SQL Project on Analyzing the CryptoPunks Dataset

Chandrima Khastagir

May 20, 2024

CryptoPunks is a collection of nonfungible tokens (NFTs) on the Ethereum cryptocurrency blockchain. With the increasing advent of digital platforms and cryptocurrency in our portfolios, a better understanding of such online transactions becomes indispensable. In this project, we use SQL to analyze sales record data for a number of NFTs undertaken on the CryptoPunks platform. We begin with a short description of the columns, before moving onto seeking specific questions about the dataset. The large volume and intricate information style of this database shows the crucial benefit of using SQL to derive valuable data-based insights.

The CryptoPunks dataset has information about NFT sales described through 8 different columns:

- ❖ buyer_address
- ❖ eth_price (price in Ethereum)
- ❖ usd_price (price in USD)
- ❖ seller_address
- ❖ event_date (date of the sale)
- ❖ token_id
- ❖ transaction_hash
- ❖ name

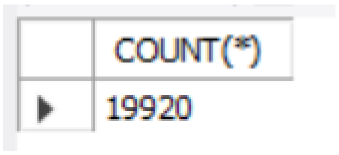
We will now understand the dataset better through a series of targeted questions. Let us start with the basics, by figuring out the volume of available sales information.

1. **How many sales occurred during this time period?**

The dataset describes one sale in each row. Thus, to answer this question, we start with looking for the number of rows in the dataset, which can be done by:

```
SELECT COUNT(*) FROM cryptopunks
```

The above code has the output:



	COUNT(*)
▶	19920

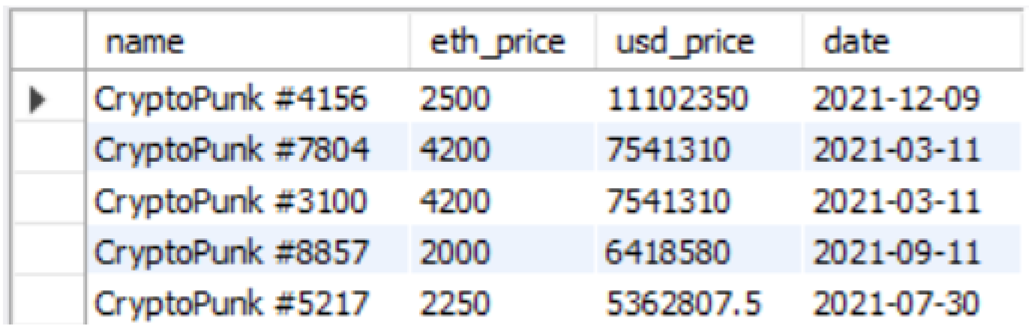
meaning that there are 19920 sales recorded in the dataset.

2. **Return the top 5 most expensive transactions (by USD price) for this data set.Return the name, ETH price, and USD price, as well as the date.**

Let us first understand which rows would be featured. Clearly, we need to sort the rows by USD price in the decreasing order and choose the top 5 among them. In SQL, this is done through ORDER BY and LIMIT commands. Now choosing the desired columns, we end up with the query:

```
SELECT name, eth_price, usd_price, event_date AS date
FROM cryptopunks
ORDER BY usd_price DESC
LIMIT 5
```

This query gave the output:



	name	eth_price	usd_price	date
▶	CryptoPunk #4156	2500	11102350	2021-12-09
	CryptoPunk #7804	4200	7541310	2021-03-11
	CryptoPunk #3100	4200	7541310	2021-03-11
	CryptoPunk #8857	2000	6418580	2021-09-11
	CryptoPunk #5217	2250	5362807.5	2021-07-30

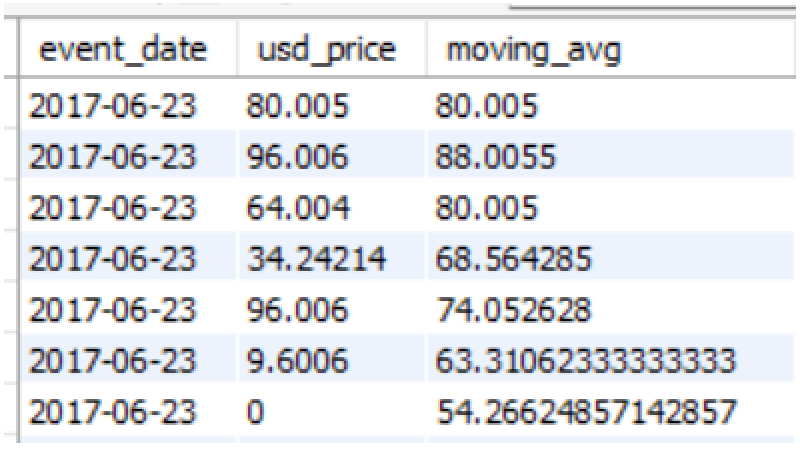
returning exactly the rows and columns we were seeking.

3. **Return a table with a row for each transaction with an event column, a USD price column, and a moving average of USD price that averages the last 50 transactions.**

To determine the moving average, we will have to average the USD price over the last 50 transactions ordered by event date. This requires a special syntax asking SQL to determine the rows between the two specified endpoints: start at the 49th preceding row, and end at the current row. Once this is settled, the rest of the query is straightforward.

```
SELECT event_date, usd_price, AVG(usd_price) OVER (ORDER BY event_date ASC
ROWS BETWEEN 49 PRECEDING AND CURRENT ROW) AS moving_avg
FROM cryptopunks
```

A partial output is as follows:



event_date	usd_price	moving_avg
2017-06-23	80.005	80.005
2017-06-23	96.006	88.0055
2017-06-23	64.004	80.005
2017-06-23	34.24214	68.564285
2017-06-23	96.006	74.052628
2017-06-23	9.6006	63.3106233333333
2017-06-23	0	54.26624857142857

Clearly, the very first transaction had no preceding row and its value coincides with the moving average.

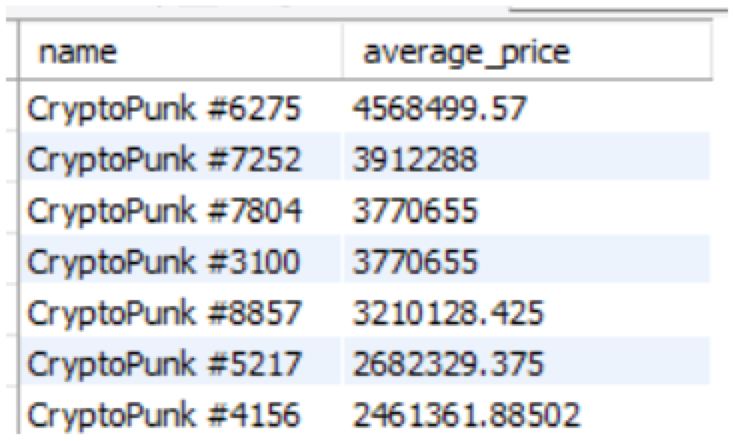
4. **Return all the NFT names and their average sale price in USD. Sort descending. Name the average column as average_price.**

This question can be broken into two simple parts. First we have to calculate the average sale price in USD by grouping over the name of the NFT: in SQL syntax, this points to a use of GROUP BY. Second, we need to sort the rows according to the average price in a descending order.

The following code block writes the required SQL code to carry out this operation.

```sql
SELECT name, AVG(usd_price) AS average_price
FROM cryptopunks
GROUP BY name
ORDER BY average_price DESC
```

A partial output is given by:



name	average_price
CryptoPunk #6275	4568499.57
CryptoPunk #7252	3912288
CryptoPunk #7804	3770655
CryptoPunk #3100	3770655
CryptoPunk #8857	3210128.425
CryptoPunk #5217	2682329.375
CryptoPunk #4156	2461361.88502

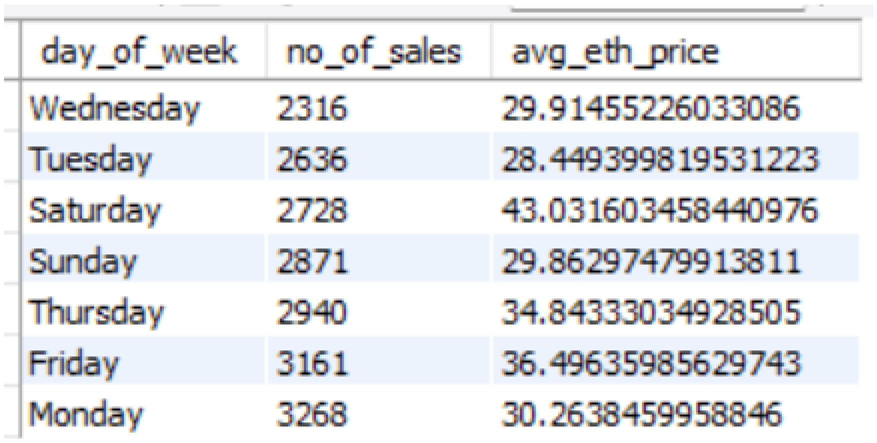
.

5. **Return each day of the week and the number of sales that occurred on that day of the week, as well as the average price in ETH. Order by the count of transactions in ascending order.**

Note that for each sale we have information on the event_date. Our first task will be to create a column which converts the event_date to the corresponding day of the week, which is achieved through the DAYNAME function in SQL. Once this is completed, we will use GROUP BY to count the number of sales, and the average price in ETH over all sales occurring for that day of the week. An ascending order sorting, via ORDER BY applied on the count column will then get the job done!

```sql
SELECT DAYNAME(event_date) AS day_of_week, COUNT(*) AS no_of_sales,
AVG(eth_price) AS avg_eth_price
FROM cryptopunks
GROUP BY day_of_week
ORDER BY no_of_sales ASC
```

We have the following output as a result of this code:



day_of_week	no_of_sales	avg_eth_price
Wednesday	2316	29.91455226033086
Tuesday	2636	28.449399819531223
Saturday	2728	43.03160345844097 6
Sunday	2871	29.86297479913811
Thursday	2940	34.8433303492850 5
Friday	3161	36.49635985629743
Monday	3268	30.263845995884 6

The most sales occur on Monday and Friday, and the lowest are in the middle of the week.

6. **Construct a column that describes each sale and is called summary. The sentence should include who sold the NFT name, who bought the NFT, who sold the NFT,the date, and what price it was sold for in USD rounded to the nearest thousandth.**

This question requires string handling in SQL. For each row, which represents a sale, we have the NFT name, the ids of the buyer and the seller coded by buyer_address and seller_address respectively. Rounding the USD price can be done through the ROUND function, with -3 to specify 3 places after the decimal. Finally, the CONCAT command can be used to write a sentence that concatenates the required information.

```
SELECT CONCAT(name, 'was sold for $', ROUND(usd_price, 3),
' to ', buyer_address, ' by ', seller_address, ' on ', event_date, '.') AS
summary
FROM cryptopunks
```

For example, the second row of the output reads:

```
'CryptoPunk #3874was sold for $207300.32 to
0xafa796c9de9b22b46f0dc1922fe017582c5e10b5 by
0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685 on 2022-01-14.'
```

and the fourth row reads:

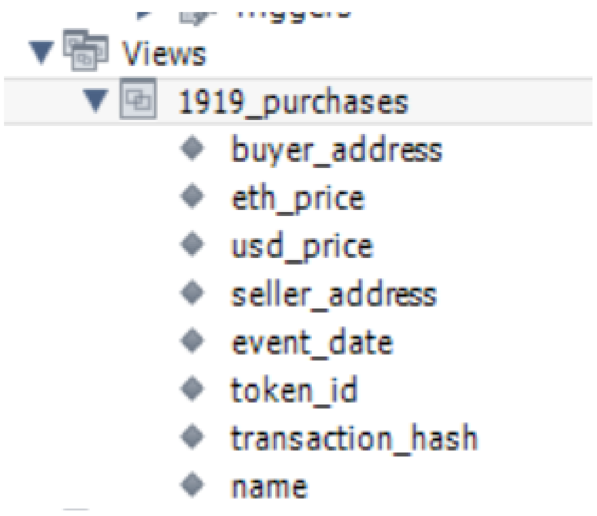
```
'CryptoPunk #5231was sold for $220266.72 to
0x3e8faf5b3a4ef575a329f8c976ff27f286ab2643 by
0x44a3ccddccae339d05200a8f4347f83a58847e52 on 2022-01-14.'
```

7. Create a view called "1919_purchases" and contains any sales where"0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685" was the buyer.

A view is a temporary table that is sometimes useful in local data visualization. In this case we use a view to isolate the sales a particular buyer was part of. The SQL syntax for this code is:

```
CREATE VIEW 1919_purchases AS
SELECT * FROM cryptopunks
WHERE buyer_address = 0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685
```

As expected, this creates a view with all columns and the corresponding rows in our schema:

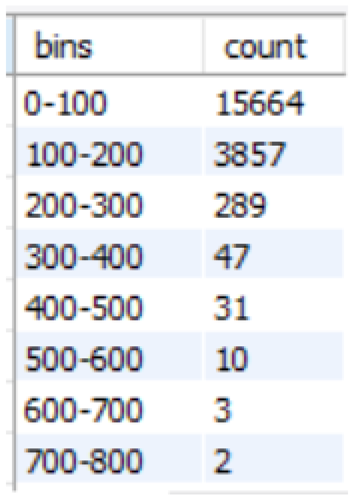


8. Create a histogram of ETH price ranges. Round to the nearest hundred value.

A quick look through the table using the MIN and MAX functions reveals that the eth_price value ranges from 0 to 4200. We will use binning into buckets of size 100. The SQL code below does exactly this, and returns the frequency counts for each bin.

```
SELECT CONCAT(ROUND(eth_price, -2),'-', LEAD(ROUND(eth_price, -2)) OVER
(ORDER BY ROUND(eth_price, -2))) AS bins,
COUNT(*) AS count
FROM cryptopunks
GROUP BY ROUND(eth_price, -2)
ORDER BY ROUND(eth_price, -2) ASC
```

A partial output is as follows:



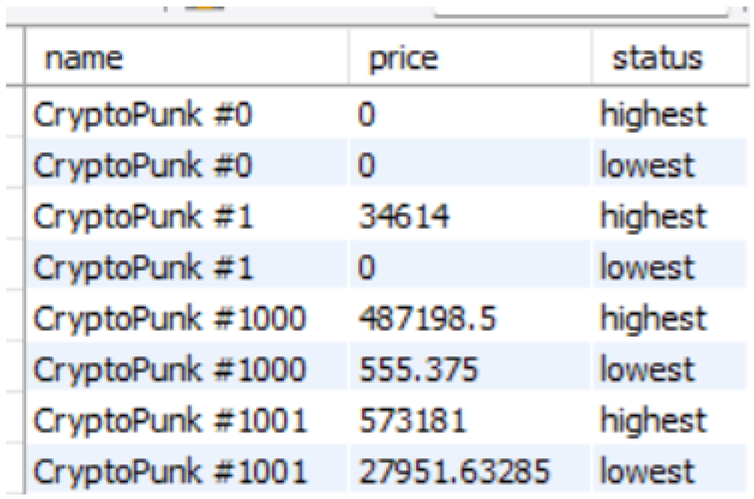
bins	count
0-100	15664
100-200	3857
200-300	289
300-400	47
400-500	31
500-600	10
600-700	3
700-800	2

9. **Return a unioned query that contains the highest price each NFT was bought for and a new column called status saying “highest” with a query that has the lowest price each NFT was bought for and the status column saying “lowest”. The table should have a name column, a price column called price, and a status column. Order the result set by the name of the NFT, and the status, in ascending order.**

We will first use two subqueries to derive the lowest price and the highest price corresponding to each NFT. Then, a Union operation over these two subqueries will give the desired result. As mentioned in the question, we have to then use ORDER BY to sort the rows according to the name and the status.

```
(SELECT name, MIN(usd_price) AS price, 'lowest' AS status
FROM cryptopunks
GROUP BY name)
UNION (SELECT name, MAX(usd_price) AS price, 'highest' AS status
FROM cryptopunks
GROUP BY name)
ORDER BY name, status ASC
```

A partial output is as follows:



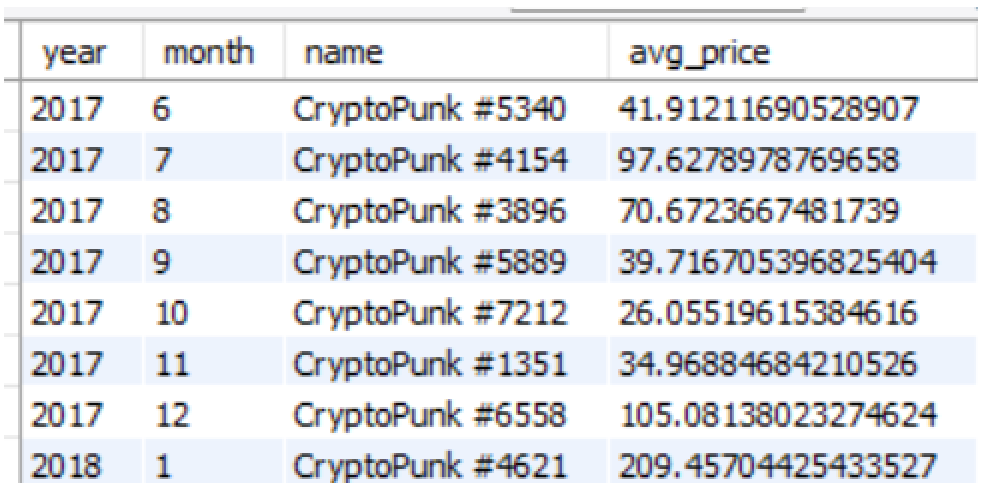
name	price	status
CryptoPunk #0	0	highest
CryptoPunk #0	0	lowest
CryptoPunk #1	34614	highest
CryptoPunk #1	0	lowest
CryptoPunk #1000	487198.5	highest
CryptoPunk #1000	555.375	lowest
CryptoPunk #1001	573181	highest
CryptoPunk #1001	27951.63285	lowest

10. What NFT sold the most each month / year combination? Also, what was the name and the price in USD? Order in chronological format.

Let us first break down the question into its components. First, we extract the month and year combination of every sale from the event_date column. For each such combination, we now need a ranking of the NFTs by total number sold in that period. This can be done through the RANK() function in SQL. The last step requires a filtering to pick only the top sold NFT, usually enforced through a WHERE condition. Since the WHERE condition operates on the newly created RANK column, we will pass the query inside to a subquery. The name and average USD price for each NFT can be derived through a GROUP BY condition. Taken together, the query becomes:

```
SELECT year, month, name, avg_price FROM (
SELECT YEAR(event_date) AS year,
         MONTH(event_date) AS month,
       name, AVG(usd_price) AS avg_price,
       RANK() OVER (PARTITION BY YEAR(event_date), MONTH(event_date) ORDER
BY COUNT(*) DESC) AS rank_most_sold
FROM cryptopunks
GROUP BY year, month) AS t
WHERE t.rank_most_sold = 1
ORDER BY year, month ASC
```

We get the following (truncated) output as a result of the above query:



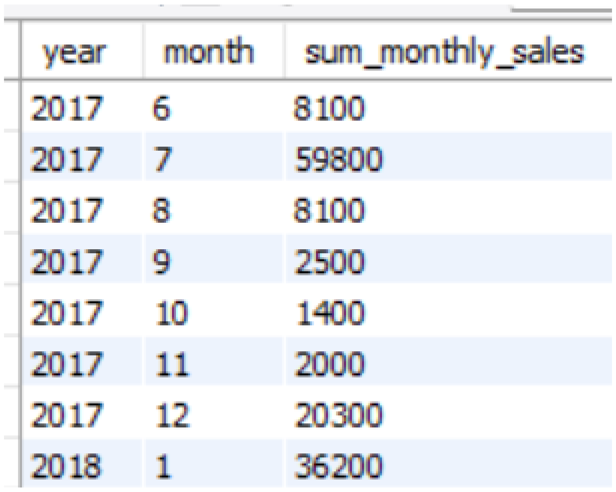
year	month	name	avg_price
2017	6	CryptoPunk #5340	41.91211690528907
2017	7	CryptoPunk #4154	97.6278978769658
2017	8	CryptoPunk #3896	70.6723667481739
2017	9	CryptoPunk #5889	39.716705396825404
2017	10	CryptoPunk #7212	26.05519615384616
2017	11	CryptoPunk #1351	34.96884684210526
2017	12	CryptoPunk #6558	105.08138023274624
2018	1	CryptoPunk #4621	209.45704425433527

11. **Return the total volume (sum of all sales), round to the nearest hundred on a monthly basis (month/year).**

Just as in the previous question, we extract the year and the month information from event_date, then sum over all available sale prices to get the monthly sale figure. This is done through a GROUP BY operation. The sales figure is rounded to the hundred's place using the ROUND function. We use the following query:

```
SELECT YEAR(event_date) AS year, MONTH(event_date) AS month,
ROUND(SUM(usd_price), -2) AS sum_monthly_sales
FROM cryptopunks
GROUP BY year, month
ORDER BY year, month ASC
```

A partial output looks as follows:



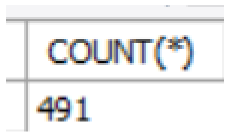
year	month	sum_monthly_sales
2017	6	8100
2017	7	59800
2017	8	8100
2017	9	2500
2017	10	1400
2017	11	2000
2017	12	20300
2018	1	36200

12. **Count how many transactions the wallet“0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685”had over this time period.**

We have to find all transactions a particular wallet has been a part of. Note that this wallet can participate either as a buyer or as a seller. Thus, we look for the count of all possible rows where either the buyer or the seller address matches with this wallet id. In SQL syntax this takes the form:

```
SELECT COUNT(*)
FROM cryptopunks
WHERE buyer_address = '0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685' OR
       seller_address = '0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685'
```

This query results in 491 sale records, as reflected below:



```
COUNT(*)
491
```

13. **Create an “estimated average value calculator” that has a representative price of the collection every day based off of these criteria: Exclude all daily outlier sales where the purchase price is below 10% of the daily average price. Take the daily average of remaining transactions. a) First create a query that will be used as a subquery. Select the event date, the USD price, and the average USD price for each day using a window function. Save it as a temporary table. b) Use the table you created in Part A to filter out rows where the USD price is below 10% of the daily average and return a new estimated value which is just the daily average of the filtered data.**

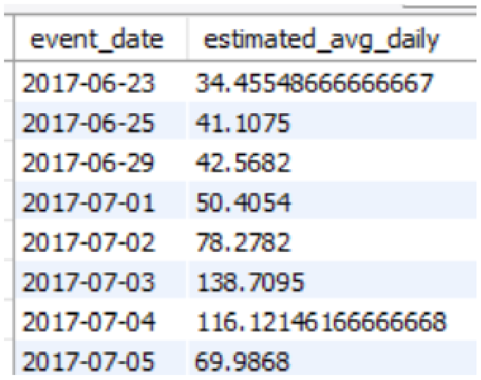
We will follow the execution plan given in the question. Let us first define a temporary table with the instructions in part a). This simply requires us to average the USD price using a Windows aggregate function over each event_date.

```
CREATE TEMPORARY TABLE avg_daily_price AS
SELECT event_date, usd_price, AVG(usd_price) OVER (PARTITION BY event_date)
AS avg_daily_price
FROM cryptopunks;
```

With the temporary table as defined above, let us now clip observations to be between 90% and 110% of the daily average. After this is done, we calculate the daily average over the observations remaining in the table.

```
SELECT event_date, AVG(usd_price) AS estimated_avg_daily
FROM avg_daily_price
WHERE usd_price > 0.9 * avg_daily_price AND usd_price < 1.1 *
avg_daily_price
GROUP BY event_date
```

When applied together, the above two queries produce an output of the form:



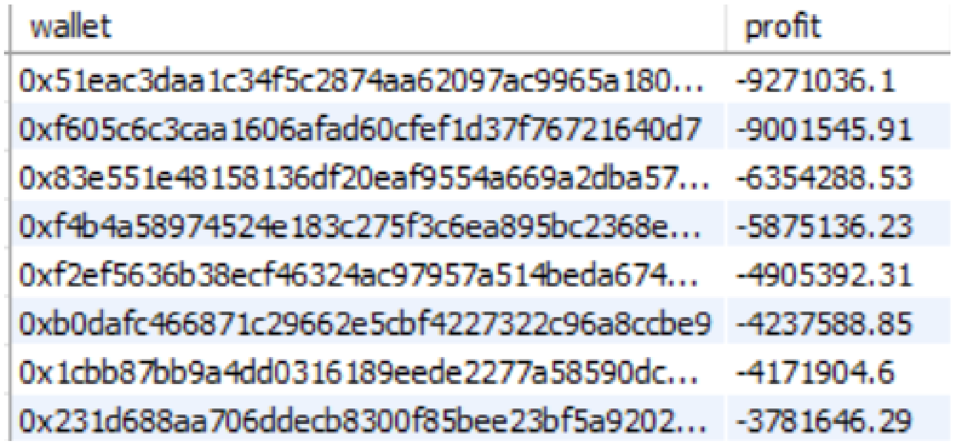
event_date	estimated_avg_daily
2017-06-23	34.45548666666667
2017-06-25	41.1075
2017-06-29	42.5682
2017-07-01	50.4054
2017-07-02	78.2782
2017-07-03	138.7095
2017-07-04	116.12146166666668
2017-07-05	69.9868

14. **Give a complete list ordered by wallet profitability (whether people have made or lost money).**

For each wallet, we first derive its total selling price and total buying price. We will then compute the profit for each wallet and order the rows according to profit. This leads to the SQL query:

```
SELECT t1.buyer_address AS wallet, ROUND(total_selling_price -
total_buying_price,2) AS profit
FROM
(SELECT buyer_address, SUM(usd_price) AS total_buying_price
FROM cryptopunks
GROUP BY buyer_address) t1
JOIN
(SELECT seller_address, SUM(usd_price) AS total_selling_price
FROM cryptopunks
GROUP BY seller_address) t2
ON t1.buyer_address = t2.seller_address
ORDER BY profit ASC
```

A partial output is shown below:



```
wallet                                          profit
0x51eac3daa1c34f5c2874aa62097ac9965a180...      -9271036.1
0xf605c6c3caa1606afad60cfef1d37f76721640d7      -9001545.91
0x83e551e48158136df20eaf9554a669a2dba57...      -6354288.53
0xf4b4a58974524e183c275f3c6ea895bc2368e...      -5875136.23
0xf2ef5636b38ecf46324ac97957a514beda674...      -4905392.31
0xb0dafc466871c29662e5cbf4227322c96a8ccbe9      -4237588.85
0x1cbb87bb9a4dd0316189eede2277a58590dc...       -4171904.6
0x231d688aa706ddecb8300f85bee23bf5a9202...      -3781646.29
```