

SQL Analyzing Business Metrics - Common Metrics

Introduction

As a data scientist, when you're not investigating spikes or dips in your data, you might be building dashboards of **KPIs, or key performance indicators for a company.**

KPIs are often displayed on TVs on the walls of the office, and serve as **high level health metrics for the business.** While every company's metrics are defined slightly differently, the basics are usually very similar.

In this lesson we'll take a look at basic KPIs like Daily Revenue, Daily Active Users, ARPU, and Retention for a video game, **Mineblocks.**

Daily Revenue

At the heart of every company is revenue, and Mineblocks is no exception. For our first KPI we'll calculate daily revenue.

Instructions

1.

Daily Revenue is simply the sum of money made per day.

To get close to Daily Revenue, we calculate the daily sum of the prices in the purchases table. Complete the query by using the sum function and passing the price column from the purchases table.

```
select
  date(created_at),
  round(sum(price), 2)
from purchases
group by 1
order by 1;
```

Great! That query doesn't take refunds into account. We'll update the query to exclude refunds.

Fields like refunded_at will only have data if the transaction was refunded, and otherwise left null.

Instructions

1.

Update our daily revenue query to exclude refunds. Complete the query by filtering for refunded_at is not null.

```
select
  date(created_at),
  round(sum(price), 2) as daily_rev
from purchases
```

where refunded_at is NOT NULL
group by 1
order by 1;

Daily Active Users

Mineblocks is a game, and one of the core metrics to any game is the number people who play each day. That KPI is called *Daily Active Users*, or *DAU*. *DAU* is defined as the number of unique players seen in-game each day. It's important not to double count users who played multiple times, so we'll use `distinct` in our count function.

Likewise, Weekly Active Users (WAU) and Monthly Active Users (MAU) are in the same family.

1.

For Mineblocks, we'll use the `gameplays` table to calculate DAU. Each time a user plays the game, their session is recorded in `gameplays`. Thus, a distinct count of users per day from `gameplays` will give us DAU.

Calculate Daily Active Users for Mineblocks. Complete the query's count function by passing in the `distinct` keyword and the `user_id` column name.

```
select
  date(created_at),
  count(distinct user_id) as dau
from gameplays
group by 1
order by 1;
```

Daily Active Users 2

Great! Since Mineblocks is on multiple platforms, we can calculate DAU per-platform.

Instructions

1.

Previously we calculated DAU only per day, so the output we wanted was `[date, dau_count]`. Now we want DAU per platform, making the desired output `[date, platform, dau_count]`.

Calculate DAU for Mineblocks per-platform. Complete the query below. You will need to select the `platform` column and add a count function by passing in the `distinct` keyword and the `user_id` column name.

```
select
  date(created_at),
  platform,
  count(distinct user_id) as dau
from gameplays
group by 1, 2
order by 1, 2;
```

Daily Average Revenue Per Purchasing User

We've looked at DAU and Daily Revenue in Mineblocks. Now we must understand the purchasing habits of our users.

Mineblocks, like every freemium game, has two types of users:

- purchasers: users who have bought things in the game
- players: users who play the game but have not yet purchased

The next KPI we'll look at *Daily ARPPU* - Average Revenue Per Purchasing User. This metric shows if the average amount of money spent by purchasers is going up over time.

Daily ARPPU is defined as the sum of revenue divided by the number of purchasers per day.

Instructions

1.

To get Daily ARPPU, modify the daily revenue query from earlier to divide by the number of purchasers.

Complete the query by adding a numerator and a denominator. The numerator will display daily revenue, or sum the price columns. The denominator will display the number of purchasers by passing the distinct keyword and the user_id column name into the count function.

```
select
  date(created_at),
  round(sum(price)/ count(distinct user_id), 2) as arppu
from purchases
where refunded_at is null
group by 1
order by 1;
```

Daily Average Revenue Per User

The more popular (and difficult) cousin to Daily ARPPU is *Daily ARPU*, **Average Revenue Per User**. ARPU measures the average amount of money we're getting across all players, whether or not they've purchased.

ARPPU increases if purchasers are spending more money. ARPU increases if more players are choosing to purchase, even if the purchase size stays consistent.

No one metric can tell the whole story. That's why it's so helpful to have many KPIs on the same dashboard.

Daily ARPU is defined as revenue divided by the number of players, per-day. To get that, we'll need to calculate the daily revenue and daily active users separately, and then join them on their dates.

One way to easily create and organize temporary results in a query is with *CTEs*, *Common Table Expressions*, also known as with clauses. The with clauses make it easy to define and use results in a more organized way than subqueries.

These clauses usually look like this:

```
with {subquery_name} as (  
    {subquery_body}  
)  
select ...  
from {subquery_name}  
where ...
```

Instructions

1.

Use a with clause to define daily_revenue and then select from it.

```
with daily_revenue as (  
    select  
        date(created_at) as dt,  
        round(sum(price), 2) as rev  
    from purchases  
    where refunded_at is null  
    group by 1  
)  
select * from daily_revenue order by dt;
```

Great! Now you're familiar with using the with clause to create temporary result sets.

You just built the first part of ARPU, daily_revenue. From here we can build the second half of ARPU in our with clause, daily_players, and use both together to create ARPU.

Instructions

1.

Building on this CTE, we can add in DAU from earlier. Complete the query by calling the DAU query we created earlier, now aliased as daily_players:

```
with daily_revenue as (  
    select  
        date(created_at) as dt,  
        round(sum(price), 2) as rev  
    from purchases  
    where refunded_at is null  
    group by 1  
)  
daily_players as (  
    select  
        created_at as dt,  
        user_id as players  
    from gameplays  
    group by 1  
)  
select * from daily_players order by dt;
```

Now that we have the revenue and DAU, join them on their dates and calculate daily ARPU. Complete the query by adding the keyword using in the join clause.

```
with daily_revenue as (  
  select  
    date(created_at) as dt,  
    round(sum(price), 2) as rev  
  from purchases  
  where refunded_at is null  
  group by 1  
)  
daily_players as (  
  select  
    date(created_at) as dt,  
    count(distinct user_id) as players  
  from gameplays  
  group by 1  
)  
select  
  daily_revenue.dt,  
  daily_revenue.rev / daily_players.players  
from daily_revenue  
join daily_players using (dt); //dt is the common column in both tables
```

ARPU 3

Nice work, you just defined ARPU for Mineblocks!

In our ARPU query, we used using instead of on in the join clause. This is a special case join.

from daily_revenue

join daily_players using (dt);

When the columns to join have the same name in both tables you can use using instead of on. Our use of the using keyword is in this case equivalent to this clause:

from daily_revenue

join daily_players on

daily_revenue.dt = daily_players.dt;

1 Day Retention

Now let's find out what percent of Mineblock players are returning to play the next day. This KPI is called *1 Day Retention*.

Retention can be defined many different ways, but we'll stick to the most basic definition. For all players on Day N, we'll consider them retained if they came back to play again on Day N+1.

This will let us track whether or not Mineblocks is getting "stickier" over time. The stickier our game, the more days players will spend in-game. And more time in-game means more opportunities to monetize and grow our business.

o calculate retention, start from a query that selects the date(created_at) as dt and user_id columns from the gameplays table.

```
select
  date(created_at) as dt,
  user_id
from gameplays as g1
order by dt
limit 100;
```

2.

Now we'll join gameplays on itself so that we can have access to all gameplays for each player, for each of their gameplays.

This is known as a self-join and will let us connect the players on Day N to the players on Day N+1. In order to join a table to itself, it must be aliased so we can access each copy separately.

We aliased gameplays in the query above because in the next step, we need to join gameplays to itself so we can get a result selecting [date, user_id, user_id_if_retained].

Complete the query by using a join statement to join gameplays to itself on user_id using the aliases g1 and g2.

```
select
  date(g1.created_at) as dt,
  g1.user_id
from gameplays as g1
join gameplays as g2 on
  g1.user_id = g2.user_id
order by 1
limit 100;
```

We don't use the using clause here because the join is about to get more complicated.

Now that we have our gameplays table joined to itself, we can start to calculate retention.

1 Day Retention is defined as the number of players who returned the next day divided by the number of original players, per day. Suppose 10 players played Mineblocks on Dec 10th. If 4 of them play on Dec 11th, the 1 day retention for Dec 10th is 40%.

The previous query joined all rows in gameplays against all other rows for each user, making a massive result set that we don't need.

We'll need to modify this query.

```
select
  date(g1.created_at) as dt,
  g1.user_id,
  g2.user_id
from gameplays as g1
  join gameplays as g2 on
    g1.user_id = g2.user_id
    and date(g1.created_at) = date(datetime(g2.created_at, '-1 day'))
order by 1
limit 100;
```

Complete the query above such that the join clause includes a date join:

`date(g1.created_at) = date(datetime(g2.created_at, '-1 day'))`

This means "only join rows where the date in g1 is one less than the date in g2", which makes it possible to see if users have returned!

2.

The query above won't return meaningful results because we're using an inner join. This type of join requires that the condition be met for all rows, effectively limiting our selection to only the users that have returned.

Instead, we want to use a left join, this way all rows in g1 are preserved, leaving nulls in the rows from g2 where users did not return to play the next day.

Change the join clause to use left join and count the distinct number of users from g1 and g2 per date.

```
select
  date(g1.created_at) as dt,
  count(distinct g1.user_id) as total_users,
  count(distinct g2.user_id) as retained_users
from gameplays as g1
  left join gameplays as g2 on
    g1.user_id = g2.user_id
    and date(g1.created_at) = date(datetime(g2.created_at, '-1 day'))
group by 1
order by 1
limit 100;
```

Now that we have retained users as `count(distinct g2.user_id)` and total users as `count(distinct g1.user_id)`, divide retained users by total users to calculate 1 day retention!

```
select
```

```
date(g1.created_at) as dt,  
round(100 * count(distinct g2.user_id) /  
count(distinct g1.user_id)) as retention  
from gameplays as g1  
left join gameplays as g2 on  
g1.user_id = g2.user_id  
and date(g1.created_at) = date(datetime(g2.created_at, '-1 day'))  
group by 1  
order by 1  
limit 100;
```

Common Metrics Conclusion

While every business has different metrics to track their success, most are based on revenue and usage.

The metrics in this lesson are merely a starting point, and from here you'll be able to create and customize metrics to track whatever is most important to your company.

And remember, data science is exploratory! The current set of metrics can always be improved and there's usually more to any spike or dip than what immediately meets the eye.

Let's generalize what we've learned so far:

- *Key Performance Indicators* are high level health metrics for a business.
- *Daily Revenue* is the sum of money made per day.
- *Daily Active Users* are the number of unique users seen each day
- *Daily Average Revenue Per Purchasing User (ARPPU)* is the average amount of money spent by purchasers each day.
- *Daily Average Revenue Per User (ARPU)* is the average amount of money across all users.
- *1 Day Retention* is defined as the number of players from Day N who came back to play again on Day N+1.