CARTO Big Data connectors

Versión 1.0.0

Alberto Romeu Carrasco

12 de mayo de 2019

Contents:

CAPÍTULO 1

Introducción

1.1 Contexto

CARTO[1] es una plataforma de *Location Intelligence* que permite resolver problemas geoespaciales con los mejores datos y análisis.

Actualmente, las organizaciones que utilizan *CARTO* como herramienta de análisis geoespacial tienen multitud de fuentes de información y aplicaciones ya instaladas que generan continuamente nuevos datos.

Uno de los valores de *CARTO* para estas organizaciones es el de conectarse con esas fuentes de información (datos de *CRM*, *ERP*, hojas de cálculo, archivos con contenido geoespacial, bases de datos relacionales, etc.) a través de una interfaz sencilla e intuitiva, para generar nueva información de valor añadido para su negocio mediante análisis geoespaciales y visualizaciones.

En determinadas organizaciones, especialmente de tamaño medio o grande, ocurre que diversos equipos gestionan sus datos con sistemas de información heterogéneos que utilizan repositorios de datos tales como:

- Hojas de cálculo y archivos CSV (valores separados por comas)
- Documentos de Google Drive[2]
- CRMs tales como Salesforce[3]
- Herramientas de marketing digital como Mailchimp[4]
- Servidores de datos geoespaciales como ArcGIS[5]
- Bases de datos operacionales (relacionales o *NoSQL*)
- Sistemas de ficheros distribuidos *HDFS*

[1] https://www.carto.com - mayo 2019
[2] https://drive.google.com - mayo 2019
[3] https://www.salesforce.com - mayo 2019
[4] https://mailchimp.com - mayo 2019
[5] https://www.arcgis.com - mayo 2019

- Otros sistemas (Twitter[6], Dropbox[7], etc.)

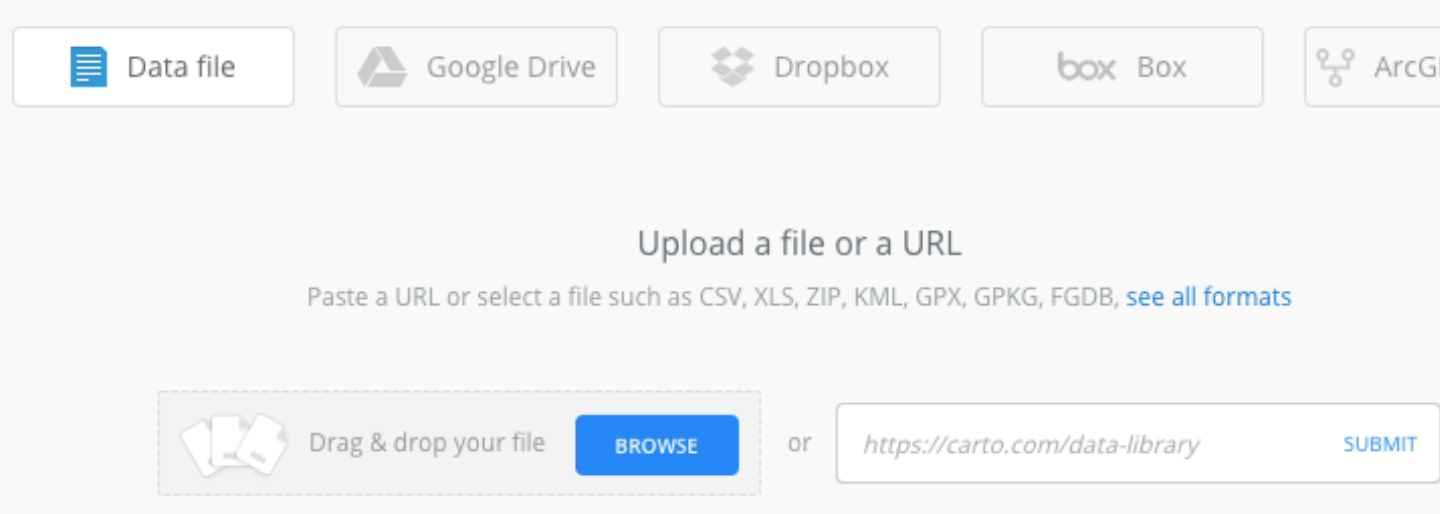
1.2 Definición del problema

Estas organizaciones utilizan *CARTO* para importar sus datos y analizar la información que generan los distintos departamentos de manera conjunta, respondiendo a sus preguntas de negocio y encontrando además respuesta a otras preguntas que no se habían planteado en un principio.

CARTO ofrece un flujo de trabajo formado por cinco pasos:

1. Ingestión de datos

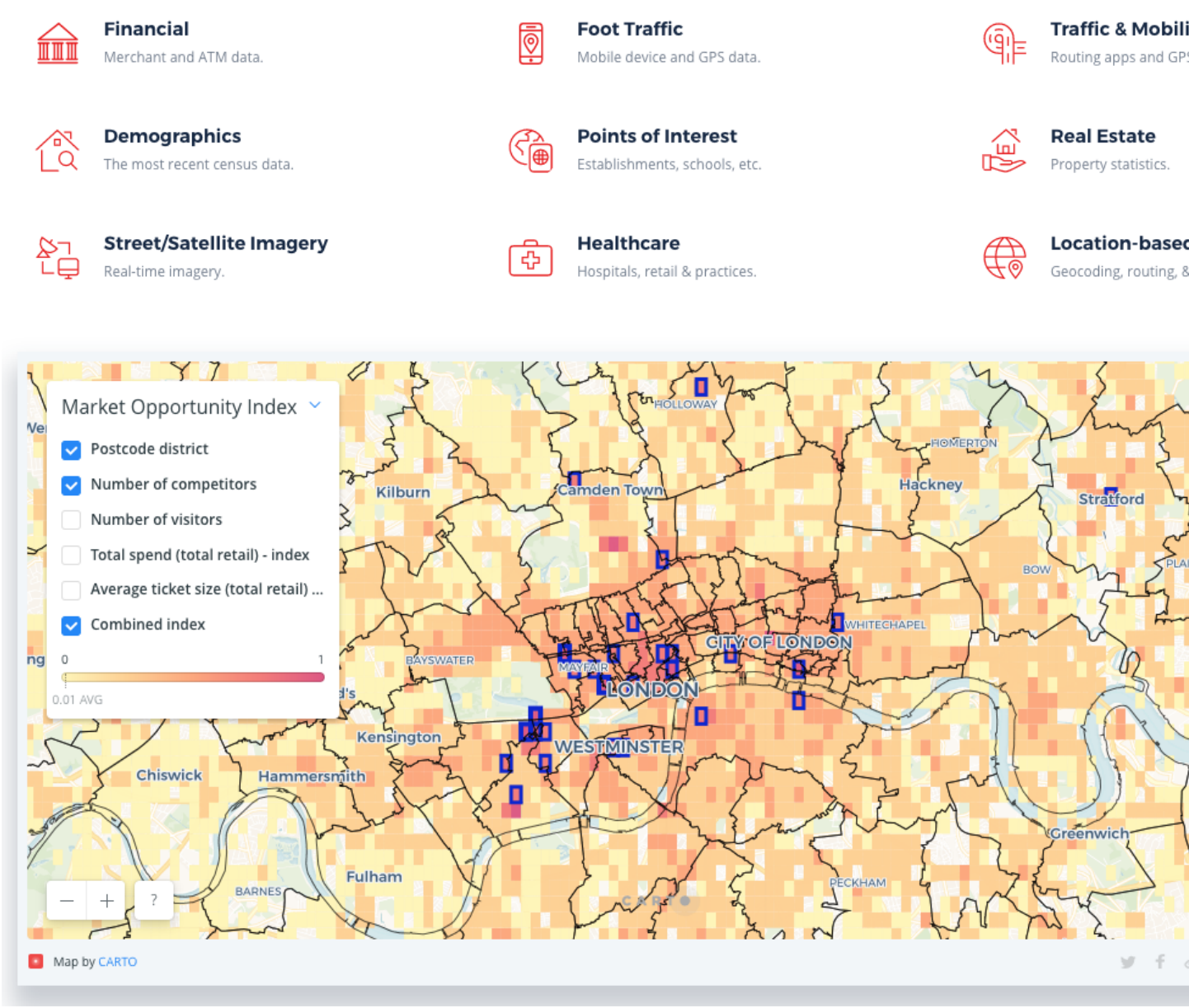
Consiste en llevar los datos de negocio, provenientes de distintas fuentes a la plataforma, a través de APIs e interfaces de usuario intuitivas.



2. Enriquecimiento de los datos

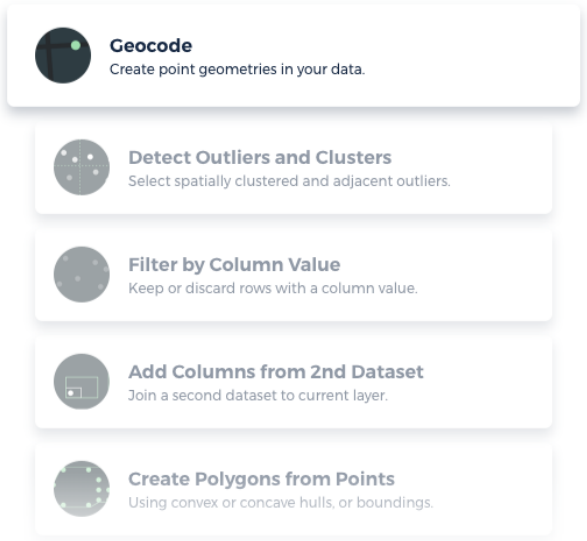
CARTO cuenta con un catálogo de datos previamente analizados y curados de diferente tipología (demográficos, económicos, movilidad, puntos de interés, . . .) que son clave para añadir nueva información a los datos de negocio y relacionar variables para buscar respuesta a preguntas complejas tales como: ¿de dónde vienen los visitantes de mis tiendas físicas? ¿cuál es su perfil demográfico? ¿cuál es su poder adquisitivo? etc.

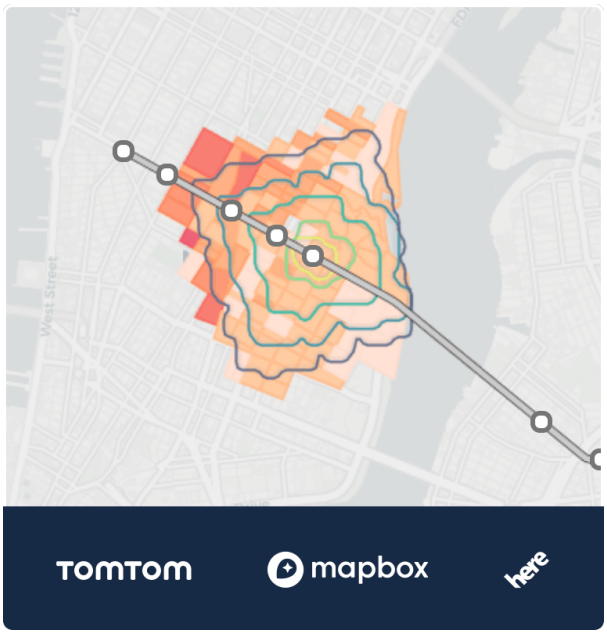
[6] https://www.twitter.com - mayo 2019
[7] https://www.dropbox.com - mayo 2019



3. Análisis

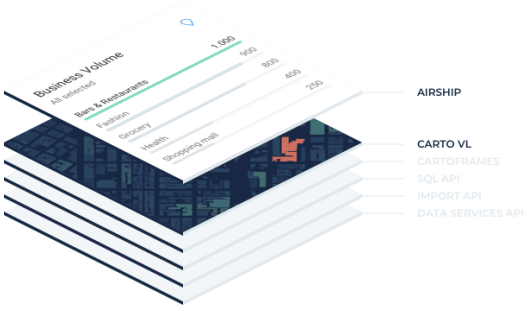
APIs y soluciones orientadas al análisis espacial

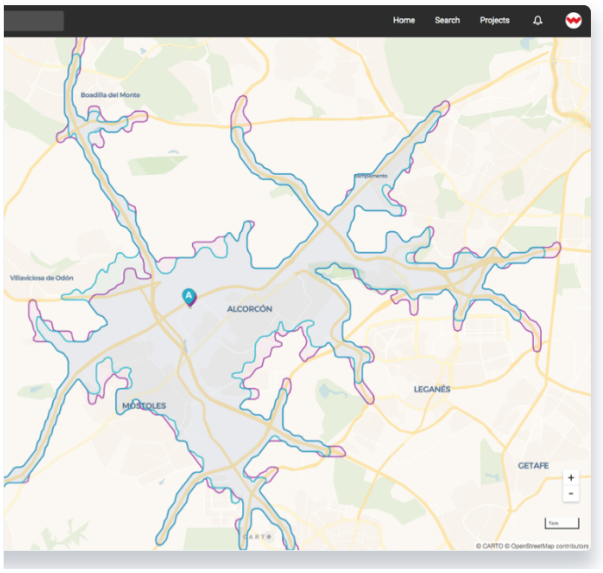




4. Soluciones y visualización

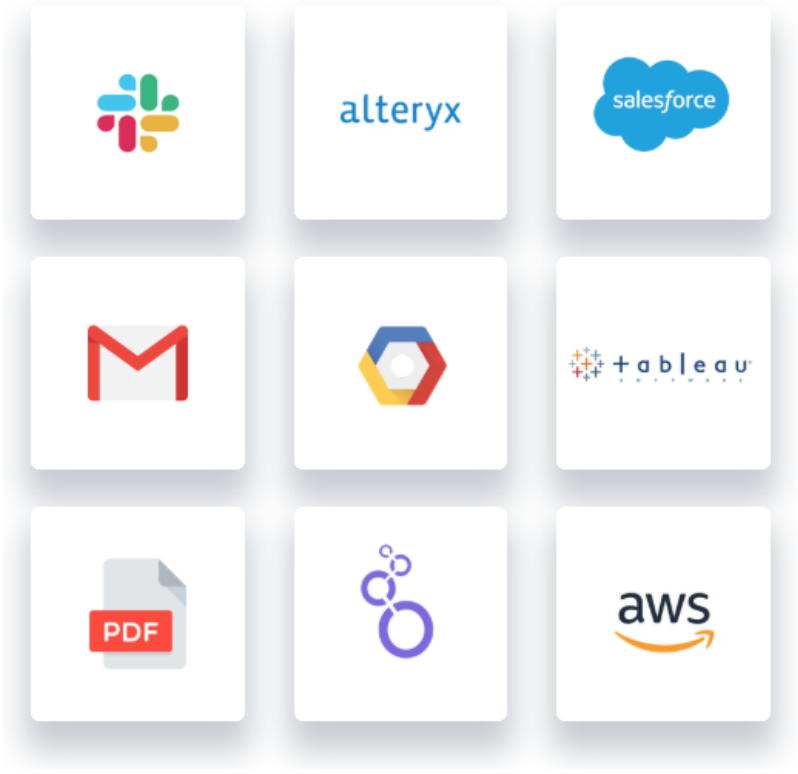
Librerías, SDKs, APIs y herramientas específicas que permiten crear soluciones y exponer de manera visual los resultados obtenidos en los análisis





5. Integraciones

Otros flujos para dar salida a los resultados obtenidos hacia la plataforma del cliente.



CARTO cuenta con la posibilidad de importar datos desde diversas fuentes de datos, pero carece de soporte nativo para conectar a muchos de estos sistemas de almacenamiento Big Data usados generalmente para almacenar datos operacionales o secuencias de datos temporales.

1.3 Objetivo

El objetivo de este trabajo final de máster, está centrado en la primera fase del flujo de trabajo descrito previamente la ingestión de datos. Más concretamente, consiste en el desarrollo de conectores para *CARTO* que permitan incluir en los cuadros de mandos (*dashboard*), información proveniente de los siguientes sistemas de almacenamiento y/o procesamiento Big Data.

El objetivo es encontrar un mecanismo fácilmente reproducible que permita en el futuro integrar otros sistemas de almacenamiento. Para el actual trabajo, el objetivo consiste en integrar al menos:

- Google BigQuery

Y describir un proceso que permitiera la integración de otros sistemas tales como:

- Apache Hive
- Impala
- MongoDB
- Amazon Redshift
- Cassandra
- SparkSQL
- Amazon Aurora
- Oracle

Los «conectores Big Data para *CARTO*» permitirán a las organizaciones mantener sus actuales flujos de ingestión y procesamiento de información, además de aprovechar lo mejor de dos mundos: el almacenamiento y procesamiento distribuido que ofrecen algunas de estas herramientas orientadas a Big Data y la visualiación y análisis geoespacial de *CARTO*.

Cabe destacar que los resultados de este trabajo no son de carácter teórico, sino que consiste en código fuente y herramientas que se incluirán en la distribución *on-premise* de *CARTO*

1.4 Organización de este trabajo final de máster

Este trabajo final de máster está organizado en capítulos, siguiendo la siguiente estructura:

1. *Estado del arte*: Se repasan las herramientas de almacenamiento y procesamiento Big Data con las que se va a trabajar y se definen algunos de los conceptos teóricos que sirven de fundamentación para el trabajo.

2. *Metodología y plan de trabajo*: Definición de una metodología de trabajo sistemática y desglose en tareas del trabajo a realizar.
3. *Desarrollo del trabajo y resultados obtenidos*: Descripción de la implementación de cada uno de los conectores, demostración de uso, etc.
4. *Espías en el cielo: Analizando con CARTO datos de vuelos almacenados en BigQuery*: Un caso de uso para entender mejor desde el punto de vista de un usuario de la plataforma, del valor de contar con los conectores desarrollados en el trabajo.
5. *Conclusiones*
6. *Bibliografía*
7. *Anexos*
8. *Glosario*

Palabras clave: *BASH, Docker, Vagrant, Location Intelligence, AWS, HDFS, Hadoop, BigQuery, Hive, Impala, Spark, NoSQL, Cassandra, MongoDB, CARTO, dashboards, análisis geoespacial*

CAPÍTULO 2

Estado del arte

En este capítulo se presenta un informe sobre los diferentes sistemas de almacenamiento y procesamiento Big Data para los que se va a dar soporte para la realización de conectores para CARTO y una breve definición de los conceptos teóricos que sirven de fundamentación para el trabajo.

2.1 CARTO

CARTO[1] es una plataforma de *Location Intelligence* que permite transformar datos geoespaciales en resultados de negocio.

A diferencia del *Business Intelligence*, *Location Intelligence* es el conjunto de herramientas y metodologías que permiten extraer conocimiento y tomar decisiones de negocio a partir de datos geoespaciales.

CARTO es una plataforma *SaaS* referente en este sector que permite de una manera sencilla e intuitiva la importación de conjuntos de datos con información geoespacial para crear a través de *dashboard* y *widgets*, mapas con capacidades de análisis, filtrado, búsqueda y predicción de variables.

La plataforma, da soporte a todo el flujo de trabajo para el análisis de datos geoespaciales. Desde la ingestión de datos, enriquecimiento con nuevas fuentes de datos, análisis, visualización e integración.

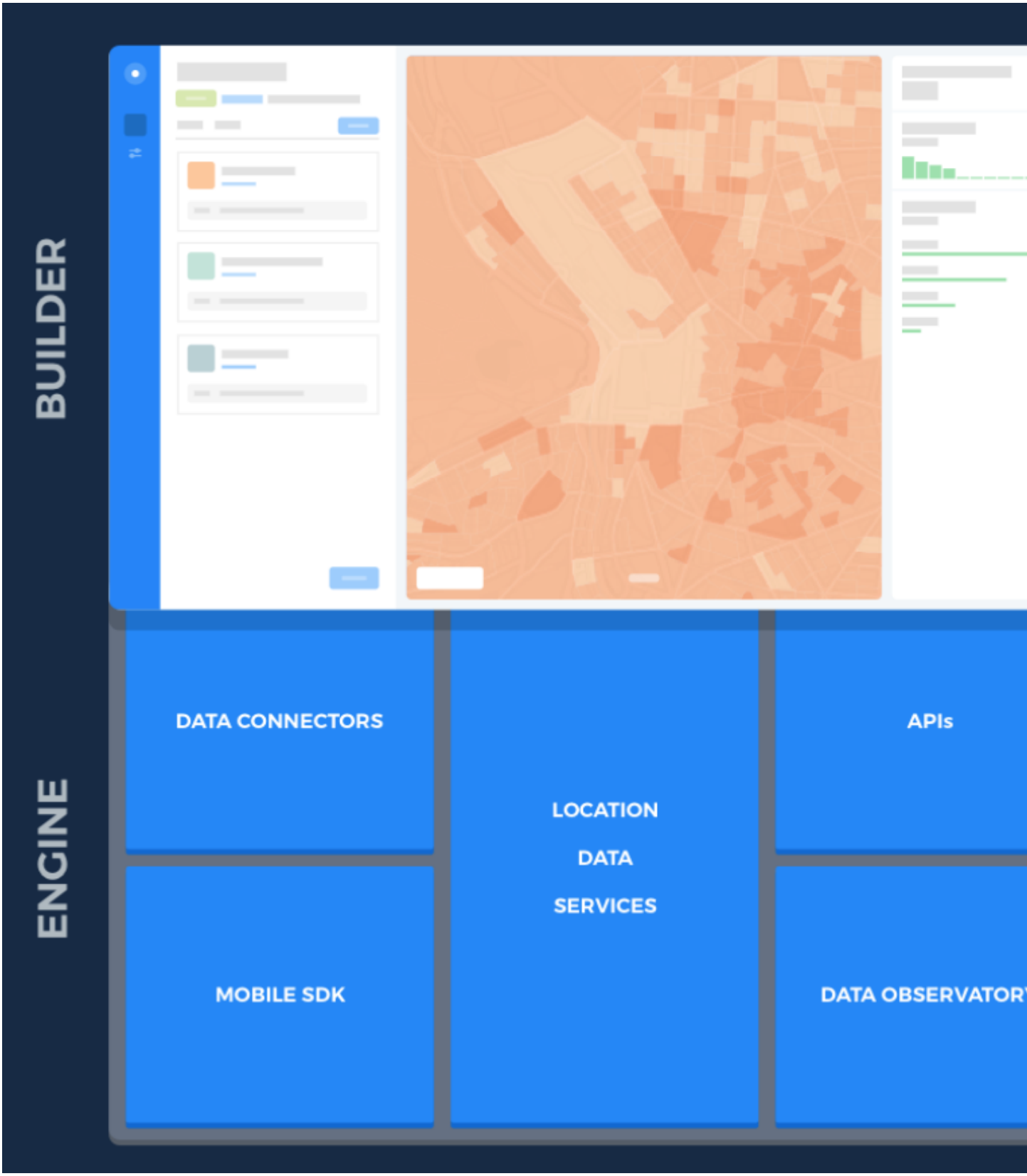
CARTO cuenta con la posibilidad de importar datos desde diversas fuentes de datos, pero carece de soporte nativo para conectar a algunos de los principales sistemas de almacenamiento Big Data usados generalmente para almacenar datos operacionales o secuencias de datos temporales.

Actualmente, *CARTO* cuenta con más de 300000 usuarios registrados y es usado por más de 2000 organizaciones de todo el mundo para tomar decisiones a partir de sus datos geolocalizados.

2.1.1 Productos

En la actualidad, CARTO es una plataforma formada por multitud de APIs, librerías, soluciones y verticales. Para el objetivo de este trabajo nos vamos a centrar en 5 productos principales.

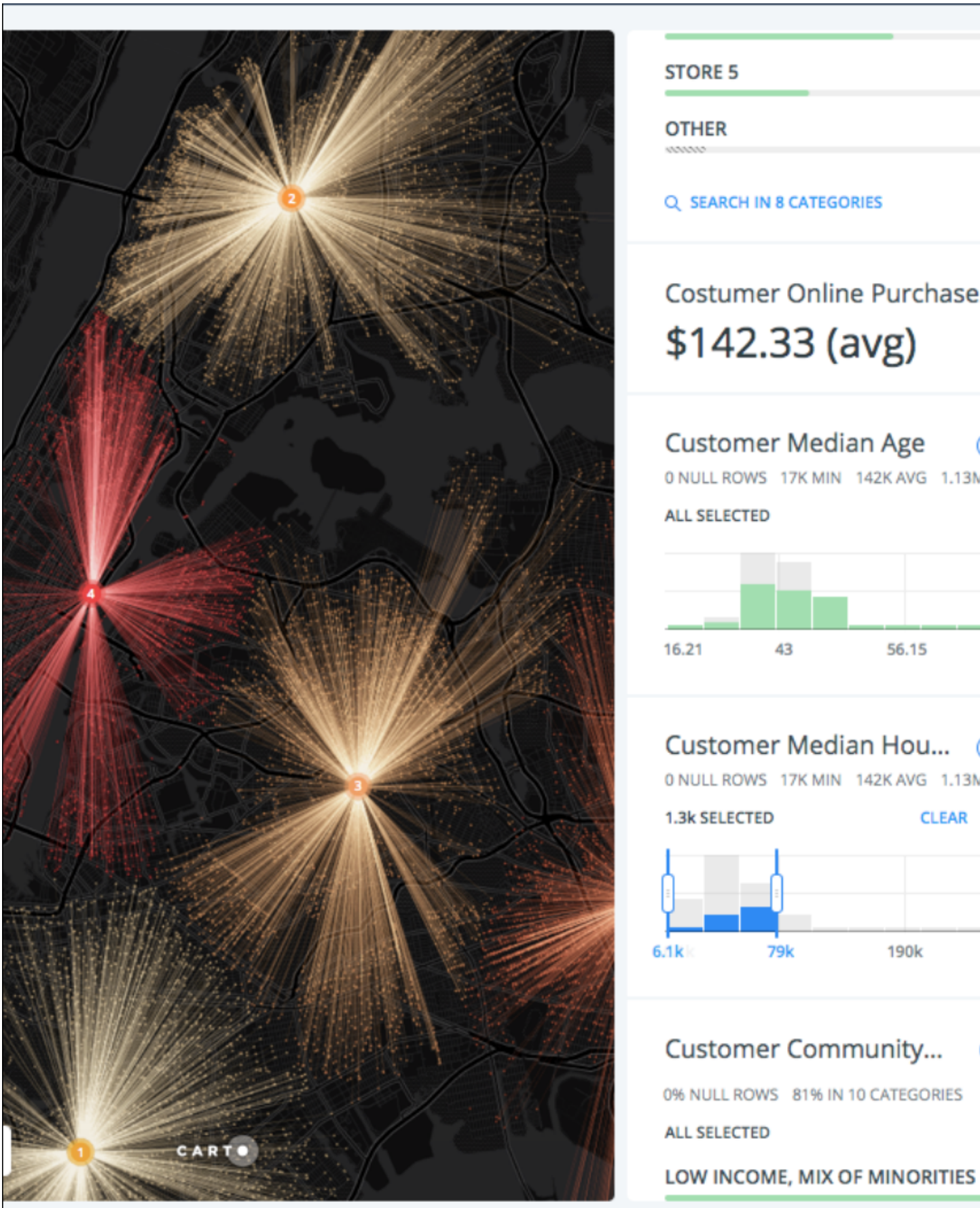
[1] https://carto.com/ - mayo 2019


BUILDER
ENGINE
DATA CONNECTORS
LOCATION
DATA
SERVICES
APIs
MOBILE SDK
DATA OBSERVATOR

- BUILDER[2]

Una herramienta web de análisis para que analistas y usuarios de negocio que permite crear cuadros de mando accionables que se pueden compartir.

[2] https://carto.com/builder/ - mayo 2019


STORE 5
OTHER
SEARCH IN 8 CATEGORIES
Costumer Online Purchase
$142.33 (avg)
Customer Median Age
0 NULL ROWS 17K MIN 142K AVG 1.13M
ALL SELECTED
16.21 43 56.15
Customer Median Hou...
0 NULL ROWS 17K MIN 142K AVG 1.13M
1.3k SELECTED
CLEAR
6.1k 79k 190k
Customer Community...
0% NULL ROWS 81% IN 10 CATEGORIES
ALL SELECTED
LOW INCOME, MIX OF MINORITIES
CARTO

- ENGINE[3]

Un conjunto de herramientas geoespaciales, servicios y APIs para el desarrollo de aplicaciones geoespaciales.

- Location Data Services[4]

Mapas base, mapas vectoriales, servicios de geocodificación y cálculo de rutas que se pueden consumir fácilmente en BUILDER o integrar a través de ENGINE.

- Data Observatory[5]

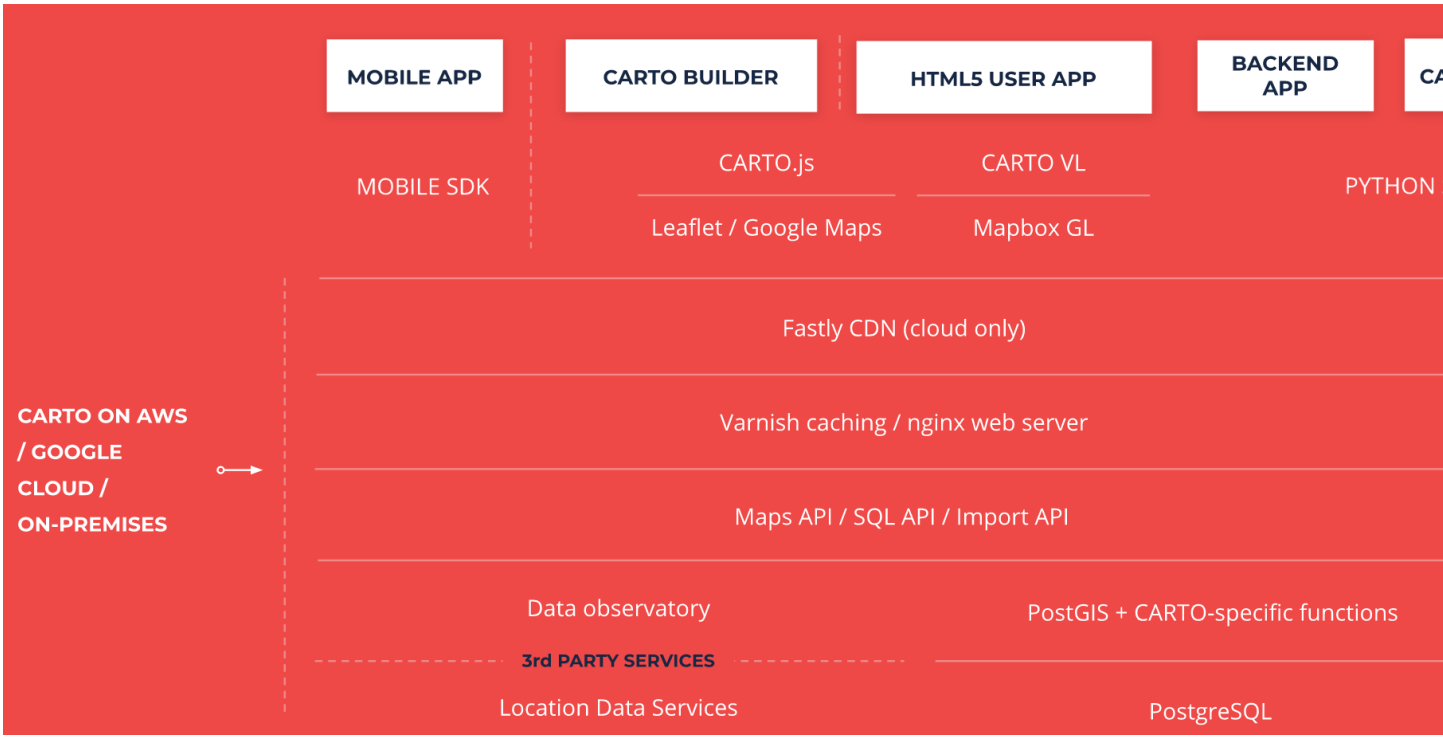
Servicios para enriquecimiento de datos, a través de fronteras, datos demográficos y otros conjuntos de datos para dar valor a los propios datos de los usuarios.

- Data connectors[6]

APIs para ingestión de datos en la plataforma

2.1.2 Arquitectura

El siguiente diagrama muestra la arquitectura de componentes simplificada de *CARTO*.



Para el objetivo de este trabajo final de máster, vamos a obviar los casos de aplicaciones móviles o HTML5 y vamos a centrarnos en *BUILDER*.

BUILDER está formado por un conjunto de tecnologías de *backend*, que están desplegadas en la nube de Amazon, Google o Azure (u *on-premise*) y un conjunto de tecnologías de *frontend* que se corresponden con librerías JavaScript que se ejecutan en el navegador.

Dentro de las tecnologías *backend* encontramos las siguientes:

[3] https://carto.com/engine/ - mayo 2019
[4] https://carto.com/location-data-services/ - mayo 2019
[5] https://carto.com/data-observatory/ - mayo 2019
[6] https://carto.com/integrations/ - mayo 2019

- PostgreSQL y PostGIS

PostgreSQL[7] es una base de datos relacional con soporte a SQL estándar distribuida con licencia libre y código abierto. PostGIS[8] es una extensión para PostgreSQL que añade soporte geoespacial a través de estructuras de datos (tipos, índices, etc.) y funciones.

CARTO utiliza PostgreSQL y PostGIS para almacenamiento de la información geoespacial generada por los usuarios y para realizar los análisis geoespaciales que permiten construir cuadros de mandos, visualizar mapas, etc.

El acceso a PostgreSQL y PostGIS está abierto a los usuarios a través del uso de las *API* de la plataforma.

Las version actuales de PostgreSQL y PostGIS utilizados por CARTO son la 10.0 y 2.4 respectivamente.

- APIs de la plataforma (maps, SQL, import, analysis, etc.)

Las APIs de la plataforma son parte de las APIs ofrecidas por *ENGINE* y utilizadas a su vez por *BUILDER* y por aplicaciones móviles o HTML5 creadas por terceros.

CARTO ofrece un conjunto amplio de APIs *REST*, JavaScript y *SDK* de desarrollo en diferentes lenguajes. A continuación se describen las más relevantes para el trabajo:

- maps API: Permite obtener teselas de los datos almacenados en PostgreSQL
- SQL API: Permite realizar consultas SQL contra PostgreSQL y PostGIS y utilizar todas las funciones disponibles incluidas las de *Location Data Services* y *Data Obervatory*
- import API: Permite importar datos en formato geoespacial
- Varnish

Varnish[9] es un acelerador de aplicaciones web, también conocido como servidor proxy de caché HTTP. Permite cachear peticiones HTTP y su contenido.

- Nginx

Nginx[10] es un servidor web HTTP.

- CDN

Una Content Delivery Network (CDN o, en español, una "Red de distribución de contenido") es un conjunto de servidores que contienen copias de una misma serie de contenidos (imágenes, vídeos, documentos, ...) y que están ubicados en puntos diversos de una red para poder servir sus contenidos de manera más eficiente.[11]

- BUILDER

BUILDER es una aplicación escrita en Ruby on Rails y JavaScript, que a través de las APIs de la plataforma permite a los usuarios finales:

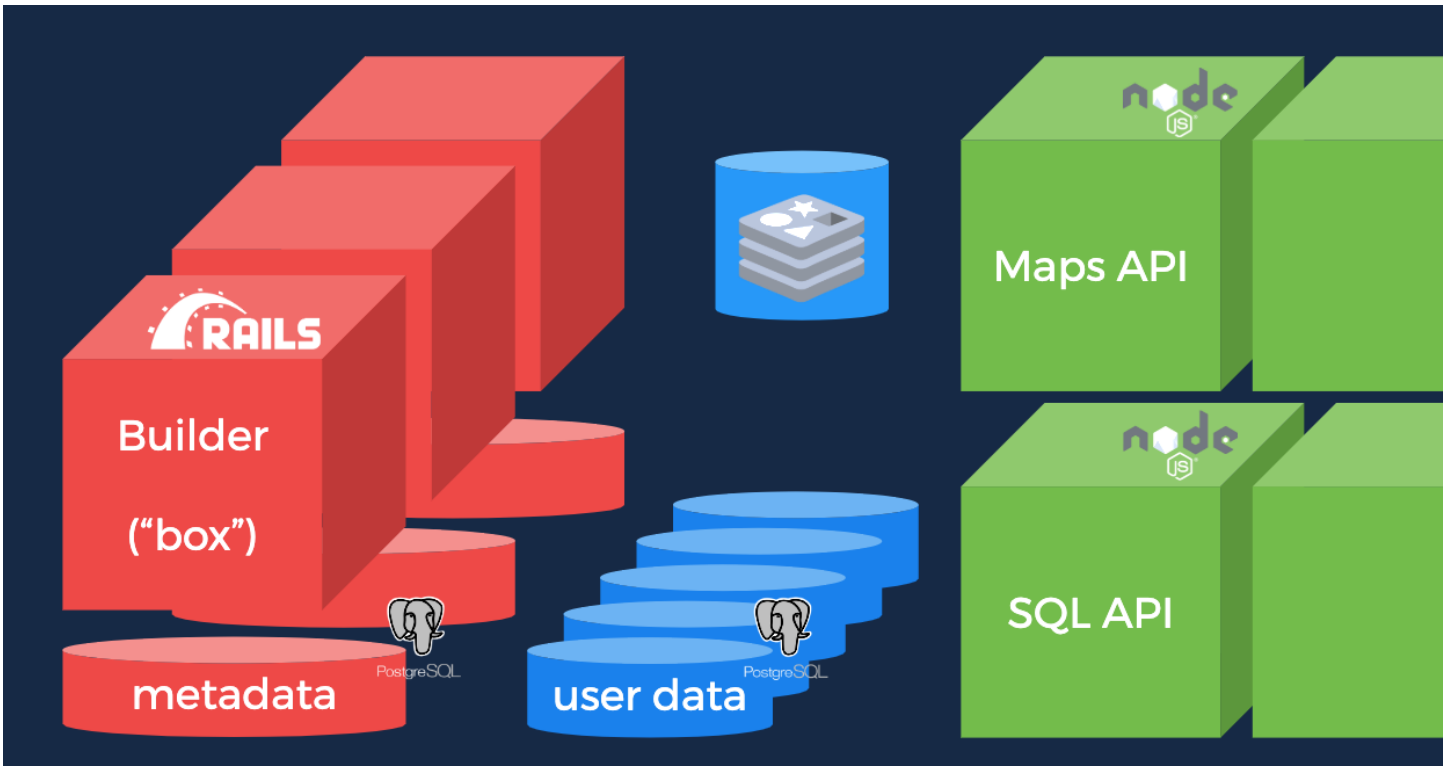
- Gestionar sus datos geoespaciales
- Gestionar sus mapas
- Definir orígenes de datos con filtros y consultas SQL
- Definir simbología a través de CartoCSS[12]
- Publicar mapas y embeberlos

Todo esto, centrado en la experiencia de usuario a través de una interfaz de usuario atractiva y fácil de usar.

[7] https://www.postgresql.org/ - mayo 2019
[8] http://postgis.net/ - mayo 2019
[9] https://varnish-cache.org/ - mayo 2019
[10] https://nginx.org/ - mayo 2019
[11] https://manueldelgado.com/que-es-una-content-delivery-network-cdn/ - mayo 2019
[12] https://carto.com/docs/carto-engine/cartocss/ - mayo 2019

2.1.3 Tecnologías backend

En lo que se refiere a tecnologías backend, el siguiente diagrama muestra un resumen de frameworks a más bajo nivel dentro de *BUILDER*.



Nos encontramos principalmente con dos frameworks de desarrollo: Por una parte, Rails[13] que se utiliza para dos soporte a servicios y APIs para la gestión de datos, tablas, mapas, visualizaciones. Y por otra parte NodeJS[14] que da soporte a APIs de alta carga de peticiones (varios cientos de millones de peticiones mensuales), para procesar, analizar y visualizar información geoespacial.

Por último, como hemos visto antes tanto la información de usuario, se almacena en PostgreSQL y PostGIS para dar soporte geoespacial.

2.2 Sistemas de almacenamiento y procesamiento Big Data

En este trabajo se estudian los siguientes sistemas de almacenamiento y procesamiento Big Data, ya que son los sistemas más utilizadas por las actuales organizaciones que usan *CARTO* como plataforma de *Location Intelligence*:

- Apache Hive[15]
- Apache Impala[16]
- Amazon Redshift[17]
- MongoDB[18]

[13] https://rubyonrails.org/ - mayo 2019
[14] https://nodejs.org/es/ - mayo 2019
[15] https://hive.apache.org/ - mayo 2019
[16] https://impala.apache.org/ - mayo 2019
[17] https://aws.amazon.com/es/redshift/ - mayo 2019
[18] https://www.mongodb.com/ - mayo 2019

- Google BigQuery[19]

En esta sección se va a hacer una breve descripción de los sistemas mencionados atendiendo a las siguientes características:

- Tipo de sistema: Si ofrece almacenamiento y procesamiento o sólo uno de ambos.
- Tipo de procesamiento: Batch (latencia del orden de minutos), interactivo (latencia del orden de decenas de segundos), tiempo real (latencia del orden de pocos segundos), etc.
- Tipo de despliegue/distribución: Nube pública, privada, SaaS, on-premises, etc.
- Interfaces de programación/consulta: SQL, SDKs en diferentes lenguajes, APIs REST, etc.
- Autenticación: Usuario y contraseña, HTTP/HTTPS, Kerberos/LDAP, OAuth, etc.
- Tipo de licencia/propietario: Software libre (Apache, GPL, etc.), propietaria (Google, Amazon, Oracle, etc.)
- Versión actual
- Driver ODBC

Para el motivo de este trabajo, no es necesario conocer otros detalles como mecanismos de replicación, particionamiento, tolerancia a fallos, etc. ya que el objetivo no consiste en administrar este tipo de sistemas.

Sin embargo, el objetivo es triple:

1. Por una parte, contar con una visión general de los sistemas con los que se va a trabajar.
2. Por otra parte, poder identificar similitudes y diferencias entre ellos.
3. Por último, abrir la puerta al soporte del mayor número posible de tecnologías de almacenamiento y procesamiento Big Data, especialmente aquellas de carácter libre.

2.2.1 Apache Hive

Apache Hive es una infraestructura de almacenamiento y procesamiento de datos almacenados sobre *HDFS* de Hadoop[20] y otros sistemas compatibles como Amazon S3[21], originalmente desarrollado por Facebook[22].

Ofrece un lenguaje de consulta basado en SQL llamado *HiveQL* que convierte las consultas en trabajos MapReduce, Tez[23] o Spark[24].

Actualmente, como gran parte de los sistemas batch es considerado un sistema *legacy*, aunque por otra parte es un sistema apliamente establecido en la industria que cuenta con gran cantidad de herramientas integradoras dentro del sistema Hadoop tales como: Pig[25], Sqoop[26], Flume[27], etc.

Se suele utilizar para procesamiento batch de ficheros almacenados en HDFS.

- Tipo de sistema: Procesamiento.
- Tipo de procesamiento: Batch.

[19] https://cloud.google.com/bigquery/ - mayo 2019
[20] http://hadoop.apache.org/ - mayo 2019
[21] https://aws.amazon.com/es/s3/ - mayo 2019
[22] https://facebook.com/ - mayo 2019
[23] https://tez.apache.org/ - mayo 2019
[24] https://spark.apache.org/ - mayo 2019
[25] https://pig.apache.org/ - mayo 2019
[26] https://sqoop.apache.org/ - mayo 2019
[27] https://flume.apache.org/ - mayo 2019

- Tipo de despliegue/distribución: Nube pública y privada (on-premises) con multitud de distribuciones (Amazon EMR[28], Cloudera[29], Hortonworks[30], MapR[31])
- Interfaces de programación/consulta: HiveQL compatible con SQL
- Autenticación: Usuario y contraseña, HTTP/HTTPS, Kerberos/LDAP
- Tipo de licencia/propietario: Apache 2.0
- Versión actual: 2.3.0
- Driver ODBC: sí

2.2.2 Impala

Apache Impala es una infraestructura de almacenamiento y procesamiento de datos almacenados sobre HDFS de Hadoop, originalmente desarrollado por Cloudera.

Apache Impala es compatible con HiveQL y utiliza la misma base de datos de metadatos para acceder a HDFS que Hive, pero a diferencia de este, cuenta con un modelo de procesamiento en memoria de baja latencia que permite realizar consultas interactivas orientadas a entornos *Business Intelligence*.

Se suele utilizar para procesamiento de ficheros almacenados HDFS con menor latencia que Hive y por tanto orientada a aplicaciones finales.

- Tipo de sistema: Procesamiento.
- Tipo de procesamiento: Interactivo.
- Tipo de despliegue/distribución: Nube pública y privada (on-premises) con multitud de distribuciones.
- Interfaces de programación/consulta: HiveQL compatible con SQL
- Autenticación: Usuario contraseña, Kerberos, otros
- Tipo de licencia/propietario: Apache 2.0
- Versión actual: 2.10.0
- Driver ODBC: sí

2.2.3 Amazon Redshift

Amazon Redshift es un almacén de datos rápido y completamente administrado que permite analizar todos los datos empleando de forma sencilla y rentable SQL estándar y las herramientas de Business Intelligence existentes.

Forma parte de la familia de servicios web de Amazon (AWS), por tanto se integra con gran parte de sus servicios, como por ejemplo Amazon S3.

Se suele utilizar para almacenar y analizar datos en entornos donde es necesaria una alta integración con otros servicios de AWS.

- Tipo de sistema: Almacenamiento y procesamiento.
- Tipo de procesamiento: Interactivo.
- Tipo de despliegue/distribución: Nube pública (Amazon Web Services)
- Interfaces de programación/consulta: SQL

[28] https://aws.amazon.com/es/emr/ - mayo 2019
[29] https://www.cloudera.com - mayo 2019
[30] https://es.hortonworks.com/ - mayo 2019
[31] https://mapr.com/ - mayo 2019

- Autenticación: Usuario y contraseña.
- Tipo de licencia/propietario: Propietario.
- Versión actual: Al ser un servicio auto-administrado por Amazon no se ofrece información de versiones
- Driver ODBC: Sí

2.2.4 MongoDB

MongoDB es una base de datos orientada a objetos que pertenece a la familia de bases de datos *NoSQL*. Está diseñada para soportar escalabilidad, particionamiento, replicación, alta disponibilidad siendo de las primeras bases de datos NoSQL en ofrecer estas características y una de las más populares en la actualidad.

Se suele utilizar como base de datos operacional y es muy popular en arquitecturas *MEAN*, en las que tanto el front como el backend están desarrollados sobre Javascript.

- Tipo de sistema: Almacenamiento y procesamiento.
- Tipo de procesamiento: Interactivo.
- Tipo de despliegue/distribución: on-premises
- Interfaces de programación/consulta: Javascript (nativo) y otros SDK con lenguajes varios.
- Autenticación: Usuario y contraseña, Kerberos/LDAP
- Tipo de licencia/propietario: AGPL v3.0
- Versión actual: 3.4
- Driver ODBC: Sí

2.2.5 Google BigQuery

Google BigQuery es el almacén de datos en la nube de Google, totalmente administrado y apto para analizar petabytes de datos.

Google BigQuery es un sistema de almacenamiento con una arquitectura *serverless* y ofrecido a modo de SaaS. Entre sus características principales destaca la integración con otros servicios de Google como Google Cloud Storage[32], el soporte de OAuth[33] y acceso a través de API REST o SDKs en diferentes lenguajes.

Se suele utilizar en entornos donde se requiere integración con otros servicios de Google y en los que se pretende evitar el coste de mantenimiento de infraestructura.

- Tipo de sistema: Almacenamiento y procesamiento.
- Tipo de procesamiento: Interactivo.
- Tipo de despliegue/distribución: SaaS
- Interfaces de programación/consulta: API REST, SDKs
- Autenticación: OAuth
- Versión actual: Al ser un servicio auto-administrado por Google no se ofrece información de versiones
- Tipo de licencia/propietario: Propietario (Google)
- Driver ODBC: Sí

[32] https://cloud.google.com/storage/ - mayo 2019
[33] https://oauth.net/ - mayo 2019

2.3 Tabla resumen

Característica	Apache Hive	Apache Impala	Amazon Redshift	MongoDB	Google BigQuery
Tipo de sistema	Procesamiento	Procesamiento	Almacenamiento Procesamiento	Almacenamiento Procesamiento	Almacenamiento Procesmiento
Tipo de procesamiento	Batch	Interactivo	Interactivo	Interactivo	Interactivo
Tipo de despliegue	Nube on-premises	Nube on-premises	SaaS	Nube on-premises	SaaS
Interfaces	SQL	SQL	SQL	SDKs, Javascript	API REST, SDKs
Autenticación	Usuario	Usuario	Usuario	Usuario	OAuth 2.0
Versión actual	2.3.0	2.10.0	▪	3.4	▪
Licencia	Libre	Libre	Propietario	Libre	Propietario
Driver ODBC	Sí	Sí	Sí	Sí	Sí

CAPÍTULO 3

Metodología y plan de trabajo

3.1 Introducción

Como hemos visto en el estudio del estado del arte de los principales sistemas de almacenamiento Big Data, nos encontramos ante un ecosistema heterogéneo en cuanto a tipos de almacenamiento, procesamiento, despliegue, etc.

Aún siendo un ecosistema tan heterógeneo, es importante definir una metodología clara y sistemática en cuanto al desarrollo de conectores Big Data para CARTO. Esto es así, porque es de esperar que este campo siga evolucionando, surgiendo nuevas tecnologías y paradigmas a los que se deba dar soporte.

Así pues, en la definición de esta metodología sistemática, debemos encontrar un nexo de unión entre todos estos sistemas y CARTO.

3.2 ¿Cómo conectar con CARTO?

De acuerdo a la arquitectura de CARTO, la integración con sistemas de terceros se puede realizar a dos niveles:

- Utilizando sus APIs, algunas de las cuales exponen interfaces para acceder directamente a todas las capacidades de PostgreSQL a través de SQL estándar.
- Utilizando las capacidades de conectividad de PostgreSQL, tales como Foreign Data Wrappers[1] (FDWs).

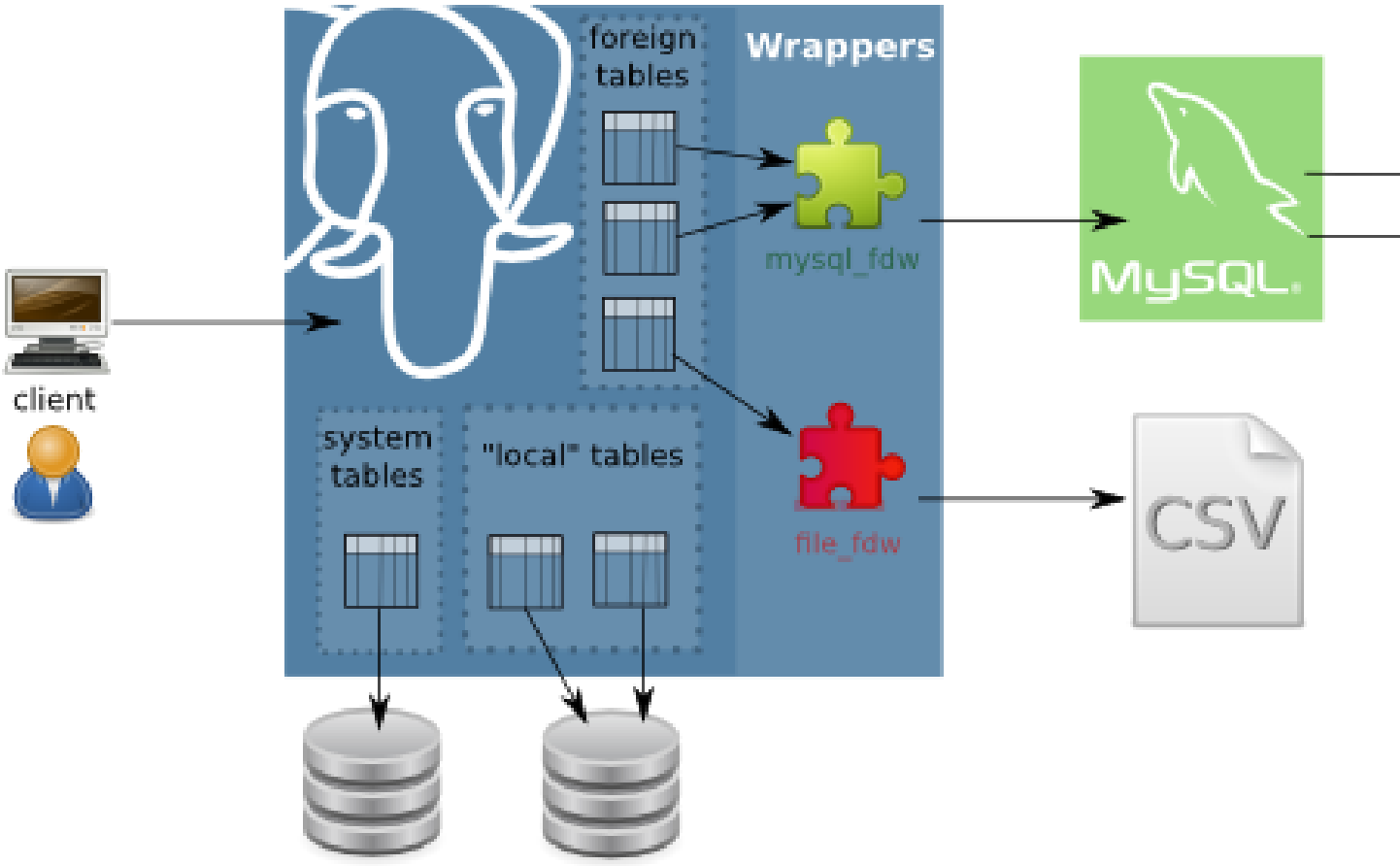
3.2.1 Una breve introducción a Foreign Data Wrappers (FDWs)

Antes de decidir qué aproximación vamos a utilizar para resolver el problema de conectar CARTO son sistemas de almacenamiento Big Data y ya que hemos mencionado FDWs como un solución válida para este problema, hagamos una pequeña introducción.

Los Foreign Data Wrappers son una funcionalidad del core de PostgreSQL que permite a desarrolladores exponer fuentes de datos externas como si fueran tablas dentro de PostgreSQL.

[1] https://carto.com/blog/postgres-fdw/ - mayo 2019

Para fuentes de datos que exponen tablas y un lenguaje de consulta SQL, FDWs actúa como un espejo de estos sistemas (Oracle, MySQL, etc.), para otros tipos de fuentes externas (APIs de redes sociales, CSVs, etc.) el FDW tiene que hacer esa conversión del modelo de datos y API de la fuente de datos de origen a tuplas de PostgreSQL.



En cualquier caso, un FDW hace que PostgreSQL actúe como una especie de *proxy* entre la aplicación cliente que está ejecutando SQLs contra PostgreSQL y la fuente de datos externa, que recibe estas peticiones y contesta en consecuencia. A la vista del cliente, está trabajando directamente contra PostgreSQL.

3.2.2 APIs vs FDWs

Analizando las virtudes y defectos de ambas aproximaciones nos encontramos con lo siguiente:

A favor de la utilización de APIs como mecanismo de integración entre CARTO y otros sistemas está la flexibilidad. Estas APIs REST, se pueden utilizar en cualquier flujo de integración. Por otra parte, como inconveniente, nos encontramos con que se requieren desarrollos concretos para cada tipo de integración.

La utilización de las capacidades nativas de PostgreSQL para conectarse con sistemas de terceros presenta a su vez ventajas e inconvenientes. Entre las ventajas, cabe destacar, que el framework de Foreign Data Wrappers, consiste en un marco bien definido y ampliamente utilizado en la industria, que además, en gran parte de sus implementaciones se basa en la utilización de drivers ODBC[2], un estándar conocido y muy extendido en los sistemas de bases de datos relacionales.

El principal defecto de esta aproximación, consiste en la necesidad de realizar una conexión directa entre sistemas de bases de datos, en este caso, desde PostgreSQL a otros (tales como Hive, Impala, MongoDB, etc.). En algún caso, esto puede comprometer la seguridad de los sistemas de bases de datos.

[2] https://docs.microsoft.com/en-us/sql/odbc/reference/what-is-odbc?view=sql-server-2017 - mayo 2019

Aún así, CARTO puede ser instalado on-premises, con lo que las organizaciones celosas de abrir una conexión fuera de su infraestrucura, podrían aprovecharse de este modo de integración.

Por último, y de nuevo haciendo referencia al estudio del estado del arte, hemos encontrado en todos los sistemas de almacenamiento Big Data dos puntos a favor de esta segunda aproximación:

- Todos los sistemas cuentan con driver ODBC
- Todos los sistemas cuentan con interfaz SQL o implementación de Foreign Data Wrapper específica para PostgreSQL

Con esto, podemos concluir que la utilización de Foreign Data Wrappers para conectar con sistemas de terceros, y en concreto, sistemas de almacenamiento Big Data, desde PostgreSQL es una solución factible y que además es susceptible de sistematizar.

3.3 Metodología

Con esta premisa, vamos a definir, una metodología, que se pueda probar y repetir, para conectar CARTO con Hive, Impala, Redshift, BigQuery, MongoDB y en definitiva, cualquier sistema de almacenamiento.

Esta metodología consta de 5 fases, que se desarrollarán para cada sistema en la siguiente sección *Desarrollo del trabajo y resultados obtenidos* y que se enumeran a continuación:

1. Despliegue de un entorno de prueba del sistema de almacenamiento Big Data
2. Búsqueda, instalación y prueba de un driver ODBC compatible
3. Búsqueda, instalación y prueba de un Foreign Data Wrapper (opcionalmente se puede utilizar la implementación base de PostgreSQL o implementar una propia)
4. Desarrollo de un conector para CARTO
5. Ingestión de datos hacia CARTO

3.4 Plan de trabajo

El trabajo final de máster consta de 3 grandes bloques en su desarrollo.

1. Despliegue de distintos sistemas de almacenamiento Big Data en la nube de Amazon.
2. Desarrollo de conectores para CARTO.
3. Ingestión de datos de prueba y creación de un dashboard de visualización de datos geoespaciales con CARTO provenientes de uno o más de los sistemas implementados.

El plan de trabajo detallado consiste en las siguientes tareas:

3.4.1 Despliegue de sistemas de almacenamiento Big Data en la nube de Amazon

Esta tarea consiste en explorar las diferentes alternativas para desplegar sistemas de almacenamiento Big Data en la nube.

El objetivo no es contar con despliegues robustos, resistentes a fallos o configurados en cluster, sino contar con diferentes entornos para realizar la ingestión de datos de prueba y conexión necesaria durante el desarrollo y demostración de los conectores para CARTO.

Para el desarrollo de este bloque se realizan las siguientes tareas:

- Aprovisionamiento de máquinas virtuales utilizando el servicio EC2 de Amazon, sobre una AMI de Ubuntu 14.04 y utilizando los servicios adicionales para configuración de *firewall*, control de acceso, disco, etc.
- Despliegue con Docker de Cloudera Quickstart para contar con instancias de Hive e Impala.
- Despliegue de una instancia de Amazon Redshift.
- Despliegue con Docker de una instancia de MongoDB.
- Configuración de una cuenta y credenciales para acceso a Google BigQuery.

3.4.2 Desarrollo de conectores para CARTO

Este bloque consiste en la implementación y prueba de los diferentes módulos que permitan conectar CARTO con los sistemas de almacenamiento mencionados previamente a través de su API de importación.

En este caso, el objetivo consiste en contar con conectores integrados en el código base de CARTO, de manera que sean incluidos en su versión *on premise*.

Para el desarrollo de deste bloque se realizan las siguientes tareas:

- Configuración de un entorno de desarrollo de CARTO utilizando Github, BASH y Vagrant.
- Desarrollo en Ruby del código necesario para los conectores.
- Configuración de las llamadas necesarias a la API REST de importación de CARTO.
- Documentación y scripts de configuración de los *drivers* necesarios para conectar con cada sistema de almacenamiento.
- Despliegue de CARTO en un servidor de *staging* en Amazon.

3.4.3 Ingestión de datos de prueba y creación de dashboard con CARTO

Una vez desplegados diferentes sistemas de almacenamiento Big Data en la nube, desarrollados los conectores y desplegada una instancia de CARTO, el último bloque consiste en realizar una pequeña demostración sobre un *dashboard* que consuma datos obtenidos de uno o más de estos sistemas desplegados.

Para el desarrollo de deste bloque se realizan las siguientes tareas:

- Ingestión de datos en los distintos sistemas de almacenamiento provistos.
- Ejecución de las llamadas necesarias mediante la API de importación de CARTO para conectar con uno o más sistemas de almacenamiento.
- Creación de un dashboard de análisis y visualización de datos geoespaciales con CARTO provenientes de uno o más de los sistemas implementados.

3.5 Metodología de trabajo

Para llevar a cabo el plan de trabajo se va a seguir una metodología de desarrollo iterativo incremental. Se trata de una metodología de desarrollo de software ágil que consiste en la ejecución de las distintas fases del proyecto en ciclos cortos de pocos días que se repiten en el tiempo, de manera que se va incrementando el valor de la solución final.

Esta metodología nos va a permitir validar en una frase temprana la solución propuesta, realizando una iteración que permita validar la integración de Hive e Impala con CARTO.

Una vez validado uno de estos sistemas de almacenamiento, se continúan realizando iteraciones cortas en las que se va dando soporte al resto de sistemas de almacenamiento propuestos, hasta contar con la solución completa.

En última instancia, se trabaja en la ingestión de datos y creación del dashboard a modo de demostración.

CAPÍTULO 4

Desarrollo del trabajo y resultados obtenidos

De acuerdo a la metodología definida en el apartado a interior, en este apartado se incluye el desarrollo de la misma para cada uno de los sistemas de almacenamiento Big Data objetivo de ser integrados con CARTO.

El objetivo es contar con un procedimiento sistemático que incluya al menos las siguientes fases, para cada sistema de almacenamiento:

1. Despliegue de un entorno de prueba del sistema de almacenamiento Big Data
2. Búsqueda, instalación y prueba de un driver ODBC compatible
3. Búsqueda, instalación y prueba de un Foreign Data Wrapper (opcionalmente se puede utilizar la implementación base de PostgreSQL o implementar una propia)
4. Desarrollo de un conector para CARTO
5. Ingestión de datos hacia CARTO

4.1 Integración de CARTO con Apache Hive

Apache Hive es una infraestructura de almacenamiento y procesamiento de datos almacenados sobre HDFS de Hadoop y otros sistemas compatibles como Amazon S3, originalmente desarrollado por Facebook.

Hive es fundamentalmente una capa de abstracción que convierte consultas SQL (escritas en un lenguaje compatible con SQL llamado HiveQL) en trabajos MapReduce, Tez o Spark.

La integración de CARTO con Apache Hive se va a realizar de acuerdo a los siguientes parámetros:

- Soporta SQL: Sí
- Driver ODBC: Sí
- Compatible con *postgres_fdw*: Sí
- Versión probada: 2.3.0
- Autenticación: Usuario y contraseña

- Distribución: Cloudera Quickstart
- Despliegue: Docker sobre AWS

4.1.1 Despliegue de un entorno de prueba de Apache Hive

Para desplegar una instancia de Apache Hive, utilizamos la imagen de Cloudera Quickstart disponible en Docker Hub, ejecutando el siguiente comando:

```
docker pull cloudera/quickstart:latest
docker run --name=cloudera -p 8888:8888 -p 10000:10000 -p 21050:21050 -v /tmp:/tmp --
↪hostname=quickstart.cloudera --privileged=true -t -i -d cloudera/quickstart /usr/
↪bin/docker-quickstart
docker exec -it cloudera /bin/bash
```

Para este caso, hay que tener en cuenta que la imagen de Cloudera Quickstart cuenta con una distribución completa de Hadoop, por tanto con la imagen se arrancan multitud de servicios y es necesaria una cantidad considerable de memoria RAM.

Para el caso en el que sea desea correr esta imagen en una máquina local o con recursos limitados, es posible que algunos de los procesos no arranquen. En estos casos, es recomendable parar el resto de procesos que no son imprescindibles para contar con una instancia de Apache Hive.

Deteniendo los siguientes servicios es posible arrancar una imagen de Cloudera Quickstart con Docker, únicamente con 2GB de memoria:

```
/etc/init.d/flume-ng-agent stop
/etc/init.d/oozie stop
/etc/init.d/spark-history-server stop
/etc/init.d/solr-server stop
/etc/init.d/flume-ng-agent stop
/etc/init.d/hive-metastore restart
/etc/init.d/hive-server2 restart
/etc/init.d/flume-ng-agent stop
/etc/init.d/hbase-master stop
/etc/init.d/hbase-regionserver stop
/etc/init.d/hbase-rest stop
/etc/init.d/hbase-solr-indexer stop
/etc/init.d/hbase-thrift stop
/etc/init.d/oozie stop
/etc/init.d/sentry-store stop
```

Después de detener los servicios es importante reiniciar la interfaz de HUE que nos permitirá realizar consultas interactivas sobre Hive con el siguiente comando: */etc/init.d/hue restart*

4.1.2 Ingestión de datos en Apache Hive

Una vez hemos desplegado Apache Hive utilizando la imagen de Cloudera Quickstart con Docker, podemos hacer una ingestión inicial de datos para posteriormente realizar las pruebas necesarias de integración con CARTO.

Abriendo una sesión de bash en el contenedor de Cloudera Quickstart:

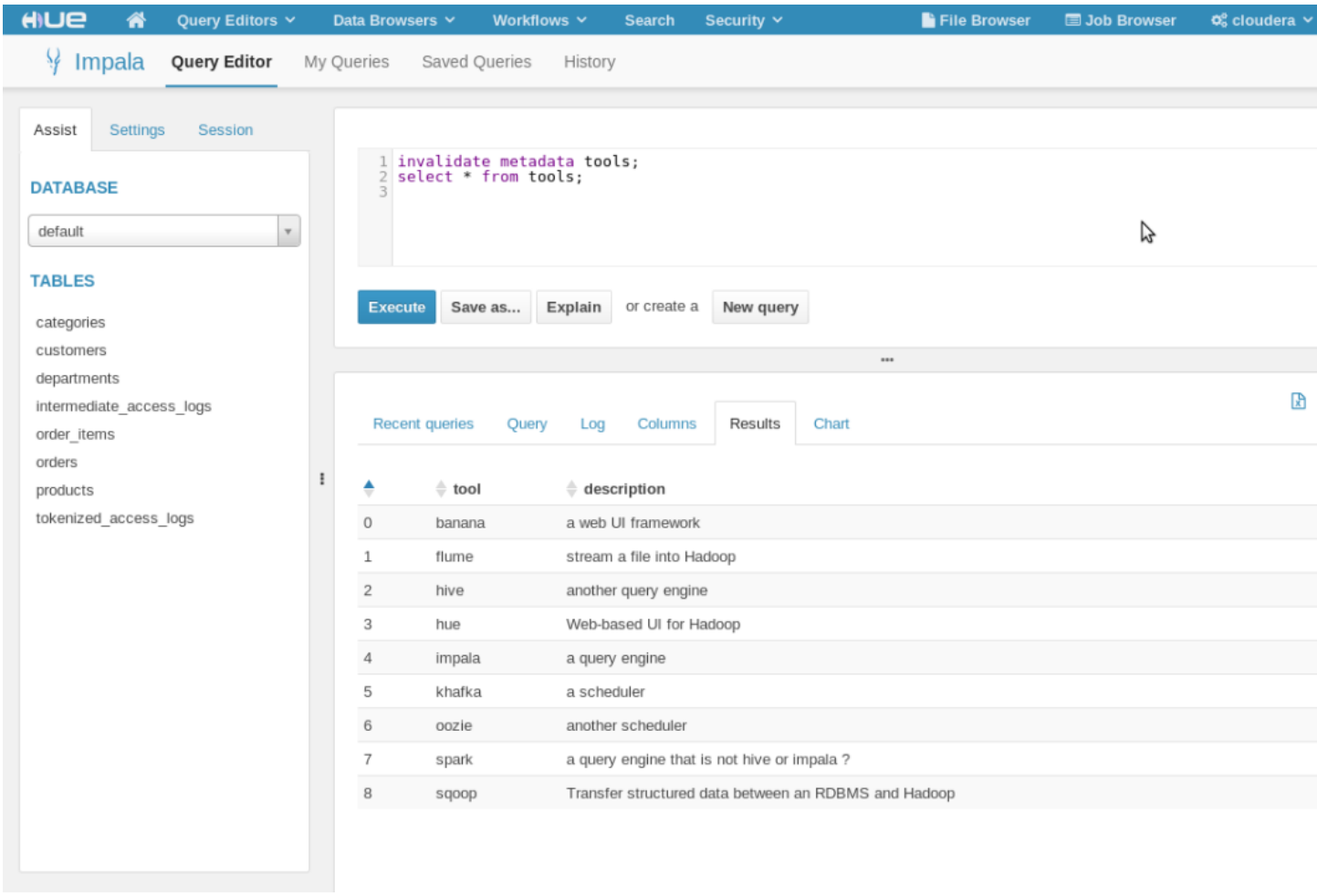
```
docker exec -it cloudera /bin/bash
```

Podemos utilizar *sqoop* para hacer una ingestión inicial de datos en Hive desde una base de datos MySQL incluida en la imagen de Cloudera con el siguiente comando:

```
sqoop import-all-tables \
    --connect jdbc:mysql://localhost:3306/retail_db \
    --username=retail_dba \
    --password=cloudera \
    --compression-codec=snappy \
    --as-parquetfile \
    --warehouse-dir=/user/hive/warehouse \
    --hive-import
```

Por último, podemos acceder a la interfaz de HUE y comprobar que efectivamente las tablas se han cargado correctamente en Hive

```
http://localhost:8888/
usr/pwd: cloudera/cloudera
```



4.1.3 Instalación y prueba de un driver ODBC para Hive

En este caso, Cloudera proporciona un driver ODBC para Hive con licencia libre que podemos instalar en distribuciones Redhat/CentOS con los siguientes comandos:

```
wget "https://downloads.cloudera.com/connectors/hive_odbc_2.5.22.1014/Linux/EL6/
↪ClouderaHiveODBC-2.5.22.1014-1.el6.x86_64.rpm"
yum install cyrus-sasl-gssapi.x86_64 cyrus-sasl-plain.x86_64
yum --nogpgcheck localinstall ClouderaHiveODBC-2.5.22.1014-1.el6.x86_64.rpm
```

4.1.4 Configuración del driver ODBC para Hive

Una vez descargado el driver ODBC para Hive es necesario editar los archivos que PostgreSQL utiliza para conocer los drivers disponibles en el sistema.

La ubicación de los archivos de configuración se puede obtener ejecutando la siguiente instrucción:

```
[root@localhost vagrant]# odbcinst -j
unixODBC 2.3.4
DRIVERS............: /opt/carto/postgresql/embedded/etc/odbcinst.ini
SYSTEM DATA SOURCES: /opt/carto/postgresql/embedded/etc/odbc.ini
FILE DATA SOURCES..: /opt/carto/postgresql/embedded/etc/ODBCDataSources
USER DATA SOURCES..: /root/.odbc.ini
SQLULEN Size.......: 8
SQLLEN Size........: 8
SQLSETPOSIROW Size.: 8
```

El comando *odbcinst* lo provee el paquete *unixODBC* que viene instalado por defecto en la distribución on-premise de CARTO.

Una vez conocemos la ubicación de los archivos de configuración, añadimos el driver de Hive a la lista de drivers disponibles:

```
printf "\n[Hive]
Description=Cloudera ODBC Driver for Apache Hive (64-bit)
Driver=/opt/cloudera/hiveodbc/lib/64/libclouderahiveodbc64.so" >> /data/production/
↪config/postgresql/odbcinst.ini
```

4.1.5 Instalación y prueba de un Foreign Data Wrapper para Hive

Una primera aproximación a la hora de probar un Foreign Data Wrapper para Hive, consiste en probar la implementación base disponible en PostgreSQL *postgres_fdw*.

En este caso, el driver ODBC de Cloudera para Apache Hive es compatible con *postgres_fdw* del que CARTO cuenta con una implementación base.

4.1.6 Desarrollo de un conector de Hive para CARTO

Puesto que el driver ODBC para Hive es compatible con *postgres_fdw* la implementación de un conector de Hive para CARTO se reduce a añadir una nueva clase al *backend* indicando cuáles son los parámetros necesarios para realizar una consulta SQL sobre Hive y configurar este conector para que sea accesible desde la API de importación de CARTO.

El código del conector *hive.rb* se adjunta en el anexo *A. Código fuente de conector para Hive*

El código de configuración del nuevo conector se adjunta en el anexo *B. Configuración del conector para Apache Hive*

4.1.7 Ingestion de datos desde Hive a CARTO

Una vez disponemos de una instalación on-premise de CARTO, con el driver ODBC de Hive correctamente instalado y configurado tanto en el sistema como en CARTO y un conector correctamente implementado, podemos realizar una ingestión de datos desde Hive a CARTO utilizando la API de importación de la siguiente manera:

```
curl -v -k -H "Content-Type: application/json"   -d '{
  "connector": {
    "provider": "hive",
    "connection": {
      "server":"{hive_server_ip}",
      "database":"default",
      "port":10000,
      "username":"{hive_user}",
      "password":"{hive_password}"
    },
    "schema": "default",
    "table": "top_order_items",
    "sql_query": "select * from order_items where price > 1000"
  }
}'   "https://carto.com/user/carto/api/v1/imports/?api_key={YOUR_API_KEY}"
```

La anterior llamada a la API de importación, crea una conexión mediante Foreign Data Wrapper desde el servidor de CARTO (en concreto desde el servidor de PostgreSQL) hacia el servidor de Hive a través del puerto 10000 (el puerto por defecto de Hive).

Una vez realizada la conexión, se crea una tabla en PostgreSQL de nombre *top_order_items* y se ejecuta la siguiente consulta en Hive para obtener los pedidos con un precio superior a mil dólares:

```
select * from order_items where price > 1000
```

Hive transformará está consulta SQL en un trabajo MapReduce y devolverá el resultado al Foreign Data Wrapper, convirtiéndose en filas de la tabla en PostgreSQL.

Esta tabla de PostgreSQL está asociada a un dataset del usuario de CARTO que lanzó la petición y por tanto puede trabajar con él, de la misma manera que con cualquier otro dataset.

4.2 Integración de CARTO con Apache Impala

Apache Impala es una infraestructura de almacenamiento y procesamiento de datos almacenados sobre HDFS de Hadoop, originalmente desarrollado por Cloudera.

Apache Impala es compatible con HiveQL y utiliza la misma base de datos de metadatos para acceder a HDFS que Hive, pero a diferencia de este, cuenta con un modelo de procesamiento en memoria de baja latencia que permite realizar consultas interactivas orientadas a entornos *Business Intelligence*.

La integración de CARTO con Apache Impala se va a realizar de acuerdo a los siguientes parámetros:

- Soporta SQL: Sí
- Driver ODBC: Sí
- Compatible con *postgres_fdw*: Sí
- Versión probada: 2.10.0
- Autenticación: Usuario y contraseña
- Distribución: Cloudera Quickstart

- Despliegue: Docker sobre AWS

4.2.1 Despliegue de un entorno de prueba de Apache Impala

Para desplegar una instancia de Apache Impala, utilizamos la imagen de Cloudera Quickstart disponible en Docker Hub, tal y como hicimos al desplegar Apache Hive. Ver *Despliegue de un entorno de prueba de Apache Hive*

4.2.2 Ingestión de datos en Apache Impala

Apache Impala es compatible con el modelo de metadatos de Apache Hive, por tanto, se pueden ingestar datos en Apache Impala tal y como se hizo para Apache Hive. Ver *Ingestión de datos en Apache Hive*

Una vez presentes los datos en el *metastore* de Hive, es necesario ejecutar la siguiente instrucción para actualizar la base de datos de metadatos de Impala:

```
invalidate metadata;
```

Dicha instrucción se puede ejecutar directamente desde la consola de Impala disponible en HUE y accesible con las siguientes credenciales:

```
http://localhost:8888/
usr/pwd: cloudera/cloudera
```

4.2.3 Instalación y prueba de un driver ODBC para Impala

El procedimiento para instalar el driver ODBC para Impala es similar al de Hive.

```
yum install -y cyrus-sasl.x86_64 cyrus-sasl-gssapi.x86_64 cyrus-sasl-plain.x86_64
wget "https://downloads.cloudera.com/connectors/impala_odbc_2.5.37.1014/Linux/EL6/
↪ClouderaImpalaODBC-2.5.37.1014-1.el6.x86_64.rpm"
yum --nogpgcheck -y localinstall ClouderaImpalaODBC-2.5.37.1014-1.el6.x86_64.rpm
```

4.2.4 Configuración del driver ODBC para Impala

Una vez descargado el driver ODBC para Impala es necesario editar los archivos que PostgreSQL utiliza para conocer los drivers disponibles en el sistema.

La ubicación de los archivos de configuración se puede obtener ejecutando la siguiente instrucción:

```
[root@localhost vagrant]# odbcinst -j
    unixODBC 2.3.4
    DRIVERS............: /opt/carto/postgresql/embedded/etc/odbcinst.ini
    SYSTEM DATA SOURCES: /opt/carto/postgresql/embedded/etc/odbc.ini
    FILE DATA SOURCES..: /opt/carto/postgresql/embedded/etc/ODBCDataSources
    USER DATA SOURCES..: /root/.odbc.ini
    SQLULEN Size.......: 8
    SQLLEN Size........: 8
    SQLSETPOSIROW Size.: 8
```

El comando *odbcinst* lo provee el paquete *unixODBC* que viene instalado por defecto en la distribución on-premise de CARTO.

Una vez conocemos la ubicación de los archivos de configuración, añadimos el driver de Impala a la lista de drivers disponibles:

```
printf "\n[Impala]
Description=Cloudera ODBC Driver for Impala (64-bit)
Driver=/opt/cloudera/impalaodbc/lib/64/libclouderaimpalaodbc64.so" >> /data/
↪production/config/postgresql/odbcinst.ini
```

4.2.5 Instalación y prueba de un Foreign Data Wrapper para Impala

Análogamente a lo que ocurría con Hive, el driver ODBC de Cloudera para Apache Impala también es compatible con *postgres_fdw* del que CARTO cuenta con una implementación base. Por tanto, no es necesaria una implementación personalizada.

4.2.6 Desarrollo de un conector de Impala para CARTO

Puesto que el driver ODBC para Impala es compatible con *odbc_fdw* la implementación de un conector de Impala para CARTO se reduce a configurar este conector para que sea accesible desde la API de importación de CARTO.

El código de configuración del nuevo conector se adjunta en el anexo *C. Configuración del conector para Apache Impala*

4.2.7 Ingestion de datos desde Impala a CARTO

Una vez más, la petición a la API de importación de CARTO es análoga a la del caso de Hive.

```
curl -v -k -H "Content-Type: application/json"   -d '{
  "connector": {
    "provider": "odbc",
    "connection": {
      "Driver":"Impala",
      "Host":"{impala_server_ip}",
      "UID":"{impala_username}",
      "PWD":"{impala_password}",
      "Schema":"default",
      "Port":"21050"
    },
    "schema": "default",
    "table": "top_order_items",
    "sql_query": "select * from order_items where price > 1000"
  }
}'   "https://carto.com/user/carto/api/v1/imports/?api_key={YOUR_API_KEY}"
```

La anterior llamada a la API de importación, crea una conexión mediante Foreign Data Wrapper desde el servidor de CARTO (en concreto desde el servidor de PostgreSQL) hacia el servidor de Impala a través del puerto 21050 (el puerto por defecto de Impala).

Una vez realizada la conexión, se crea una tabla en PostgreSQL de nombre *top_order_items* y se ejecuta la siguiente consulta en Impala para obtener los pedidos con un precio superior a mil dólares:

```
select * from order_items where price > 1000
```

En este caso, Impala no implementa el paradigma MapReduce sino que utiliza un mecanismo de procesamiento en memoria que permite la realización de consultas interactivas, por lo que la respuesta tiene una latencia menor al caso de Hive.

La tabla generada en PostgreSQL está asociada a un dataset del usuario de CARTO que lanzó la petición y por tanto puede trabajar con él, de la misma manera que con cualquier otro dataset.

4.3 Antes de continuar

Antes de continuar con el desarrollo de los siguientes conectores Big Data para CARTO, cabe destacar que hemos encontrado un procedimiento sistemático para desarrollar conectores desde sistemas de almacenamiento que cumplen las siguientes características:

- Tienen un Driver ODBC
- Soportan SQL como lenguaje de procesamiento
- Son compatibles con *postgres_fdw* o *odbc_fdw*

Tal y como hemos visto en las secciones anteriores, el desarrollo de conectores para Hive, Impala y Redshift es completamente análogo, por tanto, el mismo procedimiento sería válido para sistemas de almacenamiento que cumplan las 3 características mencionadas en esta sección.

A modo de ejemplo y sin entrar en la implementación de un conector para Amazon Redshift, a continuación se especifican las etapas necesarias para incluir este conector en una distribución de CARTO.

4.4 Integración de CARTO con Amazon Redshift

Amazon Redshift es un almacén de datos de la familia de servicios web de Amazon, completamente administrado que permite analizar datos con SQL estándar.

La integración de CARTO con Apache Redshift se realizaría de acuerdo a los siguientes parámetros:

- Soporta SQL: Sí
- Driver ODBC: Sí
- Compatible con *postgres_fdw*: Sí
- Versión probada: Amazon no proporciona información acerca del versionado de Redshift, por tanto, las pruebas realizadas son con la versión actual a fecha Mayo 2019
- Autenticación: Usuario y contraseña
- Distribución: AWS
- Despliegue: Auto-gestionado a través de la consola de administración de AWS

4.4.1 Instalación y prueba de un driver ODBC para Amazon Redshift

El procedimiento para instalar el driver ODBC para Impala es similar a los de Hive e Impala.

```
wget "https://s3.amazonaws.com/redshift-downloads/drivers/AmazonRedshiftODBC-64bit-1.
↪3.1.1000-1.x86_64.rpm"
yum --nogpgcheck localinstall AmazonRedshiftODBC-64bit-1.3.1.1000-1.x86_64.rpm
```

4.4.2 Configuración del driver ODBC para Redshift

Una vez descargado el driver ODBC para Amazon Redshift es necesario editar los archivos que PostgreSQL utiliza para conocer los drivers disponibles en el sistema.

El procedimiento es análogo a los casos de Hive e Impala:

```
printf "\n[Redshift]
Description=Amazon Redshift ODBC Driver(64-bit)
Driver=/opt/amazon/redshiftodbc/lib/64/libamazonredshiftodbc64.so" >> /data/
↪production/config/postgresql/odbcinst.ini
```

4.4.3 Instalación y prueba de un Foreign Data Wrapper para Redshift

Análogamente a lo que ocurría con Hive e Impala, el driver ODBC de Cloudera para Amazon Redshift también es compatible con *odbc_fdw* del que CARTO cuenta con una implementación base. Por tanto, tal y como ocurrió con el conector para Impala, no es necesaria una implementación personalizada.

4.4.4 Desarrollo de un conector de Impala para CARTO

Puesto que el driver ODBC para Impala es compatible con *odbc_fdw* la implementación de un conector de Redshift para CARTO se reduce a configurar este conector para que sea accesible desde la API de importación de CARTO.

El código de configuración del nuevo conector se adjunta en el anexo *D. Configuración del conector para Amazon Redshift*

4.4.5 Ingestion de datos desde Redshift a CARTO

Una vez más, la petición a la API de importación de CARTO es análoga a la del caso de Hive e Impala.

```
curl -v -k -H "Content-Type: application/json"   -d '{
  "connector": {
    "provider": "odbc",
    "connection": {
      "Driver":"Redshift",
      "Host":"{redshift_server_ip}",
      "UID":"{redshift_username}",
      "PWD":"{redshift_password}",
      "Schema":"default",
      "Port":"{redshift_port}",
    },
    "schema": "default",
    "table": "top_order_items",
    "sql_query": "select * from order_items where price > 1000"
  }
}'   "https://carto.com/user/carto/api/v1/imports/?api_key={YOUR_API_KEY}"
```

4.5 Integración de CARTO con MongoDB

MongoDB es una base de datos orientada a objetos que pertenece a la familia de bases de datos NoSQL. Se suele utilizar como base de datos operacional y es muy popular en entornos JavaScript.

- Tipo de sistema: Almacenamiento y procesamiento.
- Tipo de procesamiento: Interactivo.
- Tipo de despliegue/distribución: on-premises
- Interfaces de programación/consulta: Javascript (nativo) y otros SDK con lenguajes varios.
- Autenticación: Usuario y contraseña, Kerberos/LDAP
- Tipo de licencia/propietario: AGPL v3.0
- Versión actual: 3.4
- Driver ODBC: Sí

4.5.1 Despliegue de un entorno de prueba de MongoDB

Para el despliegue de una instancia de MongoDB vamos a utilizar esta imagen de Docker de MongoDB[1]

Ejecutamos el script de arranque del contenedor de MongoDB sobre una instancia de EC2:

```
docker run --name mongo --network=host -d -p 27017:27017 -p 28017:28017 tutum/mongodb
```

En este caso concreto, al arrancar la imagen de Docker utilizada, se crea un usuario y contraseña para acceder a la instancia de MongoDB. Para conocer el password del usuario administrador, debemos esperar a que termine de arrancar el contenedor e imprimir los logs de esta manera:

En primer lugar, obtener el ID del contenedor:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED                 ␣
↪STATUS              PORTS               NAMES
971d9c6bb9e3        tutum/mongodb       "/run.sh"           21 seconds ago      Up 18␣
↪seconds                            mongo
```

A continuación utilizar el comando *docker logs <CONTAINER ID>*, hasta obtener una salida similar a esta:

```
$ docker logs 971d9c6bb9e3
================================================================================
You can now connect to this MongoDB server using:

    mongo admin -u admin -p Ck15KQ2G4pdl --host <host> --port <port>

Please remember to change the above password as soon as possible!
================================================================================
```

4.5.2 Ingestión de datos en MongoDB

Una vez hemos obtenido las credenciales de usuario administrador en el anterior paso, podemos crear una base de datos de prueba que utilizaremos para desarrollar el conector para MongoDB sobre CARTO.

```
# open a bash session in the Docker container
docker exec -it mongo /bin/bash
# and then create a collection in the admin database
mongo -u admin -p Ck15KQ2G4pdl --authenticationDatabase 'admin'
```

(continué en la próxima página)

[1] https://hub.docker.com/r/tutum/mongodb/ - mayo 2019

(proviene de la página anterior)

```
use admin
db.createCollection("warehouse")
```

4.5.3 Instalación y prueba de un Foreign Data Wrapper para MongoDB

A diferencia de lo que ocurría en los casos de Hive, Impala o Redshift, el driver ODBC de MongoDB no es compatible con *postgres_fdw* u *odbc_fdw*, por tanto, nos encontramos con un caso en que debemos utilizar un Foreignd Data Wrapper específico.

Esto tiene sentido ya que MongoDB, es una base de datos NoSQL orientada a objetos sin interfaz SQL, por tanto la implementación de un foreign data wrapper debe ser diferente.

A la hora de elegir un FDW para MongoDB valoramos las opciones listadas en el wiki oficial de PostgreSQL[2]

Entre la lista, nos encontramos con dos FDW desarrollados con Multicorn[3] y uno desarrollado de manera nativa en C. Accediendo al código fuente de los repositorios, vemos que el más activo es el FDW nativo, por tanto, lo seleccionamos[4] como candidato para conectar a MongoDB desde PostgreSQL.

Las instrucciones de instalación a fecha mayo de 2019 de *mongo_fdw* no resultan al 100 % correctas, por tanto, adjuntamos a continuación los pasos necesarios para realizar la instalación, configuración y prueba del mismo sobre CentOS 6.9

Procedemos a ejecutar los siguientes comandos como root en la misma máquina donde tenemos PostgreSQL instalado

En primer lugar, hay que satisfacer algunas dependencias del sistema:

```
yum install -y openssl-devel patch
```

La instalación de *mongo_fdw* sólo funciona con una versión de *gcc* 4.8 o superior:

```
wget http://people.centos.org/tru/devtools-2/devtools-2.repo -O /etc/yum.repos.d/
↪devtools-2.repo
yum install devtoolset-2-gcc devtoolset-2-binutils devtoolset-2-gcc-c++ devtoolset-2-
↪gcc-gfortran -y
scl enable devtoolset-2 bash
```

Debemos asegurarnos que la versión de gcc instalada es la correcta (4.8 o superior):

```
$ gcc --version
gcc (GCC) 4.8.2 20140120 (Red Hat 4.8.2-15)
```

Descargar la última release de *mongo_fdw*, en nuestro caso la 5.0.0 compatible con CentOS:

```
wget https://github.com/EnterpriseDB/mongo_fdw/archive/REL-5_0_0.tar.gz
tar zxvf REL-5_0_0.tar.gz
cd mongo_fdw-REL-5_0_0
```

A fecha de mayo de 2019, hay un bug en una de las dependencias de *mongo_fdw*. Ver pull request.

Aplicamos el parche manualmente, sobre el archivo *autogen.sh*

A continuación compilamos e instalamos el driver nativo para MongoDB y todas las librerías necesarias:

[2] https://wiki.postgresql.org/wiki/Foreign_data_wrappers - mayo 2019
[3] https://multicorn.org/ - mayo 2019
[4] https://github.com/EnterpriseDB/mongo_fdw - mayo 2019

```
export CFLAGS=-fPIC
export CXXFLAGS=-fPIC
./autogen.sh --with-master
wget https://github.com/mongodb/mongo-c-driver/releases/download/1.6.3/mongo-c-driver-
↪1.6.3.tar.gz
tar zxvf mongo-c-driver-1.6.3.tar.gz
cd mongo-c-driver-1.6.3
./configure --prefix=/usr --libdir=/usr/lib64
make && make install
cd ..
make && make install
```

Llegados a este punto, debemos ser capaces de probar el FDW *mongo_fdw* directamente desde la consola *psql* ejecutando las siguientes instrucciones:

```
psql -U postgres
CREATE EXTENSION mongo_fdw;
CREATE SERVER mongo_server
     FOREIGN DATA WRAPPER mongo_fdw
     OPTIONS (address '192.168.99.100', port '27017');
CREATE USER MAPPING FOR postgres
     SERVER mongo_server
     OPTIONS (username 'admin', password 'Ck15KQ2G4pdl');
CREATE FOREIGN TABLE warehouse(
    _id NAME,
    warehouse_id int,
    warehouse_name text,
    warehouse_created timestamptz)
    SERVER mongo_server
        OPTIONS (database 'admin', collection 'warehouse');
INSERT INTO warehouse values (0, 1, 'UPS', '2014-12-12T07:12:10Z');
SELECT * FROM warehouse WHERE warehouse_id = 1;
```

Reemplazar los attributos 'address', 'password', etc. de acuerdo a la instancia local de MongoDB

4.5.4 Desarrollo de un conector de MongoDB para CARTO

Puesto que el driver ODBC para MongoDB no es compatible directamente con *postgres_fdw* la implementación de un conector de MongoDB pasa por crear una nueva clase en el *backend* de CARTO que permita esta conexión entre PostgreSQL y Mongo utilizando el driver ODBC previamente instalado.

El código del conector *mongo.rb* se adjunta en el anexo *E. Código fuente de conector para MongoDB*

El código de configuración del nuevo conector se adjunta en el anexo *F. Configuración del conector para MongoDB*

4.5.5 Ingestion de datos desde MongoDB a CARTO

Una vez más, la petición a la API de importación de CARTO es similar a la del caso de Hive e Impala.

```
curl -v -k -H "Content-Type: application/json"   -d '{
  "connector": {
    "provider": "mongo",
    "connection": {
      "username":"admin",
      "password":"Ck15KQ2G4pdl",
```

(continué en la próxima página)

(proviene de la página anterior)

```
      "server": "192.168.99.100",
      "database": "admin",
      "port":"27017",
      "schema": "warehouse"
    },
    "table": "warehouse",
    "columns": "_id NAME,   warehouse_id int,   warehouse_name text,   warehouse_
↪created timestamptz"
  }
}'   "https://carto.com/user/carto/api/v1/imports/?api_key={YOUR_API_KEY}"
```

En este caso, debido a la implementación de *mongo_fdw* debemos incluir un atributo más en la petición para definir las columnas de la tabla que queremos importar desde MongoDB a PostgreSQL (y en definitiva a CARTO).

Nos encontramos en este caso, ante un conector para el que hemos tenido que instalar un Foreign Data Wrapper customizado, pero cuyo comportamiento en última instancia es similar a los anteriores, ya que podemos importar datos a CARTO con una simple petición a la API de importación.

4.6 Integración de CARTO con Google BigQuery

Google BigQuery es el almacén de datos en la nube de Google, totalmente administrado y apto para analizar petabytes de datos.

El conector de Google BigQuery para CARTO es un ejemplo de implementación que utiliza autenticación OAuth y para la que además se ha desarrollado una interfaz de usuario.

La integración de CARTO con Google BigQuery se va a realizar de acuerdo a los siguientes parámetros:

- Soporta SQL: Sí
- Driver ODBC: Sí
- Compatible con *postgres_fdw*: Sí
- Versión probada: 2.3.0
- Autenticación: Google no proporciona información acerca del versionado de BigQuery, por tanto, las pruebas realizadas son con la versión actual a fecha Mayo 2019
- Distribución: SaaS
- Despliegue: SaaS

4.6.1 Despliegue de un entorno de prueba de Google BigQuery

En contraposición a otros sistemas de base de datos, Google BigQuery es una base de datos SaaS completamente auto-gestionada por Google siguiendo el paradigma *serverless*. Así que para obtener un entorno de pruebas de Google BigQuery, simplemente debemos habilitarlo con nuestra cuenta de Google.

Google ofrece una capa gratuita para BigQuery (a fecha mayo de 2019), con unos límites más que suficientes para realizar pruebas y desarrollos: 1TB por mes en lecturas e importaciones/exportaciones ilimitadas.

No se especifican detalles de cómo habilitar una cuenta de Google BigQuery, ya que es un procedimiento totalmente auto-descriptivo desde la consola de administración de Google Cloud[5].

[5] https://cloud.google.com/bigquery/docs/enable-transfer-service - mayo 2019

4.6.2 Ingestión de datos en Google BigQuery

De nuevo el proceso para ingerir datos en BigQuery esta perfectamente documentado en este enlace[6]

Lo que sí es interesante comentar es cuál es la jerarquía de datos en Google BigQuery, ya que luego influye en cómo se realizan las consultas.

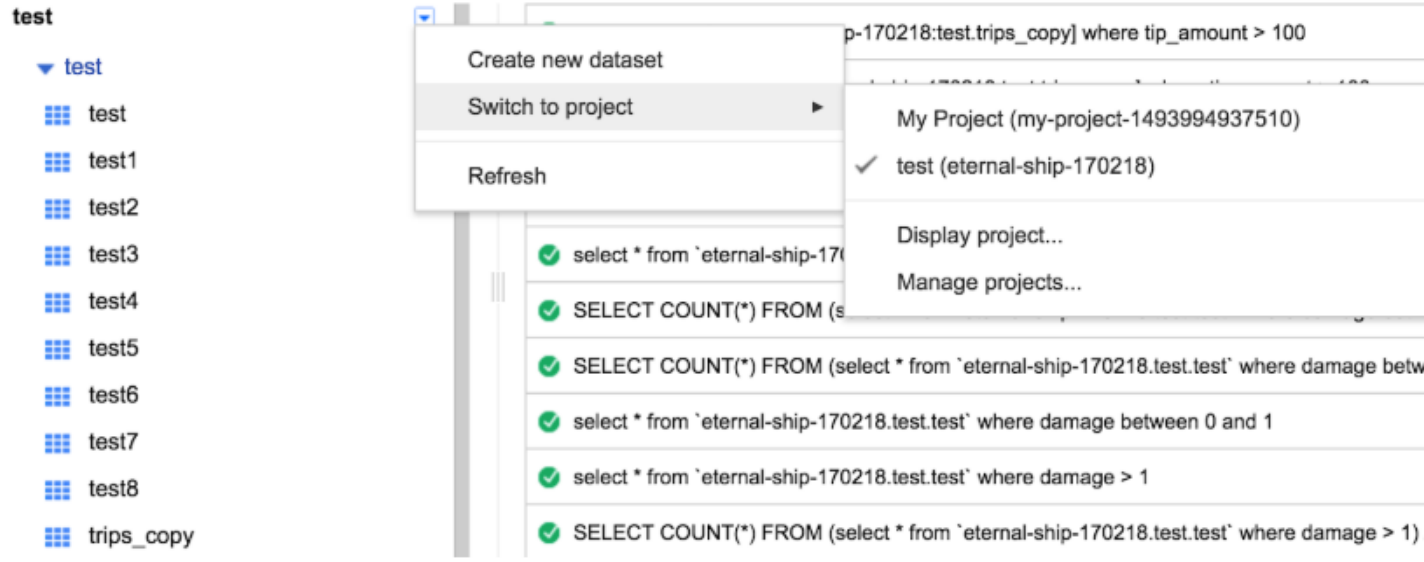
La jerarquía en BigQuery está compuesta de los siguientes niveles:

1. ID global de proyecto de Google Cloud
2. ID de proyecto de Google BigQuery
3. ID de dataset
4. ID de tabla

Las relaciones que se establecen dentro de la jerarquía son las siguientes:

- Un proyecto de Google Cloud puede tener múltiples proyectos de Google BigQuery.
- Un proyecto de Google BigQuery puede tener múltiples datasets.
- Un dataset puede tener múltiples tablas.
- Las tablas contienen la información y son una abstracción similar a las tablas disponibles en otros sistemas de bases de datos relacionales, como PostgreSQL.

Para el siguiente caso:



Contaríamos con las siguientes entidades:

- ID global de proyecto de Google Cloud: *eternal-ship-170218*
- ID de proyecto de Google BigQuery: *test*
- ID de dataset: *test*
- IDs de tablas: *test1, test2, trips_copy, etc.*

[6] https://cloud.google.com/bigquery/docs/loading-data - mayo 2019

4.6.3 Instalación y prueba de un driver ODBC Google BigQuery

Google proporciona un driver ODBC oficial para Google BigQuery, desarrollado por un proveedor externo, Simba. El procedimiento de instalación es similar al de otros drivers ODBC que hemos visto en esta sección (Hive, Impala, Redshift, etc.)

```
wget https://storage.googleapis.com/simba-bq-release/odbc/
↪SimbaODBCDriverforGoogleBigQuery64-2.0.6.1011.tar.gz
tar zxvf SimbaODBCDriverforGoogleBigQuery64-2.0.6.1011.tar.gz -C /opt
chown postgres:postgres /opt/simba
```

4.6.4 Configuración del driver ODBC para BigQuery

Una vez descargado el driver ODBC para GoogleBigQuery es necesario editar los archivos que PostgreSQL utiliza para conocer los drivers disponibles en el sistema.

La ubicación de los archivos de configuración se puede obtener ejecutando la siguiente instrucción:

```
[root@localhost vagrant]# odbcinst -j
    unixODBC 2.3.4
    DRIVERS............: /opt/carto/postgresql/embedded/etc/odbcinst.ini
    SYSTEM DATA SOURCES: /opt/carto/postgresql/embedded/etc/odbc.ini
    FILE DATA SOURCES..: /opt/carto/postgresql/embedded/etc/ODBCDataSources
    USER DATA SOURCES..: /root/.odbc.ini
    SQLULEN Size.......: 8
    SQLLEN Size........: 8
    SQLSETPOSIROW Size.: 8
```

El comando *odbcinst* lo provee el paquete *unixODBC* que viene instalado por defecto en la distribución on-premise de CARTO.

Una vez conocemos la ubicación de los archivos de configuración, añadimos el driver de BigQuery a la lista de drivers disponibles:

```
printf "\n[BigQuery]
Description = Simba ODBC Driver for Google BigQuery (64-bit)
Driver = /opt/simba/googlebigqueryodbc/lib/64/libgooglebigqueryodbc_sb64.so" >> /data/
↪production/config/postgresql/odbcinst.ini
```

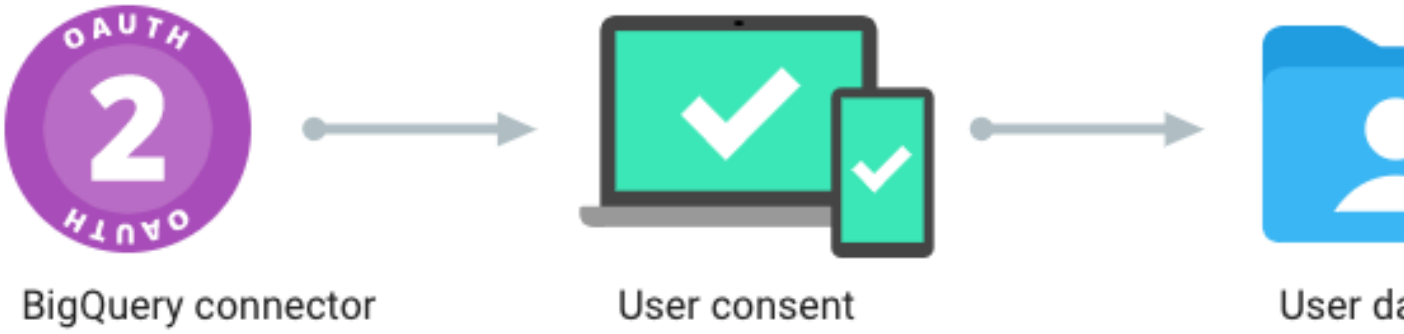
4.6.5 Configuración de OAuth para BigQuery

BigQuery soporta OAuth[7] 2.0 como mecanismo de autorización de aplicaciones. Actualmente, el soporte a OAuth en BigQuery, cuenta con dos modos de funcionamiento:

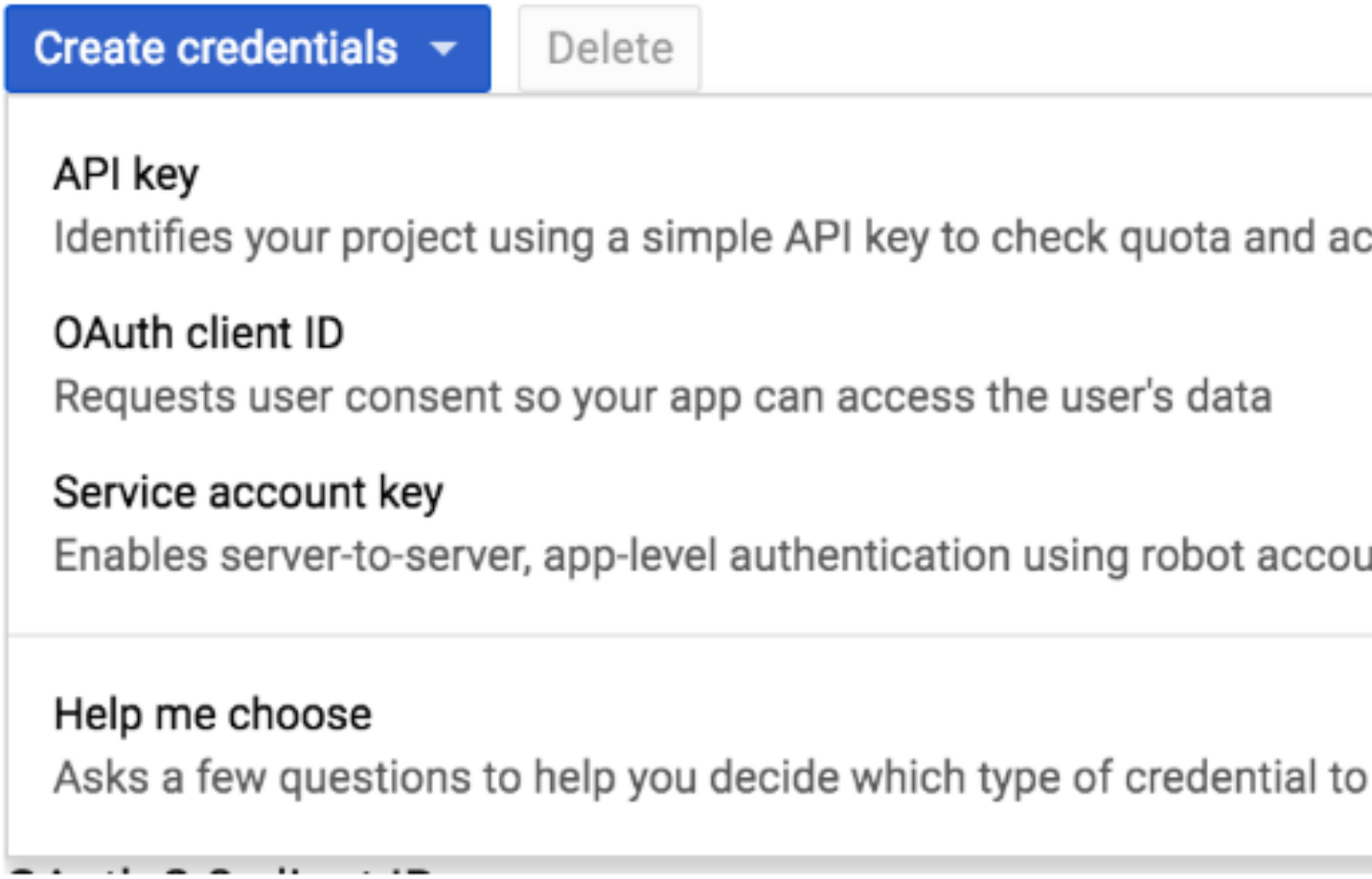
- Autenticación de usuario: Autentica contra una cuenta de usuario de Google obteniendo un *refresh token* que es temporal
- Autenticación de servicio: Autentica una aplicación a través de una *clave privada de servicio*, clave que debe ser descargada desde la consola de autenticación de Google Cloud.

Para este caso concreto, vamos a utilizar el mecanismo de autenticación de usuario, que consiste en configurar la aplicación con una clave privada generada desde la consola de administración de Google[8]

[7] https://oauth.net/2/ - mayo 2019
[8] https://console.cloud.google.com/apis/credentials - mayo 2019



La obtención de los correspondientes *client_id* y *client_secret* de configuración, así como la URL de callback, donde recibir el token de autorización, se realiza desde la consola de autenticación de Google[9] utilizando el enlace OAuth client ID.



A continuación, esta configuración debe estar presente en el archivo *app_config.yml* de CARTO[10]:

```
oauth:
    bigquery:
      application_name:     ''
```

(continué en la próxima página)

[9] https://console.cloud.google.com/apis/credentials - mayo 2019
[10] https://github.com/CartoDB/cartodb/blob/master/config/app_config.yml.sample - mayo 2019

(proviene de la página anterior)

```
      client_id:            ''
      client_secret:        ''
      callback_url:         '{your_server_url}/api/v1/imports/service/bigquery/oauth_
↪callback'
```

4.6.6 Instalación y prueba de un Foreign Data Wrapper para Google BigQuery

Una primera aproximación a la hora de probar un Foreign Data Wrapper para Google BigQuery, consiste en probar la implementación base disponible en PostgreSQL *postgres_fdw*.

En este caso, el driver ODBC de Google BigQuery es compatible con *postgres_fdw* del que CARTO cuenta con una implementación base.

4.6.7 Desarrollo de un conector de Google BigQuery para CARTO

El desarrollo de un conector para Google BigQuery es un caso especial de conector ya que debemos tratar con el flujo de autenticación de OAuth. Como hemos comentado anteriormente, en este desarrollo sólo se va a implementar la autenticación de usuario, que implica, por una parte añadir las interfaces necesarias para que el usuario realice la autenticación y por otra parte escribir el código *backend* necesario para recibir un token de autorización en la URL de callback.

El código del conector *bigquery.rb* se adjunta en el anexo *G. Código fuente de conector para BigQuery*

El código de configuración del nuevo conector se adjunta en el anexo *H. Configuración del conector para BigQuery*

El resto del código del conector, por su complejidad (en cuanto a número de ficheros e interfaces de usuario), no se adjunta en el anexo, pero se puede consultar en la siguiente rama de CARTO[11].

4.6.8 Ingestion de datos desde Google BigQuery a CARTO

En este caso, contamos con dos opciones a la hora de utilizar el conector.

1. Desde la API de importación de BUILDER

En este modo de funcionamiento, el usuario debe gestionar su token de OAuth de la siguiente manera:

En primer lugar, generando unas claves pública y privada para OAuth, como se ha descrito anteriormente.

A continuación, editando el archivo *simba.googlebigqueryodbc.ini* incluyendo las claves de acceso.

Se debe obtener un token de Oauth con la siguiente instrucción:

```
https://accounts.google.com/o/oauth2/auth?scope=https://www.googleapis.com/auth/
↪bigquery&response_type=code&redirect_uri=urn:ietf:wg:oauth:2.0:oob&client_id=YOUR_
↪CLIENT_ID&hl=en&from_login=1&as=76356ac9e8ce640b&pli=1&authuser=0
```

Para por último utilizar el script *get_refresh_token.sh* junto con el token de OAuth y las claves de autenticación generadas, para obtener el *refresh token* que se debe incluir en cada petición a Google BigQuery.

```
./get_refresh_token.sh AUTHENTICATION_TOKEN
```

Una vez disponemos de un *refresh token* se puede utilizar la API de importación de CARTO de la siguiente manera.

[11] https://github.com/CartoDB/cartodb/tree/bq-connector - mayo 2019

```
curl -v -k -H "Content-Type: application/json"   -d '{
  "connector": {
    "provider": "odbc",
    "connection": {
      "Driver": "BigQuery",
      "OAuthMechanism": 1,
      "Catalog": "eternal-ship-170218",
      "SQLDialect": 1,
      "RefreshToken": "1/nW8ZTOOrDHEuazfajXszSgd2b_X4cKSWM6xulLiP0rykdv-
↪VHAzJTWLiXLi81VFu,"
    },
    "table": "order_items",
    "sql_query": "select * from `eternal-ship-170218.test.test` limit 1;"
  }
}'   "https://carto.com/user/carto/api/v1/imports/?api_key={YOUR_API_KEY}"
```

Tanto los parámetros de conexión como el atributo *sql_query* se envían al *backend* de CARTO.

Este crea las entidades necesarias en PostgreSQL para hacer una conexión a Google BigQuery a través de un FDW que en última instancia utiliza el driver ODBC para realizar la consulta SQL con los parámetros de autenticación necesarios.

2. Desde la interfaz de importación de BUILDER

CAPÍTULO 5

Espías en el cielo: Analizando con CARTO datos de vuelos almacenados en BigQuery

Un periódico quiere realizar una investigación sobre vuelos de vigilancia realizados en Estados Unidos, concretamente en la costa oeste, por parte del FBI.

Para ello, tiene a su disposición una suscripción a FlightRadar24[1]. FlightRadar24 es un servicio que permite a miles de colaboradores, subir datos de vuelos en tiempo real, utilizando el Sistema de Vigilancia Dependiente Automática (ADS-B), un sistema obligatorio para aviones que vuelan el espacio aéreo de Estados Unidos.

El sistema ADS-B, emite periódicamente la posición de un avión, obtenida a través de la navegación por satélite. Junto a la posición (longitud, latitud) se adjuntan algunos metadatos como son:

- *flight_id*: ID del vuelo en cuestión, que permite cruzarlo con datos provenientes de aerolíneas comerciales, por ejemplo.
- *timestamp*: *epoch* en el que se emitió la posición.
- *altitude*: Altitud en metros
- *speed*: Velocidad en millas por hora.

El tamaño de la información recogida por FlightRadar24 es del orden de *petabytes* y está en contínuo aumento. Actualmente, ofrece un servicio de suscripción a través de Google BigQuery, de manera que terceras partes pueden acceder a la información tanto en tiempo real como histórica a través de una cuenta de Google.

El objetivo del periódico, es poner a disposición del público general de una investigación, que incluya una herramienta visual (en este caso un mapa) que permita demostrar que estos vuelos de vigilancia se están produciendo y un breve perfil demográfico de la población afectada.

El mapa estará siempre actualizado con datos de las últimas 24 horas y será interactivo, siendo posible efectuar filtros sobre determinadas áreas geográficas.

Para el desarrollo de este trabajo, vamos a utilizar lo mejor de dos mundos. Por una parte, las capacidades de almacenamiento masivo de Google BigQuery que nos permite acceder a datos históricos de vuelos y por otra parte CARTO como plataforma de análisis geoespacial, para realizar un análisis sobre los datos que nos permita verificar las hipótesis del periódico.

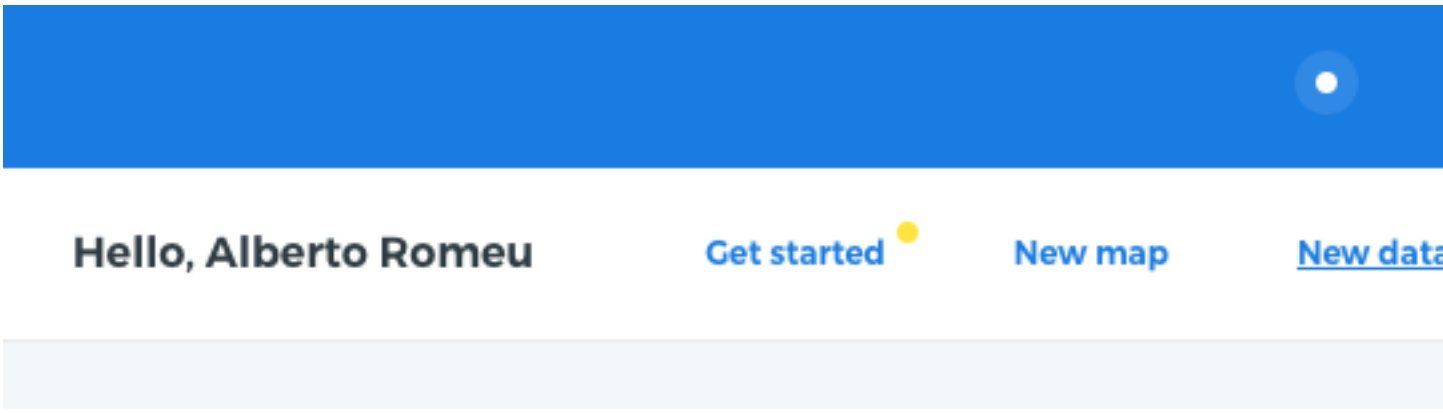
[1] https://www.flightradar24.com/60,15/6 - mayo 2019

A continuación, se describen los pasos seguidos.

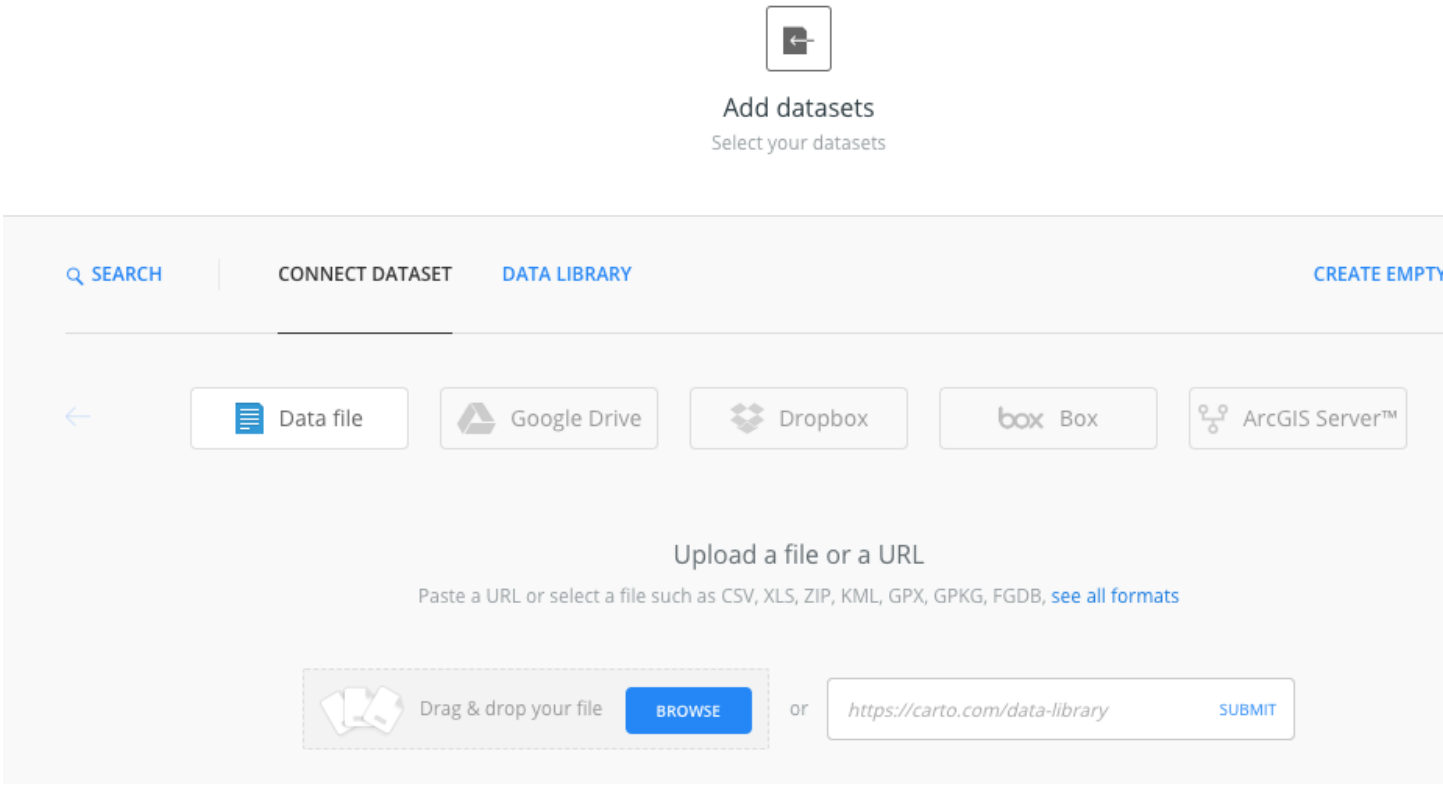
5.1 Ingestión de datos

Sincronizar CARTO con la fuente de datos de FlightRadar24 en BigQuery (últimas 24 horas)

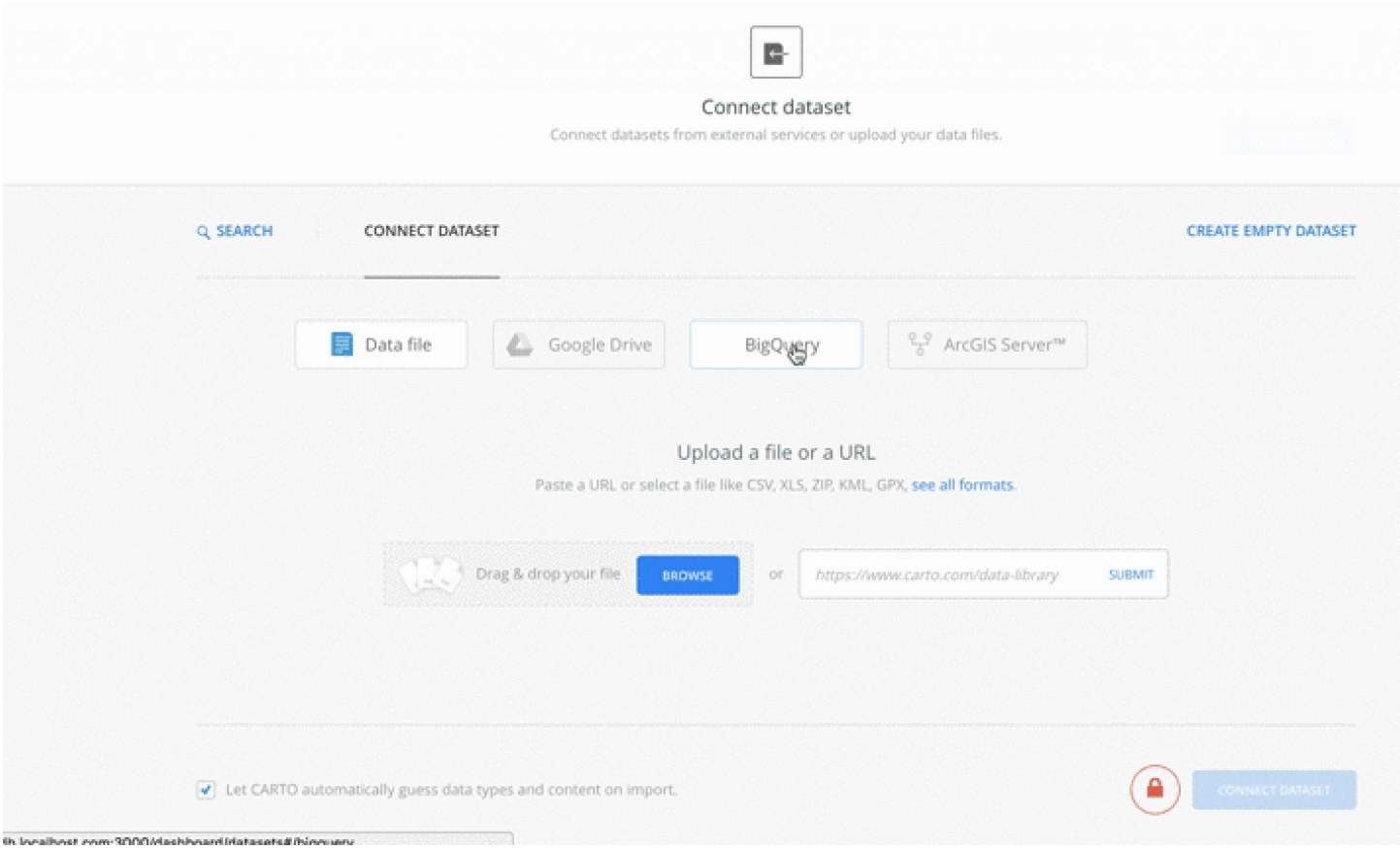
Accedemos a nuestro dashboard de CARTO y hacemos clic en *New dataset*



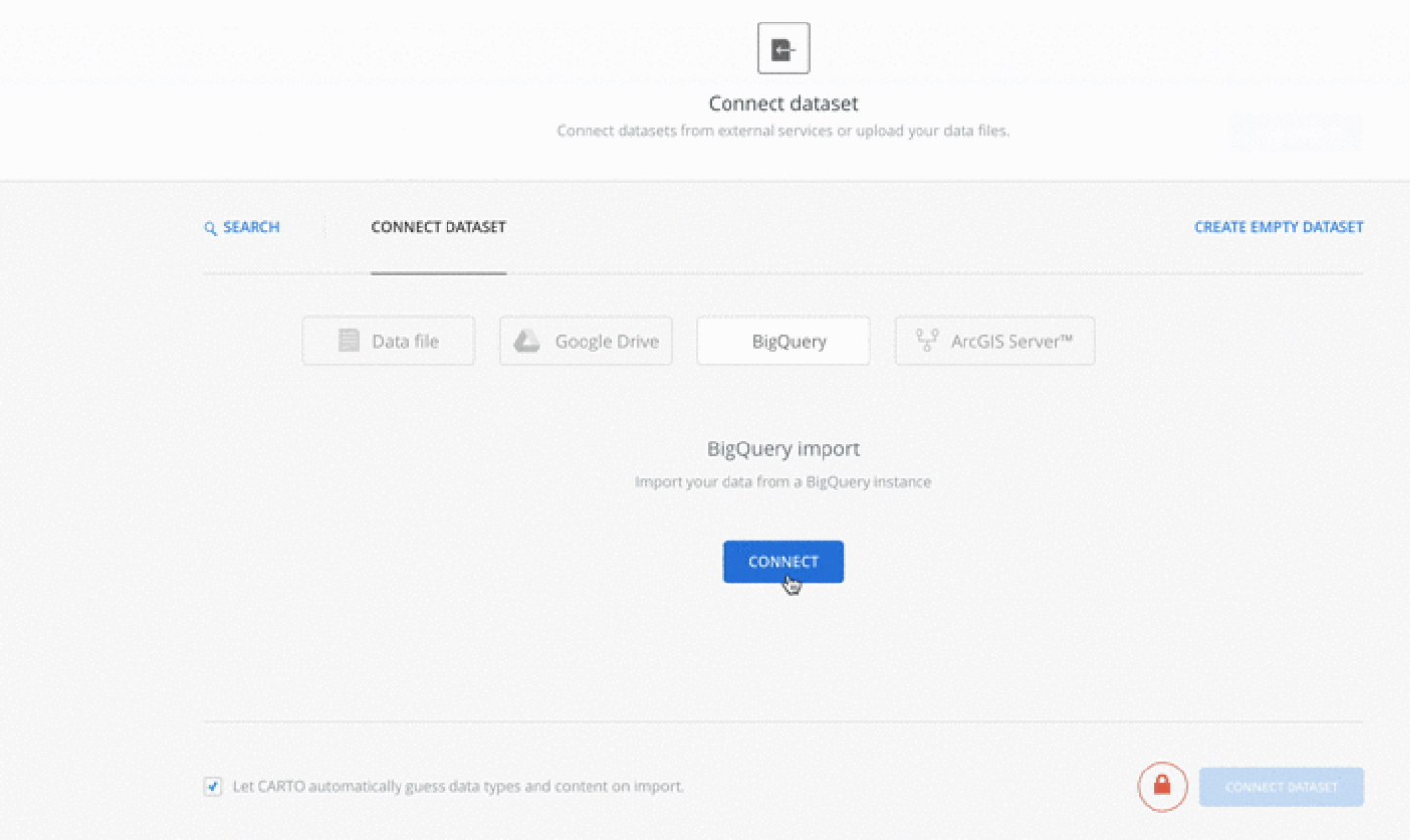
Nos aparece una interfaz con todas las fuentes de datos disponibles para conectar



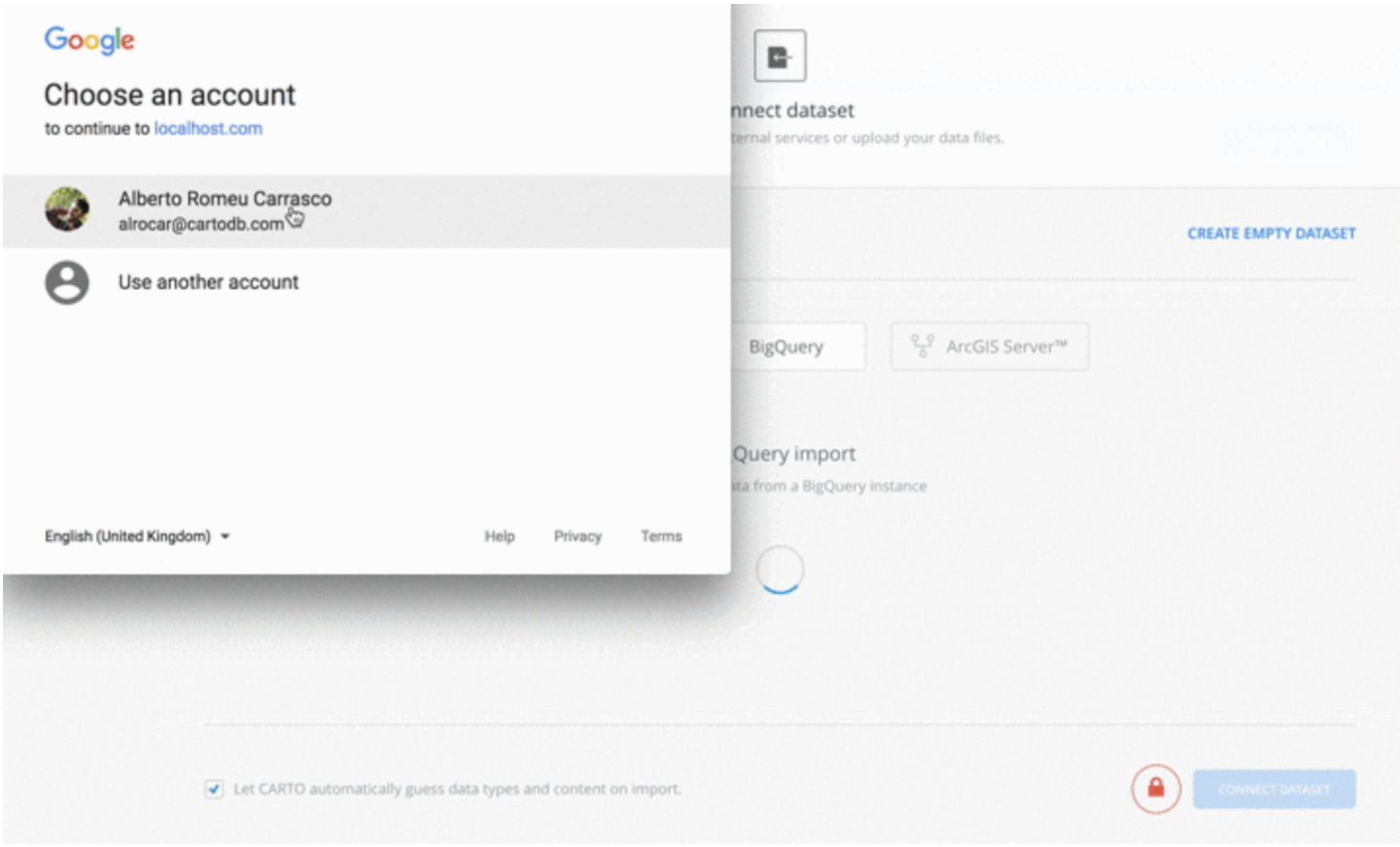
Seleccionamos el conector de BigQuery



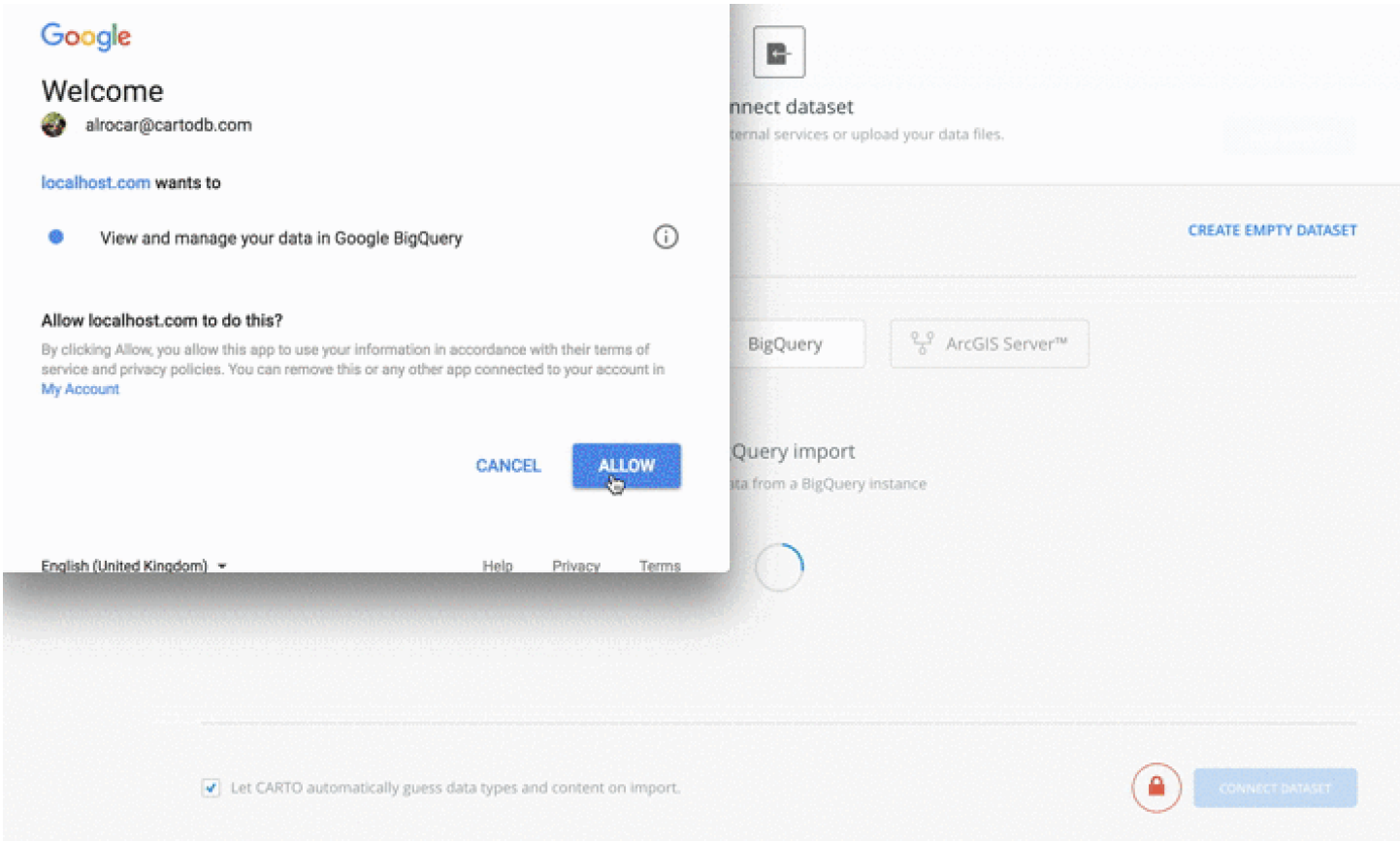
Hacemos clic en *CONNECT*



Se nos muestra un diálogo para seleccionar nuestra cuenta de Google

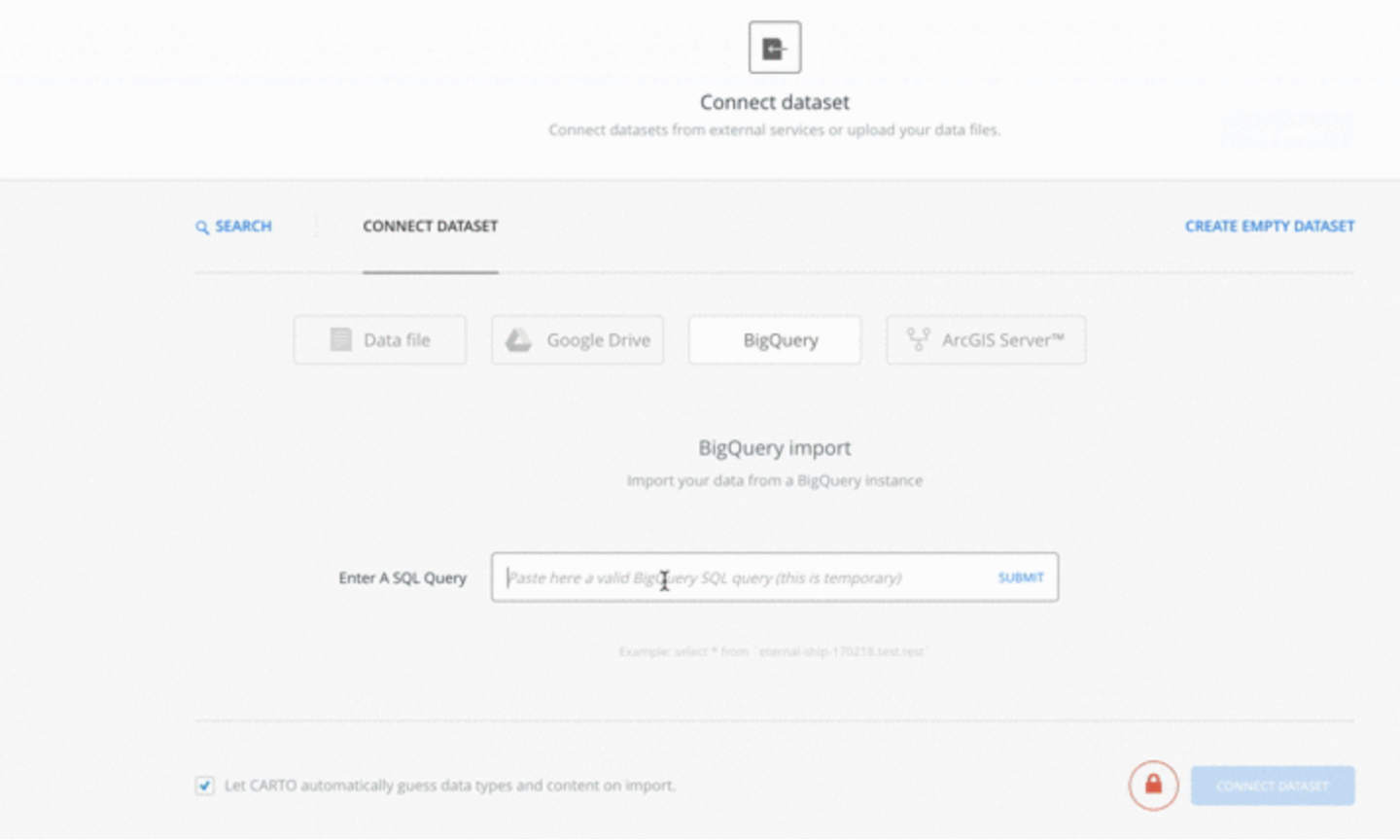


Autorizamos a CARTO a conectar con BigQuery

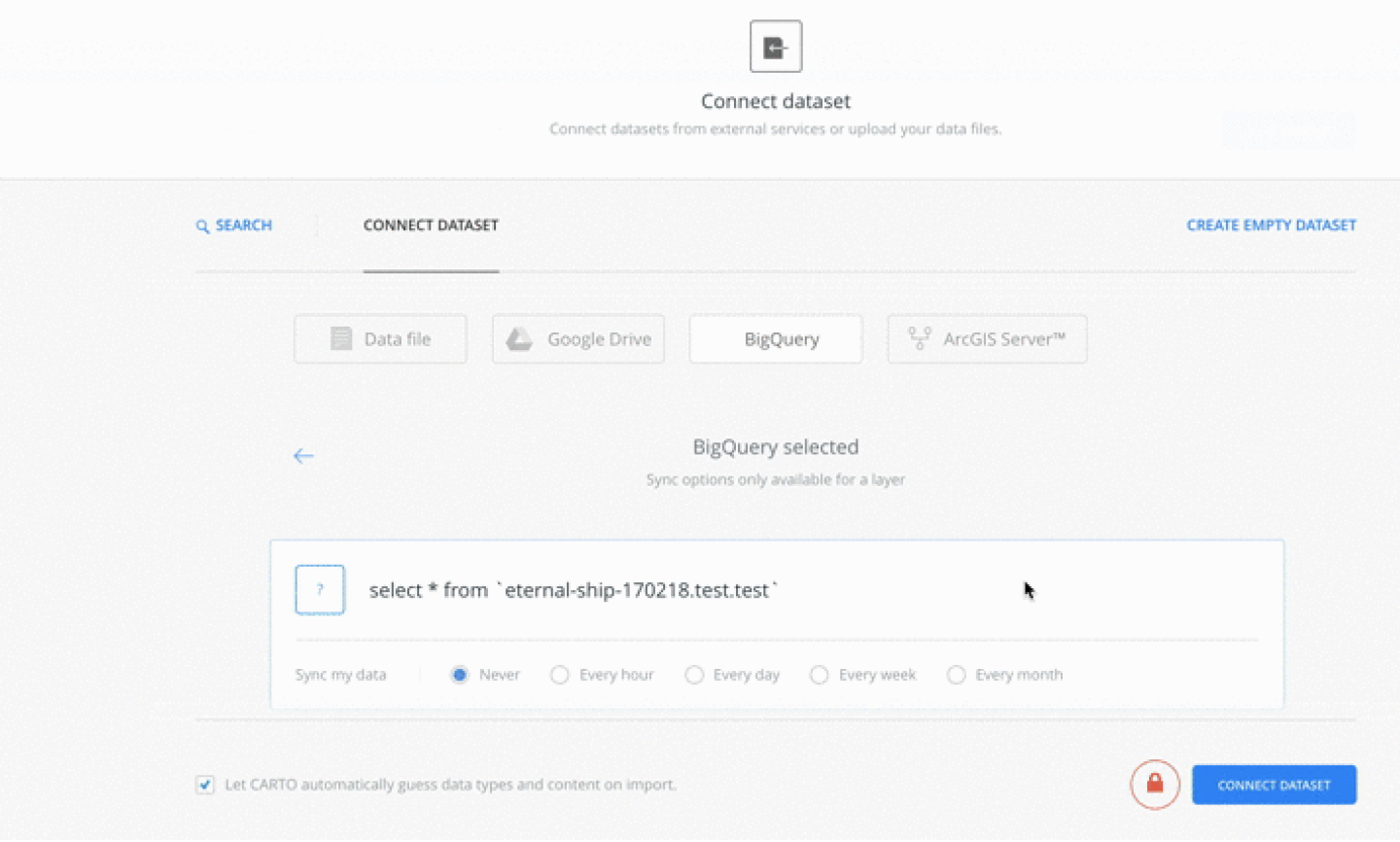


Escribimos una SQL para extraer datos desde BigQuery. En este caso queremos los datos de las últimas 24 horas

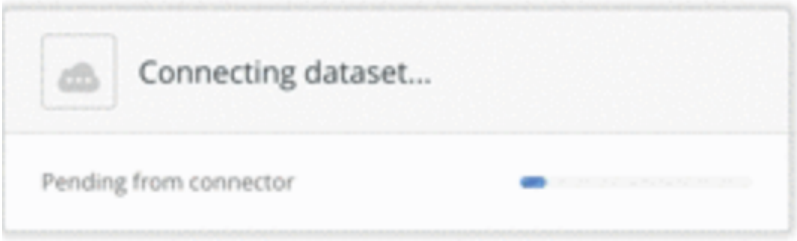
```
SELECT *
FROM `flighradar24.flights.flights_1`
WHERE TIMESTAMP <= CURRENT_TIMESTAMP
  AND TIMESTAMP >= CURRENT_TIMESTAMP - interval '24 hour'
```



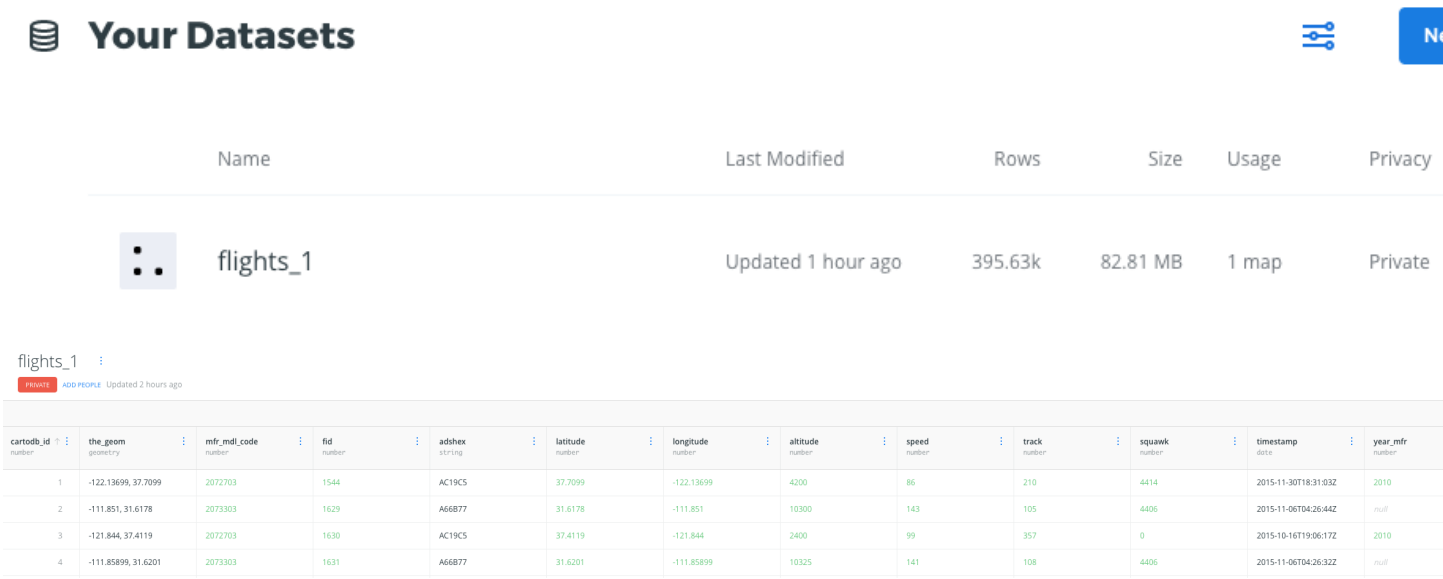
Seleccionamos la opción de *Sync my data* cada día. De esta manera los datos y visualizaciones relacionadas siempre estarán automáticamente actualizados a las últimas 24 horas de información disponible en BigQuery.



Por último, hacemos clic en *SUBMIT*. En este momento, se creará una conexión desde el PostgreSQL de la cuenta de CARTO a BigQuery, se ejecutará la consulta en BigQuery y se creará un nuevo dataset en CARTO con la información de FlightRadar24, que se actualizará diariamente.



Finalmente obtenemos un dataset en nuestra cuenta con la información que necesitamos para empezar el análisis

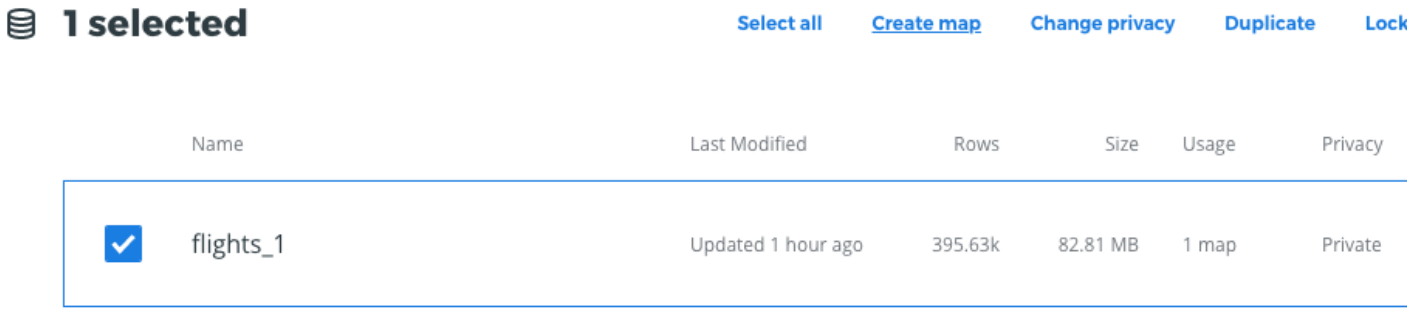


Como se puede observar en la imagen el dataset, contiene dos columnas *latitude*, *longitude* que CARTO ha transformado automáticamente en una columna de tipo *geometry* de PostGIS, que nos permitirá hacer análisis geoespacial.

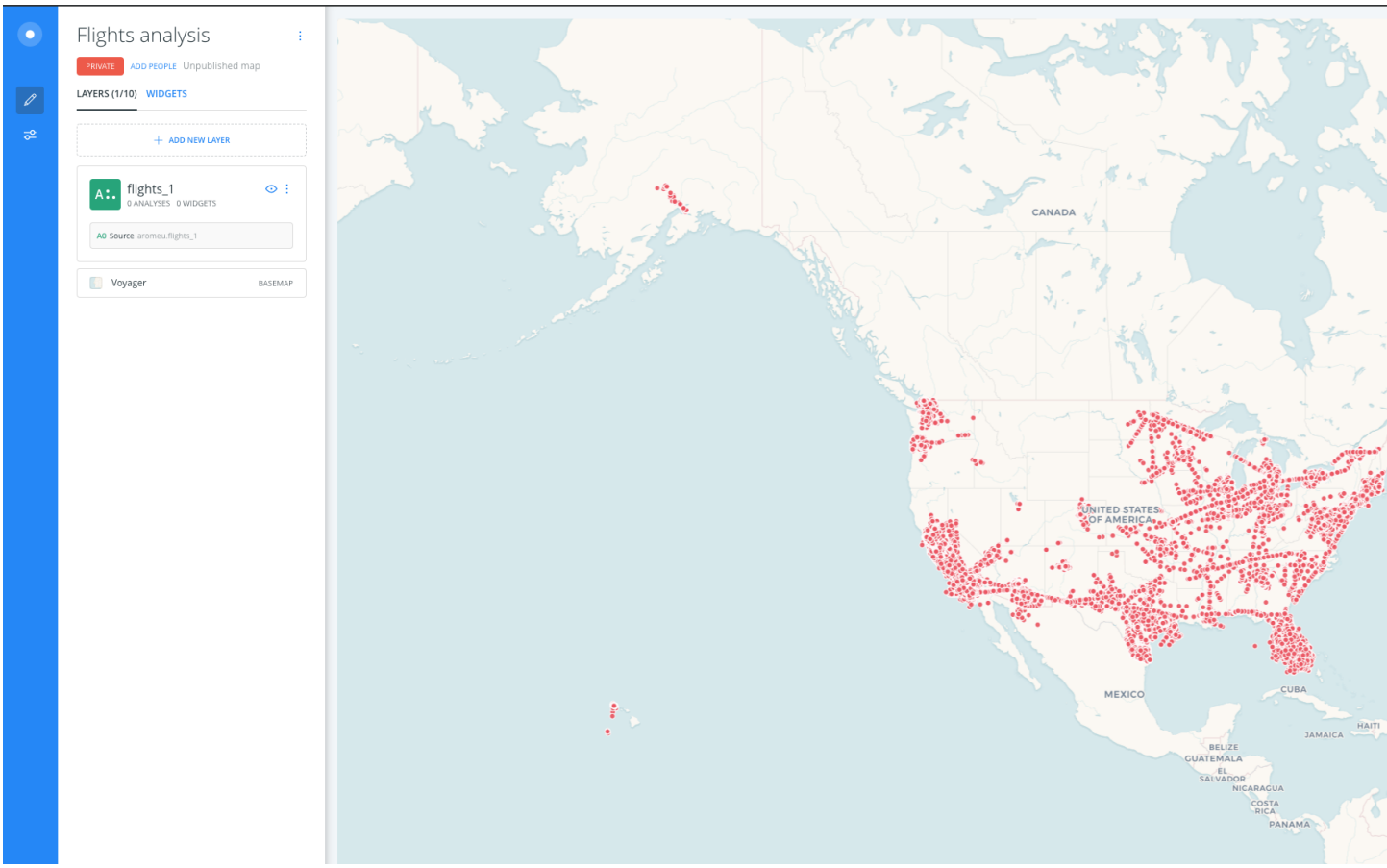
5.2 Análisis visual

Comenzamos con el análisis visual de los datos.

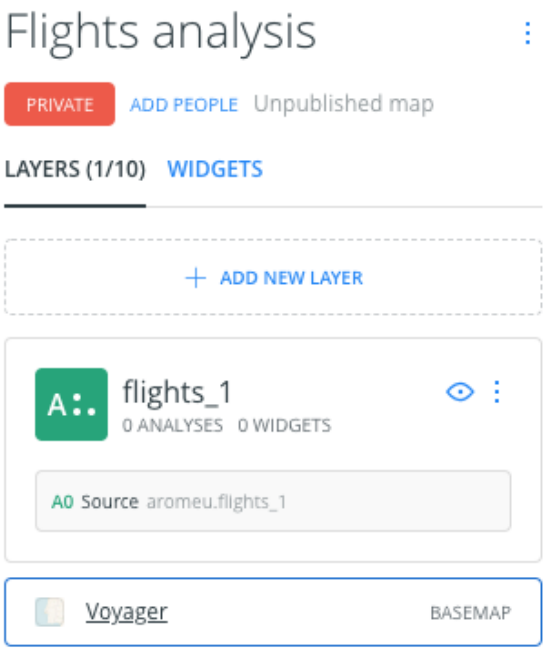
Desde el dashboard seleccionamos el dataset *flights_1* y hacemos clic en *Create Map*

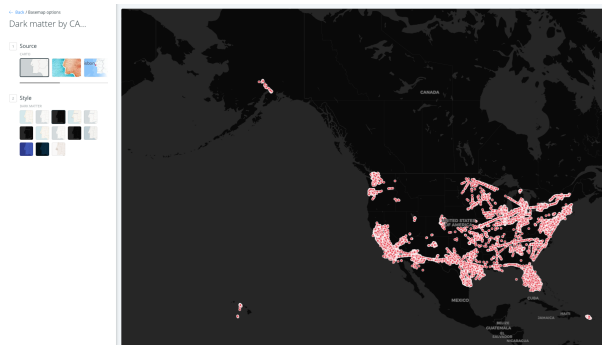


Nos aparece un mapa en *BUILDER* con el siguiente aspecto. Cada punto representa una posición GPS de un vuelo.

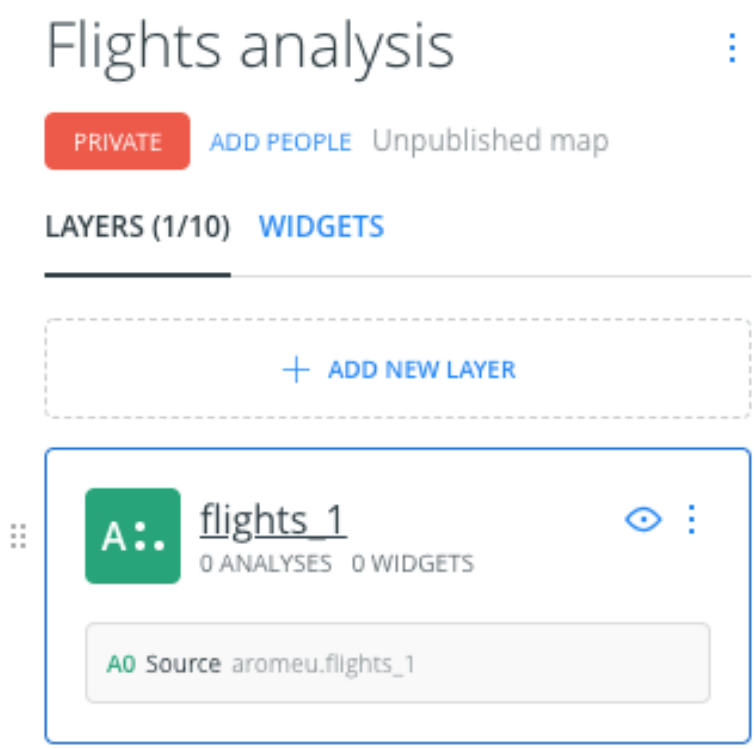


Lo primero que vamos a hacer es cambiar el mapa base a *Dark Matter*. Este mapa base nos da un mayor contraste cuando queremos encontrar patrones en datasets con gran cantidad de datos.

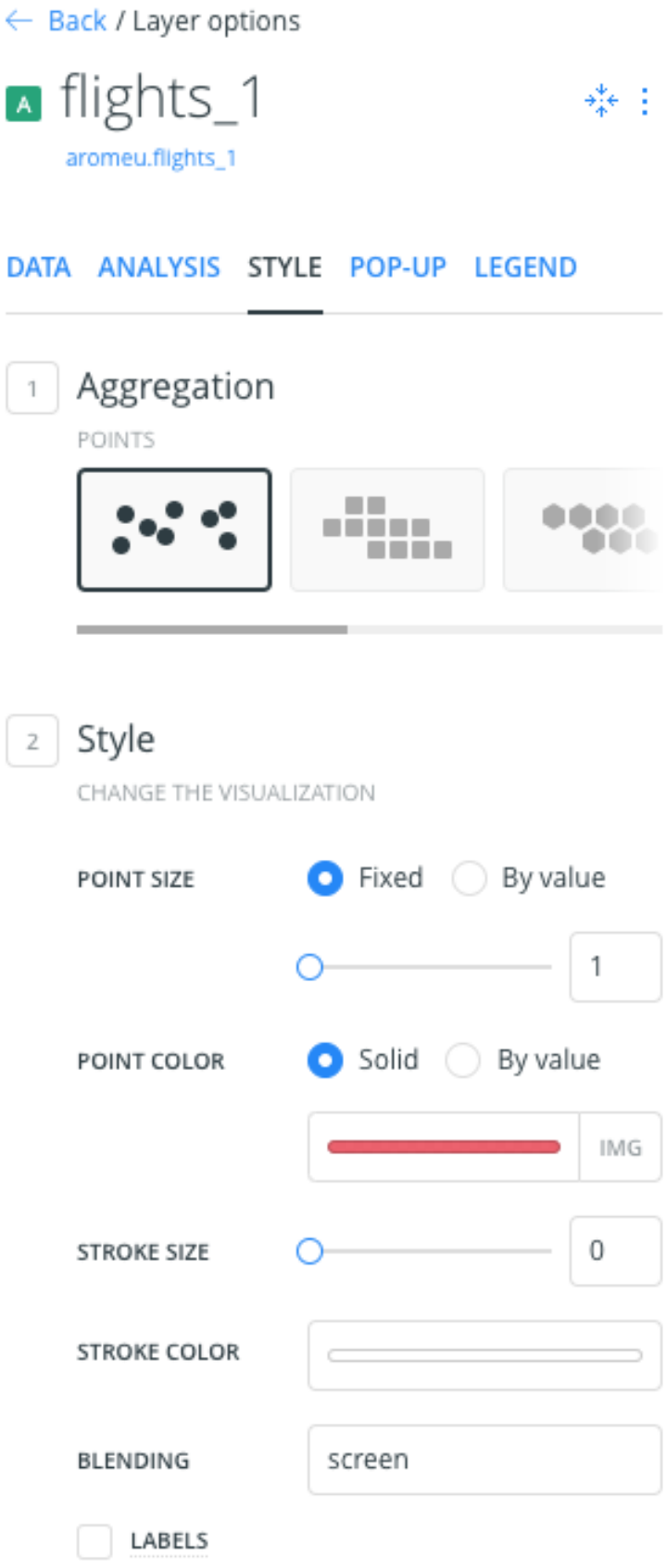




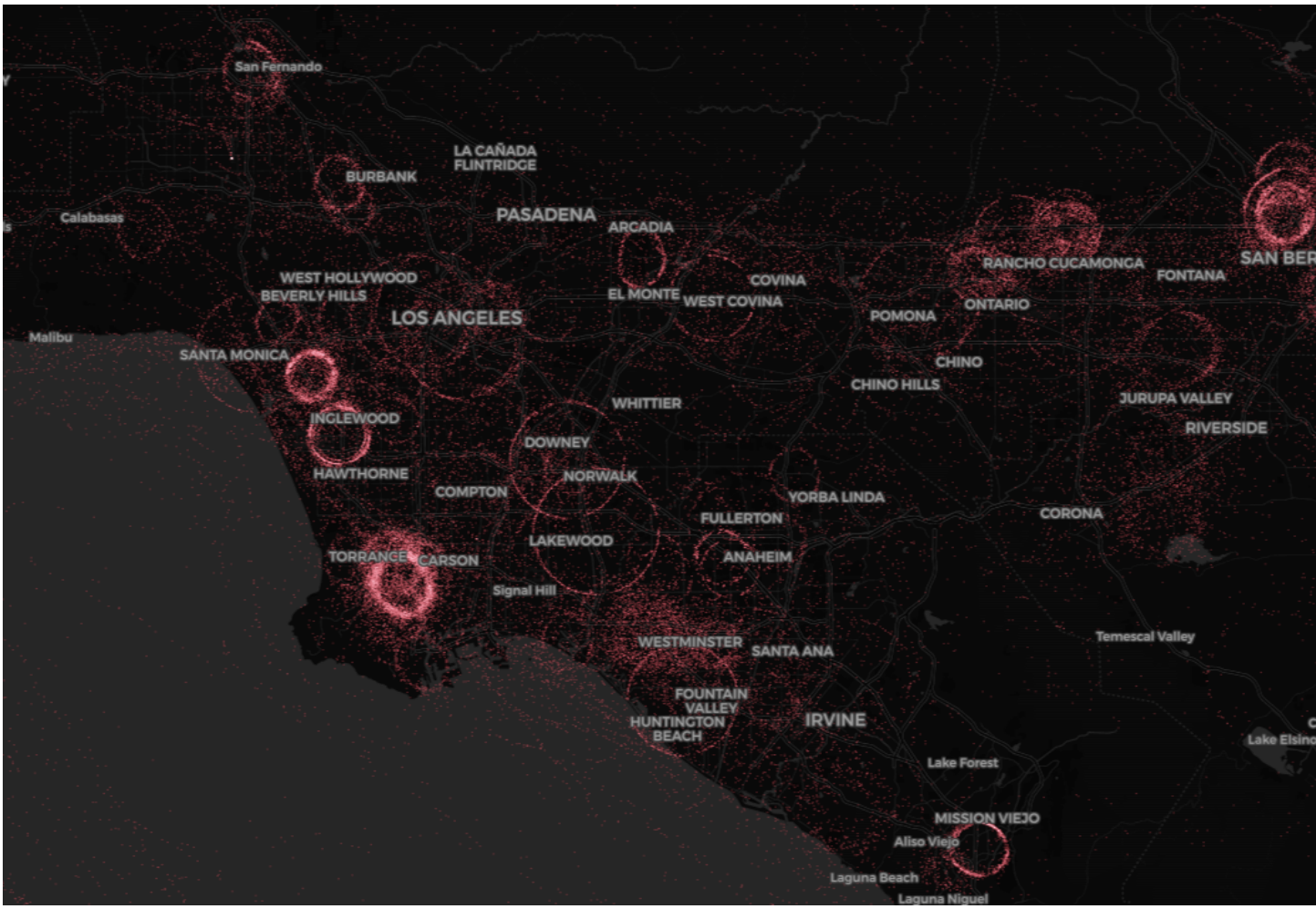
A continuación, centramos el mapa en la costa oeste (Los Angeles) y hacemos clic sobre el título del dataset, para ajustar las propiedades de estilo.



En el mapa actual, sólo se ve una maraña de puntos, pero no somos capaces de identificar ningún patrón. Ajustamos los siguientes valores:

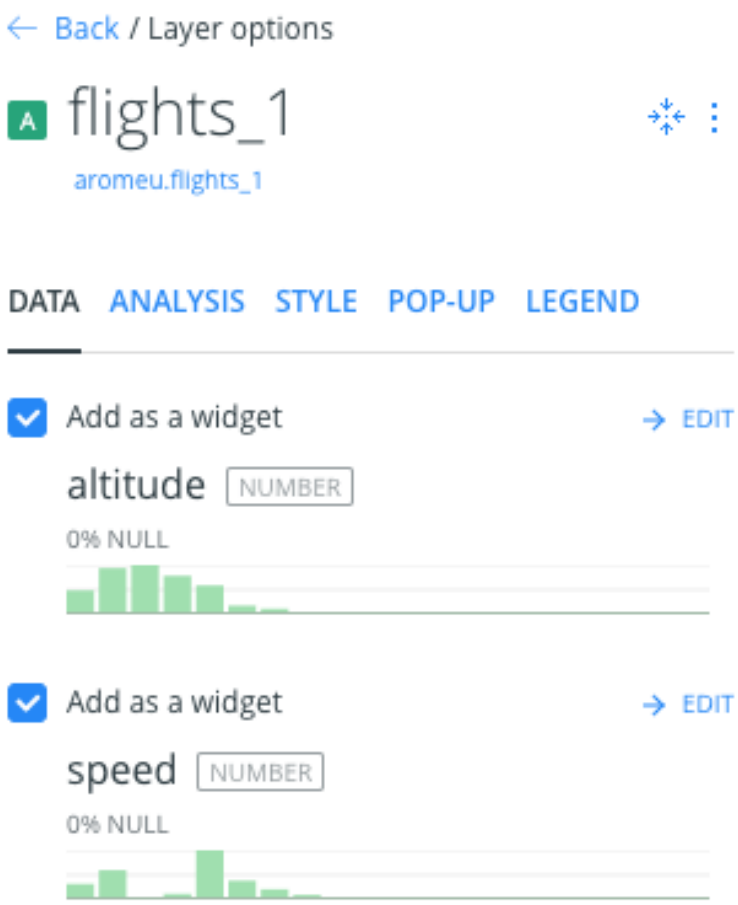

Back / Layer options
A
flights_1
aromeu.flights_1
DATA
ANALYSIS
STYLE
POP-UP
LEGEND
1
Aggregation
POINTS
2
Style
CHANGE THE VISUALIZATION
POINT SIZE
Fixed
By value
1
POINT COLOR
Solid
By value
IMG
STROKE SIZE
0
STROKE COLOR
BLENDING
screen
LABELS

Claramente somos capaces de visualizar círculos, que parecen indicar que ha habido vuelos de vigilancia, en determinadas zonas.



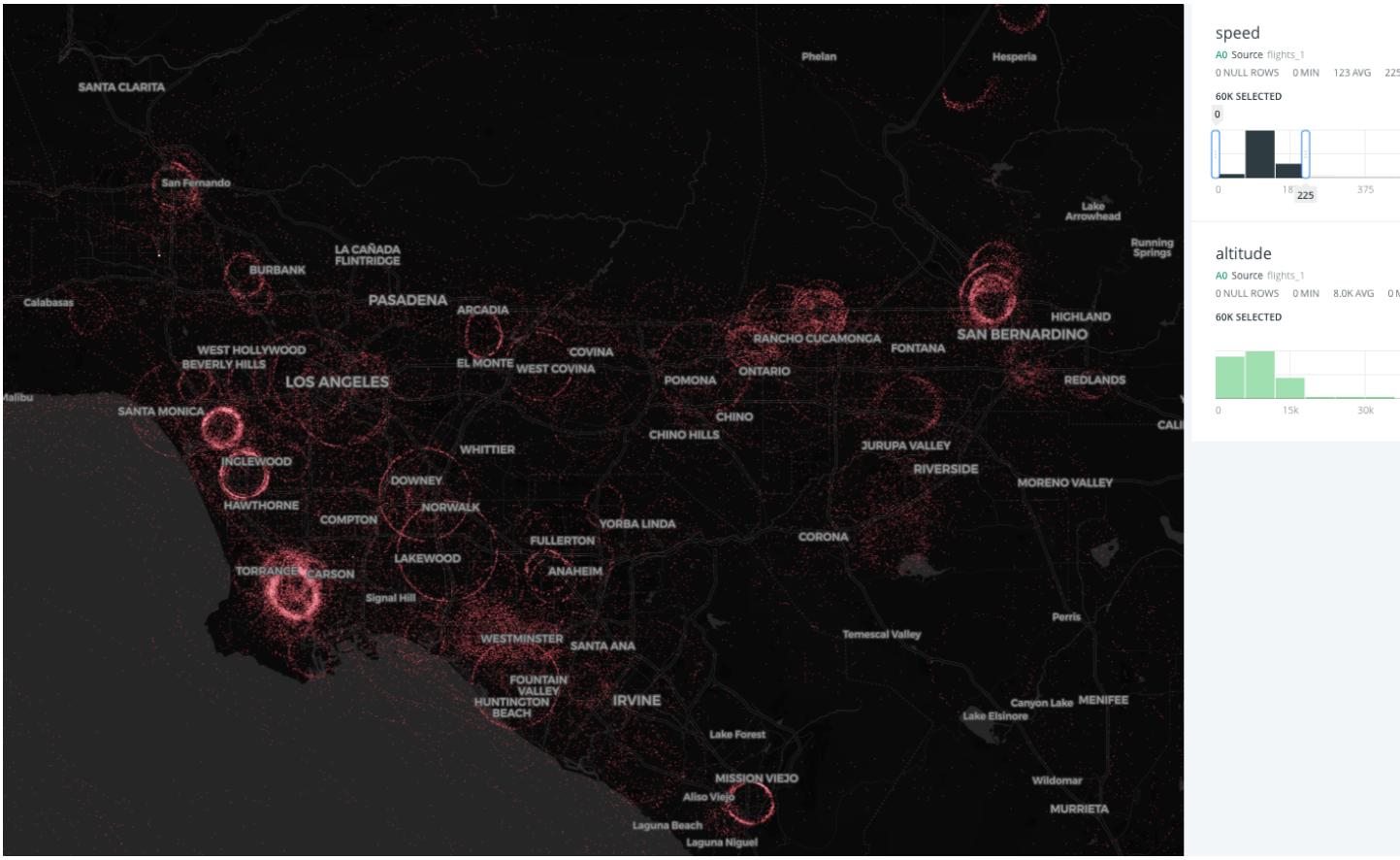
Podemos hacer un análisis preliminar de los datos, incluyendo un par de widgets al mapa, para filtrar por Altitud y Velocidad. Si la hipótesis de los vuelos de vigilancia es cierta, los patrones circulares deberán corresponder a posiciones de aviones que han volado a baja altura y baja velocidad, en comparación con un vuelo regular.

Hacemos clic en la pestaña *DATA* y hacemos clic en *Add as a widget* sobre las propiedades *altitude* y *speed* del dataset.

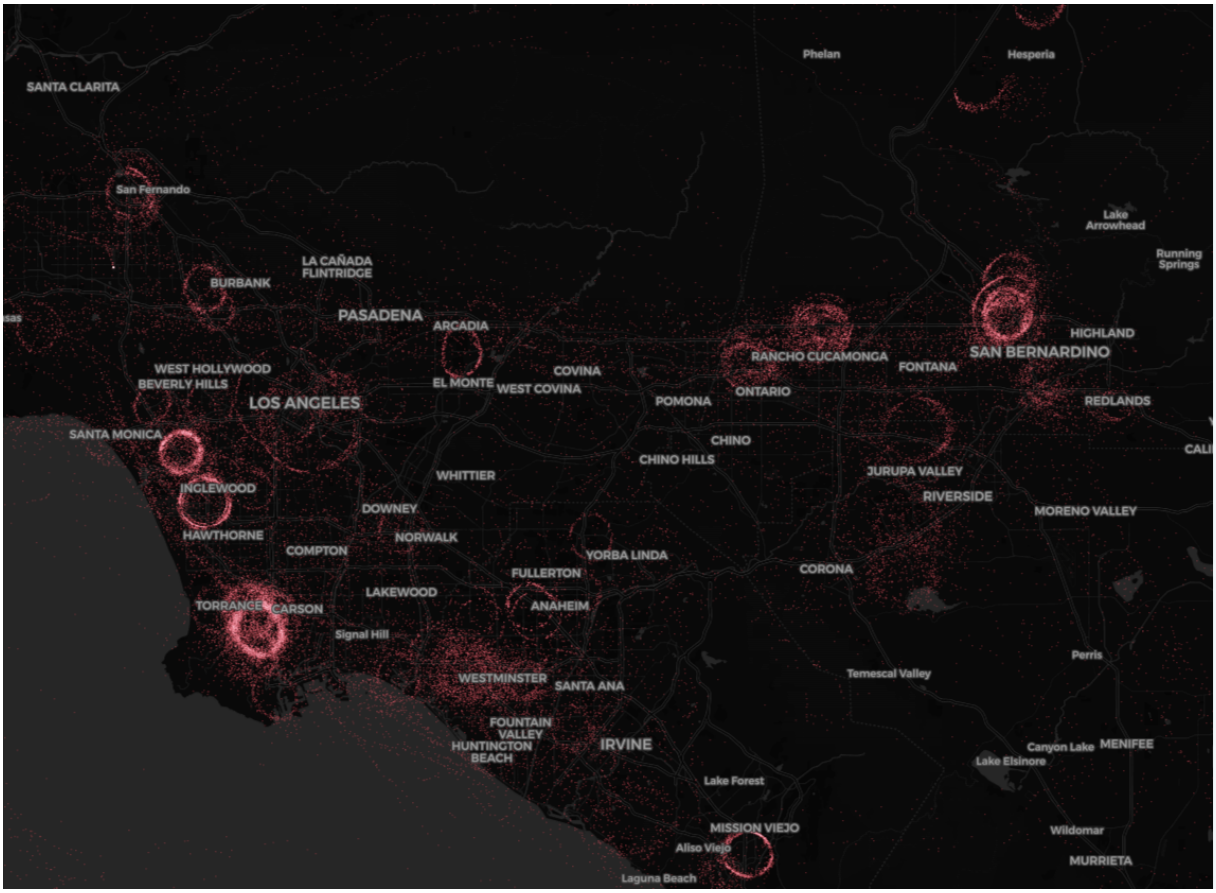


A continución, podemos aplicar un filtro sobre los widgets, para comprobar de manera visual que los círculos tienen bajos valores para ambas propiedades.

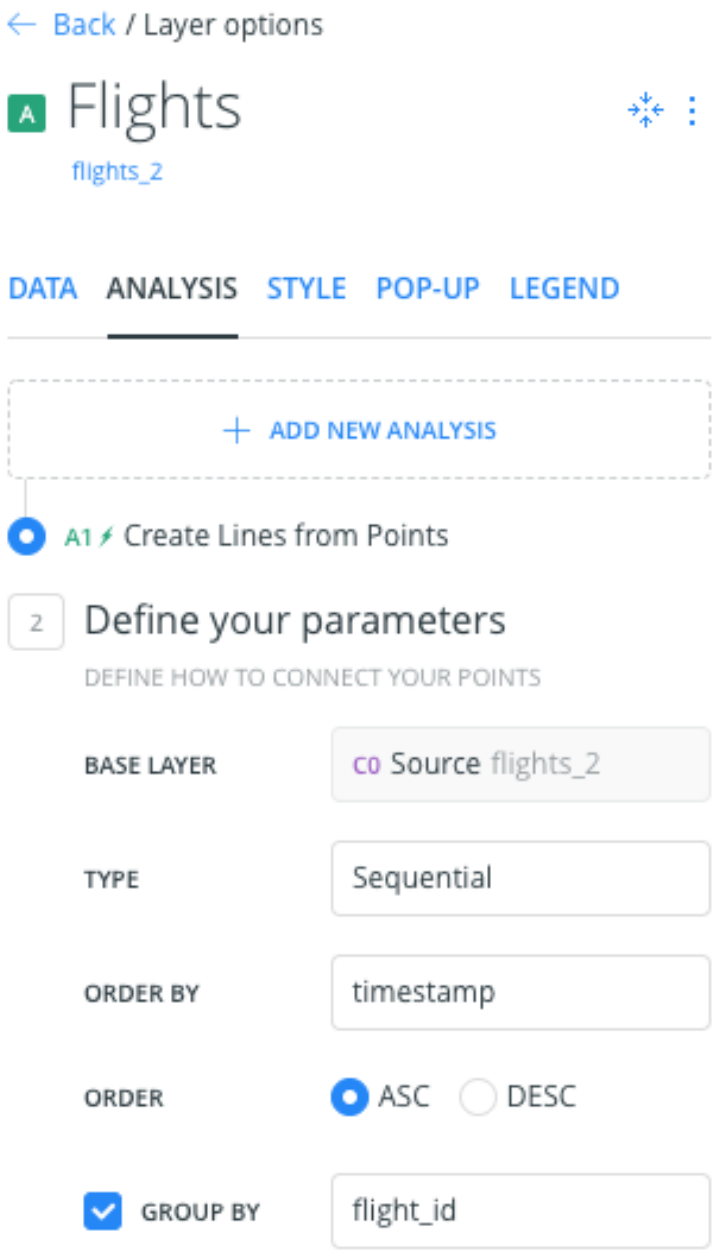
Así, seleccionando velocidades menores de 200 millas por hora, observamos que prácticamente sólo estamos filtrando los círculos.



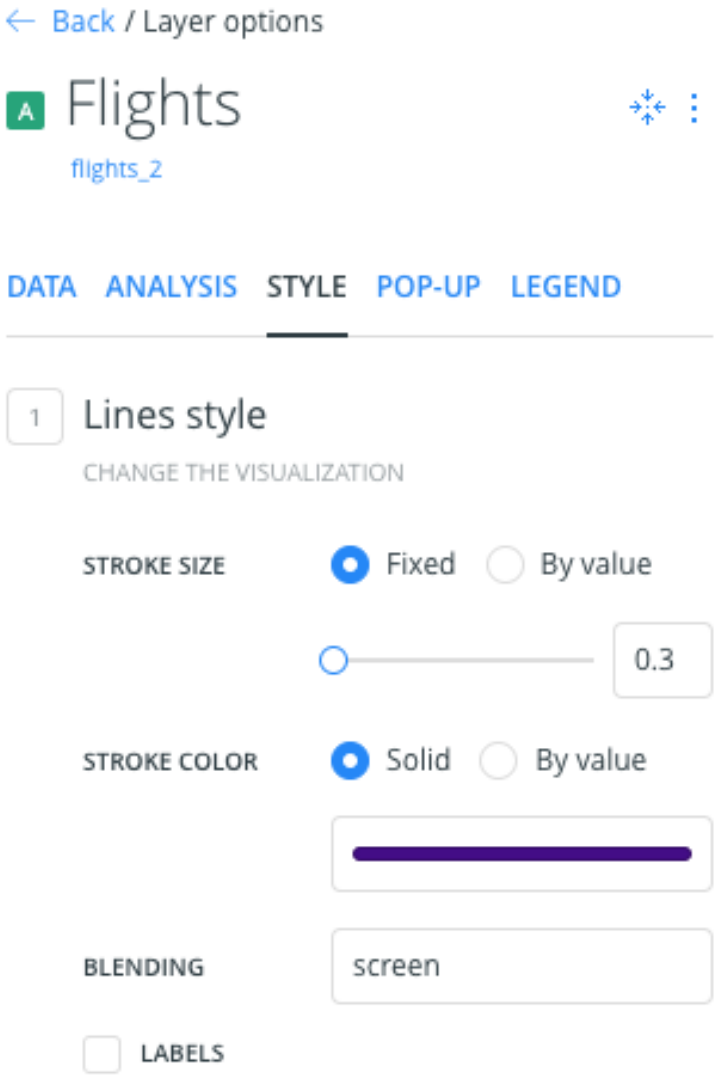
Y lo mismo ocurre para valores de altitud por debajo de los 10.000 pies.



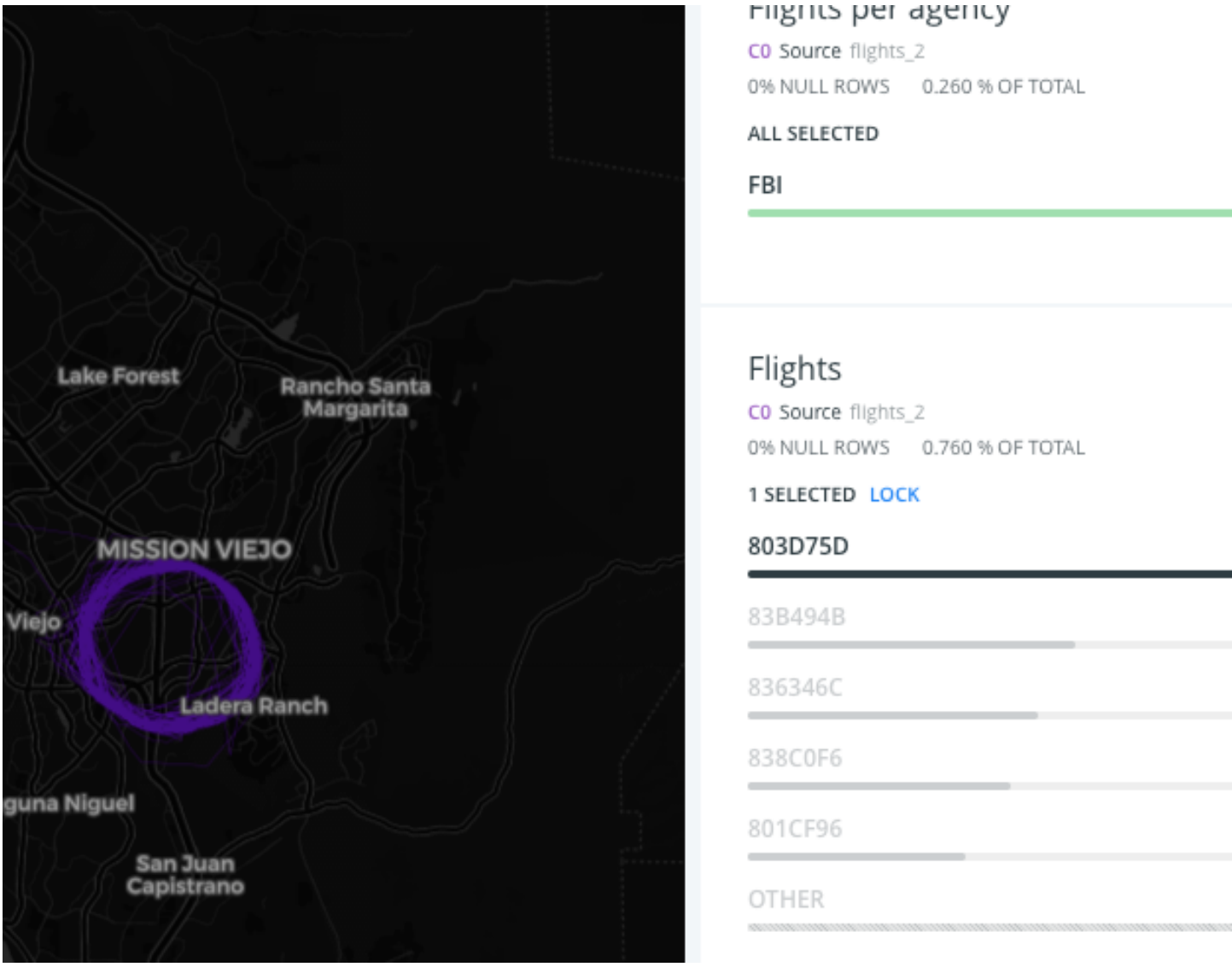
Por último, podemos convertir las secuencias de puntos de cada vuelo en líneas, para obtener una visualización más clara de lo que ha estado ocurriendo.



Hacemos clic en *ANALYSIS* > *Connect with lines* e introducimos los siguientes valores. A continuación, aplicamos las siguientes propiedades de estilo.



Y filtrando por un vuelo en concreto, comprobamos que efectivamente, hay vuelos de vigilancia sobre determinadas áreas.

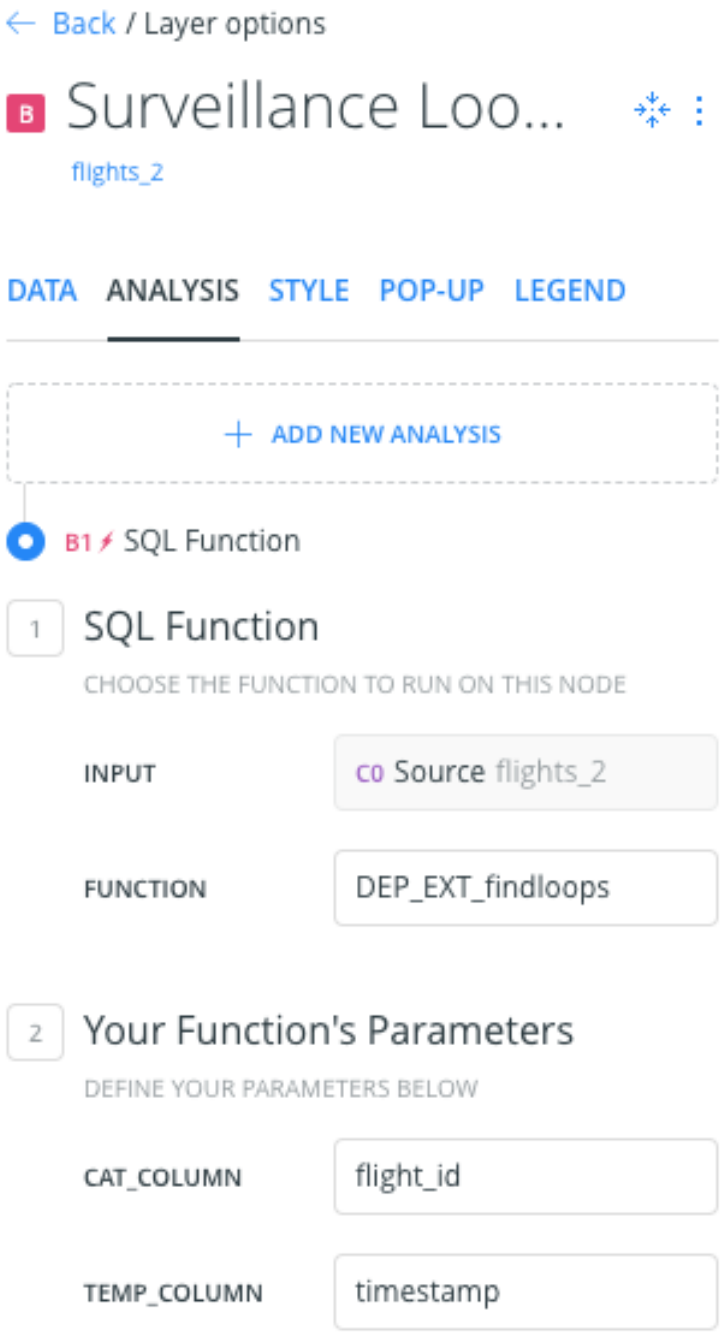


5.3 Análisis geoespacial

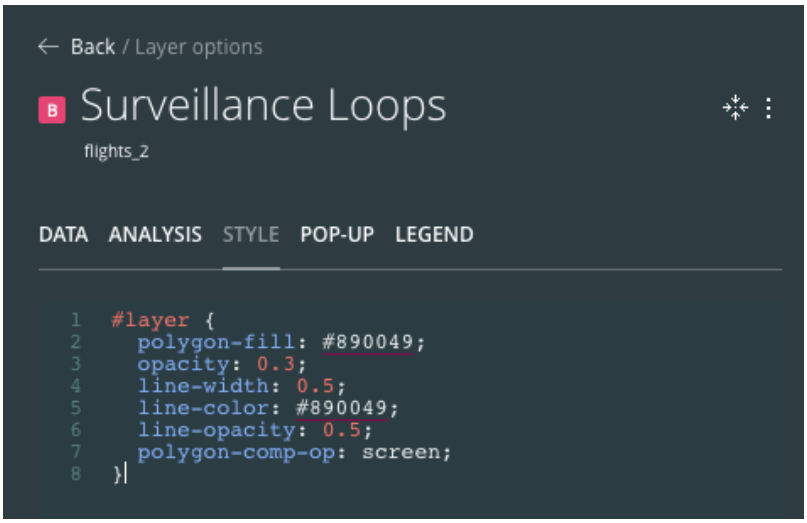
Sigamos con un análisis más en profundidad de los datos geoespaciales. Nuestro objetivo es definir cuáles son las áreas sujetas a vigilancia y obtener un perfil demográfico de las mismas.

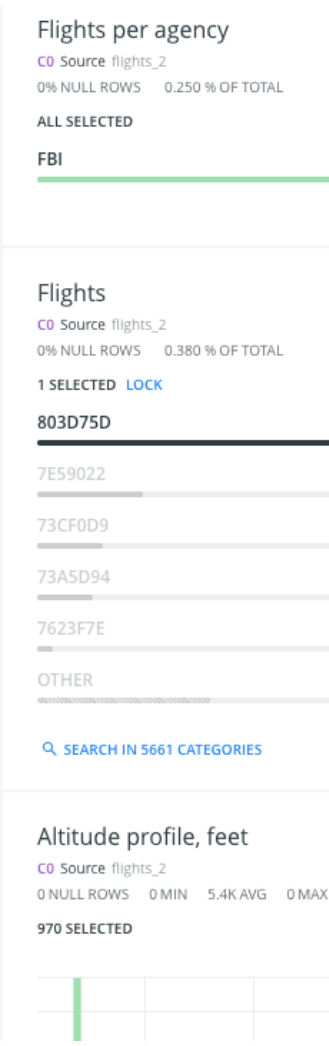
Para ello, nos vamos a centrar en el vuelo *803D75D* que hemos filtrado en el apartado anterior.

En primer lugar, definimos un análisis geoespacial que nos permite *I. Análisis para encontrar bucles en secuencias de puntos* y lo aplicamos sobre nuestro dataset *flights_1* con los siguientes parámetros:

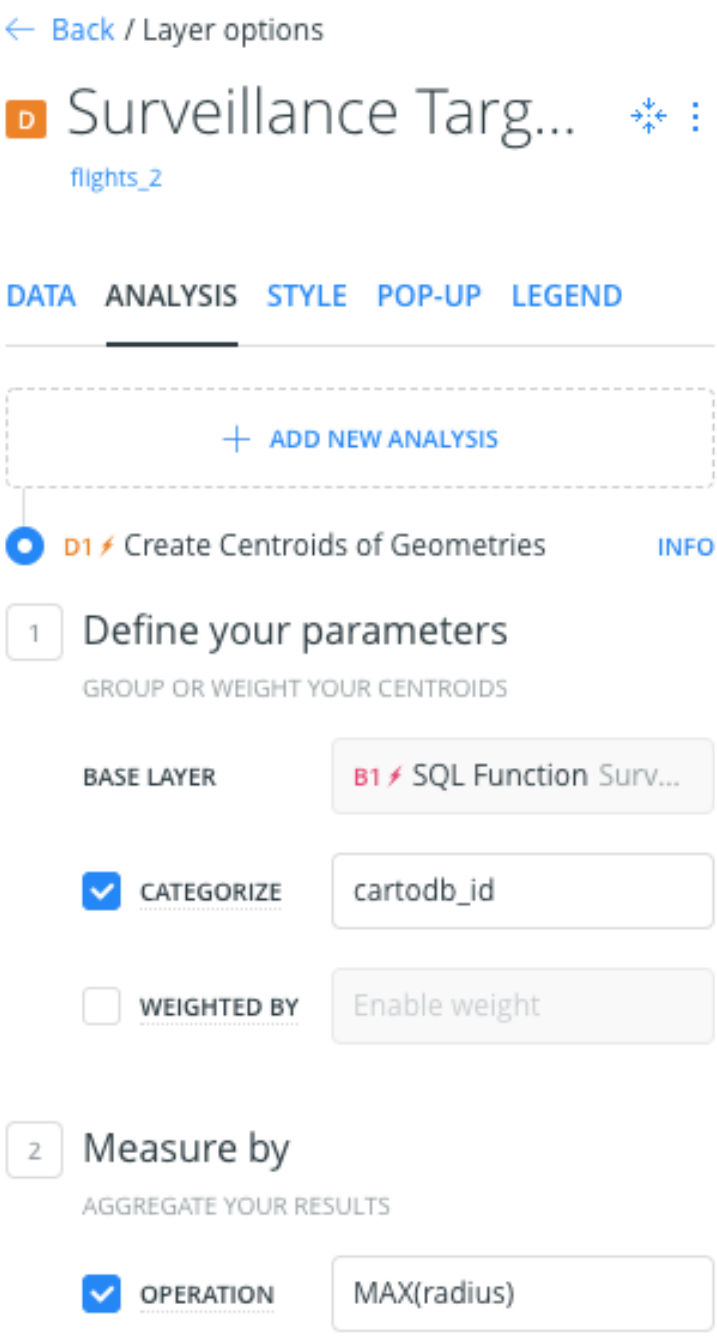


Aplicando este estilo a la capa de polígonos resultante obtenemos esta visualización, en la que se ve la área de influencia del vuelo de vigilancia.

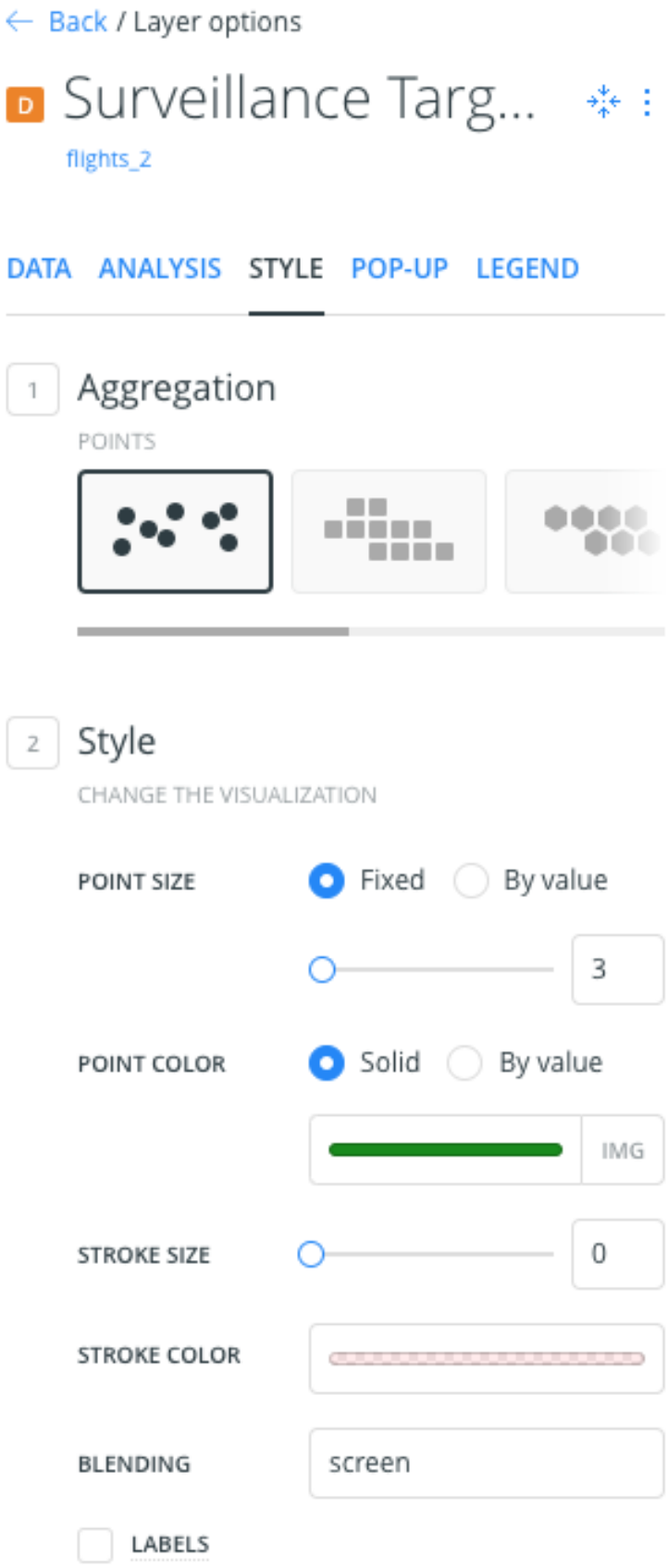




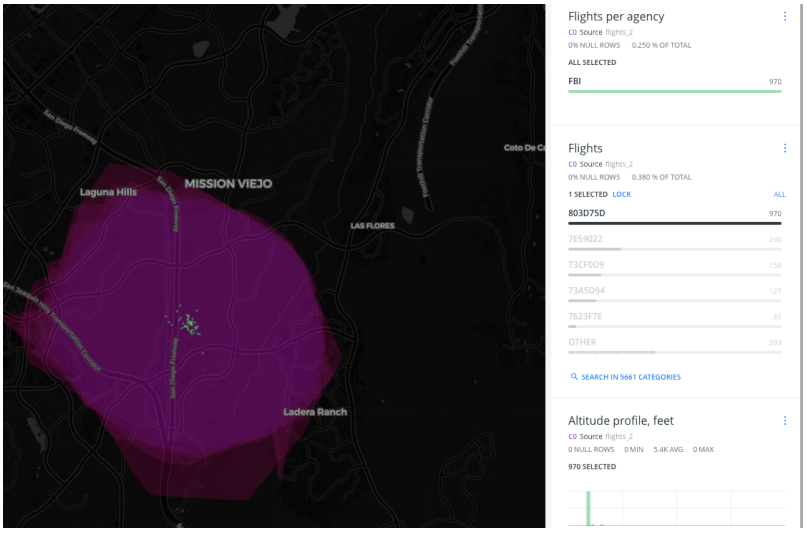
Si añadimos una capa adicional, que nos calcule los centroides de los polígonos obtenidos en el paso anterior:

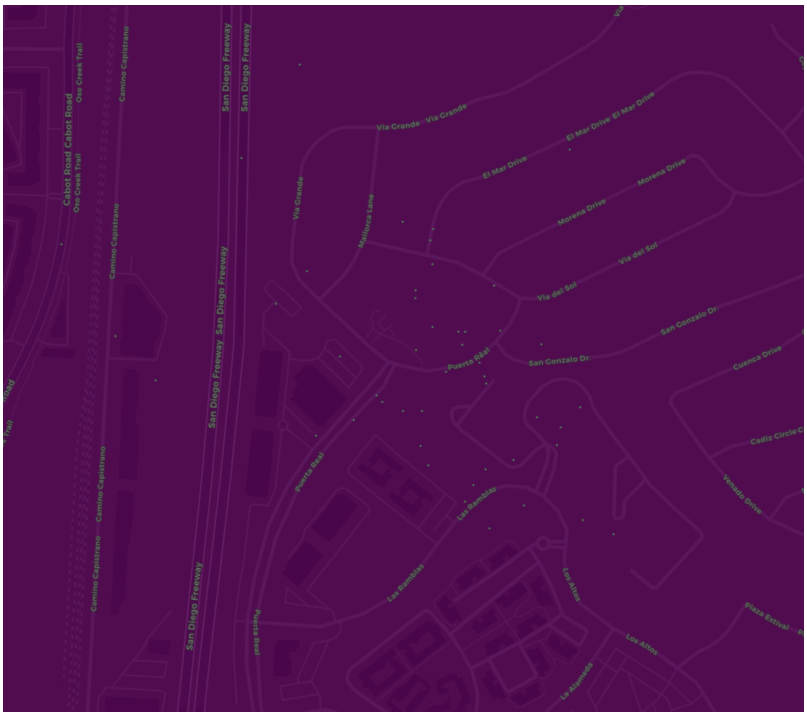


Aplicando el siguiente estilo:


Back / Layer options
Surveillance Targ...
flights_2
DATA
ANALYSIS
STYLE
POP-UP
LEGEND
1
Aggregation
POINTS
2
Style
CHANGE THE VISUALIZATION
POINT SIZE
Fixed
By value
3
POINT COLOR
Solid
By value
IMG
STROKE SIZE
0
STROKE COLOR
BLENDING
screen
LABELS

Podemos hacernos una idea de cuál o cuáles eran los objetivos de vigilancia.



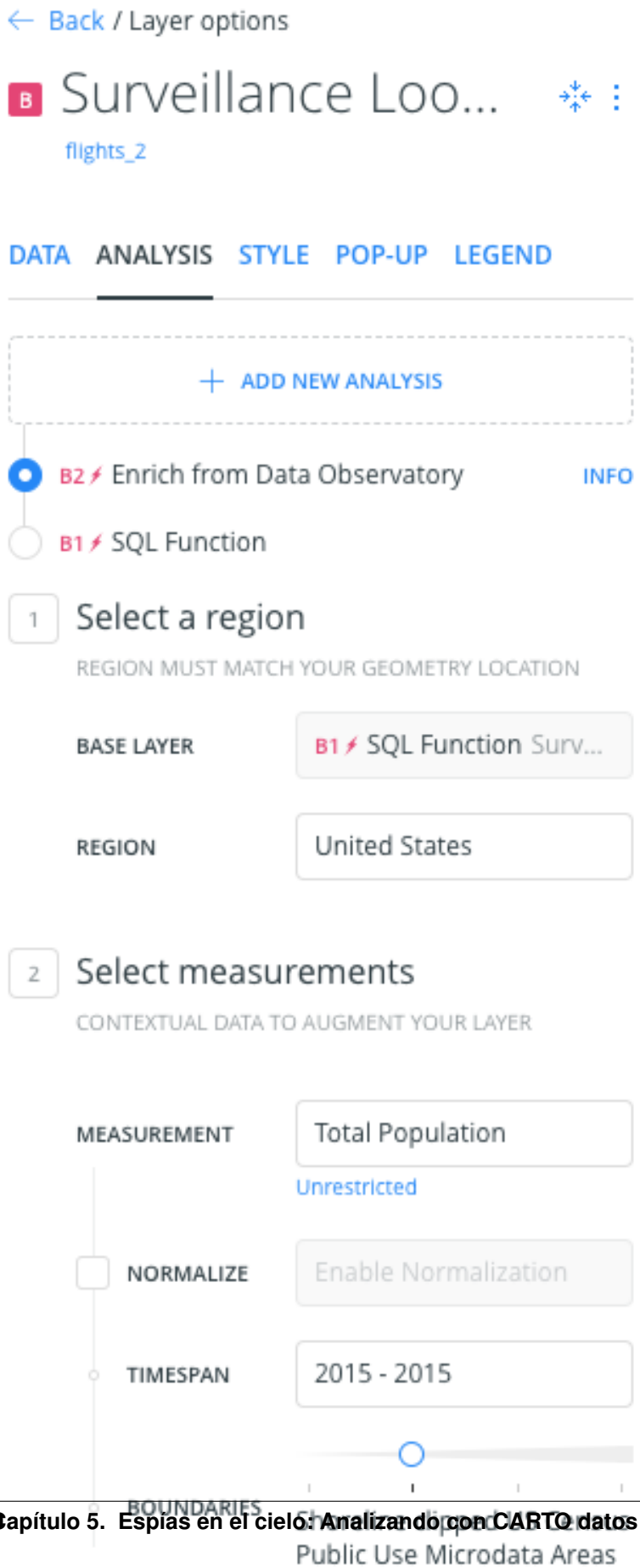


Cabe destacar que estos análisis se aplican de forma dinámica al dataset *flights_1* por tanto cada vez que se actualicen los datos diarios, a través de la sincronización que configuramos en el conector de BigQuery, los análisis se actualizarán también, proporcionando una vista actualizada de las áreas de influencia de los vuelos de vigilancia.

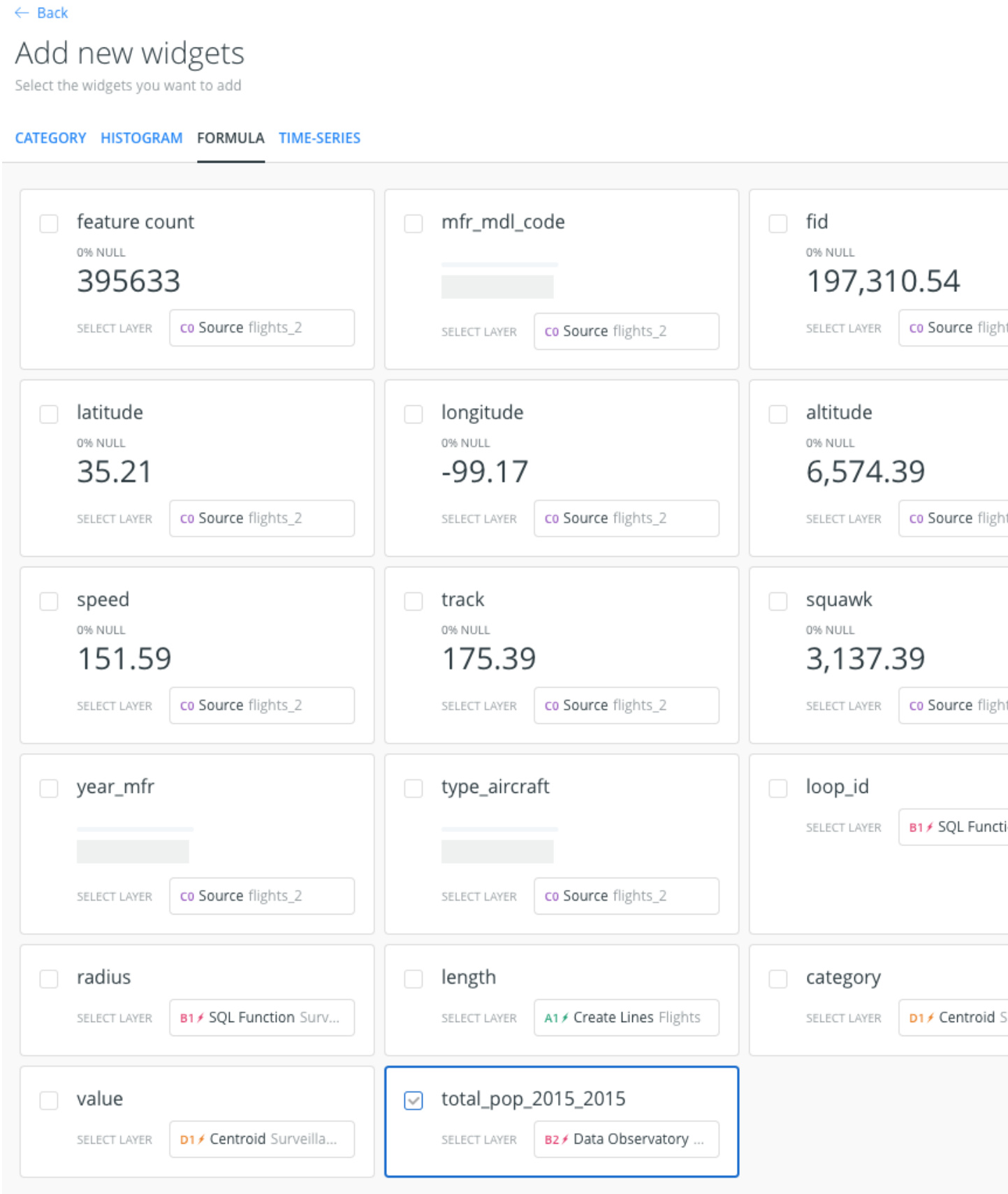
5.4 Enriquecimiento de datos

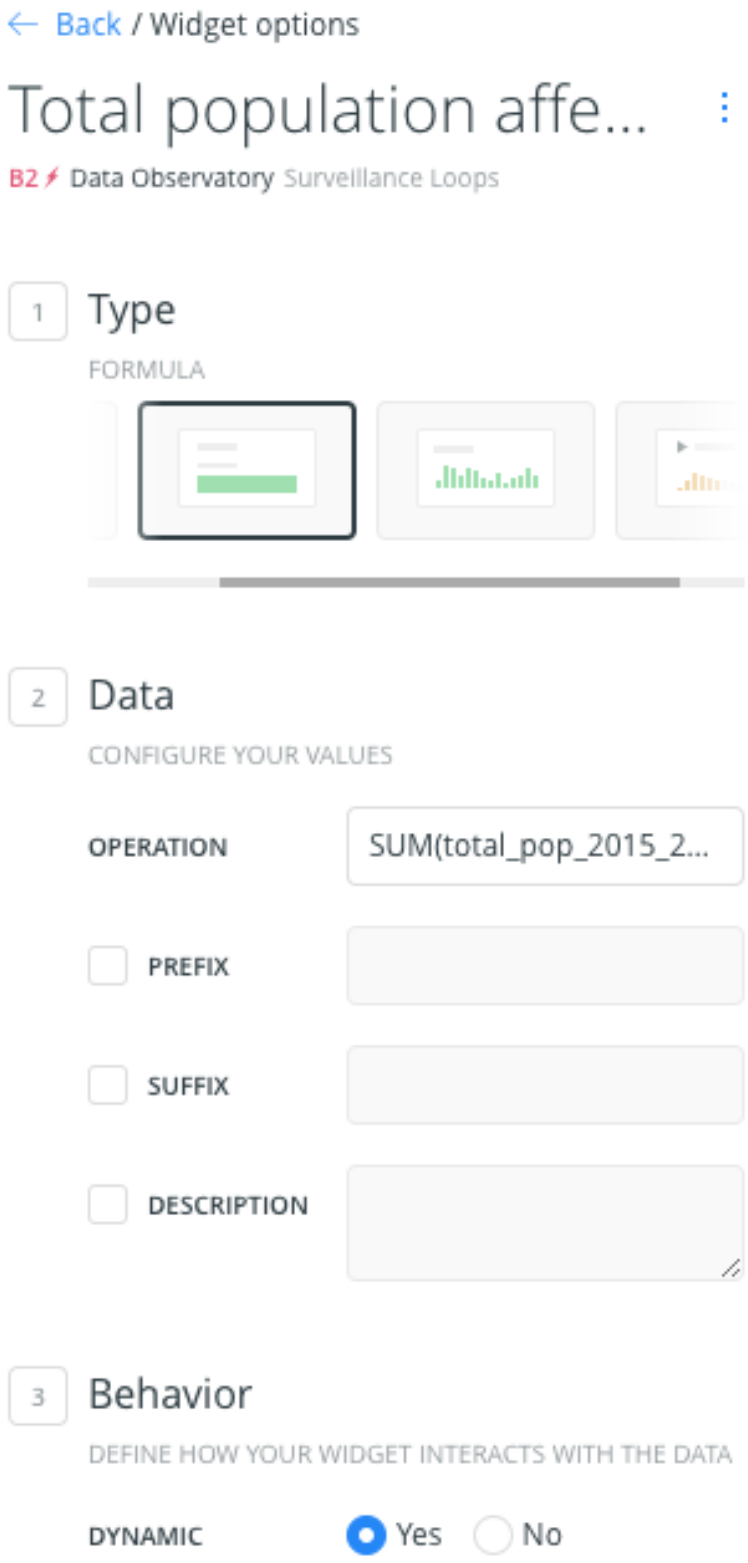
Como último paso previo a la publicación de los resultados, vamos a obtener el perfil demográfico de las áreas de influencia. En este caso, a modo de ejemplo, vamos a obtener la población total afectada, aunque podríamos aumentar los datos utilizando información de mayor interés como, el perfil económico de los habitantes de la zona, sus hábitos de movilidad o puntos de interés turístico, gubernamentales o militares en las zonas de influencia.

Aplicamos un análisis de *Enrich from Data Observatory* con los siguientes datos.

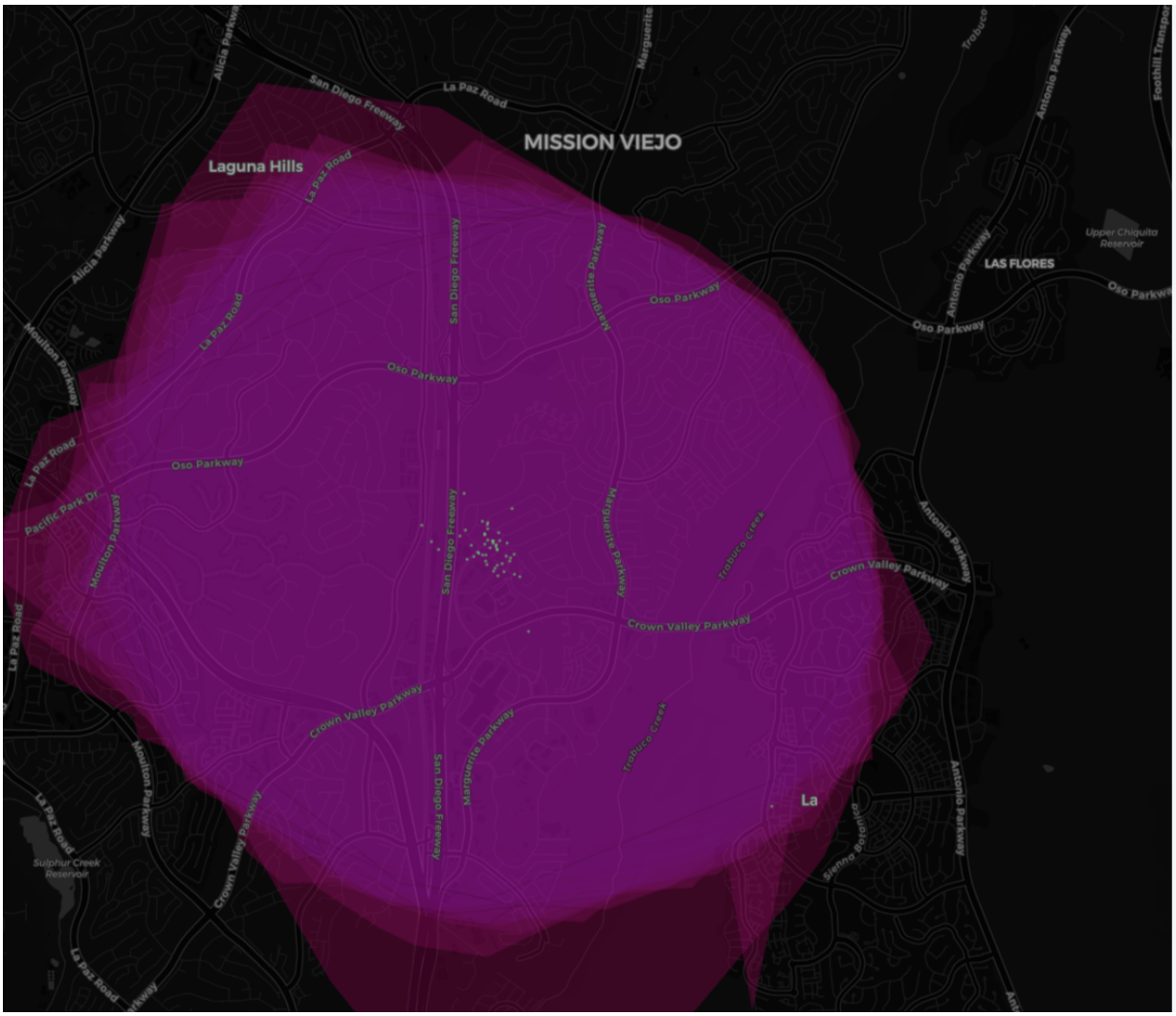

Back / Layer options
B
Surveillance Loo...
flights_2
DATA
ANALYSIS
STYLE
POP-UP
LEGEND
ADD NEW ANALYSIS
B2 Enrich from Data Observatory
INFO
B1 SQL Function
1
Select a region
REGION MUST MATCH YOUR GEOMETRY LOCATION
BASE LAYER
B1 SQL Function Surv...
REGION
United States
2
Select measurements
CONTEXTUAL DATA TO AUGMENT YOUR LAYER
MEASUREMENT
Total Population
Unrestricted
NORMALIZE
Enable Normalization
TIMESPAN
2015 - 2015
BOUNDARIES
Public Use Microdata Areas

Y añadimos un widget de tipo *Formula* sobre la nueva columna obtenida, para mostrar el número total de personas afectadas por los vuelos de vigilancia.


Back
Add new widgets
Select the widgets you want to add
CATEGORY
HISTOGRAM
FORMULA
TIME-SERIES
feature count
0% NULL
395633
SELECT LAYER
C0 Source flights_2
mfr_mdl_code
SELECT LAYER
C0 Source flights_2
fid
0% NULL
197,310.54
SELECT LAYER
C0 Source flight
latitude
0% NULL
35.21
SELECT LAYER
C0 Source flights_2
longitude
0% NULL
-99.17
SELECT LAYER
C0 Source flights_2
altitude
0% NULL
6,574.39
SELECT LAYER
C0 Source flight
speed
0% NULL
151.59
SELECT LAYER
C0 Source flights_2
track
0% NULL
175.39
SELECT LAYER
C0 Source flights_2
squawk
0% NULL
3,137.39
SELECT LAYER
C0 Source flight
year_mfr
SELECT LAYER
C0 Source flights_2
type_aircraft
SELECT LAYER
C0 Source flights_2
loop_id
SELECT LAYER
B1 SQL Functi
radius
SELECT LAYER
B1 SQL Function Surv...
length
SELECT LAYER
A1 Create Lines Flights
category
SELECT LAYER
D1 Centroid S
value
SELECT LAYER
D1 Centroid Surveilla...
total_pop_2015_2015
SELECT LAYER
B2 Data Observatory ...



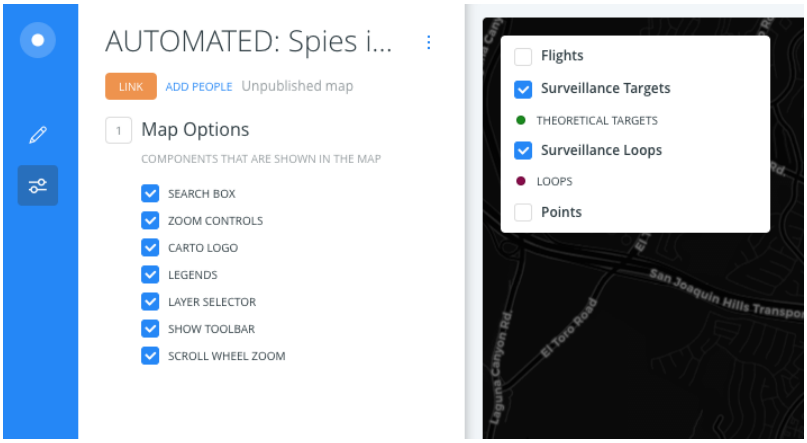
El número medio de personas afectadas por este vuelo en concreto es de 30.000 personas



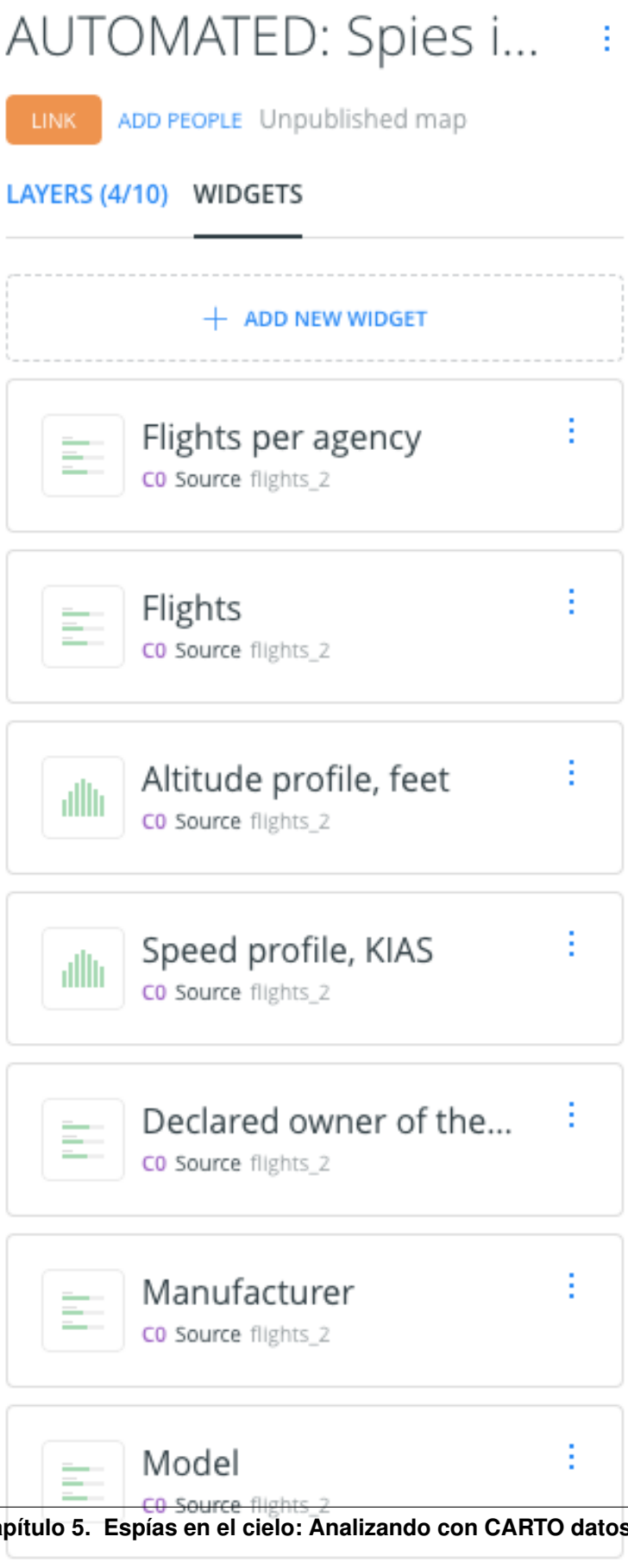
5.5 Publicación e integración de los resultados en el periódico

Es el momento de publicar los resultados obtenidos.

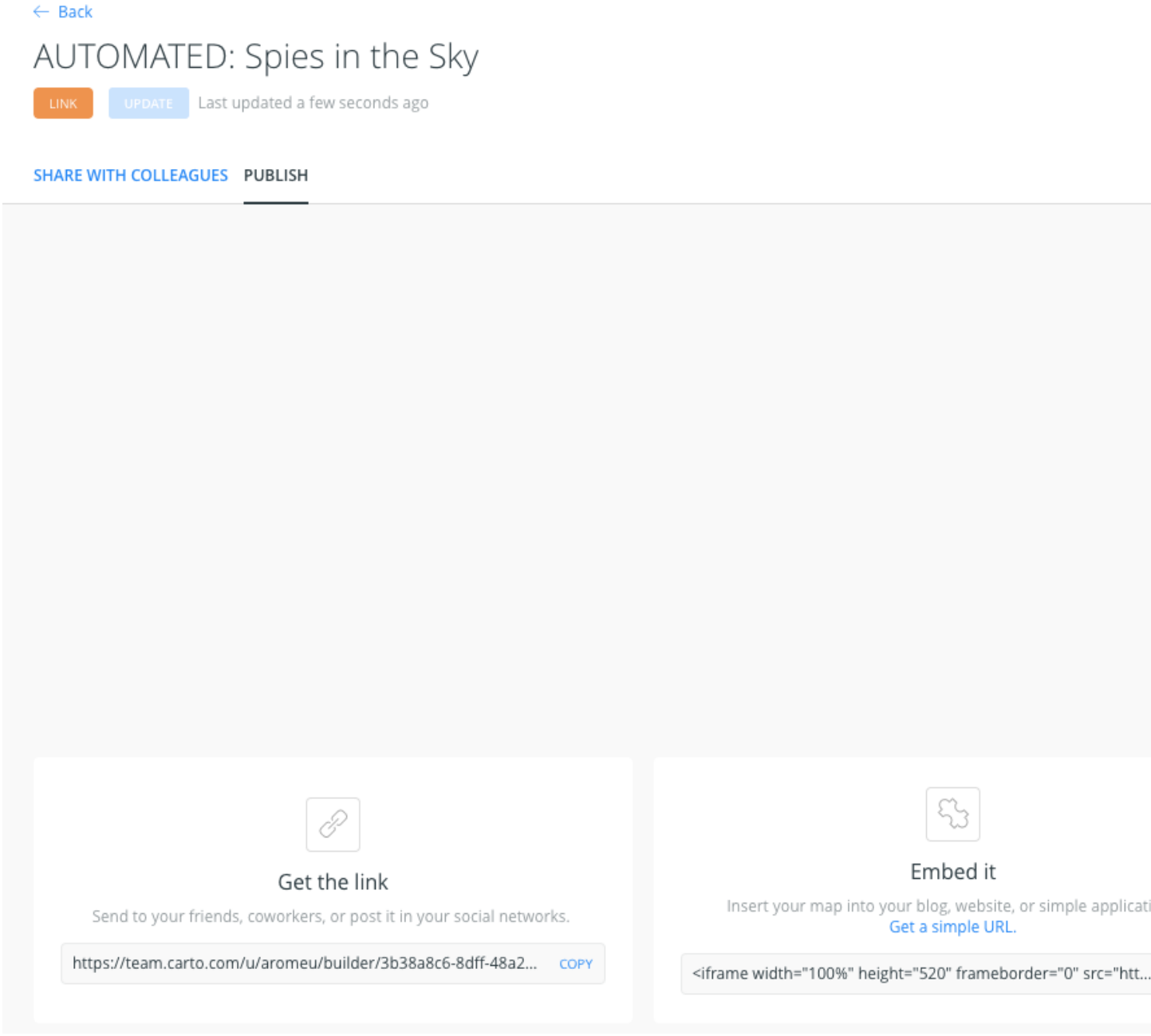
En primer lugar, activamos la leyenda sobre el mapa



A continuación, añadimos la posibilidad de filtrar sobre otros atributos del dataset original.

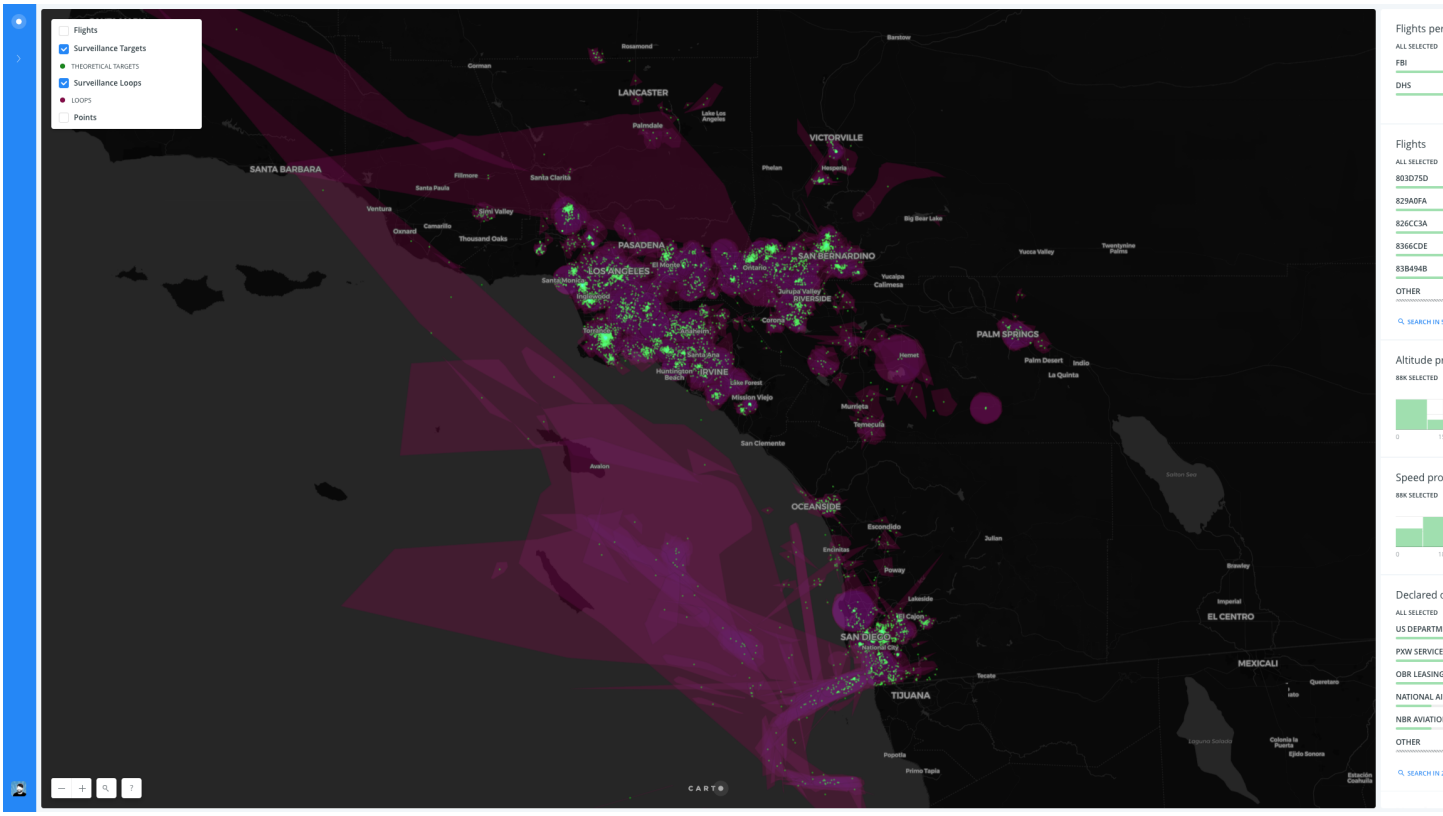

AUTOMATED: Spies i...
LINK
ADD PEOPLE
Unpublished map
LAYERS (4/10)
WIDGETS
ADD NEW WIDGET
Flights per agency
C0 Source flights_2
Flights
C0 Source flights_2
Altitude profile, feet
C0 Source flights_2
Speed profile, KIAS
C0 Source flights_2
Declared owner of the...
C0 Source flights_2
Manufacturer
C0 Source flights_2
Model
C0 Source flights_2

Por último, publicamos el mapa, obteniendo un enlace para publicarlo en el periódico digital.



El resultado final se puede consultar desde el siguiente enlace[2]

[2] https://team.carto.com/u/aromeu/builder/3b38a8c6-8dff-48a2-8360-6a8ac32f43d6/embed - mayo 2019


Flights
Surveillance Targets
THEORETICAL TARGETS
Surveillance Loops
LOOPS
Points
LANCASTER
VICTORVILLE
SANTA BARBARA
PASADENA
LOS ANGELES
SAN BERNARDINO
RIVERSIDE
PALM SPRINGS
IRVINE
OCEANSIDE
SAN DIEGO
TIJUANA
EL CENTRO
MEXICALI
CARTO
Flights per
ALL SELECTED
FBI
DHS
Flights
ALL SELECTED
803D75D
829A0FA
826CC3A
8366CDE
83B494B
OTHER
SEARCH IN
Altitude pr
88K SELECTED
Speed pro
88K SELECTED
Declared
ALL SELECTED
US DEPARTM
PXW SERVICE
OBR LEASING
NATIONAL AI
NBR AVIATIO
OTHER
SEARCH IN

CAPÍTULO 6

Conclusiones

En este trabajo final de máster nos habíamos propuesto solucionar el problema de conectar *CARTO* con los actuales sistemas de almacenamiento Big Data disponibles en el mercado.

La premisa era clara: *Realizar una conexión desde CARTO a estos sistemas, para importar datos y obtener lo mejor de dos mundos, el almacenamiento y procesamiento de sistemas preparados para trabajar con datos masivos y las capacidades de análisis y visualización geoespacial de CARTO*

Una vez analizado el estado del arte en cuanto a *CARTO* y los principales sistemas de almacenamiento y procesamiento Big Data (Hive, Impala, Redshift, MongoDB, Google Bigquery), encontramos un nexo de unión entre ambos: las capacidades de conectividad de PostgreSQL con sistemas de terceros a través de Foreign Data Wrappers.

Dado este nexo de unión, se pretende obtener una metodología sistemática que permita realizar conexiones desde PostgreSQL a sistemas de almacenamiento masivo y a través de la experimentación se exponen casos concretos de esta metodología para los principales sistemas de almacenamiento y procesamiento.

Una vez desarrollados los diferentes conectores, a través de un caso de uso consistente en analizar datos de posicionamiento de vuelos para detectar patrones de vigilancia, se demuestran las nuevas capacidades de conexión de *CARTO* con sistemas Big Data.

Por último, cabe descatar que durante este trabajo final de máster, se ha pretendido seguir la filosofía del plan de estudios del Máster Big Data Analytics de la Universidad Politécnica de Valencia, consistente en mostrar lo amplio del ecosistema Big Data y de las capacidades y aptitudes que deben poseer los profesionales de este sector.

En concreto, este trabajo está relacionado con los siguientes contenidos tratados durante el máster:

- BASH
- Virtualización: Docker y Vagrant
- Business/Location Intelligence
- Entornos de gestión Big Data: AWS, Hadoop, BigQuery, etc.
- NoSQL: MongoDB
- Técnicas y herramientas de visualización

6.1 Agradecimientos

ern goi da js ab

CAPÍTULO 7

Bibliografía

Ordenada alfabéticamente por autor

- 4rsoluciones, ¿Qué es un kit de desarrollo de software (SDK)? http://www.4rsoluciones.com/blog/que-es-un-kit-de-desarrollo-de-software-sdk-2/ - último acceso mayo 2019
- AWS, ¿Qué es NoSQL? https://aws.amazon.com/es/nosql/ - último acceso mayo 2019
- BBVA API Market, API REST: qué es y cuáles son sus ventajas en el desarrollo de proyectos. https://bbvaopen4u.com/es/actualidad/api-rest-que-es-y-cuales-son-sus-ventajas-en-el-desarrollo-de-proyectos - último acceso mayo 2019
- CARTO, ¿Qué es Location Intelligence? https://carto.com/location-intelligence/ - último acceso mayo 2019
- CampusMVP, ¿Qué es el stack MEAN? https://www.campusmvp.es/recursos/post/Que-es-el-stack-MEAN-y-como-escoger-el-mejor-para-ti.aspx - último acceso mayo 2019
- Cisco, ¿Qué es un firewall? https://www.cisco.com/c/es_es/products/security/firewalls/what-is-a-firewall.html - último acceso mayo 2019
- Francisco Javier Ruiz, 17 junio 2016. https://blog.dataprius.com/index.php/2016/06/17/on-premise-servidores-problemas-solucion-cloud/ - último acceso mayo 2019
- Hortonworks, ¿Qué es HDFS? https://es.hortonworks.com/apache/hdfs/ - último acceso mayo 2019
- Internetya, ¿Qué es y para qué sirve una API? https://www.internetya.co/que-es-y-para-que-sirve-una-api/ - último acceso octubre de 2019
- Margaret Rouse, 7 diciembre 2012, Business intelligence dashboard. http://searchbusinessanalytics.techtarget.com/definition/business-intelligence-dashboard - último acceso mayo 2019
- Microsoft Azure, ¿Qué es un Saas? https://azure.microsoft.com/es-es/overview/what-is-saas/ - último acceso mayo 2019
- Serprogramador, ¿Qué es frontend y backend? https://serprogramador.es/que-es-frontend-y-backend-en-la-programacion-web/ - último acceso mayo 2019
- Sinnexus, ¿Qué es Business Intelligence? http://www.sinnexus.com/business_intelligence/ - último acceso mayo 2019

- SolidQ, ¿Qué es MapReduce? http://blogs.solidq.com/es/big-data/que-es-mapreduce - último acceso mayo 2019
- SumaCRM, ¿Qué es CRM? https://www.sumacrm.com/soporte/customer-relationship-management - último acceso mayo 2019
- TIC.portal, Definición de un sistema ERP. https://www.ticportal.es/temas/enterprise-resource-planning/que-es-sistema-erp - último acceso mayo 2019
- Unpocodejava, ¿Qué es una arquitectura serverless? https://unpocodejava.com/2016/06/22/que-es-una-arquitectura-serverless-y-aws-lambda/ - último acceso mayo 2019

8. Bibliografía: referencias bibliográficas consultadas dispuestas por orden alfabético y citadas de acuerdo con los usos académicos (ver anexo 6.4).

See this guide to learn how to ingest data into Hive

–

Have at hand the documentation

Impala docs -> https://www.cloudera.com/documentation/other/connectors/impala-odbc/latest/Cloudera-ODBC-Driver-for-Impala-Install-Guide.pdf

CAPÍTULO 8

Anexos

9. Anexos: toda aquella información que se considere relevante para la comprensión y clarificación del trabajo desarrollado.

8.1 A. Código fuente de conector para Hive

```
# encoding: utf-8

require_relative './odbc'

module Carto
  class Connector

    # Hive (HiveServer2) provider using Hortonworks driver
    # (http://public-repo-1.hortonworks.com/HDP/hive-odbc/2.1.2.1002/debian/hive-odbc-
↪native_2.1.2.1002-2_amd64.deb)
    #
    # For all supported attributes, see https://hortonworks.com/wp-content/uploads/
↪2015/10/Hortonworks-Hive-ODBC-Driver-User-Guide.pdf
    #
    # Another driver compatible with this one is [Cloudera's](http://www.cloudera.com/
↪downloads/connectors/hive/odbc/2-5-12.html)
    #
    # The schema acts as a database name here, and can be ommitted (default schema
↪'default' or '')
    # So we'll use a `database` connection parameter for the schema for consistency␣
↪with other providers.
    # The schema parameter should not be directly used by the user.
    class HiveProvider < OdbcProvider

      private

      DEFAULT_SCHEMA = 'default'.freeze # '' would also be OK
```

(continué en la próxima página)

(proviene de la página anterior)

```
      def fixed_connection_attributes
        {
          Driver: 'Hortonworks Hive ODBC Driver 64-bit'
        }
      end

      def required_connection_attributes
        {
          server:   :HOST
        }
      end

      def optional_connection_attributes
        {
          database: { Schema: DEFAULT_SCHEMA },
          port: { PORT: 10000 },
          username: { UID: nil },
          password: { PWD: nil }
        }
      end

      def non_connection_parameters
        super.reverse_merge(schema: @connection[:database] || DEFAULT_SCHEMA)
      end

    end
  end
end
```

```
# encoding: utf-8

require_relative './odbc'

module Carto
  class Connector

    class ImpalaProvider < OdbcProvider

      def initialize(context, params)
        super
        if @connection
          @dsn        = @connection[:dsn]
          @driver     = @connection[:driver]
        end
      end

      def errors(only: nil)
        errors = super
        if @connection.blank?
          errors << "Missing 'connection' parameters"
        elsif @dns.blank? && @driver.blank?
          errors << "Must define either 'dsn' or 'driver' in 'connection'"
        end
        errors
      end
```

(continué en la próxima página)

(proviene de la página anterior)

```
      private

      def connection_attributes
        @connection
      end
    end

  end
end
```

8.2 B. Configuración del conector para Apache Hive

La configuración consiste en añadir una nuevo objecto al objeto *PROVIDERS* presente en el archivo *cartodb/lib/carto/connector/providers.rb* del repositorio https://github.com/CartoDB/cartodb

```
# encoding: utf-8

require_relative 'providers/generic_odbc'
require_relative 'providers/mysql'
require_relative 'providers/postgresql'
require_relative 'providers/sqlserver'
require_relative 'providers/hive'
require_relative 'providers/pg_fdw'

module Carto
  class Connector

    # Here we map provider identifiers (as used in APIs, etc.) to the Provider class␣
↪and basic attributes.
    # `name` is the human-readable name
    # `public` means that the provider is publicly announced (so it is accessible␣
↪through UI, visible in lists of
    # providers, etc.) A provider may be available or not (see Connector.limits)␣
↪independently of its public status,
    # so that a public provider may not be available for all users, and non-public␣
↪providers may be available to
    # some users (e.g. 'odbc' provider for tests)
    PROVIDERS = {
      'odbc' => {
        name: 'ODBC',
        class: GenericOdbcProvider,
        public: false # Intended for internal development/tests
      },
      'postgres' => {
        name: 'PostgreSQL',
        class: PostgreSQLProvider,
        public: true
      },
      'mysql' => {
        name: 'MySQL',
        class: MySqlProvider,
        public: true
      },
```

(continué en la próxima página)

(proviene de la página anterior)

```
      'sqlserver' => {
        name: 'Microsoft SQL Server',
        class: SqlServerProvider,
        public: true
      },
      'hive' => {
        name: 'Hive',
        class: HiveProvider,
        public: true
      }
    }
    ...
```

8.3 C. Configuración del conector para Apache Impala

La configuración consiste en añadir una nuevo objecto al objeto *PROVIDERS* presente en el archivo *cartodb/lib/carto/connector/providers.rb* del repositorio https://github.com/CartoDB/cartodb

```
# encoding: utf-8

require_relative 'providers/generic_odbc'
require_relative 'providers/mysql'
require_relative 'providers/postgresql'
require_relative 'providers/sqlserver'
require_relative 'providers/hive'
require_relative 'providers/pg_fdw'

module Carto
  class Connector

    # Here we map provider identifiers (as used in APIs, etc.) to the Provider class␣
↪and basic attributes.
    # `name` is the human-readable name
    # `public` means that the provider is publicly announced (so it is accessible␣
↪through UI, visible in lists of
    # providers, etc.) A provider may be available or not (see Connector.limits)␣
↪independently of its public status,
    # so that a public provider may not be available for all users, and non-public␣
↪providers may be available to
    # some users (e.g. 'odbc' provider for tests)
    PROVIDERS = {
          'odbc' => {
            name: 'ODBC',
            class: GenericOdbcProvider,
            public: false # Intended for internal development/tests
          },
          'postgres' => {
            name: 'PostgreSQL',
            class: PostgreSQLProvider,
            public: true
          },
          'mysql' => {
            name: 'MySQL',
            class: MySqlProvider,
```

(continué en la próxima página)

(proviene de la página anterior)

```
            public: true
          },
          'sqlserver' => {
            name: 'Microsoft SQL Server',
            class: SqlServerProvider,
            public: true
          },
          'hive' => {
            name: 'Hive',
            class: HiveProvider,
            public: true
          },
          'impala' => {
            name: 'Impala',
            class: GenericOdbcProvider,
            public: true
          }
        }...
```

8.4 D. Configuración del conector para Amazon Redshift

La configuración consiste en añadir una nuevo objecto al objeto *PROVIDERS* presente en el archivo *cartodb/lib/carto/connector/providers.rb* del repositorio https://github.com/CartoDB/cartodb

```
# encoding: utf-8

require_relative 'providers/generic_odbc'
require_relative 'providers/mysql'
require_relative 'providers/postgresql'
require_relative 'providers/sqlserver'
require_relative 'providers/hive'
require_relative 'providers/pg_fdw'

module Carto
  class Connector

    # Here we map provider identifiers (as used in APIs, etc.) to the Provider class␣
↪and basic attributes.
    # `name` is the human-readable name
    # `public` means that the provider is publicly announced (so it is accessible␣
↪through UI, visible in lists of
    # providers, etc.) A provider may be available or not (see Connector.limits)␣
↪independently of its public status,
    # so that a public provider may not be available for all users, and non-public␣
↪providers may be available to
    # some users (e.g. 'odbc' provider for tests)
    PROVIDERS = {
          'odbc' => {
            name: 'ODBC',
            class: GenericOdbcProvider,
            public: false # Intended for internal development/tests
          },
          'postgres' => {
            name: 'PostgreSQL',
```

(continué en la próxima página)

(proviene de la página anterior)

```
            class: PostgreSQLProvider,
            public: true
          },
          'mysql' => {
            name: 'MySQL',
            class: MySqlProvider,
            public: true
          },
          'sqlserver' => {
            name: 'Microsoft SQL Server',
            class: SqlServerProvider,
            public: true
          },
          'hive' => {
            name: 'Hive',
            class: HiveProvider,
            public: true
          },
          'impala' => {
            name: 'Impala',
            class: GenericOdbcProvider,
            public: true
          },
          'redshift' => {
            name: 'Redshift',
            class: GenericOdbcProvider,
            public: true
          }
        }...
```

8.5 E. Código fuente de conector para MongoDB

```
# encoding: utf-8

require_relative './odbc'

module Carto
  class Connector

    # Class for ODBC-based provider using mongo_fdw
    #
    # Requirements:
    #   * mongo_fdw extension must be installed in the user database
    # "connector": {
    #    "provider": "mongo",
    #    "connection": {
    #      "username":"THE_MONGO_USER_NAME",
    #      "password":"THE_MONGO_PASSWORD",
    #      "server": "THE_MONGO_SERVER",
    #      "database": "THE_MONGO_DATABASE",
    #      "port":"THE_MONGO_PORT",
    #      "schema": "THE_MONGO_COLLECTION"
    #    },
    #    "table": "THE_MONGO_TABLE",
```

(continué en la próxima página)

(proviene de la página anterior)

```
    #    "columns": "THE_COLUMNS_OF_THE_MONGO_TABLE"
    #  }
    #   "columns" is a comma separated list of column-name type.
    #   As an example "columns" can be "_id NAME,   warehouse_id int,   warehouse_
↪name text,   warehouse_created timestamptz
    #
    class MongoProvider < OdbcProvider

      def initialize(context, params)
        super
      end

      # Required connection attributes: { name: :internal_name }
      # The :internal_name is what is passed to the driver (through odbc_fdw 'odbc_'␣
↪options)
      # The :name is the case-insensitive parameter received here trhough the API
      # This can be redefined as needed in derived classes.
      def required_connection_attributes
        {
          server:         :address
        }
      end

      # Connection attributes that are optional: { name: { internal_name: default_
↪value } }
      # Those with non-nil default values will always be set.
      # name/internal_name as in `required_connection_attributes`
      # This can be redefined as needed in derived classes.
      def optional_connection_attributes
        {
          database: { database: "admin" },
          port: { port: 27017 },
          username: { username: nil },
          password: { password: nil },
          schema:   { collection: nil }
        }
      end

      def fdw_create_server(server_name)
        sql = fdw_create_server_sql 'mongo_fdw', server_name, server_options_m
        execute_as_superuser sql
      end

      def fdw_list_tables(server_name, limit)
        # TODO not tested in MongoDB
        execute %{
          SELECT * FROM ODBCTablesList('#{server_name}',#{limit.to_i});
        }
      end

      def features_information
        {
          "sql_queries":    true,
          "list_databases": false,
          "list_tables":    false,
          "preview_table":  false
        }
```

(continué en la próxima página)

(proviene de la página anterior)

```
      end

      def fdw_create_foreign_table(server_name)
        cmds = []
        foreign_table_name = foreign_table_name_for(server_name)
        if @columns.present?
          cmds << fdw_create_foreign_table_sql(
            server_name, foreign_table_schema, foreign_table_name, @columns, table_
↪options_m
          )
        else
          options = table_options.merge(prefix: unique_prefix_for(server_name))
          cmds << fdw_import_foreign_schema_sql(server_name, remote_schema_name,␣
↪foreign_table_schema, options)
        end
        cmds << fdw_grant_select_sql(foreign_table_schema, foreign_table_name,␣
↪@connector_context.database_username)
        execute_as_superuser cmds.join("\n")
        foreign_table_name
      end

      SERVER_OPTIONS = %w(dsn driver host server address port).freeze
      USER_OPTIONS   = %w(uid pwd user username password).freeze
      TABLE_OPTIONS  = %w(database collection).freeze

      def connection_options(parameters)
        parameters.map { |option_name, option_value| ["#{option_name}", quoted_
↪value(option_value)] }
      end

      def server_options_m
        connection_options(connection_attributes.slice(*SERVER_OPTIONS)).parameters
      end

      def table_options_m
        connection_options(connection_attributes.slice(*TABLE_OPTIONS)).parameters
      end
    end
  end
end
```

8.6 F. Configuración del conector para MongoDB

La configuración consiste en añadir una nuevo objecto al objeto *PROVIDERS* presente en el archivo *cartodb/lib/carto/connector/providers.rb* del repositorio https://github.com/CartoDB/cartodb

```
# encoding: utf-8

require_relative 'providers/generic_odbc'
require_relative 'providers/mysql'
require_relative 'providers/postgresql'
require_relative 'providers/sqlserver'
require_relative 'providers/hive'
require_relative 'providers/pg_fdw'
```

(continué en la próxima página)

(proviene de la página anterior)

```
module Carto
  class Connector

    # Here we map provider identifiers (as used in APIs, etc.) to the Provider class
↪and basic attributes.
    # `name` is the human-readable name
    # `public` means that the provider is publicly announced (so it is accessible
↪through UI, visible in lists of
    # providers, etc.) A provider may be available or not (see Connector.limits)
↪independently of its public status,
    # so that a public provider may not be available for all users, and non-public
↪providers may be available to
    # some users (e.g. 'odbc' provider for tests)
    PROVIDERS = {
          'odbc' => {
            name: 'ODBC',
            class: GenericOdbcProvider,
            public: false # Intended for internal development/tests
          },
          'postgres' => {
            name: 'PostgreSQL',
            class: PostgreSQLProvider,
            public: true
          },
          'mysql' => {
            name: 'MySQL',
            class: MySqlProvider,
            public: true
          },
          'sqlserver' => {
            name: 'Microsoft SQL Server',
            class: SqlServerProvider,
            public: true
          },
          'hive' => {
            name: 'Hive',
            class: HiveProvider,
            public: true
          },
          'impala' => {
            name: 'Impala',
            class: GenericOdbcProvider,
            public: true
          },
          'redshift' => {
            name: 'Redshift',
            class: GenericOdbcProvider,
            public: true
          },
          'mongo' => {
        name: 'mongo',
        class: MongoProvider,
        public: true
      }
        }...
```

8.7 G. Código fuente de conector para BigQuery

```
# encoding: utf-8

require 'uri'
require_relative './odbc'

module Carto
  class Connector

    # {
    #   "provider": "bigquery",
    #   "connection": {
    #     "Driver": "Google BigQuery 64",
    #     "OAuthMechanism": 1,
    #     "Catalog": "eternal-ship-170218",
    #     "SQLDialect": 1,
    #     "RefreshToken": "1/FyCbmKonlYAwx7FMjfow9QO5mdiOG3u9dfpi0ktYxOux_fFDF6ip-
↪PERQkXYKiDc"
    #   },
    #   "table": "destination_table",
    #   "sql_query": "select * from `eternal-ship-170218.test.test` limit 1;"
    # }
    class BigQueryProvider < OdbcProvider

      private

      DRIVER_NAME       = 'Google BigQuery 64'
      SQL_DIALECT       = 1
      USER_AUTH         = 1
      SERVICE_AUTH      = 0

      def initialize(context, params)
        super
        @oauth_config = Cartodb.get_config(:oauth, 'bigquery')
        validate_config!(context)
      end

      def validate_config!(context)
        raise 'OAuth configuration not found for bigquery provider' if @oauth_config.
↪nil?
        if @oauth_config['oauth_mechanism'] === SERVICE_AUTH \
            and @oauth_config['email'].nil? \
            and @oauth_config['key'].nil?
          raise 'bigquery provider configured in SERVICE_AUTH mode but email or key␣
↪not present'
        else
          begin
          @token = context.user.oauths.select('bigquery').token
          raise 'OAuth Token not found for bigquery provider' if @token.nil?
          rescue => e
            CartoDB::Logger.error(exception: e,
                                  message: 'OAuth Token not found for "bigquery"␣
↪provider',
                                  user_id: context.user.id)
          end
        end
```

(continué en la próxima página)

(proviene de la página anterior)

```
      end

      def fixed_connection_attributes
        oauth_mechanism = @oauth_config['oauth_mechanism']
        proxy_conf = create_proxy_conf

        if oauth_mechanism === SERVICE_AUTH
          conf = {
            Driver:         DRIVER_NAME,
            SQLDialect:     SQL_DIALECT,
            OAuthMechanism: oauth_mechanism,
            Email:          @oauth_config['email'],
            KeyFilePath:    @oauth_config['key']
          }
        else
          conf = {
            Driver:         DRIVER_NAME,
            SQLDialect:     SQL_DIALECT,
            OAuthMechanism: oauth_mechanism,
            RefreshToken:   @token
          }
        end

        if !proxy_conf.nil?
          conf = conf.merge(proxy_conf)
        end

        return conf
      end

      def required_connection_attributes
        {
          database:       :Catalog
        }
      end

      def create_proxy_conf
        proxy = ENV['HTTP_PROXY'] || ENV['http_proxy']
        if !proxy.nil?
          proxy = URI.parse(proxy)
          {
            ProxyHost: proxy.host,
            ProxyPort: proxy.port
          }
        end
      end

    end
  end
end
```

8.8 H. Configuración del conector para BigQuery

La configuración consiste en añadir una nuevo objecto al objeto *PROVIDERS* presente en el archivo *cartodb/lib/carto/connector/providers.rb* del repositorio https://github.com/CartoDB/cartodb

```
# encoding: utf-8

require_relative 'providers/generic_odbc'
require_relative 'providers/mysql'
require_relative 'providers/postgresql'
require_relative 'providers/sqlserver'
require_relative 'providers/hive'
require_relative 'providers/pg_fdw'

module Carto
  class Connector

    # Here we map provider identifiers (as used in APIs, etc.) to the Provider class␣
↪and basic attributes.
    # `name` is the human-readable name
    # `public` means that the provider is publicly announced (so it is accessible␣
↪through UI, visible in lists of
    # providers, etc.) A provider may be available or not (see Connector.limits)␣
↪independently of its public status,
    # so that a public provider may not be available for all users, and non-public␣
↪providers may be available to
    # some users (e.g. 'odbc' provider for tests)
    PROVIDERS = {
          'odbc' => {
            name: 'ODBC',
            class: GenericOdbcProvider,
            public: false # Intended for internal development/tests
          },
          'postgres' => {
            name: 'PostgreSQL',
            class: PostgreSQLProvider,
            public: true
          },
          'mysql' => {
            name: 'MySQL',
            class: MySqlProvider,
            public: true
          },
          'sqlserver' => {
            name: 'Microsoft SQL Server',
            class: SqlServerProvider,
            public: true
          },
          'hive' => {
            name: 'Hive',
            class: HiveProvider,
            public: true
          },
          'impala' => {
            name: 'Impala',
            class: GenericOdbcProvider,
            public: true
          },
          'redshift' => {
            name: 'Redshift',
            class: GenericOdbcProvider,
            public: true
```

(continué en la próxima página)

(proviene de la página anterior)

```
        },
        'mongo' => {
      name: 'mongo',
      class: MongoProvider,
      public: true
    },
    'bigquery' => {
      name: 'Google BigQuery 64',
      class: BigQueryProvider,
      public: true
    }
      }...
```

8.9 I. Análisis para encontrar bucles en secuencias de puntos

```
/*
DEP_EXT_findloops
From a points dataset representing positions of a moving object along a track,
this function finds the loops in the track

Inputs managed by CARTO, common to all DEP_EXT functions:
    operation               text: 'create' or 'populate'
    table_name              text: the name of the previous node table
    primary_source_query    text: the query on the previous node table
    primary_source_columns  text: the columns of the previous node table
User input:
    cat_column                 text: if we have more than one track that is␣
↪identified by a column value
    temp_column       text: sorting column, usually timestamp
Output:
* cartodb_id bigint
* track_id text: the track identifier, equal to cat_column of the input
* loop_id integer: ordinal of the loop for each track
* the_geom geometry(Geometry,4326): the circle that represents the loop
* radius numeric: radius of
*/
CREATE OR REPLACE FUNCTION DEP_EXT_findloops(
        operation text,
        table_name text,
        primary_source_query text,
        primary_source_columns text[],
        cat_column text,
        temp_column text
    )
    RETURNS VOID AS $$
        DECLARE
            tail text;
            categorized text;
            cat_string text;
            cat_string2 text;
            sub_q text;
            s record;
            gSegment            geometry = NULL;
            gLastPoint          geometry = NULL;
```

(continué en la próxima página)

(proviene de la página anterior)

```
            gLastTrackID        text = NULL;
            gLoopPolygon        geometry = NULL;
            gRadius             numeric;
            iLoops              integer := 0;
            cdbi                bigint := 0 ;
        BEGIN

            IF operation = 'create' THEN

                EXECUTE 'DROP TABLE IF EXISTS ' || table_name;

                EXECUTE 'CREATE TABLE ' || table_name || '(cartodb_id bigint, track_
↪id text, loop_id integer, the_geom geometry(Geometry,4326), radius numeric)';

            ELSEIF operation = 'populate' THEN

                -- DEFAULTS
                -- -- no temporal column, then use cartodb_id
                IF  trim(temp_column) = '0' THEN
                   temp_column := 'cartodb_id';
                END IF;

                -- no category, no partition
                IF  trim(cat_column) = '0' THEN
                    categorized := ' order by ' || temp_column;
                    cat_string := '';
                ELSE
                    categorized := 'partition by ' || cat_column || ' order by ' ||␣
↪temp_column;
                    cat_string := cat_column;
                END IF;

                -- partition and sorting of the input
                sub_q := 'WITH '
                    ||  'prequery as('
                    ||      'SELECT '
                    ||      cat_string || ' as cat,'
                    ||      temp_column || ' as temp_column,'
                    ||      'the_geom as point'
                    ||      ' FROM ('
                    ||          primary_source_query
                    ||      ') _q'
                    ||   '),'
                    ||  'pts as('
                    ||      'SELECT '
                    ||      ' cat'
                    ||      ', temp_column'
                    ||      ', point'
                    ||      ', row_number() over(partition by cat order by temp_
↪column) as index'
                    ||      ' FROM prequery'
                    ||      ' ORDER BY cat, temp_column'
                    ||  ')'
                    ||      'SELECT '
                    ||      ' b.cat::text as track_id'
                    ||      ', ST_MakeLine(ARRAY[a.point, b.point]) AS geom'
                    ||      ' FROM pts as a, pts as b'
```

(continué en la próxima página)

(proviene de la página anterior)

```
                    ||      ' WHERE b.index > 1'
                    ||      ' AND a.index = b.index - 1'
                    ||      ' AND a.cat = b.cat '
                    ||      ' ORDER BY b.cat, b.temp_column';

                FOR s IN EXECUTE sub_q
                LOOP

                    -- restart when new track
                    if gLastTrackID <> s.track_id then
                        gSegment := null;
                        gLastPoint := null;
                        iLoops := 0;
                    end if;

                    -- build segments
                    if gSegment is null then
                        gSegment := s.geom;
                    elseif ST_equals(s.geom, gLastPoint) = false then
                        gSegment := ST_Makeline(gSegment, s.geom);
                    end if;

                    gLoopPolygon := ST_BuildArea(ST_Node(ST_Force2D(gSegment)));


                    if gLoopPolygon is not NULL and ST_Numpoints(gSegment) > 3 then

                        iLoops := iLoops + 1;
                        gRadius := (|/ ST_area(gLoopPolygon::geography)/PI());
                        gSegment := null;

                        EXECUTE
                        'INSERT INTO '
                        || quote_ident(table_name)
                        || ' VALUES('
                        || cdbi || ', '
                        || quote_literal(s.track_id) || ', '
                        || iLoops ||', '
                        || 'ST_GeomFromText(' || quote_literal(ST_
↪astext(gLoopPolygon)) || ', 4326), '
                        || gRadius || ')';

                        cdbi := cdbi +1;

                    end if;

                    IF  trim(cat_column) <> '0' THEN
                        gLastTrackID = s.track_id;
                    END IF;

                    gLastPoint := s.geom;

                END LOOP;


            END IF;
```

(continué en la próxima página)

(proviene de la página anterior)

```
        END;
$$ LANGUAGE plpgsql;
```

CAPÍTULO 9

Glosario

9.1 API

API es la abreviatura de "Interfaz de Programación de Aplicaciones" (Aplication Programming Interface en inglés). Es una "llave de acceso" a funciones que podemos utilizar de un servicio web provisto por un tercero, dentro de nuestra propia aplicación web, de manera segura y confiable.

9.2 backend

El backend es el conjunto de tecnologías que se encuentran del lado del servidor, es decir, se encargan de lenguajes como PHP, Python, .Net, Java, etc, interactuar con bases de datos, verificar manejos de sesiones de usuarios, montar la página en un servidor, etc.

9.3 Business Intelligence

Business Intelligence es el conjunto de metodologías, aplicaciones y tecnologías que permiten reunir, depurar y transformar datos de los sistemas transaccionales e información desestructurada (interna y externa a la compañía) en información estructurada, para su explotación directa (reporting, análisis OLTP / OLAP, alertas...) o para su análisis y conversión en conocimiento, dando así soporte a la toma de decisiones sobre el negocio

9.4 dashboard

Un *dashboard* o panel de control es una herramienta que muestra el estado actual de métricas e indicadores claves para una empresa u organización.

9.5 CRM

Customer Relationship Management o CRM es un término que se usa en el ámbito del marketing y ventas. En el ámbito del marketing y ventas, CRM se define como una estrategia orientada a la satisfacción y fidelización del cliente, por lo que a veces también es denominado Customer Service Management (Gestión de Servicio al Cliente). Esta tendencia se incluye dentro del Marketing relacional, el cual se centra en las relaciones con el cliente para conocer sus necesidades con el objetivo final de fidelizarlo.

9.6 ERP

El término ERP se refiere a *Enterprise Resource Planning*, que significa «sistema de planificación de recursos empresariales». Estos programas se hacen cargo de distintas operaciones internas de una empresa, desde producción a distribución o incluso recursos humanos.

9.7 firewall

Un *firewall* es un dispositivo de seguridad de la red que monitoriza el tráfico entrante y saliente y decide si debe permitir o bloquear un tráfico específico en función de un conjunto de restricciones de seguridad ya definidas.

9.8 frontend

El frontend son todas aquellas tecnologías que corren del lado del cliente, es decir, todas aquellas tecnologías que corren del lado del navegador web, generalizandose mas que nada en tres lenguajes, Html , CSS Y JavaScript.

9.9 HDFS

HDFS o *Hadoop Distributed File-system* es un sistema distribuido de archivos basado en Java para almacenar grandes volúmenes de datos

9.10 Location Intelligence

Location intelligence es una metodología para transformar datos geolocalizados en resultados de negocio. Los datos gelocalizados pueden ser direcciones postales, coordenadas o puntos, líneas o polígonos.

9.11 MapReduce

MapReduce es un framework que proporciona un sistema de procesamiento de datos paralelo y distribuido. Su nombre se debe a las funciones principales que son Map y Reduce.

9.12 MEAN

Arquitectura de desarrollo de software para aplicaciones distribuidas utilizando el mismo lenguaje JavaScript en todas sus fases y capas. MEAN es el acrónimo formado por las iniciales de las cuatro tecnologías principales que entran en juego: MongoDB, Express, AngularJS y Node.js

9.13 NoSQL

NoSQL es un término que describe las bases de datos no relacionales de alto desempeño. Las bases de datos NoSQL utilizan varios modelos de datos, incluidos los de documentos, gráficos, claves-valores y columnas.

9.14 OAuth

9.15 on-premise

En la empresa se llama solución on-premise a aquellos sistemas que son instalados en la propia empresa. Se trata de tener en "Casa" los servidores y el software que proporcionan un determinado servicio a la empresa. Soluciones tal y como puede ser un almacenamiento y gestión de archivos.

9.16 REST

REST es cualquier interfaz entre sistemas que use HTTP para obtener datos o generar operaciones sobre esos datos en todos los formatos posibles, como XML y JSON

9.17 SaaS

El software como servicio (SaaS) permite a los usuarios conectarse a aplicaciones basadas en la nube a través de Internet y usarlas. SaaS ofrece una solución de software integral que se adquiere de un proveedor de servicios en la nube mediante un modelo de pago por uso.

9.18 SDK

Un SDK (Software Development Kit), o kit de desarrollo de software, es un conjunto de herramientas que ayudan a la programación de aplicaciones para un entorno tecnológico particular

9.19 serverless

Las arquitecturas serverless reemplazan a las máquinas virtuales de larga duración con una capacidad de computación efímera que se crea para resolver una petición y desaparece después de su uso.

CAPÍTULO 10

Indices and tables

- genindex