Glassnode BI Challenge

This task was part of the selection process when I applied for the BI/Analytics Lead position at Glassnode.

Relevant links:

- Job description
- Data for the analysis

First thing I want to do is to load the data into a Redshift instance, so I can easily run SQL queries against.

For that, I loaded the CSV into a S3 instance and then into a empty Redshift table with the correspondent structure, using the following piece of code.

```
-- Creating the table
CREATE TABLE access_logs (
  timestamp TIMESTAMP,
  id VARCHAR(255),
  metric VARCHAR(255),
  params VARCHAR(255), -- JSON
  studio BOOLEAN,
  plan VARCHAR(1),
  status_code INTEGER
);

-- loading the data from S3 into Redshift
COPY access_logs
FROM 's3://diego-test-bucket-2023/bichallenge.csv'
IAM_ROLE 'arn:aws:iam::...:my-iam-role'
FORMAT CSV
DELIMITER ','
IGNOREHEADER 1
```

In [ ]:
```
# having a hard time connect to the Redshift instance from python, so I'l
# in case I decide to do any heavy data processing on Redshift, I'll just
import pandas as pd

access_logs = pd.read_csv('./files/bichallenge.csv')
access_logs
```

Out[ ]:

	timestamp	id	metric	
0	2022-06-01 00:00:00 UTC	9f8b8849	indicators/cdd90	{"a":"E
1	2022-06-01 00:00:01 UTC	d869c911	institutions/grayscale_market_price_usd	{"a":"E
2	2022-06-01 00:00:14 UTC	7a03cdb8	transactions/transfers_to_exchanges_count	{"a"
3	2022-06-01 00:00:29 UTC	f73b198c	eth2/estimated_annual_issuance_roi_per_validator	{"a":"B
4	2022-06-01 00:00:36 UTC	33910ce6	transactions/transfers_volume_miners_net	{"a":"B
...	...	...	...	
1278924	2022-09-29 23:53:23 UTC	c4206f13	blockchain/utxo_profit_count	{"a
1278925	2022-09-29 23:56:23 UTC	c4206f13	transactions/contract_calls_internal_count	{"a
1278926	2022-09-29 23:57:05 UTC	de546516	transactions/size_mean	{"a":"Y.
1278927	2022-09-29 23:58:02 UTC	b90af828	indicators/svl_1y_2y	{"a":"ARM
1278928	2022-09-29 23:58:03 UTC	b90af828	indicators/svl_1y_2y	{"a":"RO

1278929 rows × 7 columns

Being someone used to work with structured data, I'll parse the "params" column and convert from JSON to individual columns.

In [ ]:

```
import json
json_column = access_logs['params'].apply(json.loads) # converting to JSO
normalized_json = pd.json_normalize(json_column) # spliting the keys into
access_logs = pd.concat([access_logs, normalized_json], axis=1) # merging
access_logs = access_logs.rename(columns={ # renaming new columns
    'a': 'asset',
    'c': 'currency',
    'i': 'interval'
})
access_logs = access_logs.drop(columns=['params']) # removing redundant c
access_logs
```

Out[ ]:

	timestamp	id	metric	studio	p
0	2022-06-01 00:00:00 UTC	9f8b8849	indicators/cdd90	False	
1	2022-06-01 00:00:01 UTC	d869c911	institutions/grayscale_market_price_usd	False	
2	2022-06-01 00:00:14 UTC	7a03cdb8	transactions/transfers_to_exchanges_count	False	
3	2022-06-01 00:00:29 UTC	f73b198c	eth2/estimated_annual_issuance_roi_per_validator	False	
4	2022-06-01 00:00:36 UTC	33910ce6	transactions/transfers_volume_miners_net	False	
...	...	...	...	...	
1278924	2022-09-29 23:53:23 UTC	c4206f13	blockchain/utxo_profit_count	False	
1278925	2022-09-29 23:56:23 UTC	c4206f13	transactions/contract_calls_internal_count	False	
1278926	2022-09-29 23:57:05 UTC	de546516	transactions/size_mean	False	
1278927	2022-09-29 23:58:02 UTC	b90af828	indicators/svl_1y_2y	False	
1278928	2022-09-29 23:58:03 UTC	b90af828	indicators/svl_1y_2y	False	

1278929 rows × 9 columns

I'll also modify the "studio" column to make it easier to use and interpret the data. If we were worried about performance, perhaps we'd leave it the way it is, since booleans are easier to process.

In [ ]:
```
access_logs['access_method'] = access_logs['studio'].replace({False: 'API
access_logs = access_logs.drop(columns=['studio'])
access_logs
```

Out[ ]:

	timestamp	id	metric	plan	sta
0	2022-06-01 00:00:00 UTC	9f8b8849	indicators/cdd90	A	
1	2022-06-01 00:00:01 UTC	d869c911	institutions/grayscale_market_price_usd	B	
2	2022-06-01 00:00:14 UTC	7a03cdb8	transactions/transfers_to_exchanges_count	B	
3	2022-06-01 00:00:29 UTC	f73b198c	eth2/estimated_annual_issuance_roi_per_validator	B	
4	2022-06-01 00:00:36 UTC	33910ce6	transactions/transfers_volume_miners_net	A	
...	...	...	...	...	
1278924	2022-09-29 23:53:23 UTC	c4206f13	blockchain/utxo_profit_count	A	
1278925	2022-09-29 23:56:23 UTC	c4206f13	transactions/contract_calls_internal_count	A	
1278926	2022-09-29 23:57:05 UTC	de546516	transactions/size_mean	B	
1278927	2022-09-29 23:58:02 UTC	b90af828	indicators/svl_1y_2y	A	
1278928	2022-09-29 23:58:03 UTC	b90af828	indicators/svl_1y_2y	A	

1278929 rows × 9 columns

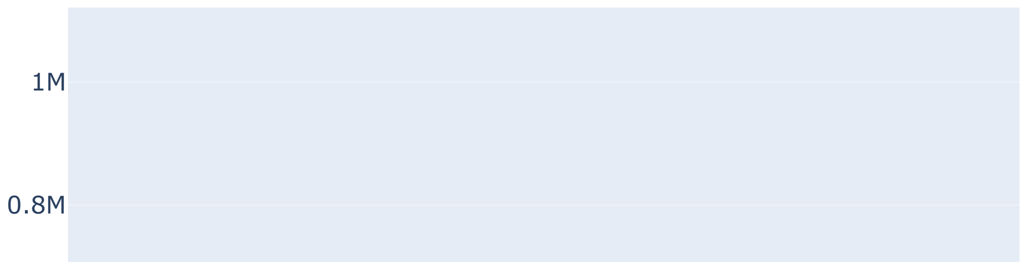
Now I'll start working on the actual questions.

Question 1

1. Is there a general user preference for accessing our data through Studio vs. API? Is there a difference for plan A users vs plan B users?

In [ ]:
```
import plotly.express as px
import plotly.offline as pyo

# Set notebook mode to work in offline
pyo.init_notebook_mode()

# Count the number of rows for each plan/studio combination
df_access_count = access_logs.groupby(['plan', 'access_method'], group_ke

# Calculate the percentage of total for each plan
df_access_count['pct_within_plan'] = df_access_count.groupby('plan', grou

# Create the bar chart
fig = px.bar(df_access_count, x='plan', y='count', color='access_method',
             text='pct_within_plan', title='Count of access by plan and a
fig.update_traces(texttemplate='%{text:.2%}', textposition='inside')

pyo.iplot(fig, filename = 'number_of_access_per_plan_by_method')

print(df_access_count)
```

Count of access by plan and access_method



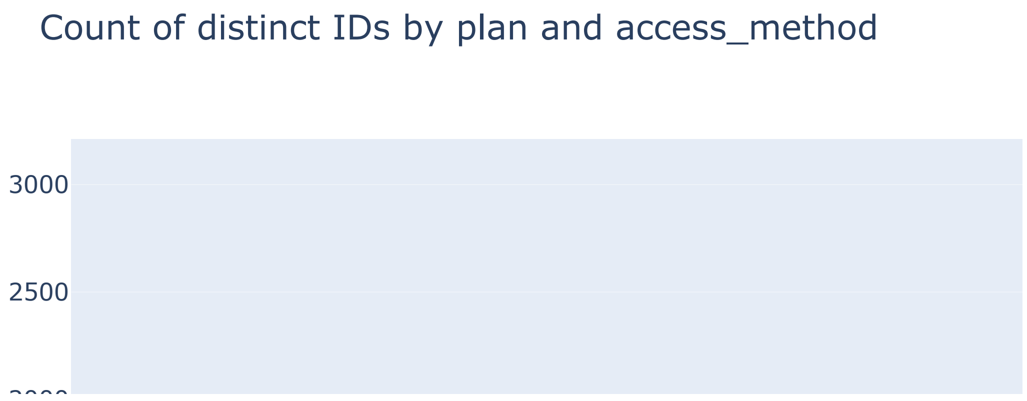
```
  plan access_method    count  pct_within_plan
0    A           API   203575         0.967424
1    A        WebApp     6855         0.032576
2    B           API  1065270         0.996978
3    B        WebApp     3229         0.003022
```

Some facts we can notice:

- Most of the access requests come from the API
- Plan B users made approximately 5x as many requests as Plan A users
- Plan A users made approximately 2x as many **WebApp** requests compared to Plan A users

Before making any conclusions, let's do the same analysis but now looking at **number of users** instead of number of requests. Some users might have made many requests while some not as many and that could make the previous analysis misleading.

In [ ]:
```
# Count the number of distinct IDs for each plan/studio combination
df_user_count = access_logs.groupby(['plan', 'access_method'], group_keys

# Count the number of distinct IDs for each plan
df_users_per_plan = access_logs.groupby(['plan'], group_keys=False)['id']

# JOIN on the 'plan' column
df_merged = pd.merge(df_user_count, df_users_per_plan, on='plan')

# Create the 'percent' column
# Notice that this is not expected to add to 100% because the same user c
df_merged['percent'] = df_merged['users'] / df_merged['total_plan_users']

# Create the bar chart
fig = px.bar(df_merged, x='plan', y='users', color='access_method', barmo
             text='percent', title='Count of distinct IDs by plan and acc
fig.update_traces(texttemplate='%{text:.2%}', textposition='inside')

pyo.iplot(fig, filename='number_of_distinct_ids_per_plan_by_access_method

df_merged
```



Out[ ]:

	plan	access_method	users	total_plan_users	percent
0	A	API	304	1924	0.158004
1	A	WebApp	1743	1924	0.905925
2	B	API	3050	3190	0.956113
3	B	WebApp	297	3190	0.093103

Important to note that the %s are not supposed to add up to 100%, since the same user can use both the API and the WebApp Very interesting how there was indeed information hidden on the previous analysis.

Some facts we can notice:

- Plan A users:
  - 16% used the API
  - 91% used the WebApp
- Plan B users:
  - 96% used the API
  - 09% used the WebApp

One last analysis, before writing the conclusion. I just want to calculate the avg number of requests per user for each access_method. The result will explain why the first and the second graph tell different stories.

In [ ]:
```
# Calculate total number of users for each access_method
users_per_access_method = access_logs.groupby('access_method', group_keys

# Calculate number of accesses for each access_method
requests_per_access_method = access_logs.groupby('access_method').size()

# Calculate average number of rows per user for each access_method
avg_requests_per_user_per_access_method = requests_per_access_method / us

avg_requests_per_user_per_access_method
```

Out[ ]:
```
access_method
API       381.722323
WebApp      4.955283
dtype: float64
```

This last analysis confirms that the number of requets per user in the API is much higher. Which is expected, since you can access the API programatically. Again, this explains why using the number of users instead of the number of requests gives us different results.

Q1 Answer

Now finally consolidating our insights to answer the question:

- Overall, there are more API users (3354) than WebApp users (2040)
- Plan A users prefer using the WebApp and Plan B users prefer to use the API (more details on the last graph plotted)
- API users tend to make more requests (expected, since you can access the API programatically)

Question 2

2. What are the top 5 metrics for plan A users and the top 5 for plan B users? Why is simply counting the number of access logs for a particular metric not an ideal approach to quantify this? What would you do instead?

As mentioned in the question, it is not ideal to use the number of access requests, because that can skew the result like in the previous excercise. Better to use the number of users who accessed the metric instead. This is the metric I'm going to calcluate.

In [ ]:
```
# Top 5 metrics for plan A users
plan_a_metrics = access_logs[access_logs['plan'] == 'A'].groupby('metric'
plan_a_metrics['plan'] = 'A'

# Top 5 metrics for plan B users
plan_b_metrics = access_logs[access_logs['plan'] == 'B'].groupby('metric'
plan_b_metrics['plan'] = 'B'

# Concatenate plan A and plan B metrics
```

```
top_metrics = pd.concat([plan_a_metrics, plan_b_metrics])

# Plot using Plotly Express
fig = px.bar(top_metrics, x='metric', y='users_count', color='plan', barm
pyo.iplot(fig, filename='top_metrics_by_plan')

print(top_metrics)
```

1000
800
600
rs_count

```
                                   metric  users_count plan
160  indicators/3iq_ethq_premium_percent          364    A
132        eth2/staking_total_volume_sum          194    A
210                 indicators/sol_1w_1m          189    A
316             mining/revenue_from_fees          175    A
221             indicators/sopr_more_155          159    A
249                 indicators/sol_1w_1m         1039    B
193  indicators/3iq_ethq_premium_percent          603    B
356                mempool/txs_value_sum          581    B
119          distribution/non_zero_count          378    B
301   institutions/3iq_qbtc_holdings_sum          362    B
```

Q2 Answer

These (check above) are the top 5 most popular metrics between users of each plan (by number of users). Two of them are present in both Plan A and Plan B users ("indicators/3iq_ethq_premium_percent" and "indicators/sol_1w_1m").

Since the metrics can be accessed via API, the number of access requests can be easily skewed by multiplte requests. A better way to verify for which metrics are the most popular is to use the number of distinct users accessing each one of them.

Question 3

3. How many users use the API regularly? How would you define regularity here and apply this to your query/solution? Does API usage generally tend to show particular usage trends with respect to the parameters used, e.g. interval?

The regularity can be defined in many different ways, but what is important to establish is that the "regularity" of a user can change over time. For example a user who used the app "regularly" during it's first year might churn after that one year.

So the ideal is to have not a single "regularity" value per user, but one value per user and period of time. We could be as granular as one value per day, but for the purpose of this exercise, I'm thinking about once a month.

I'll do the heavy lifting externally on SQL (since I have this data loaded on a Redshift instance) and then use Python to present the insights here.

```sql
-- How many total users in the data?
SELECT COUNT(distinct id) FROM access_logs; -- output: 4790

-- What is the data interval we're working with
SELECT MIN(timestamp),MAX(timestamp) FROM access_logs;
-- MIN: 2022-06-01 00:00:00
-- MAX: 2022-09-29 23:58:03
```

After noticing that we only have 4 months worth of data, I decided to try calculate by week (instead of by month). I'll simply check for each week if a user has any activity or not. The purpose is mainly to show how I'd approach this, but there might be some improvements opportunities worth exploring when adopting this in a real world scenario.

First I'll create a table that is going to tell me how many API requests were made by each user in each week:

```sql
CREATE TABLE users_weekly_activity AS
(
    WITH users_weekly_activity AS (
        SELECT
            id as user_id,
            DATE_TRUNC('week',timestamp) as date_week,
            COUNT(1) as requests_count
        FROM access_logs
        WHERE not studio -- only API requests
        GROUP BY 1,2
    )
    , all_weeks AS (
        SELECT distinct DATE_TRUNC('week',timestamp) as date_week
FROM access_logs
```

```
    )
    , all_users AS (
        SELECT distinct id as user_id FROM access_logs WHERE not
studio
    )
    , all_users_and_weeks AS (
        SELECT * FROM all_users CROSS JOIN all_weeks
    )
    , users_weekly_activity_all_weeks AS (
        SELECT
            *
        FROM all_users_and_weeks
        LEFT JOIN users_weekly_activity USING(user_id,date_week)
    )
    SELECT * FROM users_weekly_activity_all_weeks
);
```

This table can be found at the files folder and is going to look like this:

user_id |week |api_requests |:----------|----------------------:|:-----------| d869c911 |2022-06-13 00:00:00 |2951 d869c911 |2022-05-30 00:00:00 |3716 d869c911 |2022-06-06 00:00:00 |4952 ... |... |...

Following my logic, to answer the first question (How many users use the API regularly?) I need to pick a data interval and define how I'll classify a user as "regular" during that period. For the sake of simplicity I'll just check whoever has made at least one API call in all the 18 weeks of the period we're working with and consider them as regular users. Obviously this approach is extremily conservative and has many limitations (eg. not considering users who have started using the API later and have been regular since then; or users who only skipped a few weeks).

```
WITH regular_users AS (
    SELECT
        user_id,
        COUNT(1) as weeks_active
    FROM users_weekly_activity
    WHERE requests_count > 1
    GROUP BY 1
    HAVING weeks_active = 18
)
SELECT COUNT(1) FROM regular_users;
-- output: 135
```

There are 135 users who have made at least one API request in every single week of those 4 months.

A more robust approach could be to create some type of metric that takes into consideration three factors:

- How long since the first API request
- How often this user has made API requests over the whole lifetime
- How "new" is the user

And by using then in combination with the proper thresholds, create a logic that would classify users as:

- New user (user who made first API request recently)
- Dormant (users who have recently stopped making requests)
- Active
- Churned (somone who hasn't been seen for a long time)

To answer about parameters trends on API usage:

Interval

In [ ]:
```
# Only API requests
API_requests = access_logs[access_logs['access_method'] == 'API']

# GROUP BY
metrics = API_requests.groupby('interval').agg(
    unique_users=('id', 'nunique'),
    total_requests=('id', 'count')
).reset_index()

metrics['avg_requests_per_user'] = metrics['total_requests'] / metrics['u

metrics.sort_values('avg_requests_per_user', ascending=False)
```

Out[ ]:

	interval	unique_users	total_requests	avg_req
1	10m	115	464391	
3	1h	283	364358	
4	1month	44	24012	
5	1w	57	24497	
6	24h	3212	391551	
8	undefined	1	32	
0	/home/henry/Documents/glassnode_analysis/prod/...	1	1	
2	1d	2	2	
7	day	1	1	

As expected, shorter intervals (eg. 1h and 10m) tend to create more API requests for each user using them.

But the most popular interval option used (by number of users) is the 24h interval.

Asset

In [ ]:
```
# I noticed both BTC and btc, so I'm converting all to uppercase
API_requests['asset'] = API_requests['asset'].str.upper()

# GROUP BY
metrics = API_requests.groupby('asset').agg(
    unique_users=('id', 'nunique'),
    total_requests=('id', 'count')
).reset_index()

metrics['avg_requests_per_user'] = metrics['total_requests'] / metrics['u

# Calculate total requests for all assets
total_requests_all_assets = metrics['total_requests'].sum()

# Add a column for percentage of total requests
metrics['pct_of_total_requests'] = metrics['total_requests'] / total_requ

# Add a cumulative SUM
metrics = metrics.sort_values('total_requests', ascending=False).reset_in
metrics['pct_of_total_cumulative'] = metrics['pct_of_total_requests'].cum

metrics.head(5)
```

Out[ ]:

	asset	unique_users	total_requests	avg_requests_per_user	pct_of_total_requests	p
0	BTC	3080	422866	137.294156	33.326844	
1	ETH	649	215726	332.397535	17.001761	
2	USDT	113	53234	471.097345	4.195469	
3	USDC	87	46957	539.735632	3.700767	
4	LTC	108	14350	132.870370	1.130950	

These are the top 5 assets by number of requests generated.

BTC and ETH alone generated 50% of all API requests (check the table last column - "pct_of_total_cumulative")

Currency

In [ ]:

```
# I noticed both USD and usd, so I'm converting all to uppercase
API_requests['currency'] = API_requests['currency'].str.upper()

# GROUP BY
metrics = API_requests.groupby('currency').agg(
    unique_users=('id', 'nunique'),
    total_requests=('id', 'count')
).reset_index()

metrics['avg_requests_per_user'] = metrics['total_requests'] / metrics['u

metrics.sort_values('total_requests', ascending=False).head(5)
```

Out[ ]:

	currency	unique_users	total_requests	avg_requests_per_user
1	NATIVE	3310	1083600	327.371601
2	USD	121	170362	1407.950413
0	COUNT	7	14883	2126.142857

Native is the most used Currency, but the number of requests per user for USD is higher (~4x).

Q3 Answer

- There are more robust ways of defining "regularity", but for simplicity I calculated how many users have made API requests at least once a week for the 18 weeks period of the data set
- There were 135 users who have been active in all 18 weeks
- Trends related to the parameres used:
    - shorter intervals (eg. 1h and 10m) tend to create more API requests for each user using them
    - the most popular interval option used (by number of users) is the 24h interval
    - BTC and ETH alone generated 50% of all API requests
    - NATIVE is the most used Currency

Question 4

4. Based on the above data, how many users come to our platform, perform any set of actions within a short time interval, and never come back?

To check this, we can use the `user_weekly_activity` table we created on Redshift (only API access requests). My approach is going to be very simple and I'll check for all users who:

- never had any activity after the first week of activity
- and have made the first API request at least 4 weeks (arbitrary) before the end of the database timeframe

This way we don't call "churn" users who perhaps only "dormant" (following the "dormant" definition made previously).

I'll first make a few adjustments to the existing `user_weekly_activity` to facilitate finding those users. I'll remove all data for weeks before the user made the first API request and will add a column that indicates how many weeks have passed since the first API request.

```
CREATE TABLE users_weekly_activity_refined AS
(
    WITH identify_weeks_before_first_request AS (
        SELECT
            user_id,
            date_week,
            requests_count,
            MIN(CASE WHEN requests_count IS NOT NULL THEN
date_week END) OVER (PARTITION BY user_id) AS first_activity_week
        FROM users_weekly_activity
```

```sql
    )
    , remove_weeks_before_first_request AS (
        SELECT
            user_id,
            date_week,
            requests_count,
            RANK() OVER (PARTITION BY user_id ORDER BY date_week)
AS week_lifetime_number
        FROM identify_weeks_before_first_request
        WHERE date_week >= first_activity_week
    )
    SELECT
        *
    FROM remove_weeks_before_first_request
);
```

This table can also be found at the files folder and now the data is going to look like this:

user_id |week | api_requests | week_lifetime_number | |:----------|-------------------- ---:|:---------------|:---------------------| b90af828 | 2022-06-13 00:00:00 | 52 | 1 | b90af828 | 2022-06-20 00:00:00 | 503 | 2 | b90af828 | 2022-06-27 00:00:00 | 606 | 3 | ... | ... | ... | ... |

Now I just need to identify and count those who have been using the platform for over 4 weeks and never had any activity after the first week of activity:

```sql
WITH users_over_4_weeks AS (
    SELECT DISTINCT(user_id)
    FROM users_weekly_activity_refined
    WHERE week_lifetime_number > 4
)
, users_with_activity_after_first_week AS ( -- easier to identify
the complement
    SELECT DISTINCT(user_id)
    FROM users_weekly_activity_refined
    WHERE week_lifetime_number > 1 and requests_count is not null
)
SELECT COUNT(DISTINCT user_id)
FROM users_weekly_activity_refined
WHERE user_id in (SELECT user_id FROM users_over_4_weeks)
  AND user_id not in (SELECT user_id FROM
users_with_activity_after_first_week);
```

This query returns 247 users who had activity on the first week and never returned. This is aproximately 7.4% of all the 3324 API users.

Q4 Answer

- Approximately 7.4% of the API users come to our platform, perform any set of actions within a week, and never come back.