Sqoop y Hive

Competencias

- Conocer el rol de `sqoop` para transferir datos desde RDBMS a HDFS.
- Generar configuraciones personalizadas en AWS EMR.
- Implementar comandos de `sqoop` para migrar tablas y bases de datos a HDFS.
- Conocer el funcionamiento y objetivos de Hive.
- Generar tablas y queries con Hive.
- Implementar puertos dinámicos para la conexión con interfaces de usuario en AWS EMR.

Motivación

A esta altura hemos trabajado con los elementos básicos del ecosistema Hadoop: HDFS, Hadoop Streaming y sabemos que éstos permiten asignar trabajos que son administrados por YARN. Con estos elementos podemos implementar tareas de MapReduce relativamente básicas como contar ocurrencia de registros en nuestros archivos que están distribuidos en HDFS. A lo largo de esta lectura trabajaremos con un par de herramientas del ecosistema Hadoop: `sqoop` y `hive`.

Ambas herramientas están asociadas a las bases de datos relacionales (de aquí en adelante, RDBMS - Relational DataBase Management Systems), como `MySQL`, `Postgreql`, etc. `Sqoop` tiene como objetivo la transferencia de datos alojados en tablas de una base de datos relacional a nuestro sistema distribuido de archivos `hdfs`. Así, podemos generar importaciones y exportaciones de datos en situaciones donde existe un acumen substancial de datos en formatos relacionales.

Por su lado, `hive` es un software orientado al data warehousing, basado fuertemente en la implementación de queries SQL. La capacidad de añadir abstracción mediante las queries SQL facilita el trabajo de analizar grandes volúmenes de datos sin la necesidad de implementar código en Python/Java para generar trabajos MapReduce. Partamos por interactuar con `sqoop`.

Configuración avanzada de AWS EMR

Por defecto, `sqoop` no está integrado en las versiones de EMR, para lo cual vamos a tener que generar una configuración avanzada de EMR. Accediendo a nuestro servicio en el AWS Management Console, vamos a buscar EMR y crear un nuevo cluster con los pasos que ya conocemos. Así, debemos ir al link Go to advanced options que se indica en la siguiente imágen.

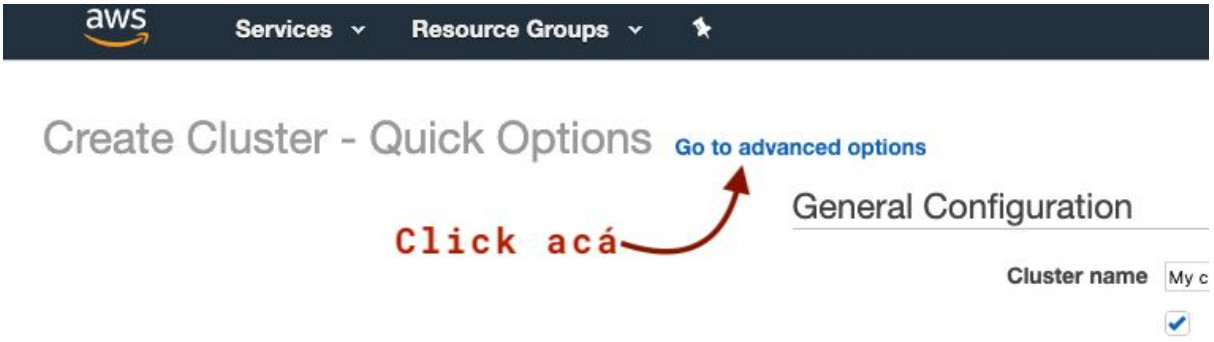


Imagen 1. AWS1.

Al acceder a *advanced options*, el primer paso será la selección del software necesario. Para este caso, nos aseguraremos que tanto `Hive`, `Hadoop`, `Hue`, `Sqoop` y `Pig` estén seleccionados, tal como lo indica la imagen.

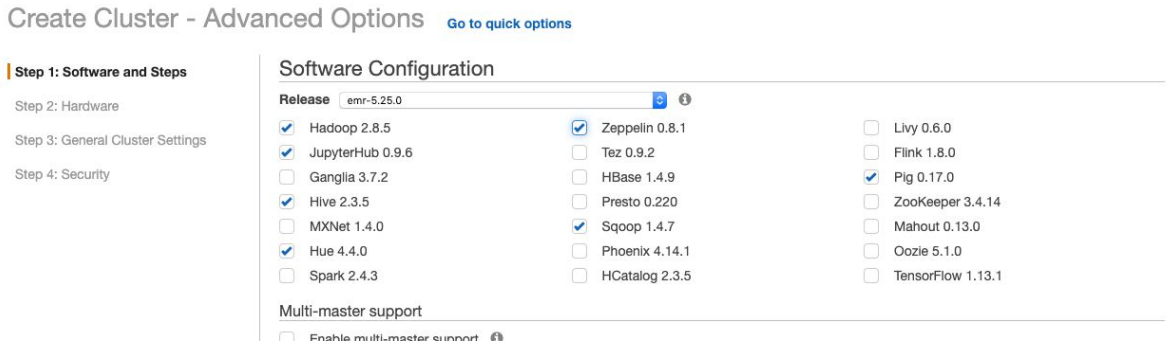


Imagen 2. Create.

Por el momento ignoremos la configuración de hardware de nuestro cluster, la cual dejaremos en un servicio de máster y dos servicios de slave. Esta es la configuración por defecto. Eventualmente en sus trabajos tendrán la oportunidad de aumentar la capacidad de cada servicio en RAM o disco duro (escalamiento vertical) o aumentar más máquinas de similares características (escalamiento horizontal).

El tercer paso es la configuración del nombre del cluster, así como la dirección del bucket donde se encontrarán los logs del proceso.

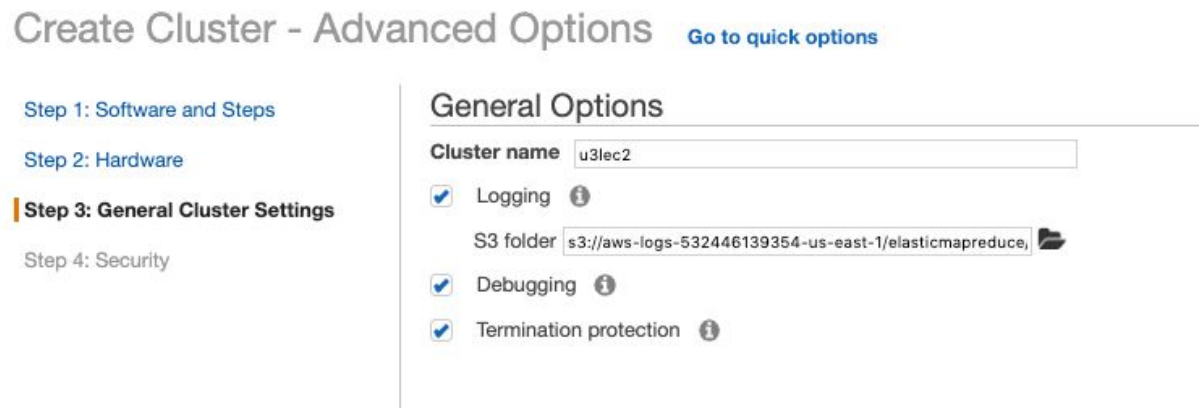


Imagen 3. Create.

La última fase es integrar nuestro archivo pem que tenemos en nuestro computador con el cluster a crear. Una vez integrada nuestra clave pem, estamos en condiciones de generar el cluster e interactuar.

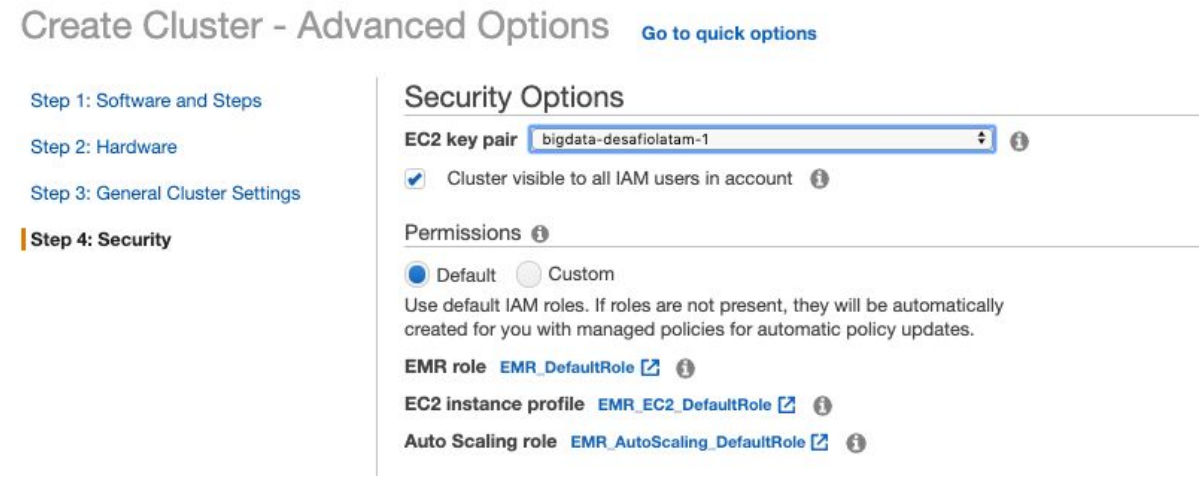


Imagen 4. Security.

Sqoop: Transfiriendo datos entre un RDBMS y HDFS

Sabemos que dentro de Hadoop tenemos una infinidad de maneras de trabajar con los datos que ya se encuentran alojados en HDFS. Un contratiempo es que Hadoop necesita de programas externos para poder interactuar con datos almacenados fuera de HDFS. Esta situación es frecuente dada la omnipresencia de datos estructurados en sistemas de bases de datos como Postgres. Apache Sqoop es la herramienta que permite al usuario extraer datos de una RDBMS hacia HDFS para posterior uso mediante Hadoop Streaming, o como veremos más tarde, Hive.

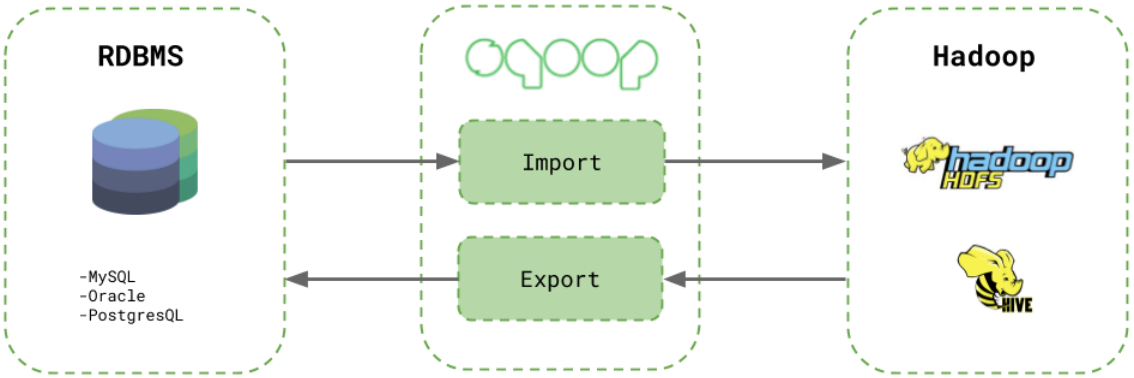


Imagen 5. Sqoop.

A grandes rasgos, sqoop permite realizar operaciones de importación de datos desde una RDBMS a HDFS, y operaciones de exportación desde resultados procesados en HDFS a RDBMS, para posterior uso por usuarios finales.

Una vez que nuestra instancia AWS EMR ya esté lista, podemos verificar la ubicación del binario de sqoop con el comando `which sqoop`. Si todo sale bien, deberíamos tener una respuesta similar a `usr/bin/sqoop`. Con este punto listo, estamos habilitados para implementar nuestro código con `sqoop`.

Importación de una tabla a HDFS

Si ejecutamos `sqoop` help en la consola, vamos a encontrar una lista con todos los posibles comandos a utilizar. Para efectos prácticos del curso, vamos a enfocarnos en dos comandos: `import` e `import-all-tables`. Partamos por generar un import básico de una tabla dentro de una base de datos.

La instrucción para importar una tabla dentro de una base de datos se encuentra en la siguiente imágen:

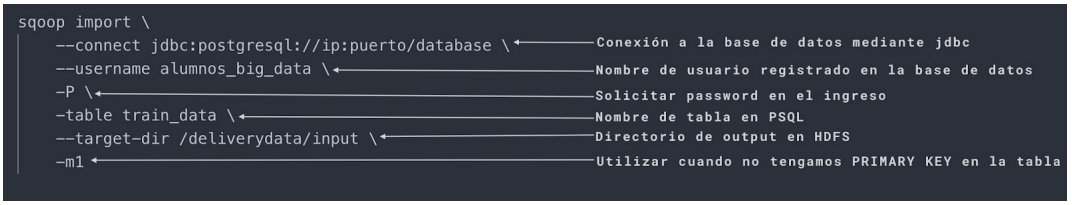


Imagen 6. Sqoop Import.

Cabe destacar que los `\` funcionan como salto de línea y son para legibilidad de la instrucción en la consola. Partimos por declarar que utilizaremos el comando `import`, donde el siguiente paso es la declaración de la base de datos que nos conectaremos. La instrucción en `--connect` es un tanto verbosa, dado que debemos especificar la ip del motor RDBMS , su puerto de conexión y la base de datos específica a conectar. La instrucción `jdbc:postgresql://` se conoce como el **Conector de sqoop**, que definiremos con detención un poco más adelante.

Posterior a la definición de la conexión, debemos ingresar el nombre del usuario con la opción `--username`. La contraseña se declara mediante la opción `-P`. Esta opción permitirá al usuario ingresar la contraseña una vez que el comando se ejecuta. La opción `-table` indica el nombre de la tabla a transferir, así como `--target-dir` especifica la dirección en ruta absoluta de la transferencia en `hdfs`. Por último, se implementó la opción `-m 1` para declarar la cantidad de mappers asignados en la tarea.

Para este ejemplo desarrollaremos la transferencia de datos desde un motor Postgresql montado en la siguiente dirección: `ls-07d09717f869f2ab37ff6e2b5aa45a1441ebfef3.cll3q1dgryg1.us-east-1.rds.amazonaws.com.` La base de datos específica a conectarnos se llama `deliverydata`, que corresponde a una serie de transacciones de una aplicación de pedidos. Utilizaremos el puerto de conexión por defecto de `Postgresql`, el 5432. Con la dirección ip, el nombre de la base de datos y el puerto, podemos generar nuestra ruta de conexión. Esta debe estar compuesta por el **conector** de sqoop con un motor de base de datos específico, la ip de ingreso seguido de su puerto de entrada, así como la base de datos.

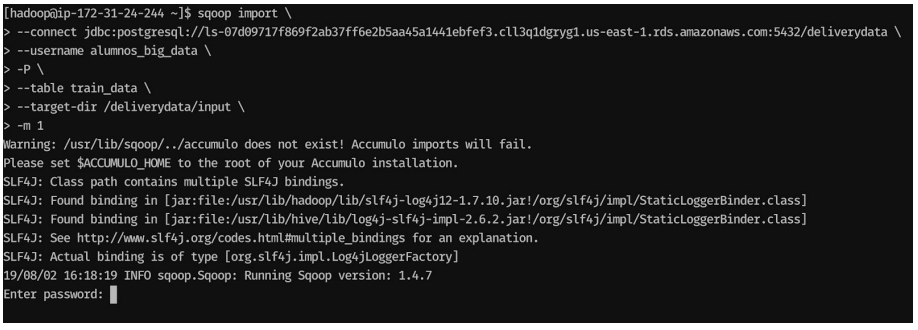


Imagen 7. Conector.

Conectores de Sqoop

Sqoop tiene un framework que permite importar y exportar datos de cualquier sistema de bases de datos relacional. Para esto, Sqoop hace uso de conectores. Un conector es un driver JDBC para la base de datos específica (MySQL, PostgreSQL, Oracle) con la cual vamos a transferir los datos. Si bien existe un conector JDBC genérico que permite conectarse a cualquier base de datos, para esta sesión implementaremos el conector específico de `PostgreSQL`.

Dentro de la base de datos `deliverydata` vamos a encontrar las tablas `train_data` y `test_data`. Para este ejercicio vamos a importar la primera. Finalmente, vamos a declarar la ruta de destino de nuestra transferencia en HDFS, la cual debe ser detallada como ruta absoluta.

Si la ejecución del comando es exitosa, `sqoop` iniciará una tarea MapReduce que conectará con la base de datos Postgres y leerá la tabla. Por defecto, la tarea a implementar será con cuatro mappers para acelerar el proceso de importación. Cada tarea escribirá los resultados en un archivo distinto **en el mismo directorio**. Una vez que la tarea termine, tendremos nuestro archivo en la dirección especificada, tal como se muestra en la imagen.

```
[hadoop@ip-172-31-24-244 ~]$ hdfs dfs -ls /
Found 5 items
drwxr-xr-x   - hdfs   hadoop          0 2019-08-02 16:01 /apps
drwxr-xr-x   - hadoop hadoop          0 2019-08-02 16:18 /deliverydata
drwxrwxrwt   - hdfs   hadoop          0 2019-08-02 16:04 /tmp
drwxr-xr-x   - hdfs   hadoop          0 2019-08-02 16:01 /user
drwxr-xr-x   - hdfs   hadoop          0 2019-08-02 16:01 /var
[hadoop@ip-172-31-24-244 ~]$ hdfs dfs -ls /deliverydata/input
Found 2 items
-rw-r--r--   1 hadoop hadoop          0 2019-08-02 16:19 /deliverydata/input/_SUCCESS
-rw-r--r--   1 hadoop hadoop      63503 2019-08-02 16:19 /deliverydata/input/part-m-00000
[hadoop@ip-172-31-24-244 ~]$ hdfs dfs -cat /deliverydata/input/part-m-00000 | head
98,VII,Free,18,5,39.06417712024473,Asian,10.429259959053653
14,IV,Trimestral,16,2,32.82654382121793,Italian,7.9008918807783
72,V,Free,17,3,7.556762164991376,Mexican,14.689806469978173
11,VIII,Semestral,12,1,34.20068369067669,Indian,12.51384063783114
81,VII,Free,27,3,21.51587429598476,French,5.141044125803192
85,IV,Semestral,21,5,29.970787225017745,French,7.534726624956967
21,VII,Prepaid,10,5,26.25805639563678,French,8.989187910382848
68,II,Yearly,14,2,41.04552212183895,Indian,11.130075681811803
19,VIII,Trimestral,13,1,11.593979232172991,French,11.358517681824518
77,IV,Yearly,14,3,33.04164267226024,Mexican,9.484584557806512
[hadoop@ip-172-31-24-244 ~]$
```

Imagen 8. Directorio.

Los resultados que se encuentran en el archivo `part-m-00000` corresponden a la transferencia exitosa de la tabla `train_data`. Por defecto, Sqoop generará un archivo de valores separado por comas con los datos.

Importación de múltiples tablas mediante Sqoop

Si quisiéramos importar todas las tablas dentro de una base de datos en Postgres, podríamos utilizar una sintaxis similar que se detalla a continuación. Uno de los elementos diferenciadores es el hecho que no especificamos la tabla específica.

```
[hadoop@ip-172-31-24-244 ~]$ sqoop import-all-tables \
> --connect jdbc:postgresql://ls-07d09717f869f2ab37ff6e2b5aa45a1441ebfef3.cll3q1dgryg1.us-east-1.rds.amazonaws.com:5432/deliverydata \
> --username alumnos_big_data \
> -P \
> -m 1
Warning: /usr/lib/sqoop/../accumulo does not exist! Accumulo imports will fail.
Please set $ACCUMULO_HOME to the root of your Accumulo installation.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/usr/lib/hadoop/lib/slf4j-log4j12-1.7.10.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/usr/lib/hive/lib/log4j-slf4j-impl-2.6.2.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
12319/08/02 17:16:47 INFO sqoop.Sqoop: Running Sqoop version: 1.4.7
Enter password:
19/08/02 17:16:55 INFO manager.SqlManager: Using default fetchSize of 1000
19/08/02 17:16:56 INFO tool.CodeGenTool: Beginning code generation
19/08/02 17:16:56 INFO manager.SqlManager: Executing SQL statement: SELECT t.* FROM "train_data" AS t LIMIT 1
19/08/02 17:16:56 INFO orm.CompilationManager: HADOOP_MAPRED_HOME is /usr/lib/hadoop-mapreduce
Note: /tmp/sqoop-hadoop/compile/2d5565602a86093f0d17fd8f17a8409d/train_data.java uses or overrides a deprecated API.
Note: Recompile with -Xlint:deprecation for details.
19/08/02 17:17:01 INFO orm.CompilationManager: Writing jar file: /tmp/sqoop-hadoop/compile/2d5565602a86093f0d17fd8f17a8409d/train_data.jar
19/08/02 17:17:01 WARN manager.PostgresqlManager: It looks like you are importing from postgresql.
19/08/02 17:17:01 WARN manager.PostgresqlManager: This transfer can be faster! Use the --direct
19/08/02 17:17:01 WARN manager.PostgresqlManager: option to exercise a postgresql-specific fast path.
19/08/02 17:17:01 INFO mapreduce.ImportJobBase: Beginning import of train_data
19/08/02 17:17:01 INFO Configuration.deprecation: mapred.jar is deprecated. Instead, use mapreduce.job.jar
19/08/02 17:17:02 INFO Configuration.deprecation: mapred.map.tasks is deprecated. Instead, use mapreduce.job.maps
19/08/02 17:17:02 INFO client.RMProxy: Connecting to ResourceManager at ip-172-31-24-244.ec2.internal/172.31.24.244:8032
```

Imagen 9. Tablas.

Una vez que nuestros datos estén transferidos en HDFS, estamos habilitados para analizarlos. A continuación presentaremos Hive como una herramienta de análisis.

Hive

Sabemos que el ecosistema Hadoop surgió como una manera costo-eficiente para trabajar con grandes volúmenes de datos. Esto impone el paradigma MapReduce para parcelar las representaciones de un programa en unidades que pueden ser distribuidas a lo largo de un cluster, permitiendo la escalabilidad horizontal. El problema es que muchas veces vamos a tener una infraestructura de datos ya existente previa a la incorporación de Hadoop, la cual estará basada en SQL.

Hive surgió de la necesidad para manejar y aprender de grandes cantidades de datos que Facebook tenía almacenado en HDFS. Se creó para facilitar el análisis por parte de profesionales con experiencia en la creación de consultas en SQL, pero que no tenían la expertise técnica para traducirlas a tareas MapReduce.

Hive provee un dialecto SQL llamado Hive Query Language (`hql`) para realizar consultas de datos en un cluster Hadoop. De esta manera, podemos evitar operaciones que no son triviales en Java, el lenguaje de facto en Hadoop. Hive está orientado para aplicaciones de *data warehousing*, donde datos estáticos son analizados y no es necesario tener una respuesta de los datos en tiempo real o cuando los datos no se cambian de manera rápida.

Hive no es un motor de base de datos completos. No permite actualizaciones a nivel de registros, no se puede actualizar, insertar o eliminar. Solo se pueden generar nuevas tablas a partir de queries o exportar las queries a archivos. Dado que utiliza MapReduce tras bambalinas, presenta un overhead substancial respecto a la latencia en las queries. Tampoco provee de las características cruciales requeridas para un OLTP (Online Transaction Processing). Es más cercano a una herramienta OLAP.

Interacción con Hue

Existen variadas formas de trabajar con Hive, para la cual nosotros haremos uso de Hue. Hue es un editor de consultas SQL basado en la nube, el cual es open source creado por Cloudera. Facilita la interacción entre el usuario y el servidor donde se encuentra alojado Hive. Cuando configuramos nuestra instancia de trabajo en AWS EMR, agregamos Hue dentro de las prestaciones de software. Para habilitar Hue desde la nube, debemos incorporar los siguientes elementos:

1. **Hacer click en enable web connection en la página de administración de nuestro cluster**: El primer punto es seguir las instrucciones provistas en el link indicado. Se debe integrar el código habilitado en el modal de AWS EMR con el plugin Foxy Proxy. https://getfoxyproxy.org/.

Imagen 10. Instrucciones.

Posterior a la correcta instalación de las instrucciones por Amazon, podemos habilitar las conexiones `emr-socks-proxy` en la ventana de extensiones de su navegador.

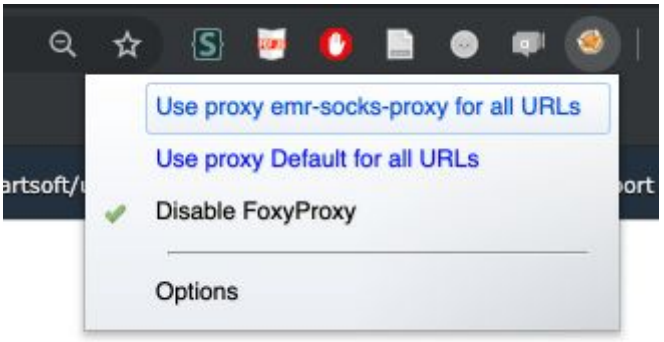


Imagen 12. Conexión.

2. **Establecer el puerto dinámico entre nuestro computador y el servidor**: Una vez que habilitamos nuestro servicio en el navegador, debemos habilitar un puerto dinámico en nuestro terminal para poder interactuar con el servicio. Esto lo logramos con la sintaxis `ssh -i ~/permisos.pem -ND 8157 usuario@servidor`. Un aspecto a considerar es que este puerto debe mantenerse abierto durante toda la interacción con el servicio.

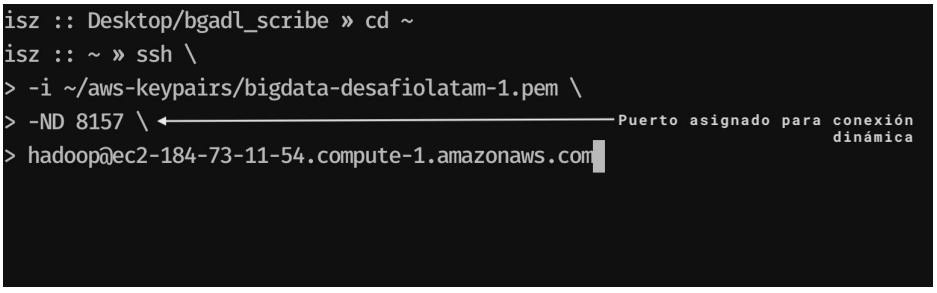


Imagen 13. Puerto asignado.

3. **Hacer click en el link de Hue**: Una vez que tenemos la conexión con el puerto dinámico habilitada y el web proxy habilitado, nuestra página de administración de la instancia tendrá actualizada la línea **Connections** con los links, donde haremos click en Hue. Una vez dentro de Hue, vamos a generar un nombre de usuario y una contraseña.

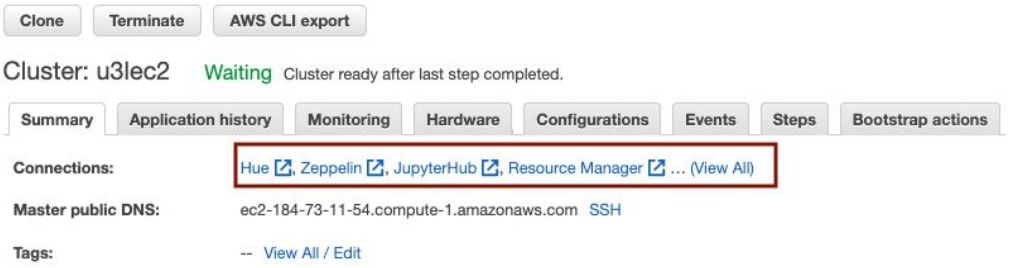


Imagen 14. Connections.

La administración de Hue se puede entender en tres zonas. A la izquierda tendremos nuestro directorio de archivos, donde podremos interactuar con motores de bases relacionales, archivos en nuestro sistema local y buckets de s3 asociados a la cuenta. En la parte superior tenemos el botón de definición de tareas, donde podremos interactuar con distintas prestaciones del ecosistema Hadoop tales como Hive, Impala, Pig, Sqoop, entre otros. Finalmente tenemos el **Editor y Ejecutor de Queries**, que será donde interactuaremos con Hive.

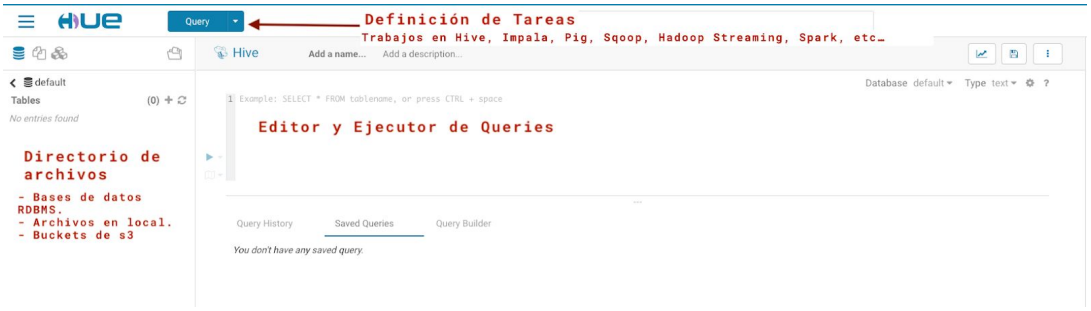


Imagen 15. Definición.

Definiendo nuestra tabla en Hive

Ahora vamos a definir nuestra primera tabla con Hive. Como ya se mencionó anteriormente, Hive cuenta con su propio lenguaje de queries basado en SQL. En base a los datos importados con sqoop, ahora definiremos la tabla con los datos de entrenamiento de la base de datos. La definición de las columnas y el tipo de datos es idéntica a la implementada en SQL. Los elementos que varían son las últimas dos líneas, donde definimos que tendremos un formato orientado por filas y delimitado, donde cada uno de los campos estará separado por comas.

Digresión: Tablas internas vs Tablas externas en Hive

La forma de crear tabla que definimos abajo se conoce como Internal table, que es la opción por defecto. Un aspecto importante a considerar es el hecho que en estas tablas Hive manejará los directorios asociados a la tabla, y será capaz de eliminar los datos asociados en HDFS si es que ejecutamos `DROP TABLE`. Por lo general, se prefiere implementar una tabla Externa con la opción `CREATE TABLE EXTERNAL nombre_tabla`.

Las instrucciones se implementan en el *Editor de Queries* y las ejecutamos con el botón de `play` o bien con `Ctrl + Enter`. El segundo panel hace referencia al output standard del terminal, que reporta todos los eventos relacionados a la ejecución del código, así como posibles errores y tracebacks. Finalmente tenemos el *Historial de Queries* con el registro de las queries iniciadas en la sesión, así como su status.

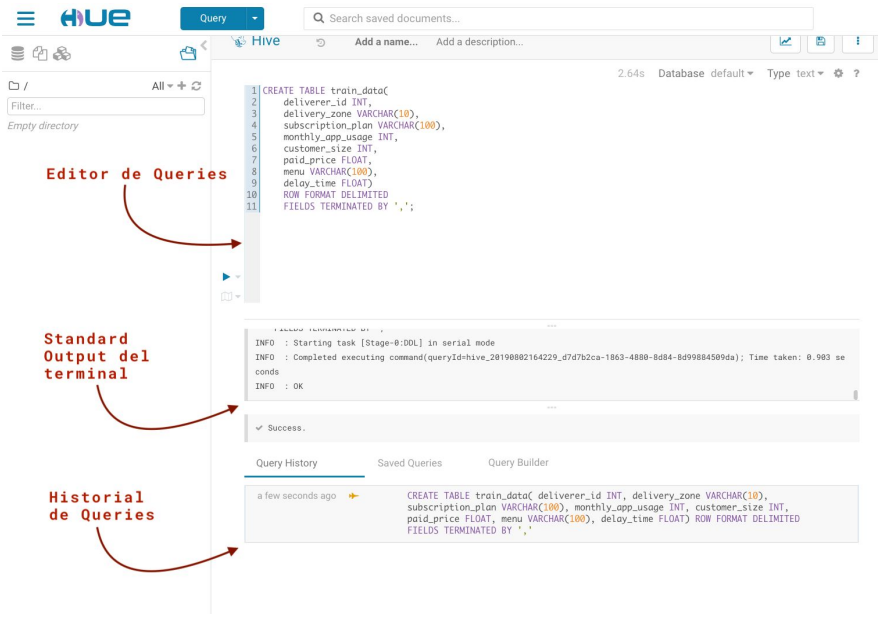


Imagen 16. Queries.

Ya tenemos definida nuestra tabla en Hive, de una forma similar a como lo haríamos en SQL. El siguiente paso es incorporar los datos importados previamente de nuestra base RDBMS a HDFS mediante `sqoop`. Estos datos se van a encontrar en nuestra ruta `/deliverydata/input`. La instrucción es `LOAD INPATH DATA 'hdfs:///deliverydata/input INTO TABLE train_data`. Si la ejecución es correcta, la tarea en Hive se mostrará en el panel del Standard Output de Hue.

En esta etapa puede existir un problema asociado a la denegación de permisos en la fase de escritura de la tabla en hive. Para solucionarlo, debemos ejecutar `sudo -u hdfs hadoop fs -chown -R usuario_de_hue /ruta/absoluta/a/datos` **desde nuestra instancia de trabajo AWS EMR**. Mediante esta línea otorgaremos permisos al usuario creado en Hue para poder interactuar efectivamente con los datos alojados en HDFS.

Digresión: Schema on Read

En la carga de datos en las RDBMS, la base de datos tiene un control total sobre el almacenamiento comportándose como un punto vigilado. De esta manera, la base de datos puede cuestionar el esquema (o declaración de las columnas y tipos de datos) como escrito. Esto se conoce como **schema on write**.

Por su parte, Hive no tiene control sobre el almacenamiento. Existen variadas formas de crear, modificar e incluso dañar los datos por los cuales Hive realizará queries. Así, Hive sólo puede implementar queries en lectura. Esto se conoce como **schema on read.**

Si tenemos un esquema de datos incongruente (algunos tipos de datos están mal especificados), Hive intentará ingresar los datos de la mejor manera que pueda.

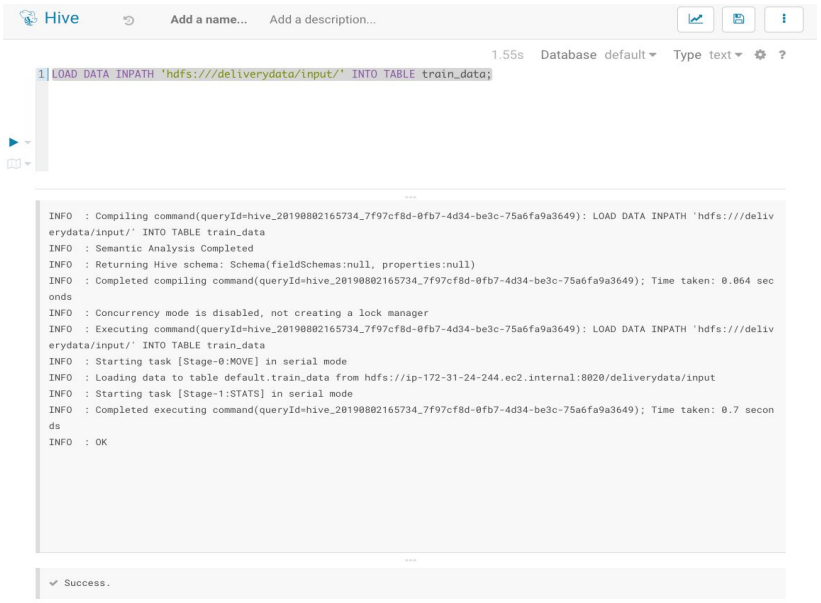
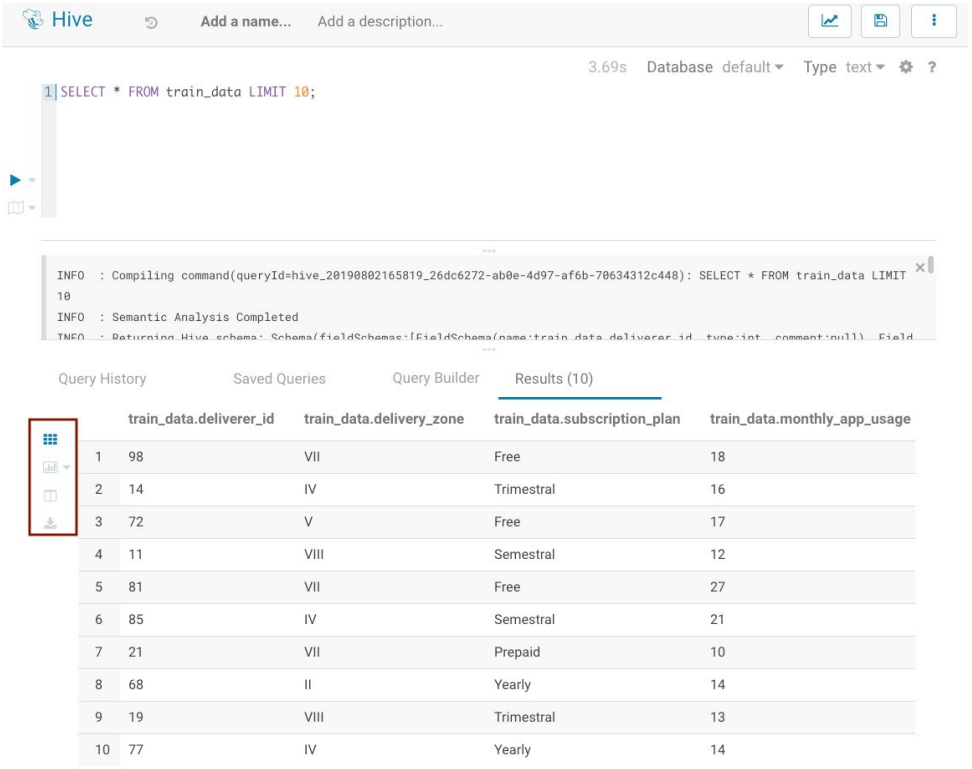


Imagen 17.Digresión.

Una vez que tenemos nuestra tabla habilitada, podemos implementar queries tal como lo haríamos con SQL. Posteriormente aprenderemos sobre las peculiaridades de Hive con el consumo de datos semiestructurados y la definición de funciones por parte del usuario. Por ahora concentrémonos en las formas en que podemos extraer resultados de una query de Hive en Hue.

La query más simple que podemos realizar es probablemente extraer una serie finita de observaciones mediante `SELECT * FROM train_data LIMIT 10;`. Al ejecutar la query, el panel de Standard Output reportará todo el proceso realizado por Hive. En el panel de abajo donde estaba el Historial de Queries, ahora también encontramos una pestaña de Results, donde se presentará el resultado de nuestra query. En el área destacada con el rectángulo rojo vamos a encontrar las distintas formas de visualizar nuestra Query. En particular, es posible visualizar los resultados mediante gráficos interactivos.



	train_data.deliverer_id	train_data.delivery_zone	train_data.subscription_plan	train_data.monthly_app_usage
1	98	VII	Free	18
2	14	IV	Trimestral	16
3	72	V	Free	17
4	11	VIII	Semestral	12
5	81	VII	Free	27
6	85	IV	Semestral	21
7	21	VII	Prepaid	10
8	68	II	Yearly	14
9	19	VIII	Trimestral	13
10	77	IV	Yearly	14

Imagen 18. Hive.

¿Qué pasa tras bambalinas en Hive?

Si bien no es necesario tener un conocimiento acabado sobre cómo funciona Hive a profundidad, si debemos tener nociones sobre el flujo de trabajo para entender qué fue lo que hicimos. A continuación se define brevemente qué pasa tras bambalinas en Hive. Partamos por definir los componentes básicos de su arquitectura:

1. **Capa de Usuario**: Por lo general interactuamos con Hive mediante alguna interfaz gráfica como Cloudera Hue o desde la línea de comando con `beeline`. Para efectos prácticos ambas apuntan a lo mismo, con la salvedad que Hue es un ambiente de trabajo más desarrollado con la posibilidad de visualizar queries e integrar múltiples formas de datos.

2. **Capa de Clientes**: Hive permite interactuar con todas las aplicaciones escritas en lenguajes como `C++`, `Python`, `Java`, entre otros; mediante el uso de dirves ODBC, JDBC y Thrift. De esta manera facilita la implementación de scripts ad-hoc en múltiples lenguajes.

3. **Capa de Servidor**: En la capa de servidor nos vamos a encontrar con el Hive Server que está encargado de administrar las tareas enviadas por los clientes. Del servidor emanan las tareas al **Hive Driver** (también conocido como Deriver) que está encargado de recibir las queries en una serie de pasos:

    - **Complier**: En esta fase Hive se asegura de revisar el tipo de datos contra el esquema de datos presente en el metastore, así como análisis sintáctico de las queries diseñadas por el usuario.

    - **Optimizer**: Mediante Grafos Acíclicos Dirigidos, se optimiza el plan lógico para implementar en tareas MapReduce y HDFS.

    - **Executor**: Posterior a la ejecución de los pasos, se ejecutan los pasos.

4. **Interacción con Procesos y Hadoop**: Dado que Hive es parte del ecosistema Hadoop, hace uso de HDFS para importar/exportar datos y resultados. Una de las características especiales de Hive es el uso de un Metastore, donde se almacena tanto el esquema de la tabla declarada, así como la ubicación de los datos y particiones en una base de datos relacionada. Provee a los clientes de información para acceder a los datos.

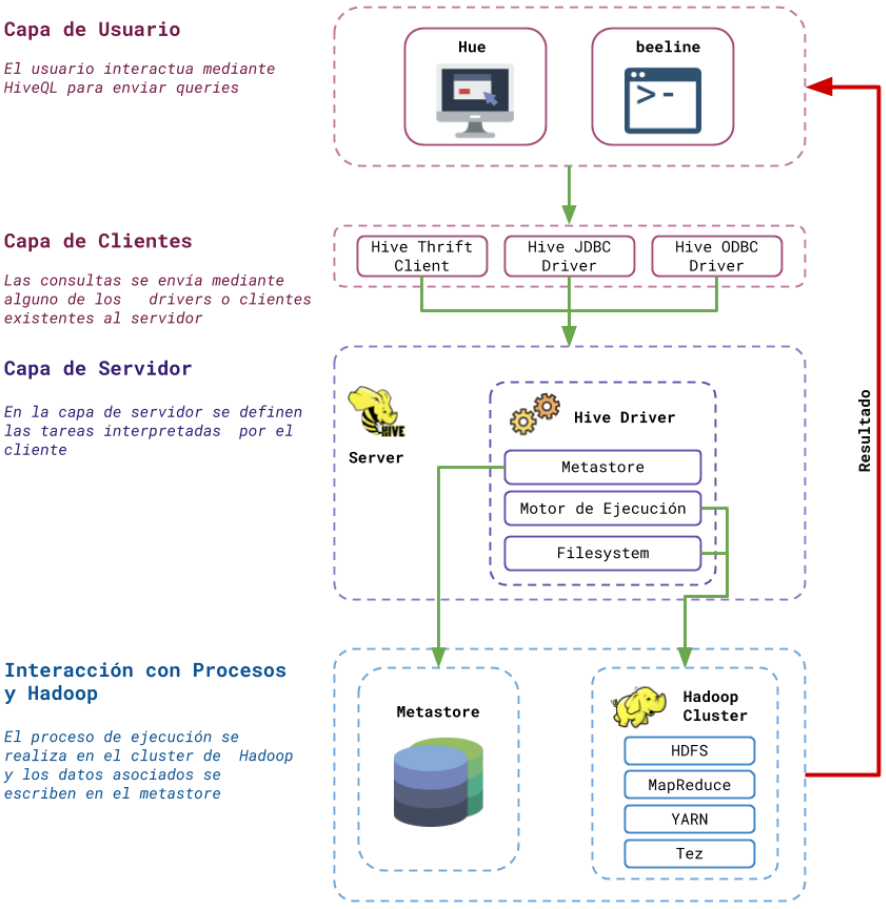


Imagen 19. Resultado.

El flujo de una tarea en Hive se puede resumir de la siguiente manera:

1. A nivel de usuario, se ejecuta la interfaz en un driver específico (por defecto es Thrift).

2. El driver crea una sesión para la query ejecutada. Envía la query al servidor para generar un plan de ejecución.

3. El compilador llama a los metadatos asociados a la query.

4. Con los metadatos asociados, el compilador genera los chequeos de tipo de datos y genera un Grafo Acíclico Dirigido que puede ser una operación MapReduce, Una operación en los metadatos o acceso a HDFS.

5. Con la tarea representada en un Grafo Acíclico Dirigido generado, se procede a la ejecución.

Carga de archivos semiestructurados en Hive

Otra característica de Hive es el soporte que presenta a muchos de los tipos de datos primitivos originales en bases de datos relacionales, así como tipos de datos de colección que rara vez se encuentran en RDBMS. En comparación a otros motores de base de datos, Hive presenta una mayor flexibilidad en cómo los datos se cifran en archivos. La mayoría de las bases de datos toman un control total de los datos, cómo persisten en el disco y su ciclo de vida. Hive facilita el manejo y el procesamiento de datos mediante distintas herramientas para controlar estos aspectos.

Para este ejemplo, vamos a cargar un archivo con extensión json que hace representación a una serie de registros asociados a una aplicación de delivery de supermercado. Tomemos un registro aleatorio:

```
{"order_id": 283275,
"priority": 0,
"purchases": ["Spaghetti", "Dishwasher Soap", "Dishwasher Soap",
"Avocado"],
"ammount": 76,
"storage_location": {
    "street": "1019 Kyle Stream Apt. 825",
    "city": "Jasonview",
    "state": "Illinois"},
"delivery_address": {"street": "0728 Miller Stravenue Suite 277",
                     "city": "Joefort",
                     "state": "Illinois"}
}
```

El registro se compone de una serie de números enteros que representan el identificador de la compra, si es que fue o no prioridad, así como el total de la compra. La particularidad de este registro es que tenemos los campos purchases, `storage_location` y `delivery_address` que son arreglos y diccionarios. Para poder ingresarlos en una tabla de Hive, necesitamos identificar cada una de estas representaciones dentro del lenguaje Hive. A continuación se presenta una tabla resumen de los tipos de datos permitidos:

Tipo	Descripción	Ejemplo
`ARRAY`	Secuencia ordenada donde los elementos corresponden a un mismo tipo, utilizando números enteros basados en cero. Si tenemos una columna `nombre` del tipo `ARRAY ['Perico', 'Los Palotes']`, podemos acceder al segundo elemento con `nombre[1]`.	`array('Perico', 'Los Palotes')`
`MAP`	Una colección de tuplas en pares `valor-llave` donde los campos se pueden acceder en notación de array. Si tenemos una columna nombre del tipo `MAP (primer -> Perico, ultimo -Los Palotes)`, para acceder al último nombre podemos hacerlo como `nombre['ultimo']`.	`map(primer, 'Perico', ultimo, 'Los Palotes')`
`STRUCT`	Análogo a un "objeto". Los campos dentro de un `STRUCT` se pueden acceder mediante la notación de punto. Si la columna nombre es del tipo `STRUCT {primer STRING, ultimo STRING}`, entonces el primer nombre del campo puede ser referenciado como nombre.primer	`struct('Perico', 'Los Palotes')`

Tabla 1. Hive.

SerDe: Serializer/Deserializer

El segundo elemento a considerar para realizar la carga de un archivo semiestructurado en Hive es el SerDe. Esta es una implementación basada en Java que permite encapsular la lógica de convertir los bytes no estructurados de un registro, en un registro válido para su posterior implementación en Hive. Por defecto Hive viene con una cantidad de `SErDe` para múltiples formatos. De manera interna, el motor de Hive leerá un archivo de manera cruda, y esta lectura será posteriormente parseada con el método `SerDe.deserialize()` para tener una representación válida en la tabla.

Práctico: Importación de Json con SerDe en Hive

Para efectos de replicabilidad de la lectura, es necesario copiar el archivo `complete_groceries.json` que se encuentra en el bucket del curso `s3://bigdata-desafio/lecture-replication-data/groceries/` en un bucket local de nuestra propiedad. En el caso de la lectura, vamos a alojar el archivo en una carpeta llamada `groceries_app`. Para esto, ejecutamos la línea `aws s3 cp s3://bigdata-desafio/lecture-replication-data/groceries/complete_groceries.json s3://iszbucketaws/groceries_app/`.

Si seguimos nuestro registro ejemplo, resulta que tendremos los siguientes tipos de datos:

- Dado que los elementos dentro de las compras es una lista de cadenas, el tipo de dato será `purchases ARRAY<STRING>`.
- Dado que la ubicación de la bodega es un diccionario donde cada valor es una cadena, el tipo de dato será `storage_location STRUCT<street:STRING, city:STRING, state:STRING>.`
- Dado que la dirección de entrega es un diccionario donde cada valor es una cadena, el tipo de dato será `delivery_address STRUCT<street:STRING, city:STRING, state:STRING>.`

Nuestra query de creación de tabla será idéntica a una conformación de tabla normal, con un par de elementos especiales:

- **Deserializador**: Vamos a incluir un deserializador para archivos Json que se puede agregar a nivel de fila con `ROW FORMAT SERDE 'org.openx.data.jsonserde.JsonSerde'`.

- **Datos**: vamos a incorporar automáticamente los datos mediante el argumento `LOCATION s3://iszbucketaws/groceries_app/`.

Imagen 20. Datos.

Ya con nuestros datos cargados en la tabla, veamos cómo se estructuran los datos con `SELECT * FROM orders LIMIT 10;`

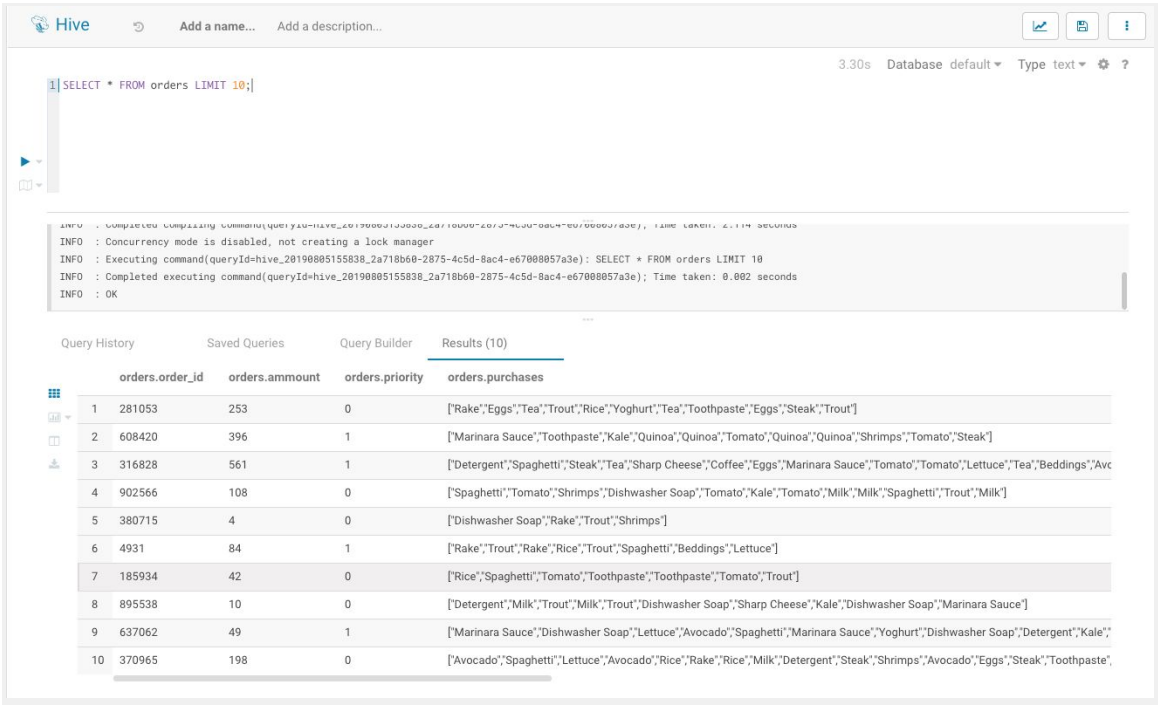


Imagen 21. Limit.

Desde la empresa dueña de la aplicación, nos solicitan los siguientes insights:

- **Query 1**: Quieren ver si eliminan la opción de prioridad en los pedidos en algunos estados desde la ubicación de las bodegas, para lo cual le solicitan sacar el promedio de prioridad a nivel de estados.

Para este caso, debemos identificar cómo acceder a la llave state dentro de la columna `storage_location`. Resulta que de manera similar a cómo lo hacemos en Python, podemos acceder al valor asociado mediante la notación `struct.key` o `struct['key']`. Con esto ya podemos identificar cómo agrupar los registros. Resulta que para extraer el promedio de la columna priority podemos utilizar una *User-Defined Aggregate Function* (UDAF), que permita agregar los elementos de una columna. En este caso implementaremos la UDAF avg que representa el promedio.

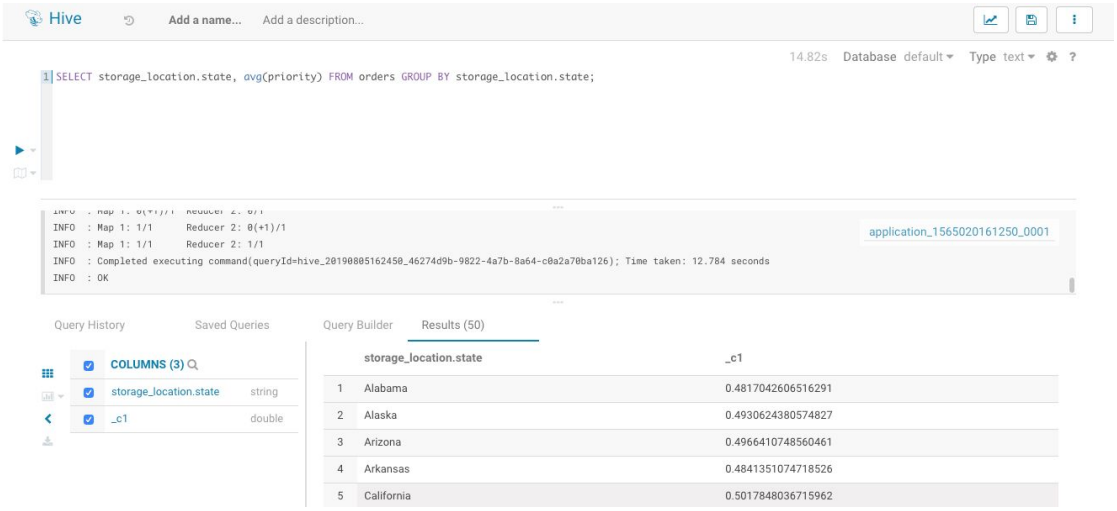


Imagen 22. UDAF.

- **Query 2**: Desean ver la correlación existente entre cantidad de compras y el total de la cuenta a nivel de estados.

Para poder realizar esta query debemos contar la cantidad de elementos dentro del array purchases, esto lo podemos lograr con el método `size`. Este método devolverá un número entero el cual podremos asociar la cantidad de elementos comprados con el total de la compra utilizando el método `corr`. Este método funciona de similar manera a como funciona en `scipy`, que necesita de dos columnas con valores del tipo numérico para funcionar. De similar manera, vamos a agrupar por el estado con `GROUP BY`.

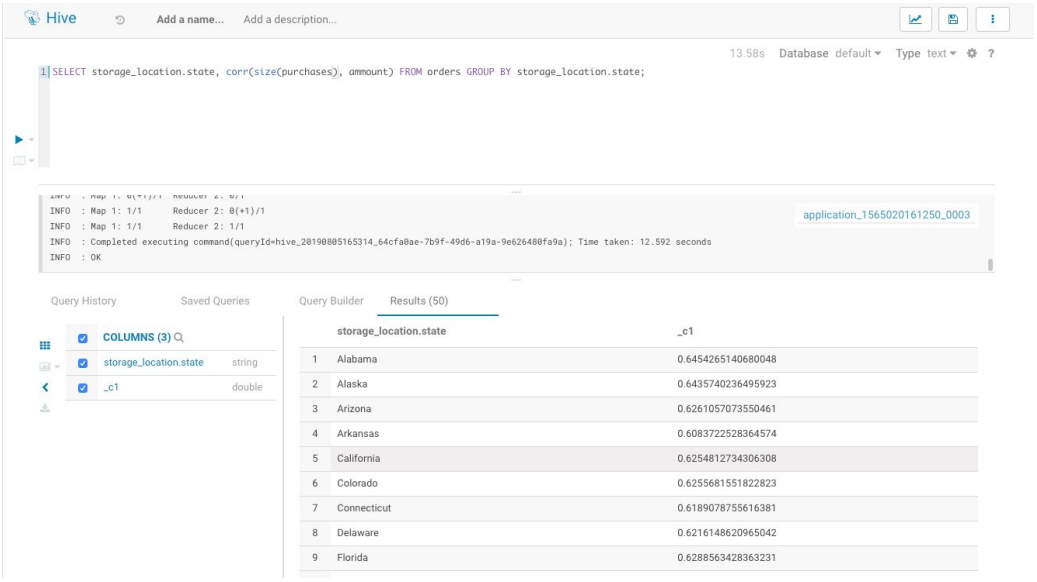


Imagen 23. Query2.

- **Query 3**: Desean ver cuánto Kale se consume a nivel de estados.

La tercera query busca contar la cantidad de veces que se ordena Kale en cada compra. Resulta que dentro del array purchases se pueden repetir los elementos. Podríamos implementar el método `array_contains` para evaluar si se encuentra, pero con el contratiempo de que `array_contains` devuelve solo booleanos, y no un número entero. Para resolver esto, implementaremos la opción `case` when que funciona relativamente similar a un condicional `if`. Una vez que identifiquemos las ocurrencias dede Kale, podemos contarlas.

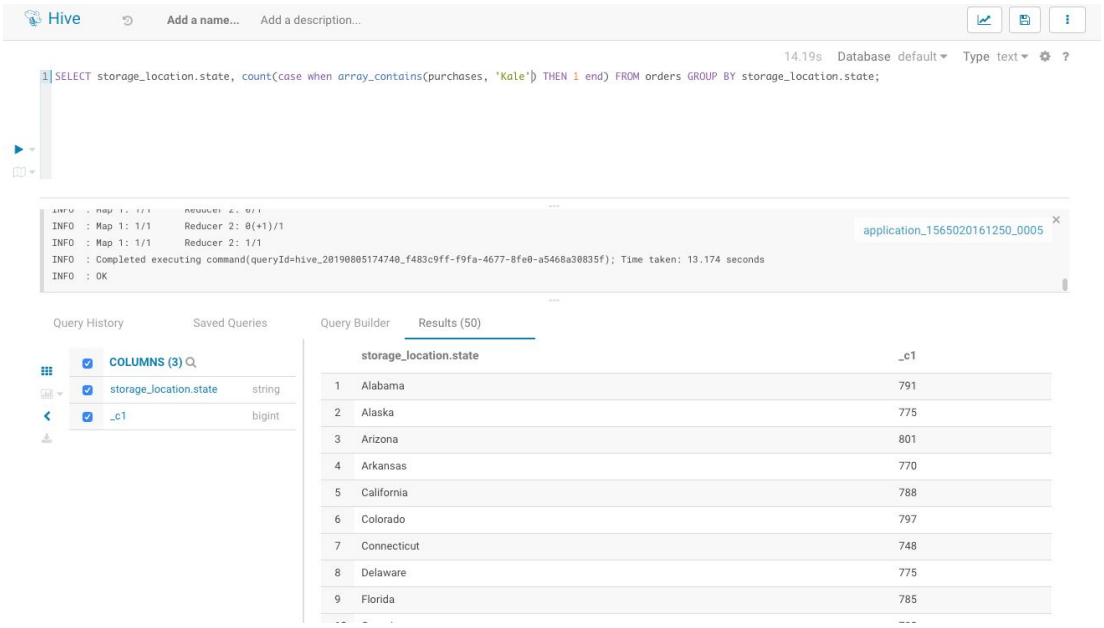


Imagen 24. Query3.

- **Query 4**: Desde Operaciones les levantan la alerta de algunos casos donde las órdenes llegan a otro estado distinto al de la bodega. Cuente su ocurrencia a nivel de estado.

La última query busca contar la cantidad de despachos donde la orden llegan a otro estado distinto al de la bodega. Para resolver esto debemos implementar un `WHERE` entre los `structs storage_location.state` y `delivery_address.state`, contar las ocurrencias y agruparlas por estado del `delivery_address`.

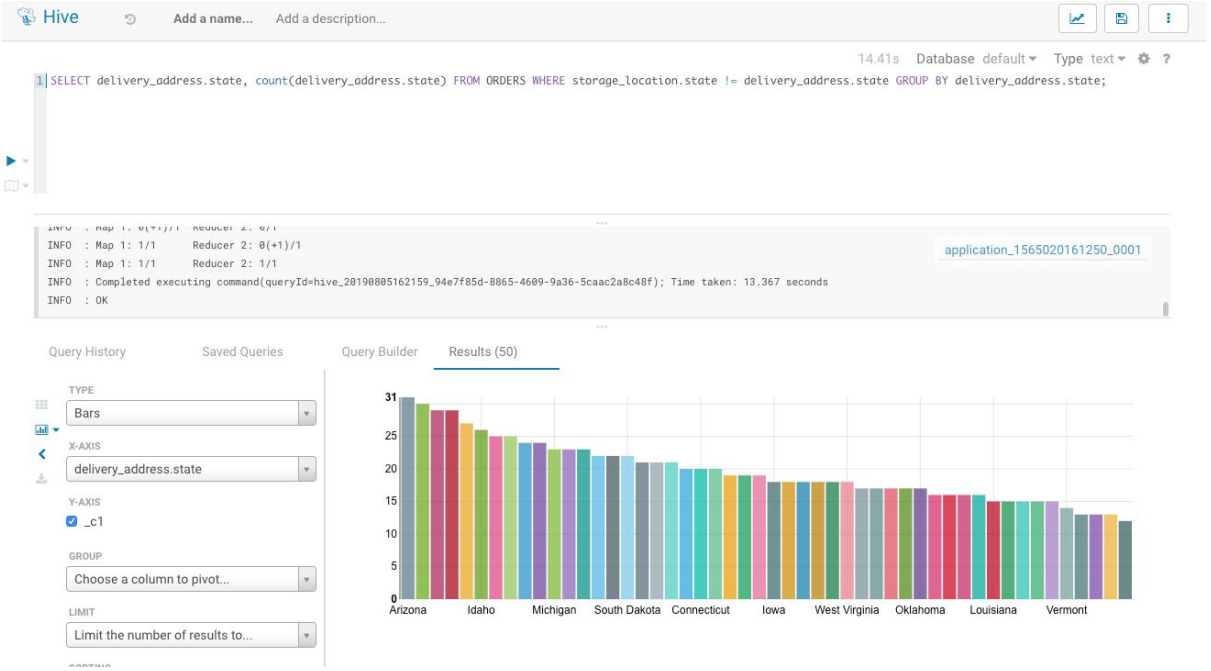


Imagen 25. Query5.

Referencias

- White, T. 2014. Hadoop The Definitive Guide: Storage and Analysis at Internet Scale. Sebastopol, CA: O'Reilly Media Inc.

- Capriolo, E., Wampler, D., & Rutherglen, J. 2012. Programming Hive: Data warehouse and query language for Hadoop. Sebastopol, CA: O'Reilly Media Inc.