Periscope Data

SQL Analytics: Best Practices, Tips and Tricks

Content

Data is the most important resource for today's companies. But, like other resources, it can't be used to create business insights in its raw form; it has to be processed and structured for analytics before it creates real value. Data analytics is the process of getting a company's data into a format that allows it to be usable in creating business insights and recommending actions.

For most companies, the way they analyze their data for business is SQL, a language that queries information in a database and transforms it to answer questions that will move the organization forward.

In a simple case, SQL can be used to translate individual pieces of transactional information into aggregates that can be used to illustrate a broader picture. For example, a list of every transaction your business has made is not valuable, it's far too dense and complex to illustrate anything meaningful. But if you were to group the information by day or week and use a COUNT to create totals by time period, you'd start to see patterns emerge. These patterns can lead to insights that are much more valuable. Using simple aggregates like that to establish some baseline KPIs is a critical foundation for your analytics structure.

In this guide, you'll learn some simple tips for creating data analytics infrastructure using SQL. We'll go through an explanation of some foundational concepts you'll need to manipulate your data and make it valuable, with useful SQL code examples at every step along the way. These fundamentals can then be used as the building blocks for a solid analytical base.

In the first section, you'll learn about joining data in the first row to focus your analysis on a specific data point, such as only the most recent information. Since recent data is often the target of your questions, this is a very easy way to pinpoint pertinent metrics and surface them quickly.

4 Ways to Join the First Row in SQL

To start, let's consider a hypothetical task of creating a list of users and the most recent widget each user has created. We have a users table and a widgets table, and each user has many widgets. users.id is the primary key on users, and widgets.user_id is the corresponding foreign key in widgets.

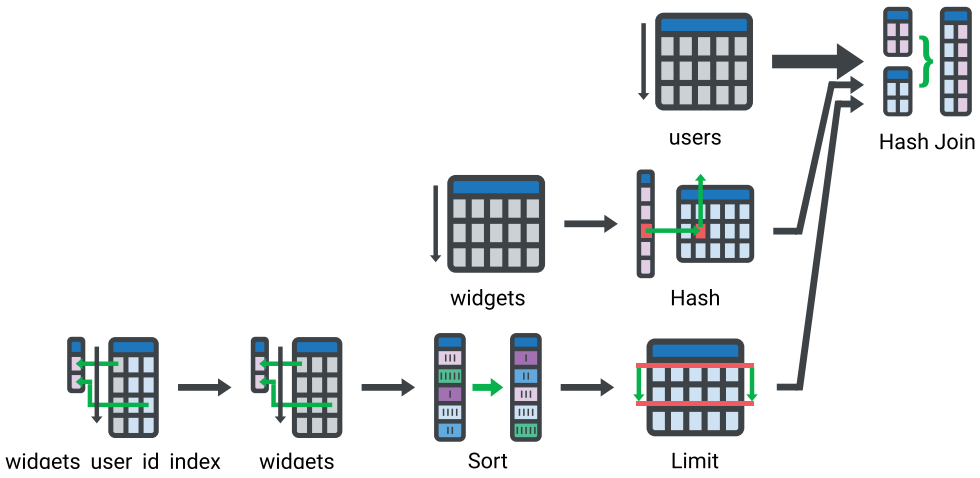
To solve this problem, we need to join only the first row. There are several ways to do this. Here are a few different techniques and when to use them.

Use correlated subqueries when the foreign key is indexed

Correlated subqueries are subqueries that depend on the outer query. It's like a for loop in SQL. The subquery will run once for each row in the outer query:

```
select * from users join widgets on widgets.id = (
    select id from widgets
    where widgets.user_id = users.id
    order by created_at desc
    limit 1
)
```

Notice the where *widgets.user_id = users.id* clause in the subquery. It queries the *widgets* table once for each *user* row and selects that user's most recent *widget* row. It's very efficient if *user_id* is indexed and there are few users.



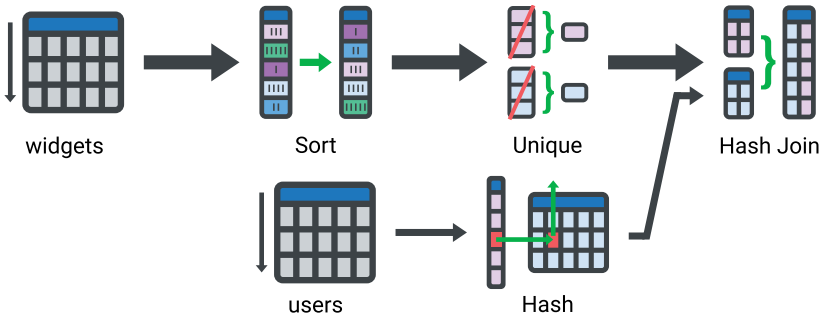
Use a complete subquery when you don't have indexes

Correlated subqueries break down when the foreign key isn't indexed, because each subquery will require a full table scan.

In that case, we can speed things up by rewriting the query to use a single subquery, only scanning the *widgets* table once:

```
select * from users join (
    select distinct on (user_id) * from widgets
    order by user_id, created_at desc
) as most_recent_user_widget
on users.id = most_recent_user_widget.user_id
```

This new subquery returns a list of the most recent widgets, one for each user. We then join it to the users table to get our list.



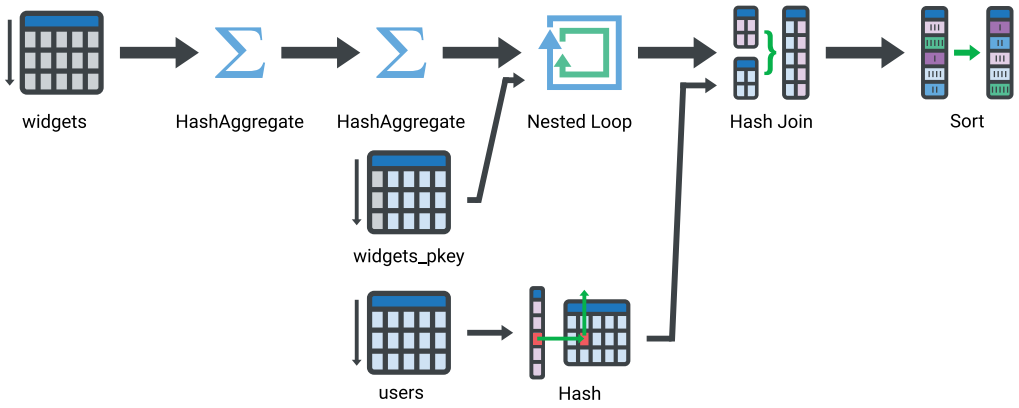
We've used Postgres' `DISTINCT ON` syntax to easily query for only one widget per `user_id`. If your database doesn't support something like `DISTINCT ON`, you have two options:

Use nested subqueries if you have an ordered ID column

In our example, the most recent row always has the highest `id` value. This means that even without `DISTINCT ON`, we can cheat with our nested subqueries like this:

```
select * from users join (
    select * from widgets
    where id in (
        select max(id) from widgets group by user_id
    )
) as most_recent_user_widget
on users.id = most_recent_user_widget.user_id
```

We start by selecting the list of IDs representing the most recent widget per user. Then we filter the main `widgets` table to those IDs. This gets us the same result as `DISTINCT ON` since sorting by `id` and `created_at` happen to be equivalent.



Use window functions if you need more control

If your table doesn't have an `id` column, or you can't depend on its min or max to be the most recent row, use `row_number` with a window function. It's a little more complicated, but a lot more flexible:

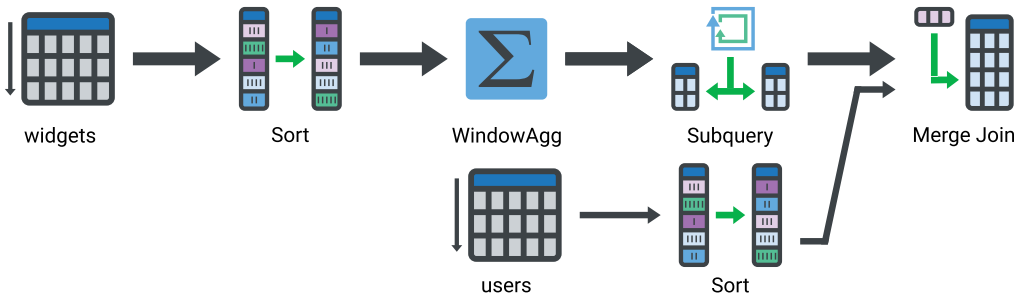
```
select * from users join (
    select * from (
        select *, row_number() over (
            partition by user_id
            order by created_at desc
        ) as row_num
        from widgets
    ) as ordered_widgets
    where ordered_widgets.row_num = 1
) as most_recent_user_widget
on users.id = most_recent_user_widget.user_id
order by users.id
```

The interesting part is here:

```
select *, row_number() over (
    partition by user_id
    order by created_at desc
) as row_num
from widgets
```

`over (partition by user_id order by created_at desc` specifies a sub-table, called a window, per `user_id`, and sorts those windows by `created_at desc`. `row_number()` returns a row's position within its window. Thus the first widget for each `user_id` will have `row_number 1`.

In the outer subquery, we select only the rows with a `row_number` of 1. With a similar query, you could get the 2nd or 3rd or 10th rows instead.



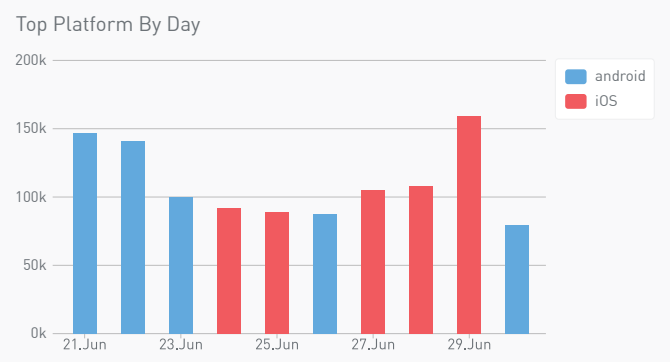
In the next section, you'll learn more about how to further pinpoint specific information. By selecting only one row per group, you can cut through the clutter to focus your analysis on a min or a max. This is an easy way to speed up your analysis and get to insights as fast as possible.

Selecting One Row Per Group

Selecting the first row for each group

Sometimes you just want to graph the winners. For example, how can you determine whether your app had more iOS or Android users today? Grouping and counting the daily usage per platform is easy, but getting only the top platform for each day can be tough.

Unlike joining only the first row, primary and foreign keys won't be of much use, so we'll need a different approach to make a chart like this:



Let's start with daily counts per platform from our `gameplays` table:

```
select date(created_at) dt, platform, count(1) ct
from gameplays
group by 1, 2
```

This gives us a familiar table, with one date spanning multiple rows:

```
    dt      | platform |   ct
------------+----------+--------
 2014-06-30 | iOS      |  49751
 2014-06-30 | android  |  80781
 2014-06-29 | iOS      | 158909
 2014-06-29 | android  |  91380
 2014-06-28 | iOS      | 108206
 2014-06-28 | android  |  95363
 2014-06-27 | iOS      | 105756
 2014-06-27 | android  |  92316
```

We want a table with one row per date with the highest count, like this:

```
  dt        | platform |   ct
------------+----------+--------
 2014-06-30 | android  |  80781
 2014-06-29 | iOS      | 158909
 2014-06-28 | iOS      | 108206
 2014-06-27 | iOS      | 105756
```

Postgres and Redshift

As usual on Postgres and Redshift, window functions make this an easy task. We'll use the `row_number()` function partitioned by date in an inner query, and then filter to `row_num = 1` in the outer query to get just the first record per group.

This function numbers each of the rows:

```
row_number() over (partition by dt order by ct desc) row_num
```

We'll plug it into an inner query, like so, to get the desired results:

```
select dt, platform, ct
from (
  select
    date(created_at) dt,
    platform,
    count(1) ct,
    row_number() over
      (partition by dt order by ct desc) row_num
  from gameplays
  group by 1, 2
) t
where row_num = 1
```

MySQL

Since MySQL doesn't have window functions, we'll do something similar using `group_concat`. With `group_concat` we can roll up the platform column into a comma-separated string per-date, ordered by its count:

```
group_concat(platform order by ct desc) platform
```

That'll give us all the platforms, with the highest-performing platform first.
The complete query looks like this:

```
select
  dt,
  group_concat(platform order by ct desc) platform
from (
  select date(created_at) dt, platform, count(1) ct
  from gameplays
  group by 1, 2
) t
group by 1
```

Which gives us results like this:

```
    dt      |  platform
------------+-------------
 2014-06-30 | android,iOS
 2014-06-29 | iOS,android
 2014-06-28 | iOS,android
 2014-06-27 | iOS,android
 2014-06-26 | android,iOS
 2014-06-25 | iOS,android
```

Perfect! The platforms with the highest counts are first in the list. Using substring_index — which grabs the first word before the comma — we can extract out only the first platform:

```
substring_index(
  group_concat(
    platform order by ct desc
  )
, ',',1) platform,
```

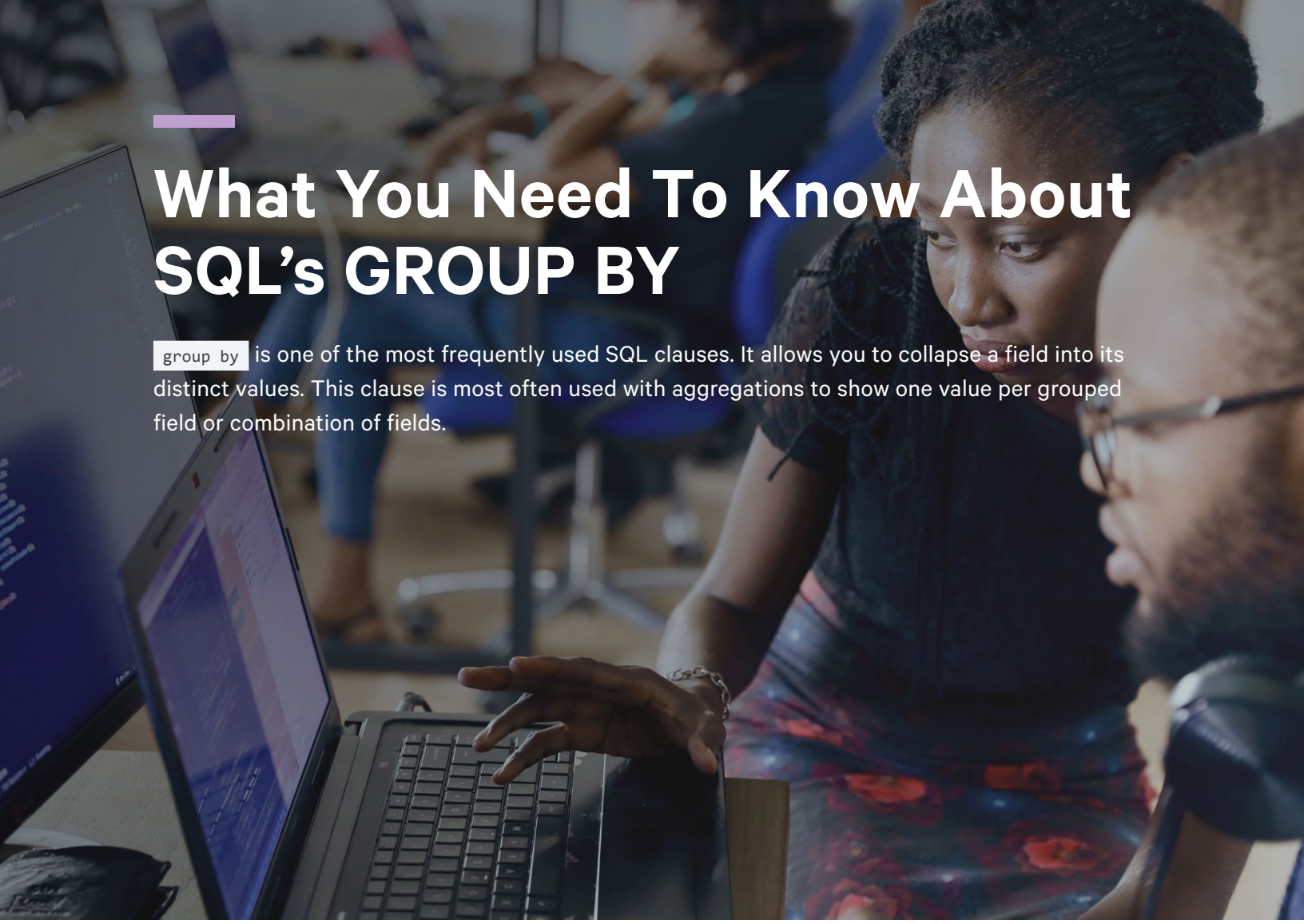
Once we know the highest-performing platform each day, we can use `max(ct)` to get the count associated with that platform.

The complete query:

```
select
  dt,
  substring_index(
    group_concat(
      platform order by ct desc
    ), ',',1
  ) platform,
  max(ct) ct
from (
  select date(created_at) dt, platform, count(1) ct
  from gameplays
  group by 1, 2
) t
group by 1
```

That's all there is to it! As always, if you spend too much time hand-crafting queries like this, consider signing up for Periscope to make it faster and easier.

In the next section, we'll zoom out and look at using group by to collect data into aggregates. When building your company's analytics infrastructure, mastering group by is a key method to condense all of the data you need into an efficient KPI dashboard.



What You Need To Know About SQL's GROUP BY

`group by` is one of the most frequently used SQL clauses. It allows you to collapse a field into its distinct values. This clause is most often used with aggregations to show one value per grouped field or combination of fields.

Consider the following table:

COUNTRY	CONTINENT	POPULATION	AREA	CURRENCY
China	Asia	1,373,541,278	9,596,960	Yuan
India	Asia	1,266,883,598	3,287,263	Rupie
United States	North America	323,995,528	9,826,675	Dollar
Indonesia	Asia	258,316,051	1,904,569	Rupiah
Brazil	South America	205,823,665	8,514,877	Real
Pakistan	Asia	201,995,540	796,095	Rupie

We can use a group by and aggregates to collect multiple types of information. For example, a group by can quickly tell us the number of countries on each continent.

```
-- How many countries are in each continent?
select
  continent
  , count(*)
from
  countries
group by
  continent
```

CONTINENT	COUNT
Africa	59
Europe	52
Asia	51
North America	30
Oceania	22
South America	14
Central America	7

Keep in mind when using `group by` :

- `group by` X means put all those with the same value for X in the same row.
- `group by` X, Y put all those with the same values for both X and Y in the same row.

More interesting things about GROUP BY

1. Aggregations can be filtered using the HAVING clause

You will quickly discover that the where clause cannot be used on an aggregation. For instance

```
select
  continent
  , max(area)
from
  countries
where
  max(area) >= 1e7
group by
  1
```

will not work, and will throw an error. This is because the where statement is evaluated before any aggregations take place. The alternate having is placed after the group by and allows you to filter the returned data by an aggregated column.

```
select
 continent
  , max(area)
from
  countries
group by
  1
having
  max(area) >= 1e7 -- Exponential notation can keep code clean!
```

Using having, you can return the aggregate filtered results!

2. You can often GROUP BY column number

In many databases you can can group by column number as well as column name. Our first query could have been written:

```
select
  continent
  , count(*)
from
  base
group by
  1
```

and returned the same results. This is called ordinal notation and its use is debated. It predates column based notation and was SQL standard until the 1980s.

- It is less explicit, which can reduce legibility for some users.
- It can be more brittle. A query select statement can have a column name changed and continue to run, producing an unexpected result.

On the other hand, it has a few benefits.

- SQL coders tend towards a consistent pattern of selecting dimensions first, and aggregates second. This makes reading SQL more predictable.
- It is easier to maintain on large queries. When writing long ETL statements, I have had group by statements that were many, many lines long. I found this difficult to maintain.
- Some databases allow using an aliased column in the group by. This allows a long case statement to be grouped without repeating the full statement in the group by clause. Using ordinal positions can be cleaner and prevent you from unintentionally grouping by an alias that matches a column name in the underlying data. For example, the following query will return the correct values:

```
-- How many countries use a currency called the dollar?
select
  case when currency = 'Dollar' then currency
    else 'Other'
  end as currency --bad alias
  , count(*)
from
  countries
group by
  1
```

CURRENCY	COUNT
Dollar	54
Other	194

But this will not, and will segment by the **base table's** currency field *while accepting the new alias column labels:*

```
select
  case when currency = 'Dollar' then currency
    else 'Other'
  end as currency --bad alias
  , count(*)
from
  countries
group by
  currency
```

CURRENCY	COUNT
Dollar	54
Other	33
Other	23
Other	13
Other	9
Other	8
Other	8

This is 'expected' behavior, but remain vigilant.

A common practice is to use ordinal positions for ad-hoc work and column names for production code. This will ensure you are being completely explicit for future users who need to change your code.

3. The implicit GROUP BY

There is one case where you can take an aggregation without using a group by. When you are aggregating the full table there is an implied `group by`. This is known as the <grand total> in SQL standards documentation.

```
-- What is the largest and average country size in Europe?
select
  max(area) as largest_country
  , avg(area) as avg_country_area
from
  countries
where
  continent = 'Europe'
```

LARGEST COUNTRY	AVG COUNTRY AREA
17,098,242	444,128

4. GROUP BY treats null as groupable value, and that is strange

When your data set contains multiple null values, group by will treat them as a single value and aggregate for the set.

This does not conform to the standard use of null, which is never equal to anything including itself.

```
select null = null
-- returns null, not True
```

From the SQL standards guidelines in SQL: 2008

```
Although the null value is neither equal to any other value nor not equal to any other value — it is unknown whether or not it is equal to any given value — in some contexts, multiple null values are treated together; for example, the <group by> treats all null values together.
```

5. MySQL allows you to GROUP BY without specifying all your non-aggregate columns

In MySQL, unless you change some database settings, you can run queries like only a subset of the select dimensions grouped, and still get results. As an example, in MySQL this will return an answer, populating the state column with a randomly chosen value from those available.

```
select
  country
  , state
  , count(*)
from
  countries
group by
  country
```

`group by` is a commonly used keyword, but hopefully you now have a clearer understanding of some of its more nuanced uses.

Now that you've learned more about using SQL to translate your data into insights and actions faster, it's time to look at some overall tips to optimize your SQL database for analysis. Periscope Data helps customers run more than 17 million queries and share more than 30,000 dashboards every day, so our best practices have been finely tuned at other companies that operate like yours.

10 Key SQL Guidelines

There are a lot of decisions to make when creating new tables and data warehouses. Some that seem inconsequential at the time end up causing you and your users pain for the life of the database.

We've worked with thousands of people and their databases and, after countless hours of reading and writing queries, we've seen almost everything. Here are our top 10 rules for creating pain-free schemas.

1. Only use lowercase letters, numbers and underscores

Don't use dots, spaces or dashes in database, schema, table or column names. Dots are for identifying objects, usually in the `database.schema.table.column` pattern.

Having dots in names of objects will cause confusion. Likewise, using spaces in object names will force you to add a bunch of otherwise unnecessary quotes to your query:

```
select "user name" from events
-- vs
select user_name from events
```

Queries are harder to write if you use capital letters in table or column names. If everything is lowercase, no one has to remember if the **users** table is **Users** or **users**.

And when you eventually change databases or replicate your tables into a warehouse, you won't need to remember which database is case-sensitive, as only some are.

2. Use simple, descriptive column names

If the **users** table needs a foreign key to the `packages` table, name the key `package_id`. Avoid short and cryptic names like `pkg_fk`; others won't know what that means. Descriptive names make it easier for others to understand the schema, which is vital to maintaining efficiency as the team grows.

Don't use ambiguous names for polymorphic data. If you find yourself creating columns with an `item_type` or `item_value` pattern, you're likely better off using more columns with specific names like `photo_count`, `view_count`, `transaction_price`.

This way, the contents of a column are always known from the schema, and are not dependent on other values in the row.

```
select sum(item_value) as photo_count
from items
where item_type = 'Photo Count'
-- vs
select sum(photo_count) from items
```

Don't prefix column names with the name of the containing table. It's generally unhelpful to have the users table contain columns like `user_birthday`, `user_created_at` or `user_name`.

Avoid using reserved keywords like `column`, `tag` and `user` as column names. You'll have to use extra quotes in your queries and forgetting to do so will get you very confusing error messages. The database can wildly misunderstand the query if a keyword shows up where a column name should be.

3. Use simple, descriptive table names

If the table name is made up of multiple words, use underscores to separate the words. It's much easier to read `package_deliveries` than `packagedeliveries`.

And whenever possible, use one word instead of two: `deliveries` is even easier to read.

```
select * from packagedeliveries
-- vs
select * from deliveries
```

Don't prefix tables to imply a schema. If you need the table grouped into a scope, put those tables into a schema. Having tables with names like `store_items`, `store_transactions`, `store_coupons`, like prefixed column names, is generally not worth the extra typing.

We recommend using pluralized names for tables (e.g. `packages`), and pluralizing both words in the name of a join table (e.g. `packages_users`). Singular table names are more likely to accidentally collide with reserved keywords and are generally less readable in queries.

4. Have an integer primary key

Even if you're using UUIDs or it doesn't make sense (e.g. for join tables), add the standard `id` column with an auto-incrementing integer sequence. This kind of key makes certain analyses much easier, like selecting only the first row of a group.

And if an import job ever duplicates data, this key will be a life-saver because you'll be able to delete specific rows:

```
delete from my_table
where id in (select ...) as duplicated_ids
```

Avoid multi-column primary keys. They can be difficult to reason about when trying to write efficient queries, and very difficult to change. Use an integer primary key, a multi-column unique constraint and several single-column indexes instead.

5. Be consistent with foreign keys

There are many styles for naming primary and foreign keys. Our recommendation, and the most popular, is to have a primary key called `id` for any table `foo`, and have all foreign keys be named `foo_id`.

Another popular style uses globally unique key names, where the `foo` table has a primary key called `foo_id` and all foreign keys are also called `foo_id`. This can get confusing or have name collisions if you use abbreviations (e.g. `uid` for the `users table`), so don't abbreviate.

Whatever style you choose, stick to it. Don't use `uid` in some places and `user_id` or `users_fk` in others.

```
select *
from packages
  join users on users.user_id = packages.uid
-- vs
select *
from packages
  join users on users.id = packages.user_id
-- or
select *
from packages
  join users using (user_id)
```

And be careful with foreign keys that don't obviously match up to a table. A column named `owner_id` might be a foreign key to the **users** table, or it might not. Name the column `user_id` or, if necessary, `owner_user_id`.

6. Store datetimes as datetimes

Don't store Unix timestamps or strings as dates: convert them to `datetimes` instead. While SQL's date math functions aren't the greatest, doing it yourself on timestamps is even harder. Using SQL date functions requires every query to involve a conversion from the timestamp to a `datetime`:

```
select date(from_unixtime(created_at))
from packages
-- vs
select date(created_at)
from packages
```

Don't store the year, month and day in separate columns. This will make every time series query much harder to write, and will prevent most novice SQL users from being able to use the date information in this table.

```
select date(created_year || '-'
  || created_month || '-'
  || created_day)
-- vs
select date(created_at)
```

7. UTC, always UTC

Using a timezone other than UTC will cause endless problems. Great tools (including Periscope) have all the functionality you need you convert the data from UTC to your current timezone. In Periscope, it's as easy as adding `:pst` to convert to from UTC to Pacific Time:

```
select [created_at:pst], email_address
from users
```

The database's time zone should be UTC, and all datetime columns should be types that strip time zones (e.g. **timestamp without time zone**).

If your database's time zone is not UTC, or you have a mix of UTC and non-UTC datetimes in your database, time series analysis will be a lot harder.

8. Have one source of truth

There should only ever be one source of truth for a piece of data. Views and rollups should be labeled as such. This way consumers of that data will know there is a difference between the data they are using and the raw truth.

```
select *
from daily_usage_rollup
```

Leaving legacy columns around like `user_id`, `user_id_old` or `user_id_v2` can become an endless source of confusion. Be sure to drop abandoned tables and unused columns during regular maintenance.

9. Prefer tall tables without JSON columns

You don't want to have super-wide tables. If there's more than a few dozen columns and some of them are named sequentially (e.g. `answer1`, `answer2` or `answer3`), you're going to have a bad time later.

Pivot the table into a schema that doesn't have duplicated columns — this schema shape will be a lot easier to query. For example, getting the number of completed answers for a survey:

```
select
  sum(
    (case when answer1 is not null
      then 1 else 0 end) +
    (case when answer2 is not null
      then 1 else 0 end) +
    (case when answer3 is not null
      then 1 else 0 end)
  ) as num_answers
from surveys
where id = 123
-- vs
select count(response)
from answers
where survey_id = 123
```

For analysis queries, extracting data from JSON columns can greatly degrade a query's performance. While there are a lot of great reasons to use JSON columns in production, there aren't for analysis. Aggressively schematize JSON columns into the simpler data types to make analysis a lot easier and faster.

10. Don’t over-normalize

Dates, zip codes and countries don’t need their own tables with foreign key lookups. If you do that, every query ends up with a handful of the same joins. It creates a lot of duplicated SQL and a lot of extra work for the database.

```
select
  dates.d,
  count(1)
from users
  join dates on users.created_date_id = dates.id
group by 1
-- vs
select
  date(created_at),
  count(1)
from users
group by 1
```

Tables are for first class objects that have a lot of their own data. Everything else can be additional columns on a more important object.

Conclusion

Better schemas await!

Armed with these rules, your next table or warehouse will be easier to query for both you and new team members as you expand.

If your company is ready to start building a sound data analytics infrastructure, Periscope Data can help. With these tips, you're more than prepared to put a solid framework in place for data analytics that will create immediate value for your organization.

We've helped more than 1000 customers of all sizes connect their data and produce valuable insights that lead to better decisions. If you have any questions about how your team can use Periscope for data analytics, request a contact and one of our experts will reach out to you soon.