1. How many sales occurred during this time period?

**Query: USE database\_name;**

**SELECT COUNT(\*) AS total\_sales**

**FROM pricedata;**

Explanation: To determine the total number of sales in the dataset, you use the SQL query SELECT COUNT(\*) AS total\_sales FROM pricedata; which counts all rows in the pricedata table. This query leverages the COUNT(\*) function to tally every sale entry, with the result labeled as total\_sales. Executing this query in MySQL Workbench provides a single number representing the total sales transactions recorded in the dataset.

1. 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.

**Query: SELECT name, eth\_price, usd\_price, event\_date**

**FROM pricedata**

**ORDER BY usd\_price DESC**

**LIMIT 5;**

Explanation: To find the top 5 most expensive transactions by USD price, this query selects the columns name, eth\_price, usd\_price, and event\_date from the pricedata table. It orders the results in descending order based on the usd\_price column, ensuring the highest USD prices appear first. By using the LIMIT 5 clause, the query restricts the output to only the top 5 transactions, giving a concise view of the most valuable sales. This allows us to quickly identify and analyze the highest value transactions within the dataset.

1. 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.

Query: SELECT

event\_date,

usd\_price,

AVG(usd\_price) OVER (ORDER BY event\_date ROWS BETWEEN 49 PRECEDING AND CURRENT ROW) AS moving\_avg\_usd\_price

FROM

pricedata;

Explanation: This query retrieves each transaction's event date and USD price, and calculates a moving average of the USD price over the last 50 transactions. The AVG function with a window clause orders the transactions by date and includes the current row and the previous 49 rows in the average calculation. This provides a trend analysis of the USD prices over time.

1. Top of Form

Bottom of Form

1. Return all the NFT names and their average sale price in USD. Sort descending. Name the average column as average\_price.

Query: SELECT

name,

AVG(usd\_price) AS average\_price

FROM

pricedata

GROUP BY

name

ORDER BY

average\_price DESC;

Explanation: This query selects the name of each NFT and calculates its average sale price in USD, renaming the average column to average\_price. It uses the GROUP BY clause to group the results by NFT name and the ORDER BY clause to sort the average prices in descending order. This allows us to see which NFTs have the highest average sale prices.

1. 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.

Query: SELECT

DAYNAME(STR\_TO\_DATE(event\_date, '%Y-%m-%d')) AS day\_of\_week,

COUNT(\*) AS number\_of\_sales,

AVG(eth\_price) AS average\_eth\_price

FROM

pricedata

GROUP BY

day\_of\_week

ORDER BY

number\_of\_sales ASC;

Explanation: This query extracts the day of the week from the event\_date and groups the transactions by this day. It counts the number of sales and calculates the average ETH price for each day. The results are sorted by the count of transactions in ascending order, allowing us to see which days have the fewest to the most sales along with their average ETH prices.

1. 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.

Query: SELECT

CONCAT(

name, ' was sold for $', ROUND(usd\_price, 3), ' to ', buyer\_address,

' from ', seller\_address, ' on ', event\_date

) AS summary

FROM

pricedata;

Explanation: This query constructs a summary column for each sale by concatenating relevant information: NFT name, rounded USD price, buyer's address, seller's address, and the date. The CONCAT function combines these elements into a descriptive sentence, and the ROUND function ensures the USD price is rounded to the nearest thousandth. This provides a detailed and readable summary of each transaction.

1. Create a view called “1919\_purchases” and contains any sales where “0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685” was the buyer.

Query: CREATE VIEW 1919\_purchases AS

SELECT \*

FROM pricedata

WHERE buyer\_address = '0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685';

Explanation: This query creates a view named 1919\_purchases that includes all columns from the pricedata table for transactions where the buyer's address is 0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685. This view allows for easy access and analysis of all purchases made by this specific buyer.

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

Query: SELECT

CONCAT(

ROUND(eth\_price, -2), ' - ', ROUND(eth\_price, -2) + 99

) AS eth\_price\_range,

COUNT(\*) AS frequency

FROM

pricedata

GROUP BY

ROUND(eth\_price, -2)

ORDER BY

ROUND(eth\_price, -2);

Explanation: This query creates a histogram by grouping ETH prices into ranges rounded to the nearest hundred and counting the number of transactions in each range. It uses the ROUND function to define the ranges and the COUNT function to determine the frequency of transactions within each range. The result shows ETH price ranges and their corresponding frequencies, providing an overview of the distribution of ETH prices in the dataset.

Top of Form

Bottom of Form

1. 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.

Query: -- Highest price each NFT was bought for

SELECT

name,

MAX(usd\_price) AS price,

'highest' AS status

FROM

pricedata

GROUP BY

name

UNION

-- Lowest price each NFT was bought for

SELECT

name,

MIN(usd\_price) AS price,

'lowest' AS status

FROM

pricedata

GROUP BY

name

ORDER BY

name,

status ASC;

Explanation: This union query combines two subqueries:

1. The first subquery retrieves the highest price each NFT was bought for, grouped by the NFT name, and labels it with a status of "highest".
2. The second subquery retrieves the lowest price each NFT was bought for, grouped by the NFT name, and labels it with a status of "lowest". The results from both subqueries are combined using UNION and ordered by the NFT name and status in ascending order.

Top of Form

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

Query: SELECT

SUBSTRING(event\_date, 1, 7) AS month\_year,

name,

MAX(usd\_price) AS max\_price

FROM pricedata

GROUP BY

month\_year,

name

ORDER BY

month\_year,

max\_price DESC;

Explanation: This query extracts the month and year from the event\_date column, groups the data by month/year and name, calculates the maximum USD price for each group, and then orders the results chronologically by month/year and then by maximum price in descending order. This information can be used to identify the most valuable NFTs sold each month/year, as well as their names and prices.

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

Query: SELECT

SUBSTRING(event\_date, 1, 7) AS month\_year,

ROUND(SUM(usd\_price), -2) AS total\_volume

FROM pricedata

GROUP BY

month\_year

ORDER BY

month\_year;

Explanation: This query calculates the total sales volume for each month/year, rounds the sum to the nearest hundred, and then orders the results chronologically by month/year. This information can be used to track the overall sales volume of NFTs over time and identify trends or seasonal variations.

1. Count how many transactions the wallet "0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685"had over this time period.

Query: SELECT

COUNT(\*) AS transaction\_count

FROM pricedata

WHERE

buyer\_address = '0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685'

OR seller\_address = '0x1919db36ca2fa2e15f9000fd9cdc2edcf863e685';

Explanation: This query counts the number of rows in the pricedata table where the buyer\_address or seller\_address column is equal to the specified wallet address. This will give you the total number of transactions that the wallet has been involved in during the specified time period.

1. 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.

Query: CREATE TEMPORARY TABLE daily\_averages AS

SELECT

event\_date,

usd\_price,

AVG(usd\_price) OVER (PARTITION BY event\_date) AS daily\_average

FROM pricedata;

Explanation:

* The TEMPORARY keyword is optional in newer versions of MySQL, but it is still a good practice to include it for clarity.
* You can use the DROP TEMPORARY TABLE statement to delete the temporary table when you are finished with it.

 b) Use the table you created in Part A to filter out rows where the USD prices is below 10% of the daily average and return a new estimated value which is just the daily average of the filtered data.

Query: SELECT

event\_date,

AVG(usd\_price) AS estimated\_value

FROM daily\_averages

WHERE

usd\_price >= 0.1 \* daily\_average

GROUP BY

event\_date;

Explanation: This query filters the daily\_averages table to exclude rows where the usd\_price is less than 10% of the daily\_average. Then, it calculates the average of the remaining usd\_price values for each event\_date and returns it as the estimated\_value.

This query will give you an estimated average value for each day, excluding outlier sales that are significantly below the daily average. You can use this estimated value to get a better sense of the overall price trend for the collection over time.