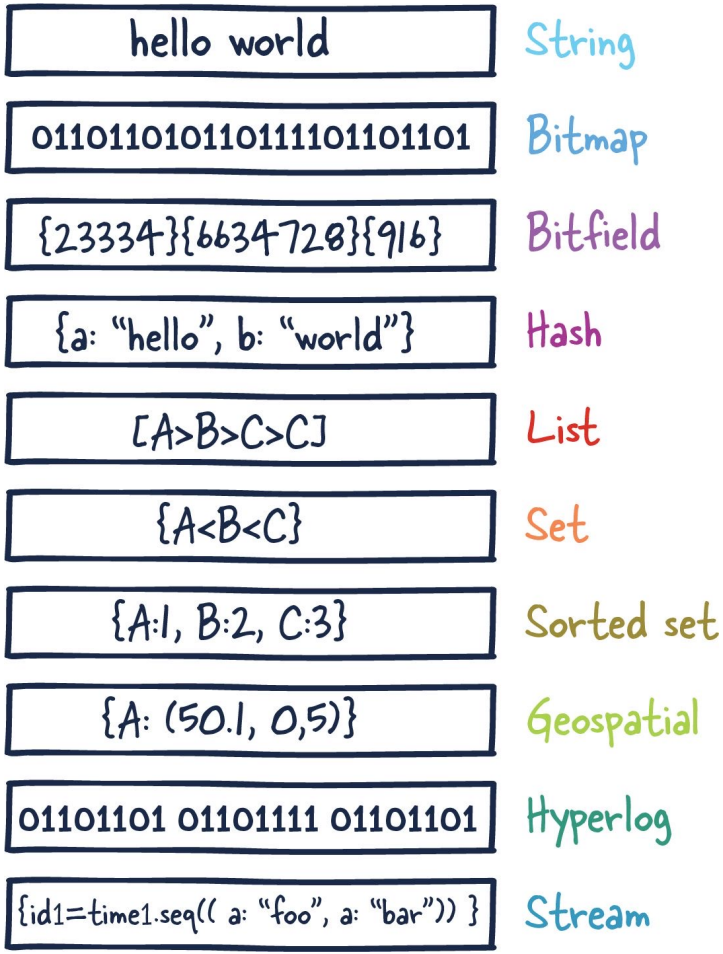
SET key 100 **EX** 10
OK

TTL key
(integer) 2

Data Types

- **Strings**
- **Lists**
- **Sets**
- **Sorted sets**
- **Hashes**
- **Bitmaps**
- **Hyperlogs**
- **Geospatial indexes**

Datatypes cannot be nested!
(no lists of hashes)



Further information: https://redis.io/docs/data-types/

Source: https://architecturenotes.co/redis/

Data Types : Strings

- **String**
  - The only **atomic data type**
  - May contain any binary data

    (e.g. string, integer counter, PNG image, …)
  - Maximal allowed size is 512 MB
  - Use cases:
    - ✓ Store JPEG's
    - ✓ STORE serialized objects

Data Types : Lists

- **List** – a list (an ordered collection) of Strings
  - Example: **[c, a, b, a]**
  - Max size 4 294 967 295
- Can be used for
  - For example, timelines of social networks, queues
- Speed
  - Actions at the start or end of the list are very fast
  - Actions in the middle are a little less fast

Data Types : Sets

- **Set**
  - **An unordered collection of Strings**
    - Example: **[d, a, b, c]**
  - Duplicate values are not allowed
  - Useful for tracking unique items
  - Allow extracting random members

    Using SPOP, SRANDMEMBER
  - Useful for intersecting and diffs

Data Types: Sorted Sets

- **Sorted set**

  - **An ordered collection of strings**
    - Example: **[0:d, 1:a, 2:b, 3:c]**.
    - Score: 0, 1, 2, 3.
  - The order is given by a score (floating number value) associated with each element (from the smallest to the greatest score)
  - The score can be a timestamp
  - Members are unique. Scores are not
  - Ordering for items with the same score is alphabetic
  - Useful for leader boards or autocomplete

Data Types: Hashes

- **Hash**
  - **Associative map between string fields and string values**
    - Example: {login:Matej, pass:1_2#}
  - Ideal for storing objects
  - Field names have to be mutually distinct

String commands

Basic commands

- **SET** key value

  Inserts / replaces a given string

- **GET** key

  Gets a given string

Counter operations

- **INCR** key
- **DECR** key

  Increments/decrements a value by 1

- **INCRBY** key increment
- **DECRBY** key decrement

  Increments/decrements a value by a given amount

String 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 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 is padded when the original string is not long enough

String examples

SET mykey somevalue
OK

GET mykey
"somevalue"

STRLEN mykey
(integer) 9

APPEND mykey 666
(integer) 12

GET mykey
"somevalue666"

GETRANGE mykey 2 8
"mevalue"

SETRANGE mykey 5 xxxx
(integer) 12

GET mykey
"somevxxxx666"

INCR counter
(integer) 101

INCR counter
(integer) 102

INCRBY counter 50
(integer) 152

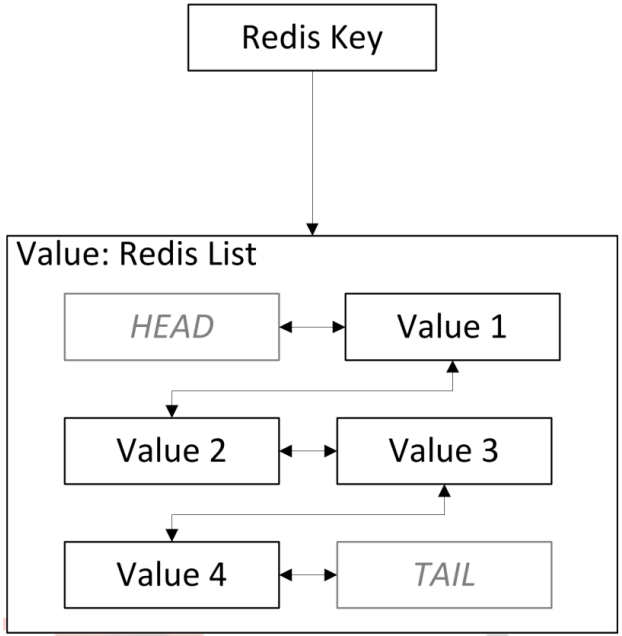
Lists commands

Insertion of new elements

- **LPUSH** key value
- **RPUSH** key value
  - Adds a new element to the head / tail
- **LINSERT** key **BEFORE|AFTER** pivot value
  - Inserts an element before / after another one

Retrieval of elements

- **LPOP** key
- **RPOP** key
  - Removes and returns the first / last element



Lists commands

Retrieval of elements

- **LINDEX** key index – gets an element by its index
  - The first item is at position 0
  - Negative positions are allowed as well
- **LRANGE** key start stop – gets a range of elements

Removal of elements

- **LREM** 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
- **LTRIM** key start stop
  - Trim an existing list so that it will contain only the specified range of elements

Other operations

- **LLEN** key – gets the length of a list

Lists examples

RPUSH mylist Alpha
(integer) 1

RPUSH mylist Beta
(integer) 2

RPUSH mylist first
(integer) 3

LRANGE mylist 0 -1
1) "first"
2) "Alpha"
3) "Beta"

LRANGE names 0 100
1) "first"
2) "Alpha"
3) "Beta"

RPUSH mylist 1 2 3 4 5 "foo bar"
(integer) 9

RPOP mylist
"foo bar"

RPOP mylist
"5"

RPOP mylist
"4"

DEL mylist
(integer) 1

RPUSH mylist 1 2 3 4 5
(integer) 5

LTRIM mylist 0 2
OK

LRANGE mylist 0 -1
1) "1"
2) "2"
3) "3"

Sets commands

Basic operations

- **SADD** key value …

  Adds an element/elements into a set

- **SREM** key value …

  Removes an element/elements from a set

Data querying

- **SISMEMBER** key value

  Determines whether a set contains a given element

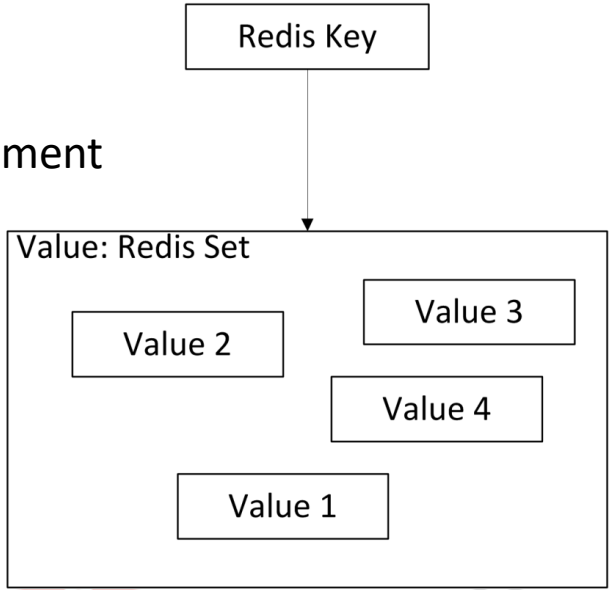
- **SMEMBERS** key

  Gets all the elements of a set

Other operations

- **SCARD** key

  Gets the number of elements in a set



SET OPERATIONS

- **SUNION** key …
- **SINTER** key …
- **SDIFF** key …
  - Calculates and returns a set union / intersection / difference of two or more sets

Set example

Create a set with elements 1, 2, 3

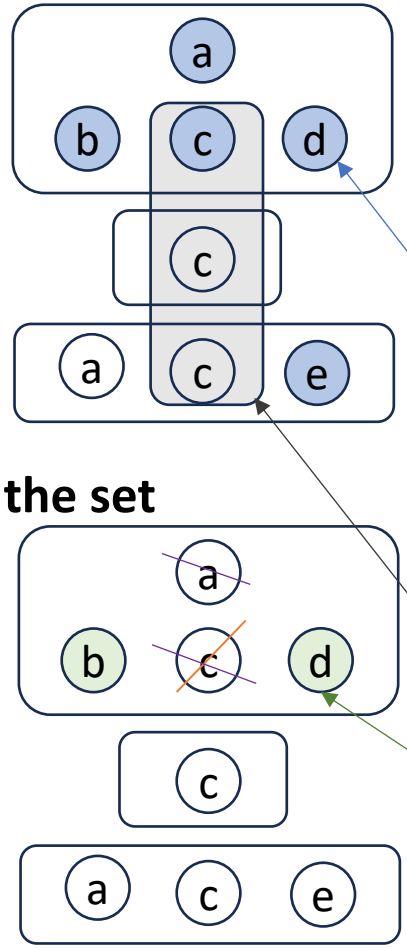
SADD myset 1 2 3
(integer) 3

SMEMBERS myset
1) "1"
2) "2"
3) "3"

Test existence of 3 and 30 in the set

SISMEMBER myset 3
(integer) 1

SISMEMBER myset 30
(integer) 0



SADD set1 **a b c d**
(integer) 4

SADD set2 c
(integer) 1

SADD set3 a c **e**
(integer) 3

SUNION set1 set2 set3
1) "a"
2) "e"
3) "c"
4) "d"
5) "b"

SINTER set1 set2 set3
1) "c"

SDIFF set1 set2 set3
1) "b"
2) "d"

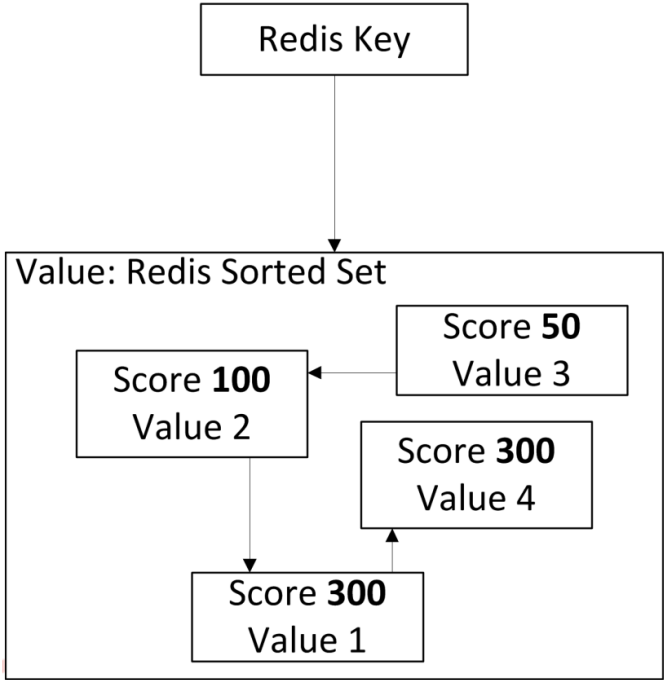
Sorted Sets commands

Basic operations

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

Retrieval of elements

- **ZRANGE** key start stop

  Returns all the elements within a given range based on positions

- **ZRANGEBYSCORE** key min max

  Returns all 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 all the elements within a given range based on the score

Sorted Sets example

```
ZADD hackers 1940 "Alan Kay"
(integer) 1
ZADD hackers 1957 "Sophie Wilson"
(integer) 1
ZADD hackers 1953 "Richard Stallman"
(integer) 1
ZADD hackers 1949 "Anita Borg"
(integer) 1
ZADD hackers 1965 "Yukihiro Matsumoto"
(integer) 1
ZADD hackers 1914 "Hedy Lamarr"
(integer) 1
ZADD hackers 1916 "Claude Shannon"
(integer) 1
ZADD hackers 1969 "Linus Torvalds"
(integer) 1
ZADD hackers 1912 "Alan Turing"
(integer) 1
```

Sorted Sets example

ZRANGE hackers 0 -1

1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
6) "Richard Stallman"
7) "Sophie Wilson"
8) "Yukihiro Matsumoto"
9) "Linus Torvalds"

ZRANGE hackers 0 -1 WITHSCORES

1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
5) "Claude Shannon"
6) "1916"
7) "Alan Kay"
8) "1940"
9) "Anita Borg"
10) "1949"
11) "Richard Stallman"
12) "1953"
13) "Sophie Wilson"
14) "1957"
15) "Yukihiro Matsumoto"
16) "1965"
17) "Linus Torvalds"
18) "1969"

Sorted Sets example

Born before 1950:

ZRANGEBYSCORE hackers **-inf 1950**
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg“

Remove hackers born between 1940 and 1960:

ZREMRANGEBYSCORE hackers **1940 1960**
(integer) 4

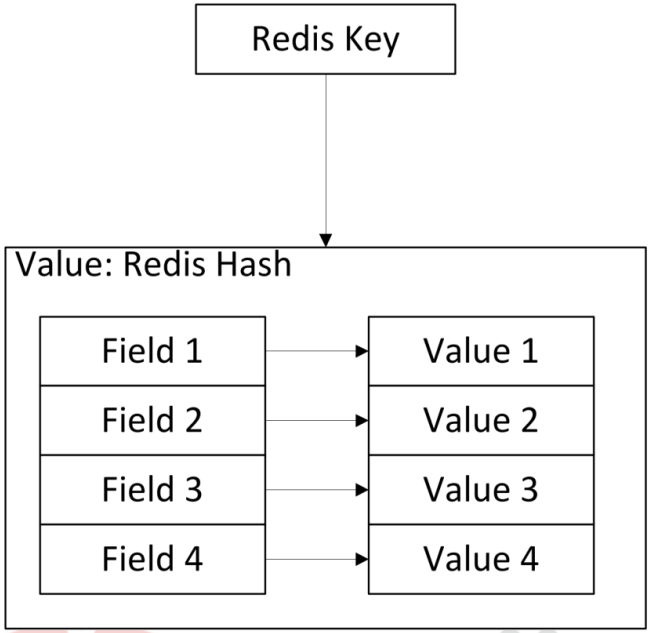
Hash operations

Basic operations

- **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



Hash operations

Field retrieval operations

- **HEXISTS** key field – determines whether a field exists
- **HGETALL** key – gets all the fields and values
  - Individual fields and values are interleaved
- **HKEYS** key – gets all the fields in a given hash
- **HVALS** key – gets all the values in a given hash

Other operations

- **HDEL** key field …
  - Removes a given field / fields from a hash
- **HLEN** key – returns the number of fields in a given hash

Hash example

HSET user:1000 *username* antirez *birthyear* 1977 *verified* 1
(integer) 3

HGET user:1000 *username*
"antirez"

HGET user:1000 *birthyear*
"1977"

HGETALL user:1000
1) *"username"*
2) "antirez"
3) *"birthyear"*
4) "1977"
5) *"verified"*
6) "1"

Hash example - Shopping Cart

Relational model

carts

CartID	User
1	matej
2	tomas
3	matej

cart_lines

Cart	Product	Qty
1	45	1
1	78	2
2	51	3
2	213	2
2	94	6

```
UPDATE cart_lines
SET Qty = Qty + 3
WHERE Cart = 1 AND Product = 45
```

Redis model

```
SADD carts_matej 1 3
(integer) 2

SADD carts_tomas 2
(integer) 1

HSET cart:1 user "matej" product:45 1 product:78 2
(integer) 3

HSET cart:2 user "tomas" product:51 3 product:213
2 product:94 6
(integer) 4

HINCRBY cart:1 product:45 3
```

Hash example - Shopping Cart

Querying data: what is in the carts of matej?

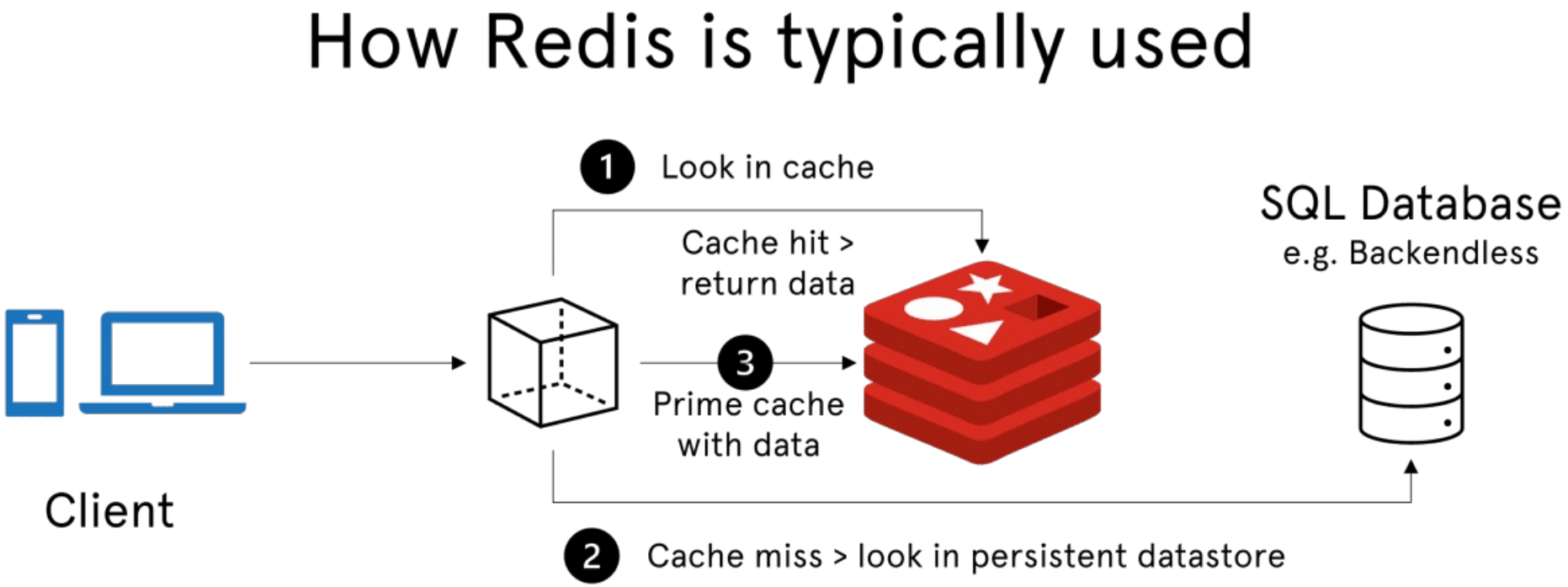
SMEMBERS carts_matej
1) "1"
2) "3"

HGETALL cart:1
1) "user"
2) "matej"
3) "product:45”
4) "4"
5) "product:78”
6) "2”

HGETALL cart:3
…

Redis supports Lua scripting, which allows you to execute a series of commands atomically.

Redis as cash



Source: https://backendless.com

Redis as a Database

- Special naming can help using Redis as database

```
SADD users:names peter
HSET users:peter name "Peter Petrov"
HSET users:peter email "pp@gmail.com"

SADD users:names maria
HSET users:maria name "Maria Ivanova"
HSET users:maria email "maria@yahoo.com"

SMEMBERS users:names
HGETALL users:peter
```

Add a user "peter".

Use "users:peter" as key to hold user data

Add a user "maria"

List all users

List all properties for user "peter"

Summary

1. Redis is an ultra-fast in-memory data store

   Not a database, used along with databases

2. Supports strings, numbers, lists, hashes, sets, sorted sets

3. Used for caching / simple apps