

Bases de datos II

Bases de datos NoSQL - Redis

Laurence Thiago, Ignacio Traberg, Ulises Geymonat

19 de junio del 2024

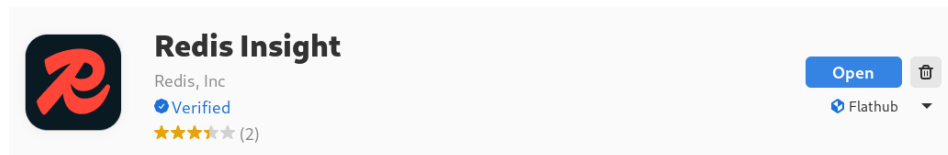
Parte 1: Introducción

Instalación de Redis utilizada

Hemos utilizado las diferentes formas de instalación y utilización del DBMS Redis, siendo la utilización online del servidor Redis en la nube mediante [try-redis](#), y la instalación manual de Redis Stack junto a su GUI Redis Insight, donde esta se realizó sobre un sistema GNU/Linux Fedora mediante el siguiente comando:

- `dnf install redis`

Y la instalación de la GUI fue mediante la tienda de software nativa del sistema operativo, como la simple instalación de una aplicación.



1)

Redis (*Remote Dictionary Server*) es un DBMS NoSQL en memoria, donde los datos se almacenan y manipulan directamente sobre la memoria basándose en el almacenamiento de tablas de hash, aunque opcionalmente pueden persistirse en disco.

2)

Redis almacena los datos de las BDs completamente en memoria, aunque también tiene la capacidad de persistirlos en disco, lo que representa un *trade off* diferente, donde se logran lecturas y escrituras a una alta velocidad, con la limitación de que los conjuntos de datos no pueden ser más grandes que la memoria RAM disponible. Además, debido a la representación interna inherente de la memoria, esta permite la manipulación de estructuras de datos más complejas de forma más simple en comparación con el disco.

3)

Redis al ser un servidor de estructuras de datos en memoria, **provee una colección de tipos de datos nativos** que ayudan a resolver una amplia variedad de problemas, desde el almacenamiento en caché, hasta las colas y procesamiento de eventos. A continuación se detallan los tipos de datos que posee:

- **Strings** → es el tipo de datos más básico, representado como una secuencia de bytes, el cual puede almacenar texto, objetos serializados o arreglos binarios.
 - Por ejemplo, se asocia el string “*bike:1*” al valor “*Deimos*”.

```
> SET bike:1 Deimos
OK
> GET bike:1
"Deimos"
```

- **List** → corresponden a listas enlazadas de strings ordenados por orden de inserción. Estas son utilizadas, generalmente, para implementar pilas, cola, o gestionar sistemas de colas de trabajos en segundo plano.
 - Por ejemplo, a la lista “*bikes:repairs*” se le agrega el elemento “*bike:1*” y “*bike:2*”, donde se la manipula como una cola (FIFO).

```
> LPUSH bikes:repairs bike:1
(integer) 1
> LPUSH bikes:repairs bike:2
(integer) 2
> RPOP bikes:repairs
"bike:1"
```

- **Set** → son conjuntos de colecciones desordenados de strings únicos (denominados miembros del conjunto), los cuales se comportan como los conjuntos en los lenguajes de programación tradicionales, como por ejemplo *HashSet* en Java, o *Set* en Python. Con este tipo de datos, en Redis es posible agregar, eliminar, y probar su existencia en tiempo $O(1)$ independientemente del número de elementos. Estos son utilizados para tratar eficientemente: rastreo de ítems únicos (como almacenar las IP únicas que acceden a un post de un blog), representar relaciones (como los conjuntos de usuarios con un rol determinado), o realizar operaciones de conjuntos comunes (como unión, intersección, diferencia).

- Por ejemplo, se agrega el elemento “*bike:1*” al conjunto de bicicletas de carrera en Francia “*bikes:racing:france*”, donde la segunda operación se ignora debido a que elemento ya pertenece al conjunto.

```
> SADD bikes:racing:france bike:1
(integer) 1
> SADD bikes:racing:france bike:1
(integer) 0
```

- **Hashes** → corresponden a tipos de registros modelados como colecciones de pares *clave:valor*. Estos se parecen a los *diccionarios* de Python, o *HashMaps* en Java. Son utilizados para almacenar objetos básicos, grupos de contadores, entre otras cosas.
 - Por ejemplo, se asocia la clave “*bike:1*” con la serialización de un objeto de varios campos “*model, brand, type, price*” y valores de los mismos “*Deimos, Ergonom, Enduro bikes, 4972*” respectivamente.

```
> HSET bike:1 model Deimos brand Ergonom type 'Enduro bikes' price 4972
(integer) 4
> HGET bike:1 model
"Deimos"
> HGET bike:1 price
"4972"
> HGETALL bike:1
1) "model"
2) "Deimos"
3) "brand"
4) "Ergonom"
5) "type"
6) "Enduro bikes"
7) "price"
8) "4972"
```

- **Sorted set** → son conjuntos de colecciones de strings únicos ordenados según el puntaje obtenido para cada cadena, donde si más de una cadena obtiene el mismo puntaje, estas se ordenan bajo el criterio lexicográfico. Pueden ser utilizados para almacenar listas ordenadas de puntuaciones en un juego.
 - Por ejemplo, se almacena en el conjunto ordenado “*racer_scores*” las puntuaciones y corredores de una carrera.

```
> ZADD racer_scores 12 "Castilla"
(integer) 1
> ZADD racer_scores 8 "Sam-Bodden" 10 "Royce" 6 "Ford" 14 "Prickett"
(integer) 4
```

```
> ZRANGE racer_scores 0 -1
1) "Ford"
2) "Sam-Bodden"
3) "Norem"
4) "Royce"
5) "Castilla"
6) "Prickett"
```

- **Streams** → es una estructura de datos que actúa como un flujo de datos o log donde solo se agregan datos por delante. Esta estructura permite registrar eventos en el orden en el que ocurren y luego distribuirlos para su procesamiento en tiempo real. Pueden ser utilizados para registrar eventos como acciones del usuario, monitoreo de sensores, o notificaciones.
 - Por ejemplo, cuando los corredores pasan checkpoint durante la carrera, se registra una nueva entrada con el corredor, velocidad, posición e identificación de ubicación.

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

- **Geospatial indexes** → son índices geoespaciales que almacenan coordenadas, los cuales son utilizados para encontrar ubicaciones dentro de un área geográfica específica.
 - Por ejemplo, desea buscar todas las ubicaciones de alquiler de bicicletas cercanas a su ubicación actual. Se busca en un radio de 5 Km en la ubicación especificada, y devuelve todas las estaciones cercanas junto a la distancia de cada una.

```
> GEOADD bikes:rentable -122.27652 37.805186 station:1
(integer) 1
```

```
> GEOSEARCH bikes:rentable FROMLONLAT -122.2612767 37.7936847 BYRADIUS 5 km
WITHDIST
1) 1) "station:1"
   2) "1.8523"
2) 1) "station:2"
   2) "1.4979"
3) 1) "station:3"
   2) "2.2441"
```

- **Bitmaps** → corresponden a mapas de bits que permiten realizar operaciones bit a bit sobre strings. En realidad, no es un tipo de datos real, sino un conjunto de operaciones sobre los strings que los manipula con un arreglo de bits. Pueden ser utilizados para representaciones de conjuntos eficientes donde sus elementos corresponden a números entre 0 y N, o definir permisos sobre objetos, donde cada bit representa un permiso particular (similar a los permisos sobre los archivos).
 - Por ejemplo, tiene 100 ciclistas corriendo donde cada uno posee un sensor y estos se encuentran numerados de 0 a 99, y desea registrar si un sensor realiza un ping a un servidor en una hora determinada. Por lo que puede utilizar como clave una fecha y hora específica, y los números de los sensores como valores.

```
> SETBIT pings:2024-01-01-00:00 123 1
(integer) 0
```

- **Bitfields** → estos campos de bits permiten codificar de manera eficiente múltiples contadores en un único valor en forma de string. Esta estructura proporciona operaciones atómicas de obtención, configuración e incremento, las cuales admiten distintas políticas de overflow. Donde los valores enteros de los contadores pueden tomar cualquier longitud arbitraria, desde 1 bit sin signo, hasta 63 bits con signo.
 - Por ejemplo, puede mantener dos métricas en el registro de bicicletas, donde una métrica indique el precio actual y otra la cantidad de dueños a lo largo del tiempo. Esto es posible representarlo mediante dos contadores de 32 bits. Donde se registra la bicicleta nueva “bike:1” con un precio de \$1000 sin dueño, luego esta se vende actualizando su precio a usada y la cantidad de dueños, y así sucesivamente.

```

> BITFIELD bike:1:stats SET u32 #0 1000
1) (integer) 0
> BITFIELD bike:1:stats INCRBY u32 #0 -50 INCRBY u32 #1 1
1) (integer) 950
2) (integer) 1
> BITFIELD bike:1:stats INCRBY u32 #0 500 INCRBY u32 #1 1
1) (integer) 1450
2) (integer) 2
> BITFIELD bike:1:stats GET u32 #0 GET u32 #1
1) (integer) 1450
2) (integer) 2


```

- **HyperLogLog** → estas estructuras proporcionan estimaciones probabilísticas de la cardinalidad (número de elementos) en conjuntos de datos grandes, ofreciendo una precisión casi perfecta junto a una utilización eficiente del espacio. Puede ser utilizado para contar la cantidad de consultas realizadas por los usuarios en un formulario de búsqueda todos los días, o la cantidad de visitantes únicos en un sitio web.

4)

Las características más importantes de redis son:

- Es una base de datos que se mantiene en memoria, por lo que mantiene restricciones respecto de la cantidad de memoria direccionable y utilizable, dependiendo del sistema es de 32 o 64 bits, y por supuesto la cantidad de memoria disponible del sistema. Donde varía el tamaño consumido y almacenable por sus elementos, como las instancias vacías, claves pequeñas y claves grandes.
- Opcionalmente puede persistir sus datos en disco, mediante la utilización de varios y/o combinados mecanismos de persistencia.
- Redis admite el uso de transacciones pero utilizando bloqueo optimista en lugar de pesimista. Además de no incluir soporte de rollbacks ante fallos en las transacciones.
- Posee una variedad de casos de uso, los cuales son sumamente distintos a los típicos en las bases de datos relacionales. Por ejemplo, uso de BD vectorial, o BD en caché como su principal uso.
- Permite la ejecución de scripts desarrollados en Lua dentro del servidor, los cuales además de otorgar la posibilidad de añadir lógica de aplicación en ellos, estos al



ejecutarse en el servidor permiten una ejecución altamente eficiente y garantizando que sea de forma atómica.

5)

La utilización de DBMS como Redis, brindan muchas herramientas que permiten una muy rápida y flexible manipulación de los datos, orientados a la implementación de esquemas estructurados y/o desestructurados que se adapten a la mayor parte de los distintos casos de uso de las aplicaciones sin afectar las finalidades ni tiempos de desarrollo. Sin embargo, esto conlleva analizar el *trade off* resultante, donde a costa de reducir la capacidad del sistema en términos de persistencia, consistencia y seguridad de los datos, a cambio de una alta flexibilidad de los modelos de datos acordes a los casos de usos específicos de la aplicación, y extremadamente rápida recuperación de los mismo respecto de los RDBMS.

Por los motivos anteriormente descritos, es muy común utilizar como patrón de diseño la utilización de ambos tipos de DBMS, siendo los relacionales o SQL y los NoSQL. Lo que implica tomar pequeños conjuntos de datos con gran cantidad de escrituras y utilizarlos sobre Redis (junto a conjuntos de datos que requieran de las estructuras de datos provistas por Redis para poder modelar su problema específico de forma eficiente), y utilizar grandes cantidades de datos estructurados en una base de datos SQL persistente y consistente en disco. De forma similar, también puede utilizarse sistemas como Redis como un almacenamiento más avanzado en caché, donde allí se guarda un pequeño conjunto de los datos estructurados almacenados en la base de datos SQL en disco, donde ante actualizaciones de los datos en Redis, estos se reflejan en disco juntos, y en caso de fallas en la caché, no se ve afectada la consistencia en disco.

6)

Las **transacciones en Redis** permiten la ejecución de un grupo de comandos en un solo paso, es decir, como la ejecución de una única operación, siendo los comandos MULTI, EXEC, DISCARD y WATCH. Por lo que todos los comandos de una transacción se serializan y ejecutan secuencialmente, donde el servidor de Redis nunca atenderá solicitudes realizadas por otro cliente durante una transacción, garantizando que los comandos se ejecuten como una única operación aislada.

El comando EXEC es el cual desencadena la ejecución de los demás comandos de la transacción, donde en caso de que el cliente pierda la conexión con el servidor antes de la llamada al comando, no se realizará ninguna operación. Además en caso de utilizar archivos

append-only, Redis asegura realizar una única operación de escritura en disco para escribir la transacción. Sin embargo, si el servidor falla durante la transacción, es posible que solo se registre una cantidad parcial de operaciones, lo que luego podrá ser detectado mediante *redis-check-aof* en el reinicio del servidor, posibilitando arreglar el archivo deshaciendo las operaciones parciales y volviendo al estado original, para luego poder re-comenzar la transacción.

Cabe destacar que (a partir de la versión 2.2) **Redis ofrece una garantía adicional** a los dos mecanismos de ejecución anteriores, en forma de **bloqueos optimistas**, similares a operaciones de verificación y configuración (CAS). Donde a través del comando WATCH se permite monitorear los cambios efectuados sobre las claves, donde en caso de que alguna de ellas sea modificada antes de la ejecución de EXEC, la transacción se cancela notificando su fallo. Esto permite evitar condiciones de carrera cuando más de un cliente modifica la misma clave al mismo tiempo donde ambos intentan realizar la transacción en el servidor, evitando inconsistencia en los datos. Y donde en caso de que la transacción falle por este motivo, solo queda re-ejecutar la transacción y esperar a que la misma no vuelva a fallar por el mismo motivo.

Durante una transacción puede surgir dos tipos de errores:

- **Que un comando falle antes de invocar a EXEC:** por ejemplo, debido a errores sintácticos o condiciones críticas por falta de memoria, lo que provoca que Redis (a partir de la versión 2.6.5) detecte el error y se niegue a ejecutar la transacción descartándola.
- **Que un comando falle después de invocar a EXEC:** por ejemplo, debido a operaciones con *clave:valor* incorrectos, lo que provoca que Redis no pueda manejar estos errores especiales, por lo que no detendrá el procesamiento de los comandos en la cola y los ejecutará inclusive si alguno falla.
 - Por lo tanto **Redis no admite rollbacks** en las transacciones, ya que tendría un impacto significativo en la simplificación y rendimiento del sistema.

7)

Redis ofrece de forma nativa **dos formas de persistencia de datos** en disco:

- **RDB (Redis database):** este tipo de persistencia realiza *snapshots* en un momento dado de su conjunto de datos a intervalos específicos, guardando el archivo en disco.

- **AOF (Append only file):** este tipo de persistencia registra cada operación de escritura recibida por el servidor. Luego estas operaciones de escritura pueden reproducirse nuevamente al iniciar el servidor, reconstruyendo el conjunto de datos original.
- **Sin persistencia:** opcionalmente, es posible deshabilitar la persistencia por completo, la cual es utilizada para almacenar los datos en caché.
- **RDB + AOF:** también es posible utilizar ambos mecanismos en la misma instancia del servidor. Lo cual es semejante a si se desea un grado de seguridad en la persistencia de datos al ofrecido por el RDBMS PostgreSQL.

8)

Los **principales usos de Redis** pueden ser la manipulación de base de datos, motor de streaming de datos, broker de mensajería, o motor de búsqueda. Es decir:

- **Redis como DB de estructuras de datos en memoria** en forma de hashes *clave:valor*, donde se define un string como clave, el cual se asocia o mapea a datos los cuales podrían ser otro string o datos más complejos como estructuras JSON.

```
res1 = r.hset(
    "bike:1",
    mapping={
        "model": "Deimos",
        "brand": "Ergonom",
        "type": "Enduro bikes",
        "price": 4972,
    },
)
```

Ejemplo en lenguaje Python, donde se define la clave "bike:1" mapeada a los datos con forma de diccionario.

- **Redis como BD de documentos JSON** similar a MongoDB, donde se permite definir schemas JSON indicando los campos del documento junto a los tipos de datos a almacenar cada uno, y luego se define un índice como prefijo del documento, el cual se puede relacionar con las *Colecciones* de Mongo, para luego referenciar a los documentos con un prefijo particular y realizar operaciones sobre ellos.

```

schema = (
    TextField("$.brand", as_name="brand"),
    TextField("$.model", as_name="model"),
    TextField("$.description", as_name="description"),
    NumericField("$.price", as_name="price"),
    TagField("$.condition", as_name="condition"),
)

index = r.ft("idx:bicycle")
index.create_index(
    schema,
    definition=IndexDefinition(prefix=["bicycle:"], index_type=IndexType.JSON),
)

```

Ejemplo en lenguaje Python, donde se define un esquema de documento JSON a almacenar como el que se verá a continuación, junto a un prefijo “bicycle:” utilizado como índice sobre este tipo de documentos.

```

{
  "brand": "brand name",
  "condition": "new | used | refurbished",
  "description": "description",
  "model": "model",
  "price": 0
}

```

Documento resultante a almacenar

- **Redis como BD vectorial** donde, en caso de utilizar datos desestructurados, como envío de textos, imágenes, videos, entre otros, se convierten y manipulan los datos en forma de arreglos unidimensionales, lo que permite almacenar los datos en arreglos y los metadatos de los mismos dentro de hashes o documentos JSON, para luego recuperar los arreglos y realizar búsquedas sobre ellos.

```

{
  "model": "Jigger",
  "brand": "Velorim",
  "price": 270,
  "type": "Kids bikes",
  "specs": {
    "material": "aluminium",
    "weight": "10"
  },
  "description": "Small and powerful, the
}

```

Por ejemplo, un esquema de documento JSON el cual una vez recuperado en forma de arreglo de datos, se debe formatear al tipo de dato específico almacenado, en este caso un formato JSON

```
json.dumps(bikes[0], indent=2)
```

Aquí se accede al primer elemento del arreglo de datos, el cual se debe procesar para manipular correctamente sus datos contenidos

```
pipeline = client.pipeline()
for i, bike in enumerate(bikes, start=1):
    redis_key = f"bikes:{i:03}"
    pipeline.json().set(redis_key, "$", bike)
res = pipeline.execute()
```

Aquí el esquema anterior se formatea en un tipo de datos “clave:valor” acorde al esquema definido anteriormente

- **Redis como motor de búsqueda** mediante la creación de índices, estructuras de búsqueda y consulta, permite la definición de un potente motor de búsqueda y consulta en las aplicaciones, lo que conlleva a la realización de consultas eficientes sobre datos estructurados (como los documentos), así como búsquedas basadas en texto y vectoriales sobre datos desestructurados.

Parte 2: Manejo simple de valores de string

1. Agregue una clave package con el valor “Bariloche 3 days”

```
> SET package "Bariloche 3 days"
```

OK

2. Agregue una clave user con el valor “Turismo BD2”. Obtenga el valor de la clave user

```
> SET user "Turismo BD2"
```

OK

```
> GET user
```

"Turismo BD2"

3. Obtenga todos los valores de claves almacenadas

```
> KEYS *
```

KEYS *

- 1) "user"
- 2) "package"

```
> MGET user package server
```

MGET user package server

- 1) "Turismo BD2"
- 2) "Bariloche 3 days"
- 3) "null"

4. Agregue una clave user con el valor “Cronos Turismo” ¿Cuál es el valor actual de la clave user?

```
> SET user "Cronos Turismo"  
  
> GET user
```

```
"Cronos Turismo"
```

5. Concatene “ S.A.” a la clave user. ¿Cuál es el valor actual de la clave user?

```
> APPEND user " S.A."
```

```
19
```

```
> GET user
```

```
"Cronos Turismo S.A."
```

6. Elimine la clave user

```
> DEL user
```

```
(integer) 1
```

7. ¿Qué valor retorna si queremos obtener la clave user?

```
> GET user
```

```
(nil)
```

Parte 3: Manejo simple de valores numéricos

1. Verificar si existe la clave visits

```
> EXISTS visits
```

```
(integer) 0
```

2. Agregue una clave visits con el valor 0

```
> SET visits 0
```

3. Incremente en 1 visits ¿Cuál es el valor actual de la clave visits?

```
> INCR visits
```

```
(integer) 1
```

```
> GET visits
```

```
"1"
```

4. Incremente en 5 visits. ¿Cuál es el valor actual de la clave visits?

```
> INCR visits 5
```

5. Decremento en 1 visits ¿Cuál es el valor actual de la clave visits?

6. Incremente en 2 visits. ¿Cuál es el valor actual de la clave visits?

7. Agregue una clave “value package” con el valor 539789.32

8. Incremente en 20000 la clave “value package”. ¿Cuál es el valor actual de “value package”?

9. ¿Cuál es el tipo de datos de “value package”, visits y user?

15



```
> TYPE visits
```

```
"string"
```

```
> TYPE user
```

```
"none"
```

Parte 4: Manejo de claves

1. Obtenga todas las claves que empiecen con v

```
> KEYS v*
```

```
1) "value package"  
2) "visits"
```

2. Obtenga todas las claves que contengan la "t"

```
> KEYS *t*
```

```
1) "visits"
```

3. Obtenga todas las claves que terminen con "age"

```
> KEYS *age
```

```
1) "value package"  
2) "package"
```

4. Renombre la clave "package" por "bariloche package"

```
> RENAME package "bariloche package"
```

```
OK
```

5. Si quisiera evitar renombrar una clave por otra existente, ¿qué comando utilizaría?

```
> RENAMENX clave_anterior clave_nueva
```

(Solo renombrara si clave_nueva no existe)

6. Elimine todas las claves

```
> FLUSHALL
```

Nótese que en TryRedis, esta instrucción no es válida, pero lo que hace es borrar todas las claves. El comando FLUSHDB, por su lado, sirve para borrar todas las claves de todas las bases de datos, pero tampoco funciona en TryRedis.

Parte 5: Expiración de claves

1. Agregue una clave agency con el valor “Cronos Tours”

```
> SET agency "Cronos Tours"
```

OK

2. ¿Cuál es el tiempo de vida de la clave agency?

```
> TTL agency
```

```
(integer) -1
```

3. Agregue una expiración de 30 segundos a la clave agency

```
> EXPIRE agency 30
```

```
(integer) 1
```

4. ¿Cuál es el tiempo de vida de la clave agency?

```
> TTL agency
```

```
(integer) 27
```

5. Pasados los 30 segundos, ¿Cuál es el tiempo de vida de la clave agency? ¿Qué retorna si pido el valor de agency? **-2 (es el valor esperado para una clave expirada)**

```
> TTL agency
```

```
(integer) -2
```

6. Agreguemos una clave agency con el valor “Cronos Tours” que expire en 20 segundos

```
> SETEX agency 20 "Cronos Tours"
```

```
OK
```

```
(integer) 17
```

Parte 6: Listas

1. Inserte una lista llamada `pets` con el valor `dog`.

```
> LPUSH pets dog
```

```
(integer) 1
```

2. ¿Qué sucede si ejecuto el comando `get pets`? ¿Cómo obtengo los valores de la lista? **Me da error porque `pets` es una lista y “`get`” es para obtener valor de tipo `String` a partir de una clave. Para trabajar con listas usamos `LRANGE`, desde el primer índice (0) al último (-1).**

```
> get pets
```

```
(error) WRONGTYPE Operation against a key holding the wrong kind of value
```

```
> LRANGE pets 0 -1
```

```
1) "dog"
```

3. Agregue a la lista `pets` el valor `cat` a la izquierda.

```
> LPUSH pets cat
```

```
(integer) 2
```

4. Agregue a la lista `pets` el valor `fish` a la derecha.

```
> RPUSH pets fish
```

```
(integer) 3
```

5. ¿Qué tipo de datos es el valor de `pets`?

```
> TYPE pets
```

```
"list"
```

6. Elimine el valor a la izquierda de la lista.

```
> LPOP pets
```

```
"cat"
```

7. Elimine el valor a la derecha de la lista. ***Voy a suponer que era “a la derecha”, en contraste con el punto anterior.***

```
> RPOP pets
```

```
"fish"
```

8. Agregue una clave “vuelo:ar389” los valores: aep, mdz, brc, nqn y mdq.

```
> RPUSH vuelo:ar389 aep mdz brc nqn mdq
```

```
(integer) 5
```

9. Ordene los valores de lista “vuelo:ar389”. ¿Qué sucede si solicito todos los valores de la lista? ***Utilizo la opción “alpha” para indicar que se haga en base a un orden alfabético de los valores.***

```
> SORT vuelo:ar389 alpha
```

```
1) "aep"  
2) "brc"  
3) "mdq"  
4) "mdz"  
5) "nqn"
```

```
> LRANGE vuelo:ar389 0 -1
```

- 1) "aep"
- 2) "mdz"
- 3) "brc"
- 4) "nqn"
- 5) "mdq"

Al imprimir los resultados de la lista, podemos ver que las entradas mantienen su orden original. Esto se debe a que el comando SORT es no-destructivo, y solo modifica el resultado de salida, sin afectar la fuente.

10. Inserte el valor "fte" luego de "brc"

```
> LINSERT vuelo:ar389 AFTER brc fte
```

6

11. Inserte el valor "ush" antes de "fte"

```
> LINSERT vuelo:ar389 BEFORE fte ush
```

7

12. Modifique el último elemento por "sla" *Se utiliza -1 para referirse a la última posición de la lista.*

```
> LSET vuelo:ar389 -1 sla
```

OK

13. Obtenga la cantidad de elementos de "vuelo:ar389"

```
> LLEN vuelo:ar389
```

(integer) 7

14. Obtenga el 3 valor de "vuelo:ar389" *Ya que los índices de listas arrancan del 0, indicamos el 2.*

```
> LINDEX vuelo:ar389 2
```

"brc"

- 1) "aep"
- 2) "mdz"
- 3) "brc"
- 4) "ush"
- 5) "fte"
- 6) "nqn"
- 7) "sla"

15. Elimine el valor "aep" de "vuelo:ar389"

El 1 indica que se elimine esa cantidad (también podría haber sido 0 para eliminar todos, pero de antemano sabía que solo había 1). De haber sido un número negativo, habría eliminado esa cantidad pero desde atrás para adelante.

```
> LREM vuelo:ar389 1 aep
```

(integer) 1

16. Quédele con los valores de las posiciones 3 a 5 de vuelo:ar389

Lo mismo, 3-1 y 5-1

```
> LTRIM vuelo:ar389 2 4
```

OK

17. Agregue en "vuelo:ar389" el valor "fte". ¿Cuántas veces aparece?

```
> RPUSH vuelo:ar389 fte
```

(integer) 4

```
> LRANGE vuelo:ar389 0 -1
```

- 1) "ush"
- 2) "fte"
- 3) "nqn"
- 4) "fte"

El último valor "fte" fue agregado por la derecha y es el número '4' (índice 3). FTE está dos veces. Esto se debe a que, a diferencia de un SET, las LISTAS pueden guardar valores repetidos, ya que se valen de su posición en la estructura para ser unívocos.

—

Parte 7: Conjuntos

1. Agregue un conjunto llamado airports los valores: eze aep nqn mdz mdq ush fte sla

aep nqn brc cpc juj aep tuc eqs

```
> SADD airports eze aep nqn mdz mdq ush fte sla aep nqn brc cpc juj aep tuc eqs
```

```
(integer) 13
```

2. ¿Cuántos valores tiene el conjunto?

```
> SCARD airports
```

```
13
```

3. Liste los valores del conjunto airports.

```
> SMEMBERS airports
```

```
1) "eqs"
2) "mdq"
3) "cpc"
4) "ush"
5) "mdz"
6) "nqn"
7) "sla"
8) "fte"
9) "eze"
10) "tuc"
11) "juj"
12) "brc"
13) "aep"
```

4. Quite el valor cpc del conjunto airports.

```
> SREM airports cpc
```

```
1
```

5. Quite un valor aleatorio del conjunto airports.

```
> SPOP airports
```

```
"aep"
```

6. ¿Qué cantidad de valores tiene ahora airports?

```
> SCARD airports
```

```
11
```

7. Compruebe si cpc es miembro del conjunto airports.

```
> SISMEMBER airports cpc
```

```
(integer) 0
```

8. Mueva los valores sla y juj a un conjunto denominado noa_airports.

```
> SMOVE airports noa_airports sla
```

```
(integer) 1
```

```
> SMOVE airports noa_airports juj
```

```
(integer) 1
```

9. Retorne la unión de los conjuntos airports y noa_airports. ¿Modifica los conjuntos base?

No, los conjuntos permanecen igual.

```
> SUNION airports noa_airports
```

```
1) "mdz"  
2) "nqn"  
3) "eze"  
4) "fte"  
5) "mdq"  
6) "eqs"  
7) "ush"  
8) "sla"  
9) "tuc"  
10) "juj"  
11) "brc"
```

```
> SMEMBERS airports
```

```
1) "eqs"  
2) "mdq"  
3) "ush"  
4) "mdz"  
5) "nqn"  
6) "fte"  
7) "eze"  
8) "tuc"  
9) "brc"
```

```
> SMEMBERS noa_airports
```

```
1) "sla"  
2) "juj"
```

10. Realice la unión de los conjuntos airports y noa_airports en un conjunto llamado total_airports.

```
> SUNIONSTORE total_airports airports noa_airports
```

11

```
> SMEMBERS total_airports
```

```
1) "mdz"  
2) "nqn"  
3) "eze"  
4) "fte"  
5) "mdq"  
6) "eqs"  
7) "ush"  
8) "sla"  
9) "tuc"  
10) "juj"  
11) "brc"
```

11. Realice la intersección entre los conjuntos total_airports y noa_airports.

```
> SINTER total_airports noa_airports
```

```
1) "sla"  
2) "juj"
```

12. Realice la diferencia entre los conjuntos total_airports y noa_airports.

```
> SDIFF total_airports noa_airports
```

```
1) "tuc"  
2) "mdq"  
3) "eqs"  
4) "ush"  
5) "nqn"  
6) "mdz"  
7) "eze"  
8) "fte"  
9) "brc"
```

Parte 8: Conjuntos ordenados

1)

```
ZADD passengers 2.5 "federico" 4 "alejandra" 3 "julian" 1 "ivan" 2  
"andrea" 2 "luciana" 2.4 "natalia"
```

```
ZADD passengers 2.5 "federico" 4 "alejandra" 3 "julian" 1 "ivan" 2 "andrea" 2 "luciana" 2.4 "natalia"
```

```
(integer) 7
```

2)

```
ZRANGE passengers 0 -1
```

```
ZRANGE passengers 0 -1
```

```
1) "ivan"  
2) "andrea"  
3) "luciana"  
4) "natalia"  
5) "federico"  
6) "julian"  
7) "alejandra"
```

3)

```
ZADD passengers 2.7 "luciana"
```

4)

```
ZADD passengers 5.1 "silvia"
```

5)

```
ZINCRBY passengers 2 "alejandra"
```

6)

```
ZRANGE passengers 0 -1 WITHSCORES
```

```
1) "ivan"
2) "1"
3) "andrea"
4) "2"
5) "natalia"
6) "2.4"
7) "federico"
8) "2.5"
9) "luciana"
10) "2.7"
11) "julian"
12) "3"
13) "silvia"
14) "5.1"
15) "alejandra"
16) "6"
```

7)

```
ZREVRANGE passengers 0 -1 WITHSCORES
```

```
1) "alejandra"
2) "6"
3) "silvia"
4) "5.1"
5) "julian"
6) "3"
7) "luciana"
8) "2.7"
9) "federico"
10) "2.5"
11) "natalia"
12) "2.4"
13) "andrea"
14) "2"
15) "ivan"
16) "1"
```

8)

ZCARD passengers

```
ZCARD passengers
```

```
(integer) 8
```

9)

ZCOUNT passengers 2 3

```
ZCOUNT passengers 2 3
```

```
(integer) 5
```

10)

ZRANK passengers "julian"

```
ZRANK passengers "julian"
```

```
(integer) 5
```

Nota: las posiciones en el ranking empiezan a partir de la posición 0.

11)

ZSCORE passengers "andrea"

```
ZSCORE passengers "andrea"
```

```
"2"
```

12)

```
ZRANGE passengers 0 0 WITHSCORES
```

```
ZRANGE passengers 0 0 WITHSCORES
```

```
1) "ivan"  
2) "1"
```

13)

```
ZREVRANGE passengers 0 0 WITHSCORES
```

```
ZREVRANGE passengers 0 0 WITHSCORES
```

```
1) "alejandra"  
2) "6"
```

14)

```
ZREM passengers "silvia"
```

```
ZRANGE passengers 0 -1 WITHSCORES
```

```
1) "ivan"  
2) "1"  
3) "andrea"  
4) "2"  
5) "natalia"  
6) "2.4"  
7) "federico"  
8) "2.5"  
9) "luciana"  
10) "2.7"  
11) "julian"  
12) "3"  
13) "alejandra"  
14) "6"
```

Parte 9: Hashes

1)

```
HSET USER:CRONOS "razon social" "cronos s.a" domicilio "457 236 La Plata"
telefono 2215556677
```

2)

```
HSET USER:CRONOS email "info@cronos.com.ar"
```

3)

```
HGETALL USER:CRONOS
```

HGETALL USER:CRONOS

```
1) "razon social"
2) "cronos s.a"
3) "domicilio"
4) "457 236 La Plata"
5) "telefono"
6) "2215556677"
7) "email"
8) "info@cronos.com.ar"
```

4)

```
HGET USER:CRONOS email
```

HGET USER:CRONOS email

```
"info@cronos.com.ar"
```


5)

`HDEL USER:CRONOS telefono`

6)

`HLEN USER:CRONOS`

`HLEN USER:CRONOS`

`(integer) 3`

7)

`HKEYS USER:CRONOS`

`HKEYS USER:CRONOS`

1) "razon social"
2) "domicilio"
3) "email"

8)

`HEXISTS USER:CRONOS cuil`

`HEXISTS USER:CRONOS cuil`

`(integer) 0`

9)

`HVALS USER:CRONOS`

■

```
HVALS USER:CRONOS
```

```
1) "cronos s.a"  
2) "457 236 La Plata"  
3) "info@cronos.com.ar"
```

10)

```
HSTRLEN USER:CRONOS email
```

```
HSTRLEN USER:CRONOS email
```

```
(integer) 18
```

Parte 10: Geospatial

1)

```
GEOADD cities -34.61315 -58.37723 "Buenos Aires" -31.4135 -64.18105  
"Córdoba" -32.94682 -60.63932 "Rosario" -32.89084 -68.82717 "Mendoza"  
-26.82414 -65.2226 "San Miguel de Tucumán" -34.92145 -57.95453 "La Plata"  
-38.00042 -57.5562 "Mar del Plata" -24.7859 -65.41166 "Salta" -31.64881  
-60.70868 "Santa Fe" -31.5375 -68.53639 "San Juan" -27.46056 -58.98389  
"Resistencia" -27.79511 -64.26149 "Santiago del Estero" -27.36708  
-55.89608 "Posadas" -24.19457 -65.29712 "San Salvador de Jujuy" -38.71959  
-62.27243 "Bahía Blanca" -31.73271 -60.528 "Paraná"
```

2)

```
ZRANGE cities 0 -1
```

ZRANGE cities 0 -1

```
1) "Mendoza"  
2) "San Juan"  
3) "Córdoba"  
4) "San Miguel de Tucumán"  
5) "Santiago del Estero"  
6) "Salta"  
7) "San Salvador de Jujuy"  
8) "Bahía Blanca"  
9) "Mar del Plata"  
10) "Buenos Aires"  
11) "La Plata"  
12) "Rosario"  
13) "Santa Fe"  
14) "Paraná"  
15) "Resistencia"  
16) "Posadas"
```

Es posible utilizar el comando ZRANGE de los conjuntos ordenados, ya que internamente crea uno para ordenar los elementos del índice.

3)

```
GEOPOS cities "Santa Fe"
```

GEOPOS cities "Santa Fe"

1) 1) "-31.64880841970443726"
2) "-60.70867938681186615"

4)

GEODIST cities "Buenos Aires" "Córdoba" km

GEODIST cities "Buenos Aires" "Córdoba" km

"667.5963"

5)

GEOSEARCH cities FROMLONLAT -27.37 -55.9 BYRADIUS 100 km ASC

GEOSEARCH cities FROMLONLAT -27.37 -55.9 BYRADIUS 100 km ASC

1) "Posadas"

6)

GEOSEARCH cities FROMMEMBER "Córdoba" BYRADIUS 700 km ASC WITHDIST

GEOSEARCH cities FROMMEMBER "Córdoba" BYRADIUS 700 km ASC WITHDIST

- 1) 1) "C\x3\xb3rdoba"
2) "0.0000"
- 2) 1) "Santiago del Estero"
2) "175.2332"
- 3) 1) "San Miguel de Tucum\x3\xa1n"
2) "246.9070"
- 4) 1) "Salta"
2) "342.2432"
- 5) 1) "San Salvador de Jujuy"
2) "364.1908"
- 6) 1) "Santa Fe"
2) "386.4079"
- 7) 1) "Rosario"
2) "401.7442"
- 8) 1) "Paran\x3\xa1"
2) "406.6477"
- 9) 1) "Bah\x3\xada Blanca"
2) "422.8023"
- 10) 1) "San Juan"
2) "484.4597"
- 11) 1) "Mendoza"
2) "520.8684"
- 12) 1) "Resistencia"
2) "614.4778"
- 13) 1) "Buenos Aires"
2) "667.5963"