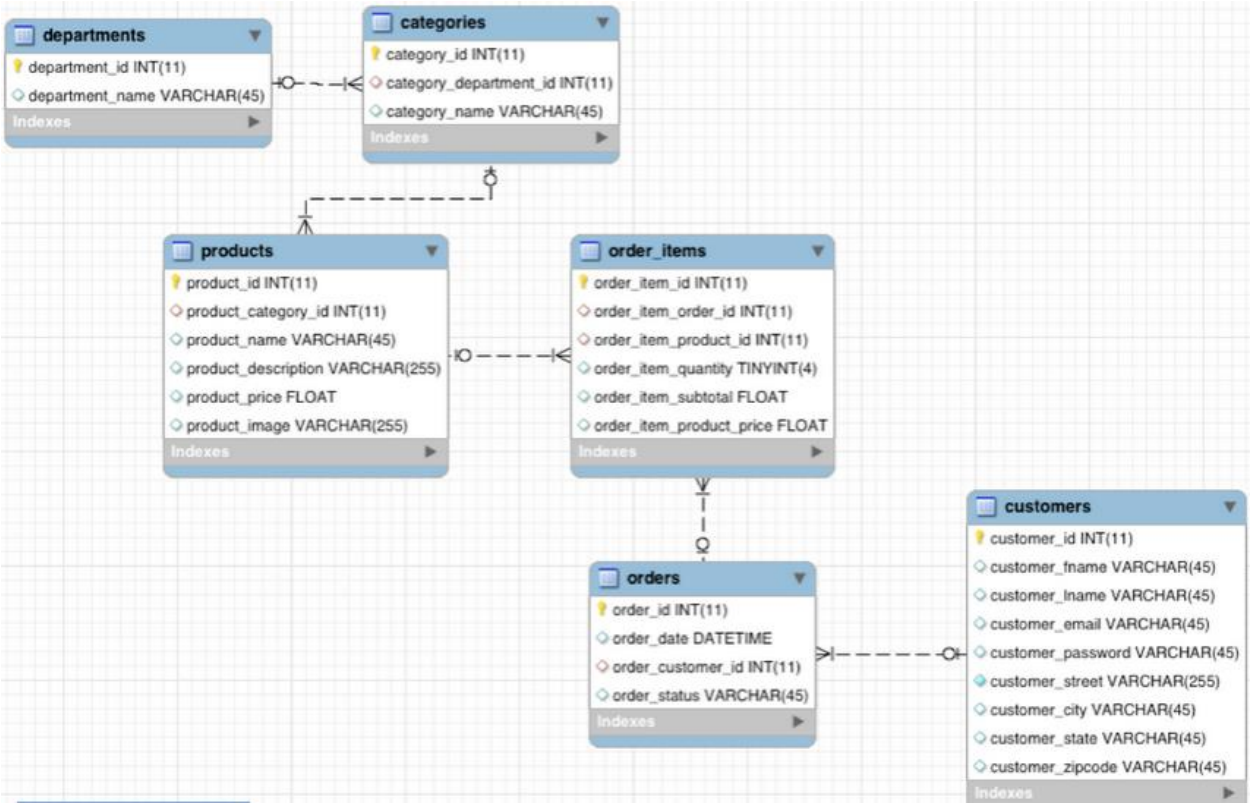
RETAIL CASE STUDY - DATACO

Ingest Structured Data:

In this scenario, DataCo's business question is: What products do our customers like to buy? To answer this question, the first thought might be to look at the transaction data, which should indicate what customers actually do buy and like to buy, right?

This is probably something you can do in your regular RDBMS environment, but a benefit with Cloudera's platform is that you can do it at greater scale at lower cost, on the same system that you may also use for many other types of analysis.

What this exercise demonstrates is how to do exactly the same thing you already know how to do, but in CDH. Seamless integration is important when evaluating any new infrastructure. Hence, it's important to be able to do what you normally do, and not break any regular BI reports or workloads over the dataset you plan to migrate.



To analyze the transaction data in the new platform, we need to ingest it into the Hadoop Distributed File System (HDFS). We need to find a tool that easily transfers structured data from a RDBMS to HDFS, while preserving structure. That enables us to query the data, but not interfere with or break any regular workload on it.

Apache Sqoop, which is part of CDH, is that tool. The nice thing about Sqoop is that we can automatically load our relational data from MySQL into HDFS, while preserving the structure.

With a few additional configuration parameters, we can take this one step further and load this relational data directly into a form ready to be queried by Impala (the open source analytic query engine included with CDH). Given that we may want to leverage the power of the Apache Avro file format for other workloads on the cluster (as Avro is a Hadoop optimized file format), we will take a few extra steps to load this data into Impala using the Avro file format, so it is readily available for Impala as well as other workloads.

```
[cloudera@quickstart ~]$ sqoop import-all-tables \
    -m 1 \
    --connect jdbc:mysql://quickstart:3306/retail_db \
    --username=retail_dba \
    --password=cloudera \
    --compression-codec=snappy \
    --as-avrodatafile \
    --warehouse-dir=/user/hive/warehouse
```

This command may take a while to complete, but it is doing a lot. It is launching MapReduce jobs to export the data from our MySQL database, and put those export files in Avro format in HDFS. It is also creating the Avro schema, so that we can easily load our Hive tables for use in Impala later.

When this command is complete, confirm that your Avro data files exist in HDFS.

```
[cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse
```

Will show a folder for each of the tables.

```
[cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse/categories/
```

Will show the files that live inside of the categories folder.

Sqoop should also have created schema files for this data in your home directory.

Avro schema files:

```
[cloudera@quickstart ~]$ ls -l *.avsc
```

Should show .avsc files for the six tables that were in our retail_db.

Note that the schema and the data are stored in separate files. The schema is only applied when the data is queried, a technique called 'schema-on-read'. This gives you the flexibility to query the data with SQL while it's still in a format usable by other systems as well. Whereas a traditional database requires the schema to be defined

before entering any data, we have already imported a lot of data and will only now specify how it's structure should be interpreted.

Apache Hive will need the schema files too, so let's copy them into HDFS where Hive can easily access them.

```
[cloudera@quickstart ~]$ sudo -u hdfs hadoop fs -mkdir /user/examples

[cloudera@quickstart ~]$ sudo -u hdfs hadoop fs -chmod +rw /user/examples

[cloudera@quickstart ~]$ hadoop fs -copyFromLocal ~/*.avsc /user/examples/
```

Now that we have the data, we can prepare it to be queried. We're going to do this in the next section using Impala, but you may notice we imported this data into Hive's directories. Hive and Impala both read their data from files in HDFS, and they even share metadata about the tables. The difference is that Hive executes queries by compiling them to MapReduce jobs. As you will see later, this means it can be more flexible, but is much slower. Impala is an MPP query engine that reads the data directly from the file system itself. This allows it to execute queries fast enough for interactive analysis and exploration.

QUERYING THE DATA FROM CDH USING IMPALA:

```
CREATE EXTERNAL TABLE categories STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/categories'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_categories.avsc');


CREATE EXTERNAL TABLE customers STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/customers'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_customers.avsc');


CREATE EXTERNAL TABLE departments STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/departments'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_departments.avsc');


CREATE EXTERNAL TABLE orders STORED AS AVRO
```

```
LOCATION 'hdfs:///user/hive/warehouse/orders'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_orders.avsc');


CREATE EXTERNAL TABLE order_items STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/order_items'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_order_items.avsc');


CREATE EXTERNAL TABLE products STORED AS AVRO

LOCATION 'hdfs:///user/hive/warehouse/products'

TBLPROPERTIES
('avro.schema.url'='hdfs://quickstart/user/examples/sqoop_import_products.avsc');


show tables;
```

Now that your transaction data is readily available for structured queries in CDH, it's time to address DataCo's business question. Copy and paste or type in the following standard SQL example queries for calculating total revenue per product and showing the top 10 revenue generating products:

```
-- Most popular product categories

select c.category_name, count(order_item_quantity) as count

from order_items oi

inner join products p on oi.order_item_product_id = p.product_id

inner join categories c on c.category_id = p.product_category_id

group by c.category_name

order by count desc

limit 10;


-- top 10 revenue generating products

select p.product_id, p.product_name, r.revenue
```

```
from products p inner join
(select oi.order_item_product_id, sum(cast(oi.order_item_subtotal as float)) as revenue
from order_items oi inner join orders o
on oi.order_item_order_id = o.order_id
where o.order_status <> 'CANCELED'
and o.order_status <> 'SUSPECTED_FRAUD'
group by order_item_product_id) r
on p.product_id = r.order_item_product_id
order by r.revenue desc
limit 10;
```

Correlate Structured Data with Unstructured Data:

Since you are a pretty smart data person, you realize another interesting business question would be: are the most viewed products also the most sold? (or for other scenarios, the most searched for, the most chatted about…). Since Hadoop can store unstructured and semi-structured data alongside structured data without remodelling an entire database, you can just as well ingest, store and process web log events. Let's find out what site visitors have actually viewed the most.

For this, you need the web clickstream data. The most common way to ingest web clickstream is to use Flume. Flume is a scalable real-time ingest framework that allows you to route, filter, aggregate, and do “mini-operations” on data on its way in to the scalable processing platform.

Later, you can explore a Flume configuration example, to use for real-time ingest and transformation of our sample web clickstream data. However, for the sake of tutorial-time, in this step, we will not have the patience to wait for three days of data to be ingested. Instead, we prepared a web clickstream data set (just pretend you fast forwarded three days) that you can bulk upload into HDFS directly.

Bulk Upload Data

For convenience, we have loaded a sample (about 20MM lines) of one month's worth of access log data into /opt/examples/log_data/access.log.2.

Let's move this data from the local filesystem, into HDFS.

Go back to your terminal and execute the following commands as root from your Master Node.

```
[cloudera@quickstart ~]$ sudo -u hdfs hadoop fs -mkdir /user/hive/warehouse/original_access_logs

[cloudera@quickstart ~]$ sudo -u hdfs hadoop fs -copyFromLocal /opt/examples/log_files/access.log.2 /user/hive/warehouse/original_access_logs

[cloudera@quickstart ~]$ hadoop fs -ls /user/hive/warehouse/original_access_logs
```

Now you can build a table in Hive and query the data via Impala and Hue. You'll build this table in 2 steps. First, you'll take advantage of Hive's flexible SerDes (serializers / deserializers) to parse the logs into individual fields using a regular expression. Second, you'll transfer the data from this intermediate table to one that does not require any special SerDe. Once the data is in this table, you can query it much faster and more interactively using Impala.

We'll query Hive using a command-line JDBC client for Hive called Beeline. You can invoke it from the terminal with the following:

```
[cloudera@quickstart ~]$ beeline -u jdbc:hive2://quickstart:10000/default -n admin -d org.apache.hive.jdbc.HiveDriver
```

Open the Hive Query Editor app in Impala, and run the following queries:

```
0: jdbc:hive2://quickstart:10000/default> CREATE EXTERNAL TABLE intermediate_access_logs (
 ip STRING,
 date STRING,
 method STRING,
 url STRING,
 http_version STRING,
 code1 STRING,
 code2 STRING,
 dash STRING,
 user_agent STRING)
 ROW FORMAT SERDE 'org.apache.hadoop.hive.contrib.serde2.RegexSerDe'
 WITH SERDEPROPERTIES (
```

```
'input.regex' = '([^ ]*) - - \\[([^\\]]*)\\] "([^\ ]*) ([^\ ]*) ([^\ ]*)" (\\d*) (\\d*) "([^"]*)" "([^"]*)"',
'output.format.string' = "%1$s %2$s %3$s %4$s %5$s %6$s %7$s %8$s %9$s"
)
LOCATION '/user/hive/warehouse/original_access_logs';

0: jdbc:hive2://quickstart:10000/default> CREATE EXTERNAL TABLE tokenized_access_logs (
ip STRING,
date STRING,
method STRING,
url STRING,
http_version STRING,
code1 STRING,
code2 STRING,
dash STRING,
user_agent STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
LOCATION '/user/hive/warehouse/tokenized_access_logs';

0: jdbc:hive2://quickstart:10000/default> ADD JAR /usr/lib/hive/lib/hive-contrib.jar;
0: jdbc:hive2://quickstart:10000/default> INSERT OVERWRITE TABLE tokenized_access_logs SELECT * FROM intermediate_access_logs;
0: jdbc:hive2://quickstart:10000/default> DROP TABLE intermediate_access_logs;
0: jdbc:hive2://quickstart:10000/default> !quit
```

To save time during queries, Impala does not poll constantly for metadata changes. So when you create new tables while Impala is running you must tell it to refresh the metadata. Go Hue and open the Impala Query Editor app, and enter the following command:

```
invalidate metadata;
```

Now, if you run 'show tables' or refresh the table list in the left-hand column, you should see the two new external tables in the default database. Paste the following query into the Query Editor

```
select count(*),url from tokenized_access_logs

where url like '%\/product\/%'

group by url order by count(*) desc;
```

By introspecting the results you quickly realize that this list contains many of the products on the most sold list from previous results, but there is one product that did not show up in the previous result. There is one product that seems to be viewed a lot, but never purchased. Why?

Well, in our example with DataCo, once these odd findings are presented to your manager, it is immediately escalated. Eventually, someone figures out that on that view page, where most visitors stopped, the sales path of the product had a typo in the price for the item. Once the typo was fixed, and a correct price was displayed, the sales for that SKU started to rapidly increase.