Práctica 2 - HBase

Nombre	Email
Carlota Monedero Herranz	**carlotamoh@alu.comillas.edu**
Jorge Ayuso Martínez	**jorgeayusomartinez@alu.comillas.edu**

En primer lugar, creamos el namespace donde procederemos a guardar nuestras tablas. Hemos escogido como nombre una combinación de nuestros usuarios (`mbd4` y `mbd25`):

```
create_namespace 'airdata_425'
```

Podemos comprobar que se ha creado correctamente:

```
list_namespace 'airdata_425'
```

Ejercicio 1

Detalle de aeropuertos: dado un código de aeropuerto se deben obtener todos sus atributos, con posibilidad de proyectar atributos específicos.

En este caso, dado que la consulta más eficiente es mediante row-key, hemos decidido utilizar el código del aeropuerto como clave, aprovechando que se trata de un campo único para cada uno de los aeropuertos, además de que se encuentra ya establecido como una columna dentro del fichero `/tmp/nosql/airData/airports.csv`.

Aunque el resto de atributos podrían organizarse en múltiples column families diferentes, hemos decidido emplear una única column family, guardando los diferentes campos de información como qualifiers diferentes dentro de la misma. Esto se debe a que el número de qualifiers diferentes que vayan a estar presentes dentro de los datos es conocido y constante, además de que probablemente ello nos facilite la carga de datos, puesto que se trata de columnas dentro del csv que podemos cargar directamente.

Para crear la tabla utilizamos el siguiente comando:

```
create 'airdata_425:airports', 'I'
```

Comprobamos que se ha creado correctamente:

```
list_namespace_tables 'airdata_425'
```

Una vez creada la tabla procedemos a realizar la carga de datos desde `/tmp/nosql/airData/airports.csv`. Para ello, utilizaremos el código de Python provisto en el script `src/1-airports.py`. Con el fin de poder ejecutar de manera limpia los scripts desde el nodo `Edge01`, así como evitar conflictos de dependencias, hemos creado un entorno en `conda` mediante el comando:

```
/opt/miniconda3/bin/conda create -n env_name python=3.8
```

Para poder ejecutar este entorno es necesario realizar un paso previo de inicialización de `conda` mediante el siguiente comando:

```
/opt/miniconda3/bin/conda
```

Tras cerrar la terminal para poder aplicar los cambios, volvemos a conectarnos al `Edge01` y observamos que en la consola aparece activado el entorno `base` de `conda`. Podemos ahora activar nuestro entorno personal mediante el comando que se muestra a continuación:

```
conda activate env_name
```

Una vez hecho esto, procedemos a instalar la librería de `happybase`:

```
pip install happybase
```

Adicionalmente, instalamos la librería `pandas`, que será necesaria para ejecutar el código:

```
pip install pandas
```

Finalmente, ejecutamos el script `src/1-airports.py` para poder realizar la carga de datos a la tabla recién creada. En nuestro primer intento de realizar la carga de datos, vimos que la librería `pandas` no estaba siendo capaz de parsear los datos correctamente a la hora de cargarlos en un DataFrame, por lo que dedujimos que había algo dentro del csv original que estaba impidiendo el correcto parseo de los registros. Para poder detectar las líneas mal formateadas dentro de los ficheros, empleamos el siguiente comando en python:

```
df = pd.read_csv(filepath, sep=',', error_bad_lines=False)
```

Donde `filepath` refiere a la ruta del fichero `airports.csv` a cargar. Ejecutando este comando pudimos recuperar el número correspondiente a los registros que parecen estar dando error. Podemos inspeccionar algunos de ellos mediante el comando:

```
awk -F ',' 'NR == 303' /tmp/nosql/airData/airports.csv
```

```
(base) [mbd4@edge01 ~]$ awk -F ',' 'NR == 303' /tmp/nosql/airData/airports.csv
35A,Union County, Troy Shelton ,Union,SC,USA,34.68680111,-81.64121167
```

Podemos comprobar que el problema con estas filas es que existe algún campo, como `airport` o `city` que posee un valor separado por comas. Por ello, cuando `pandas` trata de pasar el contenido a un DataFrame, se encuentra con que en estos registros hay 8 campos a cargar en vez de 7 (al interpretar que en ese campo hay dos campos diferentes en vez de un sólo por la presencia de dicha coma), lo cual causa errores en estos registros. Como el número de aeropuertos es limitado y no son datos que vayan a ser modificados con frecuencia, se propone realizar el cambio de estos registros a mano para poder introducirlos en la tabla. Sin embargo, es algo a tener en cuenta en caso de que se vayan a seguir introduciendo datos de cara a futuro.

Una vez corregido este fallo, procedemos a ejecutar el script de nuevo.

```
(mbd4) [mbd4@edge01 HBase_NoSQL]$ time python 1-airports.py

real    0m2.325s
user    0m1.109s
sys     0m3.636s
```

Tras ello, es necesario acceder a la shell de HBase (`hbase shell`) y ejecutar un scan de la tabla:

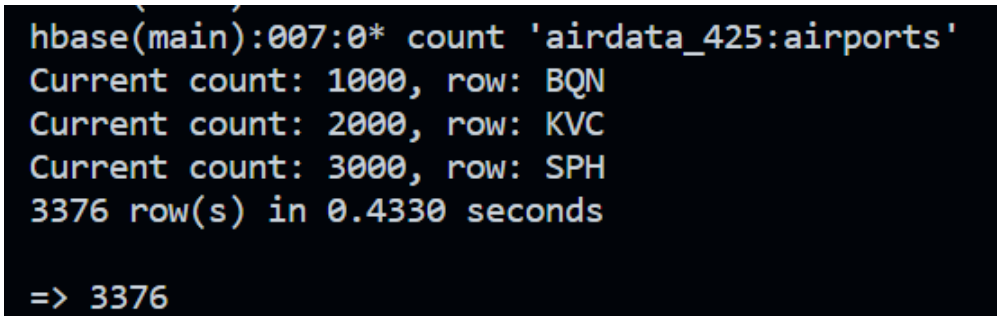
```
scan 'airdata_425:airports', {LIMIT => 3}
```

```
hbase(main):013:0* scan 'airdata_425:airports', {LIMIT => 3}
ROW                                     COLUMN+CELL
 00M                                    column=I:airport, timestamp=1679225435729, value=Thigpen
 00M                                    column=I:city, timestamp=1679225435729, value=Bay Springs
 00M                                    column=I:country, timestamp=1679225435729, value=USA
 00M                                    column=I:lat, timestamp=1679225435729, value=31.95376472
 00M                                    column=I:long, timestamp=1679225435729, value=-89.23450472
 00M                                    column=I:state, timestamp=1679225435729, value=MS
 00R                                    column=I:airport, timestamp=1679225435732, value=Livingston Municipal
 00R                                    column=I:city, timestamp=1679225435732, value=Livingston
 00R                                    column=I:country, timestamp=1679225435732, value=USA
 00R                                    column=I:lat, timestamp=1679225435732, value=30.68586111
 00R                                    column=I:long, timestamp=1679225435732, value=-95.01792778
 00R                                    column=I:state, timestamp=1679225435732, value=TX
 00V                                    column=I:airport, timestamp=1679225435733, value=Meadow Lake
 00V                                    column=I:city, timestamp=1679225435733, value=Colorado Springs
 00V                                    column=I:country, timestamp=1679225435733, value=USA
 00V                                    column=I:lat, timestamp=1679225435733, value=38.94574889
 00V                                    column=I:long, timestamp=1679225435733, value=-104.5698933
 00V                                    column=I:state, timestamp=1679225435733, value=CO
3 row(s) in 0.0320 seconds
```

Con esto podemos comprobar que la estructura de datos que hemos especificado en la carga se mantiene en la tabla. Adicionalmente, para asegurarnos de que todos los registros han sido cargados, ejecutamos

un `count`:

```
count 'airdata_425:airports'
```



Podemos ver que el número de filas (3376) coincide con el número de líneas que obtenemos si hacemos un `wc -l` sobre el fichero de `/tmp/nosql/airData/airports.csv` (descontando el header del fichero), lo que nos indica que todos los datos se han cargado en la tabla correctamente.

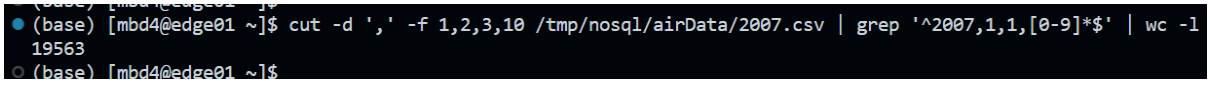
Ejercicio 2

Consulta de vuelos por mes o día: Posibilidad de obtener los vuelos de un día o mes específico (YYYYMMDD o YYYYMM).

Como detalle de información se deben obtener siempre todos los siguientes datos: hora de salida, hora de llegada, número de vuelo, origen, destino, número de aeronave y distancia recorrida.

En este caso, queremos realizar una consulta de vuelos por días o mes específico, por lo que nuestra primera aproximación al problema fue idear una clave que sea combinación de las columnas que indican esta información en los `csv` de carga, cargando los datos correspondientes a los vuelos como qualifiers dentro de una misma column family. Sin embargo, se puede comprobar que para un mismo día puede haber varios vuelos diferentes, por lo que no podemos utilizar únicamente esta información como row-key, ya que la información de vuelos sucesivos se iría cargando sobre un mismo registro y no de manera separada. Podemos comprobar que esto es cierto con el siguiente comando en la terminal del `Edge01`:

```
cut -d ',' -f 1,2,3,10 /tmp/nosql/airData/2007.csv | grep '^2007,1,1,[0-9]*$' | wc -l
```



Este comando nos permite inspeccionar el fichero de vuelos de `/tmp/nosql/airData/2007.csv` seleccionando únicamente las columnas 1, 2, 3 y 10 (correspondientes al año, mes, día del mes y número de vuelo, respectivamente). En este caso, queremos buscar el número de entradas en el año 2007 para el mes de enero y el día 1. Para ello, empleamos comando `cut` de Linux, donde mediante el parámetro `-d` le especificamos el delimitador del `csv` (en este caso, comas), así como las columnas que queremos (mediante el parámetro `-f`). Sobre el output de este comando, realizamos una búsqueda `grep` para quedarnos con todas las entradas que correspondan al día 1 de enero de 2007, en cualquier número de vuelo. Finalmente, realizamos un `wc -l` para obtener el número de líneas del output del comando, resultando en un total de 19563. Esto nos confirma que no podemos utilizar esta aproximación para el diseño de nuestra carga de datos.

La segunda potencial aproximación para tratar resolver este problema es comprobar si en cada día el `id` del vuelo es un identificador único. Esto podría ayudarnos, ya que lo único que tendríamos que hacer es utilizar la información de la fecha como prefijo para la row-key y emplear el campo correspondiente al número de vuelo para hacer que el registro sea único. Para comprobar nuestra hipótesis, podemos utilizar el siguiente comando, que muestra aquellas combinaciones de año, mes, día y número de vuelo que tienen más de una coincidencia en el fichero `csv` de carga:

```
cut -d ',' -f 1,2,3,10 /tmp/nosql/airData/2007.csv | grep '^2007,1,1,[0-9]*$' | uniq -c | grep -
v '^ *1' | wc -l
```

```
(base) [mbd4@edge01 ~]$ cut -d ',' -f 1,2,3,10 /tmp/nosql/airData/2007.csv | grep '^2007,1,1,[0-9]*$' | uniq -c | grep -v '^ *1' | wc -l
1897
```

Donde el comando `grep -v '^ *1'` indica que se muestren aquellos registros que no empiecen por 1. Tras ejecutar el comando, obtenemos un total de 1897 registros, lo que nos indica que no podemos identificar un vuelo de forma unívoca haciendo uso únicamente del año, mes, día y número de vuelo.

Tomando un ejemplo concreto, podemos seleccionar específicamente un número de vuelo (en este caso se ha escogido el vuelo 2891, pero podría ser cualquier otro que tuviera más de una coincidencia en el fichero `csv` de carga):

```
cut -d ',' -f 1,2,3,10 /tmp/nosql/airData/2007.csv | grep '^2007,1,1,2891$' | wc -l
```

```
(base) [mbd4@edge01 ~]$ cut -d ',' -f 1,2,3,10 /tmp/nosql/airData/2007.csv | grep '^2007,1,1,2891$' | wc -l
3
(base) [mbd4@edge01 ~]$
```

Cuando ejecutamos este comando se puede observar que el número de entradas para ese vuelo en esa fecha específica es de 3, lo que indica que es posible que dicho vuelo tenga entradas repetidas. Para poder ver cuál es exactamente la diferencia entre dichas entradas, se necesita el resto de columnas del fichero para ese día y vuelo específico. El comando de filtrado a emplear es el siguiente:

```
egrep '^2007,1,1,[0-9,]*([A-Z]){2},2891' /tmp/nosql/airData/2007.csv
```

```
(base) [mbd4@edge01 ~]$ egrep '^2007,1,1,[0-9,]*([A-Z]){2},2891' /tmp/nosql/airData/2007.csv
2007,1,1,1,1232,1225,1341,1340,WN,2891,N351,69,75,54,1,7,SMF,ONT,389,4,11,0,,0,0,0,0,0,0
2007,1,1,1,1408,1405,1508,1500,WN,2891,N351,60,55,42,8,3,ONT,LAS,197,8,10,0,,0,0,0,0,0,0
(base) [mbd4@edge01 ~]$
```

Con este comando podemos encontrar todas las entradas correspondientes al vuelo 2891 en el día 1 de enero de 2007. Podemos comprobar que aparecen dos entradas diferentes, puesto que se trata de un vuelo que fue desde Sacramento (SMF) hasta Los Angeles (ONT) y, tras una escala, se desplazó desde Los Angeles hasta Las Vegas (LAS). La inspección más detallada de las horas de aterrizaje de la primera entrada (13:41) y la hora de despegue de la segunda entrada (14:08) parecen corroborar nuestra hipótesis.

Con esta información en mente, llegamos a la conclusión de que el diseño de una nueva clave que permita identificar registros de forma unívoca necesitaría incluir la siguiente información:

- Año del vuelo (`YYYY`)
- Mes del vuelo (`MM`)
- Día del mes (`DD`)
- Compañía (`UniqueCarrier`)
- Número de vuelo (`FlightNum`)
- Origen (`Origin`)
- Destino (`Dest`)

Sin embargo, el diseño de esta clave es sumamente ineficiente, puesto que requiere de almacenar en disco una gran cantidad de información para cada una de las row-keys, la cual, de manera adicional, estaría almacenada dentro de las propias columnas de cada uno de los registros. Es por ello que esta aproximación se vuelve poco eficiente. Es más eficiente un planteamiento donde todos los vuelos de una misma fecha se almacenan en una misma rowKey, de forma que buscando únicamente por la fecha se obtengan todos los vuelos realizados en dicha fecha, sin necesidad de tener que buscar por campos adicionales, como ocurriría en el caso que se acaba de describir.

Por ello, se propone una nueva aproximación al problema: utilizar la información de año, mes y día de vuelo como row-key pero cargando cada registro como una nuevo qualifier dentro de una misma column family, donde el nombre del qualifier será un identificador único de registro y el valor asociado al mismo será un JSON con la información correspondiente. De este modo, seguimos garantizando que el campo de consulta de información sea la row-key, obteniendo así una mayor eficiencia en las consultas, además de ahorrar espacio en disco al evitar la duplicidad de información entre row-key y las columnas de la tabla. Adicionalmente, se aprovecha la naturaleza columnar que caracteriza a HBase.

A la hora de realizar el diseño del qualifier, se propone una combinación de los siguientes campos:

- Compañía (`UniqueCarrier`)
- Número de vuelo (`FlightNum`)

- Origen (`Origin`)
- Destino (`Dest`)

De forma que se garantiza que no se almacenan en el mismo qualifier (y en consecuencia como versiones diferentes) dos vuelos distintos. Por otra parte, el JSON a cargar como valor dentro de cada columna posee la siguiente información:

```
json = {
  "DepTime": DepTime,
  "ArrTime": ArrTime,
  "FlightNum": FlightNum,
  "Origin": Origin,
  "Dest": Dest,
  "TailNum": TailNum,
  "Distance": Distance
}
```

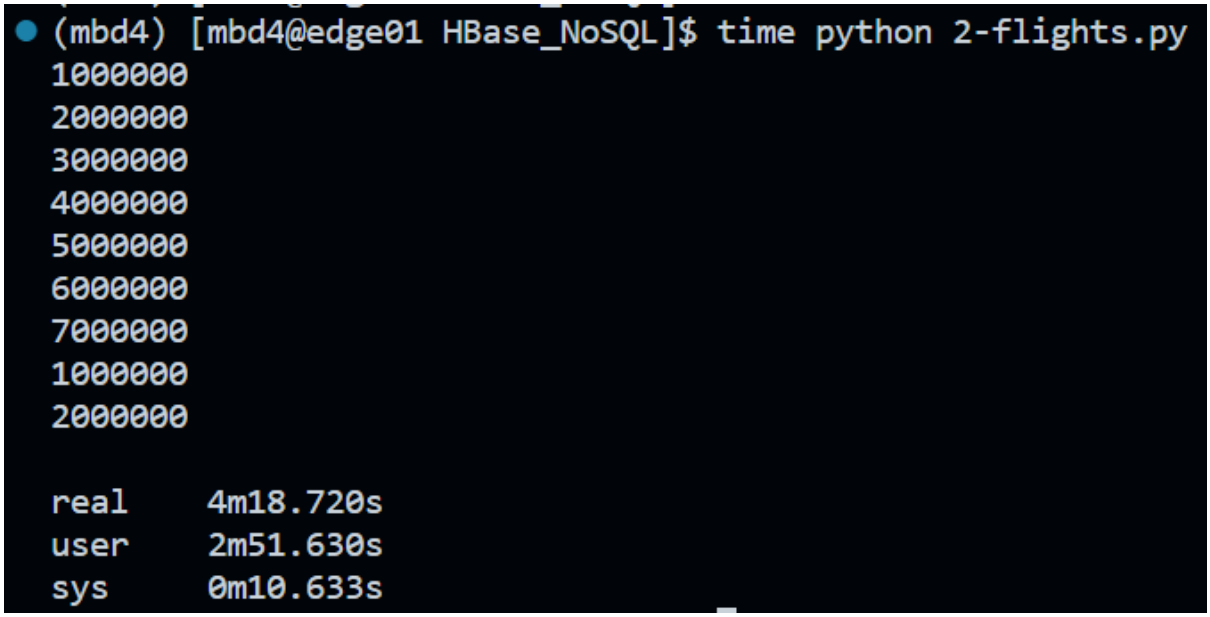
Se procede por tanto a crear la nueva tabla dentro del namespace previamente definido:

```
create 'airdata_425:flights', 'F'
```

Comprobamos que se ha creado correctamente:

```
list_namespace_tables 'airdata_425'
```

A continuación, se procede a ejecutar el script `src/2-flights.py`, de manera similar a como lo hicimos en el ejercicio 1 (activando el entorno de `conda` previo a realizar la ejecución del script desde el nodo `Edge01`). A diferencia del caso anterior, donde (debido al número reducido de registros a cargar), realizamos una carga de registros uno a uno, en este caso hemos optado por realizar una carga en `batch`. Esto permite reducir significativamente el tiempo de carga de datos, lo cual es de suma importancia en este caso, donde hay un total de casi 10 millones de registros a cargar. Además, la ventaja adicional que presenta esta aproximación es que es altamente escalable y nos permite seguir insertando un gran número de registros en poco tiempo, y sin necesidad de modificar el código.

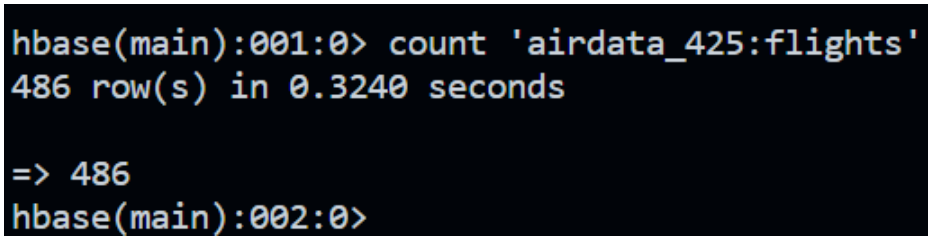


Tras ejecutar el script, se realizan las comprobaciones oportunas para ver que los datos se han insertado correctamente en nuestra tabla:

```
# Mostrar primera fila de la tabla
scan 'airdata_425:flights', { LIMIT => 1 }
```

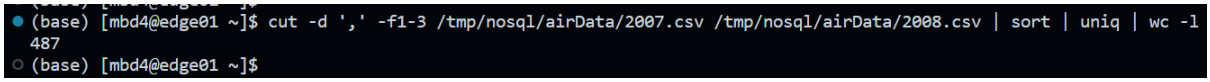
Con esto podemos comprobar que los registros dentro de la tabla tienen la estructura anteriormente definida, utilizando la fecha del vuelo como row-key, un identificador único de registro como qualifier y un JSON con la información correspondiente a dicho registro como valor.

```
# Conteo del número de regitros que contiene la tabla
count 'airdata_425:flights'
```



Podemos comprobar que hay un total de 486 registros. Podemos comprobar si esto se corresponde con el número de días de los ficheros `/tmp/nosql/airData/2007.csv` y `/tmp/nosql/airData/2008.csv` mediante los siguientes comandos:

```
cut -d ',' -f1-3 /tmp/nosql/airData/2007.csv /tmp/nosql/airData/2008.csv | sort | uniq | wc -l
```



Obtenemos un total de 487 líneas, lo cual (tras restar la línea correspondiente al header de los ficheros) se corresponde con el número de registros cargados en la tabla.

Comprobamos de igual forma que se pueden llevar a cabo las búsquedas tal y como se especifica en el ejercicio:

```
# Mostrar el registro correspondiente al 2 de enero de 2007
scan 'airdata_425:flights', { ROWPREFIXFILTER => '20070102'}
```

```
# Mostrar los registros correspondientes a abril de 2008
scan 'airdata_425:flights', { ROWPREFIXFILTER => '200804'}
```

Podemos comprobar que ambos comandos devuelven los resultados esperados. En el primer caso, obtenemos los registros correspondientes a una única row-key, mientras que en el segundo obtenemos todos los registros correspondientes a un año y mes específicos.

Ejercicio 3

Consulta de rutas: Dado un par origen-destino, se debe obtener el siguiente detalle:

- **1. Duración promedio del vuelo para esta ruta.**
- **2. Nombre de la aerolínea que ofrece dicha ruta, media de retraso en la hora de salida, media de retraso en la hora de llegada.**
- **3. Los resultados deben devolverse ordenados de forma que las aerolíneas que vuelan con mas frecuencia la ruta, aparezcan en primer lugar.**
- **4. Como extra sería interesante obtener el modelo de aeronave más usado por la aerolínea para dicha ruta.**

En este caso, el ejercicio no nos pide cargar datos que se encuentren directamente reflejados dentro de los ficheros, sino que es necesario realizar una serie de consultas de agregación de los diferentes campos para poder obtener un DataFrame final que cargar dentro de la tabla de HBase. Estos agregados se calculan mediante diferentes funciones de agregación que pueden encontrarse reflejadas dentro del script de carga de datos (`src/3-routes.py`).

Para esta tabla decidimos realizar la carga de datos utilizando el par origen-destino como row-key, con el fin de facilitar la búsqueda dentro de la tabla. De esta forma, la búsqueda se puede realizar de manera eficiente por este campo y podemos cargar la información de cada uno de los registros unívocos dentro de un qualifier perteneciente a una única column family. De forma análoga al ejercicio anterior, la unicidad del registro no se garantiza a nivel de row-key, sino a nivel de qualifier. Inicialmente pensamos en realizar el diseño de este qualifier empleando la aerolínea como identificador único, de modo que

introdujéramos la información en forma de JSON. Adicionalmente, pensamos emplear una column family diferente para introducir la información correspondiente a la duración media del vuelo, común para todas las aerolíneas que vuelan en la misma ruta, de forma que sólo tuviéramos que introducir esta información una vez por cada una de las rutas, optimizando así el espacio.

Sin embargo, esta aproximación presenta dos problemas principales. En primer lugar, no nos permite aplicar la ordenación pedida por el ejercicio, puesto que, a pesar de que hicimos esfuerzos para realizar una carga ordenada de los datos en función del número de vuelos de cada una de las aerolíneas, esta ordenación no se veía reflejada a la hora de devolver el resultado. El motivo de esto es que la ordenación de HBase siempre es de forma lexicográfica, sin tener en cuenta el timestamp de carga de los datos (primero se ordena por rowKey, después por column family y finalmente por qualifier).

En segundo lugar, el uso de dos column families diferentes acaba por resultar ineficiente e innecesario, puesto que la segunda column family (que contiene el dato global para la ruta), no es un dato suficientemente pesado como para justificar el uso de dos familias de columnas diferentes. Es por ello que decidimos utilizar un código numérico para realizar la identificación del registro que correlacionase con el ranking de las aerolíneas en función del número de vuelos realizados en esa ruta en particular. Inicialmente, planteamos el uso de un único dígito para la ordenación (1,2…10,11…n), pero pronto comprobamos que HBase no tiene en cuenta la ordenación numérica de los dígitos, sino su ordenación lexicográfica. Esto implica que, en el caso de que hubiera más de 9 aerolíneas, el orden lógico de los resultados se volvería confuso, ya que la segunda aerolínea en ser devuelta en la búsqueda no sería aquella que tuviera el número 2, sino el número 10. Por tanto, tuvimos que buscar una aproximación alternativa, basada en el uso de un código de 5 cifras, donde las cifras vacías fueran sustituidas por un 0. De este modo, la ordenación tendría el siguiente aspecto: 00001, 00002 … 00010, 00011 …, solucionando así el problema anteriormente planteado. Aunque el número pueda parecer exagerado, de cara a futuro consideramos que emplear un número amplio de cifras para la ordenación puede permitirnos escalar de manera sencilla la carga de datos, puesto que el límite de compañías aéreas que podríamos cubrir en una misma ruta es virtualmente inalcanzable (105).

Por otra parte, nos dimos cuenta de que para poder cargar los datos de la duración media de vuelo dentro de esta misma column family deberíamos incluir su clave dentro de la lógica de ordenación anteriormente planteada, no solo por consistencia de formato sino para asegurar que no haya problemas en la ordenación de registros de las aerolíneas. Por ello, decidimos cargar este dato bajo el qualifier `99999`, de forma que siempre aparezca como el último campo devuelto al buscar cada una de las rutas.

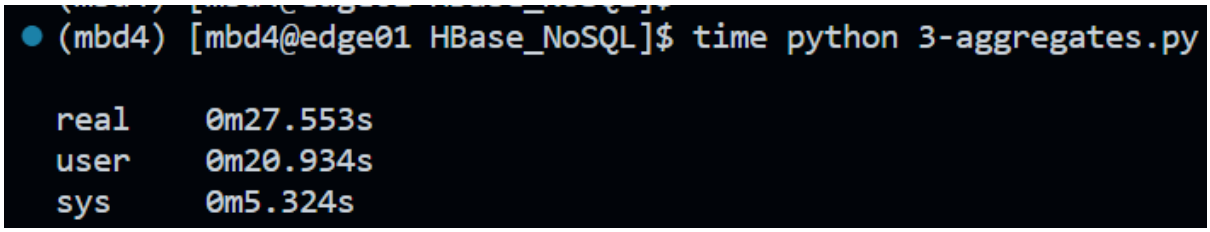
Igual que en los ejercicios anteriores, creamos la tabla mediante el comando:

```
create 'airdata_425:routes', 'C'
```

Comprobamos que se ha creado correctamente:

```
list_namespace_tables 'airdata_425'
```

A continuación, procedemos a realizar la carga de datos mediante la ejecución del script `src/3-routes.py`, donde se ejecutan las operaciones necesarias para el cálculo de los agregados y se cargan los mismos dentro de la tabla anteriormente creada en HBase.



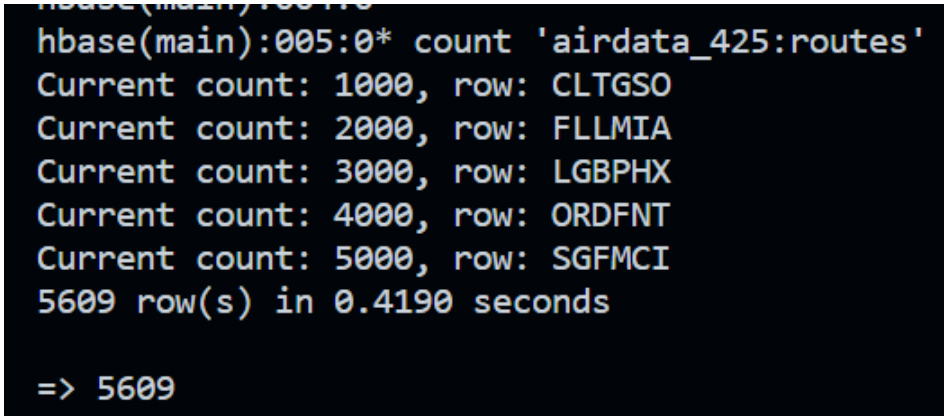
Realizamos finalmente las comprobaciones oportunas para ver que la estructura es conforme a la que buscamos:

```
# Mostrar primeras filas de la tabla
scan 'airdata_425:routes', { LIMIT => 10 }
```

En este caso nos interesa comprobar un número más amplio de registros que en los casos anteriores, para asegurarnos de que en todos los casos la ordenación es correcta, lo cual podemos comprobar visualmente de manera sencilla.

Igualmente, realizamos un conteo del número de registros en la tabla:

```
# Conteo del número de regitros que contiene la tabla
count 'airdata_425:routes'
```



Podemos observar que obtenemos un total de 5609 registros, que deberían corresponderse con el número de pares origen-destino dentro de los ficheros csv cargados. Para comprobarlo, ejecutamos el comando:

```
cut -d ',' -f17-18 /tmp/nosql/airData/2007.csv /tmp/nosql/airData/2008.csv | sort | uniq | wc -l
```

```
(base) [mbd4@edge01 ~]$ cut -d ',' -f17-18 /tmp/nosql/airData/2007.csv /tmp/nosql/airData/2008.csv | sort | uniq | wc -l
5610
(base) [mbd4@edge01 ~]$
```

Con esto, obtenemos un total de 5610 líneas, a las cuales debemos restarle el header, dando un total de 5609, lo que corresponde con el número de registros cargados en la tabla.

Finalmente, comprobamos que podemos recuperar la información correspondiente a una ruta de manera adecuada:

```
# Mostrar la información correspondiente a una ruta
get 'airdata_425:routes', 'ONTLAS'
```

```
hbase(main):008:0* get 'airdata_425:routes', 'ONTLAS'
COLUMN                                  CELL
 C:00001                                timestamp=1679227962854, value={'UniqueCarrier': 'WN', 'AvgDelayDep': 9.939108061749572, 'AvgDelayArr': 7.70205920205920
                                        25, 'Nroutes': 4676, 'TailNum': 'N612SW', 'NroutesAirplane': 21.0}
 C:00002                                timestamp=1679227962854, value={'UniqueCarrier': 'US', 'AvgDelayDep': 6.747572815533981, 'AvgDelayArr': 8.64805825242718
                                        4, 'Nroutes': 414, 'TailNum': 'N625AW', 'NroutesAirplane': 7.0}
 C:00003                                timestamp=1679227962855, value={'UniqueCarrier': 'YV', 'AvgDelayDep': 5.072847682119205, 'AvgDelayArr': 7.56622516556291
                                        4, 'Nroutes': 305, 'TailNum': 'N7305V', 'NroutesAirplane': 26.0}
 C:99999                                timestamp=1679227962855, value={'Route_avg_duration': 41.173363095238095}
4 row(s) in 0.0130 seconds
```

Comprobamos que la información puede recuperarse y que la ordenación de los registros devueltos se corresponde con el número de rutas realizadas por cada aerolínea, siendo el último registro la duración media de vuelo.