Hands-On Exercise: Interactive Analysis with Impala

In this exercise you will examine abandoned cart data using the tables created in the previous exercise. You will use Impala to quickly determine how much lost revenue these abandoned carts represent and use several "what if" scenarios to determine whether we should offer free shipping to encourage customers to complete their purchases.

IMPORTANT: This exercise builds on previous ones. If you were unable to complete any previous exercise or think you may have made a mistake, run the following command to prepare for this exercise before continuing:

```
$ ~/scripts/analyst/catchup.sh
```

Step #1: Start the Impala Shell and Refresh the Cache

1. Change to the directory for this hands-on exercise:

```
$ cd $ADIR/exercises/interactive
```

2. First, start the Impala shell:

```
$ impala-shell
```

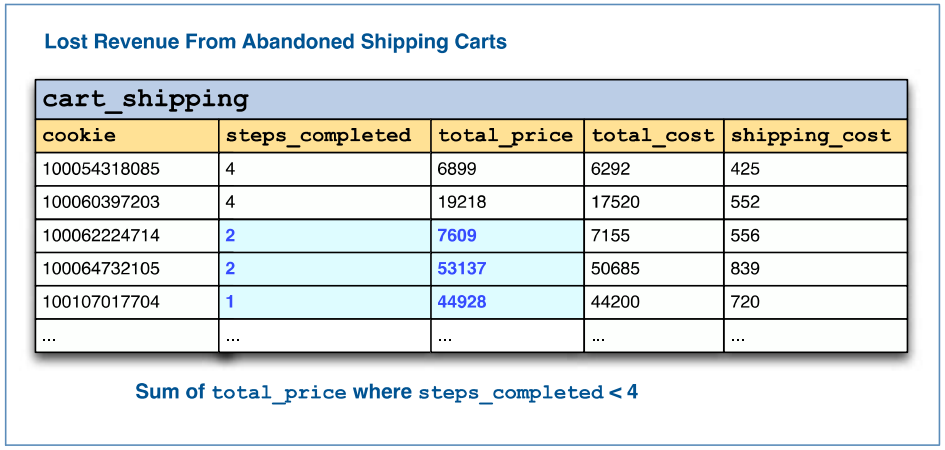
3. Since you created tables and modified data in Hive, Impala's cache of the metastore is outdated. You must refresh it before continuing by entering the following command in the Impala shell:

```
REFRESH;
```

Step #2: Calculate Lost Revenue

1. First, you'll calculate how much revenue the abandoned carts represent. Remember, there are four steps in the checkout process, so only records in the `cart_shipping` table with a `steps_completed` value of four represent a completed purchase:

```
SELECT SUM(total_price) AS lost_revenue
    FROM cart_shipping
    WHERE steps_completed < 4;
```



You should see that abandoned carts mean that Dualcore is potentially losing out on more than $2 million in revenue! Clearly it's worth the effort to do further analysis.

Note: The `total_price`, `total_cost`, and `shipping_cost` columns in the `cart_shipping` table contain the number of cents as integers. Be sure to divide results containing monetary amounts by 100 to get dollars and cents.

2. The number returned by the previous query is revenue, but what counts is profit. We calculate gross profit by subtracting the cost from the price. Write and execute a query similar to the one above, but which reports the total lost profit from abandoned carts. If you need a hint on how to write this query, you can check the `sample_solution/abandoned_checkout_profit.sql` file.

    - After running your query, you should see that we are potentially losing $111,058.90 in profit due to customers not completing the checkout process.

3. How does this compare to the amount of profit we receive from customers who do complete the checkout process? Modify your previous query to consider only those records where `steps_completed = 4`, and then execute it in the Impala shell. Check `sample_solution/completed_checkout_profit.sql` if you need a hint.

    - The result should show that we earn a total of $177,932.93 on completed orders, so abandoned carts represent a substantial proportion of additional profits.

4. The previous two queries told us the *total* profit for abandoned and completed orders, but these aren't directly comparable because there were different numbers of each. It might be the case that one is much more profitable than the other on a per-order basis. Write and execute a query that will calculate the *average* profit based on the number of steps completed during the checkout process. If you need help writing this query, check the `sample_solution/checkout_profit_by_step.sql` file.

    - You should observe that carts abandoned after step two represent an even higher average profit per order than completed orders.

Step #3: Calculate Cost/Profit for a Free Shipping Offer

You have observed that most carts – and the most *profitable* carts – are abandoned at the point where we display the shipping cost to the customer. You will now run some queries to determine whether offering free shipping, on at least some orders, would actually bring in more revenue assuming this offer prompted more customers to finish the checkout process.

1. Run the following query to compare the average shipping cost for orders abandoned after the second step versus completed orders:

```
SELECT steps_completed, AVG(shipping_cost) AS ship_cost
   FROM cart_shipping
   WHERE steps_completed = 2 OR steps_completed = 4
   GROUP BY steps_completed;
```

Average Shipping Cost for Carts Abandoned After Steps 2 and 4

cart_shipping

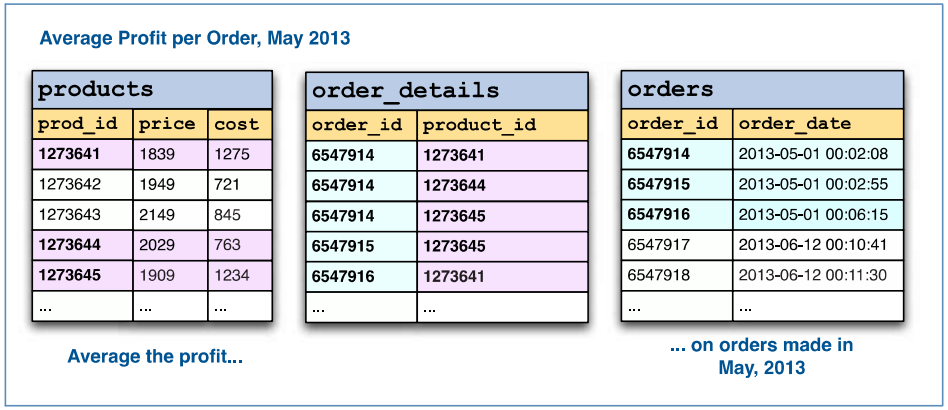
cookie	steps_completed	total_price	total_cost	shipping_cost
100054318085	4	6899	6292	425
100060397203	4	19218	17520	552
100062224714	2	7609	7155	556
100064732105	2	53137	50685	839
100107017704	1	44928	44200	720
...	...	...	...	...

Average of `shipping_cost` where `steps_completed` = 2 or 4

- You will see that the shipping cost of abandoned orders was almost 10% higher than for completed purchases. Offering free shipping, at least for some orders, might actually bring in more money than passing on the cost and risking abandoned orders.

2. Run the following query to determine the average profit per order over the entire month for the data you are analyzing in the log file. This will help you to determine whether we could absorb the cost of offering free shipping:

```
SELECT AVG(price - cost) AS profit
  FROM products p
  JOIN order_details d
    ON (d.prod_id = p.prod_id)
  JOIN orders o
    ON (d.order_id = o.order_id)
 WHERE YEAR(order_date) = 2013
       AND MONTH(order_date) = 05;
```



- You should see that the average profit for all orders during May was $7.80. An earlier query you ran showed that the average shipping cost was $8.83 for completed orders and $9.66 for abandoned orders, so clearly we would lose money by offering free shipping on all orders. However, it might still be worthwhile to offer free shipping on orders over a certain amount.

3. Run the following query, which is a slightly revised version of the previous one, to determine whether offering free shipping only on orders of $10 or more would be a good idea:

```
SELECT AVG(price - cost) AS profit
  FROM products p
  JOIN order_details d
    ON (d.prod_id = p.prod_id)
  JOIN orders o
    ON (d.order_id = o.order_id)
  WHERE YEAR(order_date) = 2013
        AND MONTH(order_date) = 05
        AND PRICE >= 1000;
```

   - You should see that our average profit on orders of $10 or more was $9.09, so absorbing the cost of shipping would leave very little profit.

4. Repeat the previous query, modifying it slightly each time to find the average profit on orders of at least $50, $100, and $500.

   - You should see that there is a huge spike in the amount of profit for orders of $500 or more (we make $111.05 on average for these orders).

5. How much does shipping cost on average for orders totaling $500 or more? Write and run a query to find out (`sample_solution/avg_shipping_cost_50000.sql` contains the solution, in case you need a hint).

   - You should see that the average shipping cost is $12.28, which happens to be about 11% of the profit we bring in on those orders.

6. Since we won't know in advance who will abandon their cart, we would have to absorb the $12.28 average cost on *all* orders of at least $500. Would the extra money we might bring in from abandoned carts offset the added cost of free shipping for customers who would have completed their purchases anyway? Run the following query to see the total profit on completed purchases:

```
SELECT SUM(total_price - total_cost) AS total_profit
   FROM cart_shipping
   WHERE total_price >= 50000
     AND steps_completed = 4;
```

   - After running this query, you should see that the total profit for completed orders is $107,582.97.

7. Now, run the following query to find the potential profit, after subtracting shipping costs, if all customers completed the checkout process:

```
SELECT gross_profit - total_shipping_cost AS potential_profit
   FROM (SELECT
           SUM(total_price - total_cost) AS gross_profit,
           SUM(shipping_cost) AS total_shipping_cost
         FROM cart_shipping
         WHERE total_price >= 50000) large_orders;
```

   Since the result of $120,355.26 is greater than the current profit of $107,582.97 we currently earn from completed orders, it appears that we could earn nearly $13,000 more by offering free shipping for all orders of at least $500.

   Congratulations! Your hard work analyzing a variety of data with Hadoop's tools has helped make Dualcore more profitable than ever.

This is the end of the Exercise

Data Model Reference

Tables Imported from MySQL

The following depicts the structure of the MySQL tables imported into HDFS using Sqoop. The primary key column from the database, if any, is denoted by bold text:

customers: 201,375 records (imported to `/dualcore/customers`)

Index	Field	Description	Example
0	**cust_id**	Customer ID	1846532
1	fname	First name	Sam
2	lname	Last name	Jones
3	address	Address of residence	456 Clue Road
4	city	City	Silicon Sands
5	state	State	CA
6	zipcode	Postal code	94306

employees: 61,712 records (imported to `/dualcore/employees` and later used as an external table in Hive)

Index	Field	Description	Example
0	**emp_id**	Employee ID	BR5331404
1	fname	First name	Betty
2	lname	Last name	Richardson
3	address	Address of residence	123 Shady Lane
4	city	City	Anytown
5	state	State	CA
6	zipcode	Postal Code	90210
7	job_title	Employee's job title	Vice President
8	email	e-mail address	br5331404@example.com
9	active	Is actively employed?	Y
10	salary	Annual pay (in dollars)	136900

orders: 1,662,951 records (imported to `/dualcore/orders`)

Index	Field	Description	Example
0	**order_id**	Order ID	3213254
1	cust_id	Customer ID	1846532
2	order_date	Date/time of order	2013-05-31 16:59:34

order_details: 3,333,244 records (imported to `/dualcore/order_details`)

Index	Field	Description	Example
0	order_id	Order ID	3213254
1	prod_id	Product ID	1754836

products: 1,114 records (imported to `/dualcore/products`)

Index	Field	Description	Example
0	**prod_id**	Product ID	1273641
1	brand	Brand name	Foocorp
2	name	Name of product	4-port USB Hub
3	price	Retail sales price, in cents	1999
4	cost	Wholesale cost, in cents	1463
5	shipping_wt	Shipping weight (in pounds)	1

suppliers: 66 records (imported to `/dualcore/suppliers`)

Index	Field	Description	Example
0	**supp_id**	Supplier ID	1000
1	fname	First name	ACME Inc.
2	lname	Last name	Sally Jones
3	address	Address of office	123 Oak Street
4	city	City	New Athens
5	state	State	IL
6	zipcode	Postal code	62264
7	phone	Office phone number	(618) 555-5914

Hive Tables

The following is a record count for Hive tables that are created or queried during the hands-on exercises. Use the `DESCRIBE` *`tablename`* command in Hive to see the table structure.

Table Name	Record Count
cart_items	33,812
cart_orders	12,955
cart_shipping	12,955
cart_zipcodes	12,955
checkout_sessions	12,955
customers	201,375
employees	61,712
order_details	3,333,244
orders	1,662,951
products	1,114
ratings	21,997
web_logs	412,860

Other Data Added to HDFS

The following describes the structure of other important data sets added to HDFS.

Combined Ad Campaign Data: (788,952 records total), stored in two directories:

- `/dualcore/ad_data1` (438,389 records)
- `/dualcore/ad_data2` (350,563 records).

Index	Field	Description	Example
0	campaign_id	Uniquely identifies our ad	`A3`
1	date	Date of ad display	`05/23/2013`
2	time	Time of ad display	`15:39:26`
3	keyword	Keyword that triggered ad	`tablet`
4	display_site	Domain where ad shown	`news.example.com`
5	placement	Location of ad on Web page	`INLINE`
6	was_clicked	Whether ad was clicked	`1`
7	cpc	Cost per click, in cents	`106`

access.log: 412,860 records (uploaded to `/dualcore/access.log`)
This file is used to populate the `web_logs` table in Hive. Note that the RFC 931 and Username fields are seldom populated in log files for modern public Web sites and are ignored in our RegexSerDe.

Index	Field / Description	Example
0	IP address	`192.168.1.15`
1	RFC 931 (Ident)	`-`
2	Username	`-`
3	Date/Time	`[22/May/2013:15:01:46 -0800]`
4	Request	`"GET /foo?bar=1 HTTP/1.1"`
5	Status code	`200`
6	Bytes transferred	`762`
7	Referer	`"http://dualcore.com/"`
8	User agent (browser)	`"Mozilla/4.0 [en] (WinNT; I)"`
9	Cookie (session ID)	`"SESSION=8763723145"`

Regular Expression Reference

The following is a brief tutorial intended for the convenience of students who don't have experience using regular expressions or may need a refresher. A more complete reference can be found in the documentation for Java's Pattern class:

`http://tiny.cloudera.com/dae9a`

Introduction to Regular Expressions

Regular expressions are used for pattern matching. There are two kinds of patterns in regular expressions: literals and metacharacters. Literal values are used to match precise patterns while metacharacters have special meaning; for example, a dot will match any single character. Here's the complete list of metacharacters, followed by explanations of those that are commonly used:

`< ( [ { \ ^ - = $ ! | ] } ) ? * + . >`

Literal characters are any characters not listed as a metacharacter. They're matched exactly, but if you want to match a metacharacter, you must escape it with a backslash. Since a backslash is itself a metacharacter, it must also be escaped with a backslash. For example, you would use the pattern `\\.` to match a literal dot.

Regular expressions support patterns much more flexible than simply using a dot to match any character. The following explains how to use *character classes* to restrict which characters are matched.

Character Classes

`[057]`	Matches any single digit that is either 0, 5, or 7
`[0-9]`	Matches any single digit between 0 and 9
`[3-6]`	Matches any single digit between 3 and 6
`[a-z]`	Matches any single lowercase letter
`[C-F]`	Matches any single uppercase letter between C and F

For example, the pattern `[C-F][3-6]` would match the string `D3` or `F5` but would fail to match `G3` or `C7`.

There are also some built-in character classes that are shortcuts for common sets of characters.

Predefined Character Classes	
`\\d`	Matches any single digit
`\\w`	Matches any word character (letters of any case, plus digits or underscore)
`\\s`	Matches any whitespace character (space, tab, newline, etc.)

For example, the pattern `\\d\\d\\d\\w` would match the string `314d` or `934X` but would fail to match `93X` or `Z871`.

Sometimes it's easier to choose what you don't want to match instead of what you do want to match. These three can be negated by using an uppercase letter instead.

Negated Predefined Character Classes	
`\\D`	Matches any single non-digit character
`\\W`	Matches any non-word character
`\\S`	Matches any non-whitespace character

For example, the pattern `\\D\\D\\W` would match the string `ZX#` or `@ P` but would fail to match `93X` or `36_`.

The metacharacters shown above match each exactly one character. You can specify them multiple times to match more than one character, but regular expressions support the use of quantifiers to eliminate this repetition.

Matching Quantifiers	
`{5}`	Preceding character may occur exactly five times
`{0,6}`	Preceding character may occur between zero and six times
`?`	Preceding character is optional (may occur zero or one times)
`+`	Preceding character may occur one or more times
`*`	Preceding character may occur zero or more times

By default, quantifiers try to match as many characters as possible. If you used the pattern `ore.+a` on the string `Dualcore has a store in Florida`, you might be surprised to learn that it matches `ore has a store in Florida` rather than `ore ha` or `ore in Florida` as you might have expected. This is because matches a "greedy" by default. Adding a question mark makes the quantifier match as few characters as possible instead, so the pattern `ore.+?a` on this string would match `ore ha`.

Finally, there are two special metacharacters that match zero characters. They are used to ensure that a string matches a pattern only when it occurs at the beginning or end of a string.

Boundary Matching Metacharacters	
`^`	Matches only at the beginning of a string
`$`	Matches only at the ending of a string

NOTE: When used inside square brackets (which denote a character class), the `^` character is interepreted differently. In that context, it negates the match. Therefore, specifying the pattern `[^0-9]` is equivalent to using the predefined character class `\\d` described earlier.