

Sistemas de Big data

Redis

Conectar Redis con Jupiter

Import redis

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

Redis es una base de datos clave-valor

Clave = Siempre es una cadena

Valor = Puede ser de diferentes tipos, depende de la información que queramos almacenar

Comandos (Recordemos todo es clave-valor)

Los ejemplos van a estar siempre en modo cuadernos jupiter

SET = Almacena una clave y su valor asociado

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

Opciones con SET

EX = Establece un tiempo de expiración en segundos

```
r.set("nombre", "Angel") EX 300
```

PX = EX pero en milisegundos

```
r.set("nombre", "Angel") PX 300
```

NX = Solo guarda la clave si no existe ya

```
r.set("nombre", "Angel") NX
```

XX = Solo guarda la clave si existe ya

```
r.set("nombre", "Angel") XX
```

GET = Devuelve los valores de las claves indicadas

```
r.get("nombre")
```

KEYS = Busca claves que coincidan con un patrón

Ejemplo clave

```
r.SET( "user:name" "Angel")
```

```
r.KEYS( user:*)
```

Resultado

```
"user:Angel"
```

EXISTS = Verifica si una clave existe

```
r.EXISTS("user:name")
```

DEL = Elimina una clave

```
r.DELETE("user:apellidos")
```

EXPIRE = Establece un TTL de una clave en segundos

GETRANGE = Devuelve el valor que se encuentra en ese rango

```
r.getRange("clave",int(inicio),int(final))
```

```
r.expire("clave",tiempo en segundos)
```

TTL = Te indica el ttl de esa clave(Si es numero positivo, te da los segundos que le quedan, si es -1, la clave existe pero no tiene caducidad y si es -2, la clave no existe)

```
r.TTL("user:nombre")
```

Type = Vemos el tipo de datos que almacena una clave

```
r.TYPE("user:name")
```

Flushdb() = Elimina todas las claves de la base de datos actual

```
r.flushdb()
```

flushall() = Elimina todas las claves de todas las bases de datos

COMANDO TRABAJAR CON **CADENAS**

SET

GET

DEL

INCR = Incrementa el valor numérico de una clave en 1

DECR = Decrementa el valor numérico de una clave en 1

EXPIRE

TTL

COMANDO TRABAJAR CON **LISTAS**

LPUSH = Añade uno o más valores al inicio de la lista

r.LPUSH("clave_pedidos","valor_pedido")

RPUSH = Añade uno o mas valores al final de la lista

r.RPUSH("clave_pedidos","valor_pedido")

LPOP = Elimina y devuelve el primer elemento de la lista

r.LPOP("Clave_de_la_lista")

RPOP = Elimina y devuelve el primer elemento de la lista

r.RPOP("Clave_de_la_lista")

LRANGE = Obtiene un rango de elementos de la lista

r.LRANGE("clave_lista",0,3)

Clave, valor por el que empieza el rango, valor por el que acaba

LLEN = Obtiene la longitud de la lista

r.LLEN("clave_de_la_lista")

BLPOP = Version de LPOP que es bloqueante, espera hasta que haya elementos

```
r.BLPOP("Clave_de_la_lista",timeout=0)
```

COMANDOS PARA CONJUNTOS (SETS)

Un conjunto es una colección desordenada de cadenas únicas

No mantienen un orden

SADD = Añade uno o más miembros a un conjunto

```
r.SADD("Clave_conjunto","valor1","valor2")
```

SREM = Elimina uno o más miembros de un conjunto

```
r.SREM("Clave_conjunto","valor1","valor2")
```

SMEMBERS = Obtiene todos los miembros del conjunto

```
r.smembers("Clave_conjunto")
```

SISMEMBER = Comprueba si un miembro existe en un conjunto

```
r.SISMEMBER("Clave_conjunto", "miembro")
```

SCARD = Obtiene el número de miembros en un conjunto

```
r.SCARD("Clave_conjunto")
```

SINTER = Compara Conjuntos y te dice los valores en común

```
r.SINTER("Clave_conjunto","Clave_conjunto2")
```

SUNION = Devuelve la unión de dos o más conjuntos

```
r.SUNION("Clave_conjunto","Clave_conjunto2")
```

COMANDOS PARA CONJUNTOS ORDENADOS (SORTED SETS)

SORTED SETS = SETS, pero cada miembro único tiene un SCORE asociado (Valor numérico en flotante)

Los elementos se ordenan según su score, Si varios miembros tienen el mismo score se ordena lexicográficamente

ZADD = Añade uno o más miembros a un conjunto ordenado con su score

```
r.ZADD("Clave_Sorted_SETS", {Clave_miembro:Valor_Miembro,  
Clave_miembro2:Valor_Miembro2})
```

ZRANGE = Obtiene un rango de miembros por índice (Ordenado por score)

```
r.ZRANGE("Clave_Sorted_SETS",0,1, WITHSCORES = True)  
  
clave_sorted_sets, Inicio,final y con valores
```

ZREVRANGE = ZRANGE pero de mayor a menor

```
r.REVRANGE("Clave_Sorted_SETS",0,1, WITHSCORES = True)  
  
clave_sorted_sets, final, inicio y con valores
```

ZRANGEBYSCORE = Obtiene miembros dentro de un rango de scores

```
r.ZRANGEBYSCORE("clave_SORTED_SETS", Valor inicial,Valor final,  
WITHSCORES = True)
```

ZREM = Elimina uno o más miembros

```
r.ZREM("Clave_Sorted_sets","Clave_miembro")
```

ZSCORE = Obtiene el score de un miembro

```
r.ZCORE("Clave_Sorted_sets","Clave_miembro")
```

ZCARD = Obtiene el número de miembros

```
r.ZCARD("Clave_SORTED_SETS")
```


COMANDOS PARA HASHES (HASH MAPS)

Un hash es un mapa de campos de valores

HSET = Establece un valor a un hash (solo un campo)

```
r.HSET("clave_hash",campo,valor)
```

HSET + MAPPING -> Sirve para poner varios campos a la vez

```
r.hset("hashes_usuario", mapping={  
    "nombre": nombre,  
    "email": email  
})  
  
r.hset("usuario:1001", "nombre", "Alice")  
r.hset("usuario:1001", "edad", 30)  
r.hset("usuario:1001", "email", "alice@example.com")
```

HGET = Obtiene el valor de un campo específico

```
r.HGET("Clave_hash",campo)
```

HGETALL = Obtener los valores de todos los campos

```
r.HGETALL("Clave_hash")
```

HDEL = Elimina uno o más campos de un hash

```
r.HDEL("Clave_hash",campo)
```

HKEYS = Obtiene todos los nombres de campo de un hash

```
r.HKEYS("Clave_hash")
```

HVALS = Obtiene todos los valores de un Hash

```
r.HVALS("Clave:hash")
```

HLEN = Obtiene el número de campos de un hash

```
r.HLEN("Clave_Hash")
```

COMANDOS PARA STREAMS

Un stream en Redis es una estructura de datos que almacena secuencias de mensajes de manera ordenada. Puedes imaginarlo como un registro de eventos o una cola de mensajes, donde cada mensaje tiene:

Un ID único (basado en timestamp y secuencia).

Un conjunto de campos y valores (similar a un diccionario o hash).

XADD = Añade una nueva entrada al stream (El id puede ser * para generar uno automáticamente)

```
r.xadd("Clave_stream", {"usuario": "Bob", "mensaje": "Hola Alice"}, id="*")
```

XRANGE = Lee un rango de entradas de un stream

```
r.XRANGE("Clave_stream", min=
("mimensaje_stream", min='id_inicial', max='id_final')
```

XREAD = Lee entradas de uno o mas streams opcionalmente bloqueando hasta que hasta nuevas entradas

```
r.XREAD = ({ "mimensaje_stream": "$"}, count=1, block=5000)
```

XGROUP CREATE = Crea un grupo de consumidores para un Stream

```
r.XGROUP_CREATE (name="mimensaje_stream", groupname="grupo1",
id="$", mkstream=True)
```

XREADGROUP GROUP = Lee entrada de un stream usando un grupo de consumidores

```
r.xreadgroup(groupname="grupo1",
consumername="consumidor1",
streams={ "mimensaje_stream": ">"}, # ">" = mensajes nuevos no
entregados
count=10)
```

COMANDOS PARA **BITMAPS**

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

SETBIT = Establece o borra el bit en un offset dado

```
r.setbit("Clave_bit", 10, 1)
```

GETBIT = Obtiene el valor del bit en un offset dado

```
r.GETBIT("Clave_bit")
```

BITCOUNT = Cuenta el número de bits establecidos (a 1) en una cadena

```
r.BITCOUNT("Clave_bit",bit inicio,bit final)
```

BITPOS = Encuentra la posición del primer bit establecido o no establecido

```
r.BITPOS("Clave_bit",bit_inicio,bit_final)
```

COMANDOS PARA **HYPERLOGS**

Es un algoritmo probabilístico para estimar la cardinalidad de un conjunto (números de elementos únicos)

PFADD = Añade elementos a un Hyperlog

```
r.pfadd("Clave_Hyperlog","valor")
```

PFCOUNT = Estima la cardinalidad de uno o más hyperloglog

```
r.pfcount("Clave_Hyperlog","Clave_hyperlog2")
```

PFMERGE = Combina varios Hyperloglog en uno

```
r.pfmerge("Clave_hiperlogunion ", "Hyperlog1", " Hyperlog2")
```

COMANDOS PARA Datos Geoespaciales

Permite almacenar coordenados de latitud y longitud y realizar consultas basadas en la distancia

GEOADD = Añade uno o más puntos geoespaciales

```
r.geoadd("Clave_datos", (-3.70379, 40.41677, "Usuario:"))
```

clave, longitud,latitud,id

GEODIST = Calcula la distancia entre dos miembros

```
r.GEODIST ("Clave_datos", "Miembro1", "Miembro2", unit="km")
```

GEORADIUS = Encuentra miembros dentro de un radio dado

```
r.GEORADIUS("Clave_datos",longitud,latitud, distancia,unit= km)
```

GEORADIUSBYMEMBER = Georadius pero centrado en un miembro existente

```
r.georadiusbymember("Clave_datos", "Miebro", 1500, unit="km")
```

GEOHASH = Devuelve el Geohash de uno o mas miembros

```
r.GEOHASH("clave_datos","Miembro")
```

Geosearch = georadius mejorado

```
r.geosearch(
```

```
"clave_dnd voy a buscar(en la que inserte los datos)",
```

```
longitude = lon,
```

```
latitude = lat,
```

```
radius=100,
```

```
unit = "m"
```

```
)
```

InfluxDB

InfluxDb es una base de datos optimizada para manejar datos que tienen una marca de tiempo(timestamp)

En este tipo de bases de datos lo importantes es analizar tendencias en lugar de únicamente guardar el estado actual.

Influx db está optimizado para ingerir millones de puntos de datos por segundo

Usos habituales de InfluxDB

- Monitorización (métricas de servidores, aplicaciones)

- IoT (Sensores enviando datos)

- Analítica en tiempo real (Precios de acciones, volumen de transacciones, detección de fraude)

Evolución de Influxdb

InfluxDB 1.x -> Era una pila de componentes separados (Telegraf, influxdb, chronograf, kapacitor)

Su lenguaje era parecido al de sql

InfluxDB 2.x -> Se unifica la base de datos, la interfaz y el motor de alertas/tareas en el mismo binario

Se introduce nuevo lenguaje llamado Flux

InfluxDB 3.x -> Es la versión más reciente que está reescrita en Rust y basada en Apache Arrow y Apache Parquet, soporta Sql

Características InfluxDB

Los datos son inmutables(solo se pueden añadir)

Altas tasas de escritura

Consultas de Agregación

Borrado de datos Viejos

No se puede usar como base de datos de negocio

No se puede hacer update de puntos individuales

No se puede hacer select* sin un rango de tiempo

El punto es la unidad fundamental de almacenamiento

Es la representación de un dato en un momento específico en el tiempo

Cada punto consta de cuatro componentes esenciales que lo definen:

- Measurement(medicion)

- Tags (Etiquetas)

- Fields(campos)

- Timestamp(marca de tiempo)

Measurement es el contenedor lógico de la información es lo que seria una tabla en sql

Tags Son metadatos que describen el punto, con ellas se pueden hacer operaciones de filtrado (where, group by) en las consultas

Están indexados

Fields son los valores reales que se están midiendo, estos no están indexados

Timestamp, es el índice primario de la base de datos

Es un valor de tiempo de alta precisión que indica cuando ocurrió la precision

CUIDADOOOO

PARA FILTRAR TAG, PARA CALCULAR COSAS FIELD

Conceptos clave con los puntos

Organizaciones -> Es el nivel más alto de la jerarquía, representa al usuario , la empresa o el proyecto que posee los recursos

La función es proporcionar un espacio de nombres para aislar usuarios y daots

Bucket -> Es el contenedor de datos principal, equivalente a una base de datos

Un bucket agrupa Measurements con el mismo propósito y la mismo política de retención

Estos se configuran con un TTL para ahorrar espacio

Token de API(Autenticacion)

Influxdb utiliza un sistema de tokens, para autenticar las aplicaciones, estos están asociados a las organizaciones

Line protocol, es el protocolo de texto para recibir e ingerir datos de series temporales de manera eficiente, esta basado en el envio de líneas de texco donde cada línea representa un único punto y debe contener los cuatro componentes esenciales en un orden y formato estricto

Separadores y espacios, la estructura es muy sensible a los espacios, el tag y el field debn estar separados exactamente por un espacio y dentro de ellos no pueden haber espacion

Establecer conexión en Influx

```
import influxdb_client

from influxdb_client.client.write_api import ASYNCHRONOUS, WriteOptions
from influxdb_client.client.exceptions import InfluxDBError
from urllib3.exceptions import NewConnectionError
from influxdb_client import Point

INFLUX_URL = "http://influxdb-influxdb2-1:8086"
INFLUX_TOKEN = "MyInitialAdminToken0="

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

client = None

try:
    client = influxdb_client.InfluxDBClient(
        url=INFLUX_URL,
        token=INFLUX_TOKEN,
        org="docs"
    )
    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}")
        ##Aqui va la logica
    else:
```

```

        print(f"[Error] Conexion 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:
    if client:
        client.close()

    print("--- Conexión Cerrada ---")

```

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

Hay que distinguir entre el cliente en si mismo y la Write api, que es una capa que se encarga de la escritura

El cliente -> Es la conexión general

WriteAPI es el metodo especifico para enviar datos de la manera más eficiente y segura

WriteAPI, su función mas importante es la de escritura en lote (batching)

Cuando le das un objeto POINT para escribir no necesariamente lo envia de inmediato, si está en modo asíncrono, se almacena los puntos en un buffer interno, cuando este buffer se llene o pase un tiempo definido, se envia todo el lote al servidor

(Lo normal es probar con el modo síncrono para ver si se hace correctamente ya que no hay que esperar)

Creación del punto

```

p = POINT("temperatura_medicion") \
    .tag("location","oficina") \
    .field("value",23,5) \
    .time(None) #Usara el tiempo de Ingesta

```

Si queremos incluir el tiempo tenemos que utilizar la librería datetime

Import datetime

```
Tiempo_actual = datetime.datetime.utcnow().isoformat() + "Z"
```

```
.time(Tiempo_actual)
```

Otra forma(tiempo según venga en csv)

```
from datetime import datetime
```

```
fecha_original = fila["date_x"].strip()
```

```
fecha_obj = datetime.strptime(fecha_original, "%m/%d/%Y") con todo -> "%m/%d/%Y  
%H:%M:%S"
```

```
fecha_iso = fecha_obj.isoformat()
```

```
.time(fecha_iso)
```

Modo síncrono

```
Write_api = client.write_api(write_options=SYNCHRONOUS) /ASYNCHRONOUS
```

El modo sincrónico, no usa batching por lo que envía nada más tener el punto

El modo asíncrono sí crea lotes, (tiene valores por defecto) se puede controlar de la siguiente manera

```
write_api = client.write_api(
```

```
    write_options=WriteOptions(
```

```
        batch_size=1000, #Tamaño del lote
```

```
        flush_interval=5000, #Tiempo de espera en caso de que no llegue al tamaño del lote
```

```
    )
```

```
)
```

Escribir el punto

```
write_api.write(bucket="prueba",org="docs", record=p)
```

