Práctica 1: REDIS

Martínez Ostoa Néstor Iván
Bases de Datos No Estructuras
Ciencia de Datos, IIMAS, UNAM

Abril 2021

1. Introducción

Redis es una base de datos no estructurada enfocada, principalmente, en proveer incrementos considerables de velocidad a comparación de las otras bases no estructuradas. Esto lo logra gracias a lo siguiente:

- **Base de datos en memoria**: esto significa que todos los datos con los que trabaja están almacenados en memoria
- **Base de datos llave-valor**: todas las operaciones sobre Redis se llevan principalmente a cabo por medio de estructuras de datos formadas por una llave y valor. La ventaja de esto es que las operaciones computacionales para buscar llaves es constante en su mayoría ($O(n)$) o logarítmica ($O(\log n)$) para algunas operaciones en particular
- **Flexibilidad**: esto lo logra a través de la exposición de cinco estructuras de datos principales (pero no son las únicas): *Strings, Lists, Hash, Sets, Sorted Sets*. Cada una de estas estructuras funciona por medio de llave y valor y tienen un propósito en específico y son adaptables a muchas problemáticas. Esta es una gran ventaja frente a bases de datos relacionales pues éstas solo exponen una estructura de datos (tablas) y todos los problemas se tienen que resolver mediante ellas.

Ahora, con respecto a ser una base de datos en memoria, esto es parcialmente cierto pues Redis también se encarga de hacer copias al disco pero no es su prioridad principal. Redis fue diseñada originalmente por Salvatore Sanfilippo [1] para proveer velocidad y flexibilidad de escalamiento ante problemas que las bases de datos relacionales mediante SQL simplemente no serían capaces de

[1] `http://antirez.com/latest/0`

realizar. Sabiendo esto, podemos afirmar que Redis es una base de datos que desde el principio su enfoque fue la velocidad, sin embargo, también se puede ocupar como una base de datos principal para una empresa.

1.1. Estructuras de Datos en Redis

A continuación se presenta una breve descripción de las cinco estructuras de datos principales en Redis.

1.1.1. Strings

Son las estructuras de datos más básicas y son la implementación ideal de una **llave-valor**.

```
set url www.google.com
```

En donde la llave es **url** y el valor **www.google.com**
Para obtener el valor de la llave ocupamos la siguiente sintaxis:

```
get url
>>> www.google.com
```

1.1.2. List

Las listas son otra estructura de datos clásica que nos ofrecen gran flexibilidad. Internamente, al igual que Redis, están implementadas en C utilizando listas ligadas. Las listas nos permiten manipular un arreglo de valores para una determinada llave y podemos insertar y remover elementos de las listas al principio o al final de ellas.

```
lpush wishlist:nestor www.google.com
lpush wishlist:nestor www.apple.com
rpush wishlist:nestor www.samsung.com
```

Para leer la lista ocupamos **lrange**:

```
lrange wishlist:nestor 0 -1
>>> www.apple.com
>>> www.google.com
>>> www.samsung.com
```

1.1.3. Hash

Los hashes son similares a las cadenas en el sentido de que se manejan igual por llave y valor, pero la diferencia es que a cada valor se le puede asignar un atributo.

```
hset url:nestor url www.google.com categoy tecnologia
```

En donde **url:nestor** es la llave del hash, **url** y **category** son los campos para los atributos **www.google.com** y **tecnologia** respectivamente. Si quisiéramos obtener los campos de un hash haríamos lo siguiente:

```
hmgetall url:nestor
>>> url
    www.google.com
>>> category
    tecnologia
```

1.1.4. Set

Los conjuntos son otra estructura de datos muy versátil que se asemeja mucho a las listas. Su principal ventaja es que no pueden contener elementos repetidos, esto es muy útil porque a la hora de querer preguntar si un elemento se encuentra en el conjunto, la complejidad de la operación es logarítmica. Esto se debe a que los conjuntos, internamente, están implementados con estructuras de datos de tipo árbol.

```
sadd categories:url:nestor tecnologia libros cocina viajes
sadd categories:url:yazmin viajes politica libros museos
```

Para preguntar si un elemento se encuentra dentro de un conjunto empleamos **sismember**

```
sismember categories:url:yazmin viajes
```

También podemos aplicar operaciones clásicas de conjuntos, como la intersección entre dos o más conjuntos:

```
sinter categories:url:nestor categories:url:yazmin
```

1.1.5. Sorted Set

Los conjuntos ordenados son equivalentes a los conjuntos con la diferencia de que agregan a cada elemento del conjunto un valor numérico sobre el que se puede ordenar. Esta estructura de datos resulta ser muy útil para valores que requieren de atención en tiempo real, como por ejemplo, un *leaderboard* de un videojuego o las llegadas de aviones en un aeropuerto.

```
zadd favorites:url:nestor 20 www.overleaf.com 100 www.apple.com 55 www.firefox.com
```

Se pueden obtener información importante de los conjuntos ordenados mediante un rango de valores:

```
zcount favorites:url:nestor 40 100
```

Lo cual nos devolvería todas las URLs con un rango de valores de 40 y 100.

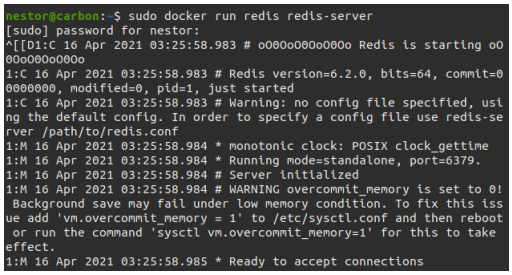


Figura 1: Ejemplo de un servidor de Redis listo para recibir peticiones

2. Desarrollo

2.1. Requerimientos

Para esta práctica se buscó generar un sistema de reducciones de URL con los siguientes requerimientos:

- El sistema debe soportar el manejo de usuarios
- La reducción de URLs debe ser por usuario y permitir que se tengan ligas públicas y ligas privadas (solo distinguirlas)
- Cada usuario debe tener una wishlist de los sitios que se desean visitar en el futuro al igual que la posibilidad de categorizar URLs
- Como administrador uno debe ser capaz de hacer consultas sobre las intersecciones entre las categorías de dos usuarios

2.2. Implementación

2.2.1. Esquema estructural

Para realizar la implementación generé el siguiente esquema estructural [2]:

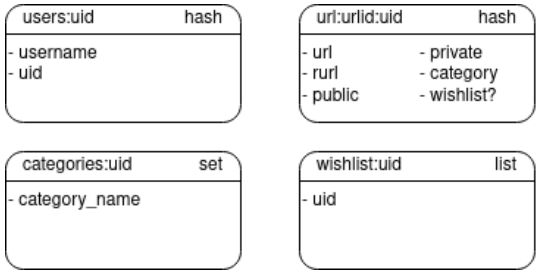


Figura 2: Esquema estructural

2.2.2. Diagrama de flujo de actividades

En la siguiente figura se muestra el esquema básico que sigue la implementación en Redis de las actividades que puede realizar un usuario.

1. Un usuario se da de alta y se registra en el hash `users:uid`
2. Un usuario puede agregar una URL al hash `url:urlid:uid` o,
3. Un usuario puede agregar una URL a la lista `wishlist:uid` encargada de almacenar la URL junto sus atributos en el hash del punto anterior y simplemente almacenar el id de la URL previamente almacenada en el hash

[2] El campo **urlid** de la estructura **wishlist:uid** se refiere al id de la URL que se acaba de agregar al wishlist y para consultar su contenido se tiene que hacer sobre la estructura **url:urlid:uid**

4. Finalmente, se almacena la categoría de la URL (proveniente de la adición de una nueva URL para reducción o de agregarla al wishlist) en el conjunto `categories:uid`

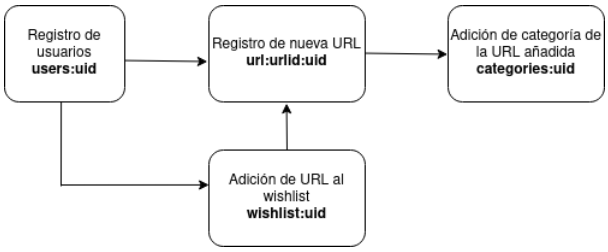


Figura 3: Diagrama de actividades

2.2.3. Llaves para secuencias

Las llaves siguientes (representadas por *strings*) se utilizaron para generar secuencias de usuarios y URLs, es decir, son las que asignan los identificadores únicos a los diferentes esquemas que se utilizaron en la implementación.

- **url:ids**: Empieza con el número 3000 y se encarga de asignar identificadores únicos a los registros de las URLs de los usuarios
- **user:ids**: Empieza con el número 1000 y se encarga de asignar identificadores únicos a los usuarios registrados en la base de datos
- **url:base:ids**: Empieza con el número 2500 y se encarga de asignar identificar únicamente a las URLs reducidas. En la sección 2.2.5 se habla con más detalle sobre la manera en que manejé la reducción de URLs
- **client:base:url**: Tiene el valor de **nestor.ly/** el cual es el dominio base para la reducción de las URLs

Implementación en REDIS:

```
127.0.0.1:6379> set url:ids 3000
OK
127.0.0.1:6379> set url:base:ids 2500
OK
127.0.0.1:6379> set user:ids 1000
OK
127.0.0.1:6379> set client:base:url nestor.ly/
OK
127.0.0.1:6379>
```

Figura 4: Defínción de llaves en REDIS

2.2.4. Generación de usuarios

La generación de usuarios está dada por un hash que tiene la siguiente llave:

user:uid

donde uid es el identificador único de cada usuario. Este hash (como se muestra en la figura 2), tiene como atributos un nombre de usuario y el identificador del usuario.

Para optimizar la interfaz de agregación de usuarios, escribí un script en Lua encargado de realizar la generación de usuarios en un nuevo hash. A grandes rasgos, el script se encarga de realizar los siguientes pasos:

1. Incrementa la llave **user:ids** para generar un nuevo id de usuario
2. Genera un hash compuesto por **users:uid** con el nombre de usuario recibido como parámetro y el user id.

Scipt en Lua:

```
local user_seq_key = KEYS[1]
local user_id = redis.call('incr', user_seq_key)
local user_key = 'users:'..user_id

local username = ARGV[1]
redis.call('hset', user_key, 'username', username, 'uid', user_id)

return redis.call('hgetall', user_key)
```

Implementación en REDIS:

1. Primero cargamos el script y guardamos la llave SHA en una llave

```
127.0.0.1:6379> script load "local user_seq_key = KEYS[1] local user_id = redis.call('incr', user_seq_key) local user_key = 'users:'..user_i
d local username = ARGV[1] redis.call('hset', user_key, 'username', username, 'uid', user_id) return redis.call('hgetall', user_key)"
"91a6ed3ae3ba273649fbf221564c3304fe3b00fe"
127.0.0.1:6379> set add_user:sha 91a6ed3ae3ba273649fbf221564c3304fe3b00fe
OK
127.0.0.1:6379>
```

Figura 5: Script para agregar usuarios a la base de datos

2. Agregamos usuarios

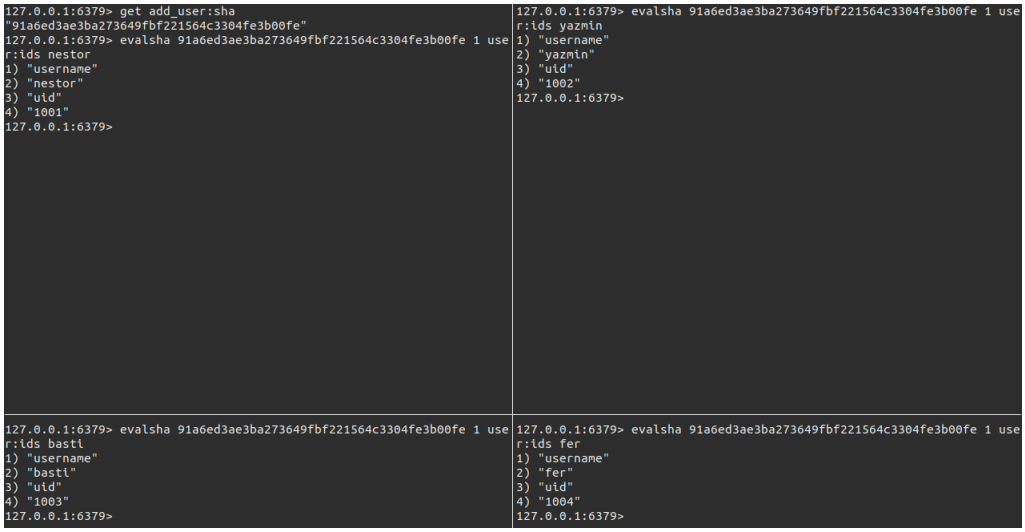


Figura 6: Agregación de cuatro nuevos usuarios a la base de datos

3. Verificamos

Figura 7: Validación de agregación de usuarios

2.2.5. Reducción de URLs

Para la reducción de URLs seguimos un esquema similar:

- El usuario indica la URL a reducir
- La base de datos obtiene la URL y la cambia por el nombre base de dominio (*nets.ly/*) y agrega un id único de identificación a la URL reducida
- Guarda los cambios en un hash específico por URL y por usuario.

Ejemplo:

- URL a reducir: *www.google.com/url-to-reduce.html*
- URL reducia: *nestor.ly/2189*

Para realizar la implementación, al igual que para agregar usuarios, escribí un script en LUA que automatiza la agregación de URLs y la reducción de las mismas. Este script, además de que reduce la URL, la guarda en el hash, también se encarga de insertar en el conjunto de categorías para el usuario dado la categoría de la URL a reducir. El script es el siguiente:

```
local url_seq_key = KEYS[1]
local rurl_base_domain_key = KEYS[2]
local rurl_seq_key = KEYS[3]

local user_id = ARGV[1]
local url = ARGV[2]
local public = ARGV[3]
local private = ARGV[4]
local category = ARGV[5]

local url_id = redis.call('incr', url_seq_key)
local url_user_key = 'url:'..url_id..':'..user_id
local rurl_base_domain = redis.call('get', rurl_base_domain_key)
local rurl_id = redis.call('incrby', rurl_seq_key, 17)
local rurl = rurl_base_domain..rurl_id

redis.call('hset', url_user_key, 'url', url, 'rurl', rurl, 'public', public,
    'private', private, 'category', category)

local category_user_key = 'categories:'..user_id
redis.call('sadd', category_user_key, category)

return redis.call('hgetall', url_user_key)
```

Implementación en REDIS:

1. Carga del script

```
127.0.0.1:6379> script load "local url_seq_key = KEYS[1] local rurl_base_domain_key = KEYS[2] local rurl_seq_key = KEYS[3] local user_id = A
RGV[1] local url = ARGV[2] local public = ARGV[3] local private = ARGV[4] local category = ARGV[5] local url_id = redis.call('incr', url_seq
_key) local url_user_key = 'url:'..url_id..':'..user_id local rurl_base_domain = redis.call('get', rurl_base_domain_key) local rurl_id = red
is.call('incrby', rurl_seq_key, 17) local rurl = rurl_base_domain..rurl_id redis.call('hset', url_user_key, 'url', url, 'rurl', rurl, 'publi
c', public, 'private', private, 'category', category) local category_user_key = 'categories:'..user_id redis.call('sadd', category_user_key,
 category) return redis.call('hgetall', url_user_key)"
"0d659e609c21eb7d6790f58ac77a971f525470a4"
127.0.0.1:6379> set add_url:sha 0d659e609c21eb7d6790f58ac77a971f525470a4
OK
127.0.0.1:6379>
```

Figura 8: Carga del script para agregar URLs

2. Adición de URLs de cada usuario

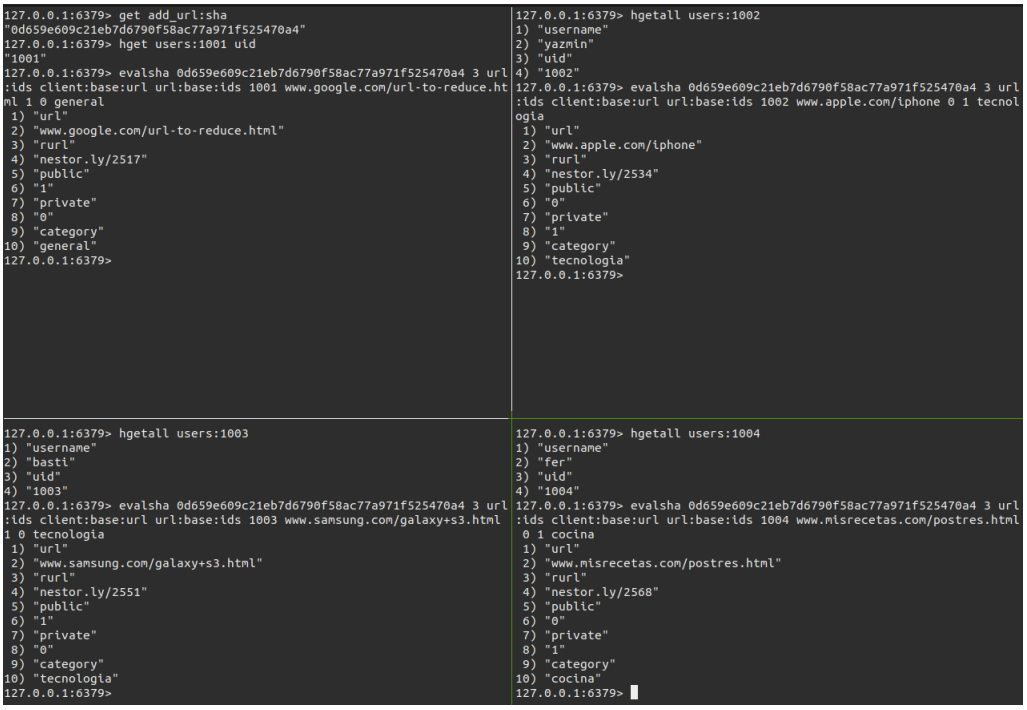
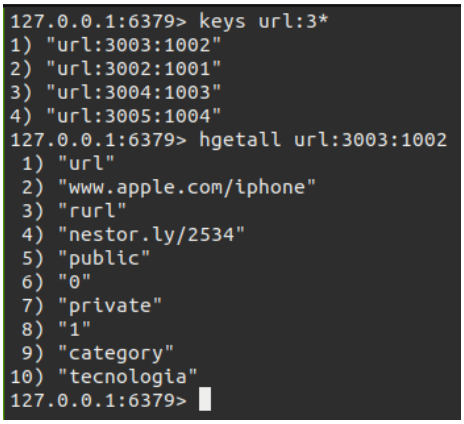
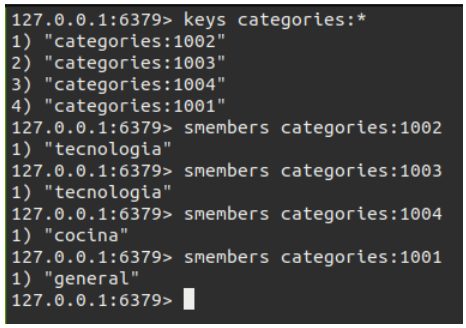


Figura 9: Adición de URLs por usuario y reducción de las mismas

3. Verificación de URLs y categorías por usuario



(a) Validación de inserción de URLs y reducción de las mismas



(b) Validación de categorías por usuario

Figura 10: Validaciones

2.2.6. Wishlist de URLs

Para la implementación de una wishlist ocupé una lista de Redis siguiendo el siguiente script de Lua:

```
local user_id = KEYS[1]
local url_seq_key = KEYS[2]
local rurl_seq_key = KEYS[3]
local base_client_key = KEYS[4]
local base_client = redis.call('get', base_client_key)

local url_to_add = ARGV[1]
local category = ARGV[2]
local public = ARGV[3]
local private = ARGV[4]

local wishlist_key = 'wishlist:'..user_id
local new_url_key = redis.call('incr', url_seq_key)
local rurl_new_key = redis.call('incrby', rurl_seq_key, 25)
local rurl = base_client..rurl_new_key
local key = 'url:'..new_url_key..':'..user_id

redis.call('hset', key, 'url', url_to_add, 'rurl', rurl, 'public', public, 'private', private, 'category', category, 'wishlist', '1')
redis.call('rpush', wishlist_key, new_url_key)

local categories_key = 'categories:'..user_id
redis.call('sadd', categories_key, category)

return redis.call('lrange', wishlist_key, 0, -1)
```

La idea de este script es recibir la url a agregar, la categoría e indicar si es privado o público. Posteriormente, se reduce la url y se agrega el id de la URL al wishlist del usuario en cuestión, pues todos los parámetros se agregan al hash `url:<url_id>:<user_id>` con un parámetro extra que indica que la URL que acabamos de agregar pertenece al wishlist del usuario: `wishlist = 1`. Adicionalmente, se toma la categoría de la URL a agregar al wishlist y se agrega al conjunto `categories:<user_id>`

Implementación en Redis:

1. Carga de script

```
127.0.0.1:6379> script load "local user_id = KEYS[1] local url_seq_key = KEYS[2] local rurl_seq_key = KEYS[3] local base_client = KEYS[4] lo
cal url_to_add = ARGV[1] local category = ARGV[2] local public = ARGV[3] local private = ARGV[4] local wishlist_key = 'wishlist:'..user_id l
ocal new_url_key = redis.call('incr', url_seq_key) local rurl_new_key = redis.call('incrby', rurl_seq_key, 25) local rurl = base_client..rur
l_new_key local key = 'urls:'..new_url_key..':'..user_id redis.call('hset', key, 'url', url_to_add, 'rurl', rurl, 'public', public, 'private
', private, 'category', category, 'wishlist', '1') redis.call('rpush', wishlist_key, new_url_key) local categories_key = 'categories:'..user
_id redis.call('sadd', categories_key, category) return redis.call('lrange', wishlist_key, 0, -1)"
"346068a656922cca96c6267768de29d64ca7d251"
127.0.0.1:6379> set add_to_wishlist:sha 346068a656922cca96c6267768de29d64ca7d251
OK
127.0.0.1:6379>
```

Figura 11: Carga de script para wishlist

2. Adición de URLs al wishlist

En la siguiente figura se muestra la adición de una URL al wishlist de dos usuario. Tenemos que verificar sobre el hash `url:<URL_id>:<user_id>`, la lista `wishlist:<user_id>` y el conjunto `categories:<user_id>`. Notemos ademas que en el hash, cuando un usuario agrega una URL a su wishlist, éste tiene un parámetro extra: `wishlist = 1`.

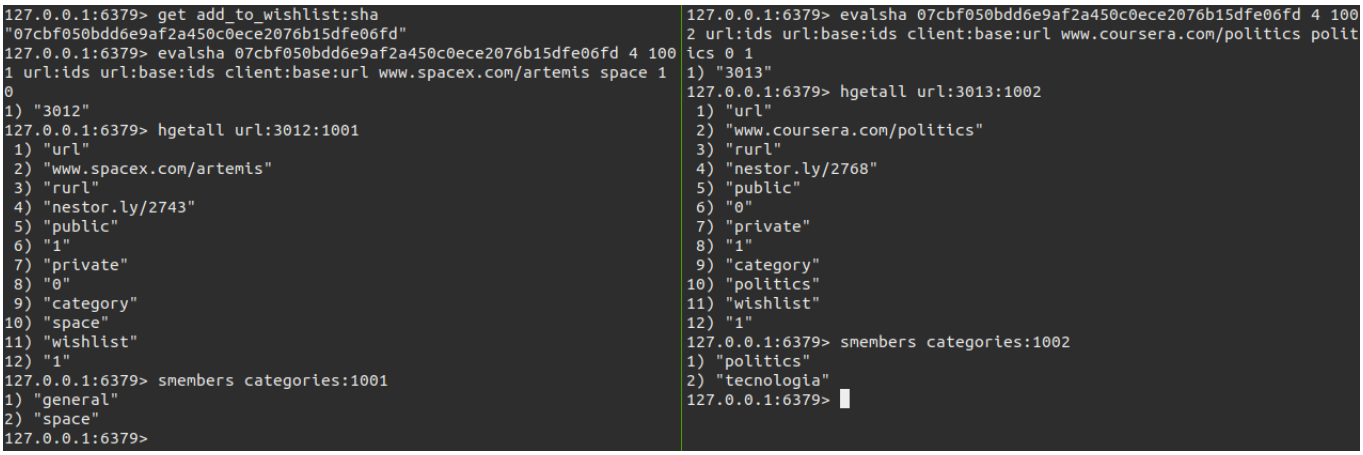


Figura 12: Adición de URLs al wishlist

3. Verificación

Figura 13: Verificación del wishlist

2.2.7. Intersección de categorías - ADMIN

En la siguiente figura vemos la intersección de categorías ocupando conjuntos (sets).

3. Conclusiones

Una de las desventajas principales de Redis, algo que no mencioné en la introducción pero que sí me tope, es la complejidad para hacer queries. Uno tiene que cambiar de paradigma completamente para diseñar los modelos y obtener información de la base de datos. De los conceptos de Redis con los cuales me quedo para el desarrollo de esta práctica sin duda son la velocidad y flexibilidad para

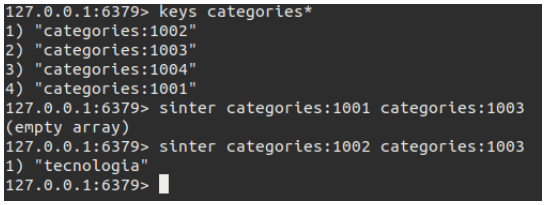


Figura 14: Intersección de categorías

modelar prácticamente cualquier problema. Hablando de problemas, esta práctica la dejo como un desarrollo a futuro para implementarla ocupando otra estructura de datos (introducida en la versión 5 de Redis) llamada **Streams**. Dicha estructura hubiera facilitado la manera de interactuar con la base de datos porque su funcionalidad es tener una serie de productores (en este caso los clientes) y otra serie de consumidores (el servidor) para atender a las peticiones. Si hubiera ocupado **Streams**, la práctica hubiera tomado un camino más sólido en cuanto al desarrollo de la práctica como una web app o una app móvil.

Finalmente, lo que más trabajo me costó a la hora de desarrollar la práctica fue el diseño del modelo (figura 2) pues no estoy acostumbrado a trabajar con bases de datos llave valor.

Referencias

[1] Seguin K. *The Little Redis Book*. (2015). Disponible en: `https://github.com/karlseguin/the-little-redis-book`

[2] Carlson J. *Redis in Action*. (2013). Manning. Disponible en: `https://redislabs.com/ebook/redis-in-action/`

[3] Pimentel, A. *Notas de clase: Bases de Datos No Estructuradas: Redis*. (2021). Instituto de Investigaciones en Matemáticas Aplicadas y en Sistemas. Universidad Nacional Autónoma de México

[4] Redis. *Command Reference*. (2021). Disponible en: `https://redis.io/commands#`