**GSP763**



**Overview**

[Redis](https://redis.io/) is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker. It is designed for applications requiring the fastest possible response times. To manage data, Redis uses key/pair values, and those keys point to any number of data structures. Some of the more popular data structures include [strings](https://redis.io/commands#string), [hashes](https://redis.io/commands#hash), [sets](https://redis.io/commands#set), and [lists](https://redis.io/commands#list).

In this lab you will use Redis and [RediSearch](https://oss.redislabs.com/redisearch/" \t "_blank), an add-on module that enables full-text search and secondary indexing in Redis, to install and run a Redis instance, import example hashes, index them with RediSearch, and query them.

Using Redis hashes

Application developers often use Redis hashes to represent their domain objects. A hash is just a set of field/value pairs. For instance, suppose you wanted to store a catalog of films in your Redis database. You might store the information for a Star Wars film in a hash with the HSET command:

HSET movie:001 title "Star Wars: Episode IV - A New Hope" director "George Lucas" plot "The galaxy is in a period of civil war...."

Copied!

content\_copy

This command stores a hash at the key "movie:001". The hash contains three fields: title, director, and plot.

Now, if you want to access this data, you need to know the name of the key at which it's stored. To retrieve this hash, you can run the HGETALL command:

HGETALL movie:001

Copied!

content\_copy

This will return all of the movie data for *Star Wars*.

Using RediSearch

What if you want to retrieve hashes based on their contents, not just based on their key? What if, for example, you want to get a list of all movies directed by George Lucas? RediSearch is the tool that makes this possible.

RediSearch indexes the data you store in your Redis hashes and provides you with a query language for looking up that data. In effect, RediSearch makes Redis into a much more general purpose database by letting you run complex queries and aggregations on your data.

What you'll learn

In this lab, you'll learn how to work with Redis and RediSearch by performing the following tasks:

* Install and run a Redis instance with RediSearch using Docker
* Connect to Redis and insert some sample data
* Index and query the database using FT.CREATE and FT.SEARCH
* Insert, update, delete, and expire Redis hashes
* Import movie, theater, and user datasets
* Query the datasets using simple and complex conditions, sorting, pagination, and counting
* Aggregate the datasets using FT.AGGREGATE, and apply functions to queries

**Setup and requirements**

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

To complete this lab, you need:

* Access to a standard internet browser (Chrome browser recommended).

**Note:** Use an Incognito or private browser window to run this lab. This prevents any conflicts between your personal account and the Student account, which may cause extra charges incurred to your personal account.

* Time to complete the lab---remember, once you start, you cannot pause a lab.

**Note:** If you already have your own personal Google Cloud account or project, do not use it for this lab to avoid extra charges to your account.

How to start your lab and sign in to the Google Cloud Console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is the **Lab Details** panel with the following:
   * The **Open Google Console** button
   * Time remaining
   * The temporary credentials that you must use for this lab
   * Other information, if needed, to step through this lab
2. Click **Open Google Console**. The lab spins up resources, and then opens another tab that shows the **Sign in** page.

***Tip:*** Arrange the tabs in separate windows, side-by-side.

**Note:**If you see the **Choose an account** dialog, click **Use Another Account**.

1. If necessary, copy the **Username** from the **Lab Details** panel and paste it into the **Sign in** dialog. Click **Next**.
2. Copy the **Password** from the **Lab Details** panel and paste it into the **Welcome** dialog. Click **Next**.

**Important:**You must use the credentials from the left panel. Do not use your Google Cloud Skills Boost credentials.**Note:**Using your own Google Cloud account for this lab may incur extra charges.

1. Click through the subsequent pages:
   * Accept the terms and conditions.
   * Do not add recovery options or two-factor authentication (because this is a temporary account).
   * Do not sign up for free trials.

After a few moments, the Cloud Console opens in this tab.

**Note:** You can view the menu with a list of Google Cloud Products and Services by clicking the **Navigation menu** at the top-left. 

Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell** Activate Cloud Shell icon at the top of the Google Cloud console.

When you are connected, you are already authenticated, and the project is set to your **PROJECT\_ID**. The output contains a line that declares the **PROJECT\_ID** for this session:

Your Cloud Platform project in this session is set to YOUR\_PROJECT\_ID

gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

1. (Optional) You can list the active account name with this command:

gcloud auth list

Copied!

content\_copy

1. Click **Authorize**.
2. Your output should now look like this:

**Output:**

ACTIVE: \*

ACCOUNT: student-01-xxxxxxxxxxxx@qwiklabs.net

To set the active account, run:

$ gcloud config set account `ACCOUNT`

1. (Optional) You can list the project ID with this command:

gcloud config list project

Copied!

content\_copy

**Output:**

[core]

project = <project\_ID>

**Example output:**

[core]

project = qwiklabs-gcp-44776a13dea667a6

**Note:**For full documentation of gcloud, in Google Cloud, refer to [the gcloud CLI overview guide](https://cloud.google.com/sdk/gcloud).

**Task 1. Install a Redis database with RediSearch**

1. To start, you'll need to get a Redis database with RediSearch enabled. For simplicity, this lab will use pre-built Docker images. To start your Redis instance with Docker, in Cloud Shell run the following command:

docker run -it --rm --name redis-search-2 -p 6379:6379 redislabs/redisearch:2.0.0

Copied!

content\_copy

1. In a new cloud shell tab, clone the [Getting Started with RediSearch](https://github.com/RediSearch/redisearch-getting-started) repository (used later for importing datasets in the lab), and install the redis-cli, which is a simple program that allows you to send commands to Redis, and read the replies sent by the server, directly from the terminal:

git clone https://github.com/RediSearch/redisearch-getting-started

sudo apt-get install redis-tools

Copied!

content\_copy

You now have a Redis instance running with RediSearch installed! In the next sections you will discover the basics.

**Task 2. Connect to Redis and insert data**

Before creating the index, you will first need to understand the dataset and insert some example entries. For this lab, you will use a simple dataset describing movies. A movie has the following attributes:

|  |  |
| --- | --- |
| **Attribute** | **Description** |
| movie\_id | the unique ID of the movie, internal to this database |
| title | the title of the movie |
| plot | a summary of the movie |
| genre | the genre of the movie (for now a movie will only have single genre) |
| release\_year | the year the movie was released (as a numerical value) |
| rating | the average rating from the public (numerical value) |
| votes | the number of votes |
| poster | link to the movie poster |
| imdb\_id | ID of the movie in the [IMDB](https://imdb.com/) database |

As a Redis developer, one of the first things you need to decide is how to structure your data. This starts with *key naming*.

A common way of naming keys in Redis is to use a specific pattern. For example, when the database includes various business objects like movies, actors, theaters, and users, you can use the following pattern: business\_object\_type:ID.

For example:

* movie:001 would be the key for the movie with the ID 001
* user:001 would be the key for the user with the ID 001

For the movies data itself, you should use Redis [hashes](https://redis.io/topics/data-types#hashes). A Redis hash will allow the application to structure all of the movie attributes in individual fields; RediSearch will then index those fields based on an index definition.

Insert movies

It's now time to add some data into your database.

1. In the **new** Cloud Shell tab, run the following command to use the redis-cli embedded in the container that connects to the server running in the other tab. Make sure to leave the other tab open to keep the Redis instance running.

docker exec -it redis-search-2 redis-cli

Copied!

content\_copy

1. You're now ready to insert some data. This example uses movie data stored as Redis hashes, so start by inserting a few movies (note that you can copy these commands rather than manually typing them in):

HSET movie:11002 title "Star Wars: Episode V - The Empire Strikes Back" plot "After the Rebels are brutally overpowered by the Empire on the ice planet Hoth, Luke Skywalker begins Jedi training with Yoda, while his friends are pursued by Darth Vader and a bounty hunter named Boba Fett all over the galaxy." release\_year 1980 genre "Action" rating 8.7 votes 1127635 imdb\_id tt0080684

Copied!

content\_copy

HSET movie:11003 title "The Godfather" plot "The aging patriarch of an organized crime dynasty transfers control of his clandestine empire to his reluctant son." release\_year 1972 genre "Drama" rating 9.2 votes 1563839 imdb\_id tt0068646

Copied!

content\_copy

HSET movie:11004 title "Heat" plot "A group of professional bank robbers start to feel the heat from police when they unknowingly leave a clue at their latest heist." release\_year 1995 genre "Thriller" rating 8.2 votes 559490 imdb\_id tt0113277

Copied!

content\_copy

HSET "movie:11005" title "Star Wars: Episode VI - Return of the Jedi" genre "Action" votes 906260 rating 8.3 release\_year 1983 plot "The Rebels dispatch to Endor to destroy the second Empire's Death Star." ibmdb\_id "tt0086190"

Copied!

content\_copy

1. Now you can retrieve this data using the movie ID. Run the following command to get the title and rating:

HMGET movie:11002 title rating

Copied!

content\_copy

**Output:**

1) "Star Wars: Episode V - The Empire Strikes Back"

2) "8.7"

1. You can increment the rating of this movie using the following command:

HINCRBYFLOAT movie:11002 rating 0.1

Copied!

content\_copy

**Output:**

"8.8"

But how do you get a movie or list of movies by release year, rating value, or title?

With “core” Redis, you would have to manually index this data using other Redis data structures (e.g., sets). This is relatively complex. With RediSearch, you can simply define an index associated with your data and let the database do all the indexing for you. You can then query your data using the RediSearch query syntax.

RediSearch and indexing

To create an index, you must define a schema specifying the fields to index. For this example, you'll be indexing four fields:

* Title
* Release Year
* Rating
* Genre

When creating an index you define:

* Which data you want to index: for instance, all *hashes* with a key starting with movies
* Which fields in the hashes you want to index

**Warning: Do not index all fields.** Indexes take space in memory, and must be updated when the primary data is updated. So be sure to only index the fields that you absolutely need to query on.

Create the index

1. Create the index with the following command:

FT.CREATE idx:movie ON hash PREFIX 1 "movie:" SCHEMA title TEXT SORTABLE release\_year NUMERIC SORTABLE rating NUMERIC SORTABLE genre TAG SORTABLE

Copied!

content\_copy

Before any running queries, take a closer look at the FT.CREATE command.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| idx:movie | the name of the index |
| ON hash | the type of data structure to be indexed |
| PREFIX 1 "movie: | the prefix of the keys that should be indexed. This is a list, so since you want to only index movie:\* keys, the number is 1. With two key prefixes, your definition would look like this: PREFIX 2 "movie:" "tv\_show:" |
| SCHEMA ... | defines the fields, their types, and whether the index should be sortable. As you can see in the command, possible types are [TEXT](https://redis.io/docs/stack/search/reference/query_syntax/#a_few_query_examples), [NUMERIC](https://redis.io/docs/stack/search/reference/query_syntax/#numeric-filters-in-query) and [TAG](https://redis.io/docs/stack/search/reference/query_syntax/#tag-filters). [SORTABLE](https://redis.io/docs/stack/search/reference/sorting/) is an additional parameter. |

The RediSearch engine will scan the database using the PREFIX values, and update the index based on the schema definition. This makes it easy to add an index to an existing application that uses hashes. You can find more information about the [FT.CREATE](https://oss.redislabs.com/redisearch/Commands/#ftcreate) command in the [documentation](https://oss.redislabs.com/redisearch/Commands/#ftcreate).

1. Next, look at the index information with the FT.INFO command:

FT.INFO idx:movie

Copied!

content\_copy

You're ready to use the index and query the database.

Test your understanding

Below are multiple choice questions to reinforce your understanding of this lab's concepts. Answer them to the best of your abilities.

1. RediSearch indexes data on Redis \_\_\_\_\_\_\_\_\_\_\_\_ data structures.



hash



list



set



string

Submit

2. Suppose you create a RediSearch index, defining it as follows:  
"FT.CREATE idx:films ON hash PREFIX 1 "film:" SCHEMA title TEXT. Which hashes will be included in this index?



All hashes whose keys contain the word "film"



All hashes in the Redis database



All hashes containing the text "film:"



All hashes whose keys begin with "film:"

Submit

3. When you create a RediSearch index, you must declare a SCHEMA. What does the SCHEMA do?



Defines the hash fields to index and their types



Defines how to validate hashes when they're saved to the database



Ensures that only those hashes having the exact fields defined in the schema will be indexed

Submit

**Task 3. Query data**

In this section, you'll learn how to query data using the FT.SEARCH command.

1. Start by searching for all movies that contain “war”-related information by running the following command:

FT.SEARCH idx:movie "war"

Copied!

content\_copy

**Output:**

1) (integer) 2

2) "movie:11005"

3) 1) "title"

2) "Star Wars: Episode VI - Return of the Jedi"

...

14) "tt0086190"

4) "movie:11002"

5) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

...

13) "imdb\_id"

14) "tt0080684"

The FT.SEARCH command returns the number of results, then the list of each individual result.

As you can see, the movie *Star Wars: Episode V - The Empire Strikes Back* is found, even though you used only the word “war” to match “Wars” in the title. This is because the title has been indexed as text, so the field is [tokenized](https://redis.io/docs/stack/search/reference/escaping/) and [stemmed](https://redis.io/docs/stack/search/reference/stemming/).

1. You can also limit which fields are returned by using the RETURN parameter. Run the same query, but this time return only the title and release\_year:

FT.SEARCH idx:movie "war" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 2

2) "movie:11005"

3) 1) "title"

2) "Star Wars: Episode VI - Return of the Jedi"

3) "release\_year"

4) "1983"

4) "movie:11002"

5) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

3) "release\_year"

4) "1980"

This query does not specify a field, so the word “war” (and related words) is searched in all text fields of the index. You can also limit your query to specific fields.

1. To query on a specific field, use the @field: syntax:

FT.SEARCH idx:movie "@title:war" RETURN 2 title release\_year

Copied!

content\_copy

1. Now try a more complex query. Suppose you want all movies containing the string *war* but NOT the string jedi. Adding the string -jedi (minus) will tell the query engine to exclude fields containing the word jedi.

FT.SEARCH idx:movie "war -jedi" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 1

2) "movie:11002"

3) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

3) "release\_year"

4) "1980"

1. Next, query all the movies that contain the string gdfather using fuzzy search. You can request fuzzy matching by surrounding your search term with % signs:

FT.SEARCH idx:movie " %gdfather% " RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 1

2) "movie:11003"

3) 1) "title"

2) "The Godfather"

3) "release\_year"

4) "1972"

As you can see, even though the word "godfather" is incorrectly spelled, this query is still possible using a [fuzzy matching](https://redis.io/docs/stack/search/reference/query_syntax/#fuzzy-matching). Fuzzy matches are performed based on [Levenshtein distance](https://en.wikipedia.org/wiki/Levenshtein_distance" \t "_blank) (LD).

1. Query all Thriller movies. The genre field is indexed as a TAG and allows exact match queries. The syntax to query a TAG field is @field\_name:{value}.

FT.SEARCH idx:movie "@genre:{Thriller}" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 1

2) "movie:11004"

3) 1) "title"

2) "Heat"

3) "release\_year"

4) "1995"

1. Next, query all Thriller **or** Action movies; this is done with the | symbol:

FT.SEARCH idx:movie "@genre:{Thriller|Action}" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 3

2) "movie:11004"

3) 1) "title"

2) "Heat"

3) "release\_year"

4) "1995"

4) "movie:11005"

5) 1) "title"

2) "Star Wars: Episode VI - Return of the Jedi"

3) "release\_year"

4) "1983"

6) "movie:11002"

7) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

3) "release\_year"

4) "1980"

To learn more, refer to [Tag filters in the Query syntax reference](https://redis.io/docs/stack/search/reference/query_syntax/#tag-filters).

1. Building on the last query, you can select all Thriller or Action movies that do not have Jedi in the title:

FT.SEARCH idx:movie "@genre:{Thriller|Action} @title:-jedi" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 2

2) "movie:11004"

3) 1) "title"

2) "Heat"

3) "release\_year"

4) "1995"

4) "movie:11002"

5) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

3) "release\_year"

4) "1980"

Next, query all the movies released between 1970 and 1980 (inclusive). For this, the FT.SEARCH syntax has two ways to query numeric fields:

* Using the FILTER parameter
* Using the @field in the query string.

1. First, try the FILTER parameter:

FT.SEARCH idx:movie \* FILTER release\_year 1970 1980 RETURN 2 title release\_year

Copied!

content\_copy

1. Now use the @field parameter:

FT.SEARCH idx:movie "@release\_year:[1970 1980]" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 2

2) "movie:11003"

3) 1) "title"

2) "The Godfather"

3) "release\_year"

4) "1972"

4) "movie:11002"

5) 1) "title"

2) "Star Wars: Episode V - The Empire Strikes Back"

3) "release\_year"

4) "1980"

1. To exclude a value, prepend it with ( in the FILTER or query string, for example to exclude 1980:

FT.SEARCH idx:movie "@release\_year:[1970 (1980]" RETURN 2 title release\_year

Copied!

content\_copy

Summary

* By default, queries apply to all TEXT fields and use the words and their base, stemmed form
* To restrict your query to a specific field, you must use the @field: notation
* Multiple conditions "intersection" (AND condition) and "union" (OR condition, using |) are possible

Test your understanding

Below are multiple choice questions to reinforce your understanding of this lab's concepts. Answer them to the best of your abilities.

1. What does the following query return?  
FT.SEARCH idx:movie "future"



All indexed hashes where any field contains the word "future"



All indexed hashes whose keys contain the word "future"



All indexed hashes having a field that exactly matches the word "future"



All indexed hashes with at least one field named "future"

Submit

2. Suppose you want to look up all movies whose title contains "world" but not "Jurassic". Which query would you run?



FT.SEARCH idx:movie "world !jurassic"



FT.SEARCH idx:movie "world"



FT.SEARCH idx:movie "@title:world -jurassic"



FT.SEARCH idx:movie "@title:world"

Submit

3. Imagine you want to look up all "Horror" movies released between 1960 and 1980. How do you express this query?



FT.SEARCH idx:movie "@genre:horror @release\_year:[1960 1980]"



FT.SEARCH idx:movie "@genre:{horror} && @release\_year:[1960 1980]"



FT.SEARCH idx:movie "@genre:horror && @release\_year > 1960 < 1980"



FT.SEARCH idx:movie "@genre:{horror} @release\_year:[1960 1980]"

Submit

**Task 4. Insert, update, delete, and expire documents**

In this section you will see how RediSearch responds when you insert new documents, update old ones, delete some others, and what happens when documents expire.

As part of this lab you have:

* Created a few movies as Redis hashes (documents) and stored them using the movie:[ID] key pattern
* Created an index using the FT.CREATE command
* Queried the data using FT.SEARCH

When creating the index, using the idx:movie ON hash PREFIX 1 "movie:" parameter, you are asking the indexing engine to look at all existing keys and index them. As such, new information that matches this pattern/type, will also be indexed.

1. Start by counting the number of movies:

FT.SEARCH idx:movie "\*" LIMIT 0 0

Copied!

content\_copy

**Output:**

1) (integer) 4

1. Now, add a new movie:

HSET movie:11033 title "Tomorrow Never Dies" plot "James Bond sets out to stop a media mogul's plan to induce war between China and the U.K in order to obtain exclusive global media coverage." release\_year 1997 genre "Action" rating 6.5 votes 177732 imdb\_id tt0120347

Copied!

content\_copy

1. Count them again:

FT.SEARCH idx:movie "\*" LIMIT 0 0

Copied!

content\_copy

**Output:**

1) (integer) 5

As you can see, the new movie has been indexed.

1. Next, search on any of the indexed fields:

FT.SEARCH idx:movie "never" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 1

2) "movie:11033"

3) 1) "title"

2) "Tomorrow Never Dies"

3) "release\_year"

4) "1997"

1. Now, run the following to *update* one of the fields:

HSET movie:11033 title "Tomorrow Never Dies - 007"

Copied!

content\_copy

1. And search for 007:

FT.SEARCH idx:movie "007" RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 1

2) "movie:11033"

3) 1) "title"

2) "Tomorrow Never Dies - 007"

3) "release\_year"

4) "1997"

When you *delete* the hash, the index is also updated, and the same happens if the key is expired (for example, by using a TTL, or Time To Live).

1. Set a 20-second TTL on the James Bond movie:

EXPIRE "movie:11033" 20

Copied!

content\_copy

1. You can run the following query and see that after 20 seconds, the document does not exist anymore:

FT.SEARCH idx:movie "007" RETURN 2 title release\_year

Copied!

content\_copy

The search query will not return any result, showing that the index has been updated:

**Output:**

1) (integer) 0

**Note:**When you're using Redis as your primary database you're not necessarily using the TTLs to delete records. TTLs are common for caching use cases. But another interesting use case is [ephemeral search](https://redislabs.com/blog/the-case-for-ephemeral-search/), where you need lightweight, fast expiration.

**Task 5. Managing indexes**

In this section, you will learn how to list, inspect, and update your indexes.

1. Use the FT.\_LIST command to get a list of all RediSearch indexes of your database:

FT.\_LIST

Copied!

content\_copy

**Output:**

1) "idx:movie"

1. Use the FT.INFO command to get information about the index:

FT.INFO "idx:movie"

Copied!

content\_copy

**Output:**

1) "index\_name"

2) "idx:movie"

...

5) "index\_definition"

...

7) "fields"

...

9) "num\_docs"

10) "4"

...

Updating your index definitions

As you are building your application and adding more information to the database, you may need to add new fields to the index. Adding a new field is possible using the FT.ALTER command.

1. Use the FT.ALTER command to add a new field to the movie index:

FT.ALTER idx:movie SCHEMA ADD plot TEXT WEIGHT 0.5

Copied!

content\_copy

The WEIGHT parameter declares the importance of this field when calculating results. This is a multiplication factor (default is 1); so in this example the plot is less important than the title.

1. Now, run a query with the new indexed field:

FT.SEARCH idx:movie "empire @genre:{Action}" RETURN 2 title plot

Copied!

content\_copy

1. You can now drop the index using the FT.DROPINDEX command:

FT.DROPINDEX idx:movie

Copied!

content\_copy

1. Dropping the index does **not** impact the indexed hashes; this means that the movies are still inside the database. Check this by running the following query:

SCAN 0 MATCH movie:\*

Copied!

content\_copy

**Output:**

1) "0"

2) 1) "movie:11002"

2) "movie:11004"

3) "movie:11003"

4) "movie:11005"

**Note:** You can delete the indexed document/hashes by adding the DD parameter to the FT.DROPINDEX command.

Test your understanding

Below are multiple choice questions to reinforce your understanding of this lab's concepts. Answer them to the best of your abilities.

1. True or false: Changing the contents of an indexed Redis hash also updates the index.



True



False

Submit

2. True or false: Dropping an index deletes all of the indexed hashes.



True



False

Submit

3. True or false: If you need to make changes to an index definition, you must first drop the index and then recreate it.



True



False. You can modify the definition of an index using the FT.ALTER command

Submit

**Task 6. Import sample datasets**

In the previous sections you used just a few movies to perform some basic queries. In this section, you will import existing datasets that include more movies to discover more query features, theaters to discover the geospatial capabilities, and users to do some aggregations.

Movies

The file sample-app/redisearch-docker/dataset/import\_actors.redis is a script that creates 922 Hashes.

The **movie** hashes contain the following fields:

|  |  |
| --- | --- |
| **Field** | **Description** |
| movie\_id | the unique ID of the movie, internal to this database |
| title | the title of the movie |
| plot | a summary of the movie |
| genre | the genre of the movie (for now a movie will only have single genre) |
| release\_year | the year the movie was released (as a numerical value) |
| rating | the average rating from the public (numerical value) |
| votes | the number of votes |
| poster | link to the movie poster |
| imdb\_id | ID of the movie in the [IMDB](https://imdb.com/) database |

Theaters

The file sample-app/redisearch-docker/dataset/import\_theaters.redis is a script that creates 117 Hashes (used for Geospatial queries).

The **theater** hashes contain the following fields:

|  |  |
| --- | --- |
| **Field** | **Description** |
| theater:id | the unique ID of the theater, internal to this database (used as the key to the hash) |
| name | the name of the theater |
| address | the street address |
| city | the city (in this sample dataset all the theaters are in New York) |
| zip | the zip code |
| phone | the phone number |
| url | the URL of the theater |
| location | the longitude and latitude used to create the geo-indexed field |

Users

The file sample-app/redisearch-docker/dataset/import\_users.redis is a script that creates 5996 Hashes.

The **user** hashes contain the following fields:

|  |  |
| --- | --- |
| **Field** | **Description** |
| user:id | the unique ID of the user |
| first\_name | the first name of the user |
| last\_name | the last name of the user |
| email | the email address of the user |
| gender | the gender of the user |
| country | the country name |
| country\_code | the country code |
| city | the city of the user |
| longitude | the longitude of the user |
| latitude | the latitude of the user |
| last\_login | the last login of the user, in EPOCH time |
| ip\_address | the IP address of the user |

Import the movies, theaters, and users

1. Before importing the data, flush the database using the FLUSHALL command:

FLUSHALL

Copied!

content\_copy

1. The easiest way to import the files is to use redis-cli. Open a new Cloud Shell tab and run the following commands:

cd redisearch-getting-started

redis-cli -h localhost -p 6379 < ./sample-app/redisearch-docker/dataset/import\_movies.redis

redis-cli -h localhost -p 6379 < ./sample-app/redisearch-docker/dataset/import\_theaters.redis

redis-cli -h localhost -p 6379 < ./sample-app/redisearch-docker/dataset/import\_users.redis

Copied!

content\_copy

1. Return to the previous Cloud Shell tab, with the redis-cli prompt open, and run the following to take a look at the dataset:

HMGET "movie:343" title release\_year genre

Copied!

content\_copy

**Output:**

1) "Spider-Man"

2) "2002"

3) "Action"

1. Get the theater name and location at index 20:

HMGET "theater:20" name location

Copied!

content\_copy

**Output:**

1) "Broadway Theatre"

2) "-73.98335054631019,40.763270202723625"

1. Get the data associated with user number 343:

HMGET "user:343" first\_name last\_name last\_login

Copied!

content\_copy

**Output:**

1) "Umeko"

2) "Castagno"

3) "1574769122"

You can also use the DBSIZE command to see how many keys you have in your database.

Create indexes

1. Before you can do more queries, you'll need to create the idx:movie index with the following command:

FT.CREATE idx:movie ON hash PREFIX 1 "movie:" SCHEMA title TEXT SORTABLE plot TEXT WEIGHT 0.5 release\_year NUMERIC SORTABLE rating NUMERIC SORTABLE votes NUMERIC SORTABLE genre TAG SORTABLE

Copied!

content\_copy

1. The movies have now been indexed. Run the following command and look at the num\_docs returned value. The value should be 922.

FT.INFO "idx:movie"

Copied!

content\_copy

1. Create the idx:theater index. This index will mostly be used to show the geospatial capabilities of RediSearch. In the previous examples you created indexes with fields having three different types:

* Text
* Numeric
* Tag

1. You will now discover a new type of field: Geo. The theater hashes contains a field location with the longitude and latitude that will be used in the index as follows:

FT.CREATE idx:theater ON hash PREFIX 1 "theater:" SCHEMA name TEXT SORTABLE location GEO

Copied!

content\_copy

1. The theaters have now been indexed. Run the following command and look at the num\_docs returned value. The value should be 117.

FT.INFO "idx:theater"

Copied!

content\_copy

1. Lastly, create the idx:user index:

FT.CREATE idx:user ON hash PREFIX 1 "user:" SCHEMA gender TAG country TAG SORTABLE last\_login NUMERIC SORTABLE location GEO

Copied!

content\_copy

**Task 7. Querying the movie dataset**

As described earlier in the lab, one of the goals of RediSearch is to provide rich querying capabilities such as *simple and complex conditions*, *sorting*, *pagination*, and *counting*. This section will provide some examples and insight into the various capabilities of querying a sizable dataset.

Conditions

The best way to start to work with RediSearch's query capabilities is to look at the various conditions options.

1. Run the following to find all the movies that contain the word 'heat':

FT.SEARCH "idx:movie" "heat" RETURN 2 title plot

Copied!

content\_copy

**Output:**

1) (integer) 4

2) "movie:1141"

3) 1) "title"

2) "Heat"

3) "plot"

4) "A group of professional bank robbers start to feel the heat from police when they unknowingly leave a clue at their latest heist."

4) "movie:818"

5) 1) "title"

2) "California Heat"

3) "plot"

4) "A lifeguard bets he can be true to just one woman."

6) "movie:736"

7) 1) "title"

2) "Chicago Justice"

3) "plot"

4) "The State's Attorney's dedicated team of prosecutors and investigators navigates heated city politics and controversy head-on,while fearlessly pursuing justice."

8) "movie:1109"

9) 1) "title"

2) "Love & Hip Hop: Miami"

3) "plot"

4) "'Love and Hip Hop Miami' turns up the heat and doesn't hold back in making the 305 the place to be. Multi-platinum selling hip-hop legend Trick Daddy is back in the studio collaborating ..."

The first line contains the number of documents (4) that match the query condition, then the list of movies.

This query is a "fieldless" condition, meaning that the query engine has:

* Searched all of the TEXT fields from the index (title and plot)
* For the word heat and related words, returned movie:736 since it has the word heated in the plot. To learn more, refer to the [Stemming reference](https://redis.io/docs/stack/search/reference/stemming/).
* Returned the result sorted by the score. Remember that the title has a weight of 1.0, and the plot a weight of 0.5. So when the word or related words are found in the title, the score is bigger.

1. Now, find all the movies with a **title** that contain the word "heat" or related to "heat". This is a field-specific query:

FT.SEARCH "idx:movie" "@title:heat" RETURN 2 title plot

Copied!

content\_copy

**Output:**

1) (integer) 2

2) "movie:1141"

3) 1) "title"

2) "Heat"

3) "plot"

4) "A group of professional bank robbers start to feel the heat from police when they unknowingly leave a clue at their latest heist."

4) "movie:818"

5) 1) "title"

2) "California Heat"

3) "plot"

4) "A lifeguard bets he can be true to just one woman."

Sort

A common use case when querying data is to sort the data on a specific field, and paginate over the result.

* Query all the Action movies, sorted by release year from newest to the oldest.

FT.SEARCH "idx:movie" "@genre:{Action}" SORTBY release\_year DESC RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 186

2) "movie:360"

3) 1) "release\_year"

2) "2019"

3) "title"

4) "Spider-Man: Far from Home"

...

20) "movie:272"

21) 1) "release\_year"

2) "2016"

3) "title"

4) "13 Hours"

The first line contains the number of documents (186) that match the query condition.

The FT.SEARCH command, by default, returns the first ten documents. You will see in the next query how to paginate.

Paginate

1. Query all the Action movies, sorted by release year from the oldest to the most recent one, returning the record by batch of 100 movies:

FT.SEARCH "idx:movie" "@genre:{Action}" LIMIT 0 100 SORTBY release\_year ASC RETURN 2 title release\_year

Copied!

content\_copy

**Output:**

1) (integer) 186

2) "movie:892"

3) 1) "release\_year"

2) "1966"

3) "title"

4) "Texas,Adios"

...

200) "movie:6"

201) 1) "release\_year"

2) "2014"

3) "title"

4) "John Wick"

The result is very similar to the previous query:

* 186 documents found
* The first document is the oldest one, released in 1966
* The latest movie of the batch is released in 2014

1. To paginate to the next batch, you'll need to change the limit as follows:

FT.SEARCH "idx:movie" "@genre:{Action}" LIMIT 100 200 SORTBY release\_year ASC RETURN 2 title release\_year

Copied!

content\_copy

Count

* Count the number of Action movies. Based on the sample queries you've already seen, if you put the LIMIT 0 0, RediSearch will return the number of documents that match the query condition:

FT.SEARCH "idx:movie" "@genre:{Action}" LIMIT 0 0

Copied!

content\_copy

**Output:**

1) (integer) 186

Geospatial queries

Suppose you're at MOMA, located at "11 W 53rd St, New York", and you want to find all the theaters located within a 400 meter radius. For this, you need to start with the location (-73.9798156,40.7614367) (long/lat) of your current location.

* Find theaters, names, and addresses that are at less than 400 meters from MOMA:

FT.SEARCH "idx:theater" "@location:[-73.9798156 40.7614367 400 m]" RETURN 2 name address

Copied!

content\_copy

**Output:**

1) (integer) 5

2) "theater:115"

3) 1) "name"

2) "Winter Garden Theatre"

3) "address"

4) "1634 Broadway"

...

10) "theater:88"

11) 1) "name"

2) "Snapple Theater Center"

3) "address"

4) "1627 Broadway"

**Task 8. Aggregation**

A common need for an application, in addition to retrieving information as a document list, is aggregation. For example, if you look at the movie documents, you may want to retrieve the number of movies and group by release year, starting with the most recents ones. For this, RediSearch provides the FT.AGGREGATE command.

GROUPBY & SORTBY

1. Start by aggregating the number of movies by year:

FT.AGGREGATE "idx:movie" "\*" GROUPBY 1 @release\_year REDUCE COUNT 0 AS nb\_of\_movies

Copied!

content\_copy

**Output:**

1) (integer) 60

2) 1) "release\_year"

2) "1964"

3) "nb\_of\_movies"

4) "9"

...

61) 1) "release\_year"

2) "2010"

3) "nb\_of\_movies"

4) "15"

1. Building on the last command, aggregate the number of movies by year, from the newest to the oldest:

FT.AGGREGATE "idx:movie" "\*" GROUPBY 1 @release\_year REDUCE COUNT 0 AS nb\_of\_movies SORTBY 2 @release\_year DESC

Copied!

content\_copy

**Output:**

1) (integer) 60

2) 1) "release\_year"

2) "2019"

3) "nb\_of\_movies"

4) "14"

...

11) 1) "release\_year"

2) "2010"

3) "nb\_of\_movies"

4) "15"

Apply functions

The idx:user index contains the last\_login field. This field stores the last login time as an EPOCH timestamp. RediSearch aggregation allows you to apply transformations to each record. This is done using the [APPLY](https://redis.io/docs/stack/search/reference/aggregations/#apply-expressions) parameter.

For this example, you can use a [date/time](https://redis.io/docs/stack/search/reference/aggregations/#list-of-datetime-apply-functions) function to extract the month and year from the timestamp.

1. Run the following command to retrieve the number of logins per year and month:

FT.AGGREGATE idx:user \* APPLY year(@last\_login) AS year APPLY "monthofyear(@last\_login) + 1" AS month GROUPBY 2 @year @month REDUCE count 0 AS num\_login SORTBY 4 @year ASC @month ASC

Copied!

content\_copy

**Output:**

1) (integer) 13

2) 1) "year"

2) "2019"

3) "month"

4) "9"

5) "num\_login"

6) "230"

...

14) 1) "year"

2) "2020"

3) "month"

4) "9"

5) "num\_login"

6) "271"

1. Use filtering to retrieve the number of logins per month for the year 2020. This is similar to the previous query with the addition of a filter on the year:

FT.AGGREGATE idx:user \* APPLY year(@last\_login) AS year APPLY "monthofyear(@last\_login) + 1" AS month GROUPBY 2 @year @month REDUCE count 0 AS num\_login FILTER "@year==2020" SORTBY 2 @month ASC

Copied!

content\_copy

**Output:**

1) (integer) 13

2) 1) "year"

2) "2020"

3) "month"

4) "1"

5) "num\_login"

6) "520"

...

10) 1) "year"

2) "2020"

3) "month"

4) "9"

5) "num\_login"

6) "271"

Test your understanding

Below are multiple choice questions to reinforce your understanding of this lab's concepts. Answer them to the best of your abilities.

1. Suppose you want to count the number of "Horror" movies in your database. What query should you run?



FT.AGGREGATE "idx:movie" "@genre:{Horror}"



FT.COUNT "idx:movie" "@genre:{Horror}"



FT.COUNT "idx:movie" "@genre:{Horror}" LIMIT 0 0



FT.SEARCH "idx:movie" "@genre:{Horror}" LIMIT 0 0

Submit

2. What does the following query do?  
FT.AGGREGATE "idx:movie" "world" GROUPBY 1 @genre REDUCE COUNT 0 AS num



Returns the total number of indexed movies containing the term "world"



Returns the number of movies belonging to at least one genre



Returns the number of movies in each genre having "world" in the title



Returns the number of movies in each genre with an indexed field containing the term "world"

Submit

**Task 9. Advanced features**

In the previous examples, the indices have been created using a PREFIX, where all the keys matching the data structure type and prefix are indexed.

But you can also create an index using a filter. For example, you can create an index with all the "Drama" movies released between 1990 and 2000 (2000 not included).

The FILTER expression uses the [aggregation filter syntax](https://redis.io/docs/stack/search/reference/aggregations/#filter-expressions). Here's a filter expression on genre and release year:

FILTER "@genre=='Drama' && @release\_year>=1990 && @release\_year<2000"

Copied!

content\_copy

1. Here's how to create a drama movie index using this FILTER expression:

FT.CREATE idx:drama ON Hash PREFIX 1 "movie:" FILTER "@genre=='Drama' && @release\_year>=1990 && @release\_year<2000" SCHEMA title TEXT SORTABLE release\_year NUMERIC SORTABLE

Copied!

content\_copy

1. Run the FT.INFO idx:drama command to look at the index definitions and statistics:

FT.INFO idx:drama

Copied!

content\_copy

Notes:

* The PREFIX is not optional.
* In this application, this index is not useful since you can get the same data from the idx:movie

Finally, you can check that the index has been correctly initialized by running the following queries that should return the same number of documents.

1. Check the number of documents for idx:drama:

FT.SEARCH idx:drama " @release\_year:[1990 (2000]" LIMIT 0 0

Copied!

content\_copy

**Output:**

1) (integer) 24

1. Now check the number of documents for idx:movie:

FT.SEARCH idx:movie "@genre:{Drama} @release\_year:[1990 (2000]" LIMIT 0 0

Copied!

content\_copy

**Output:**

1) (integer) 24

**Congratulations!**

You've learned a bit more about Redis and the powerful indexing capabilities of RediSearch. To review, you should now know how to install and run a Redis instance with RediSearch using Docker. You should also know how to import a few example hashes, index them with RediSearch, and query them. Finally, you should understand some of RediSearch's aggregation features.

Next steps / Learn more

* [Redis Labs](https://console.cloud.google.com/marketplace/browse?q=Redis+Labs&utm_source=Redis&utm_medium=qwiklabs&utm_campaign=Qwiklabs+to+Marketplace) on the Google Cloud Marketplace!
* The [Redis University YouTube channel](https://www.youtube.com/channel/UCybK6TMZFQeSN74jzTiDWfg?view_as=subscriber)
* Free online courses with [Redis University](https://university.redislabs.com/)
* [RediSearch Site, Tutorial, Use Cases and Projects](https://redislabs.com/rediscover/projects/)

Google Cloud training and certification

...helps you make the most of Google Cloud technologies. [Our classes](https://cloud.google.com/training/courses) include technical skills and best practices to help you get up to speed quickly and continue your learning journey. We offer fundamental to advanced level training, with on-demand, live, and virtual options to suit your busy schedule. [Certifications](https://cloud.google.com/certification/) help you validate and prove your skill and expertise in Google Cloud technologies.