



FAKULTA ELEKTROTECHNICKÁ

České vysoké učení technické v Praze

B4M36DS2 – Database Systems 2

Lecture 5 – **Key-Value stores**: Redis. Part 2

21. 10. 2024

Yuliia Prokop

prokoyul@fel.cvut.cz, Telegram [@Yulia_Prokop](https://t.me/Yulia_Prokop)



CourseWare Wiki

<https://cw.fel.cvut.cz/wiki/courses/b4m36ds2/start>

Examples

Publish / Subscribe

Geospatial

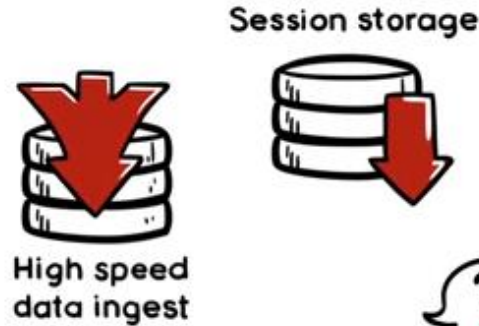
Redis Streams

Transactions

RediSearch

RedisJSON

Persistence



EXAMPLES

1. Rating Systems and Leaderboards

- Implementation details:
 - Uses Redis **sorted** sets.
 - Each player or participant is represented as a member of the set.
 - The player's score is used as the score in the sorted set.
 - The ZADD command adds or updates a player's score.
 - ZREVRANGE retrieves the top N players.
 - ZRANK determines a specific player's position in the ranking.

Adding players to the leaderboard

```
ZADD leaderboard 1000 "player1" 2000 "player2" 3000 "player3"
```

Getting the top 3 players with their scores

```
ZREVRANGE leaderboard 0 2 WITHSCORES
```

Getting the rank of a player (zero-based ranking)

```
ZRANK leaderboard "player1"
```

Incrementing a player's score by 500 points

```
ZINCRBY leaderboard 500 "player1"
```

Getting the count of players with scores between 2000 and 3000

```
ZCOUNT leaderboard 2000 3000
```

2. Session Management Systems

- Implementation details:
 - Each session is stored as a hash in Redis.
 - The key is a unique session identifier.
 - The hash stores various session attributes (user ID, last access time, etc.).
 - The EXPIRE command is used to automatically delete outdated sessions.

```
HSET session:abc123 user_id 1000 last_access 1631234567 is_logged_in 1  
# Session expires in one hour  
EXPIRE session:abc123 3600  
# Get all session data  
HGETALL session:abc123
```

3. Caching Systems

- Implementation details:
 - It uses simple string keys to store cached data.
 - Values can be serialized objects or JSON strings.
 - The **SET** command with **EX** parameter sets a value with a lifetime.

```
SET "user:profile:1000" "{\"name\":\"John\", \"email\":\"john@example.com\"}" EX 300  
GET "user:profile:1000"
```

4. Real-time Counters and Statistics

- Implementation details:
 - Uses string keys for simple counters.
 - Hashes are used for more complex statistics.
 - The **INCR** command increases the counter value.
 - **HINCRBY** is used to increase values in a hash.

```
HSET stats:2023-09-10 pageviews 150 unique_visitors 75
```

```
INCR "visits:total"
```

```
HINCRBY "stats:2023-09-10" pageviews 1
```

```
HINCRBY "stats:2023-09-10" unique_visitors 1
```

5. Rate Limiting Systems

- Implementation details:
 - Uses keys that include a user identifier or IP address.
 - The key value is the number of requests.
 - **INCR** increases the request counter.
 - **EXPIRE** sets the key's lifetime.

```
INCR "rate:ip:192.168.1.1"  
# Limit resets after 60 seconds  
EXPIRE "rate:ip:192.168.1.1" 60  
# Check the current number of requests  
GET "rate:ip:192.168.1.1"
```

6. Simple Message Queue Systems

- Implementation details:
 - Uses Redis lists to implement queues.
 - **LPUSH** adds elements to the beginning of the queue.
 - **RPOP** extracts elements from the end of the queue.
 - For reliability, **BRPOP** can be used to block extraction.

```
# Add a task to the queue
```

```
LPUSH "queue:tasks" '{"task": "send_email", "to": "user@example.com"}'
```

```
# Get the length of the queue
```

```
LLEN "queue:tasks"
```

```
# Limit the queue size to 1000 elements
```

```
LTRIM "queue:tasks" 0 999
```

```
# Wait and extract a task
```

```
BRPOP "queue:tasks" 0
```

7. Application Configuration Storage

- Implementation details:
 - Uses hashes to store settings.
 - Each configuration section can be a separate hash.
 - **HSET** sets or updates settings.
 - **HGETALL** retrieves all settings of a section.

```
HSET "config:app" debug_mode "true"  
HSET "config:app" max_connections "1000"  
HGETALL "config:app"
```

Output:

```
1) "debug_mode"  
2) "true"  
3) "max_connections"  
4) "1000"
```

We want to place banners on the page at a specific position and rotate them evenly; after each page reloads, they change.

```
# Add a banner to the rotation  
ZADD banners 0 {banner}  
# Return a banner with fewer views  
ZRANGE banners 1  
# Increase banner's views  
ZINCRBY banners 1 {banner}  
# Remove an outdated or irrelevant banner from rotation  
ZREM banners {banner}  
# Get the total number of banners in rotation  
ZCARD banners  
# Get up to 10 banners with view counts between 0 and 100  
ZRANGEBYSCORE banners 0 100 LIMIT 0 10
```

Purchasing and payment

Top up balance

ZINCRBY balance 500 user:1234

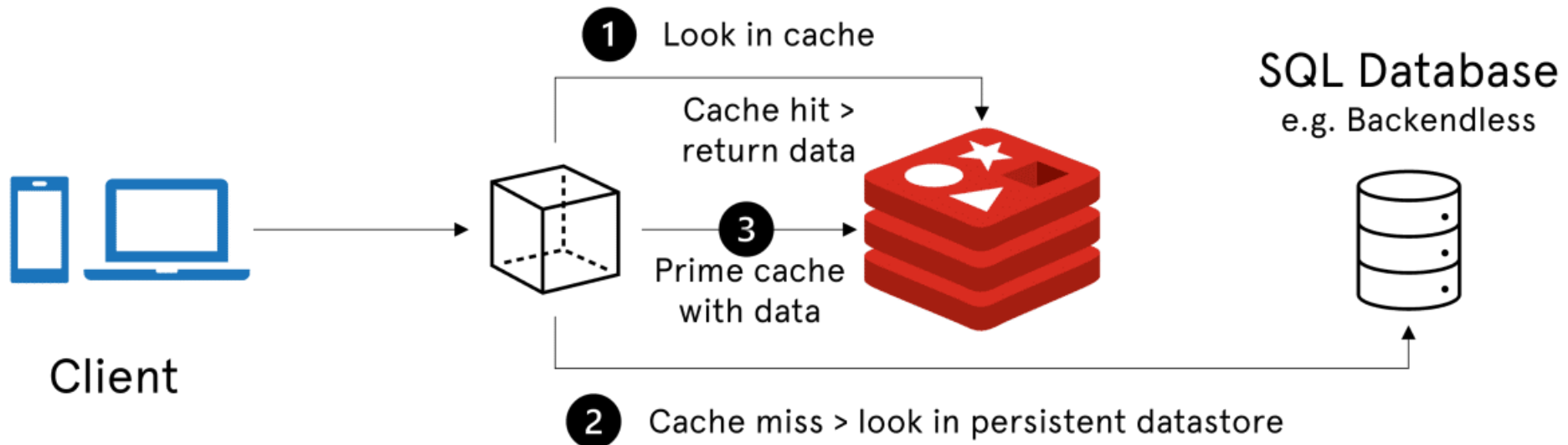
Withdraw the sum

ZINCRBY balance -500 user:1234

Add information to a log

ZADD purchases 1634567890 "product:1234:quantity:2:price:100"

How Redis is typically used



Source: <https://backendless.com>

Redis as cache

```
def get_tourists():
    # Check the cache first
    key = "tourists"
    tourists = redis.get(key)
    if tourists:
        # The tourists are in the cache, return them
        return tourists.decode('utf-8')

        # The tourists are not in the cache, query the database
        cursor = pool.cursor()
        cursor.execute("SELECT * FROM tourists")
        tourists = cursor.fetchall()

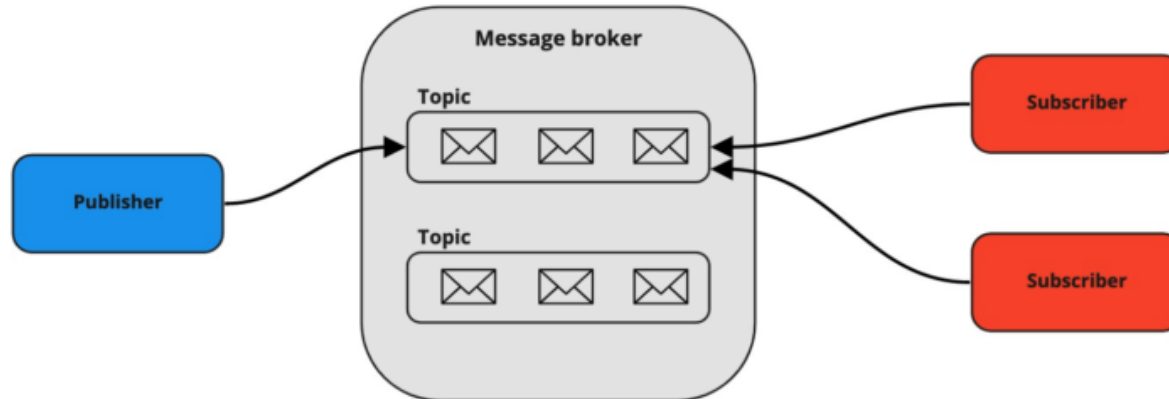
        # Save the tourists to the cache
        redis.set(key, str(tourists))

    return tourists

tourists1 = get_tourists()
print('Tourists from DB:', tourists1, '\n')

# Will be retrieved faster because of caching
tourists2 = get_tourists()
print('Tourists from CACHE:', tourists1)
```

Publish / Subscribe



miro

<https://hevodata.com>

For notifications and alerts

Redis is also a message broker that supports typical pub/sub operations.

- The first user subscribes to certain channel , "news"

SUBSCRIBE news

- Another user sends messages to the same channel , "news"

PUBLISH news "hello"

- Subscribed clients receive the message:

- 1) "message"
- 2) "news"
- 3) "hello"

Subscribers can listen to multiple channels, and publishers can send to multiple channels.

- Learn more at <http://redis.io/commands#pubsub>

Redis Streams

- A data structure for stream processing
- Designed to store multiple records ordered by insertion time

Main Characteristics:

- Append-only logs
- Unique IDs for each entry (timestamp + sequence number)
- Consumer Groups support
- Built-in persistence

Applications:

- Event logging
- Message processing
- Real-time analytics
- Event sourcing
- Messaging systems

Read more: <https://redis.io/docs/latest/develop/data-types/streams/>

Redis Streams: Basic commands

XADD mystream [ID] field1 value1 [field2 value2 ...]

Add new entry

XREAD [COUNT count] [BLOCK milliseconds] STREAMS key [key ...] ID [ID ...]

Read stream entries

XRANGE key start end [COUNT count]

Read stream entries in a specific range

XLEN key

Get stream length

XGROUP CREATE key groupname id [MKSTREAM]

Creating and Managing Groups

XREADGROUP GROUP group consumer [COUNT count] STREAMS key [key ...] ID [ID ...]

Reading as Consumer Group

XACK key group ID [ID ...]

Message Acknowledgment

Redis Streams: Examples

XADD race:france * rider Castilla speed 30.2 position 1 location_id 1 "1692632086370-0"

Add a stream entry for each racer that includes the racer's name, speed, position, and location ID

XREAD COUNT 100 BLOCK 300 STREAMS race:france \$

Read up to 100 new stream entries, starting at the end of the stream, and block for up to 300 ms if no entries are being written .

XRANGE race:france 1692632086370-0 + COUNT 2

Read two stream entries starting at ID 1692632086370-0

XLEN race:france

Get the number of items inside a Stream

XGROUP CREATE race:france france_riders \$

Create a consumer group

XREADGROUP GROUP italy_riders Alice COUNT 1 STREAMS race:italy >

Add riders to the race:italy stream and try reading something using the consumer group

XACK race:italy italy_riders 1692632639151-0

Acknowledges processing of message ID 1692632639151-0

Redis Bitfields

- Space-efficient data structure for storing multiple counters/integers
- Allows manipulation of integer values at the bit level
- Introduced to handle binary data and numeric arrays efficiently

Key Features:

- Multiple integers packed into a single Redis key
- Support for different integer sizes (1-63 bits)
- Atomic operations guaranteed
- Signed and unsigned integers support
- Memory efficient storage

Use Cases :

- Rate limiting
- User presence tracking
- Performance metrics
- Feature flags

Read more: <https://redis.io/docs/latest/develop/data-types/bitfields/>

Redis Bitfields : Basic command structure

BITFIELD key [GET type offset] [SET type offset value] [INCRBY type offset increment]

Type Specification:

- i[bits] – signed integer (i8, i16, i32)
- u[bits] – unsigned integer (u8, u16, u32)

Overflow Handling:

- WRAP: wrap around values (default)
- SAT: saturate at min/max
- FAIL: return null on overflow

Command Options:

- GET: retrieve integer values
- SET: set integer values
- INCRBY: increment values
- OVERFLOW: set overflow behavior

Redis Bitfields : Examples

BITFIELD mykey SET u8 0 42

Sets an 8-bit unsigned integer at offset 0 to value 42

BITFIELD mykey SET u8 0 42 SET u8 8 24

Sets two 8-bit integers: 42 at offset 0 and 24 at offset 8

BITFIELD mykey OVERFLOW SAT INCRBY u8 0 100

Increments value by 100, saturating at maximum value (255 for u8)

BITFIELD mykey GET u8 0 GET u8 8

Retrieves two 8-bit integers from offsets 0 and 8

BITFIELD mykey OVERFLOW WRAP

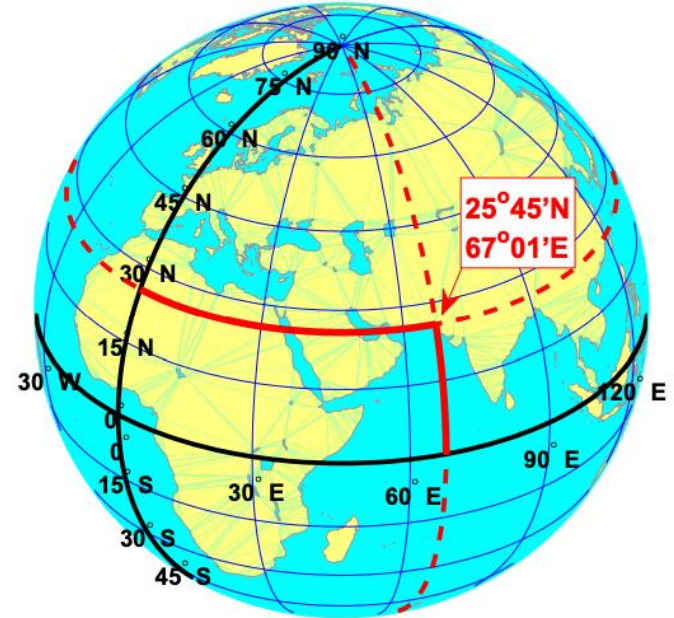
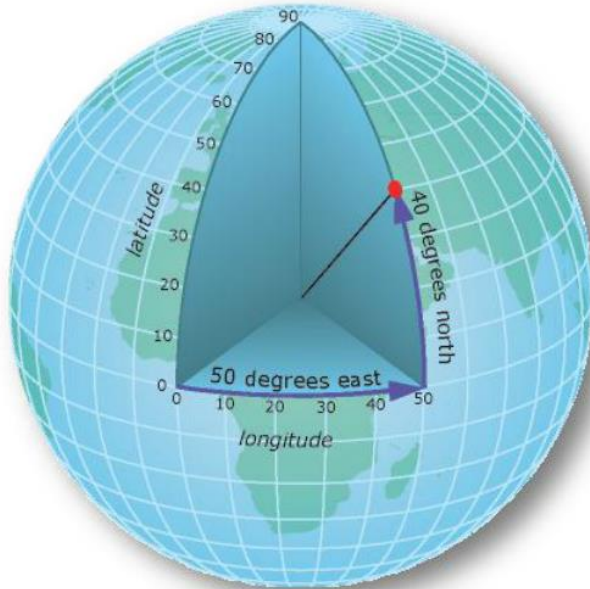
SET i8 0 100

INCRBY u16 8 1

GET i8 0

Sets 8-bit signed integer to 100, increments 16-bit unsigned integer at offset 8, retrieves the first value, uses WRAP overflow behavior.

GEOSPATIAL



Redis **geospatial** indexes let you store **coordinates** and search for them. This data structure is useful for finding nearby points within a given radius or bounding box.

GEOADD

adds a location to a given geospatial index
(note that **longitude** comes before **latitude** with this command).

GEOPOS

Returns the position of one or more members in a geospatial index.

GEOSEARCH

Returns locations with a given radius or a bounding box.

GEODIST

Returns the distance between two members in the geospatial index represented by the sorted set.

GEORADIUS

Queries a sorted set representing a geospatial index to fetch members within a given distance.

Redis example – Data format

For Redis geospatial commands, the correct format is longitude followed by latitude. Examples:

12.4964 41.9028

12.4964, 41.9028

Valid ranges:

Longitude: -180 to 180

Latitude: -85.05112878 to 85.05112878

The first number is the **longitude**, and the second is the **latitude**.

An option like

longitude 2.2945 latitude 48.8584

is not in the correct format for Redis geospatial commands.

Redis uses geohashing internally for efficient storage and querying of coordinates. Coordinate precision in Redis is limited to 5-6 decimal places.

Redis example – Filter by location

GEOADD Addresses 43.361389 18.115556 "Addr1"
25.087269 37.502669 "Addr2"

GEODIST Addresses Addr1 Addr2 km
Distance between two addresses

GEOSEARCH Addresses FROMLONLAT 15 37 BYRADIUS 15 km ASC
Everything within a 15-kilometer radius of the point

Read more about geospatial: <https://redis.io/docs/data-types/geospatial/>

Transactions

Transaction

- All commands are serialized and executed sequentially
- Either all commands or no commands are processed
- Keys must be explicitly specified in Redis transactions
- Redis does not support transactions between multiple shards.
- Redis commands for transactions:
 - ✓ **WATCH**
 - Marks the given keys to be watched for conditional execution of a transaction.
 - ✓ **MULTI**
 - Marks the start of a transaction block. Subsequent commands will be **queued** for atomic execution using **EXEC**.
 - ✓ **DISCARD**
 - Flushes all previously queued commands in a transaction
 - ✓ **EXEC**
 - Executes all previously queued commands in a transaction
 - If a watched key has been modified, the transaction will fail, no command will be executed
 - ✓ **UNWATCH**
 - Forgets about all watched keys

Transaction - Example

*# We update the player's score,
their position on the leaderboard,
and set a TTL for the player's data*

MULTI

OK

HINCRBY user:1234 score 50

QUEUED

ZINCRBY leaderboard 50 "player1234"

QUEUED

EXPIRE user:1234 86400

QUEUED

EXEC

- 1) (integer) 250 # New player score
- 2) (float) 1430.5 # New leaderboard position
- 3) (integer) 1 # Successfully set key expiration

*# We attempt to update the player's
inventory and decrease their gold,
but the transaction is discarded*

MULTI

OK

SADD inventory:5678 "health_potion"

QUEUED

SREM inventory:5678 "empty_bottle"

QUEUED

HINCRBY user:5678 gold -10

QUEUED

DISCARD

OK

Transaction – Example (all or nothing)

MULTI

```
SET key1 "value1"  
INCR key2  
SADD set1 "member1"  
INCR nonexistent_key  
SET key3 "value3"
```

EXEC

- 1) OK
- 2) (integer) 1
- 3) (integer) 1
- 4) (integer) 1
- 5) OK

MULTI

```
MULTI  
SET key1 "value1"  
INCR key2  
SADD set1 "member1"  
INCRBY key4 # Error: missing second argument  
SET key3 "value3"
```

EXEC

MULTI

```
+ OK  
SET key1 "value1"  
+ QUEUED  
INCRBY key2 # Syntax error: missing second argument  
- ERR wrong number of arguments for 'incrby' command  
SET key3 "value3"
```

```
+ QUEUED
```

EXEC

```
- EXECABORT Transaction discarded because of previous errors.
```

Transaction – Example

SET nonexistent_string "abc"

OK

MULTI

OK

SET key1 "value1"

QUEUED

INCR key2

QUEUED

HSET hash1 field1 "value"

QUEUED

LPUSH list1 "item1"

QUEUED

SADD set1 "member1"

QUEUED

INCR nonexistent_string

QUEUED

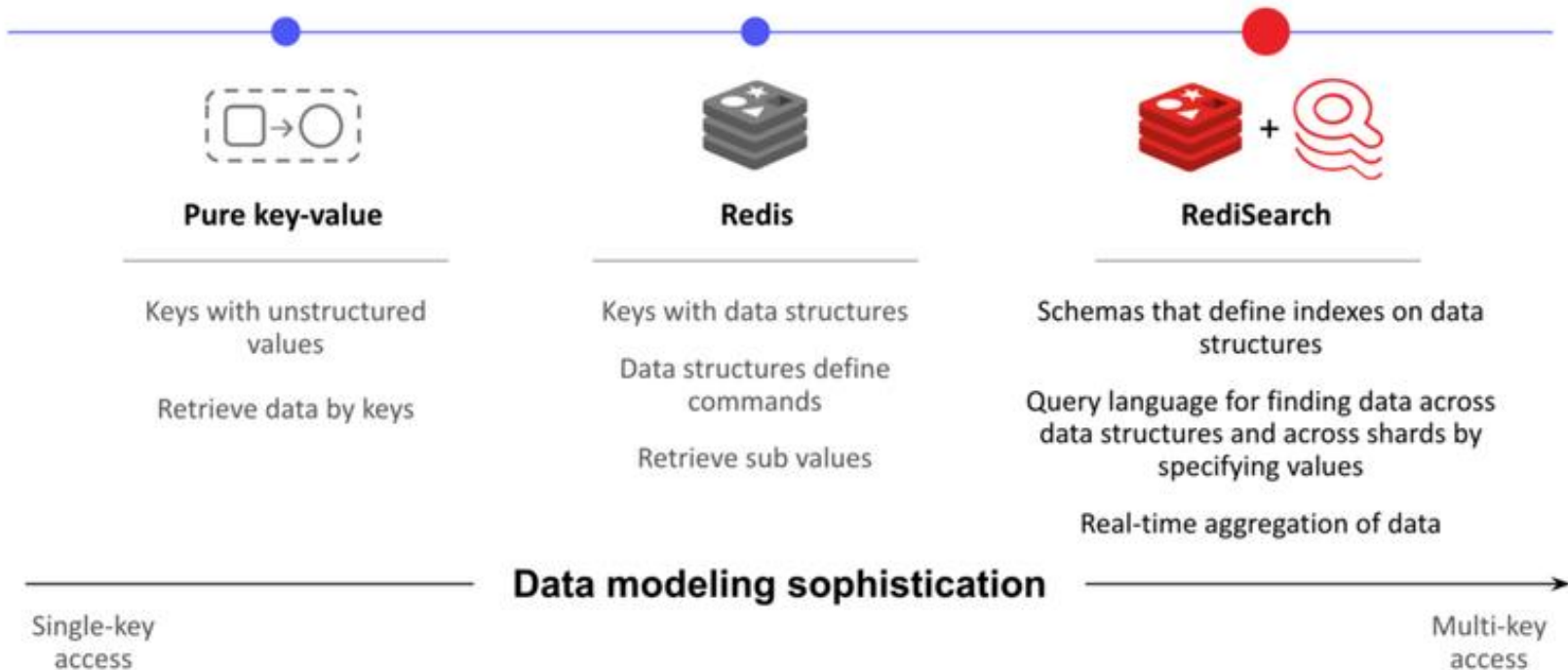
EXEC

- 1) OK
- 2) (integer) 1
- 3) (integer) 1
- 4) (integer) 1
- 5) (integer) 1
- 6) (error) ERR value is not an integer or out of range

Transaction - Errors inside a transaction

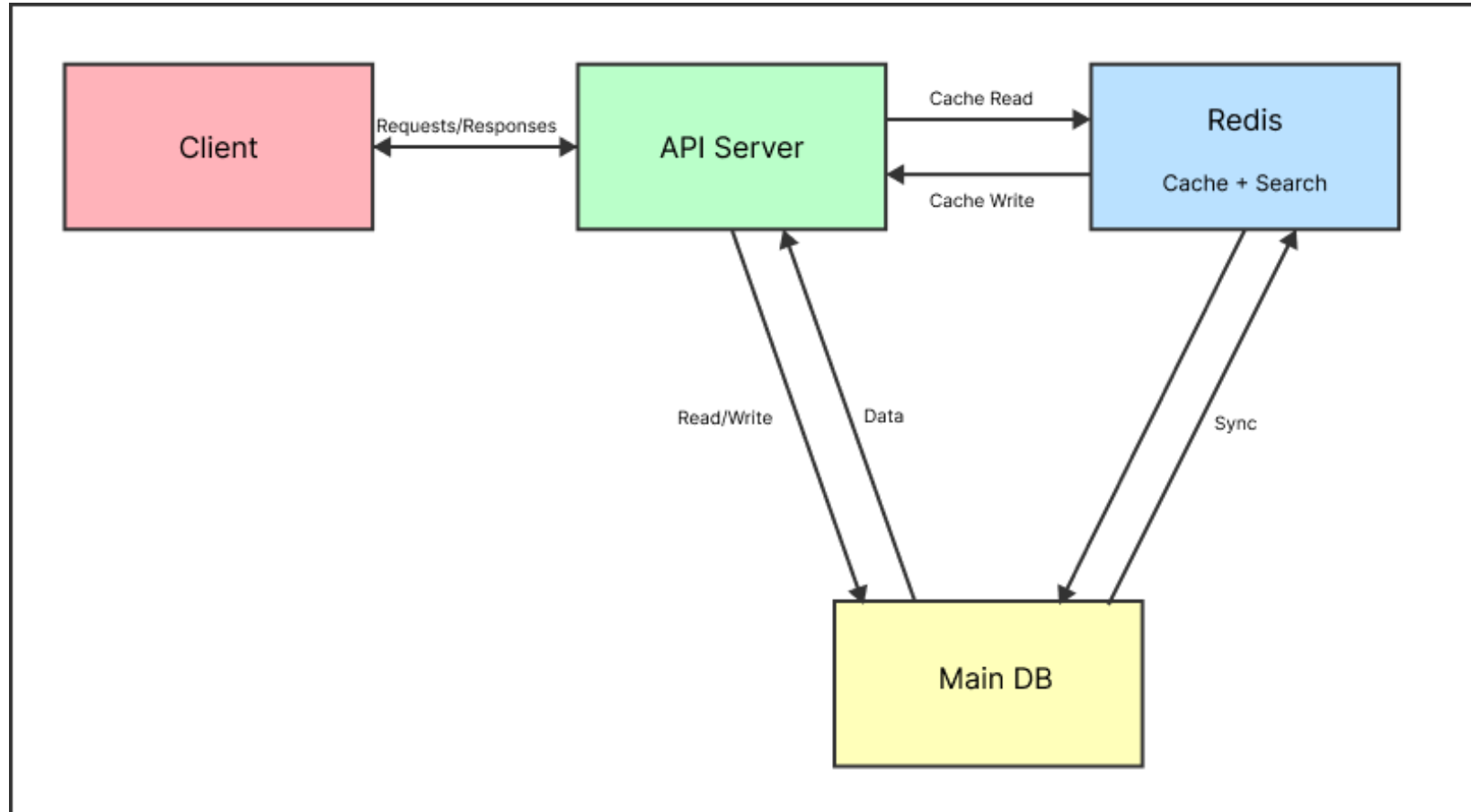
- ✓ Before EXEC is called
 - The command may be syntactically wrong (wrong number of arguments, wrong command name, ...),
 - There may be some critical conditions like an out-of-memory condition.
- ✓ After EXEC is called
 - If we operated against a key with the wrong value (like calling a list operation against a string value).
- ❑ Redis does not support rollbacks of transactions.
- ❑ DISCARD can be used to abort a transaction. In this case, no commands are executed, and the state of the connection is restored to normal.

Redisearch



Source: <https://redis.io/blog/redisearch-2-build-modern-applications-interactive-search/>

System Architecture with Redis



- ✓ Redisearch is a **Secondary Index** over Redis
 - Takes a document and breaks it apart
 - Maps terms/properties and gets a list of terms
 - Searching is getting documents that are linked to the terms
- ✓ **Full-Text** engine (Prefix, Fuzzy, Phonetic, Stemming, Synonyms...)
- ✓ **Incremental indexing**
 - Updates index in real-time without performance degradation
- ✓ Data **aggregation**
 - Supports grouping, sorting, and filtering of search results
- ✓ Auto-complete **suggestions**
 - Provides real-time suggestions as users type
- ✓ **Geo** indexing and filtering
 - Enables location-based search and filtering

Inverted index

Document 1

(BSD licensed), in-memory data structure store used as a database, cache, message broker, and streaming engine. Redis

Document 2

You can run atomic operations on these types, like appending to a string; incrementing the value in a hash; pushing an element to a

Document 3

To achieve top performance, Redis works with an in-memory dataset. Depending on your use case, Redis can persist

Stopword list

a
achieve
also
an
and
appending
as
atomic
automatic
availability
bitmaps
broker
BSD
built-in
by
cache
can
case
Cluster
command
computing
data

Inverted index

1	a	1, 2, 3
2	achieve	3
3	also	3
4	an	1, 2, 3
5	and	1, 2
6	appending	2, 3
7	as	1
8	atomic	2
9	automatic	1
10	availability	1
11	bitmaps	1
12	broker	1
13	BSD	1
14	built-in	1
15	by	3
16	cache	1, 3
17	can	2, 3
18	case	3
19	Cluster	1
20	command	3
21	computing	2
22	data	1, 3

Redisearch is a search module for Redis that provides advanced full-text search capabilities.

1. Creating an index (FT.CREATE):

- **FT.CREATE** index_name
ON HASH|JSON
PREFIX number prefix1 [prefix2 ...]
SCHEMA field_name [TEXT|TAG|NUMERIC|GEO|VECTOR] [SORTABLE]
[NOINDEX] ...
- Searching is getting documents that are linked to the terms.

2. Adding documents:

- For hashes: use standard Redis **HSET** commands
- For JSON: use **JSON.SET**

3. Searching (FT.SEARCH):

- **FT.SEARCH** index_name "query_string" [NOCONTENT] [VERBATIM]
[NOSTOPWORDS] [WITHSCORES] [WITHPAYLOADS] [WITHSORTKEYS]...

Query syntax:

Creating an index (FT.CREATE):

- Simple text search: "hello world"
- Field-specific search: @field:value
- Boolean operators: AND, OR, NOT
- Phrase search: "hello world"
- Prefix search: he*
- Fuzzy search: %hello%
- Numeric ranges: @price:[100 200]
- Geo search: @location:[-73.98 40.75 10 km]

Redisearch - example

```
HSET user:1 name "Anna" year "2000" friends "Tom, Michael, Helen"
```

```
HSET user:2 name "Alex" year "1998" friends "Jakub, Helen"
```

```
HSET user:3 name "Sandra" year "1999" friends "Jonas"
```

```
FT.CREATE usr_ind prefix 1 user: SCHEMA name TEXT year NUMERIC friends TEXT
```

Search for users whose friend is Helen

```
FT.SEARCH usr_ind "Helen"
```

- 1) (integer) 2
- 2) "user:1"
- 3) 1) "name"
 - 2) "Anna"
 - 3) "year"
 - 4) "2000"
 - 5) "friends"
 - 6) "Tom, Michael, Helen"
- 4) "user:2"
- 5) 1) "name"
 - 2) "Alex"
 - 3) "year"
 - 4) "1998"
 - 5) "friends"
 - 6) "Jakub, Helen"

```
FT.SEARCH usr_ind "@friends:Helen"
```

- 1) (integer) 2
- 2) "user:1"
- 3) 1) "name"
 - 2) "Anna"
 - 3) "year"
 - 4) "2000"
 - 5) "friends"
 - 6) "Tom, Michael, Helen"
- 4) "user:2"
- 5) 1) "name"
 - 2) "Alex"
 - 3) "year"
 - 4) "1998"
 - 5) "friends"
 - 6) "Jakub, Helen"

- Perform a search query, filtering for records you wish to process.
- Build a pipeline of operations that transform the results by zero or more steps of:
 - **Group and Reduce:** grouping by fields in the results, and applying reducer functions on each group.
 - **Sort:** sort the results based on one or more fields.
 - **Apply Transformations:** Apply mathematical and string functions on fields in the pipeline, optionally creating new fields or replacing existing ones
 - **Limit:** Limit the result, regardless of sorting the result.
 - **Filter:** Filter the results (post-query) based on predicates relating to its values.

- Aggregations (FT.AGGREGATE):
 - **FT.AGGREGATE** index_name query
 - [LOAD number @field1 [@field2 ...]]
 - [GROUPBY number @property1 @property2 ...]
 - [REDUCE function NUMBER_OF_ARGS @field [AS name]]
 - [SORTBY number @field [ASC|DESC]]
 - [APPLY expression AS name]
 - [LIMIT offset num]
 - Updating schema (FT.ALTER)

Aggregation - example

Log of visits to our website, each record containing the following fields/properties:

- **url** (text, sortable)
- **timestamp** (numeric, sortable) - unix timestamp of visit entry.
- **country** (tag, sortable)
- **user_id** (text, sortable, not indexed)

Select all records in the index, group the results by hour, and count the distinct user IDs in each hour.

Then, format the hour as a human-readable timestamp

```
FT.AGGREGATE myIndex "*"
  APPLY "@timestamp - (@timestamp % 3600)" AS hour
  GROUPBY 1 @hour
    REDUCE COUNT_DISTINCT 1 @user_id AS num_users
  SORTBY 2 @hour ASC
  APPLY timefmt(@hour) AS hour
```

RedisJSON

The JSON capability of Redis Stack provides JavaScript Object Notation (JSON) support for Redis.

- Store, update, and retrieve JSON values.
 - Index and query JSON documents.
 - Support for nested JSON structures.
- ✓ Full support for the JSON standard
 - ✓ A **JSONPath** syntax for selecting/updating elements inside documents
 - ✓ Documents are stored as binary data in a tree structure, allowing fast access to sub-elements
 - ✓ Typed atomic operations for all JSON value types

Read more about Redis JSON: <https://redis.io/docs/latest/develop/data-types/json/>

JSONPath is a query language for JSON. Some key features of JSONPath:

The root node (\$) – a JSON structure's root member.

Current node (@) – the node currently being processed.

Child operator (. or []) – child members of an object or array.

Recursive descent (..) – all children of the current node, regardless of depth.

Wildcard (*) – all objects/elements irrespective of their names.

Subscript operator ([]) – Iterate over element collections and for predicates².

Union operator ([,]) - a combination of node sets.

Array slice operator ([start:end: step]) – selects elements from an array.

Filter operator (? ()) - filter (script) expression.

Read more about JSONPath: <https://goessner.net/articles/JsonPath/>

JSONPath Online Evaluator: <https://jsonpath.com>

JSON.SET key path value [NX | XX]

- Sets the JSON value at path in key
 - *Key* is a key to modify
 - *Path* is JSONPath to specify. The default is root \$
 - *Value* is value to set at the specified path
 - *NX* sets the key only if it does not already exist
 - *XX* sets the key only if it already exists.

JSON.GET key [INDENT indent] [NEWLINE newline] [SPACE space] [path [path ...]]

- Returns the value at path in JSON serialized form

Example:

```
JSON.SET doc $ '{"a":2, "b": 3, "nested": {"a": 4, "b": null}}'
```

```
OK
```

```
JSON.GET doc $..b    # $..b matches all objects with a key of b, regardless of depth  
"[3,null]"
```

RedisJSON - Example

```
JSON.SET item:1 $ '{"name":"Noise-cancelling Bluetooth headphones",  
"description":"Wireless Bluetooth headphones with noise-cancelling technology",  
"connection":{"wireless":true,"type":"Bluetooth"},"price":99.98,"stock":25,  
"colors":["black","silver"],"embedding":[0.87,-0.15,0.55,0.03]}'
```

OK

```
JSON.SET item:2 $ '{"name":"Wireless earbuds",  
"description":"Wireless Bluetooth in-ear headphones",  
"connection":{"wireless":true,"type":"Bluetooth"},"price":64.99,"stock":17,  
"colors":["black","white"],"embedding":[-0.7,-0.51,0.88,0.14]}'
```

OK

<https://redis.io/docs/interact/search-and-query/indexing/>

RedisJSON - Example

Examples of retrieving data

JSON.GET item:1

```
"{\"name\":\"Noise-cancelling Bluetooth headphones\", \"description\":\"Wireless Bluetooth headphones with noise-cancelling technology\", \"connection\": {\"wireless\":true, \"type\":\"Bluetooth\"}, \"price\":99.98, \"stock\":25, \"colors\":[\"black\", \"silver\"], \"embedding\":[0.87,-0.15,0.55,0.03]}"
```

JSON.GET item:1 \$.name

```
"[\"Noise-cancelling Bluetooth headphones\"]"
```

JSON.GET item:2 \$.name \$.price \$.stock

```
"{\"$.stock\":[17],\"$.price\":[64.99],\"$.name\":[\"Wireless earbuds\"]}"
```

JSON.GET item:1 \$.connection.type

```
"[\"Bluetooth\"]"
```

JSON.GET item:1 \$.colors[0]

```
"[\"black\"]"
```

RedisJSON - Example

Example of updating data

JSON.SET item:1 \$.weight 250

OK

JSON.NUMINCRBY item:1 \$.stock 5

"[30]"

JSON.ARRAPPEND item:2 \$.colors ""red""

1) (integer) 3

JSON.DEL item:2 \$.embedding

(integer) 1

JSON.ARRLEN item:1 \$.colors

1) (integer) 2

JSON.TYPE item:1 \$.connection.wireless

1) "boolean"

```
FT.CREATE {index_name} ON JSON SCHEMA {json_path} AS {attribute} {type}
```

Example:

Create an index that indexes the name, description, price, and image vector embedding of each JSON document that represents an inventory item.

```
FT.CREATE itemIdx  
ON JSON PREFIX 1 item: SCHEMA $.name AS name TEXT $.description as  
description TEXT $.price AS price NUMERIC $.embedding AS embedding VECTOR  
FLAT 6 DIM 4 DISTANCE_METRIC L2 TYPE FLOAT32
```

<https://redis.io/docs/interact/search-and-query/indexing/>

item:1

name:

"Noise-cancelling Bluetooth headphones"

description:

"Wireless Bluetooth headphones with noise-cancelling technology"

connection:

wireless: true

type: "Bluetooth"

price: 99.98

stock: 25

colors: ["black", "silver"]

embedding:

[0.87, -0.15, 0.55, 0.03]

item:2

name:

"Wireless earbuds"

description:

"Wireless Bluetooth in-ear headphones"

connection:

wireless: true

type: "Bluetooth"

price: 64.99

stock: 17

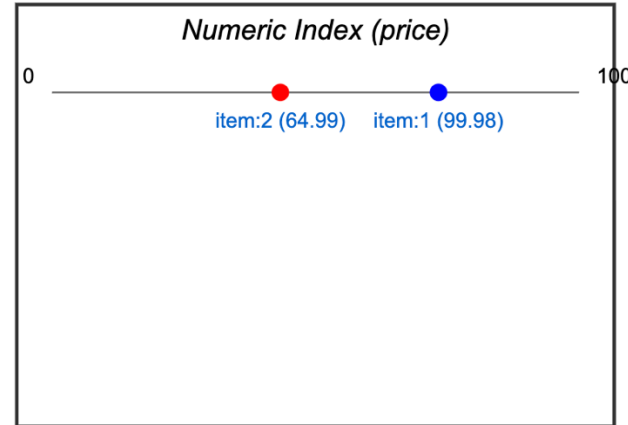
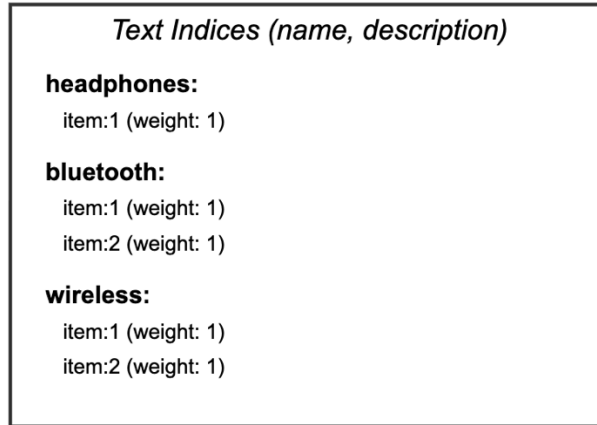
colors: ["black", "white"]

embedding:

[-0.7, -0.51, 0.88, 0.14]

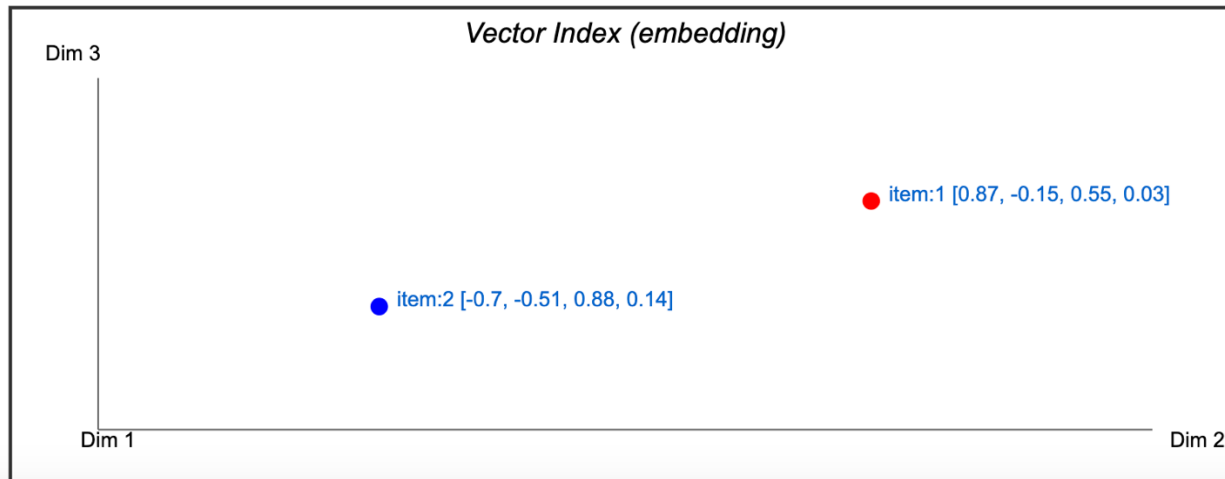
Conceptual Structure of itemIdx Index

Uses an inverted index for fast keyword search



Usually implemented as an ordered tree for efficient range searching

Uses specialized data structures (e.g., HNSW) to find nearest neighbors



Search for earbuds:

```
FT.SEARCH itemIdx '@name:(earbuds)'
```

1) (integer) 1

2) "item:2"

3) 1) "\$"

2) "{\"name\":\"Wireless **earbuds**\",

\"description\":\"Wireless Bluetooth in-ear headphones\",

\"connection\":{\"wireless\":true,\"type\":\"Bluetooth\"},

\"price\":64.99,\"stock\":17,\"colors\":[\"black\",\"white\"],

\"embedding\":[-0.7,-0.51,0.88,0.14]}\"

RedisJSON - Search the index

Search for Bluetooth headphones with a price of less than 70:

```
FT.SEARCH itemIdx '@description:(bluetooth headphones) @price:[0 70]'
```

1) (integer) 1

2) "item:2"

3) 1) "\$"

2) "{\"name\":\"Wireless earbuds\",

\"description\":\"Wireless Bluetooth in-ear headphones\",

\"connection\":{\"wireless\":true,\"type\":\"Bluetooth\"},\"price\":64.99,\"stock\":17,

\"colors\":[\"black\",\"white\"],\"embedding\":[-0.7,-0.51,0.88,0.14]}"

Read more:

<https://redis.io/docs/interact/search-and-query/query/>

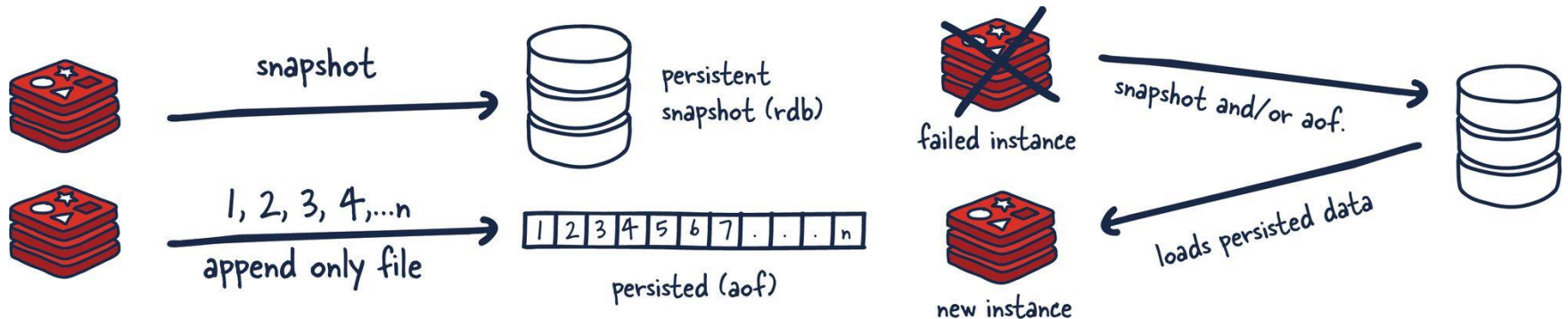
<https://redis.io/docs/interact/search-and-query/indexing/>

PERSISTENCE

Persistence

Datasets can be saved to disk

Persistence refers to the writing of data to durable storage, such as a solid-state disk (SSD).



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

Redis provides a range of persistence options. These include:

- **RDB** (Redis Database): RDB persistence performs point-in-time snapshots of your dataset at specified intervals.
- **AOF** (Append Only File): AOF persistence logs every write operation the server receives. These operations can then be replayed at server startup, reconstructing the original dataset.
- **No persistence**: You can disable persistence completely. This is sometimes used when caching.
- **RDB + AOF**: You can combine AOF and RDB in the same instance.

Example 1: RDB Persistence with Custom Save Points

In **redis.conf**, set the following options

save 900 1

Save the DB if at least 1 key changed in 900 seconds

save 300 10

Save the DB if at least 10 keys changed in 300 seconds

save 60 10000

Save the DB if at least 10000 keys changed in 60 seconds

Example 2: AOF Persistence with Every Second fsync

In **redis.conf**, set the following options

appendonly yes

Enable AOF persistence

appendfsync everysec

fsync every second

aof-use-rdb-preamble yes

Optimize AOF loading in Redis 7.0+

Example: RDB + AOF Persistence with No fsync

In **redis.conf**, set the following options

save 3600 1

Save the DB if at least 1 key changed in 3600 seconds

appendonly yes

Enable AOF persistence

appendfsync no

Do not fsync, leave it to the OS

RDB advantages and disadvantages

- ✓ RDB is a very compact single file for backups and disaster recovery.
 - ✓ RDB maximizes Redis's performance since the only work Redis's parent process needs to do to persist is forking a child who will do all the rest. The parent process will never perform disk I/O or similar work.
 - ✓ RDB allows faster restarts with big datasets than AOF.
-
- **Data Loss:** RDB snapshots are taken periodically, which means that in case of a system crash, you could lose data that has not yet been included in the most recent snapshot.
 - **Forking Overhead:** The Redis process needs to fork a child process to create the RDB snapshot, which can be resource-intensive for large datasets.

AOF advantages and disadvantages

- ✓ **Better Durability:** AOF provides better data durability, as it logs every write operation, reducing the risk of data loss.
- ✓ **Human-Readable Format:** AOF files store the commands in plain text, making them easy to inspect and understand.
- ✓ **Flexible Configuration:** You can configure the AOF fsync policy to balance durability and performance based on your requirements.

Disadvantages of AOF

- **Larger File Size:** AOF files can be significantly larger than RDB files, as they store every write operation.
- **Slower Recovery:** The recovery process for AOF can be slower than that of RDBs, as Redis needs to replay all the logged commands to reconstruct the dataset.

Persistence: RDB vs AOF

RDB (Redis Database File)	AOF (Append Only File)
Provides point in time snapshots	Logs every write
Creates complete snapshot at specified interval	Replays at server startup. If log gets big, optimization takes place
File is in binary format	File is easily readable
On crash minutes of data can be lost	Minimal chance of data loss
Small files, fast (mostly)	Big files, 'slow'

Source: <https://www.slideshare.net/MaartenSmeets1/introduction-redis-93365594>

BGSAVE

Save the DB in the background. Redis forks, the parent continues serving the clients, and the child saves the dataset on disk and exits.

SAVE

Perform a synchronous save of the dataset. Other clients are blocked – never use in production!

LASTSAVE

Return the Unix time of the last successful DB save.

BGREWRITEAOF

Instruct Redis to start an AOF rewrite process. The rewrite will create a small optimized version of the current AOF log.

If **BGREWRITEAOF** fails, no data gets lost, as the old AOF will be untouched

Read more about persistence:

https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/

What persistence is used for?

- ✓ **Backups**
- ✓ **Disaster Recovery**
- ✓ **Performance Maximization**
- ✓ **Faster Restarts with Big Datasets**
- ✓ **Replicas**

Summary. Why Redis?

- ✓ **In-Memory Data Storage**
- ✓ **Data Structure Support**
- ✓ **Persistence Options**
- ✓ **Pub/Sub Messaging**
- ✓ **Caching**
- ✓ **Distributed Architecture**
- ✓ **Extensibility**