

Procesado de datos

6. Almacenamiento y monitorización de resultados en streaming

Boni García

<http://bonigarcia.github.io/>
boni.garcia@uc3m.es

Departamento de Ingeniería Telemática
Escuela Politécnica Superior

2020/2021

uc3m | Universidad **Carlos III** de Madrid

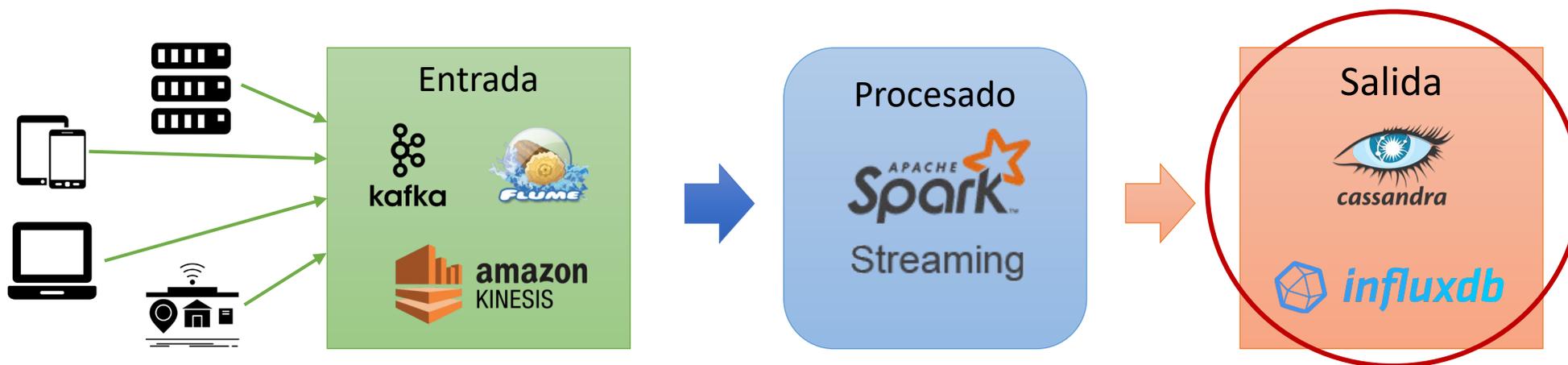


Contenidos

1. Introducción
2. Bases de datos
3. Cassandra
4. InfluxDB
5. Resumen

1. Introducción

- Spark Streaming implementa un data pipeline en diferentes etapas:
 1. Recepción de datos de entrada (data ingestion) de diferentes tipos fuentes
 2. Procesado con Spark Streaming usando una técnica llamada micro-batching
 3. Entrega de resultados a sistemas de salida (downstream system), como:
 - Un sistema gestor de base de datos (DBMS)
 - Un live dashboard



Contenidos

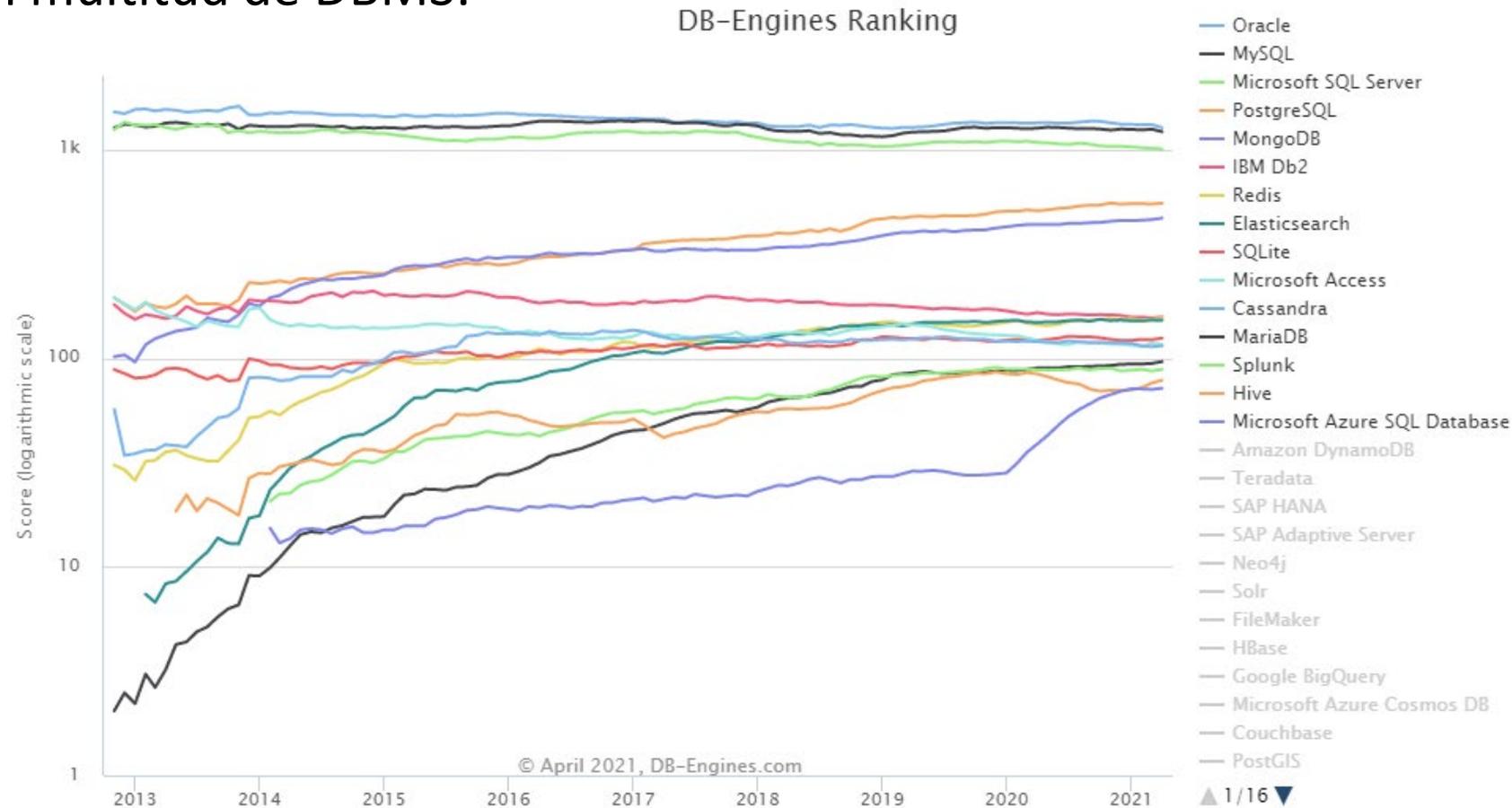
1. Introducción
2. Bases de datos
 - Relacionales
 - NoSQL
 - De series temporales
3. Cassandra
4. InfluxDB
5. Resumen

2. Bases de datos

- Una **base de datos** es un conjunto ordenado de datos perteneciente al mismo contexto
- Un **sistema gestor de bases de datos** (DBMS) es el software que permite almacenar y gestionar la información de una base de datos
 - Usualmente no hacemos distinción entre DBMS y base de datos
- Los dos grandes tipos de DBMS son:
 - **Relacionales** (RDBMS)
 - No relacionales (**NoSQL**)
- Además, existen DBMS optimizados para un caso de uso específico
 - En esta asignatura, por su interés en el procesamiento de datos en streaming, estudiaremos las bases de datos de series temporales (time series database, **TSDB**)

2. Bases de datos

- Existen multitud de DBMS:



http://db-engines.com/en/ranking_trend

2. Bases de datos - Relacionales

- Un **base de datos relacional** (RDBMS) almacena datos como un conjunto de tablas con columnas y filas siguiendo el modelo relacional (RM), que fue postulado en 1970 por Edgar Frank Codd en los laboratorios IBM:
 - Las **tablas** guardan información sobre los objetos que se representan
 - Cada **columna** de una tabla guarda un determinado tipo de datos
 - Las **filas** (o registro) de la tabla representan una recopilación de valores relacionados de un objeto o entidad
 - Un **campo** almacena el valor de una fila y columna
 - Cada fila de una tabla puede tener un identificador único llamado **clave primaria**
 - Las filas de diferentes tablas pueden relacionarse con **claves foráneas**

Id	Name	AlterEgo
1	Superman	Clark Kent
2	Spiderman	Peter Parker
3	Hulk	Bruce Banner

SuperHeroes

IdHero	IdUniverse
1	2
2	1
3	1

SuperHeroesUniverse

Id	Name
1	Marvel
2	DC

Universe

2. Bases de datos - Relacionales

- Los aspectos más importantes de las base de datos relacionales son:
 1. Uso de **SQL** (Standard Query Language), que es un lenguaje estándar que permite gestionar un DBMS. Los comandos SQL se dividen en 2 categorías:
 - Lenguaje de manipulación de datos (DML). Permite realizar las operaciones básicas sobre los datos (CRUD = create, read, update, delete): `SELECT`, `INSERT`, `DELETE`, `UPDATE`
 - Lenguaje de definición de datos (DDL). Permite crear, borra y modificar tablas, usuarios, vistas, o índices: `CREATE TABLE`, `DROP TABLE`, `ALTER TABLE`
 2. La **integridad de datos** es total en un RDBMS. Esto implica:
 - Integridad de entidad, que garantiza que una clave primaria es única y no nula
 - Integridad referencial, por la cual se garantiza un estado coherente de la base de datos (la relación entre dos tablas permanece sincronizada durante las operaciones de actualización y eliminación)

2. Bases de datos - Relacionales

- Aspectos importantes de bases de datos relacionales (continuación):
3. Soporta las **transacciones**, que son una o más sentencias SQL que forman una unidad lógica de trabajo. Las transacciones cumplen los parámetros **ACID**:
 - Atomicidad: requiere que la transacción completa se ejecute correctamente (commit). Si una parte de la transacción falla, toda ella queda invalidada (rollback)
 - Coherencia: los datos escritos en la base de datos como parte de la transacción deben cumplir todas las reglas definidas (integridad de datos)
 - Aislamiento: control de concurrencia e independencia
 - Durabilidad: todos los cambios son permanentes si la transacción se completa correctamente
 - Hay un tipo de DBMS llamado **objeto-relacional** (ORDBMS), que permite persistir objetos provenientes del paradigma de programación orientado a objetos (OOP)

2. Bases de datos - Relacionales

- Ejemplos de RDBMS:
 - MySQL (software libre): <http://www.mysql.org>
 - MS SQL Server (comercial): <http://www.microsoft.com/sql>
- Ejemplos de ORDBMS:
 - Oracle (comercial): <http://www.oracle.com>
 - PostgreSQL (software libre): <http://www.postgresql.org/>



2. Bases de datos - NoSQL

- Tradicionalmente, para añadir grandes cantidades de datos en RDBMS se emplea escalado vertical (añadir más recursos a la misma máquina)
- Con la explosión de datos manejados por las grandes compañías de Internet (Google, Amazon, Facebook, etc.) al principio de los años 2000, surge el problema del **Big Data**, y la necesidad de **escalado horizontal** (conseguir escalabilidad añadiendo más nodos a un clúster)
 - Los artículos fundamentales (seminal papers) que influenciaron una forma de DBMS presentaban los sistemas de almacenamiento distribuidos BigTable (Google, 2006) y Dynamo (Amazon, 2007)
 - En ese momento se comenzó a hablar de un nuevo tipo de bases de datos no relaciones con el nombre de **NoSQL**

2. Bases de datos - NoSQL

- El término **NoSQL** (“no sólo SQL”) define un tipo de DBMS que difieren del modelo relacional:
 - No utilizan estructuras fijas como tablas para el almacenamiento de los datos
 - No usan el modelo entidad-relación
 - No suelen permitir operaciones JOIN (para evitar sobrecargas en búsquedas)
 - Arquitectura distribuida (datos pueden estar compartidos en varias máquinas)
- Se recomienda usar bases de datos NoSQL:
 - Cuando el volumen de datos crece rápidamente en momentos puntuales (>Terabyte)
 - Cuando la escalabilidad de la solución relacional no es viable (muy costosa)
 - Cuando tenemos elevados picos de uso del sistemas
 - Cuando el esquema de la base de datos no es homogéneo, es decir, cuando en cada inserción de datos la información que se almacena puede tener campos distintos

2. Bases de datos - NoSQL

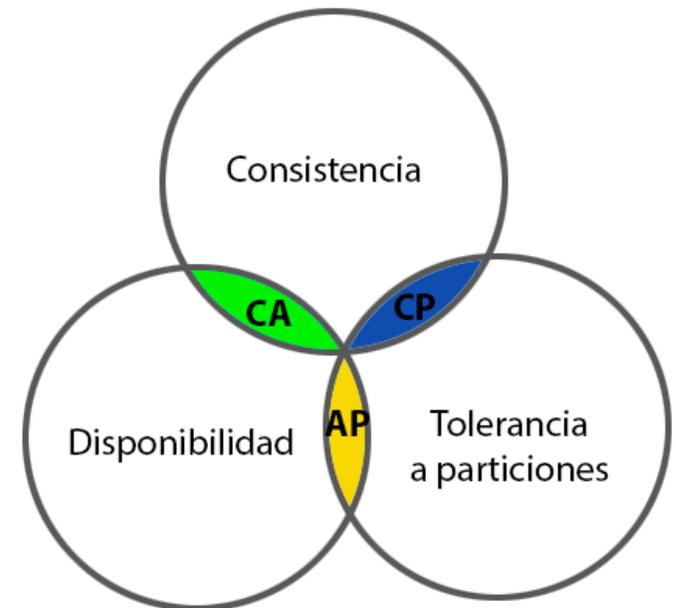
- Se puede clasificar los diferentes tipos de bases de datos NoSQL usando diferentes criterios. En cuanto al **modelo de datos**, hay 4 categorías principales de bases de datos NoSQL:
 1. Orientadas a documentos. Este tipo almacena la información como un documento, por ejemplo JSON
 2. Orientadas a columnas. Funcionan de forma parecida a las bases de datos relacionales, pero almacenando columnas de datos en lugar de registros
 3. Clave-valor. Cada elemento está identificado por una clave, lo que permite la recuperación de la información de forma muy rápida
 4. En grafo. La información se representa como nodos de un grafo y sus relaciones con las aristas del mismo

2. Bases de datos - NoSQL

- Otra forma de clasificar las bases de datos NoSQL es usando el **teorema CAP**
 - Conjeturado por Eric Brewer en el año 2000 y demostrado por Gilbert y Lynch en 2002
 - Postula que es imposible garantizar las siguientes funciones de forma completa y simultánea en sistemas distribuidos (como mucho se garantiza un par de ellas a la vez):
 1. **Consistencia** (consistency): al realizar una consulta o inserción siempre se tiene que recibir la misma información, con independencia del nodo que procese la petición
 2. **Disponibilidad** (availability): todos los nodos operativos deben permitir operaciones de lectura y escritura, devolviendo una respuesta a los clientes
 3. **Tolerancia a particiones** (partition-tolerance): el sistema tiene que seguir funcionando aunque existan fallos parciales en las particiones que tenga el sistema distribuido (referido a veces como tolerancia a fallos)

2. Bases de datos - NoSQL

- De esta forma, en cuanto a consistencia/disponibilidad podemos clasificar las bases de datos en 3 categorías:
 - **CA**: garantizan consistencia y disponibilidad, pero tienen problemas con la tolerancia a particiones. Este problema lo suelen gestionar replicando los datos. En esta categoría se encontrarían todos los RDBMS
 - **CP**: garantizan consistencia y tolerancia a particiones. Para lograr la consistencia y replicar los datos a través de los nodos, sacrifican la disponibilidad
 - **AP**: garantizan disponibilidad y tolerancia a particiones, pero no la consistencia, al menos de forma total. Se puede conseguir consistencia parcial a través de la replicación y la verificación



2. Bases de datos - NoSQL

- Algunas de las bases de datos NoSQL más utilizadas son:
 - MongoDB (<https://www.mongodb.com/>): Orientada a documento. CP por defecto (aunque se permite mejorar la disponibilidad mediante replicación en nodo secundarios)
 - Redis (<https://redis.io/>): Clave valor. CP. Base de datos en memoria, aunque opcionalmente implementa durabilidad de datos
 - HBase (<https://hbase.apache.org/>): Orientada a columna. AP. Forma parte del ecosistema Hadoop
 - Cassandra (<http://cassandra.apache.org/>): Orientada a columna. AP. Alto rendimiento en operaciones de lectura y escritura



2. Bases de datos - De series temporales

- Las bases de datos de **series temporales** (time series databases, **TSDB**) son DBMS optimizadas para el almacenamiento de secuencias de datos ordenados cronológicamente
 - Internamente pueden estar basadas en el esquema relacional o NoSQL
- Algunas TSDB disponibles actualmente son:
 - InfluxDB (<https://www.influxdata.com/>): Open-source. NoSQL de tipo columna. Ofrece mecanismos de visualización y alertas
 - Prometheus (<https://prometheus.io/>): Open-source. NoSQL. Ofrece alertas
 - TimescaleDB (<https://www.timescale.com/>): Open-source. Relacional (diseñada como extensión de PostgreSQL)
 - kdb+ (<https://kx.com/>): Comercial. Relacional. Focalizada en garantizar un acceso rápido a las series temporales (datos in-memory)



Contenidos

1. Introducción
2. Bases de datos
- 3. Cassandra**
 - Modelo de datos
 - Instalación
 - CQL
 - PySpark
4. InfluxDB
5. Resumen

3. Cassandra

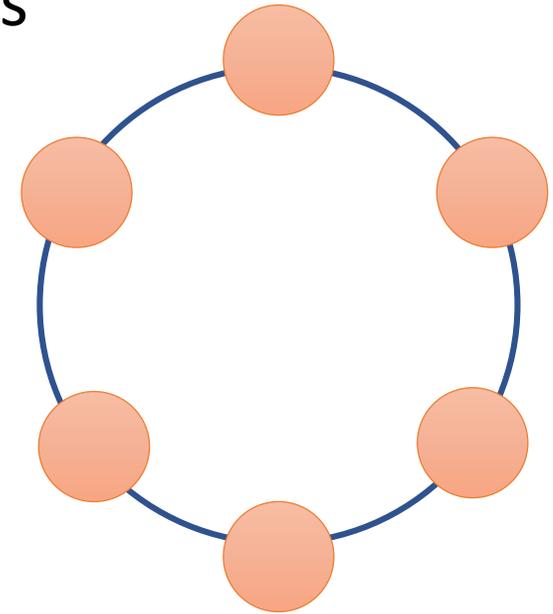
- Apache Cassandra es un DBMS distribuido NoSQL (escalabilidad horizontal al ser desplegado en un clúster)
- Creado por Facebook en 2008 y donado como proyecto open-source a la Apache Software Foundation en 2010
- Principales características: alta disponibilidad, escalabilidad, tolerancia a fallos, alto rendimiento



<http://cassandra.apache.org/>

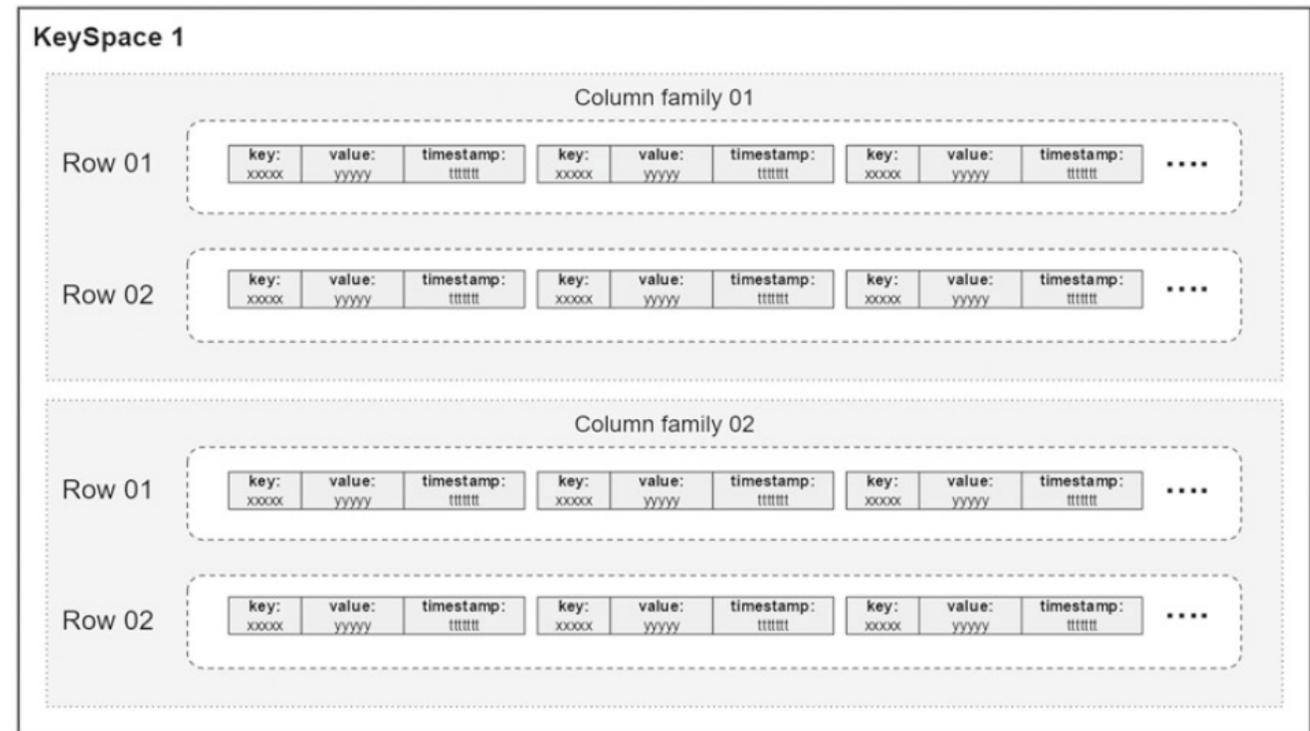
3. Cassandra

- La característica fundamental de Cassandra en comparación con otros DBMS distribuidos es su **alta disponibilidad** de datos
 - A diferencia de los DBMS distribuidos basados en el enfoque maestro-esclavo, los nodos del clúster son todos iguales (*masterless*)
 - Estos nodos se disponen en una arquitectura en forma de anillo, de forma que la redundancia sea más eficiente
 - Esta arquitectura permite un alto rendimiento en operaciones de lectura y escritura
- Por esta razón, Cassandra es una DBMS muy utilizada para persistir resultados provenientes de Spark (típicamente en procesados batch)



3. Cassandra - Modelo de datos

- El modelo de datos manejado en Cassandra es similar al que se usa en RDBMS, lo que ha facilitado su adopción
 - La unidad básica de información es la **columna**, que contiene tres campos: clave, valor, y marca de tiempo (key-value-timestamp)
 - Una **fila** es una colección de varias columnas
 - Una **familia de columnas** es una colección ordenada de filas
 - Análogo al concepto de tabla en una RDBMS
 - Un **keyspace** contiene una o mas familias de columnas
 - Análogo al concepto de esquema (schema) en una RDBMS



3. Cassandra - Modelo de datos

- Cassandra permite tres tipos de datos:
 1. Nativos (built-in): boolean, blob, ascii, bigint, Counter, float, inet, int, text, varchar, timestamp, variant
 2. Colecciones: list, map, set
 3. Datos definidos por el usuario (user-defined types, UDT). Creados con la sentencia CREATE TYPE

<http://cassandra.apache.org/doc/latest/cql/types.html>

3. Cassandra - Instalación

- Cassandra está desarrollado en Java, con lo que necesitamos una JVM para ejecutarlo
- Podemos la distribución en <http://cassandra.apache.org/download/>
- En local, por defecto arranca en el puerto 9042

Ejemplo de arranque de Cassandra en Linux Mint (máquina virtual)

```
(base) user@linux-mint:~/bin/apache-cassandra-4.0-beta4/bin$ ./cassandra
(base) user@linux-mint:~/bin/apache-cassandra-4.0-beta4/bin$ CompilerOracle: dontinline
org/apache/cassandra/db/Columns$Serializer.deserializeLargeSubset
(Lorg/apache/cassandra/io/util/DataInputPlus;Lorg/apache/cassandra/db/Columns;I)Lorg/apache/cassandra/db/Columns;
...
INFO [main] 2021-04-12 10:29:53,985 StorageService.java:1599 - JOINING: Finish joining ring
INFO [main] 2021-04-12 10:29:54,000 StorageService.java:2712 - Node localhost/127.0.0.1:7000 state
jump to NORMAL
INFO [main] 2021-04-12 10:29:54,003 StorageService.java:2712 - Node localhost/127.0.0.1:7000 state
jump to NORMAL
```

3. Cassandra - CQL

- Cassandra define un lenguaje de consulta propio llamado **CQL** (Cassandra Query Language), muy similar a SQL (SQL-like)
- Podemos realizar consultas CQL usando el script `cqlsh` disponible en la distribución de Cassandra
 - Se conecta a una instancia de Cassandra (por defecto localhost:9042)

```
(base) user@linux-mint:~/bin/apache-cassandra-4.0-beta4/bin$ ./cqlsh
Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 4.0-beta4 | CQL spec 3.4.5 | Native
protocol v4]
Use HELP for help.
cqlsh>
```

<http://cassandra.apache.org/doc/latest/tools/cqlsh.html>

3. Cassandra - CQL

- Algunos de los comandos más habituales en CQL son:

- DESCRIBE: para obtener información sobre un aspecto

```
cqlsh> DESCRIBE KEYSPACES;  
  
system_schema  system_auth  system  system_distributed  system_traces
```

- CREATE KEYSPACE: para crear un keyspace.

- Hay que especifica la estrategia de replicación (SimpleStrategy para replicación en todos los nodos o NetworkTopologyStrategy para sistemas en producción) y factor de replicación (número de veces que los datos se replica en los nodos)

```
cqlsh> CREATE KEYSPACE test WITH replication = {'class':'SimpleStrategy', 'replication_factor' : 1};  
  
cqlsh> DESCRIBE KEYSPACES;  
  
system_schema  system_auth  system  system_distributed  test  system_traces
```

- USE: para seleccionar un keyspace

```
cqlsh> USE test;  
cqlsh:test>
```

<http://cassandra.apache.org/doc/latest/cql/index.html>

3. Cassandra - CQL

- CREATE TABLE: para crear una tabla dentro de un keyspace:
 - Es obligatorio escoger una clave primaria formada por 1 o varios campos

```
cqlsh:test> CREATE TABLE people (  
    id text,  
    name text,  
    age int,  
    PRIMARY KEY (id)  
);
```

Ejemplo de consulta para leer la marca de tiempo (timestamp) en la que fue escrita una columna. El tiempo se almacena en formato Unix time (número de segundos desde el 01/01/1970)

- SELECT ... FROM: Para leer los elementos de una tabla

```
cqlsh:test> SELECT * FROM people;
```

id	age	name
37221cc0-469b-4ddc-b41a-045ba6b9dd53	19	Justin
426bd0cb-1f80-4861-9aec-3a76cbc70e60	30	Andy
926f97c6-afb2-41d1-86b1-e2b2d7e8ba59	null	Michael

```
cqlsh:test> SELECT name, writetime(name) FROM people;
```

name	writetime(name)
Justin	1585419955242000
Andy	1585419820468000
Michael	1585419955244001

3. Cassandra - CQL

- DESCRIBE: Para ver las tablas disponibles en un keyspace

```
cqlsh:test> DESCRIBE tables;
people
```

- TRUNCATE: Para eliminar los datos de una tabla

```
cqlsh:test> TRUNCATE people;
cqlsh:test> SELECT * FROM people;

 id | age | name
----+-----+-----
(0 rows)
```

- DROP TABLE: Para eliminar completamente una tabla

```
cqlsh:test> DROP TABLE people;
```

3. Cassandra - PySpark

- Podemos leer y escribir datos a través de PySpark usando la API de DataFrames y la dependencia **Spark Cassandra Connector**
 - Para la versión que estamos usando de Spark, habrá que usar la dependencia `com.datastax.spark:spark-cassandra-connector_2.11:2.4.3`
 - Además, al crear la sesión Spark, habrá que especificar el nombre de máquina y puerto donde está arrancada Cassandra (parámetros `spark.cassandra.connection.host` y `spark.cassandra.connection.port`)

```
spark = (SparkSession
    .builder
    .master("local[*]")
    .appName("MyAppName")
    .config("spark.jars.packages", "com.datastax.spark:spark-cassandra-connector_2.11:2.4.3")
    .config("spark.cassandra.connection.host", "localhost")
    .config("spark.cassandra.connection.port", "9042")
    .getOrCreate())
```

<https://github.com/datastax/spark-cassandra-connector>

3. Cassandra - PySpark

- Una vez configurado el contexto Spark adecuadamente, podemos leer y escribir DataFrames de Cassandra como sigue:

```
# Write DataFrame to Cassandra
(df.write
 .format("org.apache.spark.sql.cassandra")
 .options(keyspace="my-keyspace", table="my-table")
 .mode("append")
 .save())

# Read DataFrame from Cassandra
readDf = (spark.read
 .format("org.apache.spark.sql.cassandra")
 .options(keyspace="my-keyspace", table="my-table")
 .load())
```

La tabla `my-table` en el keyspace `my-keyspace` deben existir previamente en Cassandra

- Si queremos usar este conector de Cassandra para leer-escribir RDDs, habría que convertir el RDD en cuestión a DataFrame en primer lugar:

```
df = spark.createDataFrame(rdd)
```

3. Cassandra - PySpark

Este ejemplo crea un DataFrame desde un fichero JSON, lo escribe en Cassandra, y luego lee el contenido de la tabla

Previamente, se debe haber creado una tabla en Cassandra usando el siguiente comando CQL:

```
CREATE TABLE people (  
  id text,  
  name text,  
  age int,  
  PRIMARY KEY (id)  
);
```

Se crea una clave primaria usando una función definida de usuario (UDF) que genera un identificador usando una función de identificador único universal (UUID)

```
from pyspark.sql import SparkSession  
from pyspark.sql.functions import udf  
from pyspark.sql.types import StringType  
from uuid import uuid4  
  
# Local SparkSession  
spark = (SparkSession.builder.master("local[*]")  
        .appName("JSON-DataFrame-Cassandra")  
        .config("spark.jars.packages", "com.datastax.spark:spark-cassandra-connector_2.11:2.4.3")  
        .config("spark.cassandra.connection.host", "localhost")  
        .config("spark.cassandra.connection.port", "9042")  
        .getOrCreate())  
spark.sparkContext.setLogLevel("ERROR")  
  
# Input data: Create DataFrame object from JSON file  
people = spark.read.json("../data/people.json", multiLine=True)  
people.printSchema()  
people.show()  
  
# Write DataFrame in Cassandra including id column  
randomId = udf(lambda: str(uuid4()), StringType())  
(people.withColumn("id", randomId())  
 .write  
 .format("org.apache.spark.sql.cassandra")  
 .options(keyspace="test", table="people")  
 .mode("append")  
 .save())  
  
# Read DataFrame from Cassandra  
readDf = (spark.read  
        .format("org.apache.spark.sql.cassandra")  
        .options(keyspace="test", table="people")  
        .load())  
readDf.printSchema()  
readDf.show(truncate=False)
```

3. Cassandra - PySpark

Este ejemplo crea un DataFrame en streaming usando una secuencia de datos de prueba como entrada, Realiza un filtrado eliminando los valores impares y almacena el resultado en Cassandra

Previamente, se debe haber creado una tabla en Cassandra usando el siguiente comando CQL:

```
CREATE TABLE seq (  
  timestamp timestamp,  
  value text,  
  PRIMARY KEY (timestamp)  
);
```

```
from pyspark.sql import SparkSession  
  
def writeToCassandra(dataframe, batchId):  
    print(f"Writing DataFrame to Cassandra (micro-batch {batchId})")  
    (dataframe.write  
     .format("org.apache.spark.sql.cassandra")  
     .options(keyspace="test", table="seq")  
     .mode("append")  
     .save())  
  
# Local SparkSession  
spark = (SparkSession.builder.master("local[*]").appName("Rate-DataFrame-Cassandra")  
         .config("spark.jars.packages", "com.datastax.spark:spark-cassandra-connector_2.11:2.4.3")  
         .config("spark.cassandra.connection.host", "localhost")  
         .config("spark.cassandra.connection.port", "9042")  
         .getOrCreate())  
  
# 1. Input data: test DataFrame with sequence and timestamp  
df = (spark  
     .readStream  
     .format("rate")  
     .option("rowsPerSecond", 1)  
     .load())  
  
# 2. Data processing: filter odd values  
even = df.filter(df["value"] % 2 == 0)  
  
# 3. Output data: show results in the console  
query = (even.writeStream  
         .outputMode("update")  
         .format("console")  
         .foreachBatch(writeToCassandra)  
         .start())  
  
query.awaitTermination()
```

Cassandra no permite escritura en streaming, con lo que es necesario hacerlo dataframe a dataframe

3. Cassandra - PySpark

Este ejemplo crea un DataFrame en streaming usando un socket como entrada, cuenta las palabras que recibe en cada línea, y almacena el resultado en Cassandra. Utiliza el mismo mecanismo del ejemplo anterior para crear identificadores únicos

Previamente, se debe haber creado una tabla en Cassandra usando el siguiente comando CQL:

```
CREATE TABLE wordcount (  
  id text,  
  word text,  
  count int,  
  PRIMARY KEY (id)  
);
```

```
from pyspark.sql import SparkSession  
from pyspark.sql.functions import explode, split, udf  
from pyspark.sql.types import StringType  
from uuid import uuid4  
  
def writeToCassandra(dataframe, batchId):  
    print(f"Writing DataFrame to Cassandra (micro-batch {batchId})")  
    randomId = udf(lambda: str(uuid4()), StringType())  
    (dataframe.withColumn("id", randomId())  
     .write  
     .format("org.apache.spark.sql.cassandra")  
     .options(keyspace="test", table="wordcount")  
     .mode("append")  
     .save())  
  
# Local SparkSession  
spark = (SparkSession.builder.master("local[*]").appName("Socket-DataFrame-Cassandra")  
        .config("spark.jars.packages", "com.datastax.spark:spark-cassandra-connector_2.11:2.4.3")  
        .config("spark.cassandra.connection.host", "localhost")  
        .config("spark.cassandra.connection.port", "9042")  
        .config("spark.sql.shuffle.partitions", "2")  
        .getOrCreate())  
spark.sparkContext.setLogLevel("ERROR")  
  
# 1. Input data: Create DataFrame from socket stream and 2. Data processing: word count  
...  
  
# 3. Output data: write results to Cassandra  
query = (wordCounts  
        .writeStream  
        .outputMode("update")  
        .foreachBatch(writeToCassandra)  
        .start())  
  
query.awaitTermination()
```

Contenidos

1. Introducción
2. Bases de datos
3. Cassandra
- 4. InfluxDB**
 - Modelo de datos
 - Cloud 2.0
 - API Python
 - Ejemplos
5. Resumen

4. InfluxDB

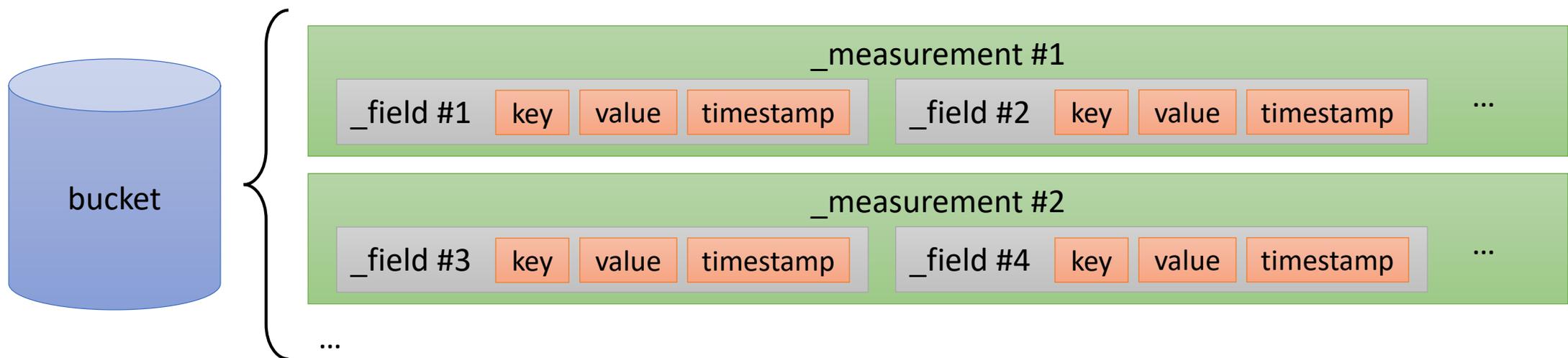
- InfluxDB es una base de datos de serie temporal (TSBD) open-source
- Implementa un SGBD NoSQL de tipo columna y ofrece un lenguaje de consultas SQL-like (similar a SQL) llamado Flux
- Forma parte del stack TICK creado por la empresa InfluxData:
 - Telegraf: Agente para la recolección de datos
 - InfluxDB: Base de datos TSBD
 - Chronograf: Interfaz de usuario para gestión y visualización de datos
 - Kapacitor: Motor de procesado de datos
- Los componentes de este stack se pueden descargar e instalar localmente o hacerlo a través de InfluxDB cloud (que ofrece una capa gratuita)



<https://www.influxdata.com/>

4. InfluxDB - Modelo de datos

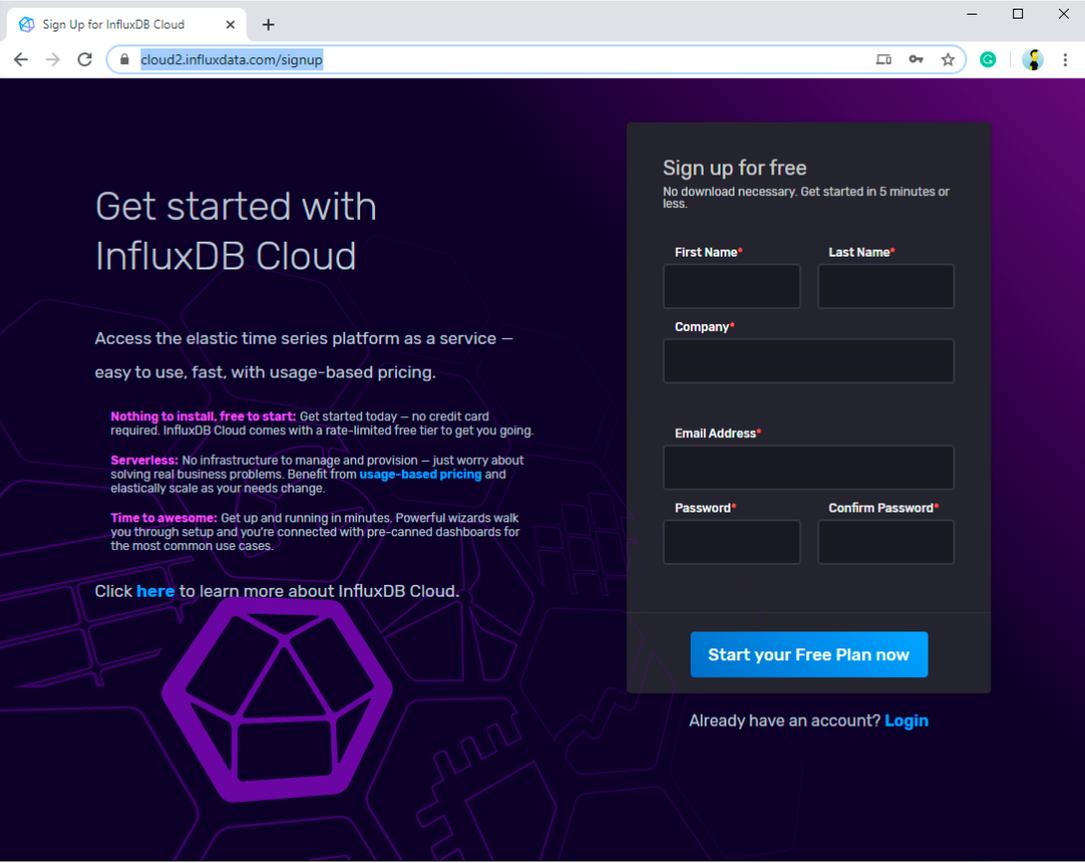
- Los datos en InfluxDB se organizan en conjuntos llamados cubetas (**buckets**)
- En cada bucket, se pueden encontrar diferentes medidas (**_measurement**)
- Cada medida tiene diferentes campos (**_field**), también llamados puntos, que están contienen clave, valor, y marca de tiempo (timestamp)
- Podemos etiquetar los campos usando **tags** (que pueden ser usado a posteriori en los mecanismos de visualización)



4. InfluxDB - Cloud 2.0

- Por simplicidad, vamos a usar InfluxDB Cloud 2.0 (<https://www.influxdata.com/products/influxdb-cloud/>)

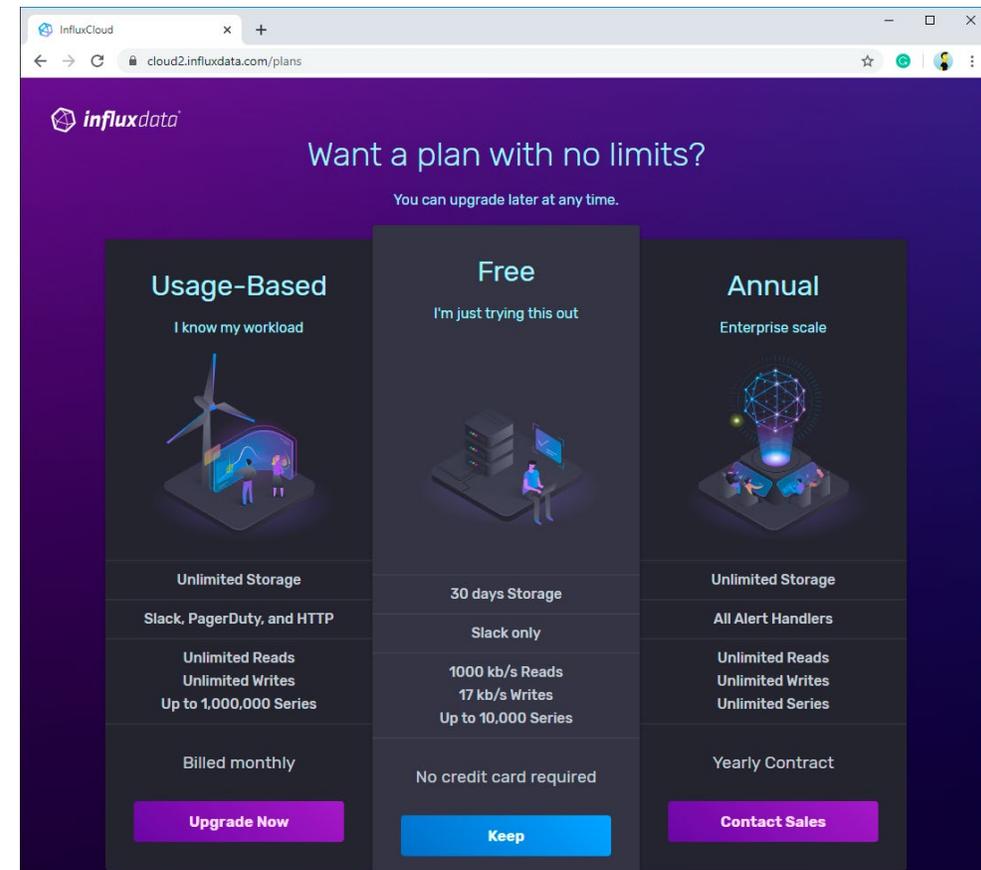
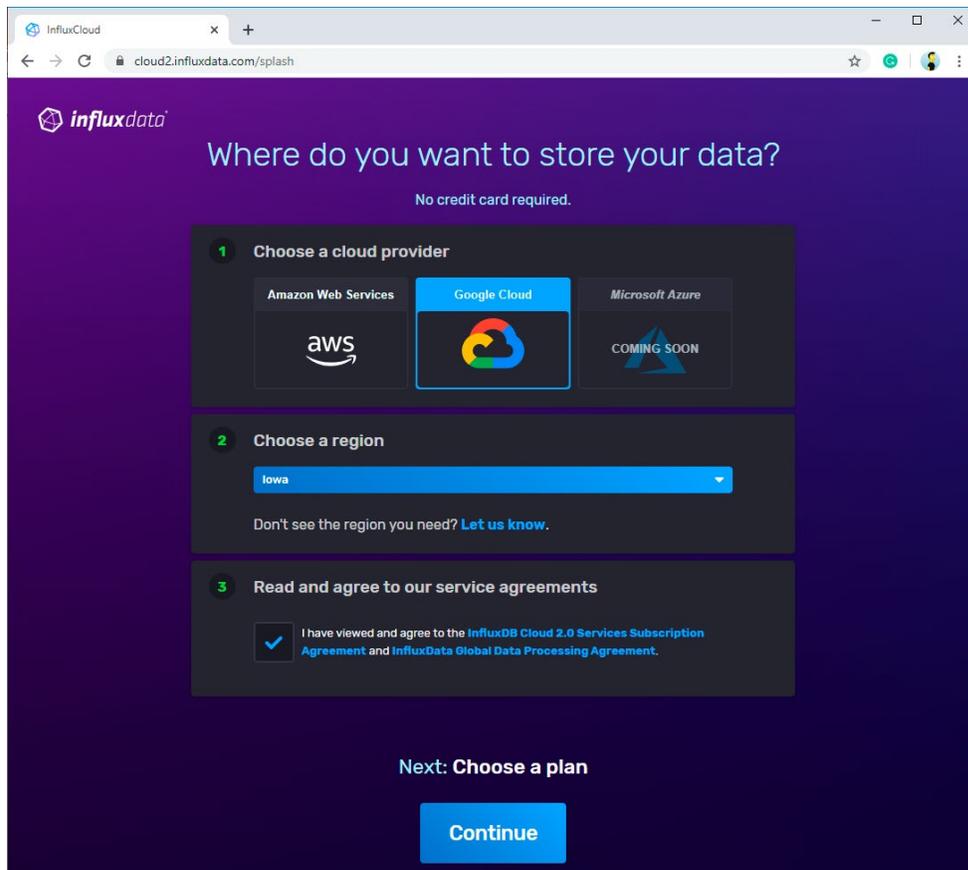
En primer lugar, hay que crearse cuenta en <https://cloud2.influxdata.com/signup>



The screenshot shows a web browser window with the URL cloud2.influxdata.com/signup. The page is titled "Sign Up for InfluxDB Cloud" and features a dark purple background with a white InfluxDB logo. The main heading is "Get started with InfluxDB Cloud". Below this, there is a sub-heading: "Access the elastic time series platform as a service – easy to use, fast, with usage-based pricing." Three key benefits are listed: "Nothing to install, free to start", "Serverless", and "Time to awesome". A "Click here to learn more about InfluxDB Cloud." link is provided. On the right side, there is a "Sign up for free" form with fields for "First Name*", "Last Name*", "Company*", "Email Address*", "Password*", and "Confirm Password*". A blue "Start your Free Plan now" button is at the bottom of the form, and a "Login" link is at the bottom right.

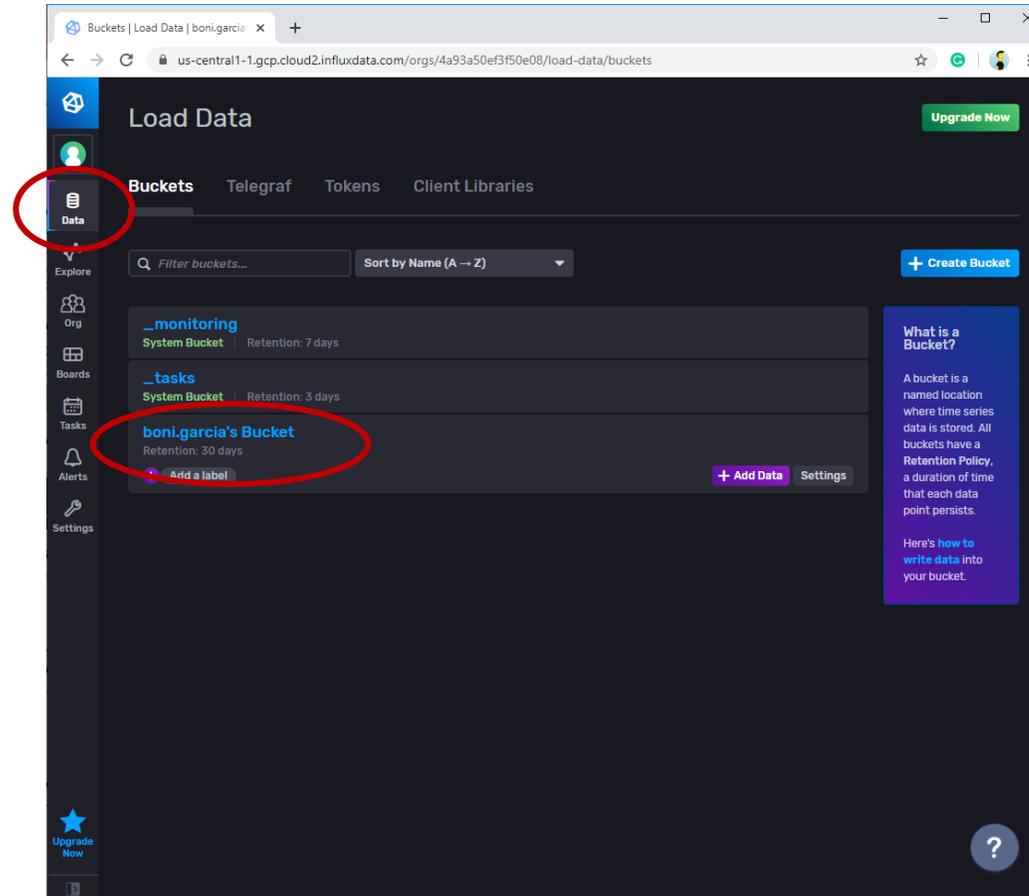
4. InfluxDB - Cloud 2.0

- En la creación de la cuenta habrá que elegir el proveedor de infraestructura (AWS o GCP) y el plan (gratis en nuestro caso):



4. InfluxDB - Cloud 2.0

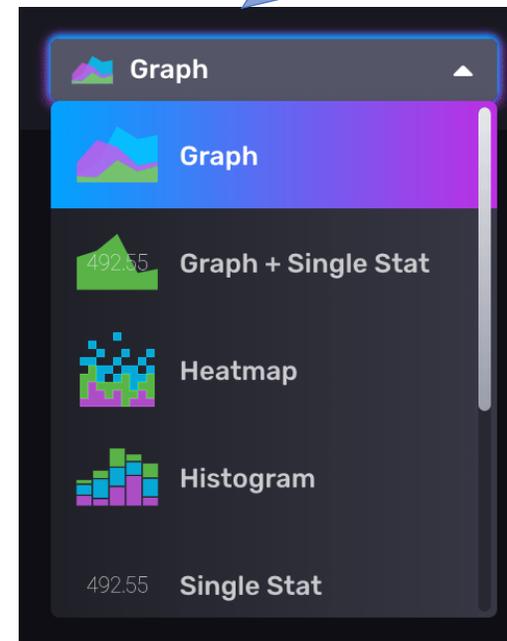
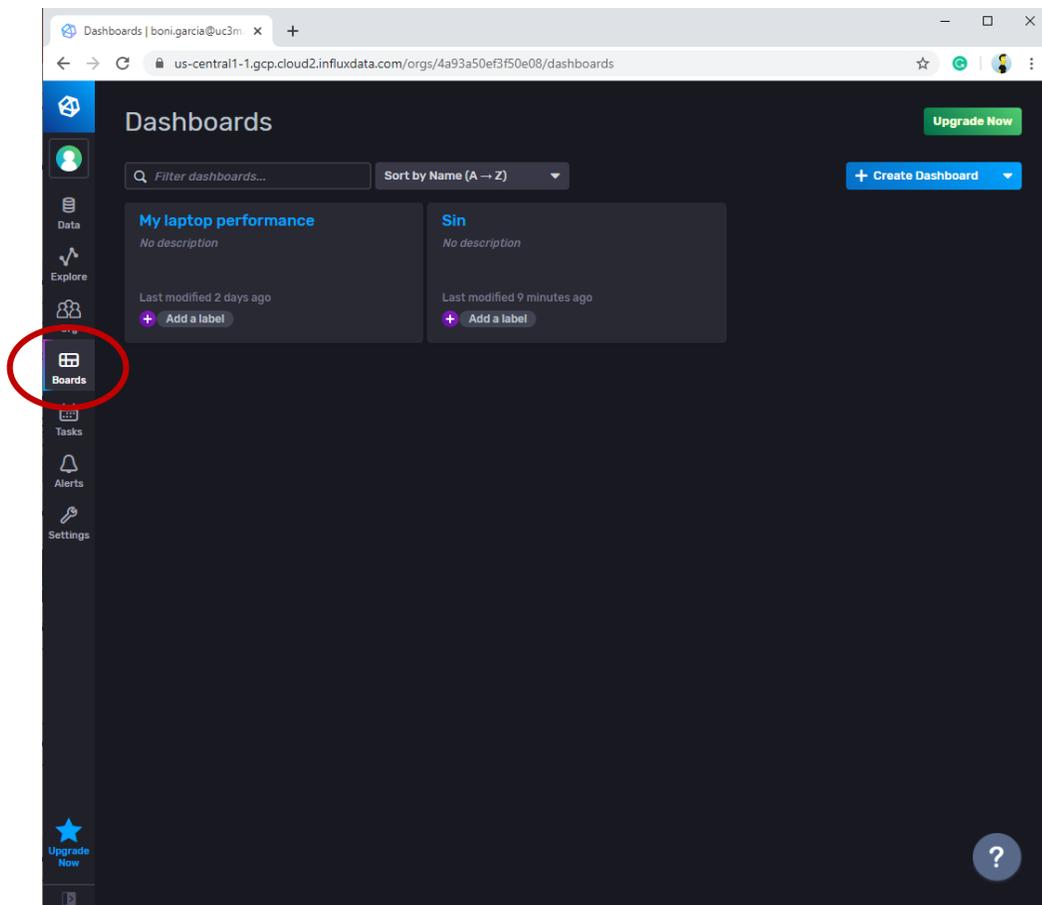
- En la versión gratuita, se proporciona un bucket que almacena nuestros datos hasta 30 días



4. InfluxDB - Cloud 2.0

- En la interfaz gráfica (Chronograf) podemos encontrar mecanismos de visualización de datos (live dashboards)

InfluxDB Cloud proporciona diferentes tipos de gráficas. El más común es el tipo "Graph", en el que se muestra un conjunto de datos en función del tiempo



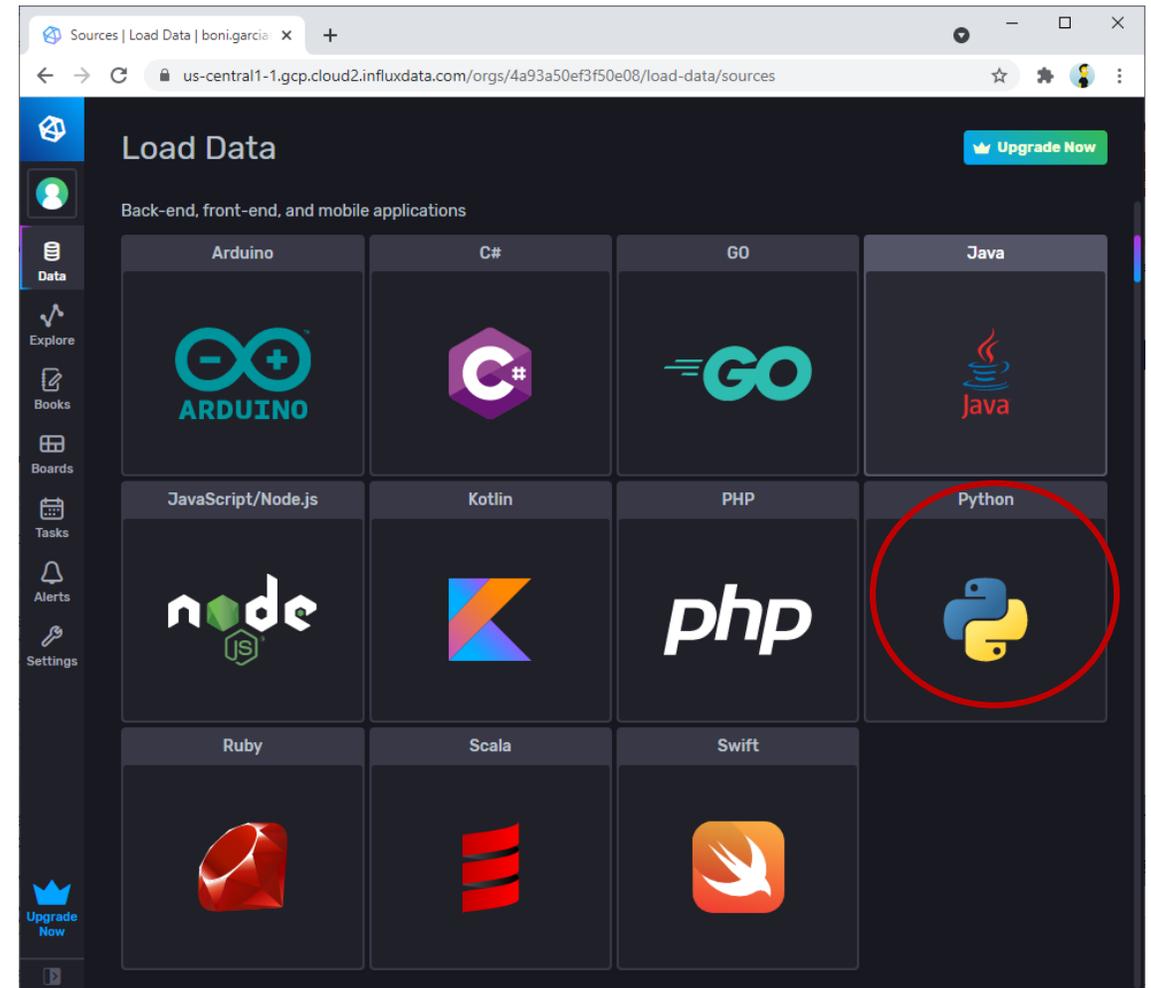
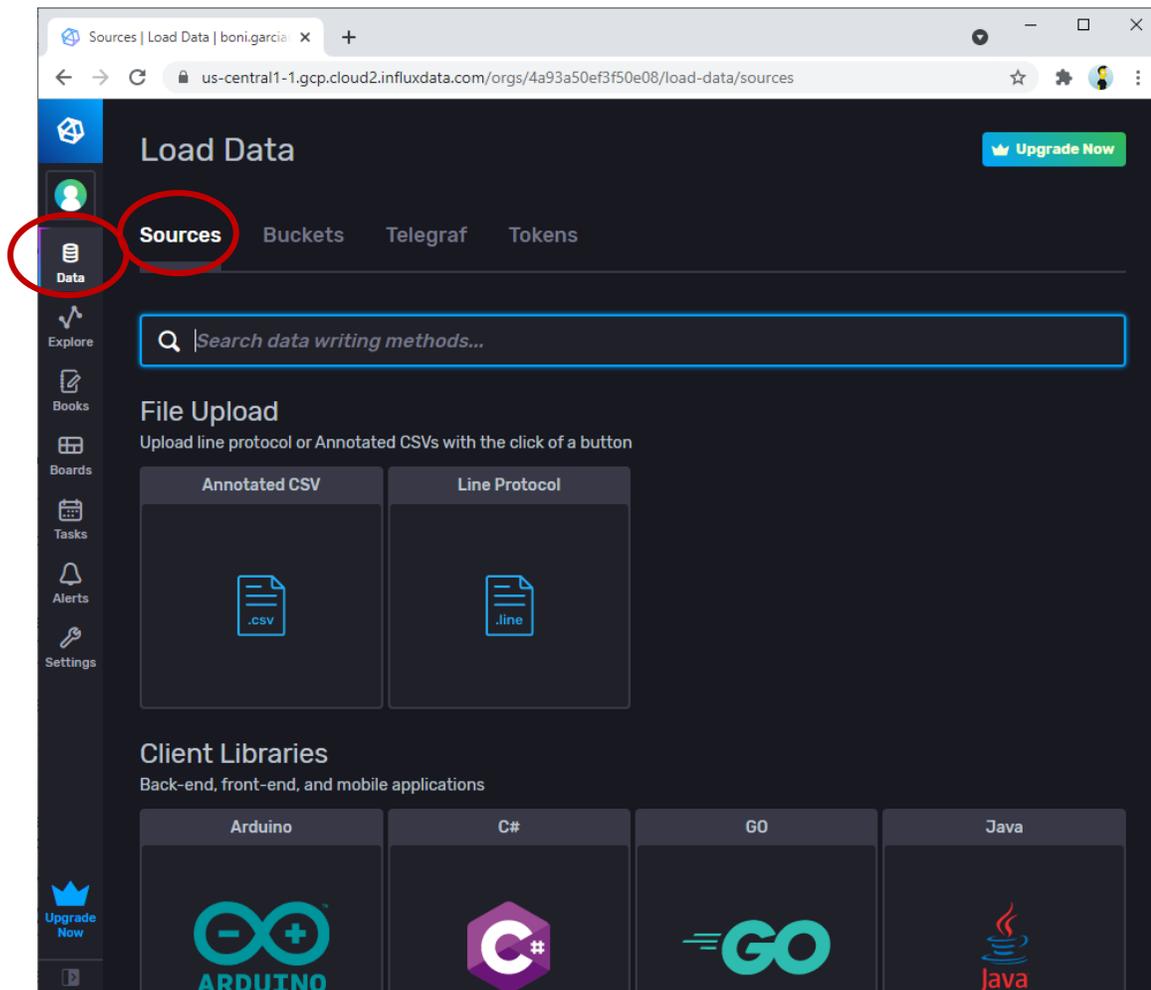
El eje de tiempo para estas gráficas usa formato **UTC** (Coordinated Universal Time)

4. InfluxDB - API Python

- No existe conector nativo de InfluxDB en Spark, así que tendremos que escribir los datos de los RDDs contenidos en DStreams o DataFrames a través de la **API Python de InfluxDB**
 - Código fuente <https://github.com/influxdata/influxdb-client-python>
 - Paquete Python: <https://pypi.org/project/influxdb-client/>
- Para usar esta API, necesitamos recopilar la siguiente información de nuestra instancia de InfluxDB (Cloud 2.0 en nuestro caso):
 1. URL de InfluxDB
 2. Nombre de organización
 3. Nombre de bucket
 4. Token de acceso (habrá que crearlo)

4. InfluxDB - API Python

- En InfluxDB Cloud 2.0, podemos encontrar esa información en:



4. InfluxDB - API Python

- En InfluxDB Cloud 2.0, podemos encontrar esa información en:

The screenshot shows the InfluxDB Cloud 2.0 interface for the Python client library. The page title is "Python" and the URL is "us-central1-1.gcp.cloud2.influxdata.com/orgs/4a93a50ef3f50e08/load-data/client-libraries/python". The interface includes a sidebar with navigation options like Data, Explore, Books, Boards, Tasks, Alerts, and Settings. The main content area is titled "Code Sample Options" and contains a "Token" field with a "+ Generate Token" button and a "Bucket" field with a "+ Create Bucket" button. Below these fields, there are two input boxes: "allbuckets" for the token and "boni.garcia's Bucket" for the bucket. A code block shows the installation and usage of the InfluxDB client library. Three callouts point to specific parts of the code: 1. URL de InfluxDB (pointing to the URL in the client constructor), 2. Nombre de organización (pointing to the organization name in the client constructor), and 3. Nombre de bucket (pointing to the bucket name in the client constructor).

```
pip install influxdb-client
```

```
from influxdb_client import InfluxDBClient
from influxdb_client.client.write_api import WriteApi

# InfluxDB Cloud 2.0
client = InfluxDBClient(url="https://us-central1-1.gcp.cloud2.influxdata.com", token="...", org="boni.garcia's Bucket", bucket="boni.garcia's Bucket")
write_api = WriteApi(client)

# Example usage
write_api.write(bucket="boni.garcia's Bucket", org="boni.garcia's Bucket", data=[{"time": "2023-01-01T00:00:00Z", "value": 1}])
```

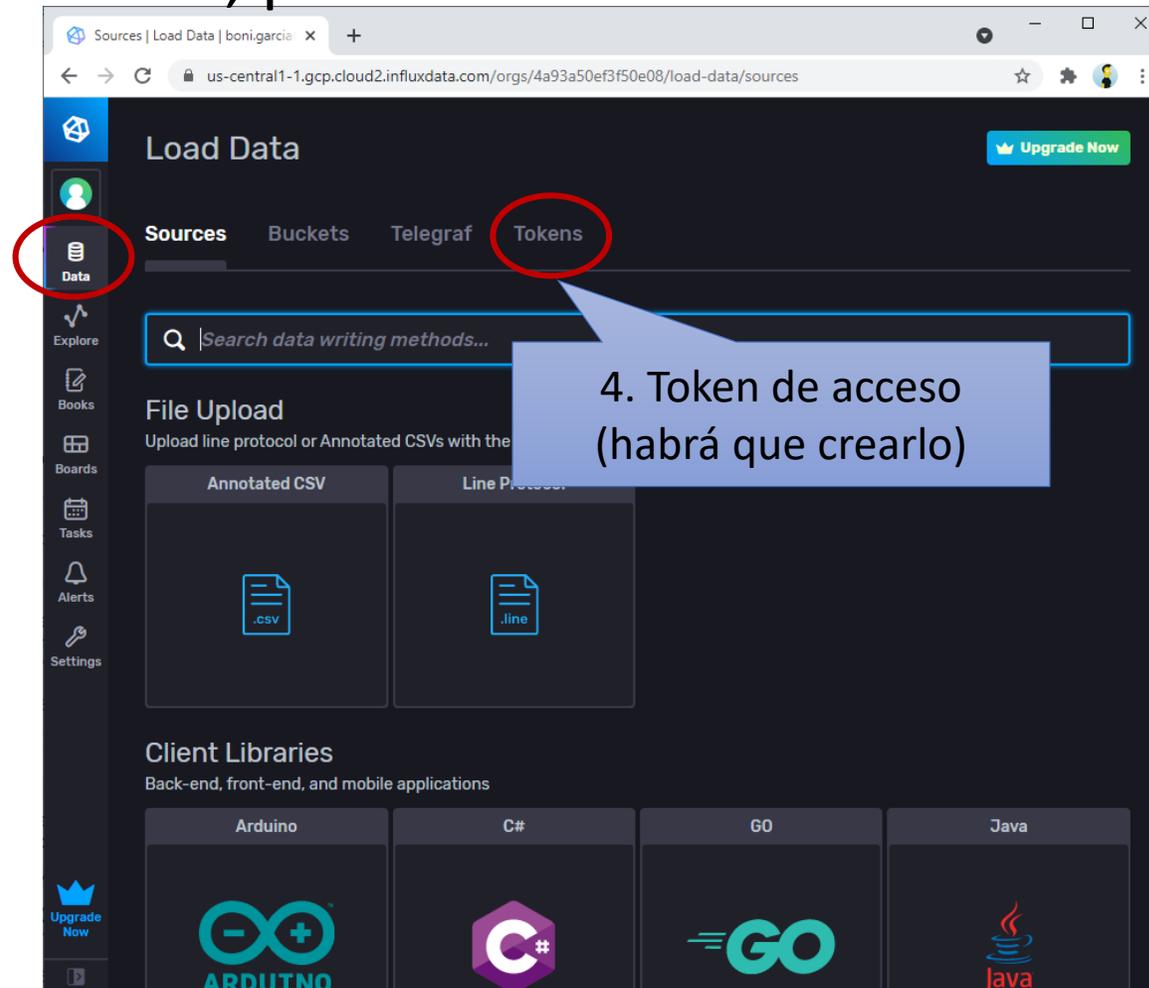
2. Nombre de organización
(por defecto es el correo con
el que nos hemos registrado)

3. Nombre de bucket

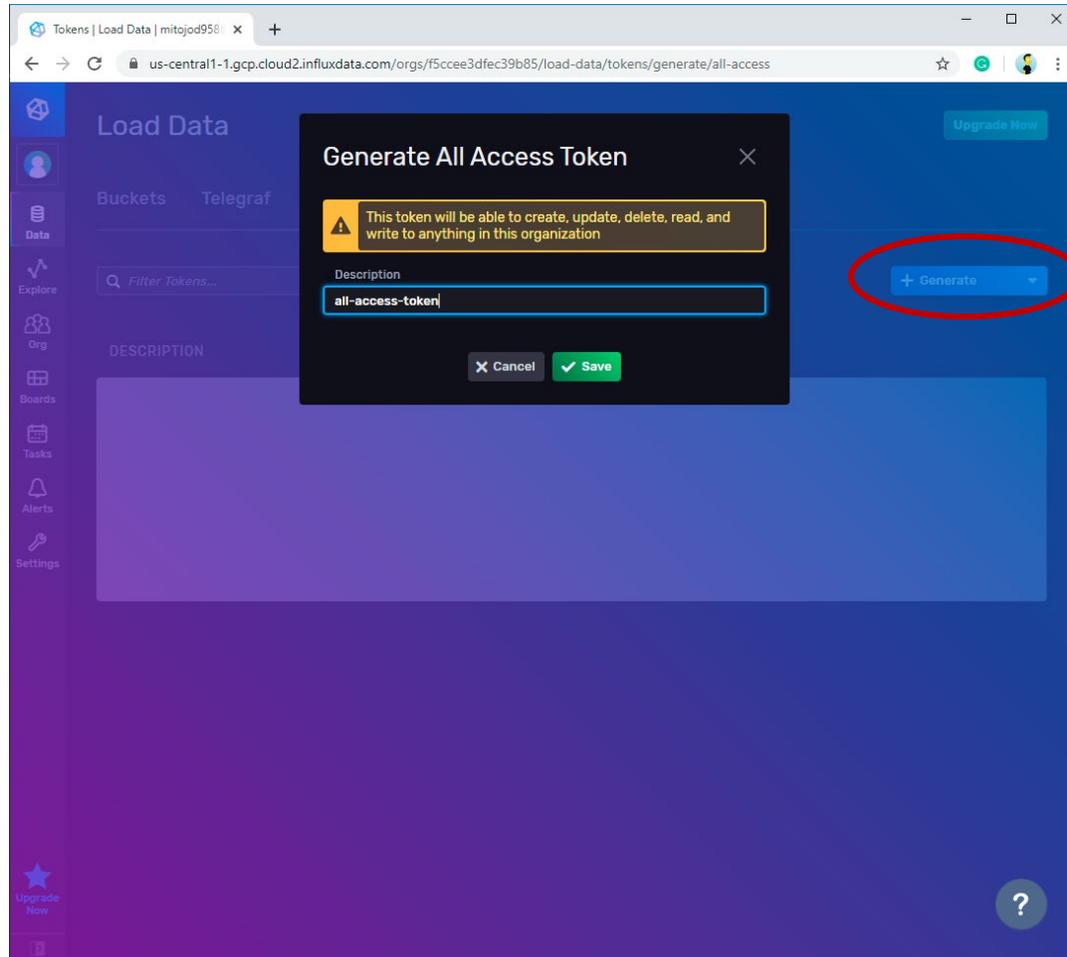
1. URL de InfluxDB

4. InfluxDB - API Python

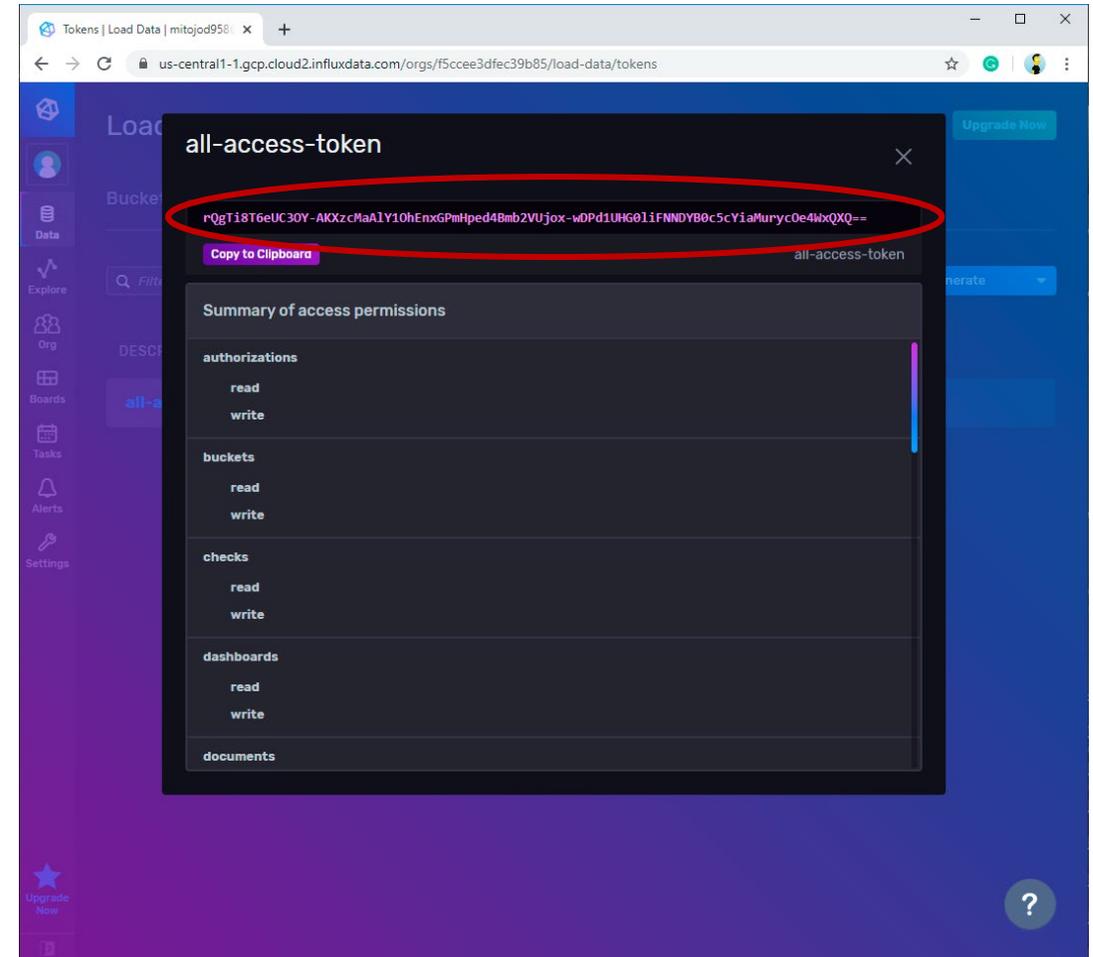
- En InfluxDB Cloud 2.0, podemos encontrar esa información en:



4. InfluxDB - API Python



The screenshot shows the InfluxDB interface with a modal dialog titled "Generate All Access Token". A warning message states: "This token will be able to create, update, delete, read, and write to anything in this organization". The "Description" field contains the text "all-access-token". A red circle highlights the "+ Generate" button. The dialog has "Cancel" and "Save" buttons at the bottom.



The screenshot shows the InfluxDB interface displaying the details of the generated "all-access-token". The token value is highlighted with a red circle: "rQgTi8T6eUC30Y-AKXzcMaAlY10hEnxGpmHped4Bmb2VUjox-wDPd1UHG01iFNNDYB0c5cYiaMuryc0e4WxQXQ==". Below the token, there is a "Copy to Clipboard" button. A "Summary of access permissions" section lists the following permissions:

Resource	Permissions
authorizations	read, write
buckets	read, write
checks	read, write
dashboards	read, write
documents	

4. InfluxDB - API Python

- Para usar la API en nuestro programa Python, en primer lugar hay que instalar la librería en nuestra máquina:

```
$ pip install influxdb-client
```

- Después, hay que crear un objeto `InfluxDBClient` usando la información anterior (URL, token, nombre de organización)

```
from influxdb_client import InfluxDBClient  
  
influxClient = InfluxDBClient(url="InfluxDB URL", org="my org", token="my token")
```

- Alternativamente, se puede crear el cliente usando la información de un fichero de configuración:

```
from influxdb_client import InfluxDBClient  
  
influxClient = InfluxDBClient.from_config_file("influxdb.ini")
```

```
[influx2]  
url=http://localhost:9999  
org=my-org  
token=my-token
```



4. InfluxDB - API Python

- Para escribir datos en InfluxDB, hay que crear objetos de tipo Point
 - Obligatoriamente, hay que especificar el nombre de la medida (`_measurement`) y uno o varios campos (clave-valor)
 - Opcionalmente, podemos especificar la marca de tiempo (si no, se usará el tiempo de la base de datos)
 - También de forma opcional, podemos especificar etiquetas (tags)
- La escritura se realiza a través hacerlo a través del método `write_api` del objeto `InfluxDBClient` previamente creado

```
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
from datetime import datetime

point = Point("measurement").field("key", value).tag("tag-key", tag-value).time(time=datetime.utcnow())
influxClient.write_api(write_options=SYNCHRONOUS).write(bucket="my bucket", org="my org", record=point)
```

4. InfluxDB - API Python

- Las consultas se realizan a través hacerlo a través del método `query_api` del objeto `InfluxDBClient` previamente creado
- Se usará el lenguaje de consultas SQL-Like llamado **Flux**:

```
from influxdb_client import InfluxDBClient

influxClient = InfluxDBClient.from_config_file("../config/influxdb.ini")
bucket="boni.garcia's Bucket"
kind="sine-wave"
myquery = f'from(bucket: "{bucket}") |> range(start: -1d) |> filter(fn: (r) => r._measurement == "{kind}")'
tables = influxClient.query_api().query(myquery)

for table in tables:
    print(table)
    for row in table.records:
        print(row.values)
```

```
FluxTable() columns: 8, records: 423
{'result': '_result', 'table': 0, '_start': datetime.datetime(2020, 4, 13, 12, 9, 37, 768982, tzinfo=datetime.timezone.utc), '_stop': datetime.datetime(2020, 4, 14, 12, 9, 37, 768982, tzinfo=datetime.timezone.utc), '_time': datetime.datetime(2020, 4, 14, 10, 20, 5, 3866, tzinfo=datetime.timezone.utc), '_value': 15.577068175668554, '_field': 'value', '_measurement': 'sine-wave'}
...
```

4. InfluxDB - Ejemplos

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode, split
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
from datetime import timezone

def saveDataFreameToInfluxDB(dataframe, batchId):
    for row in dataframe.rdd.collect():
        sinValue = float(row["value"])
        timeValue = row["timestamp"].astimezone(timezone.utc)
        print(f"Writing {sinValue} {timeValue} to InfluxDB")
        point = Point("sine-wave").field("value",
                                         sinValue).time(time=timeValue)
        influxClient.write_api(write_options=SYNCHRONOUS).write(
            bucket=bucket, record=point)

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Kafka-DataFrame-InfluxDB")
        .config("spark.jars.packages", "org.apache.spark:spark-sql-kafka-0-10_2.11:2.4.7")
        .getOrCreate())
spark.sparkContext.setLogLevel("ERROR")
```

Este ejemplo lee datos procedentes de Kafka y los envía a InfluxDB. El procesamiento de datos se hace con la API de Dataframes

```
# InfluxDB client (update this info to run this example)
influxClient = InfluxDBClient.from_config_file("../config/influxdb.ini")
bucket = "boni.garcia's Bucket"

# 1. Input data: DataFrame from Apache Kafka
df = (spark
      .readStream
      .format("kafka")
      .option("kafka.bootstrap.servers", "localhost:9092")
      .option("subscribe", "test-topic")
      .load())
df.printSchema()

# 2. Data processing: read value and timestamp
values = df.selectExpr("CAST(value AS STRING)", "timestamp")

# 3. Output data: store results in InfluxDb
query = (values
        .writeStream
        .outputMode("append")
        .foreachBatch(saveDataFreameToInfluxDB)
        .start())

query.awaitTermination()
```

4. InfluxDB - Ejemplos

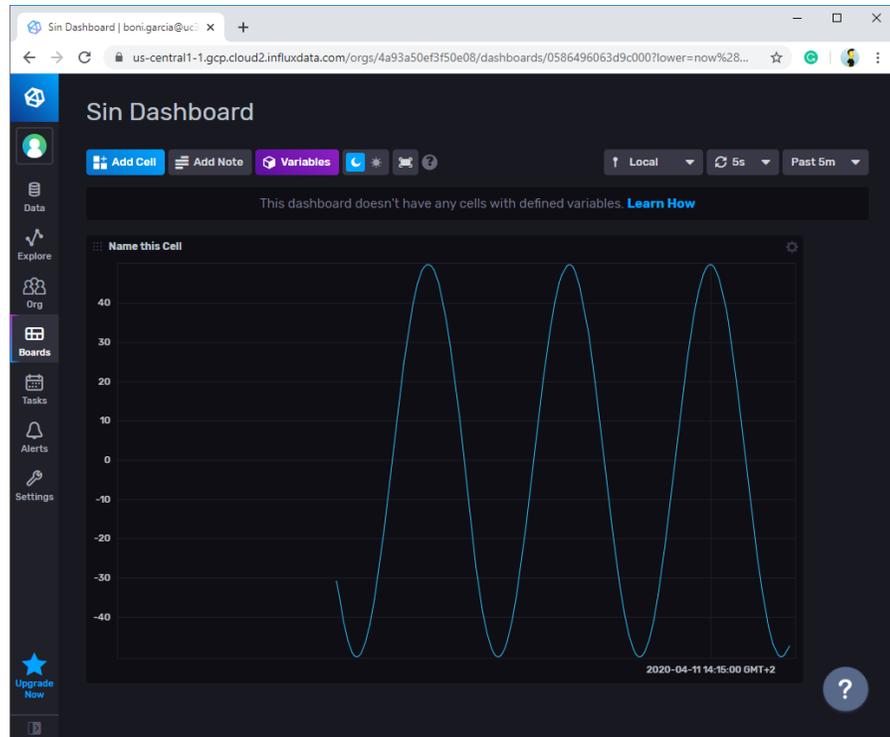
El productor de datos envía los valores de una señal sinusoidal cada segundo a Kafka

```
from kafka import KafkaProducer
from random import randrange
from math import sin
import time

producer = KafkaProducer(bootstrap_servers=["localhost:9092"])
startTime = time.time()
waitSeconds = 1.0
sin_counter = 0
sin_increment = 0.1
sin_amplitude = 50

while True:
    value = sin_amplitude * sin(sin_counter)
    print("Sending sin value to Kafka", value)
    producer.send("test-topic", str(value).encode())

    # Wait a number of second until next message
    time.sleep(waitSeconds - ((time.time() - startTime) % waitSeconds))
    sin_counter += sin_increment
```



Podemos crear un dashboard en InfluxDB y observar como evolucionan los datos (señal sinusoidal)

4. InfluxDB - Ejemplos

```
from pyspark import SparkContext, SparkConf
from pyspark.streaming.kafka import KafkaUtils
from pyspark.streaming import StreamingContext
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
from datetime import datetime
import json

def saveToInfluxDB(rdd):
    data = rdd.collect()
    for i in data:
        sinValue = float(i.get("sin"))
        timeValue = datetime.strptime(i.get("time"), "%Y-%m-%d %H:%M:%S.%f")
        print(f"Writing {sinValue} {timeValue} to InfluxDB")
        point = Point("sine-wave").field("value",
                                         sinValue).time(time=timeValue)
        influxClient.write_api(write_options=SYNCHRONOUS).write(
            bucket=bucket, record=point)

# Local SparkContext and StreamingContext
sc = SparkContext(master="local[*]",
                  appName="Kafka-DStream_SinWave-InfluxDB",
                  conf=SparkConf()
                  .set("spark.jars.packages", "org.apache.spark:spark-streaming-kafka-0-8_2.11:2.4.7"))
sc.setLogLevel("ERROR")
ssc = StreamingContext(sc, 1)
```

```
# InfluxDB client (update this info to run this example)
influxClient = InfluxDBClient.from_config_file("../config/influxdb.ini")
bucket = "boni.garcia's Bucket"

# 1. Input data: create a DStream from Apache Kafka
stream = KafkaUtils.createStream(
    ssc, "localhost:2181", "spark-streaming-consumer", {"test-topic": 1})

# 2. Data processing: get JSON values
out = (stream.map(lambda x: json.loads(x[1])))

# 3. Output data: store results in InfluxDb
out.foreachRDD(saveToInfluxDB)

ssc.start()
ssc.awaitTermination()
```

Este ejemplo es equivalente al anterior (recibe datos de Kafka y los manda a InfluxDB) pero esta vez usamos la API de DStream

4. InfluxDB - Ejemplos

Debido a la limitación en la API de DStream para leer la marca de tiempo, mandamos el tiempo (en formato UTC directamente esta vez) desde Kafka, en un mensaje en formato JSON

```
from kafka import KafkaProducer
from random import randrange
from random import randrange
from datetime import datetime
from math import sin
import time
import json

producer = KafkaProducer(bootstrap_servers=["localhost:9092"])
startTime = time.time()
waitSeconds = 1.0
sin_counter = 0
sin_increment = 0.1
sin_amplitude = 50

while True:
    time_value = str(datetime.utcnow())
    sin_value = sin_amplitude * sin(sin_counter)
    msg = {"time": time_value, "sin": sin_value}
    print("Sending JSON to Kafka", msg)
    producer.send("test-topic", json.dumps(msg).encode())

    # Wait a number of second until next message
    time.sleep(waitSeconds - ((time.time() - startTime) % waitSeconds))
    sin_counter += sin_increment
```

4. InfluxDB - Ejemplos

```

from pyspark.sql import SparkSession
from influxdb_client import InfluxDBClient, Point
from influxdb_client.client.write_api import SYNCHRONOUS
from pyspark.sql.functions import udf
from pyspark.sql.types import FloatType
from datetime import timezone

def triangle(x, phase, length, amplitude):
    alpha = (amplitude)/(length/2)
    return -amplitude/2+amplitude*((x-phase) % length == length/2) \
        + alpha*((x-phase) % (length/2))*((x-phase) % length <= length/2) \
        + (amplitude-alpha*((x-phase) % (length/2))) * \
        ((x-phase) % length > length/2)

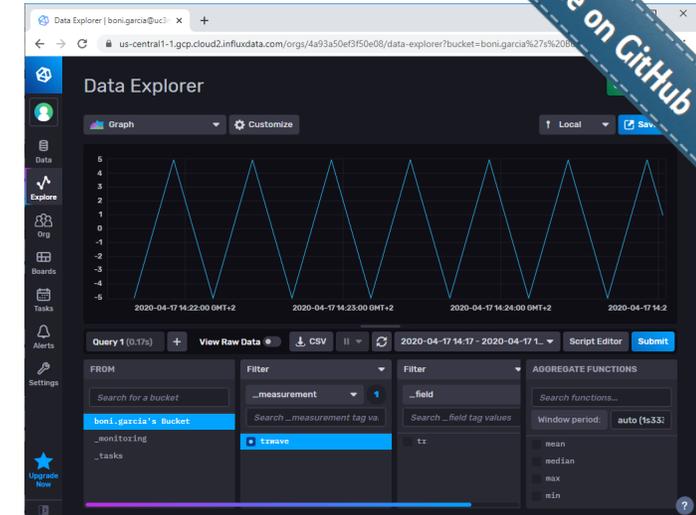
def saveDataFrameToInfluxDB(dataframe, batchId):
    for row in dataframe.rdd.collect():
        tr = row["tr"]
        ts = row["timestamp"].astimezone(timezone.utc)
        print(f"Writing {tr} to InfluxDB (timestamp {ts})")
        point = Point("trwave").field("tr", tr).time(time=ts)
        influxClient.write_api(write_options=SYNCHRONOUS).write(
            bucket=bucket, record=point)

# Local SparkSession
spark = (SparkSession
        .builder
        .master("local[*]")
        .appName("Rate-DataFrame-InfluxDB")
        .getOrCreate())
spark.sparkContext.setLogLevel("ERROR")

# InfluxDB client (update this info to run this example)
influxClient = InfluxDBClient.from_config_file("../config/influxdb.ini")
bucket = "boni.garcia's Bucket"
org = "boni.garcia@uc3m.es"

```

Este ejemplo vuelve a hacer uso de la API DataFrames, realizando un procesamiento de los datos recibidos antes de mandarlos a InfluxDB



```

# 1. Input data: test DataFrame with sequence and timestamp
df = (spark
     .readStream
     .format("rate")
     .option("rowsPerSecond", 1)
     .load())

# 2. Data processing: add new column with the value of a triangle wave
trwave = udf(lambda x: triangle(x, 0, 30, 10), FloatType())
triangleDf = df.withColumn("tr", trwave(df["value"]))

# 3. Output data: show results in the console
query = (triangleDf
        .writeStream
        .outputMode("append")
        .foreachBatch(saveDataFrameToInfluxDB)
        .start())

query.awaitTermination()

```

Este ejemplo transforma una secuencia en una señal de forma triangular

Fork me on GitHub

Contenidos

1. Introducción
2. Bases de datos
3. Cassandra
4. InfluxDB
5. Resumen

5. Resumen

- Los **resultados** obtenidos del procesamiento en streaming con Spark pueden ser entregados a **sistemas de salida** (downstream systems)
- Un sistema habitual de sistema de salida son las bases de datos (DBMS), que pueden ser **relacionales** o **NoSQL**
- **Apache Cassandra** es un DBMS NoSQL que proporciona alta disponibilidad de datos y se puede usar en Spark mediante un **conector PySpark** y a través de la API de **DataFrames**
- **InfluxDB** es un DBMS NoSQL de **serie temporal** (TSBD) que proporciona mecanismos de almacenamiento, visualización, y alertas para series temporales (datos con marca de tiempo). No existe conector nativo Spark, pero se puede usar a través de su API Python