IES SAN ANDRÉS

ESPECIALIZACIÓN EN INTELIGENCIA ARTIFICIAL Y BIG DATA

SISTEMAS DE BIG DATA

UT02: ALMACENAMIENTO DE DATOS

ÍNDICE

1

INTRODUCCIÓN AL ALMACENAMIENTO EN BIG DATA

1 INTRODUCCIÓN AL ALMACENAMIENTO EN BIG DATA

1.1 ¿QUÉ ES EL ALMACENAMIENTO EN BIG DATA?

El almacenamiento en Big Data se refiere a las **arquitecturas, sistemas** y **tecnologías** diseñadas para **almacenar, gestionar y procesar volúmenes masivos de datos**.

Son cantidades tan grandes que los sistemas de bases de datos tradicionales no pueden manejarlos

Las soluciones comunes incluyen:

- Almacenamiento en la **nube** (S3, Azure Blob Storage)
- Sistemas de archivos **distribuidos** (Hadoop)
- Bases de datos **NoSQL** para datos no estructurados

ESQUEMA

Esquema fijo y rígido (Schema-on-Write).

Se define la estructura de la tabla antes de insertar datos.

Esquema flexible o dinámico (Schema-on-Read).

La estructura se interpreta al leer los datos, permitiendo variedad.

ESCALABILIDAD

Escalabilidad vertical (Scale-Up)

Se aumenta la potencia (CPU, RAM, disco) de un único servidor. Tiene límites físicos y es costosa

Escalabilidad horizontal (Scale-Out).

Se añaden más servidores commodity (económicos) al clúster. Es casi ilimitada y más económica.

ARQUITECTURA

Centralizada

Un único servidor gestiona todos los datos y operaciones.

Distribuida

Los datos y el procesamiento se reparten entre decenas, cientos o miles de nodos

TIPOS DE DATOS

Datos principalmente estructurados

Tablas con filas y columnas bien definidas

Todo tipo de datos

Estructurados, semi-estructurados (JSON, logs) y no estructurados (imágenes, videos).

MODELO DE DATOS

Relacional

Basado en tablas, claves primarias/foráneas y SQL

No relacional (NoSQL) o sistemas de archivos distribuidos

Optimizados para patrones de acceso específicos.

RENDIMIENTO

Optimizado para **transacciones complejas** (OLTP) y **consultas ACID.**

Optimizado para el procesamiento por lotes (batch) y, en algunos casos, tiempo real. Prioriza el *throughput* sobre la latencia en algunas operaciones.

COSTO

Alto

Suele utilizar hardware propietario y de alta gama, lo que incrementa el costo.

Bajo

Se basa en hardware commodity (estándar y económico) y software de código abierto, reduciendo costos.

2

BASES DE DATOS CLAVE-VALOR REDIS

2 BASES DE DATOS CLAVE-VALOR REDIS

2.1 ¿QUÉ ES REDIS?

Redis (REmote DIctionary Server) es una base de datos en **memoria**, de tipo **clave-valor**, **no relacional** (**NoSQL**).

Se caracteriza por su altísima velocidad, ya que guarda los datos principalmente en memoria RAM, aunque también permite la persistencia en disco.

Es software libre, desarrollado en C, y está disponible bajo la licencia BSD

Sus principales **características** son:

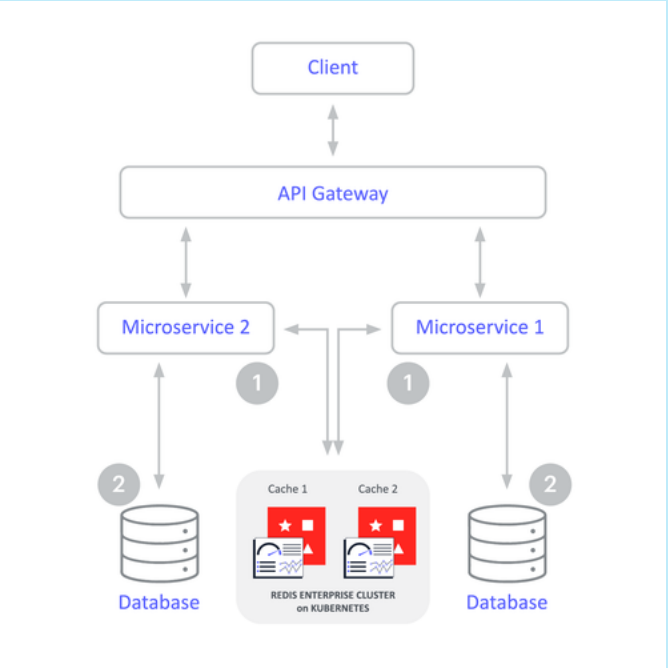
Almacenamiento en memoria	**Estructuras de datos avanzadas**	**Persistencia opcional**
Redis guarda los datos en RAM, lo que permite operaciones extremadamente rápidas	No solo almacena cadenas, sino también listas, conjuntos, hashes, conjuntos ordenados, streams, …	Aunque funciona en memoria, puede guardar datos en disco mediante mecanismos como RDB o AOF.
Simplicidad y rendimiento	**Alta disponibilidad y escalabilidad**	**Soporte para Pub/Sub**
Su arquitectura es muy eficiente, basada en un solo hilo, lo que facilita la predicción del rendimiento	Soporta replicación maestro-esclavo, clústeres y Redis Sentinel para alta disponibilidad	Redis permite enviar y recibir mensajes entre procesos mediante un sistema de publicación y suscripción

Algunos ejemplos de **casos de uso** de Redis:

Caché de alto rendimiento

Se usa frecuentemente como **caché en memoria** para almacenar datos que deben consultarse rápidamente y que no cambian con frecuencia.

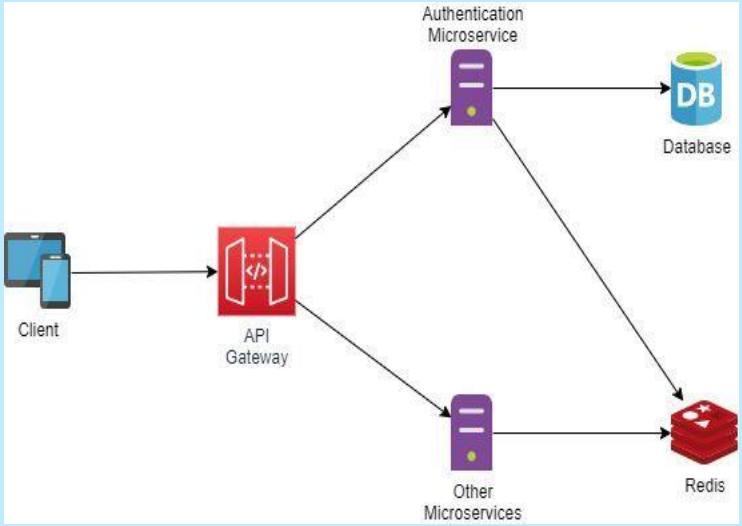
Esto reduce la carga del servidor principal y mejora significativamente el tiempo de respuesta de las aplicaciones web.



Almacenamiento de sesiones

Redis permite almacenar datos de sesión de usuarios, como tokens, ID de usuario, preferencias, …

Compatible con múltiples *frameworks* y permite establecer tiempos de expiración automáticos.



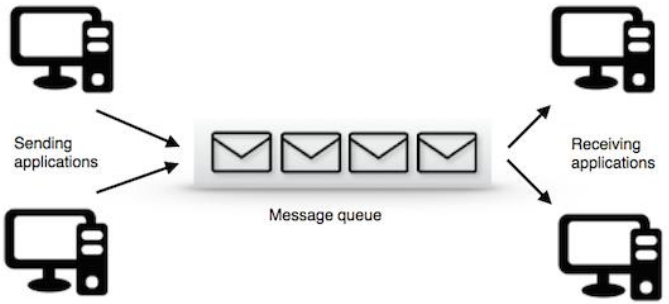
Contadores y estadísticas tiempo real

Gracias a sus operaciones atómicas y velocidad, Redis es ideal para implementar **contadores** y **métricas en tiempo real**.

Por ejemplo: visitas a una página, número de clicks de un botón o seguimiento de eventos en tiempo real (*likes*)

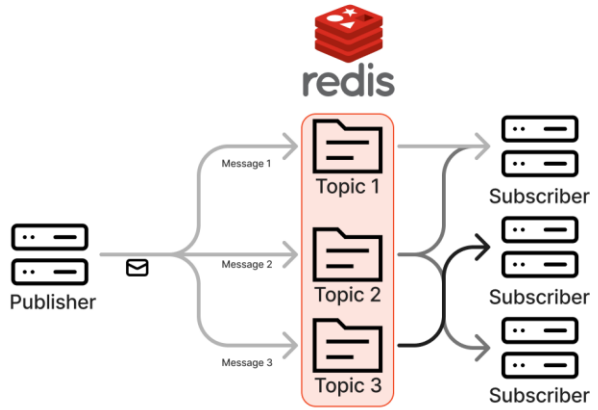
Colas de trabajo

Redis permite implementar **colas de tareas asíncronas** entre procesos o servicios. Esto es útil para **desacoplar procesos pesados** que no deben ejecutarse durante la respuesta al usuario: procesamiento de imágenes, e-mails, generación de PDFS…



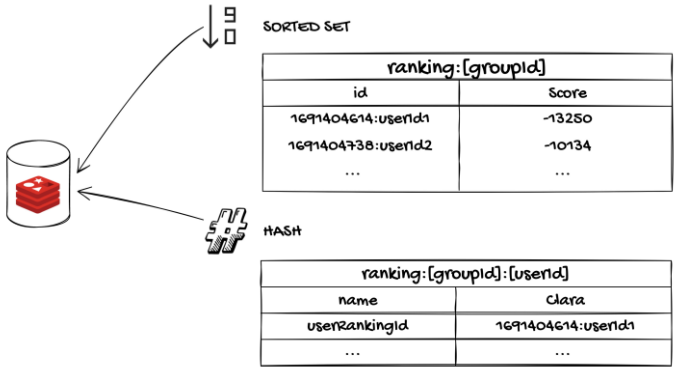
Sistemas de publicación/suscripción

Redis incluye un **sistema Pub/Sub** para permitir la comunicación entre procesos, útil en sistemas distribuidos o aplicaciones en tiempo real: chats, notificaciones a clientes, juegos multijugador,…



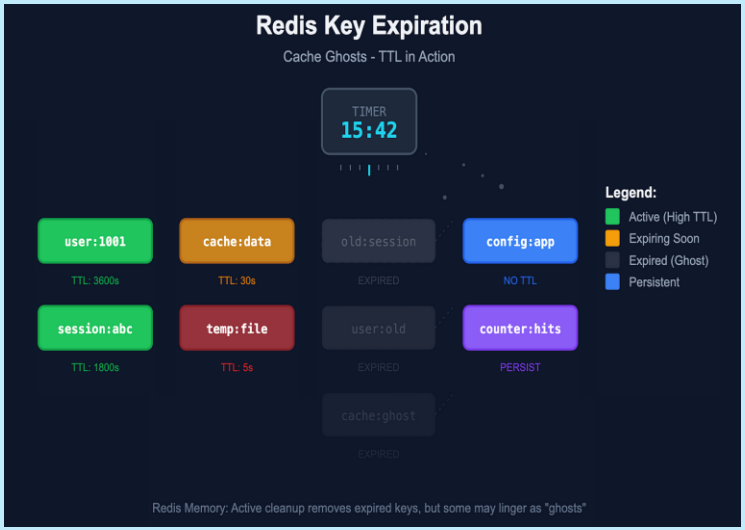
Rankings y puntuaciones

El tipo de datos **Sorted Set** permite mantener elementos ordenados por una puntuación, ideal para rankings (usuarios en un videojuego, puntuaciones en un concurso) o simplemente resultados ordenados por relevancia.



Almacenamiento temporal

Redis permite asignar **tiempos de vida (TTL)** a las claves, lo que es útil para almacenar datos que deben eliminarse automáticamente: tokens de autenticación, códigos temporales de verificación (OTP) o cachés de resultados que caducan.



2 BASES DE DATOS CLAVE-VALOR REDIS

2.2 INSTALACIÓN DE REDIS

docker-compose.yml

```yaml
Run All Services
services:
  Run Service
  redis:
    image: redis:7.2
    container_name: redis-server
    ports:
      - "6379:6379"
    volumes:
      - redis-data:/data
    command: ["redis-server", "--appendonly", "yes"]
    restart: unless-stopped
    networks:
      - bigdata-network

volumes:
  redis-data:

networks:
  bigdata-network:
    external: true
```

En este ejemplo he instalado la versión 7.2

Redis se expone en el puerto 6379

Si queremos persistencia debemos mapear el volumen /data

Recuerda que esta red es la que vamos a utilizar para interconectar todos nuestros contenedores

Acceso con Redis CLI

Una vez arrancado el contenedor comprobamos que funciona correctamente ejecutando **redis-cli** dentro del mismo

```
PS C:\> docker exec `
>> -it `
>> redis-server `
>> redis-cli
127.0.0.1:6379>
```

Veamos ahora los comandos básicos de **administración general** de Redis.

PING

Verifica si el servidor Redis está activo, responde PONG si hay conexión con el servidor.

```
127.0.0.1:6379> PING
PONG
127.0.0.1:6379>
```

QUIT

Cierra la conexión con Redis

INFO [sección]

Muestra información del servidor, opcionalmente se puede indicar una única sección (server, memory, …)

```
127.0.0.1:6379> INFO memory
# Memory
used_memory:938704
used_memory_human:916.70K
used_memory_rss:14868480
used_memory_rss_human:14.18M
used_memory_peak:1158976
used_memory_peak_human:1.11M
used_memory_peak_perc:80.99%
used_memory_overhead:868056
used_memory_startup:865752
```

CONFIG GET <parámetro>

Consulta la configuración actual. Algunos parámetros son:

- `server_threads` → Hilos del servidor.
- `maxclients` → Número máximo de clientes conectados.
- `timeout` → Tiempo en segundos que se espera antes de cerrar conexiones inactivas.
- `tcp-keepalive` → Tiempo para mantener conexiones TCP vivas.
- `maxmemory` → Límite máximo de memoria (0 = sin límite).
- `maxmemory-policy` → Política de expulsión (noeviction, allkeys-lru, volatile-lru, etc.).
- `maxmemory-samples` → Número de muestras para algoritmos LRU/LFU.
- `hash-max-ziplist-entries` / `hash-max-ziplist-value` → Optimización de hashes pequeños.
- `bind` → Interfaces en las que escucha Redis.
- `port` → Puerto del servidor (por defecto 6379).
- `unixsocket` → Ruta de un socket UNIX (si se usa)
- `logfile` → Fichero de logs (si está configurado)

CONFIG GET <parámetro>

```
127.0.0.1:6379> config get maxclients
1) "maxclients"
2) "10000"
127.0.0.1:6379> config get logfile
1) "logfile"
2) ""
127.0.0.1:6379> config get maxmemory
1) "maxmemory"
2) "0"
```

CONFIG SET <parámetro> <valor>

Cambia los parámetros de configuración en tiempo real.

```
127.0.0.1:6379> config get maxclients
1) "maxclients"
2) "10000"
127.0.0.1:6379> config set maxclients 100
OK
127.0.0.1:6379> config get maxclients
1) "maxclients"
2) "100"
```

SELECT <índice>

Cambia a una base de datos diferente, identificando cada base de datos por un número entre 0 y 15

El uso de múltiples bases de datos no es común en producción, donde se suele utilizar **prefijos de claves** para manejar el aislamiento.

SELECT solo afecta a la conexión actual, si se abre una nueva conexión se empezará nuevamente en la base de datos 0

DBSIZE

Devuelve el número de claves en la base de datos actual

```
127.0.0.1:6379> DBSIZE
(integer) 3
127.0.0.1:6379> SELECT 1
OK
127.0.0.1:6379[1]> DBSIZE
(integer) 0
127.0.0.1:6379[1]>
127.0.0.1:6379[1]>
```

FLUSHDB

Borra todas las claves de la base de datos actual

```
127.0.0.1:6379> DBSIZE
(integer) 3
127.0.0.1:6379> FLUSHDB
OK
127.0.0.1:6379> DBSIZE
(integer) 0
127.0.0.1:6379>
```

FLUSHALL

Borra todas las claves de **todas** las bases de datos

CLIENT LIST

Muestra los clientes conectados

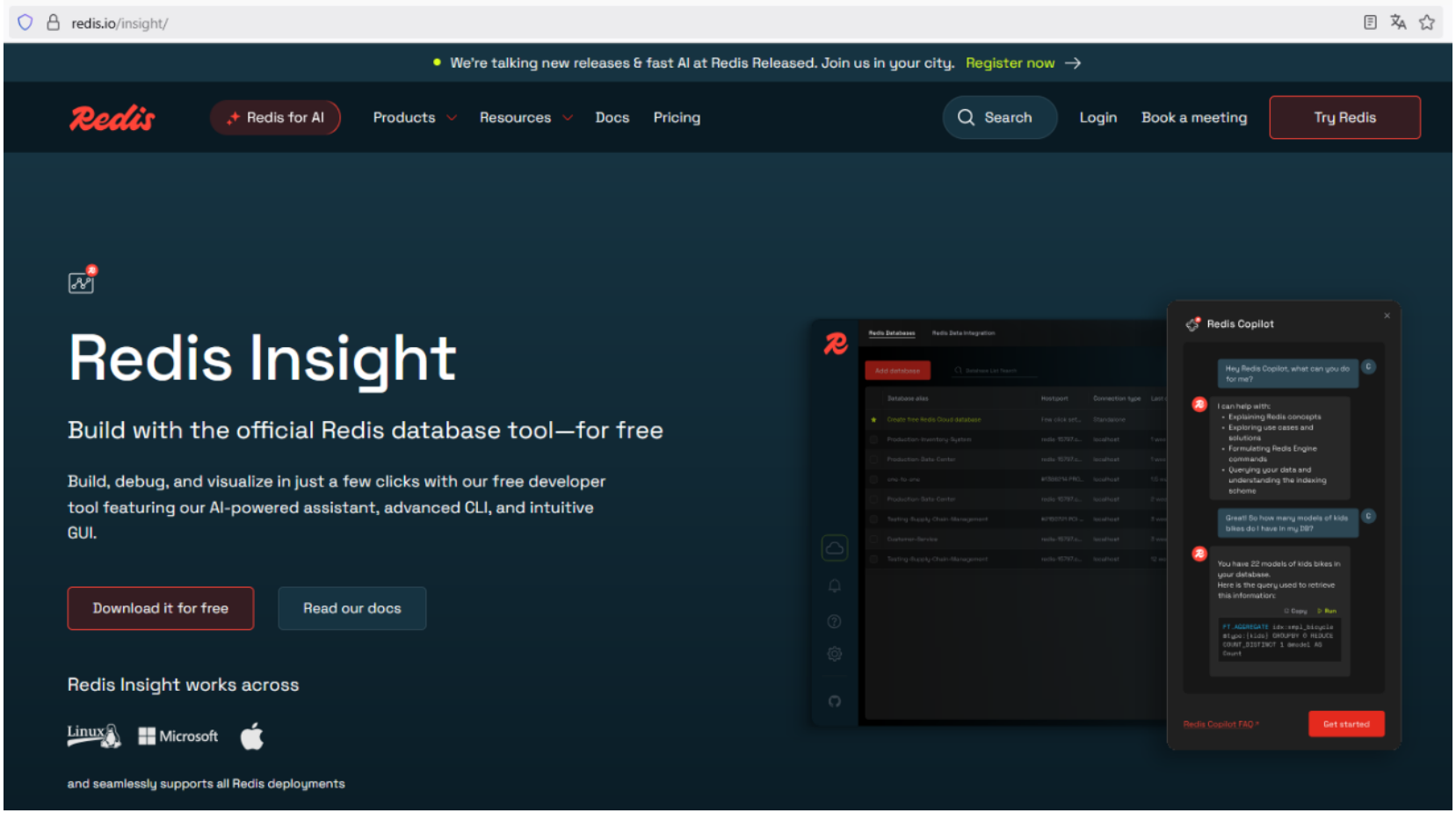
```
127.0.0.1:6379> client list
id=25 addr=127.0.0.1:39940 laddr=127.0.0.1:6379 fd=8 name=
lti-mem=0 rbs=1024 rbp=0 obl=0 oll=0 omem=0 tot-mem=22426
127.0.0.1:6379>
```

CLIENT KILL <id>

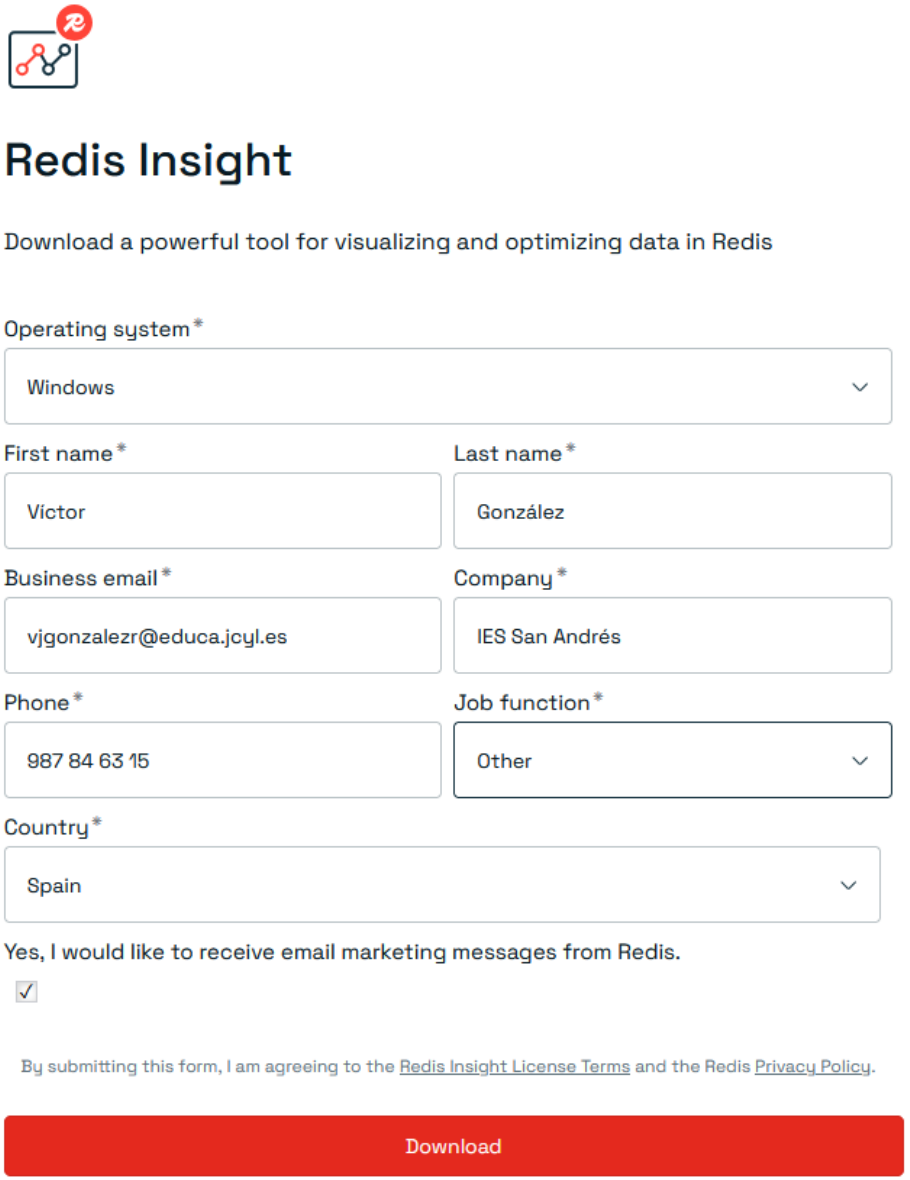
Cierra la conexión de un cliente

Acceso con Redis Insight

Si queremos un cliente gráfico podemos instalar **Redis Insight**

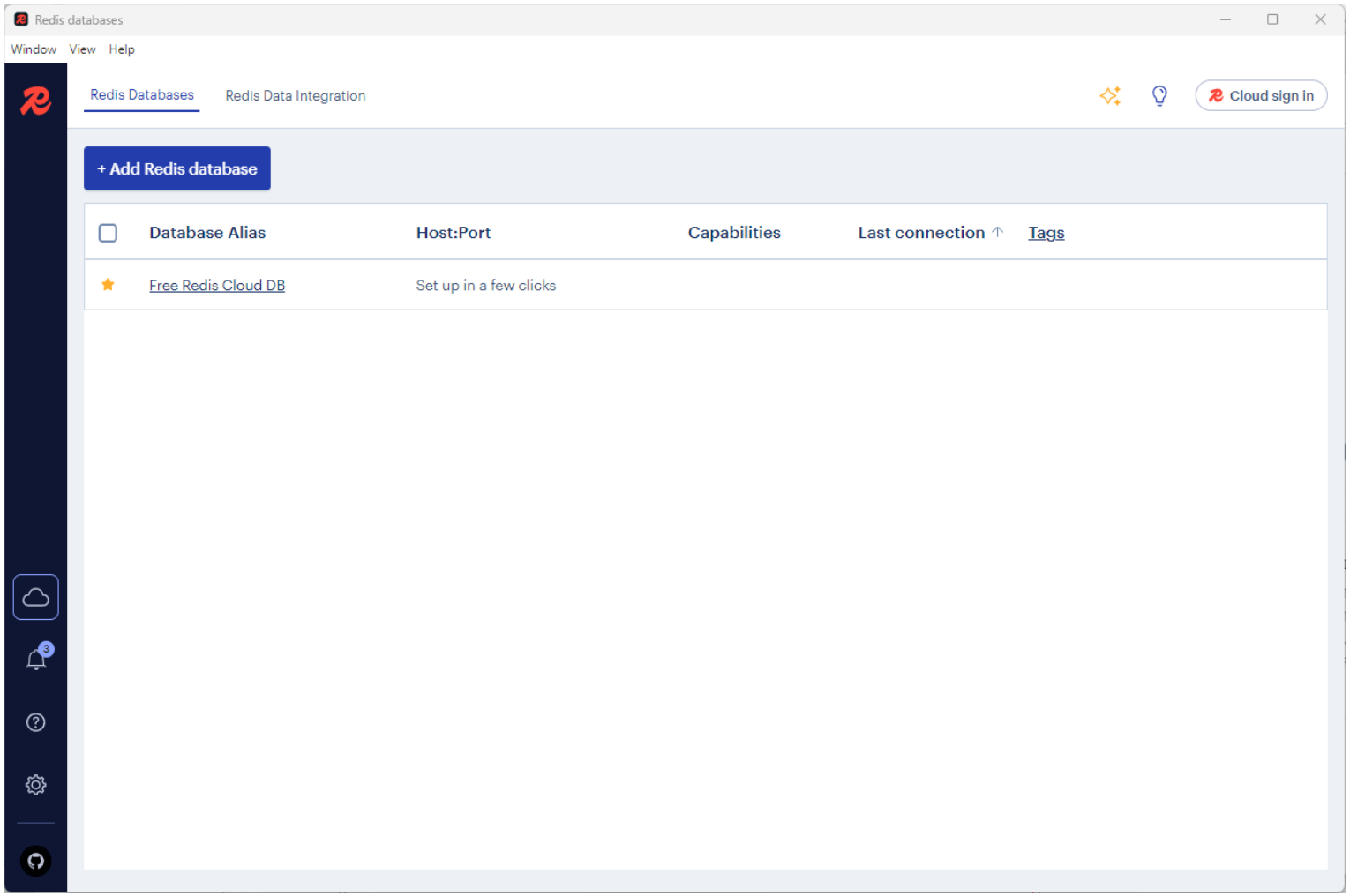


Para descargarlo tendremos que introducir nuestros datos

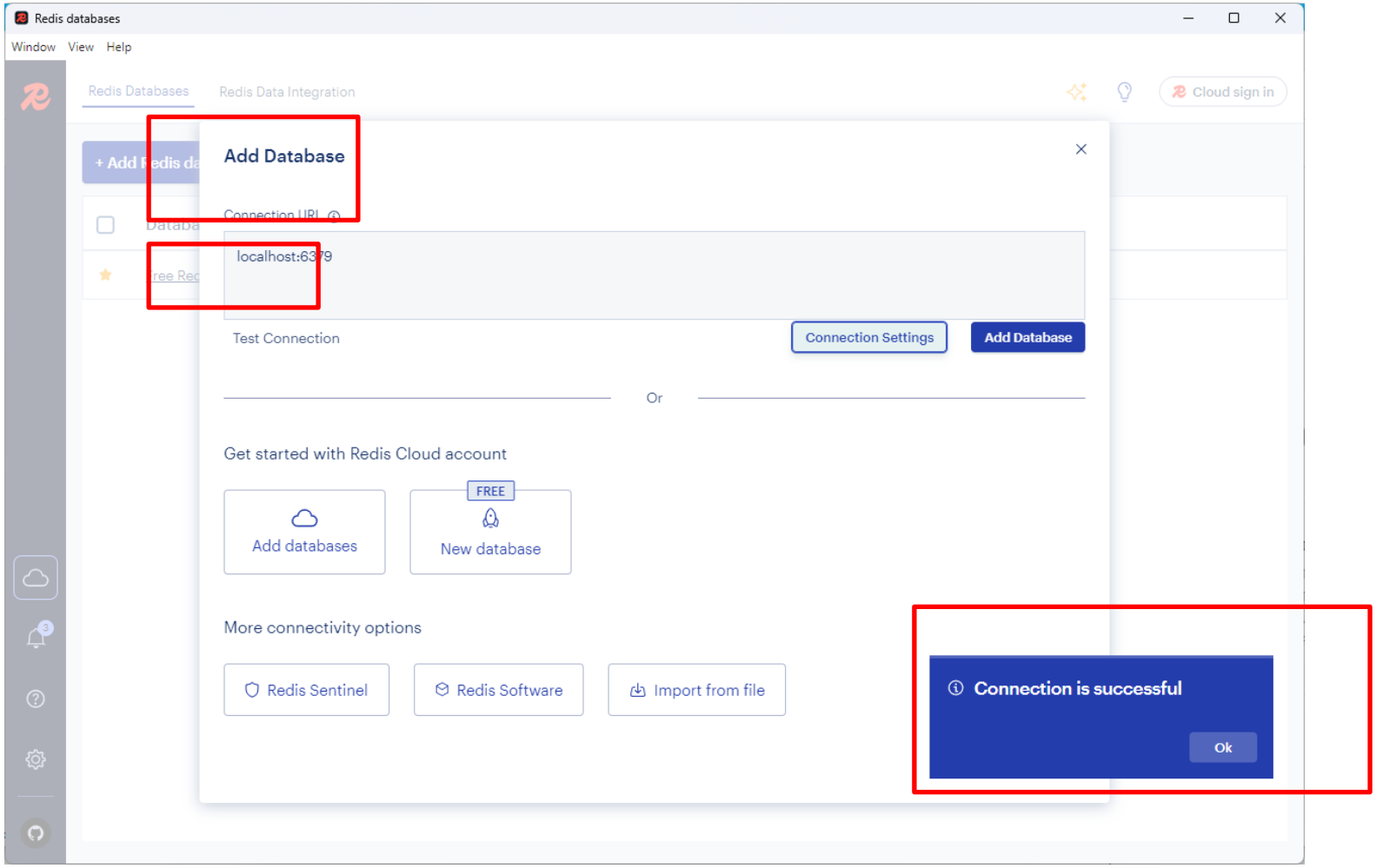


Aceptamos el EULA

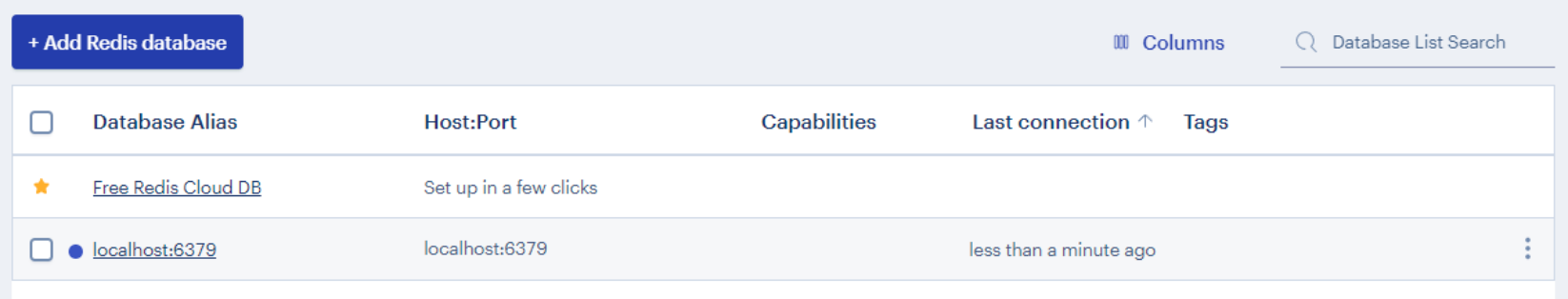
Y ya accedemos a la pantalla principal



El primer paso será conectarnos a una base de datos



Podremos tener conexiones a diferentes bases de datos



Acceso desde Python

Por último, también podremos acceder a Redis desde Python utilizando la librería `redis-py`

redis-py

The Python interface to the Redis key-value store.

Installation | Usage | Advanced Topics | Contributing

NOTA: las siguientes capturas se realizan en un entorno configurado con un contenedor ejecutando **Redis** y otro con **Jupyter Notebook**.

```
PS C:\proyectos\docker_resources\docker_compose\jupyter_notebook> docker network create shared-network
a675558a092cd6dbed7c6f4aa5d4b1e3a362ebefc0ed535ed89bfad91c5dad1b
PS C:\proyectos\docker_resources\docker_compose\jupyter_notebook> docker compose up -d
[+] Running 1/1
 ✓ Container jupyter_datascience  Started
PS C:\proyectos\docker_resources\docker_compose\jupyter_notebook> cd ..\redis\
PS C:\proyectos\docker_resources\docker_compose\redis> docker compose up -d
[+] Running 1/1
 ✓ Container redis-server  Started
PS C:\proyectos\docker_resources\docker_compose\redis>
```

Recuerda que Redis se expone a través del puerto **6379** y Jupyter del **8888**. Asimismo, al estar conectados a la misma red externa (`shared-network`), podremos acceder a Redis desde Jupyter usando la resolución local de nombres de Docker.

☐	⌄	●	jupyter_notebook	-	-	-
☐		●	jupyter_datascience	a44d8dd78797	jupyter/datascience-noteb	8888:8888
☐	⌄	●	redis	-	-	-
☐		●	server	c5b180c89451	redis:7.2	6379:6379

El primer paso será instalar el módulo `redis-py` utilizando el comando `pip`

```
[1]: !pip install redis

Collecting redis
  Downloading redis-6.4.0-py3-none-any.whl.metadata (10 kB)
Downloading redis-6.4.0-py3-none-any.whl (279 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 279.8/279.8 kB 4.4
Installing collected packages: redis
Successfully installed redis-6.4.0
```

Ten en cuenta que **`pip`** no es un comando Python, sino un comando del sistema para gestionar los paquetes instalados.

Para indicarle a Jupyter que se tiene que ejecutar en una terminal **lo precedemos del símbolo exclamación.**

Y ya podremos conectarnos

Importo el módulo redis

Uso la clase Redis que representa la conexión a la base de datos

Utilizo la resolución local de nombres de Docker para indicar el servicio que contiene Redis

En este caso me estoy conectando a la base de datos 0

Con este parámetro indicamos que devuelva strings en lugar de bytes

```
import redis

r = redis.Redis(
    host='redis',
    port=6379,
    db=0,
    decode_responses=True
)

print(r.ping())
```

```
True
```

Y pruebo la conexión

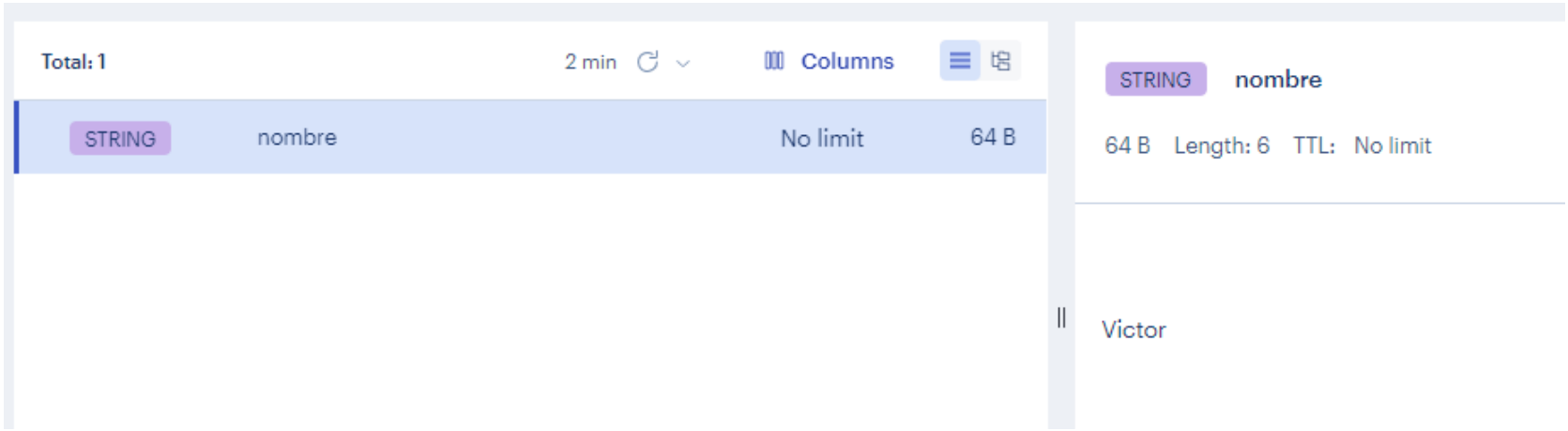
Si no ha habido errores ya podré almacenar y recuperar valores en Redis

Uso la variable que me devolvió la conexión

```
r.set("nombre", "Victor")
r.get("nombre")
```

```
'Victor'
```

El método `set` almacena un par clave-valor de tipo string, mientras que `get` lo recupera

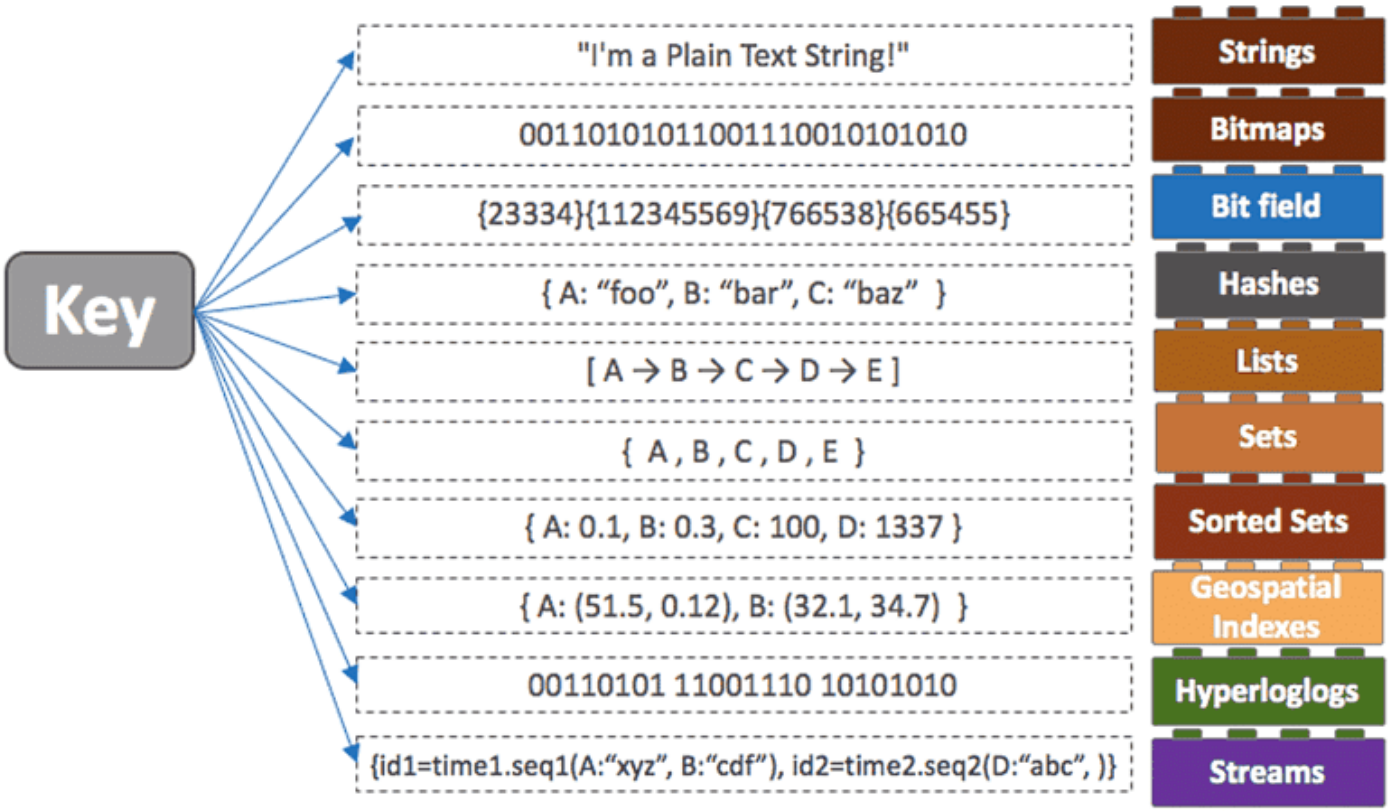


2 BASES DE DATOS CLAVE-VALOR REDIS

2.3 PRIMEROS PASOS CON REDIS

Redis es una **base de datos clave-valor**, donde

- La **clave** siempre es una cadena
- El **valor** puede ser de diferentes tipos, según la información que queramos almacenar o el objetivo de la misma.



REDIS CLI SET <clave> <valor> / GET

Almacena **una clave y su valor** asociado en la base de datos. Guarda los valores en forma de **cadena de texto** (string)

Los valores se pueden recuperar con el comando **GET**

```
127.0.0.1:6379> SET nombre "Victor"
OK
127.0.0.1:6379> GET nombre
"Victor"
127.0.0.1:6379>
```

```
r.set("nombre", "Victor")
r.get("nombre")
```

```
'Victor'
```

SET dispone de una serie de opciones adicionales:

- **EX**: establece un tiempo de expiración en segundos
- **PX**: igual que EX, pero en milisegundos
- **NX**: solo guarda la clave si no existe ya
- **XX**: solo guarda la clave si ya existe

```
127.0.0.1:6379> SET token "1234" EX 300
OK
127.0.0.1:6379> GET token
"1234"
```

La clave se eliminará automáticamente en 5 minutos

```
127.0.0.1:6379> SET nombre "Victor"
OK
127.0.0.1:6379> SET "nombre" "Victor Gonzalez" NX
(nil)
```

Como la clave ya existe no escribe nada en la base de datos

PYTHON `r.set(name, value, ex=None, px=None, nx=False, xx=False)`

```
[7]: r.set("mensaje", "hola", ex=10)

[7]: True

[8]: r.get("mensaje")

[8]: 'hola'

[9]: r.get("mensaje")

[ ]:
```

Han pasado más de 10 segundos

```
[10]: # Se guardará solo si 'contador' no existe
      r.set("contador", 1, nx=True)

[10]: True

[11]: r.get("contador")

[11]: '1'

[12]: r.set("contador", 100, nx=True)

[13]: r.get("contador")

[13]: '1'
```

Como la clave ya existe no tiene efecto

REDIS CLI KEYS <patrón>

Busca claves que coincidan con un patrón (ej: KEYS user:*).

No es conveniente usarlo en producción ya que es bloqueante.

```
127.0.0.1:6379> SET "user:name" "Victor"
OK
127.0.0.1:6379> SET "user:apellidos" "Gonzalez Rodriguez"
OK
127.0.0.1:6379> SET "pass" "1234"
OK
127.0.0.1:6379> KEYS user:*
1) "user:apellidos"
2) "user:name"
```

Observa que en Redis se suele usar una nomenclatura jerárquica con el carácter dos puntos como separador lógico.
El formato suele ser:
`objeto:id:atributo`
Por ejemplo:
`usuario:100:apellidos`

PYTHON `r.keys (pattern="*")`

```
r.set("nombre", "Ana")
r.set("curso", "Redis")
r.set("contador", 5)

print(r.keys("c*"))
```

```
['contador', 'curso']
```

REDIS CLI EXISTS <clave>

Verifica si una clave existe

```
127.0.0.1:6379> EXISTS "user:name"
(integer) 1
127.0.0.1:6379> EXISTS "user:nickname"
(integer) 0
127.0.0.1:6379>
```

REDIS CLI DEL <clave>

Elimina una clave

```
127.0.0.1:6379> DEL "user:apellidos"
(integer) 1
127.0.0.1:6379> EXISTS "user:apellidos"
(integer) 0
127.0.0.1:6379>
```

PYTHON `r.exists (key)`

PYTHON `r.delete (key1, key2, …)`

```
r.set("nombre", "Ana")

print(r.exists("nombre"))
print(r.exists("apellido"))
```

```
1
0
```

REDIS CLI EXPIRE <clave> <segundos>

Establece el tiempo de vida (TTL) de una clave en segundos

REDIS CLI TTL <clave>

Muestra el tiempo restante de vida de una clave (-2 si no tiene establecido tiempo de vida)

```
127.0.0.1:6379> EXPIRE "user:nombre" 60
(integer) 1
127.0.0.1:6379> TTL "user:nombre"
(integer) 52
127.0.0.1:6379> TTL "user:nombre"
(integer) 46
127.0.0.1:6379>
```

PYTHON `r.expire(name, time)`

PYTHON `r.ttl(name)`

PYTHON `r.type(name)`

```python
import time

r.set("mensaje", "Hola mundo", ex=10)
print("TTL inicial:", r.ttl("mensaje"))
time.sleep(3)
print("TTL después de 3s:", r.ttl("mensaje"))
time.sleep(8)
print("TTL después de 11s:", r.ttl("mensaje"))
```

Espera el tiempo indicado como parámetro

```
TTL inicial: 10
TTL después de 3s: 7
TTL después de 11s: -2
```

Valores posibles:

- **>= 0**: segundos restantes
- **-1**: la clave existe, pero no tiene expiración
- **-2**: la clave no existe

REDIS CLI TYPE <clave>

Hasta ahora los valores solo han sido cadenas de texto, pero Redis soporta **múltiples tipos de datos** (que veremos en el siguiente apartado). Con el comendo TYPE podemos ver qué tipo de datos almacena una determinada clave.

```
127.0.0.1:6379> TYPE "user:nombre"
string
127.0.0.1:6379>
```

2 BASES DE DATOS CLAVE-VALOR REDIS

2.4 TIPOS DE DATOS EN REDIS

CADENAS (STRINGS)

El tipo de dato más básico. Una **cadena** en Redis puede contener cualquier tipo de datos binarios (imágenes, texto serializado, etc.), hasta un tamaño de 512 MB.

Son ideales para almacenar valores simples como texto, números, o JSON.

Casos de uso comunes para las cadenas en Redis son:

- **Caching**: almacenar resultados de consultas a bases de datos o APIs.
- **Contadores**: contadores de visitas, "*me gusta*", etc.
- **Almacenamiento de sesiones**: guardar datos de sesión de usuario serializados.
- **Banderas (flags):** almacenar estados simples (ej. "activo", "inactivo").

Comandos para trabajar con cadenas:

- **SET** ***key value***: establece el valor de una clave
- **GET** ***key***: recupera el valor de una clave
- **DEL** ***key***: elimina una clave
- **INCR** ***key***: incrementa el valor numérico de una clave en 1
- **DECR** ***key***: decremente el valor numérico de una clave en 1
- **EXPIRE** ***key seconds***: establece un tiempo de vida (TTL) para una clave
- **TTL** ***key***: obtiene el tiempo de vida restante de una clave

```
127.0.0.1:6379> SET user:1:name "Victor"
OK
127.0.0.1:6379> GET user:1:name
"Victor"
```

```
127.0.0.1:6379> SET page:views 0
OK
127.0.0.1:6379> INCR page:views
(integer) 1
127.0.0.1:6379> GET page:views
"1"
```

Incrementamos el valor en 1

```
127.0.0.1:6379> SET user:token "1234" EX 60
OK
127.0.0.1:6379> TTL user:token
(integer) 55
```

La clave se eliminará automáticamente en 60 segundos

LISTAS (LISTS)

Las listas son una **colección ordenada** de cadenas.

Se implementan como listas enlazadas, lo que permite operaciones de **inserción y eliminación muy rápidas en los extremos**.

Son ideales para implementar **colas** o **pilas**.

Se utilizan habitualmente para:

- **Colas de mensajes**: implementar colas de tareas para procesadores en segundo plano.
- **Pilas (stacks)**: historial de acciones recientes.
- **Feeds de redes sociales**: almacenar los últimos N elementos de un feed.
- **Listas de reproducción**: mantener el orden de elementos.

Comandos para trabajar con cadenas:

- **LPUSH** ***key value [value …]*****:** añade uno o más valores al inicio de la lista.
- **RPUSH** ***key value [value …]*****:** añade uno o más valores al final de la lista.
- **LPOP** ***key*****:** elimina y devuelve el primer elemento de la lista.
- **RPOP** ***key*****:** elimina y devuelve el último elemento de la lista.
- **LRANGE** ***key start stop*****:** obtiene un rango de elementos de la lista.
- **LLEN** ***key*****:** obtiene la longitud de la lista.
- **BLPOP** ***key [key …] timeout*****:** versión bloqueante de LPOP, espera hasta que haya elementos.

```
127.0.0.1:6379> RPUSH tasks "task1" "task2" "task3"
(integer) 3
127.0.0.1:6379> LRANGE tasks 0 -1
1) "task1"
2) "task2"
3) "task3"
127.0.0.1:6379> LLEN tasks
(integer) 3
127.0.0.1:6379> LPOP tasks
"task1"
127.0.0.1:6379> LLEN tasks
(integer) 2
127.0.0.1:6379> LRANGE tasks 0 -1
1) "task2"
2) "task3"
```

Inicializamos una lista con 3 elementos

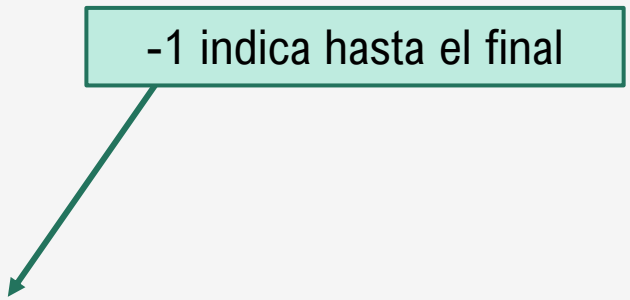
Con 0 -1 indicamos desde el primer elemento hasta el último

POP devuelve el primer elemento de la lista y también lo elimina

PYTHON

```python
r.lpush(name, *values)
r.rpush(name, *values)
r.lpop(name, count=None)
r.rpop(name, count=None)
r.lrange (name, start, end)
r.llen(name)
r.blpop(keys, timeout=None)
```

```python
lista_nombre = "tareas"

# Agregar elementos a la lista (al final)
r.rpush(lista_nombre, "Estudiar Python")
r.rpush(lista_nombre, "Hacer ejercicio")
r.rpush(lista_nombre, "Comprar comida")

# Mostrar todos los elementos de la lista
todas_tareas = r.lrange(lista_nombre, 0, -1)
print("Lista completa de tareas:", todas_tareas)

# Sacar un elemento desde el principio (FIFO)
tarea_inicial = r.lpop(lista_nombre)
print("Tarea eliminada desde el inicio:", tarea_inicial)

# Sacar un elemento desde el final (LIFO)
tarea_final = r.rpop(lista_nombre)
print("Tarea eliminada desde el final:", tarea_final)

# Mostrar la lista actualizada
tareas_actualizadas = r.lrange(lista_nombre, 0, -1)
print("Lista actualizada de tareas:", tareas_actualizadas)
```



CONJUNTOS (SETS)

Un conjunto es una **colección desordenada de cadenas únicas**.

Los conjuntos son ideales para almacenar elementos donde **el orden no importa y se necesita asegurar la unicidad**, o para realizar operaciones de teoría de conjuntos.

Los conjuntos se suelen utilizar para:

- **Etiquetas (tags):** asociar tags a artículos, fotos, etc.
- **Miembros únicos**: lista de usuarios únicos en un chat, visitantes únicos.
- **Relaciones**: "usuarios que siguen a X", "productos en el carrito".
- **Operaciones de conjuntos**: amigos en común, usuarios con intereses compartidos

Los comandos para trabajar con conjuntos son:

- **SADD** ***key member [member …]***: añade uno o más miembros a un conjunto.
- **SREM** ***key member [member …]***: elimina uno o más miembros de un conjunto.
- **SMEMBERS** ***key***: obtiene todos los miembros de un conjunto.
- **SISMEMBER** ***key member***: comprueba si un miembro existe en un conjunto.
- **SCARD** ***key***: obtiene el número de miembros en un conjunto.
- **SINTER** ***key [key …]***: devuelve la intersección de dos o más conjuntos.
- **SUNION** ***key [key …]***: Devuelve la unión de dos o más conjuntos.

```
127.0.0.1:6379> SADD users:online "Alice" "Bob" "Charlie"
(integer) 3
127.0.0.1:6379> SMEMBERS users:online
1) "Alice"
2) "Bob"
3) "Charlie"
127.0.0.1:6379> SISMEMBER users:online "Alice"
(integer) 1
127.0.0.1:6379> SISMEMBER users:online "Edgar"
(integer) 0
127.0.0.1:6379> SADD users:premium "Alice" "David"
(integer) 2
127.0.0.1:6379> SINTER users:online users:premium
1) "Alice"
```

Muestra todos los valores del conjunto

Si contiene la clave devuelve 1 y 0 en caso contrario

Obtenemos la intersección de dos conjuntos

PYTHON

```
r.sadd(name, *values)
r.srem(name, *values)
r.smembers(name)
r.sismember(name, value)
r.scard (name)
r.sinter(keys, *args)
r.sunion(keys, *args)
```

CONJUNTOS ORDENADOS (SORTED SETS)

Son similares a los conjuntos, pero cada miembro único tiene un **score** (puntuación) asociado, que es un valor numérico de coma flotante.

Los elementos se mantienen ordenados por su score. Si varios miembros tienen el mismo score, se ordenan lexicográficamente.

Su uso más común es para:

- **Tablas de clasificación (leaderboards):** rankings de juegos, puntuaciones.
- **Clasificación de elementos**: artículos más populares por votos, usuarios más activos.
- **Series temporales simples**: eventos con un *timestamp* como score.

Se puede trabajar con conjuntos ordenados con los siguientes comandos:

- **ZADD** ***key score member [score member …]:*** añade uno o más miembros a un conjunto ordenado con su score.
- **ZRANGE** ***key start stop [*****WITHSCORES*****]:*** obtiene un rango de miembros por índice (ordenado por score).
- **ZREVRANGE** ***key start stop [*****WITHSCORES*****]:*** obtiene un rango de miembros en orden descendente.
- **ZRANGEBYSCORE** ***key min max [*****WITHSCORES*****]:*** obtiene miembros dentro de un rango de scores.
- **ZREM** ***key member [member …]:*** elimina uno o más miembros.
- **ZSCORE** ***key member:*** obtiene el score de un miembro.
- **ZCARD** ***key:*** obtiene el número de miembros.

```
127.0.0.1:6379> ZADD leaderboard 100 "Alice" 85 "Bob" 120 "Charlie"
(integer) 3
127.0.0.1:6379> ZRANGE leaderboard 0 -1 WITHSCORES
1) "Bob"
2) "85"
3) "Alice"
4) "100"
5) "Charlie"
6) "120"
127.0.0.1:6379> ZREVRANGE leaderboard 0 -1 WITHSCORES
1) "Charlie"
2) "120"
3) "Alice"
4) "100"
5) "Bob"
6) "85"
127.0.0.1:6379> ZSCORE leaderboard "Alice"
"100"
```

Obtenemos todas las claves ordenadas por puntuación

Ahora ordenado de mayor a menor

Puntuación asociada a un valor

PYTHON

```
r.zadd(name, mapping, nx=False, xx=False, ch=False, incr=False)

r.zrange(name, start, end, desc=False, withscores=False, score_cast_func=Float)

r.zrevrange(name, start, end, withscores=False, score_cast_func=float)

r.zrangebyscore (name, min, max, start=none, num=none, withscores=False, score_cast_func=float)

r.zrem (name, *values)

r.zscore(name, value)

r.zcard(name)
```

HASHES (HASH MAPS)

Un hash es un **mapa de campos a valores**.

Cada hash puede almacenar hasta $2^{32} - 1$ pares campo-valor.

Son perfectos para representar objetos o registros de datos donde cada campo tiene un nombre y un valor asociado.

Los hashes se suelen utilizar para:

- **Almacenamiento de objetos**: perfiles de usuario, detalles de productos.
- **Caché de objetos**: almacenar objetos complejos de forma estructurada.
- **Configuraciones**: guardar configuraciones de aplicaciones o usuarios.

Redis dispone de los siguientes comandos para trabajar con hashes:

- **HSET** ***key field value [field value …]:*** establece el valor de uno o más campos en un hash.
- **HGET** ***key field:*** obtiene el valor de un campo específico.
- **HGETALL** ***key:*** obtiene todos los campos y valores de un hash.
- **HDEL** ***key field [field …]:*** elimina uno o más campos de un hash.
- **HKEYS** ***key:*** obtiene todos los nombres de campo en un hash.
- **HVALS** ***key:*** obtiene todos los valores en un hash.
- **HLEN** ***key:*** obtiene el número de campos en un hash

PYTHON

```
r.hset(name, key=none, value=none, mapping=none)

r.hget (name, key)

r.hgetall (name)

r.hdel (name, *keys)

r.hkeys (name)

r.hvals (name)

r.hlen(name)
```

STREAMS

Son un tipo de dato de **solo anexar** (*append-only*) que representa un log de eventos.

Cada entrada en un *stream* tiene un ID único y un conjunto de pares campo-valor.

Los *streams* están diseñados para sistemas de procesamiento de eventos y colas de mensajes persistentes.

Algunos casos de uso comunes son:

- **Logs de eventos**: almacenar un historial inmutable de eventos.
- **Sistemas de mensajería**: colas de mensajes persistentes con procesamiento distribuido (grupos de consumidores).
- **Procesamiento de datos en tiempo real**: ingesta y análisis de datos de sensores o IoT

Los principales comandos para trabajar con streams son:

- **XADD** ***key*** **ID** ***field value [field value …]:*** añade una nueva entrada al stream. El ID puede ser * para generar uno automáticamente.
- **XRANGE** ***key start end:*** lee un rango de entradas de un stream.
- **XREAD** ***[*****COUNT** ***count] [*****BLOCK** ***milliseconds]*** **STREAMS** ***key [key …]*** **ID** ***[*****ID** ***…]:*** lee entradas de uno o más streams, opcionalmente bloqueando hasta que haya nuevas entradas.
- **XGROUP CREATE** ***key groupname*** **ID** ***[*****MKSTREAM*****]:*** crea un grupo de consumidores para un stream.
- **XREADGROUP GROUP** ***groupname consumername [*****COUNT** ***count] [*****BLOCK** ***milliseconds]*** **STREAMS** ***key [key …]*** **ID*****:*** lee entradas de un stream usando un grupo de consumidores.

PYTHON

```
r.xadd(name, fields, id='*', maxlen=None,
approximate=True, nomkstream=False)

r.xrange (name, min='-', max='+', count=None)

r.xread (streams, count=None, block=None)

r.xgroup_create (name, groupname, id='$', mkstream=False)

r.xgroup_destroy (name, groupname)

r.xgroup_delconsumer (name, groupname, consumername)

r.xreadgroup(groupname, consumername, streams, count=None,
block=None, noack=False)
```

BITMAPS

Son un tipo de dato que permite tratar las cadenas de Redis como un **array de bits**.

Puedes establecer o borrar bits individuales en un offset específico.

Son extremadamente eficientes para almacenar información booleana a gran escala.

Los bitmaps se usan principalmente para:

- **Presencia de usuarios**: saber qué usuarios están online (cada offset es un ID de usuario).
- **Flags de características**: marcar si un usuario ha usado una característica específica.
- **Análisis de cohortes**: identificar usuarios que realizaron una acción en un día específico.
- **Votaciones binarias**: registrar votos "sí/no".

Podemos trabajar con bitmaps con los siguientes comandos:

- **SETBIT** ***key offset value:*** establece o borra el bit en un offset dado.
- **GETBIT** ***key offset***: obtiene el valor del bit en un offset dado.
- **BITCOUNT** ***key [start end]:*** cuenta el número de bits establecidos (a 1) en una cadena.
- **BITPOS** ***key bit [start] [end]:*** encuentra la posición del primer bit establecido o no establecido

PYTHON

```
r.setbit(name, offset, value)

r.getgit (name, offset)

r.bitcount (name, start=None, end=None)

r.bitpos (name, bit, start=None, end=None)
```

HYPERLOGLOG

Es un algoritmo **probabilístico** para estimar la cardinalidad de un conjunto (el número de elementos únicos) con un uso de memoria muy bajo.

Es ideal cuando necesitas **contar elementos únicos** en grandes volúmenes de datos y estás dispuesto a aceptar un **pequeño margen de error**.

Se usa habitualmente para:

- **Contadores de visitantes únicos**: en sitios web o aplicaciones.
- **Estimación de elementos únicos**: IPs únicas, IDs de productos vistos.
- **Análisis de tendencias**: ver la cardinalidad de eventos a lo largo del tiempo.

Dispone de los siguientes comandos:

- **PFADD** ***key element [element …]:*** añade elementos a un *HyperLogLog*.
- **PFCOUNT** ***key [key …]:*** estima la cardinalidad de uno o más *HyperLogLogs*.
- **PFMERGE** ***destkey sourcekey [sourcekey …]:*** combina múltiples *HyperLogLogs* en uno.

PYTHON

```
r.pfadd(name, *values)

r.pdcount (*sources)

r.pfmerge (dest, *sources)
```

GEO-SPATIAL (Datos Geoespaciales)

Permite almacenar **coordenadas de latitud y longitud** y realizar **consultas basadas en la distancia**.

Utiliza conjuntos ordenados internamente para almacenar los puntos geoespaciales, lo que permite operaciones eficientes de búsqueda por radio o por caja.

Se usa habitualmente para:

- **Contadores de visitantes únicos**: en sitios web o aplicaciones.
- **Estimación de elementos únicos**: IPs únicas, IDs de productos vistos.
- **Análisis de tendencias**: ver la cardinalidad de eventos a lo largo del tiempo.

Los comandos de que disponemos son:

- **GEOADD** ***key longitude latitude member [longitude latitude member …]:*** añade uno o más puntos geoespaciales.
- **GEODIST** ***key member1 member2 [unit]:*** calcula la distancia entre dos miembros.
- **GEORADIUS** ***key longitude latitude radius unit [*****WITHCOORD*****] [*****WITHDIST*****] [*****WITHHASH*****] [*****COUNT** ***count] [*****ASC|DESC*****]:*** encuentra miembros dentro de un radio dado de un punto central.
- **GEORADIUSBYMEMBER** ***key member radius unit [*****WITHCOORD*****] [*****WITHDIST*****] [*****WITHHASH*****] [*****COUNT** ***count] [*****ASC|DESC*****]:*** similar a GEORADIUS, pero centrado en un miembro existente.
- **GEOHASH** ***key member [member …]:*** devuelve el geohash de uno o más miembros.

PYTHON

```
r.geoadd(name, mapping, nx=False, xx=False, ch=False)

r.geodist (name, member1, member2, unit='m')

r.georadius (streams, longitude, latitude, radius,
unit='m', withdist=False, withcoord=False, withhash=False,
count=None, store=None, store_dist=None)

r.georadiusbymember(name, member, radius, unit='m',
withdist=False, withcoord=False, withhash=False,
count=None, sort=None, store=None, store_dist=None)

r.geohash (name, *members)
```

3

BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

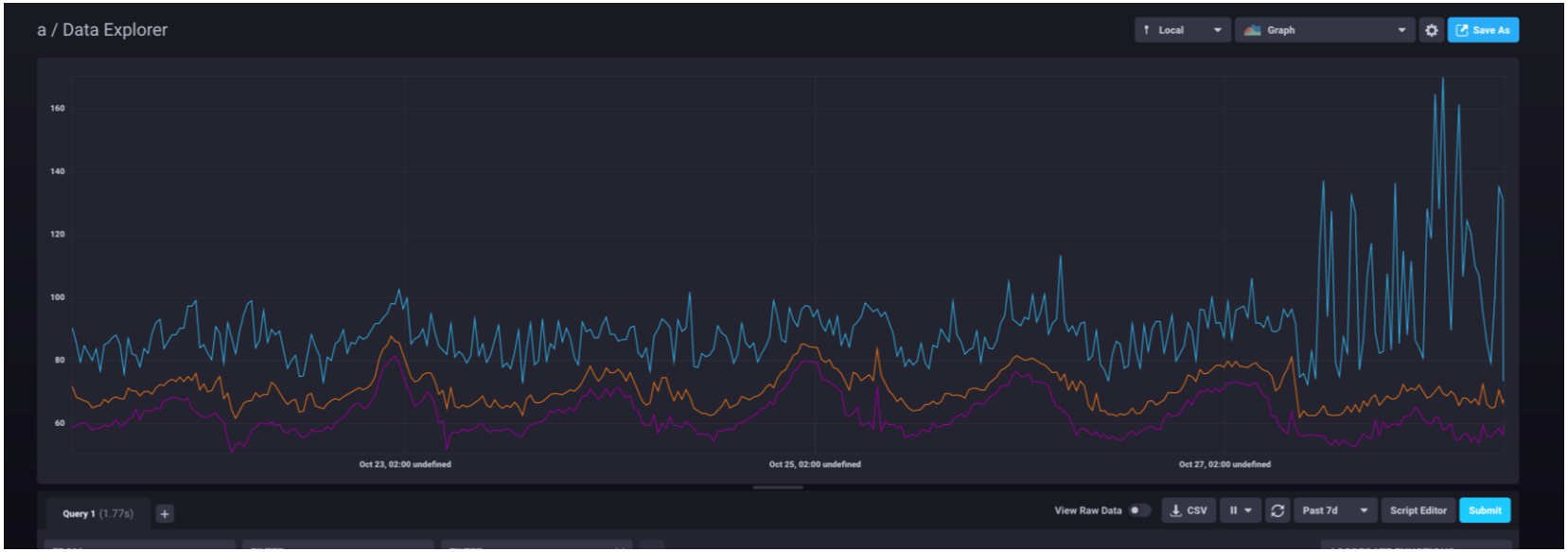
3 BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

3.1 FUNDAMENTOS DE INFLUXDB

BASES DE DATOS DE SERIES TEMPORALES (TSDB)

Una base de datos de series temporales (TSBD) es una base de datos **optimizada** para manejar datos que tienen una **marca de tiempo (timestamp)**

Lo importante de estas bases de datos es **analizar tendencias** en lugar de únicamente guardar el estado actual.



¿POR QUÉ NO USAR MYSQL O MONGODB CON TIMESTAMPS?

Aunque otros sistemas de bases de datos tienen un tipo de datos que permite guardar fecha y hora, las TSDB tienen algunas ventajas al manejar este tipo de datos:

Rendimiento de ingesta (escritura)

Las TSDB están diseñadas para ingerir millones de puntos de datos por segundo (por ejemplo, de miles de sensores IoT enviando datos constantemente)

MySQL/Mongo se ahogarían, ya que sus índices no están pensados para este patrón de “solo-añadir”

Compresión eficiente

Los datos de series temporales son muy repetitivos. Por ejemplo, la temperatura no cambia de 20º a 80º en un segundo.

InfluxDB usa algoritmos de compresión específicos que pueden reducir el tamaño en disco hasta un 90%

Consultas de tiempo (lectura)

InfluxDB tiene funciones nativas para responder preguntas como:

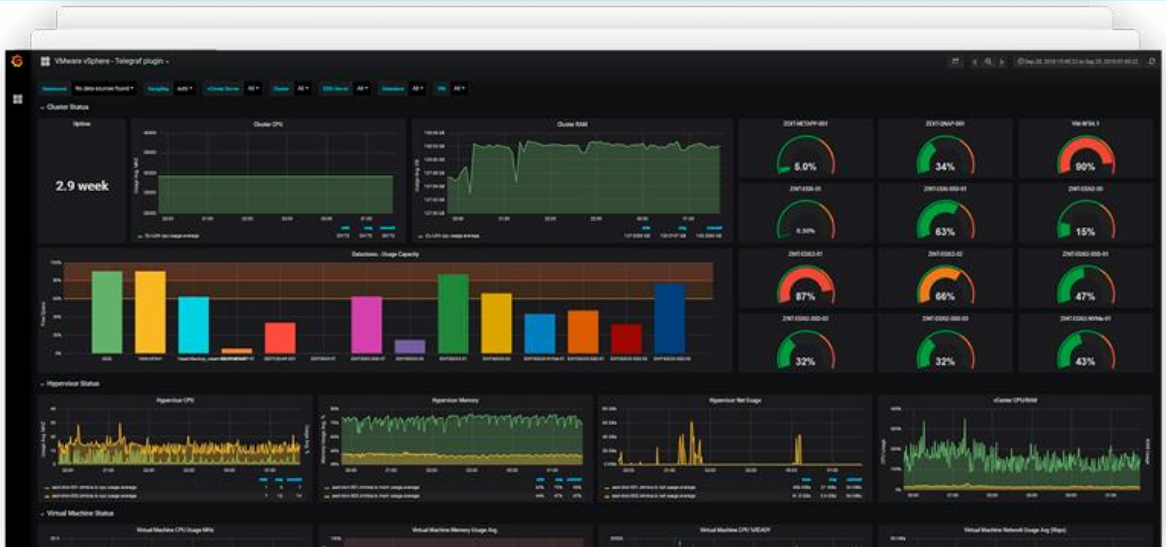
- "*Dame la temperatura media en ventanas de 5 minutos durante las últimas 24 horas*"
- "*Muéstrame el percentil 99 de uso de CPU*"

Esto es posible en MySQL, pero es lento y las consultas muy complejas

CASOS DE USO DE INFLUXDB

Algunos usos habituales de InfluxDB son:

- **Monitorización (DevOps)**: métricas de servidores, aplicaciones (CPU, RAM, peticiones por segundo, ..)
- **IoT (Internet of Things)**: millones de sensores enviando datos (temperatura, humedad, ubicación, energía)
- **Analítica en tiempo real (Fintech):** precios de acciones, volumen de transacciones, detección de fraude.



EVOLUCIÓN DE INFLUXDB

InfluxDB ha evolucionado desde su aparición, con 3 versiones principales:

InfluxDB 1.x

Era una pila de componentes separados (el TICK stack)

T	Telegraf (agente de Ingesta)
I	InfluxDB (la base de datos
C	Chronograf (la interfaz gráfica)
K	Kapacitor (el motor de alertas y tareas

Su lenguaje de consulta era InfluxQL (muy parecido a SQL)

InfluxDB 2.x

Se ha **unificado** la base de datos, la interfaz (Chronograf) y el motor de alertas/tareas (Kapacitor) en el mismo binario. Es mucho más simple.

Se introduce un nuevo lenguaje llamado **Flux,** un lenguaje funcional mucho más potente que InfluxQL, que permite no solo consultar, sino también analizar, transformar y actuar sobre los datos.

InfluxDB 3.x

Es la versión más reciente

Reescrito en **Rust** y basado en **Apache Arrow** y **Apache Parquet**

Soporta SQL de forma nativa además de Flux.

Está diseñado para una escala aún mayor (petabytes).

CÓMO USAR INFLUXDB

Hay que tener en cuenta algunas cosas a la hora de utilizar InfluxDB:

Qué SI hacer

- Datos inmutables que "solo se añaden" (append-only)
- Altas tasas de escritura (miles de puntos por segundo
- Consultas de agregación sobre rangos de tiempo
- Borrado de datos viejos (políticas de retención automáticas)

Qué NO hacer

- Nunca usarlo como base de datos de negocio (facturas, …)
- Nunca hacer UPDATE de puntos individuales
- Nunca hacer SELECT * sin un rango de tiempo

3 BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

3.2 CONCEPTOS BÁSICOS DE INFLUXDB

PUNTO

El **punto** es la **unidad fundamental** de almacenamiento en InfluxDB

Un punto es la representación de un dato en un momento específico en el tiempo.

Cada punto consta de cuatro componentes esenciales que lo definen:

- Measurement (medición)
- Tags (etiquetas)
- Fields (campos)
- Timestamp (marca de tiempo

Measurement (medición)

Es el **contenedor lógico** de la información, el equivalente más cercano a una tabla en el mundo SQL.

Agrupa datos que tienen la misma estructura y que miden el mismo tipo de fenómeno.

Ejemplo: `cpu_usage`, `weather_metrics`, `stock_prices`

Tags (etiquetas)

Son **metadatos** que describen el punto.

Son **indexados** y se utilizan para las mayorías de las operaciones de filtrado (WHERE) y agrupación (GROUP BY) en las consultas.

Deben usarse para **información de baja a media cardinalidad** (pocos valores únicos), como nombres de servidores, países o tipos de sensores.

Ejemplo: `host=server1`, `location=madrid`, `sensor_type=thermometer`

Fields (campos)

Son los valores reales que se están midiendo.

Son los datos numéricos o de cadena de texto que varían con el tiempo.

Los *fields* **no están indexados** como los *tags*.

Ejemplo: `temperatura=25.5`, `pressure=1012`, `error_count=5`

Timestamp (marca de tiempo)

El **índice primario** de la base de datos1

Es un valor de tiempo de alta precisión (hasta nanosegundos) que indica cuándo ocurrió la medición.

Diferencia clave: si quieres filtrar o agrupar datos (dame la temperatura media para Madrid), el valor debe ser un *tag*. Si solo quieres calcular promedios o sumas (calcula el promedio de temperatura) debe ser un *field*.

Además del punto, en InfluxDB hay otros conceptos clave:

ORGANIZACIÓN (Organization/Org)

Es el nivel más alto de la jerarquía.

Representa al usuario, la empresa o el proyecto que posee los recursos.

Su función es proporcionar un **espacio de nombres** para aislar usuarios y datos.

BUCKET (Cubo)

Es el **contenedor de datos principal**, equivalente a una base de datos o *keyspace*.

Un **bucket** agrupa **Measurements** con el mismo propósito y la misma **Política de Retención** (Retention Policy)

Los *buckets* se configuran con un **Time to Live (TTL)**, que indica por cuánto tiempo se deben guardar los datos. InfluxDB elimina automáticamente los datos más antiguos que el TTL para ahorrar espacio

TOKEN DE API (Autenticación)

InfluxDB utiliza un **sistema de tokens** (en lugar de usuario/contraseña) para **autenticar** las aplicaciones.

Cada token está asociado a una organización y tiene permisos granulares (leer/escribir) sobre *buckets* específicos.

LINE PROTOCOL

El **Line Protocol** es el protocolo de texto que utiliza InfluxDB para recibir e **ingerir (escribir)** datos de series temporales de manera eficiente.

Es un protocolo sencillo, basado en el envío de líneas de texto donde cada línea representa un único **punto** (medición en un momento dado) y debe contener los cuatro componentes esenciales (*measurement, tags, fields y timestamp*) en un orden y formato estricto

Una línea completa de Line Protocol tiene la siguiente estructura:

```
Measurement,TagSet FieldSet Timestamp
```

Measurement:

- Siempre es el primer elemento
- Se separa del resto de campos mediante una coma.
- Ejemplo: `cpu_usage`

TagSet:

- Múltiples tags separados por comas.
- Se separa del FieldSet por un espacio
- Ejemplo: `location=madrid,host=server1`

FieldSet:

- Múltiples *fields* separados por comas.
- Ejemplo: `idle=80.1,user=15.5`

Timestamp:

- Opcional, pero muy recomendado
- Debe ser un valor entero (nanosegundos por defecto)
- Ejemplo: `1678886400000000000`

Propiedades clave de Line Protocol

- **Ingesta de alta velocidad**: es deliberadamente simple y basado en texto. Esto minimiza la sobrecarga de procesamiento para el servidor, permitiendo una **ingesta extremadamente rápida** de millones de puntos por segundo.
- **El timestamp es opcional** (pero esencial)
  - Si se incluye se guarda el dato con ese valor de tiempo exacto
  - Si se omite, Influxdb le asigna el valor del reloj del servidor en el momento en que recibe el dato. Esto se llama **tiempo de ingesta** y solo debe usarse si no se puede obtener el tiempo en el dispositivo de origen.
- **Tipos de datos**: el tipo de dato de un *Field* se infiere automáticamente:
  - `10i`: entero (si no se pone la i se infiere como flotante)
  - `10.5`: flotante
  - `"online"`: cadena
  - `t o true`: booleano

- **Separadores y espacios**: la estructura es muy sensible a los espacios. El *TagSet* y el *FieldSet* deben estar separados **exactamente** por un espacio. Además, no puede haber espacios dentro de los *TagSet* y *FieldSet*.

FLUX (Lenguaje de consulta)

Flux es el **lenguaje de consulta** y **scripting** (automatización) funcional y de flujo de datos de InfluxDB (2.x/3.x)

A diferencia de SQL, que es declarativo, Flux es **funcional** y utiliza el operador **pipe-forward(|>)** para encadenar operaciones y funciones.

```
from(bucket: "example-bucket")
    |> range(start: -1h)
    |> filter(fn: (r) => r._measurement == "example-measurement" and r.tag == "example-tag")
    |> filter(fn: (r) => r._field == "example-field")
```

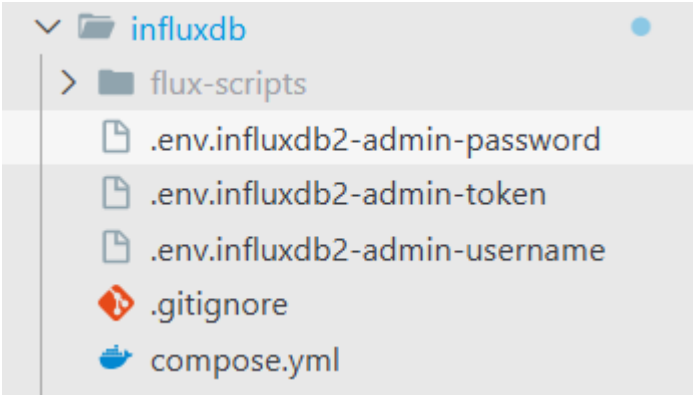
3 BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

3.3 INSTALACIÓN Y CONEXIÓN

Instalaremos InfluxDB en Docker utilizando el fichero compose.yml disponible en la web indicada por el profesor.

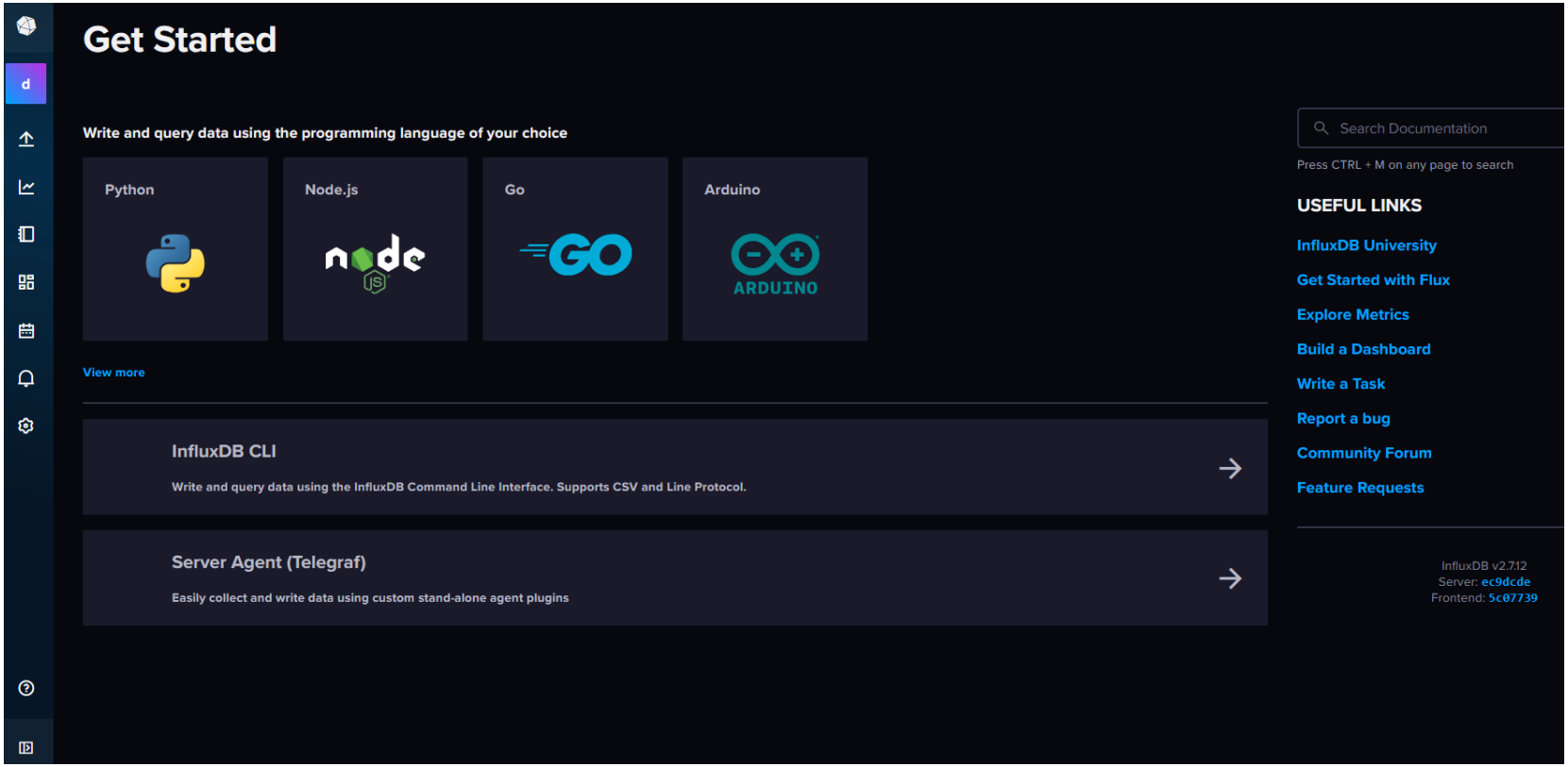
●	influxdb	-	-	-	0.06%	47 minutes ago
●	influxdb2-1	96cae0d9015d	influxdb:2	8086:8086	0.06%	47 minutes ago

Como poder ver, mapea el puerto **8086**, donde podremos acceder a la interfaz gráfica.



Otra cosa a tener en cuenta es que las variables de entorno (nombre de usuario y contraseña) están definidas en ficheros externos.

Una vez instalado e iniciada la sesión veremos algo similar a la siguiente interfaz



El primer paso será instalar la librería influxdb-client que nos permitirá operar sobre InfluxDB desde Python

```
[1]: !pip install influxdb-client

Collecting influxdb-client
  Downloading influxdb_client-1.49.0-py3-none-any.whl.metadata (65 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 65.5/65.5 kB 2.2 MB/s eta 0:00:00
Collecting reactivex>=4.0.4 (from influxdb-client)
  Downloading reactivex-4.1.0-py3-none-any.whl.metadata (5.7 kB)
```

Código para conectarnos a InfluxDB

```python
import influxdb_client
from influxdb_client.client.write_api import SYNCHRONOUS
from influxdb_client.client.exceptions import InfluxDBError
from urllib3.exceptions import NewConnectionError

INFLUX_URL = "http://influxdb2:8086"
INFLUX_TOKEN = "MyInitialAdminToken0=="

print("--- Iniciando conexión a InfluxDB ---")

client = None
try:
    # 1. Inicializar el cliente
    client = influxdb_client.InfluxDBClient(
        url=INFLUX_URL,
        token=INFLUX_TOKEN,
        org="dev_pruebas"
    )

    # 2. Verificar la conexión con el servidor
    print(f"Verificando estado de salud de InfluxDB en {INFLUX_URL}...")
    health = client.health()

    if health.status == "pass":
        print("[INFO] ¡Conexión exitosa!")
        print(f"[INFO]  Versión del servidor: {health.version}")
    else:
        print(f"[ERROR] Conexión fallida. Estado: {health.status}")
        print(f"[INFO] Mensaje: {health.message}")

except (InfluxDBError, NewConnectionError) as e:
    print("[ERROR] Error al conectar con InfluxDB:")
    print(f"   Detalle: {e}")

finally:
    # 3. Es buena práctica cerrar siempre el cliente
    if client:
        client.close()
        print("--- Conexión cerrada ---")
```

Importamos la librería del cliente de InfluxDB y una constante y dos clases para referenciarlas de forma más corta en el código

```python
import influxdb_client
from influxdb_client.client.write_api import SYNCHRONOUS
from influxdb_client.client.exceptions import InfluxDBError
from urllib3.exceptions import NewConnectionError

INFLUX_URL = "http://influxdb2:8086"
INFLUX_TOKEN = "MyInitialAdminToken0=="

print("--- Iniciando conexión a InfluxDB ---")

client = None
try:
    # 1. Inicializar el cliente
    client = influxdb_client.InfluxDBClient(
        url=INFLUX_URL,
        token=INFLUX_TOKEN,
        org="dev_pruebas"
    )
```

Como Jupyter está en un contenedor diferente del de InfluxDB utilizo la resolución local de Docker para indicar la URL

Intento conectarme al cliente

Usamos el comando `health()` para comprobar el estado operativo y de salud del servidor

```
    # 2. Verificar la conexión con el servidor
    print(f"Verificando estado de salud de InfluxDB en {INFLUX_URL}...")
    health = client.health()

    if health.status == "pass":
        print("[INFO] ¡Conexión exitosa!")
        print(f"[INFO]  Versión del servidor: {health.version}")
    else:
        print(f"[ERROR] Conexión fallida. Estado: {health.status}")
        print(f"[INFO] Mensaje: {health.message}")

except (InfluxDBError, NewConnectionError) as e:
    print("[ERROR] Error al conectar con InfluxDB:")
    print(f"   Detalle: {e}")

finally:
    # 3. Es buena práctica cerrar siempre el cliente
    if client:
        client.close()
        print("--- Conexión cerrada ---")
```

El campo `status` me da información sobre el estado del servidor

En este caso gestiono si hay algún error grave al conectarse con el servidor

3 BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

3.3 ESCRITURA DE DATOS EN INFLUXDB

Para escribir datos en InfluxDB tenemos que utilizar la **API de escritura (Write API)** que es el componente que se encarga de enviar los datos desde la aplicación Python al servidor de InfluxDB.

Se puede definir como un **canal de comunicación optimizado** para la ingesta de datos.

Hay que distinguir entre el cliente en sí mismo y la Write API, que es una capa que se encarga de la logística de la escritura:

- El **cliente** (`InfluxDBClient`) es la conexión general que gestiona el token, la URL, …
- La **WriteAPI** (`client.write_api(…)`) es el método específico para enviar datos de la manera más eficiente y segura posible.

Su función más importante es gestionar la **escritura en lote (batching)**.

Cuando le das un objeto Point para escribir, la Write API no necesariamente lo envía de inmediato:

Si está configurado en modo **asíncrono** (opción por defecto y recomendada), la API:

- Almacena los puntos en un **buffer interno** (una cola)
- Espera hasta que el buffer se llene (p.e. 5000 puntos) o hasta que pase un tiempo definido (p.e. 1 seg.)
- Envía todo el lote al servidor en una sola solicitud HTTP

Para pruebas se puede configurar en modo **síncrono** donde cada punto se envía inmediatamente y espera la confirmación antes de continuar.

```
# Para producción masiva se suele usar batching (asíncrono)
write_api = client.write_api(write_options=SYNCHRONOUS)
```

Una vez que hemos definido la Write API ya podemos escribir en la base de datos, para ello, haremos uso de **la clase Point**.

El parámetro que recibe el constructor de esta clase es el *Measurement*, y luego podremos utilizar el encadenamiento para agregar *Tags*, *Fields* y *Timestamp.*

```
# Creamos un punto
p = Point("temperatura_medicion") \
    .tag("location", "oficina") \
    .field("value", 23.5) \
    .time(None)  # Usará el tiempo de ingesta

# Lo escribimos
write_api.write(bucket=bucket, org=org, record=p)
```

Si queremos incluir el tiempo debemos utilizar la librería datetime.

```python
import datetime

# Usar el tiempo actual en formato ISO 8601 (preferido)
tiempo_actual = datetime.datetime.utcnow().isoformat() + "Z"

punto = (
    Point("datos_iot")
    .tag("ubicacion", "bodega_sur")
    .tag("sensor_id", "S003")
    .field("humedad_relativa", 75.8)
    .field("nivel_bateria", 95)
    .field("activo", True)
    .time(tiempo_actual)
)
```

3 BASES DE DATOS SERIES DE TIEMPO: INFLUXDB

3.3 EL LENGUAJE DE CONSULTA FLUX

Flux es un lenguaje **funcional** y **orientado a flujos de datos** diseñado para consultar, transformar y manipular series temporales.

Flux trabaja sobre **streams**, es decir, **secuencias de tablas** que pasan por distintas etapas del pipeline:

- Cada función recibe un flujo de tablas
- Realiza una transformación
- Devuelve un nuevo flujo de tablas

Genera un flujo de tablas

Recorta esas tablas a la última hora

Elimina filas que no cumplan la condición

```
from(bucket: "sensores")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "temperatura")
  |> mean()
```

Calcula medias por grupo

FUNCIONES BÁSICAS

Todas las consultas en Flux comienzan obteniendo datos de bucket con **from()** y luego pasan por varias transformaciones usando el operador |>

`from()`

La función **from()** recibe como parámetro el *bucket* con los datos y devuelve un **stream de tablas**, una por cada combinación de {measurement, tag1, tag2, …}

range()

La función **range()** es obligatoria e indica el rango de tiempo que va a devolver la consulta. Tiene los parámetros:

- **`start`**: tiempo de inicio del rango. Se puede indicar de dos maneras diferentes:
  - `-<duración>`: donde duración es un número y una letra indicando la unidad de medida: segundos (s), minutos (m), horas (h), días (d), semanas(w), meses (mo) o años (y)
  - `YYYY-MM-DDTHH:MM:SSZ`: valor absoluto en formato RFC3339
- **`stop`**: las opciones pueden ser las mismas que en start, o bien, `now()`, que es el valor por defecto si se omite

Veamos cómo sería una consulta básica.

```python
from influxdb_client import InfluxDBClient
from influxdb_client.client.exceptions import InfluxDBError

INFLUX_URL = "http://influxdb2:8086"
INFLUX_TOKEN = "MyInitialAdminToken0=="
INFLUX_ORG = "docs"
INFLUX_BUCKET = "weather_data"

# Inicializar el cliente y la API de consulta
client = InfluxDBClient(url=INFLUX_URL, token=INFLUX_TOKEN, org=INFLUX_ORG)
query_api = client.query_api()
```

Preparamos el cliente, igual que al cargar los datos

Necesitamos un objeto query_api() para lanzar las consultas contra la base de datos

Ejecución de la consulta

La consulta Flux la almacenamos en una cadena.
Utilizo el formato multilínea para que quede más clara

```
query_ultimo_dia = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1d)
"""

# Ejecutar y procesar la respuesta
result = query_api.query(org=INFLUX_ORG, query=query_ultimo_dia)
```

Limito el tiempo a las últimas 24 horas

Por último, lanzo la consulta utilizando la función query() de query_api

Influx devuelve los datos en una serie de tablas (de tipo **FluxTable**), por lo que debemos iterar sobre ellas para acceder a la respuesta

```python
# El resultado es una lista de objetos FluxTable
for table in result:
    # Cada tabla contiene una serie de registros
    for record in table.records:
        # Imprimir la hora, la estación (Tag) y el valor (Field)
        print(f"[{record.get_time().strftime('%H:%M:%S')}] Estación: {record['city']} \
                | Campo: {record['_field']} = {record['_value']}")
```

Y cada tabla tendrá una serie de registros

Para entender el resultado de las consultas Flux debemos ver la **estructura de cada registro de salida**

```
{'result': '_result',
 'table': 14,
 '_start': datetime.datetime(2025, 11, 23, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_stop': datetime.datetime(2025, 11, 24, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_time': datetime.datetime(2025, 11, 24, 11, 0, tzinfo=datetime.timezone.utc),
 '_value': 16.6,
 '_field': 'wind_speed_kph',
 '_measurement': 'weather_report',
 'city': 'Sevilla',
 'station_id': 'Estacion_03_SEV'}
```

Si observas, verá que no se parece en nada a cada una de las líneas del fichero CSV que hemos cargado en la base de datos.

Eso se debe a la forma en la que Influx carga los datos internamente.

Veamos qué es cada uno de los registros.

El valor de este campo casi siempre es _result

```
{'result': '_result',
 'table': 14,
 '_start': datetime.datetime(2025, 11, 23, 11, 54,
 '_stop': datetime.datetime(2025, 11, 24, 11, 54,
 '_time': datetime.datetime(2025, 11, 24, 11, 0, t
 '_value': 16.6,
 '_field': 'wind_speed_kph',
 '_measurement': 'weather_report',
 'city': 'Sevilla',
 'station_id': 'Estacion_03_SEV'}
```

Cuando ejecutamos una consulta, el motor de procesamiento no devuelve una única tabla, sino un **conjunto de tablas pequeñas**. Este campo identifica de forma única la clave de agrupación actual del registro

Todas las consultas tienen que estar acotadas en un rango de tiempo. **_start** indica el inicio de este rango

```
{'result': '_result',
 'table': 14,
 '_start': datetime.datetime(2025, 11, 23, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_stop': datetime.datetime(2025, 11, 24, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_time': datetime.datetime(2025, 11, 24, 11, 0, tzinfo=datetime.timezone.utc),
 '_value': 16.6,
 '_field': 'wind_speed_kph',
 '_measurement': 'weather_report',
 'city': 'Sevilla',
 'station_id': 'Estacion_03_SEV'}
```

De forma similar, también se almacena el momento final del rango.

Y **_time** almacena el momento que corresponde al punto

```
{'result': '_result',
 'table': 14,
 '_start': datetime.datetime(2025, 11, 23, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_stop': datetime.datetime(2025, 11, 24, 11, 54, 9, 820410, tzinfo=datetime.timezone.utc),
 '_time': datetime.datetime(2025, 11, 24, 11, 0, tzinfo=datetime.timezone.utc),
 '_value': 16.6,
 '_field': 'wind_speed_kph',
 '_measurement': 'weather_report',
 'city': 'Sevilla',
 'station_id': 'Estacion_03_SEV'}
```

Habrá un campo por cada uno de los **tags** que tengan los datos. Los **tags** están en todas las tablas devuelta.

Pero solo habrá un **field** en cada tabla devuelta, el nombre del field estará en **_field** y el valor en **_value**. Por ejemplo, en esta tabla tenemos el campo wind_speed_kph

¿Y por qué guarda así los datos? El motivo es por la **desnormalización**.

La **desnormalización** es uno de los cambios de mentalidad más grandes al pasar de una base de datos relacional (SQL) a una base de datos NoSQL, como InfluxDB o Cassandra.

Normalización

Consisten en dividir los datos en múltiples tablas relacionadas mediante **claves foráneas**

El objetivo es **ahorrar espacio** en disco y garantizar la **integridad** de los datos (evitar redundancia)

Para obtener los datos es necesario hacer **joins**

Desnormalización

Consiste en duplicar los datos en la misma tabla, de modo que cada consulta **se satisfaga con una sola lectura**.

El objetivo es **optimizar la velocidad de lectura** y la disponibilidad, asumiendo que el disco es barato y los JOIN son lentos o imposibles en un clúster distribuido.

En InfluxDB la desnormalización se aplica de dos maneras:

Tags y Fields (desnormalización por registro)

El Line Protocol obliga a la desnormalización en cada punto de datos ya que los metadatos (tags) y los valores (fields) se guardan en el mismo registro.

Ejemplo:

Supongamos que el sensor 12A está siempre en Madrid. En un modelo normalizado tendríamos una tabla `Sensores(ID, ciudad)` y otra tabla `Mediciones(id_sensor, temperatura)`.

En cambio, en un modelo desnormalizado se repetiría la información en cada punto de datos que envía el sensor.

`_time, station_id` (tag), `city` (tag), `temperatura_c` (field)

De esta forma, al leer el punto se tendría toda la información sin necesidad de hacer ningún JOIN.

Múltiples tablas por patrón de consulta (desnormalización por propósito)

Si tienes dos consultas que necesitan el mismo dato, pero ordenado o agrupado de forma diferente, en InfluxDB, la solución es crear dos Measurements o Buckets distintos.

Ejemplo: una aplicación web que tiene logs y métricas

Consulta A: métricas del dashboard, donde necesitar ver CPU por servidor.

Habría una tabla donde cada registro sería de la forma:

```
cpu_server, host_ID=SVR_a value=45
```

Consulta B: métricas de alerta, necesitas ver la CPU por aplicación

La tabla sería:

```
cpu_application,app_name=WebApp_X value=45
```

El valor de CPU está duplicado en ambas tablas, pero cada una está optimizada (o diseñada) para un tipo de consulta diferente.

filter()

El siguiente paso después de seleccionar la fuente y el rango de tiempo es filtrar los datos para quedarnos solo con los que necesitamos.

La función **filter()** recibe un argumento llamado **fn** (función) al que se le pasa una función anónima que se aplica a cada fila del conjunto de datos.

La función anónima debe devolver `true` para mantener la fila o `false` para descartarla.

Sintaxis:

```
|> filter(fn: (r) => <condición booleana>)
```

Donde r es la variable que representa la fila actual o registro.

Ejemplo 1: Filtrar por campo.

```
print("--- EJERCICIO 2: filtrar por campo ---")

my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1w)
    |> filter( fn: (r) => r._field == "temperature_c" )

"""
```

Tipo	Nombre
Tag	station_id
Tag	city
Field	**temperature_c**
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Aquí estamos seleccionando únicamente los registros que contienen la temperatura

Ejemplo 2: Filtrar por etiqueta.

```
print("--- EJERCICIO 3: filtrar por etiqueta ---")

my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1w)
    |> filter( fn: (r) => r.city == "Madrid" )

"""
```

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

En este caso estamos escogiendo un tag y queremos que tenga un valor determinado

Ejemplo 3: Filtrar con múltiples condiciones

Me quedo con los registros cuyo valor del *Tag* city es "Madrid"

Y que tengan el *Field* `temperature_c`

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

```
print("--- EJERCICIO 4: Filtrar con múltiples condiciones -

my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1w)
    |> filter( fn: (r) => r.city == "Madrid" and r._field == "temperature_
    |> filter( fn: (r) => r._value > 20.0 )

"""
```

Y además tengo otro filtro para tener solo los registros cuyo valor sea mayor que 20

map()

La función `map()` servirá para aplicar una **transformación** o cálculo a cada valor de una columna y **crear una nueva columna** o **modificar** una existente.

La sintaxis de map es:

```
|> filter(fn: (r) => ({ r with <nueva_columna>: <expresión>) )
```

Donde `r with` mantiene todas las columnas existentes en el nuevo registro y luego añades o sobreescribes columnas.

Ejemplo 1: Conversión de temperatura (C a F)

Las llaves tienen significado dentro de la f-string, por lo que hay que **escaparlas** para que no de error. Eso lo hacemos duplicando las llaves.

```
my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> map( fn: (r) => ( {{ r with
        temperature_f: r._value * 1.8 + 32.0
    }} ))

"""
```

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Creamos un campo nuevo que se llama `temperature_f`

Fórmula para calcular el nuevo campo

Ejemplo 2: Redondeo del valor

Voy a usar la librería math de Flux, por lo que debo importarla.

```
my_query = f"""
import "math"

from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> map( fn: (r) => ( {{ r with
        _value: math.round(x: r._value)
    }} ))

"""
```

Ahora no quiero crear un campo nuevo, sino modificar el existente cuyo valor está en **_value**

Utilizo la función math.round() de Flux para redondear el valor

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Ejemplo 3: Columna de estado basada en umbral

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Voy a usar la librería math de Flux, por lo que debo importarla.

```
my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> map( fn: (r) => ( {{ r with
        estado_temperatura: if r._value > 30.0 then "ALERTA" else "Normal"
    }} ))
"""
```

Creo un Tag nuevo llamado *estado_temperatura*

Si el valor de la temperatura es > 30 le doy el valor “Alerta” y, si no, “Normal”

rename()

La función `rename()` es muy simple y se utiliza para cambiar el nombre de una o varias columnas, lo cual es útil para preparar los datos para la visualización.

Su sintaxis es:

```
|> rename( columns: { <nombre_antiguo>:<nombre_nuevo>,.. })
```

Ejemplo 1: Renombrar columnas clave

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Voy a usar la librería math de Flux, por lo que debo importarla.

```
my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -1d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> rename( columns: {{
        station_id: "Estación", _value: "Temperatura Celsius"
        }} )
"""
```

Renombro un Tag

Y también puedo renombrar la columna del valor

```
--- Columna de estado basada en umbral) ---
FluxRecord() table: 0, {'result': '_result', 'table': 0, '_start': datetime.datetime(2025, 11, 23, 17,
31, 1, 639890, tzinfo=datetime.timezone.utc), '_stop': datetime.datetime(2025, 11, 24, 17, 31, 1, 63989
0, tzinfo=datetime.timezone.utc), '_time': datetime.datetime(2025, 11, 23, 18, 0, tzinfo=datetime.timez
one.utc), 'Temperatura Celsius': 4.06, '_field': 'temperature_c', '_measurement': 'weather_report', 'ci
ty': 'Leon', 'Estación': 'Estacion_01_LE'}
FluxRecord() table: 0, {'result': '_result', 'table': 0, '_start': datetime.datetime(2025, 11, 23, 17,
```

FUNCIONES DE AGREGACIÓN

Hasta ahora hemos visto funciones para **seleccionar** y **limpiar** datos.

Es momento de ver como **resumir y agregar** esos datos, que el dónde el valor de una base de datos de series temporales se hace evidente.

`aggregateWindow()`

Si calculamos el promedio de 30 días solo tenemos un número, lo importante es poder ver cómo evoluciona ese promedio a lo largo del tiempo.

La función `aggregateWindow()` divide el rango en intervalos uniformes (ventana) y luego aplica una función de agregación a los datos dentro de cada ventana.

```
|> aggregateWindow(every: <duración>, fn: <func>, createEmpty: bool)
                                                                )
```

Hasta ahora hemos visto funciones para **seleccionar** y **limpiar** datos.

Es momento de ver como **resumir y agregar** esos datos, que el dónde el valor de una base de datos de series temporales se hace evidente.

Su sintaxis es:

```
|> aggregateWindow(every: <duración>, fn: <func>, createEmpty: bool)
```

Donde:

- `every`: define el tamaño de la ventana
- `fn`: la función de agregación (p.e. `mean`, `max`, `sum`, `count`, `min`)
- `createEmpty`: si no hay datos en una ventana se salta (recomendado false)

Ejemplo 1: Temperatura media diaria

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

Me quedo con los registros que tienen la temperatura

Indicamos que calcule el máximo para cada día

```
my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -30d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> aggregateWindow(
        every: 1d,
        fn: max,
        createEmpty: false
        )
"""
```

```
FluxRecord() table: 0, {'result': '_result', 'table': 0, '_start': datetim
e.datetime(2025, 10, 26, 8, 13, 13, 99308, tzinfo=datetime.timezone.utc), '
_stop': datetime.datetime(2025, 11, 25, 8, 13, 13, 99308, tzinfo=datetime.t
imezone.utc), '_time': datetime.datetime(2025, 11, 2, 0, 0, tzinfo=datetim
e.timezone.utc), '_value': 11.35, '_field': 'temperature_c', '_measurement'
: 'weather_report', 'city': 'Leon', 'station_id': 'Estacion_01_LE'}
```

group()

Mientras que `aggregateWindow()` permite el agrupamiento por tiempo, la función `group()` permite el **agrupamiento por Tags** para consolidad los datos.

Esta función redefime el conjunto de columnas que definen a qué grupo pertenece cada fila.

Su sintaxis es:

```
|> group( [“tag1”, “tag2, … )
```

Ejemplo 1: Temperatura media por estación y ciudad

Tipo	Nombre
Tag	station_id
Tag	city
Field	temperature_c
Field	humidity_percent
Field	rainfall_mm
Field	wind_speed_kph
Field	uv_index

```
my_query = f"""
from(bucket: "{INFLUX_BUCKET}")
    |> range(start: -30d)
    |> filter( fn: (r) => r._field == "temperature_c" )
    |> group( columns: ["city", "station_id"] )
    |> mean()
"""
```

Esto creará una tabla por cada combinación de ciudad y estación

Para cada tabla calculamos la media

pivot()

La función `pivot()` nos sirve para revertir el almacenamiento columnar de InfluxDB para que los datos se parezcan más a lo que podemos ver en un CSV.

Su función es transformar las filas basadas en los nombres de los Fields en nuevas columnas, **uniendo todas las métricas de un solo punto de tiempo en una única fila**.

La sintaxis es:

```
|> pivot(
        rowKey: [“_time”, “city”],
        columnKey:[“_field”],
        valueColumn:”_value”
)
```