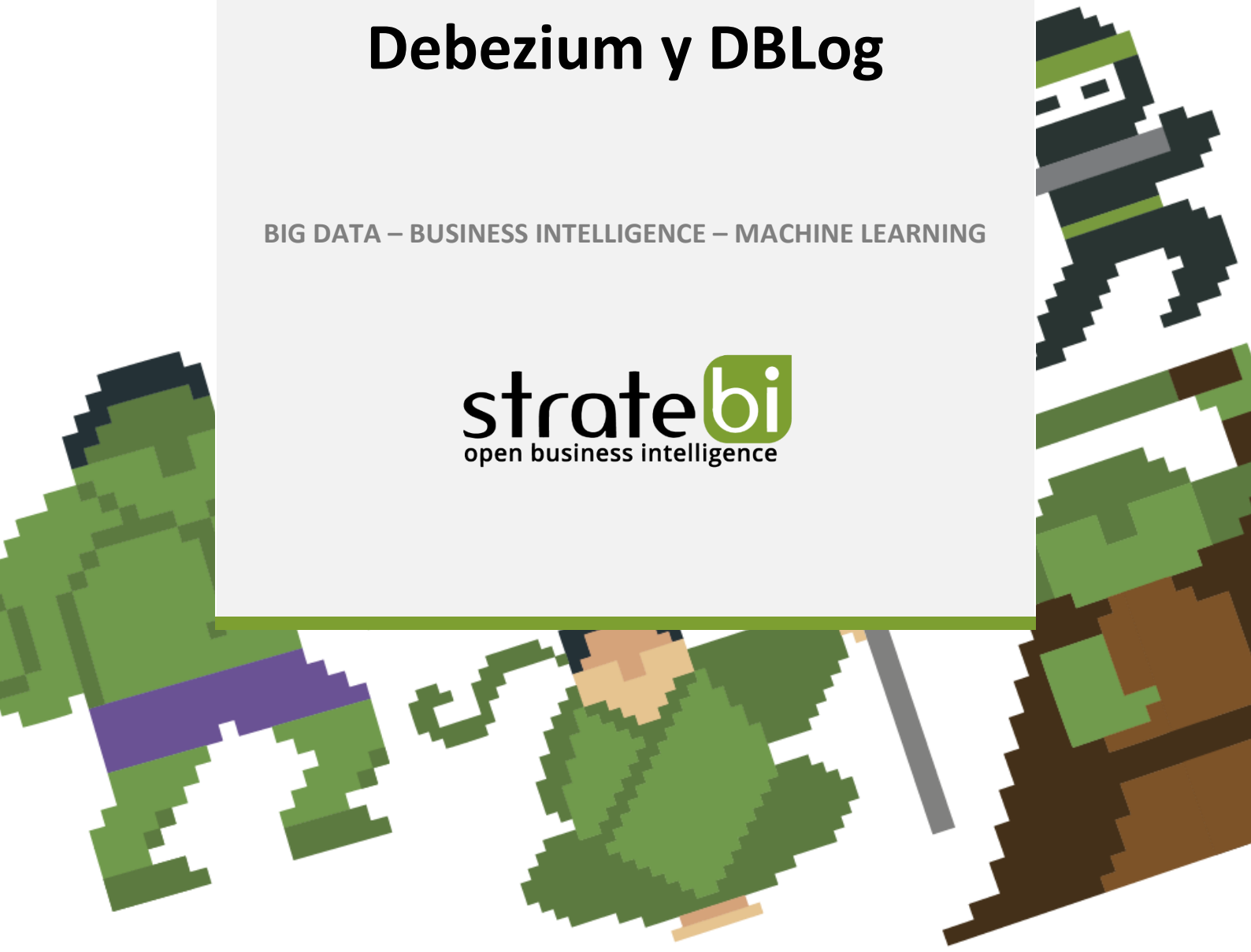
Herramientas Change-Data-Capture: Debezium y DBLog

BIG DATA – BUSINESS INTELLIGENCE – MACHINE LEARNING



1. INTODUCCIÓN A LAS CDC

En bases de datos, las CDC (Change Data Capture) son patrones de diseño software que se emplean para capturar cambios que se producen en los datos y propagarlos a clientes intermedios. Normalmente se emplean en entornos de data-warehouse, para preservar el estado de los datos a lo largo del tiempo, o se emplean también en soluciones en las que hay que mantener un conjunto de bases de datos heterogéneo, ya que las CDC producen el mismo formato de salida independientemente de cuál sea la base de datos origen del registro.

Existen varias soluciones de CDC: Maxwell, SpinalTap, Yelp's MySQL Streamer, Debezium, DBLog…

Nos centraremos en una de estas soluciones, que a nuestro punto de vista es la más simple de utilizar: Debezium.

2. DEBEZIUM: INTRODUCCIÓN

Debezium es una herramienta que realiza funciones de CDC (Change Data Capture) proporcionando una plataforma de streaming de datos de baja latencia para capturar cambios en los datos. Reutiliza las herramientas Kafka y Kafka Connect, con lo que consigue agregar varias conexiones de bases de datos y que todos los eventos queden bien ordenados para que varios clientes puedan consumirlos sin afectar demasiado la captura de los eventos. Posee las características comunes de una herramienta CDC. Entre ellas, destaca:

- Puede monitorizar un conjunto heterogéneo de bases de datos, produciendo una salida estandarizada independientemente de la base de datos en la que se produjo el cambio, lo que facilita bastante la administración y manejo de los cambios.
- Debezium es tolerante a fallos: si por algún motivo Debezium se detuviera, al reiniciarse registrará los cambios que se produjeron mientras estaba apagado para asegurar que todos los eventos se registran y procesan adecuadamente. Adicionalmente, agrega esta característica pero para el lado del cliente, es decir, si un cliente se desconecta del servicio, cuando se vuelva a conectar recibirá todos los eventos que sucedieron mientras estaba desconectado.
- Debezium proporciona multitud de módulos para conexión con bases de datos. Algunos de ellos son genéricos, para soportar cualquier base de datos, con la desventaja de que son algo limitados; y otros son específicos de algunas bases de datos. En concreto, Debezium proporciona conectores con bases de datos MySQL, PostgreSQL, MongoDB y SQL Server. Además, se está desarrollando conectores para Oracle, Cassandra y Db2. Estos tres últimos se pueden emplear pero al estar en desarrollo, podrían estar sujetos a cambios.

3. DEBEZIUM: REQUISITOS DE INSTALACIÓN

Para poder usar Debezium, es necesario que la máquina cuente con las siguientes herramientas:

- Git 2.2.1 o superior.
- JDK 8 o OpenJDK 8.
- Maven 3.2.1 o superior.
- Docker Engine 1.9 o superior.

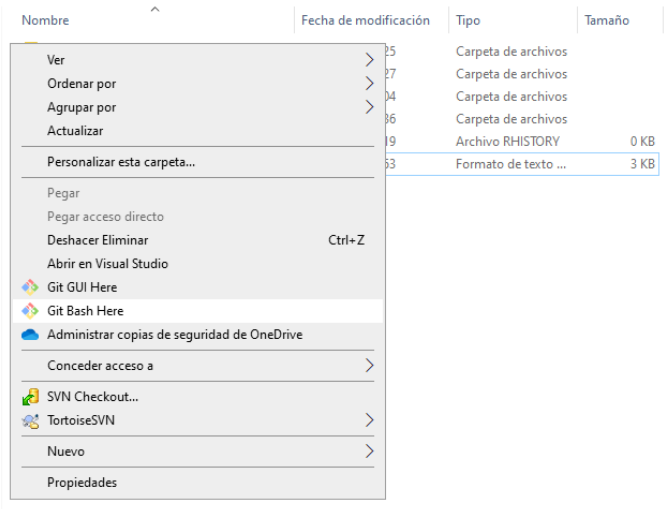
Debezium posee imágenes de Kafka, Kafka Connector, Zookeeper, los conectores de bases de datos y algunas imágenes de bases de datos para hacer pruebas, así que no es necesario instalar nada más.

Para verificar que se tienen esas herramientas, se pueden emplear los siguientes comandos:

```
git --version
javac -version
mvn -version
docker –version
```

4. DEBEZIUM: INSTALACIÓN (WINDOWS)

Para instalar Debezium, simplemente hay que descargar el código de GitHub. Para ello, sitúese en el directorio en el que quiera que esté el directorio de Debezium, haga click derecho sobre la pantalla y use la opción: “Git Bash Here”



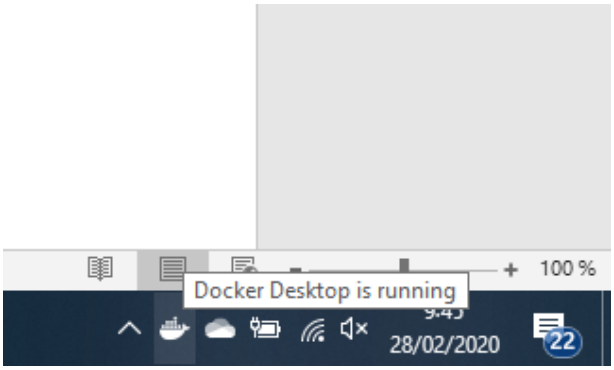
En la consola de comandos que se despliega, introduzca el siguiente comando:

```
git clone https://github.com/debezium/debezium.git
```

Y el código se le descargará automáticamente.

5. DEBEZIUM: EJEMPLO DE USO CON MYSQL (WINDOWS)

Antes de empezar el ejemplo, asegúrese de que Docker Engine (Docker Desktop en el caso de Windows) está ejecutándose. En este caso que se hará en Windows, debería ver en la barra inferior, en la sección derecha, un icono como el siguiente:



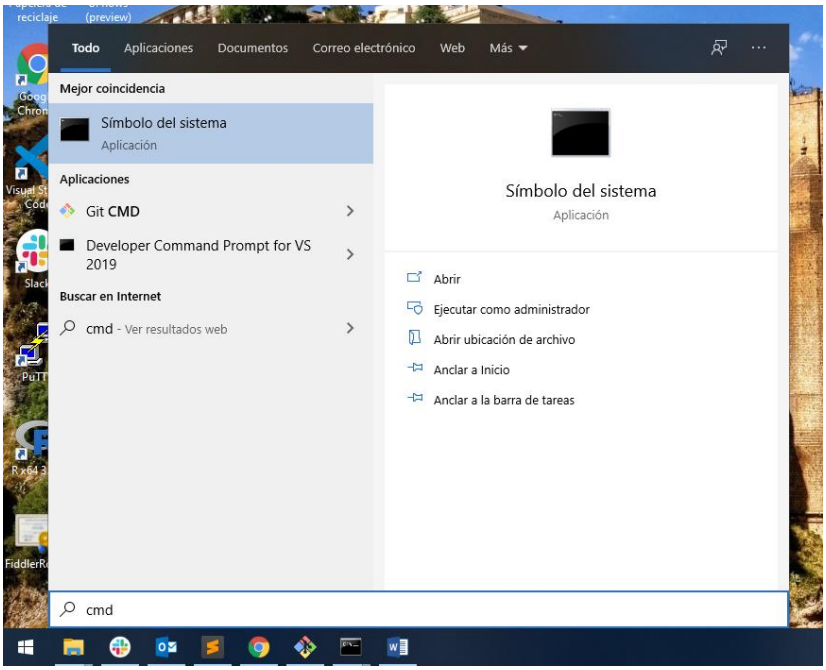
Si no encuentra el icono, arranque Docker Engine buscando el programa en el Menú de Windows o desde el acceso directo del Escritorio.

Para poder ejecutar Debezium, es necesario activar tres plataformas, que se montarán sobre tres dockers: Zookeeper, Kafka y Kafka Connect. Además, para este ejemplo se usará una de las imágenes de base de datos de ejemplo que ofrece Debezium. En este caso, una imagen de MySQL server.

Nota: Zookeeper y Kafka pueden almacenar datos en directorios que se pueden configurar, para que en el caso de que haya algún problema y dejen de funcionar, los datos no se pierdan. Para este ejemplo se va a saltar esta configuración para facilitar la limpieza al final. Puede encontrar cómo configurar la persistencia de los datos en la documentación de la herramienta.
También se va a arrancar cada herramienta en una terminal diferente y en primer plano, para poder comprobar los logs que producen y así se puede asegurar que todo está funcionando correctamente.

1. Arrancar Zookeeper

Es el primer servicio que se tiene que arrancar. Para ello, abra una terminal de Windows:



Y ejecute el siguiente comando:

```
docker run -it --rm --name zookeeper -p 2181:2181 -p 2888:2888 -p 3888:3888
debezium/zookeeper:1.0
```

Con esto, se empezará a descargar la imagen de Zookeeper. Debería ver algo como:

```
C:\Users\Stratebi>docker run -it --rm --name zookeeper -p 2181:2181 -p 2888:2888 -p 38
Unable to find image 'debezium/zookeeper:1.0' locally
1.0: Pulling from debezium/zookeeper
f34b00c7da20: Pull complete
4bb2f52c2c3b: Pull complete
3bc039d0006b: Pull complete
bb003d07fcc4: Pull complete
26a042752bd9: Pull complete
16de8bfbf6e6: Pull complete
afcbed2e2469: Pull complete
0cce5f7683f3: Pull complete
09e45e889a43: Pull complete
9082c7518a18: Pull complete
523ff12e9ea8: Pull complete
e72d7e388cb1: Pull complete
4ca4a5d2bf73: Pull complete
```

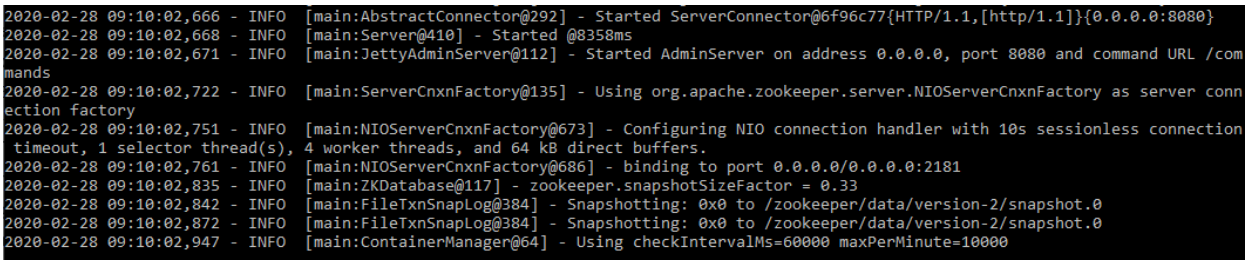
Nota importante: si alguna de las descargas parece haberse quedado colgada (si espera más de 2 minutos y no ha cambiado nada en la terminal y no aparecen todos los mensajes como “Pull complete”), presione la tecla Enter varias veces en la terminal hasta que vuelva a ejecutar.

Puede comprobar que se haya ejecutado correctamente buscando los siguientes logs:

```
Starting up in standalone mode
ZooKeeper JMX enabled by default
Using config: /zookeeper/conf/zoo.cfg
2017-09-21 07:15:55,417 - INFO  [main:QuorumPeerConfig@134] - Reading
configuration from: /zookeeper/conf/zoo.cfg
2017-09-21 07:15:55,419 - INFO  [main:DatadirCleanupManager@78] -
autopurge.snapRetainCount set to 3
2017-09-21 07:15:55,419 - INFO  [main:DatadirCleanupManager@79] -
autopurge.purgeInterval set to 1
...
port 0.0.0.0/0.0.0.0:2181
```

(Los puntos suspensivos indican que puede haber varios logs entre medias)

En nuestro caso, el log final es el siguiente:



Con esto, Zookeeper estará conectado y escuchando en el puerto 2181.

2. Arrancar Kafka

Una vez arrancado Zookeeper, arranque Kafka abriendo una nueva consola de comandos e introduciendo el siguiente:

```
docker run -it --rm --name kafka -p 9092:9092 --link zookeeper:zookeeper
debezium/kafka:1.0
```

Al igual que en Zookeeper, y en el resto de servicios que vamos a desplegar, se van a descargar los ficheros de la imagen.

Nota: En nuestro caso, en esta instalación tres de los ficheros dejaron la descarga atascada. Para solucionar esto si le pasa, recuerde pulsar la tecla Enter varias veces en la consola.

Sabrá que el arranque salió bien si consigue localizar los siguientes logs:

```
...
2017-09-21 07:16:59,085 - INFO  [main-EventThread:ZkClient@713] - zookeeper
state changed (SyncConnected)
2017-09-21 07:16:59,218 - INFO  [main:Logging$class@70] - Cluster ID =
LPtcBFxzRvOzDSXhc6AamA
...
2017-09-21 07:16:59,649 - INFO  [main:Logging$class@70] - [Kafka Server 1],
started
```

En nuestro caso, los últimos logs son los siguientes:

```
2020-02-27 10:48:34,378 - INFO  [ExpirationReaper-1-Rebalance:Logging@66] - [ExpirationReaper-1-Rebalance]: Starting
2020-02-27 10:48:34,382 - INFO  [ExpirationReaper-1-Heartbeat:Logging@66] - [ExpirationReaper-1-Heartbeat]: Starting
2020-02-27 10:48:34,551 - INFO  [controller-event-thread:Logging@66] - Successfully created /controller_epoch with initi
al epoch 0
2020-02-27 10:48:34,665 - INFO  [main:Logging@66] - [GroupCoordinator 1]: Starting up.
2020-02-27 10:48:34,697 - INFO  [main:Logging@66] - [GroupCoordinator 1]: Startup complete.
2020-02-27 10:48:34,779 - INFO  [group-metadata-manager-0:Logging@66] - [GroupMetadataManager brokerId=1] Removed 0 expi
red offsets in 79 milliseconds.
2020-02-27 10:48:35,046 - INFO  [main:Logging@66] - [ProducerId Manager 1]: Acquired new producerId block (brokerId:1,bl
ockStartProducerId:0,blockEndProducerId:999) by writing to Zk with path version 1
2020-02-27 10:48:35,352 - INFO  [main:Logging@66] - [TransactionCoordinator id=1] Starting up.
2020-02-27 10:48:35,359 - INFO  [main:Logging@66] - [TransactionCoordinator id=1] Startup complete.
2020-02-27 10:48:35,407 - INFO  [TxnMarkerSenderThread-1:Logging@66] - [Transaction Marker Channel Manager 1]: Starting
2020-02-27 10:48:35,635 - INFO  [ExpirationReaper-1-AlterAcls:Logging@66] - [ExpirationReaper-1-AlterAcls]: Starting
2020-02-27 10:48:35,779 - INFO  [/config/changes-event-process-thread:Logging@66] - [/config/changes-event-process-threa
d]: Starting
2020-02-27 10:48:35,984 - INFO  [main:Logging@66] - [SocketServer brokerId=1] Started data-plane processors for 1 accept
ors
2020-02-27 10:48:36,043 - INFO  [main:AppInfoParser$AppInfo@117] - Kafka version: 2.4.0
2020-02-27 10:48:36,080 - INFO  [main:AppInfoParser$AppInfo@118] - Kafka commitId: 77a89fcf8d7fa018
2020-02-27 10:48:36,084 - INFO  [main:AppInfoParser$AppInfo@119] - Kafka startTimeMs: 1582800516002
2020-02-27 10:48:36,109 - INFO  [main:Logging@66] - [KafkaServer id=1] started
```

3. Arrancar MySQL

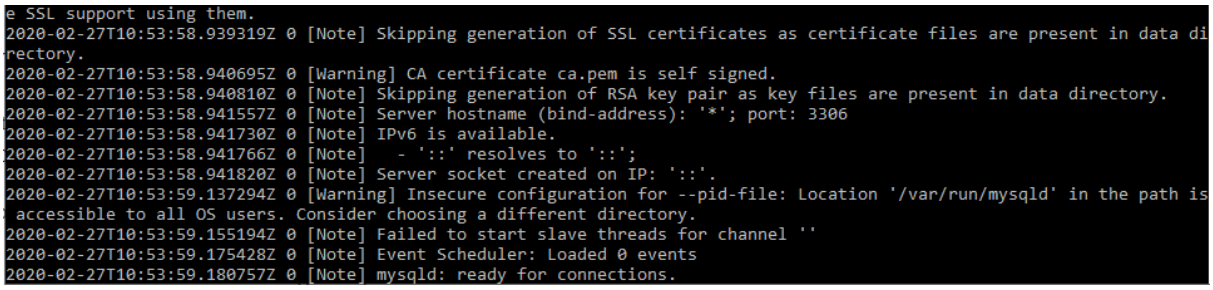
Una vez arrancados Zookeeper y Kafka, vamos a arrancar un servidor con una base de datos desde donde Debezium pueda capturar los cambios. Abra una terminal nueva y use el siguiente comando:

```
docker run -it --rm --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=debezium
-e MYSQL_USER=mysqluser -e MYSQL_PASSWORD=mysqlpw debezium/example-mysql:1.0
```

Puede comprobar que el arranque fue bien si encuentra estos logs al final:

```
...
017-09-21T07:18:50.824629Z 0 [Note] mysqld: ready for connections.
Version: '5.7.19-log'  socket: '/var/run/mysqld/mysqld.sock'  port: 3306
MySQL Community Server (GPL)
```

En nuestro caso, los logs finales son los siguientes:



4. Arrancar un cliente de comandos SQL

Para poder manejar y hacer cambios y consultas sobre la base de datos que acabamos de crear, vamos a arrancar un cliente SQL.

Abra una consola de comandos nueva y ejecute el siguiente comando:

```
docker run -it --rm --name mysqlterm --link mysql --rm mysql:5.7 sh -c 'exec
mysql -h"$MYSQL_PORT_3306_TCP_ADDR" -P"$MYSQL_PORT_3306_TCP_PORT" -uroot -
p"$MYSQL_ENV_MYSQL_ROOT_PASSWORD"'
```

ATENCION: Este comando es el oficial de la documentación de la herramienta, pero a nosotros no nos funciona en Windows. Si le pasa lo mismo, use el siguiente:

```
docker run -it --rm --name mysqlterm --link mysql --rm mysql:5.7 sh -c "exec
mysql -h$MYSQL_PORT_3306_TCP_ADDR -P$MYSQL_PORT_3306_TCP_PORT -uroot -
p$MYSQL_ENV_MYSQL_ROOT_PASSWORD"
```

Puede comprobar que se ha ejecutado correctamente si ve logs como los siguientes:

```
mysql: [Warning] Using a password on the command line interface can be
insecure.
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.17-log MySQL Community Server (GPL)

Copyright (c) 2000, 2016, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input
statement.
```

Y el prompt de la consola debería haberle cambiado a el siguiente:

```
mysql>
```

En el caso del ejemplo, se va a trabajar con una base de datos llamada “inventory”, en específico se va a trabajar con la tabla “customers”.

Seleccione la base de datos “inventory” con el siguiente comando:

```
use inventory;
```

Que debería devolver:

```
Database changed
```

Puede hacer unas primeras consultas para ver las tablas disponibles en la base de datos y para ver el contenido de la tabla que vamos a usar:

```
show tables;
```

Que debería devolver:

```
+----------------------+
| Tables_in_inventory |
+----------------------+
| addresses           |
| customers           |
| geom                |
| orders              |
| products            |
| products_on_hand    |
+----------------------+
6 rows in set (0.00 sec)
```

```
SELECT * FROM customers;
```

Que debería devolver:

```
+------+------------+-----------+-----------------------+
| id   | first_name | last_name | email                 |
+------+------------+-----------+-----------------------+
| 1001 | Sally      | Thomas    | sally.thomas@acme.com |
| 1002 | George     | Bailey    | gbailey@foobar.com    |
| 1003 | Edward     | Walker    | ed@walker.com         |
| 1004 | Anne       | Kretchmar | annek@noanswer.org    |
+------+------------+-----------+-----------------------+
```

```
4 rows in set (0.00 sec)
```

5. Arrancar Kafka Connect

Vamos a desplegar un servicio que ofrecerá una Api REST que nos permitirá introducir el conector de Debezium para MySQL.

Abra otra consola de comandos y ejecute el siguiente:

```
docker run -it --rm --name connect -p 8083:8083 -e GROUP_ID=1 -e
CONFIG_STORAGE_TOPIC=my_connect_configs -e
OFFSET_STORAGE_TOPIC=my_connect_offsets -e
STATUS_STORAGE_TOPIC=my_connect_statuses --link zookeeper:zookeeper --link
kafka:kafka --link mysql:mysql debezium/connect:1.0
```

El arranque habrá sido correcto si puede ver unos logs como los siguientes:

```
...
2020-02-06 15:48:33,939 INFO   ||  Kafka version: 2.4.0
[org.apache.kafka.common.utils.AppInfoParser]
...
2020-02-06 15:48:34,485 INFO   ||  [Worker clientId=connect-1, groupId=1]
Starting connectors and tasks using config offset -1
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2020-02-06 15:48:34,485 INFO   ||  [Worker clientId=connect-1, groupId=1]
Finished starting connectors and tasks
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
```

En nuestro caso, los últimos logs son los siguientes:

```
2020-02-27 11:21:19,200 INFO   ||  [Worker clientId=connect-1, groupId=1] (Re-)joining group   [org.apache.kafka.clients
.consumer.internals.AbstractCoordinator]
2020-02-27 11:21:19,680 INFO   ||  [Worker clientId=connect-1, groupId=1] Successfully joined group with generation 1
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2020-02-27 11:21:19,688 INFO   ||  [Worker clientId=connect-1, groupId=1] Joined group at generation 1 with protocol ver
sion 2 and got assignment: Assignment{error=0, leader='connect-1-8e5b6101-0e08-4ae6-bdba-f8b82f3b9fb9', leaderUrl='http:
//172.17.0.6:8083/', offset=-1, connectorIds=[], taskIds=[], revokedConnectorIds=[], revokedTaskIds=[], delay=0} with re
balance delay: 0   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2020-02-27 11:21:19,697 INFO   ||  [Worker clientId=connect-1, groupId=1] Starting connectors and tasks using config off
set -1   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2020-02-27 11:21:19,699 INFO   ||  [Worker clientId=connect-1, groupId=1] Finished starting connectors and tasks   [org.
apache.kafka.connect.runtime.distributed.DistributedHerder]
2020-02-27 11:21:20,169 INFO   ||  [Worker clientId=connect-1, groupId=1] Session key updated   [org.apache.kafka.connec
t.runtime.distributed.DistributedHerder]
```

Puede comprobar también que esté bien desplegado abriendo otra terminal y ejecutando el siguiente comando:

```
curl -H "Accept:application/json" localhost:8083/
```

Debería devolver algo del estilo:

```
{"version":"2.4.0","commit":"77a89fcf8d7fa018","kafka_cluster_id":"SP1wTIUoS4
q9ttSRT18jDA"}
```

6. Desplegar el conector MySQL

Para desplegar el conector, vamos a hacer una petición HTTP a Kafka Connect enviando la configuración del mismo. La configuración que se va a enviar es la siguiente:

```
{
  "name": "inventory-connector",
  "config": {
    "connector.class": "io.debezium.connector.mysql.MySqlConnector",
    "tasks.max": "1",
    "database.hostname": "mysql",
    "database.port": "3306",
    "database.user": "debezium",
    "database.password": "dbz",
    "database.server.id": "184054",
    "database.server.name": "dbserver1",
    "database.whitelist": "inventory",
    "database.history.kafka.bootstrap.servers": "kafka:9092",
    "database.history.kafka.topic": "schema-changes.inventory"
  }
```

Puede encontrar información específica sobre la configuración en la documentación de la herramienta, pero podemos destacar los siguientes apartados:

- .**hostname**: indica el nombre del Docker en el que se encuentra el servidor de MySQL.
- **.port, .user, .password**: el puerto en el que se encuentra MySQL y el usuario y contraseña que se van a usar para acceder.
- **.whitelist**: las bases de datos del servidor MySQL de las que se van a capturar cambios.
- **.history.kafka.bootstrap.servers**: el nombre del Docker en el que está Kafka y el puerto por donde escucha.
- **.history.kafka.topic**: el nombre que le dará Kafka Connector al topic de los logs para este conector.

Para realizar la petición y así registrar el conector, puede usar la misma consola de comandos en la que ejecutó el comando "curl" para comprobar si Kafka Connect había sido desplegado correctamente. Introduzca el siguiente comando:

```
curl -i -X POST -H "Accept:application/json" -H "Content-
Type:application/json" localhost:8083/connectors/ -d '{ \"name\":
\"inventory-connector\", \"config\": { \"connector.class\":
\"io.debezium.connector.mysql.MySqlConnector\", \"tasks.max\": \"1\",
\"database.hostname\": \"mysql\", \"database.port\": \"3306\",
\"database.user\": \"debezium\", \"database.password\": \"dbz\",
\"database.server.id\": \"184054\", \"database.server.name\": \"dbserver1\",
\"database.whitelist\": \"inventory\",
\"database.history.kafka.bootstrap.servers\": \"kafka:9092\",
\"database.history.kafka.topic\": \"dbhistory.inventory\" } }'
```

ATENCION: Este comando es el oficial de la documentación de la herramienta, pero tampoco nos funcionaba en Windows. Si le pasa lo mismo, use el siguiente:

```
curl -i -X POST -H "Accept:application/json" -H "Content-
Type:application/json" localhost:8083/connectors/ -d "{ \"name\":
\"inventory-connector\", \"config\": { \"connector.class\":
\"io.debezium.connector.mysql.MySqlConnector\", \"tasks.max\": \"1\",
\"database.hostname\": \"mysql\", \"database.port\": \"3306\",
\"database.user\": \"debezium\", \"database.password\": \"dbz\",
\"database.server.id\": \"184054\", \"database.server.name\": \"dbserver1\",
\"database.whitelist\": \"inventory\",
\"database.history.kafka.bootstrap.servers\": \"kafka:9092\",
\"database.history.kafka.topic\": \"dbhistory.inventory\" } }"
```

Deben pasar dos cosas: por una parte, debería devolver en esa terminal algo parecido al siguiente log:

```
HTTP/1.1 201 Created
Date: Thu, 27 Feb 2020 11:35:10 GMT
Location: http://localhost:8083/connectors/inventory-connector
Content-Type: application/json
Content-Length: 487
Server: Jetty(9.4.20.v20190813)

{"name":"inventory-
connector","config":{"connector.class":"io.debezium.connector.mysql.MySqlConn
ector","tasks.max":"1","database.hostname":"mysql","database.port":"3306","da
tabase.user":"debezium","database.password":"dbz","database.server.id":"18405
4","database.server.name":"dbserver1","database.whitelist":"inventory","datab
ase.history.kafka.bootstrap.servers":"kafka:9092","database.history.kafka.top
ic":"dbhistory.inventory","name":"inventory-
connector"},"tasks":[],"type":"source"}
```

Compruebe ahora que se ha registrado bien escribiendo los siguientes comandos:

```
curl -H "Accept:application/json" localhost:8083/connectors/
```

Que debería devolver:

```
["inventory-connector"]
```

Y:

```
curl -i -X GET -H "Accept:application/json"
localhost:8083/connectors/inventory-connector
```

Que debería un log parecido a:

```
HTTP/1.1 200 OK
Date: Thu, 27 Feb 2020 11:35:52 GMT
Content-Type: application/json
Content-Length: 531
Server: Jetty(9.4.20.v20190813)

{"name":"inventory-
connector","config":{"connector.class":"io.debezium.connector.mysql.MySqlConn
ector","database.user":"debezium","database.server.id":"184054","tasks.max":"
1","database.hostname":"mysql","database.password":"dbz","database.history.ka
fka.bootstrap.servers":"kafka:9092","database.history.kafka.topic":"dbhistory
.inventory","name":"inventory-
connector","database.server.name":"dbserver1","database.whitelist":"inventory
","database.port":"3306"},"tasks":[{"connector":"inventory-
connector","task":0}],"type":"source"}
```

Por otro lado, la consola donde se arrancó Kafka Connect ha debido generar mucho log correspondiente a la conexión del nuevo conector. Intente localizar el último log de cuando lo arrancó, y a partir de ahí debería encontrar el siguiente log:

```
...
2017-09-21 07:23:59,051 INFO   ||  Connector inventory-connector config
updated   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  Rebalance started
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  Finished stopping tasks in preparation for
rebalance   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,550 INFO   ||  (Re-)joining group 1
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:23:59,556 INFO   ||  Successfully joined group 1 with
generation 2
[org.apache.kafka.clients.consumer.internals.AbstractCoordinator]
2017-09-21 07:23:59,556 INFO   ||  Joined group and got assignment:
Assignment{error=0, leader='connect-1-4d60cb71-cb93-4388-8908-6f0d299a9d94',
```

```
leaderUrl='http://172.17.0.7:9092/', offset=1, connectorIds=[inventory-
connector], taskIds=[]}
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,557 INFO   ||  Starting connectors and tasks using config
offset 1   [org.apache.kafka.connect.runtime.distributed.DistributedHerder]
2017-09-21 07:23:59,557 INFO   ||  Starting connector inventory-connector
[org.apache.kafka.connect.runtime.distributed.DistributedHerder]
...
```

Que indica que ha arrancado el conector. Luego, más abajo, debería encontrar:

```
...
2017-09-21 07:24:01,151 INFO   MySQL|dbserver1|task  Kafka version : 0.11.0.0
[org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:24:01,151 INFO   MySQL|dbserver1|task  Kafka commitId :
cb8625948210849f   [org.apache.kafka.common.utils.AppInfoParser]
2017-09-21 07:24:01,584 INFO   MySQL|dbserver1|task  Found no existing
offset, so preparing to perform a snapshot
[io.debezium.connector.mysql.MySqlConnectorTask]
2017-09-21 07:24:01,614 INFO   ||  Source task WorkerSourceTask{id=inventory-
connector-0} finished initialization and start
[org.apache.kafka.connect.runtime.WorkerSourceTask]
2017-09-21 07:24:01,615 INFO   MySQL|dbserver1|snapshot  Starting snapshot
for
jdbc:mysql://mysql:3306/?useInformationSchema=true&nullCatalogMeansCurrent=fa
lse&useSSL=false&useUnicode=true&characterEncoding=UTF-
8&characterSetResults=UTF-8&zeroDateTimeBehavior=convertToNull with user
'debezium'   [io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,617 INFO   MySQL|dbserver1|snapshot  Snapshot is using
user 'debezium' with these MySQL grants:
[io.debezium.connector.mysql.SnapshotReader]
2017-09-21 07:24:01,618 INFO   MySQL|dbserver1|snapshot         GRANT SELECT,
RELOAD, SHOW DATABASES, REPLICATION SLAVE, REPLICATION CLIENT ON *.* TO
'debezium'@'%'   [io.debezium.connector.mysql.SnapshotReader]
...
```

Esto es log que genera el propio conector. Entre las acciones que realiza, en la penúltima línea indica que se va a conectar a la base de datos con ese usuario y con los permisos que vienen en la línea de después.

Nota: Si el conector no se conecta o no puede ver tablas de la base de datos, compruebe esos permisos. Con este ejemplo no debería tener ningún problema.

Tras esto, el conector escribe la configuración del servidor MySQL al que se ha conectado. Entre todas las líneas, destaca la siguiente:

```
 2017-09-21 07:24:01,626 INFO MySQL|dbserver1|snapshot binlog_format
= ROW [io.debezium.connector.mysql.SnapshotReader]
```

Tras eso, el conector escribe nueve steps:

```
2017-09-21 07:24:01,628 INFO   MySQL|dbserver1|snapshot  Step 0: disabling
autocommit and enabling repeatable read transactions
[io.debezium.connector.mysql.SnapshotReader]
…
2017-09-21 07:24:02,020 INFO   MySQL|dbserver1|snapshot  Step 9: committing
transaction   [io.debezium.connector.mysql.SnapshotReader]
```

Los steps terminan con la línea:

```
2017-09-21 07:24:02,021 INFO   MySQL|dbserver1|snapshot  Completed snapshot
in 00:00:00.405   [io.debezium.connector.mysql.SnapshotReader]
```

Tras esto, Kafka Connect imprimirá unos errores y advertencias que puede ignorar:

```
...
2017-09-21 07:24:02,632 WARN   ||  Error while fetching metadata with
correlation id 1 : {dbserver1=LEADER_NOT_AVAILABLE}
[org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:02,775 WARN   ||  Error while fetching metadata with
correlation id 5 : {dbserver1.inventory.customers=LEADER_NOT_AVAILABLE}
[org.apache.kafka.clients.NetworkClient]
2017-09-21 07:24:02,910 WARN   ||  Error while fetching metadata with
correlation id 9 : {dbserver1.inventory.orders=LEADER_NOT_AVAILABLE}
[org.apache.kafka.clients.NetworkClient]
...
```

Por último, Kafka Connect informará que el conector va a pasar a modo de lectura continua del log, y con esto ya estaría correctamente instalado:

```
2020-02-27 11:35:26,567 INFO   MySQL|dbserver1|task  Transitioning from the
snapshot reader to the binlog reader
[io.debezium.connector.mysql.ChainedReader]
2020-02-27 11:35:26,623 INFO   MySQL|dbserver1|task  Creating thread
debezium-mysqlconnector-dbserver1-binlog-client   [io.debezium.util.Threads]
2020-02-27 11:35:26,636 INFO   MySQL|dbserver1|task  Creating thread
debezium-mysqlconnector-dbserver1-binlog-client   [io.debezium.util.Threads]
Feb 27, 2020 11:35:26 AM com.github.shyiko.mysql.binlog.BinaryLogClient
connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:7)
2020-02-27 11:35:26,788 INFO   MySQL|dbserver1|binlog  Connected to MySQL
binlog at mysql:3306, starting at binlog file 'mysql-bin.000003', pos=154,
skipping 0 events plus 0 rows   [io.debezium.connector.mysql.BinlogReader]
2020-02-27 11:35:26,791 INFO   MySQL|dbserver1|binlog  Creating thread
debezium-mysqlconnector-dbserver1-binlog-client   [io.debezium.util.Threads]
```

En Kafka Connector, los eventos da cada tabla se escriben en un topic específico. Un topic no es más que un prefijo que se añade al log para hacer más fácil el saber dónde se produjo ese evento. Para este ejemplo, se ha comentado que se va a centrar en la tabla “customers”, con lo que el topic al que tenemos que prestar atención es: **dbserver1.inventory.customers**.

7. Visualización de cambios en la base de datos: watch-topic

Para visualizar los cambios que se han capturado en el topic **dbserver1.inventory.customers**, debemos extraer los logs de ese topic. En este ejemplo, usaremos “watch-topic”, que es bastante limitado y no se debería usar para realmente consumir los logs, sino que en un escenario real, se emplearían consumidores Kafka.

Abra otra ventana de comandos y ejecute el siguiente:

```
docker run -it --rm --name watcher --link zookeeper:zookeeper --link
kafka:kafka debezium/kafka:1.0 watch-topic -a -k
dbserver1.inventory.customers
```

Estamos creando otro Docker con la herramienta watch-topic de Kafka. Debería imprimir una serie de logs (cuatro en total), ya que hemos habilitado la opción -a. Estos logs corresponden a las inserciones que se hicieron en la tabla a la hora de crearla.

Cada evento es un par de estructuras en JSON. Una corresponde a la clave (primary key) del registro que se ha visto modificado, y otra es el valor, que contiene información sobre los valores previos y actuales (tras el evento) de ese registro.

Cada estructura tiene un “schema” y una “payload”. El “schema” contiene información sobre los tipos de datos que contiene la “payload”.

Aquí se puede observar una estructura de clave:

```
{
  "schema": {
    "type": "struct",
    "name": "dbserver1.inventory.customers.Key"
    "optional": false,
    "fields": [
      {
        "field": "id",
        "type": "int32",
        "optional": false
      }
    ]
  },
  "payload": {
    "id": 1004
  }
}
```

En este caso se puede observar que el registro que ha sufrido un cambio ha sido uno con un campo llamado id de tipo entero y valor 1004.

La estructura valor cambia con respecto a la estructura clave:

- "schema" sigue indicando los tipos de datos que contiene la "payload".
- "payload" sigue conteniendo los datos del registro.

Esta vez, tanto "schema" como "payload" están divididas en dos por dos etiquetas: "before" y "after", que indican el valor que tenía el registro antes y después del evento. En el caso de "schema", el before y el after se encuentran después de la descripción de los tipos de datos; y en el caso de "payload", el before y el after están antes de indicar los valores del registro.

Además, se añade a la payload:

- "source": que indica alguna metainformación sobre el evento. Destacan los campos:
    - "name": para indicar de qué servidor proviene el evento.
    - "db": para indicar de qué base de datos proviene el evento.
    - "table": para indicar de qué tabla proviene el evento.
- "op": que indica qué cual es la operación que provocó el evento. Hay varios valores:
    - "c": para crear e insertar.
    - "u": para actualizar.
    - "d": para borrar.
    - "r": para leer.
- "ts_ms": que indica el tiempo en el que el conector registró el evento.

A continuación, se puede observar un ejemplo de estructura de tipo valor, en el que se han señalado en negrita las partes más importantes:

```
{
  "schema": {
    "type": "struct",
    "fields": [
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "email"
          }
        ],
        "optional": true,
        "name": "dbserver1.inventory.customers.Value",
        "field": "before"
      },
      {
        "type": "struct",
        "fields": [
          {
            "type": "int32",
            "optional": false,
            "field": "id"
          },
          {
            "type": "string",
            "optional": false,
            "field": "first_name"
          },
          {
            "type": "string",
            "optional": false,
            "field": "last_name"
          },
```

```
        {
          "type": "string",
          "optional": false,
          "field": "email"
        }
      ],
      "optional": true,
      "name": "dbserver1.inventory.customers.Value",
      "field": "after"
    },
    {
      "type": "struct",
      "fields": [
        {
          "type": "string",
          "optional": true,
          "field": "version"
        },
        {
          "type": "string",
          "optional": false,
          "field": "name"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "server_id"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "ts_sec"
        },
        {
          "type": "string",
          "optional": true,
          "field": "gtid"
        },
        {
          "type": "string",
          "optional": false,
          "field": "file"
        },
        {
          "type": "int64",
          "optional": false,
          "field": "pos"
        },
        {
          "type": "int32",
          "optional": false,
          "field": "row"
        },
        {
          "type": "boolean",
          "optional": true,
```

```
            "field": "snapshot"
          },
          {
            "type": "int64",
            "optional": true,
            "field": "thread"
          },
          {
            "type": "string",
            "optional": true,
            "field": "db"
          },
          {
            "type": "string",
            "optional": true,
            "field": "table"
          }
        ],
        "optional": false,
        "name": "io.debezium.connector.mysql.Source",
        "field": "source"
      },
      {
        "type": "string",
        "optional": false,
        "field": "op"
      },
      {
        "type": "int64",
        "optional": true,
        "field": "ts_ms"
      }
    ],
    "optional": false,
    "name": "dbserver1.inventory.customers.Envelope",
    "version": 1
  },
  "payload": {
    "before": null,
    "after": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "version": "1.1.0.Beta2",
      "name": "dbserver1",
      "server_id": 0,
      "ts_sec": 0,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 154,
      "row": 0,

      "snapshot": true,
```

```
      "thread": null,
      "db": "inventory",
      "table": "customers"
    },
    "op": "c",
    "ts_ms": 1486500577691
  }
}
```

Este es la estructura valor que le corresponde a la estructura clave que se ha mostrado anteriormente. En este caso, sobre ese registro de clave 1004, se está realizando una operación “c”, es decir, crear o insertar.

En la “payload” se observa que el valor en “before” es null, y en “after” ya hay valores. Esto es debido a, como ya se comentó, este evento corresponde a una de las inserciones que se hicieron al principio cuando se creó la tabla. En este caso, se ha insertado a la cliente “Anne Kretchmar”, con e-mail “annek@noanswer.org”, como se puede observar en la “payload”, en el apartado “after”.

Como este evento, se deberían ver tres más en la consola donde está watch-topic arrancado, correspondiente a las 4 filas de la tabla customer (que ya habíamos consultado en el apartado “4. Arrancar un cliente de comandos SQL”).

Como se observa, cada evento produce un JSON bastante extenso. Se puede usar “Avro converter” para serializarlos. Se puede encontrar más información al respecto en la documentación de la herramienta.

8. Visualización de cambios en la base de datos: actualizar un registro

Se va a actualizar uno de los registros para ver cómo Debezium lo captura.

En la consola de comandos en la que tenemos la consola SQL (que abrimos en el apartado “4. Arrancar un cliente de comandos SQL”), escribimos el siguiente comando, para cambiar el nombre de “Anne” a “Anne Marie”:

```
UPDATE customers SET first_name='Anne Marie' WHERE id=1004;
```

Debería devolver:

```
Query OK, 1 row affected (0.18 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Puede comprobar que se ha ejecutado correctamente ejecutando el SELECT del apartado “4. Arrancar un cliente de comandos SQL”.

En la consola de comandos en la que tenemos el “watch-topic”, debería haber aparecido un nuevo evento (pulsar enter varias veces si no es así). Podemos destacar el apartado “payload” de la estructura valor del nuevo evento:


```
"payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "source": {
      "name": "1.1.0.Beta2",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501486,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 364,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "u",
    "ts_ms": 1486501486308
  }
```


Como se aprecia, “before” tiene el valor del registro antes del evento, con el nombre “Anne”, y “after” tiene el valor del registro después del evento, con el nombre “Anne Marie”.

La operación también ha cambiado. Ahora tiene el valor “u”, de actualizar.

9. Visualización de cambios en la base de datos: eliminar un registro.

Al igual que antes, en la ventana de comandos con el cliente de comandos SQL, escriba el siguiente comando para eliminar un registro:

```
DELETE FROM customers WHERE id=1004;
```

Que debería devolver lo siguiente:

```
Query OK, 1 row affected (0.25 sec)
```

ATENCION: Probablemente el comando de un error porque ese cliente es clave ajena de un registro en la tabla “addresses”. Si eso ocurre, ejecutar antes el siguiente comando:

```
DELETE FROM addresses WHERE customer_id=1004;
```

Si volvemos a la ventana de comandos con el “watch-topic”, podemos ver que se han generado dos o tres eventos (dependiendo de si se tuvo que ejecutar el comando DELETE de la tabla “addresses”, en ese caso, el primero correspondería a ese borrado).

Olvidando el primero que aparece, si se tuvo que ejecutar el otro comando, el siguiente que aparece corresponde al borrado del customer. Destacamos el “payload” de la estructura valor del evento:

```
{
  "schema": {...},
  "payload": {
    "before": {
      "id": 1004,
      "first_name": "Anne Marie",
      "last_name": "Kretchmar",
      "email": "annek@noanswer.org"
    },
    "after": null,
    "source": {
      "name": "1.1.0.Beta2",
      "name": "dbserver1",
      "server_id": 223344,
      "ts_sec": 1486501558,
      "gtid": null,
      "file": "mysql-bin.000003",
      "pos": 725,
      "row": 0,
      "snapshot": null,
      "thread": 3,
      "db": "inventory",
      "table": "customers"
    },
    "op": "d",
    "ts_ms": 1486501558315
  }
}
```

Como se observa, en este caso el apartado “before” contiene los datos de “Anne Marie”, pero el apartado “after” es null. Esto es debido a que el registro con clave 1004 ya no existe porque se ha borrado.

La operación en este caso es “d” de borrar.

El siguiente evento que se genera es un evento de tipo “tombstone”. Kafka por defecto viene configurado como “log-compacted”, lo que significa que va a eliminar mensajes antiguos del topic si hay al menos un mensaje después en el topic con la misma clave.

Aunque se borren del topic, el evento tombstone indica que los clientes pueden aún así leer desde el principio del topic y no perderse ningún evento.

10. Recuperación tras caída.

Vamos a comprobar cómo Debezium es capaz de recuperar los eventos que se produjeron mientras estaba apagado.

Primero, apague Kafka Connect con el siguiente comando:

```
docker stop connect
```

Luego, desde la consola de cliente de comandos SQL, ejecute estas dos inserciones:

```
INSERT INTO customers VALUES (default, "Sarah", "Thompson", "kitt@acme.com");
INSERT INTO customers VALUES (default, "Kenneth", "Anderson",
"kander@acme.com");
```

Volvemos a arrancar Kafka Connect con el siguiente comando desde su ventana de comandos:

```
docker run -it --rm --name connect -p 8083:8083 -e GROUP_ID=1 -e
CONFIG_STORAGE_TOPIC=my_connect_configs -e
OFFSET_STORAGE_TOPIC=my_connect_offsets -e
STATUS_STORAGE_TOPIC=my_connect_statuses --link zookeeper:zookeeper --link
kafka:kafka --link mysql:mysql debezium/connect:1.0
```

Espere hasta que arranque completamente. Debería ver unos logs parecidos a estos:

```
2017-09-21 07:38:48,403 INFO   MySQL|dbserver1|task  Setting newly assigned
partitions [dbhistory.inventory-0] for group inventory-connector-dbhistory
[org.apache.kafka.clients.consumer.internals.ConsumerCoordinator]
2017-09-21 07:38:48,888 INFO   MySQL|dbserver1|task  Step 0: Get all known
binlogs from MySQL   [io.debezium.connector.mysql.MySqlConnectorTask]
2017-09-21 07:38:48,903 INFO   MySQL|dbserver1|task  MySQL has the binlog
file 'mysql-bin.000003' required by the connector
[io.debezium.connector.mysql.MySqlConnectorTask]
Sep 21, 2017 7:38:49 AM com.github.shyiko.mysql.binlog.BinaryLogClient
connect
INFO: Connected to mysql:3306 at mysql-bin.000003/154 (sid:184054, cid:10)
2017-09-21 07:38:49,045 INFO   MySQL|dbserver1|binlog  Connected to MySQL
binlog at mysql:3306, starting at binlog file 'mysql-bin.000003', pos=154,
skipping 0 events plus 0 rows   [io.debezium.connector.mysql.BinlogReader]
```

```
2017-09-21 07:38:49,046 INFO   ||  Source task WorkerSourceTask{id=inventory-
connector-0} finished initialization and start
[org.apache.kafka.connect.runtime.WorkerSourceTask]
```

Si vamos a la ventana de comandos de "watch-topic" deberían haberse generado dos eventos nuevos correspondientes a las dos inserciones que hemos realizado mientras Debezium estaba apagado.

11. Limpiar los servicios arrancados

Una vez terminado el ejemplo, debe asegurarse cerrar bien todos los servicios arrancados. Para ello, en cualquier consola de comandos, ejecute:

```
docker stop mysqlterm watcher connect mysql kafka zookeeper
```

Puede verificar si se han cerrado ejecutando el comando:

```
docker ps -a
```

No debería aparecer ningún proceso activo (si no tenía alguno ya activo que no corresponda a este ejemplo).

6. DBLOG: INTRODUCCIÓN

DBLog es una infraestructura software para realizar tareas CDC creada por el equipo de desarrollo de la plataforma de streaming de vídeos Netflix.

Esta herramienta permite recoger en logs los cambios que se producen en las bases de datos, y además permite que los logs convivan con posibles volcados (dumps) que se realicen sobre las bases de datos.

DBLog nace como una solución a una serie de problemas que el equipo de Netflix encontró en otras soluciones CDC ya existentes. Los problemas identificados fueron:

- Se detiene el procesamiento de logs mientras se hace el volcado de la base de datos, lo cual es un problema si el volcado es de dimensiones considerables y los consumidores de logs necesitan actualizaciones muy rápidas de los datos.
- No tienen la posibilidad de ejecutar volcados manualmente: algunas soluciones solo realizan volcados durante fases de arranque o si detectan pérdidas de datos en el procesamiento de los logs, pero no permiten que se ejecuten volcados cuando se desee.
- Bloquean el tráfico de escritura bloqueando las tablas: muchas soluciones emplean técnicas complejas a la hora de realizar el volcado, y todas implican bloquear las tablas para evitar la escritura.

- Usan características específicas de cada base de datos, lo que impide que la herramienta se pueda reusar para otras bases de datos diferentes.

Todos estos impedimentos se abordan y solucionan en DBLog.

7. DBLOG: CARACTERÍSTICAS

A continuación, se listan las características que ofrece DBLog:

- Procesa todos los eventos en orden temporal, independientemente de la base de datos origen en la que se produzca.
- Permite realizar volcados de las bases de datos en orden y de forma customizada, es decir, se pueden hacer volcados completos, de tablas, o de registros con una clave específica en una tabla.
  Además, intercala logs con volcados, de tal forma que ambos procesos pueden coexistir sin tener pérdidas de datos. Esto es debido a que los volcados se dividen en "chunks", lo que permite no solo la recuperación del proceso en caso de la caída del servicio, sino que también permite cancelar el proceso o pausarlo y reanudarlo más tarde.
- Nunca se bloquean las tablas.
- Permite cualquier tipo de salida: streams, datastores e incluso APIs.

DBLog tiene compatibilidad con bases de datos relacionales como: MySQL, PostgreSQL, MariaDB, y cualquier base de datos que sea similar a alguna de las anteriores.

8. DBLOG: PROCESAMIENTO DE LOGS

Cada operación de tipo Create, Update o Delete que se realiza en alguna de las bases de datos con las que está conectada la herramineta, se genera y serializa un log con el formato de DBLog y se envía a un proceso de escritura para que se envíe a los consumidores de logs.

Este envío de logs al proceso de escritura es una operación no bloqueante, ya que el proceso de escritura se ejecuta en un hilo de ejecución distinto, lo que mejora el transporte de los logs.

9. DBLOG: PROCESAMIENTO DE VOLCADOS (DUMPS)

Los volcados son muchas veces necesarios debido a que los logs tienen una retención limitada, lo que impediría una reconstrucción completa de los datos en caso de pérdida.

Cuando se realiza un volcado, se genera un evento de log por cada fila que se selecciona, se serializa en formato DBLog al igual que los logs usuales y se envían al mismo proceso de escritura, lo que permite que los consumidores sean inconscientes de qué logs son originarios del proceso de logs normal o de un proceso de volcado.

Como se ha indicado en la sección DBLog: Características, los volcados se producen en “chunks”. DBLog permite configurar el tamaño de estos chunks y un tiempo de retardo entre el volcado de dos chunks antes o durante la ejecución de la herramienta, en el caso que se no se desee detener demasiado tiempo el procesamiento de logs.

Nota: Los chunks se seleccionan ordenando las dilas de las tablas de forma ascendente usando las claves primarias e incluye las filas cuya clave primaria es mayor que última clave primaria del chunk anterior. Esto implica que sea necesario que la base de datos origen ejecute esta consulta de manera eficiente para no retardar el procesamiento.

El proceso de generación de un chunk, se generan dos marcas de agua. Estas marcas de agua se implementan con una tabla en la base de datos con una única columna y una única fila, que contiene un UUID que representa la marca de agua. Esta tabla se encuentra en un namespace único de la base de datos, para que no colisione con otras tablas.

Cada marca de agua se genera actualizando esa fila de la tabla, y al ser una actualización de una fila en la base de datos, se genera un log que contiene la marca de agua.

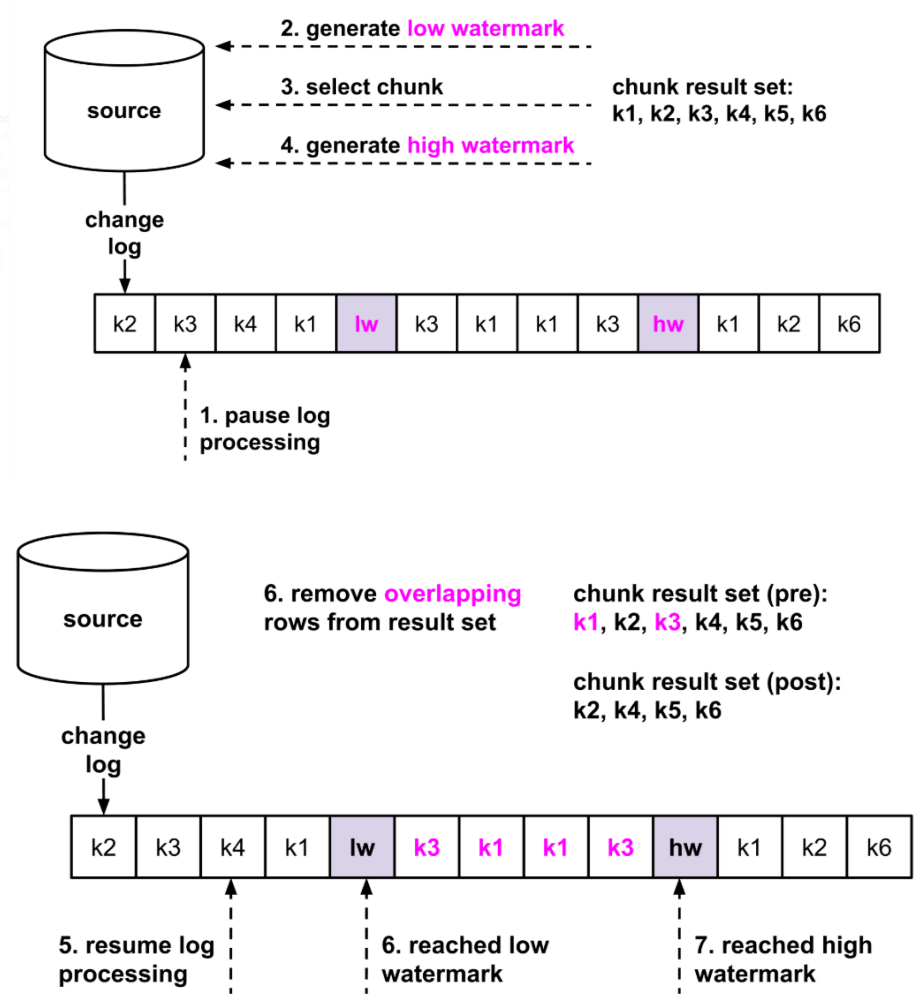
Con las marcas de agua, se pretende solventar el caso en el que, durante la generación de un chunk, se actualice uno de los registros que contiene, en cuyo caso ese registro no debe escribirse en la salida de DBLog. El proceso que se sigue para la generación de un chunk es el siguiente:

1. Se pausa el proceso de captura de logs.
2. Se genera la primera marca de agua actualizando la tabla.
3. Se ejecuta el SELECT de los registros que conforman el chunk.
4. Se genera la segunda marca de agua actualizando la tabla.
5. Se reanuda el procesamiento de logs. Empezarán a escribirse los logs producidos mientras el procesamiento de logs estaba parado. Se espera a que se registre la primera marca de agua.
6. Cuando se recibe la primera marca de agua, se eliminan del chunk todos los registros cuya clave primaria coincida con los logs que se registren antes de que llegue la segunda marca de agua. Los logs que se registran entre la primera y la segunda marca de agua, son las actualizaciones que se produjeron mientras se ejecutaba el SELECT. Si alguno de esos logs tiene la misma clave primaria que alguno de los registros del chunk, hay que eliminar ese registro del chunk, pues el valor más reciente es el del log que se acaba de recibir.

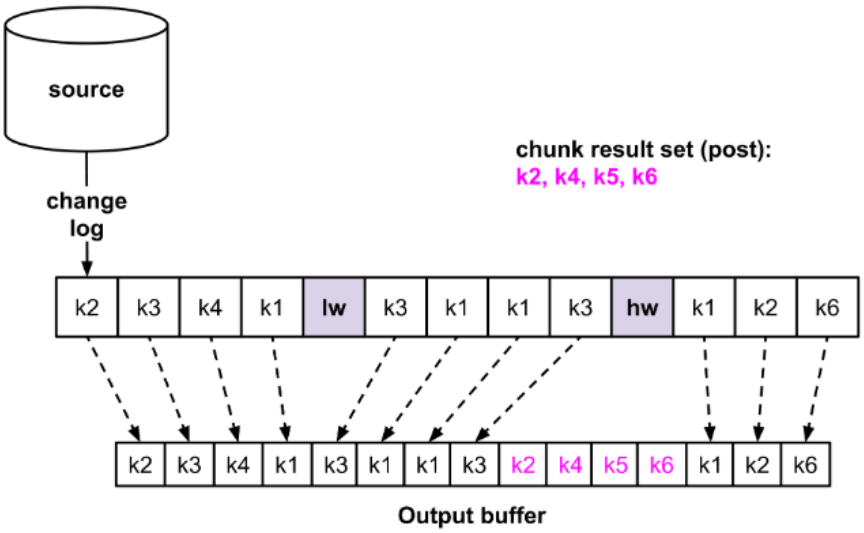
7. Cuando se recibe la segunda marca de agua, se escriben el resto de registros que queden en el chunk.
8. Si quedan más chunks por generar, volver al paso 1.

Para que este proceso se realice de forma correcta, se debe asegurar que el SELECT que se ejecuta, se ejecute completamente entre la generación de la primera y la segunda marca de agua.

Puede ver un ejemplo gráfico a continuación del proceso de generación de un chunk. Este ejemplo se ha extraído del post escrito por el equipo de desarrollo de Netflix (https://netflixtechblog.com/dblog-a-generic-change-data-capture-framework-69351fb9099b).



La salida que generaría este proceso sería la siguiente:



10. DBLOG: ALTA DISPONIBILIDAD

DBLog usa una arquitectura Activa-Pasiva, en la que se tiene una instancia activa que está ejecutando, y otras instancias pasivas que se encuentran a la espera. Se usa ZooKeeper para realizar la orquestación de las instancias, y se emplean leases para decidir cuál ejecuta en cada momento.

El lease lo tiene siempre la instancia activa, y debe renovarse cada cierto tiempo. En el caso de que el lease no se renueve, significaría que la instancia activa ha dejado de funcionar, con lo que el lease se cede a una de las instancias pasivas, que pasa a ser activa. Esto permite que ante cualquier fallo, no se detenga el procesamiento de logs y dumps.

Se permite que la instancia activa se encuentre en la misma situación geográfica de la base de datos y que las instancias pasivas se encuentren en situaciones diferentes; sin embargo, como se quiere primar la baja latencia en el procesamiento de logs, lo más eficiente es que todas las instancias se encuentren en el mismo lugar.

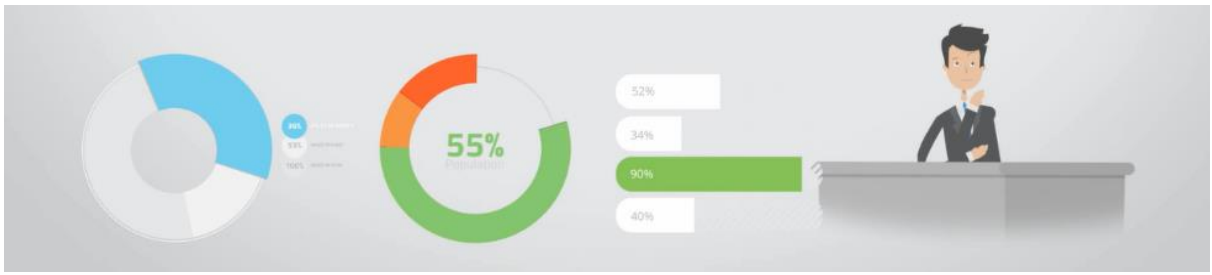
11. DBLOG: PROXIMAMENTE OPEN SOURCE

El equipo de Netflix va a publicar DBLog como herramienta Open Source este año 2020 para que cualquiera la pueda usar, además de añadir más documentación sobre ella, sin embargo, aún no hay noticias de en qué fecha se producirá la publicación exactamente.

12. DESCRIPCIÓN STRATEBI

En Stratebi ofrecemos **gran cantidad de soluciones analíticas** por una compañía de **rápido crecimiento,** innovando en las áreas tecnológicas de mayor desarrollo en la actualidad: **Business Intelligence, Big Data y Social Intelligence**, muchas de ellas, basadas en soluciones **Open Source**.

Además, somos **Partners Certificados en Microsoft PowerBI y Vertica**, con gran número de proyectos con ámbas tecnologías



Desarrollamos nuevas soluciones analíticas basadas en Open Source, para la generación de Cuadros de Mando en tiempo real, con tecnologías IoT para SmartCities, machine learning, etc...

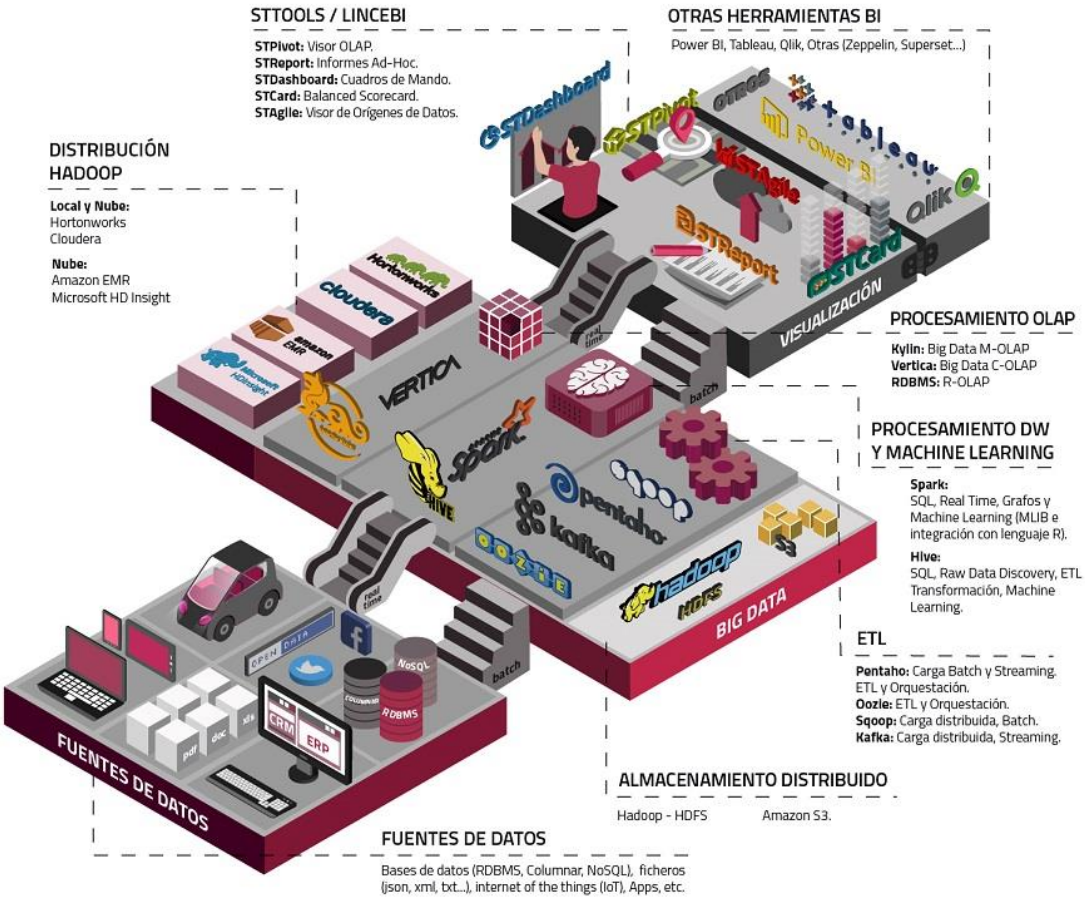
LinceBI
The best open source BI based solution

13. TECNOLOGÍAS

Recientemente, hemos sido nombrados Partners Certificados de Vertica, Talend, Microsoft, Snowflake, Kylligence, Pentaho, etc…

14.DESCRIPCIÓN DE LAS SOLUCIONES

Stratebi son los creadores de la solución LinceBI.com sobre la que podrás desarrollar e innovar.



15.INFORMACIÓN SOBRE STRATEBI

Stratebi es una empresa española, con sede en Madrid y oficinas en Barcelona, Alicante y Sevilla, creada por un grupo de profesionales con amplia experiencia en sistemas de información, soluciones tecnológicas y procesos relacionados con soluciones de Open Source y de inteligencia de Negocio.

Esta experiencia, adquirida durante la participación en proyectos estratégicos en compañías de reconocido prestigio a nivel internacional, se ha puesto a disposición de nuestros clientes a través de Stratebi.

Stratebi es la única empresa española que ha estado presente todos los Pentaho Developers celebrados en Europa (Mainz-Alemania, Barcelona, Lisboa, Roma, Amsterdam, Sintra, Amberes (2) Londres…), habiendo organizado el de Barcelona.

En Stratebi nos planteamos como **objetivo** dotar a las compañías e instituciones, de herramientas escalables y adaptadas a sus necesidades, que conformen una estrategia Business Intelligence capaz de rentabilizar la información disponible. Para ello, nos basamos en el desarrollo de soluciones de Inteligencia de Negocio, mediante tecnología Open Source.

Stratebi **son profesores y responsables de proyectos del Master en Business Intelligence de la Universidad UOC, UCAM, EOI…**

Los profesionales de Stratebi son los creadores y autores del primer weblog en español sobre el mundo del Business Intelligence, Data Warehouse, CRM, Dashboards, Scorecard y Open Source.

Stratebi es partner de las principales soluciones Analytics: Microsoft PowerBI, Talend, Pentaho, Vertica, Snowflake, Kyligence, Cloudera…

Todo Bi, se ha convertido en una referencia para el conocimiento y divulgación del Business Intelligence en español.

16.REFERENCIAS STRATEBI

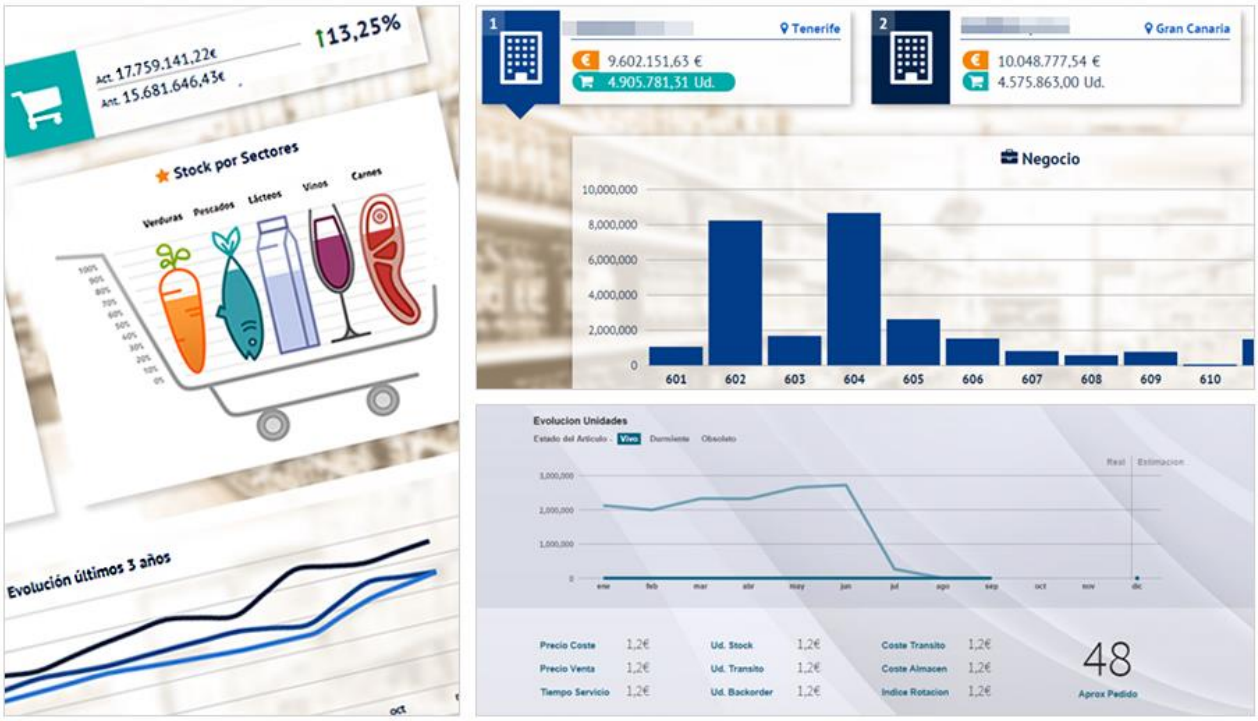
Trabajamos en los principales sectores y con algunas de las compañías y organizaciones más importantes de España.

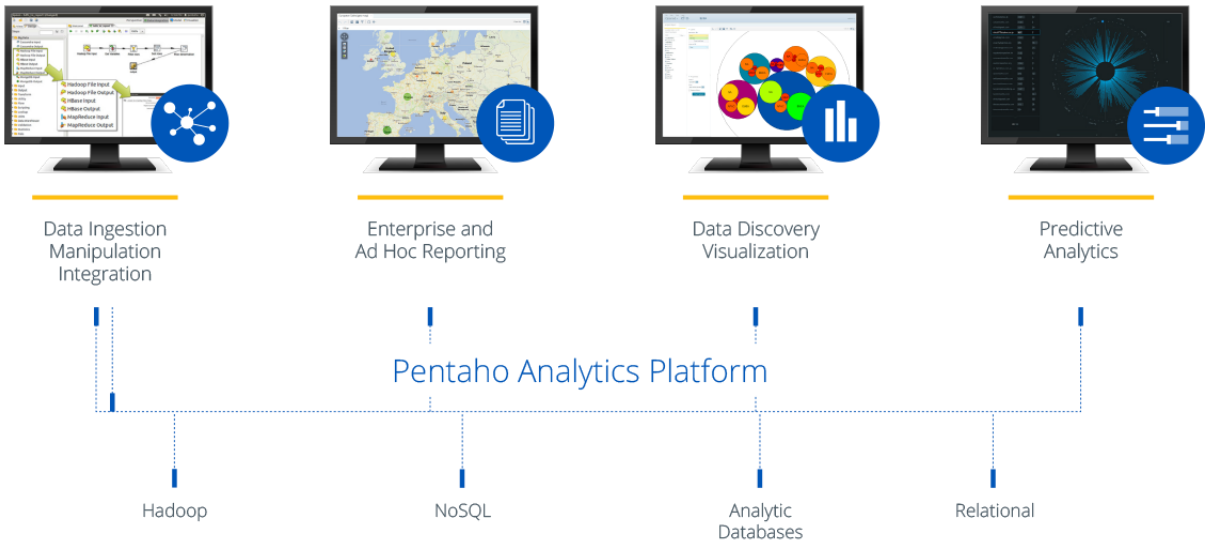
SECTOR PRIVADO

SECTOR PÚBLICO

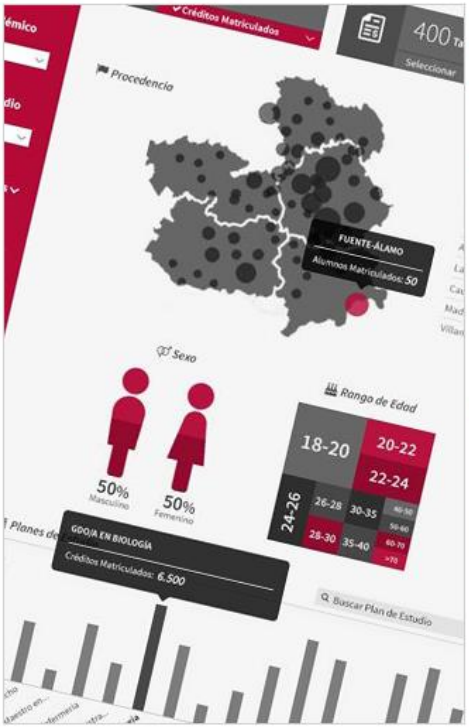
17. EJEMPLOS DE DESARROLLOS ANALYTICS

A continuación se presentan **ejemplos de algunos screenshots** de cuadros de mando diseñados por Stratebi, con el fin de dar a conocer lo que se puede llegar a obtener, así como Demos Online en la web de Stratebi:



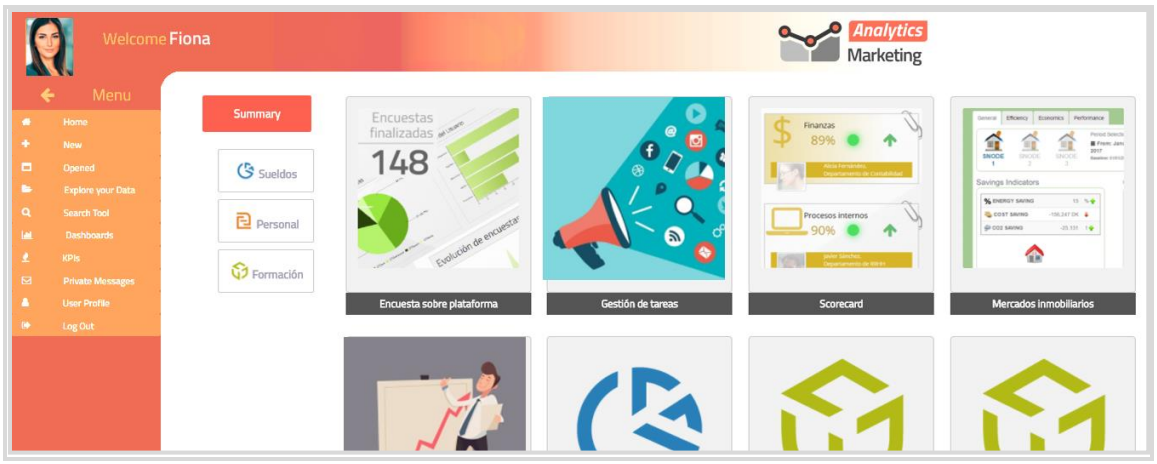


facebook DASHBOARD COMENTARIOS
Sexo
Sentimiento
Localización
Tipo de recurso
Evolución Comentarios por días
smart MADRID
Última Actualización
18:24
TOP 5 PUNTOS DE MEDICIÓN
PROXIMIDAD DE PUNTOS DE MEDIDA
EVOLUCIÓN INTENSIDAD DE TRÁFICO


Créditos Matriculados
Procedencia
FUENTE-ÁLAMO
Alumnos Matriculados: 50
Sexo
50% Masculino
50% Femenino
Rango de Edad
18-20
20-22
22-24
24-26
26-28
28-30
30-35
35-40
GRADO EN BIOLOGÍA
Créditos Matriculados: 6.500
Buscar Plan de Estudio

INTERÉS PLATAFORMA
181 Inscritos en Cursos
8 Desinscritos en Cursos
181 Inicios de Cursos
110 Inscritos en Eventos Virtuales
63 Inscritos en Debates
181 Inscritos en Talleres
USUARIOS ACTIVOS POR PAÍS
INSCRITOS EN CURSOS POR COMUNIDAD EDU

JORNADA COMPLETA
IDIOMAS
TIC
TIPO DE CENTRO
TIPO DE AUSENCIA
Nº de profesores por sexo
36%
64%
Nº total de profesores
1.587
Media de edad
43 años
Búsqueda


Welcome Fiona
Analytics
Marketing
Menu
Home
New
Opened
Explore your Data
Search Tool
Dashboards
KPIs
Private Messages
User Profile
Log Out
Summary
Sueldos
Personal
Formación
Encuestas finalizadas
148
Encuesta sobre plataforma
Gestión de tareas
Finanzas
89%
Procesos internos
90%
Scorecard
Mercados inmobiliarios