The SQL Starter Kit: Basics, Tips, and Pasteable Snippets

Jumping into the World of SQL

Welcome to the world of data! You're living in it right now. Since you've decided to read this paper, you already know that data is everywhere, there's tons of it, and it's the key to unlocking your company's potential and owning the future.

What you might not know (but can probably guess from quickly googling "SQL," often pronounced like "sequel"), is that SQL is one of the fundamental ways in which people interact with their data. Simply put, it's a query language. Programmers use it to talk to the database, find specific pieces of information, and then serve those up or use the pieces for further analysis. If it's in your database, you can get to it with SQL.

As an example, let's say you have a list of all your users and all the widgets they've created since the beginning of time. That's too much data. It's data you want and need, but taken as an aggregate, it's voluminous enough to be meaningless. However, if you want to see widgets created by quarter or by user or by user and quarter, you can do that with SQL. It's a great way to pull out info on vital KPIs (Key Performance Indicators) and see which patterns emerge.

But *how* you use that SQL is a different matter. We're going to go over a bunch of useful SQL techniques and best practices that'll help you get the data you need more quickly.

Here's the rundown of what we'll cover:

- Joining the first row in SQL
- Choosing one row per group
- The mighty GROUP BY
- The top 10 SQL guidelines you should know

Joining the First Row in SQL

Joining the first row is a useful tactic when dealing with highly recent information. Let's go back to that set of hypothetical users creating widgets. We want to create a list of users with their most recent widgets. We've got one table with our users and another with their widgets.

Each user has multiple widgets, but we're only concerned with the most recent one. The primary key for users is user.id and the corresponding foreign key for widgets is widgets.user_id.

SQL is a very versatile language, so there are several ways to join the first row. We'll dig into four different techniques:

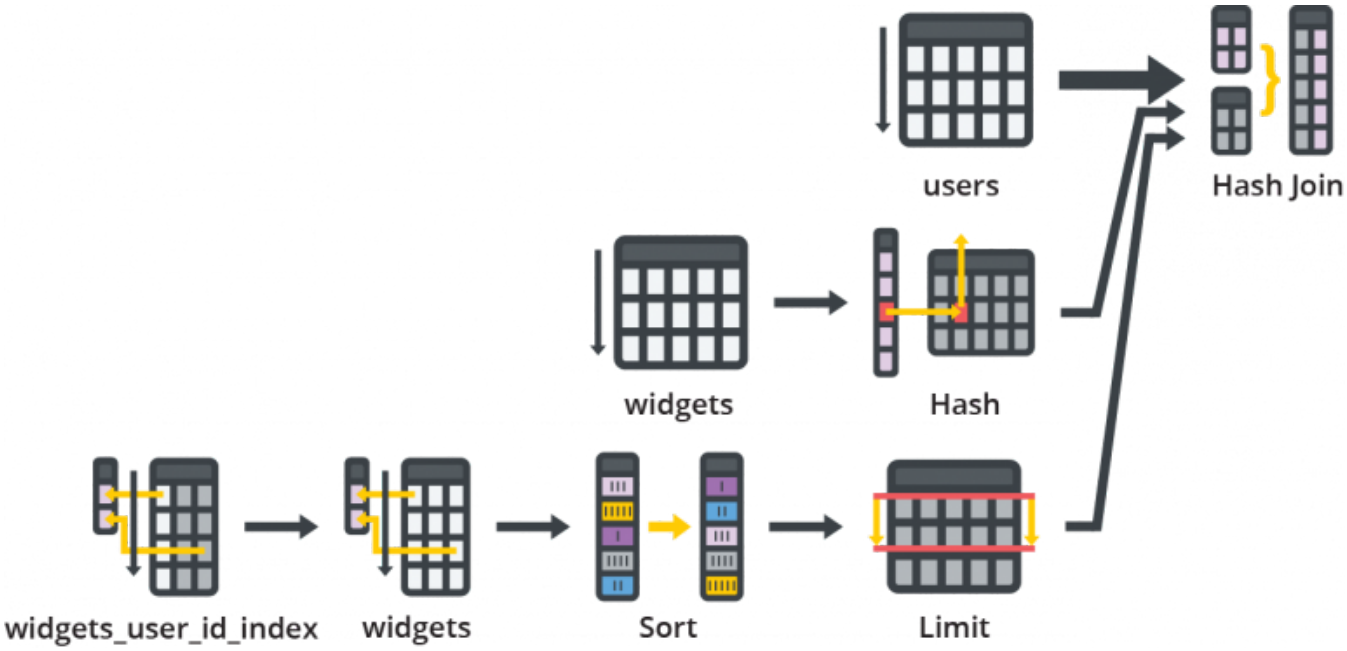
Using Correlated Subqueries When the Foreign Key is Indexed

As the name implies, correlated subqueries depend on an outer query. The subquery runs once per each row of the outer query. (You can think of it as a "for loop" for SQL.)

The SQL looks like this:

```
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
)
```

Check out the “widgets.user_id = users.id” clause inside the subquery. This tells the system to query the **widgets** table once per **user** row, then pick out the most recent **widget row** for that user. This is super efficient if **user_id** is indexed and you don’t have a ton of users.

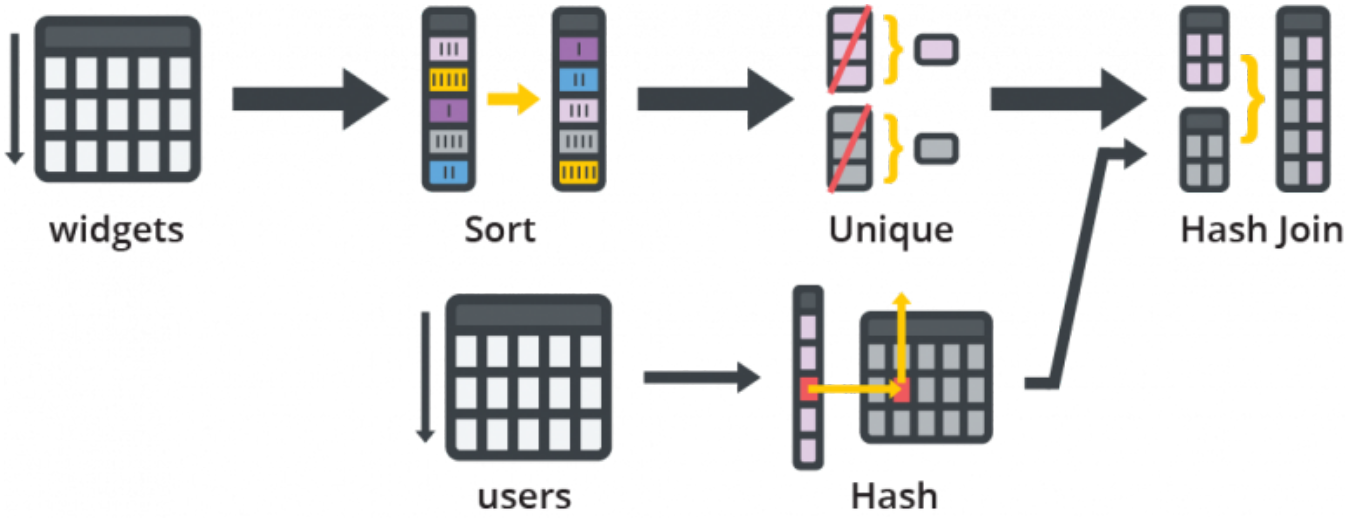


Using Complete Subqueries (When You Don’t Have Indexes)

If the foreign key isn’t indexed, correlated subqueries can run into trouble, since each one will require a full table scan, making them less efficient. It’s possible to speed this process up by rewriting the query to use a single subquery that only scans the widgets table once. It looks like this:

```
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
```

The subquery pulls of list of the most recent widgets, one per user, which we then join to the users table to create our list:



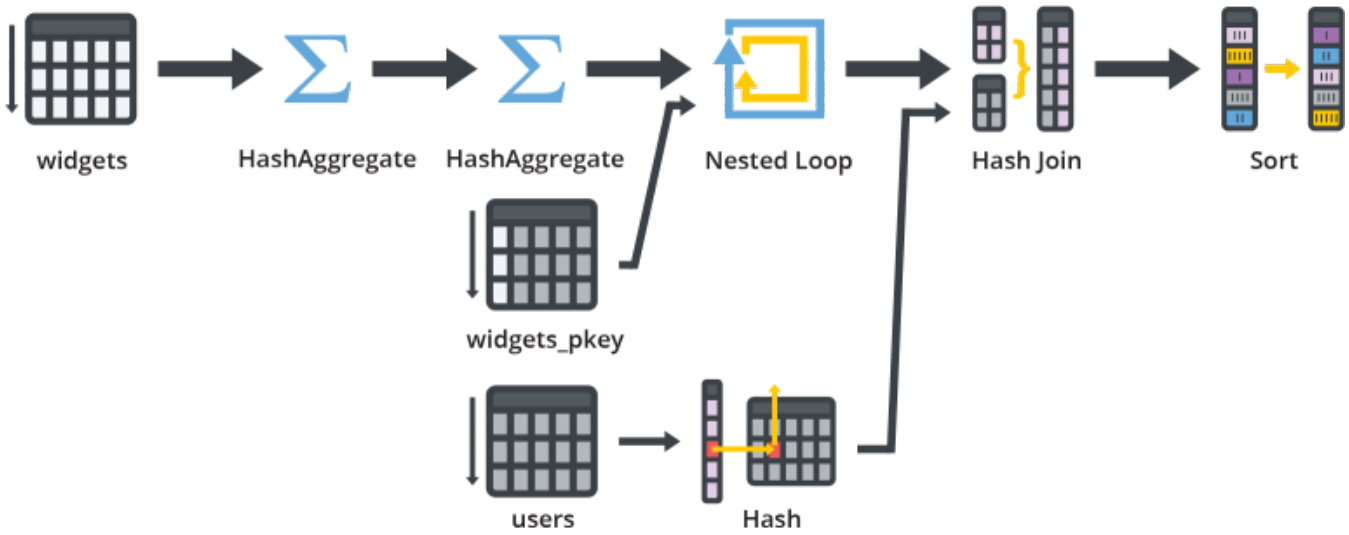
The Postgres DISTINCT ON clause streamlines querying the dataset to only return one widget per user_id. However, not every database supports DISTINCT ON. That's not the end of the world, though! You have a couple of great options to deal with this snag.

Option 1: Using Nested Subqueries with an Ordered ID Column

For this example, we're using a setup where the most recent row always has the highest id value. This makes it possible to cheat a bit in our nested subquery in this way:

```
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
```

How this works is the system pulls the list of IDs representing the latest widget each user created. Next, we filter the widgets table according to those IDs. This acts like DISTINCT ON, because sorting by id and created_at are equivalent.



Option 2: Getting More Control with Window Functions

Let's say that you don't have an id column in your table, or else its min and max values don't give you the most recent row. Using row_number with a window can get you the results you need. It takes a few more steps, but it's very versatile:

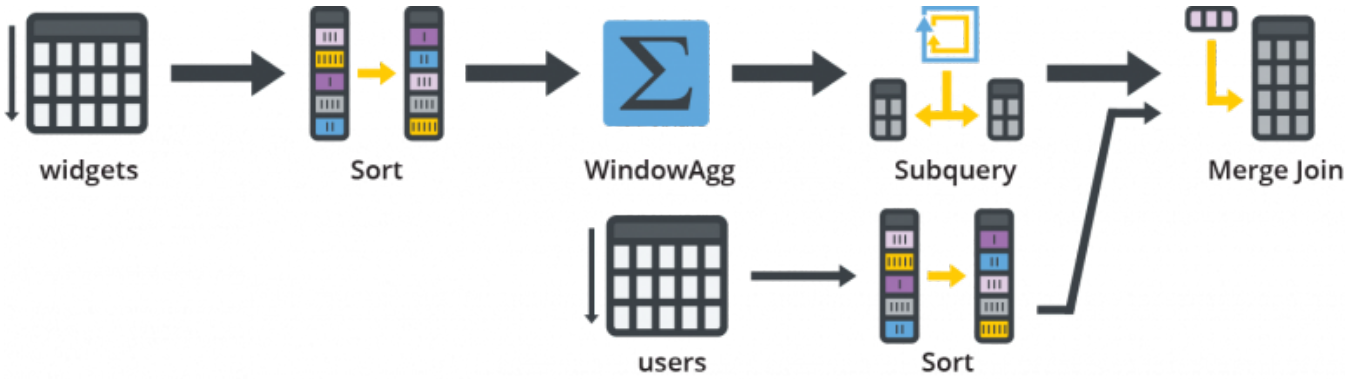
```
select * from users join (
 select * from (
 select *, row_number() over (
 partition by user_id
 order by created_at desc
 ) as row_num
 from widgets
 ) as ordered_widgets
 where ordered_widgets.row_num = 1
) as most_recent_user_widget
on users.id = most_recent_user_widget.user_id
order by users.id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

Pay special attention to this section:

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

We're doing a few things at once. First off, "over (partition by user_id order by created_at desc" is used to specify a sub-table (the window) around the user_id attribute and sorts those windows by created_at desc. row_number() and comes back with each row's position within the window. The end result is that the first widget for each user_id now is now in row_number 1.

Meanwhile, we direct the outer subquery to select only the rows with a row_number of 1 (you can also use this kind of query to get the second, third, or tenth rows instead).

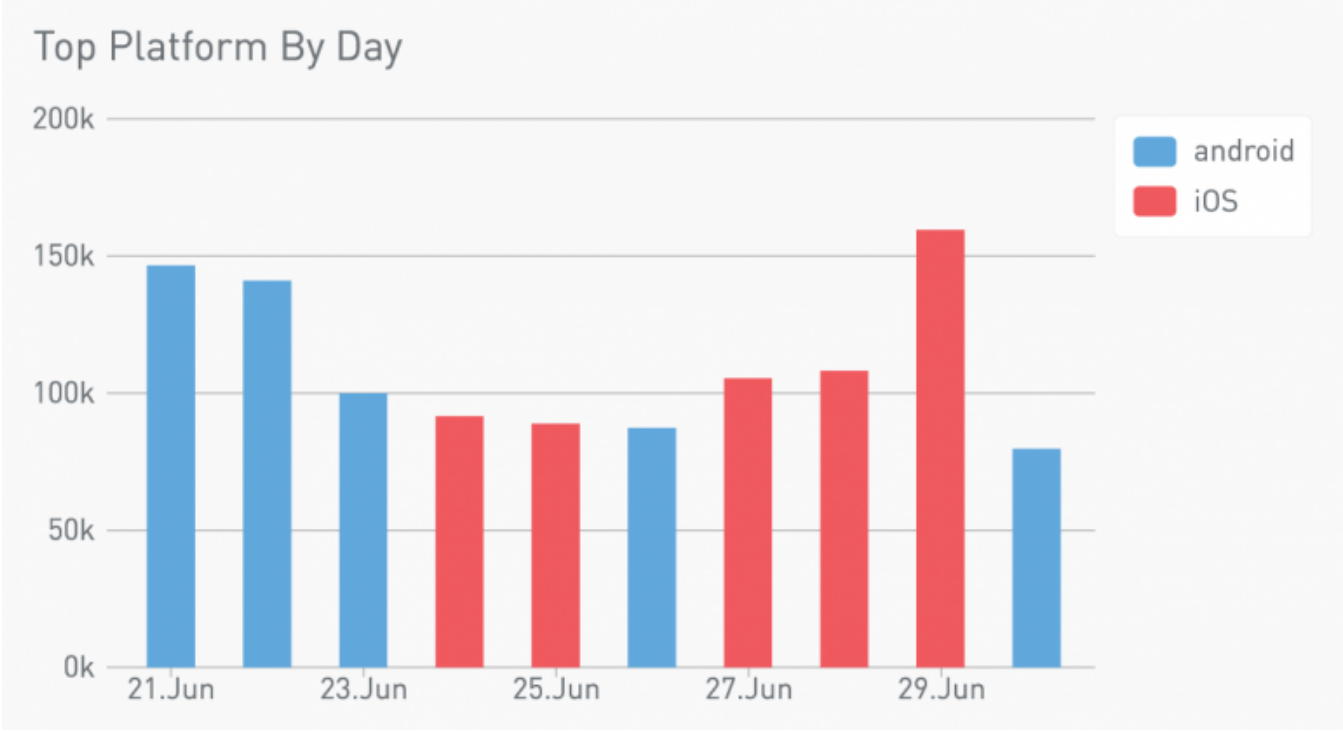


As we continue, we'll talk about other ways to pick out specific bits of information. The ability to select one row per group lets you cut through the noise and easily pull out mins and maxes for your analyses, speeding up the process and returning your insights at speed.

Selecting One Row Per Group

Quickly and easily grabbing the top result is a must-have when putting together KPI reports. Do you get more customers from LinkedIn ads or Twitter ads? Did more people buy the red pumps or the black ones? Do users access your app on Android or iOS?

When trying to quickly get the top result from your data, primary and foreign keys aren't super useful. Let's say we're trying to build a chart like this, tracking the top platform (Android or iOS) for mobile app gameplays per day:



The first place to gather data for this widget is the gameplays table, where we can get the daily counts by platform:

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

The result is a table like this:

```
    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
```

However, all we want is one row per date with the highest count, which looks 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
```

Using Postgres and Redshift

Window functions with Postgres and Redshift can also do this pretty easily. Using the row_number() function, partitioned by date, inside an inner query, we'll then filter it to row_num=1 in the outer query. This returns the first record per group.

Here's the function numbering each of the rows:

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

Plug that into an inner query and we'll get the results we want:

```
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

Without window functions, we need to take a slightly different tack in MySQL. Using group_concat, we turn the “platform” column into a comma-separated string of results, ordered by count.

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

That will return all the platform numbers, with the highest-performing one first. Here’s the full query:

```
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
```

The result?

```
    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
```

Okay! So we've got the info we want, with the highest counts first. Now we can use "substring_index" (this pulls the first word before the comma) to get just the top platform:

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

It's a bit of the long way around, but now we can use max(ct) to get the count of each platform:

```
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 was a long way to go to pull a pretty simple bit of information (the right analytics and BI platform can massively simplify this), but it works!

Next up, we'll use group_by to aggregate data, which is a vital process when it comes to gathering up the data you need for a KPI dashboard.

The Mighty GROUP BY

GROUP BY a heavy hitter of the SQL world. It collapses a field to its distinct values. Often people use GROUP BY in aggregations to display one value per grouped field (or combination of fields). There's a lot we can do with this!

Let's say we're working with this 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

GROUP BY allows us to aggregate a bunch of different types of information. Want the number of countries per continent? Here's how to get that:

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

We get this table:

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

First set of tips when using group by:

- GROUP BY X tells the system to place all items with the same value for X in the same row
- GROUP BY X,Y tells the system to put everything with the same value for X and Y in the same row (makes sense)

Here are five cool things to remember about GROUP BY:

1. Filter aggregations via the HAVING clause

The “where” clause can’t be used on aggregations. This code will return an error:

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

This returns an error because the system tries to evaluate the “where” statement before the aggregation even takes place! Alternatively, using “having” after the GROUP BY function lets you filter the returned data in an aggregated column, like so:

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

"HAVING" lets you return the aggregate filtered results.

2. Using GROUP BY with a column number

You can often use a column number instead of a column name when grouping. An alternate version of our first column could be written as:

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

Using ordered numbers, this is called “ordinal notations.” It was the SQL standard until the 1980s, but its current usage is the subject of some debate. Why?

- It can be less explicit, reducing legibility for some users.
- Brittleness: if a query select statement has a column name changed and continues to run, it can produce an unexpected result.

However, there are some benefits:

- SQL coders often select dimensions first, then aggregates, which can make reading SQL more predictable.
- Ordinal notation is easier to maintain on large queries. When writing long ETL statements, GROUP BY statements can become many lines long and can be hard to maintain.
- There are databases that allow the use of an aliased column in the GROUP BY. In this situation, a long case statement can be grouped without redoing the whole full statement in the GROUP BY clause. Ordinal positions can be cleaner and keep you from accidentally grouping by an alias matching a column name in the underlying data. This example code will come back with 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

This code will not, though. Instead, it'll segment by the **base table's** currency field and accept 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

Commonly you see the use of ordinal positions for ad hoc work and column names for production code. This can make things clearer for future users who might need to change your code.

3. The Implicit Group By

When you are aggregating an entire table, also known as a <grand total> in SQL standards documentation, there is an implied GROUP BY, like so:

```
-- 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 will group null values. Beware!

If you've got multiple null values in your dataset, GROUP BY will treat them all as a single value, aggregating them for the set. (This is out of keeping with the standard use of "null," which is never supposed to equal anything, including itself.).

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

The SQL standards guidelines in SQL:2008 says this:

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. You can use GROUP BY with MySQL without specifying all non-aggregate columns

In MySQL, barring some changes you might make to the database settings, running a query with only a subset of selected dimensions grouped will still return results. For example, this code will come back with the state column and a randomly chosen value from the available values:

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

This rounds out our look at some of the vital uses of the mighty GROUP BY.

Next up, let's dig into some SQL best practices.

Top 10 SQL Guidelines

Whether you're putting up a building or creating the next generation of applications and services backed by tons of amazing data, the decisions you make when creating the foundation *matter*. After years in the analytics world, we've come with these 10 guidelines to help you create easy-to-use schemas that will stand the test of time. Let's dive in.

1. Stick to lowercase letters, numbers, and underscores only

Make sure not to use dots (periods), spaces, or dashes for database, schema, table, or column names. Dots are usually used when identifying objects, in this pattern: database.schema.table.column

Putting dots in object names will make your database confusing for users. Relatedly, underscores make writing your queries easier by keeping items together; without underscores, you need to put quotes around your object names, like so:

```
select “user name” from events
-- vs.
select user_name from events
```

The rule against capital letters in table or column names has to do with specificity: you have to write your queries *exactly* as the terms appear in the database. Basically, if you use capital letters anywhere, you have to remember to use them *everywhere, every time*. Omitting caps just makes it easier for everyone.

2. Make column names simple and descriptive

Let’s say your table, users, needs a foreign key for the packages table. A good name for this could be package_id. Something like pkg_fk might seem like an even shorter, more elegant solution, and maybe for you it would be. But you’re not the only person who’ll be using your database. Descriptive names help everyone who uses the database get the most out of it.

Additionally, if you’re dealing with polymorphic data, make sure the names aren’t ambiguous. If you’re building a table and find yourself creating columns with names like item_type or item_value, take the extra time to break out the attributes with specific names (photo_count, view_count, etc.)

This keeps the column’s contents easily understood from the schema level, and independent from the other values in a row:

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

Prefixes referencing the name of the table are unnecessary, e.g., user_birthday, user_created_at, etc.

Using "reserved" keywords like column, tag, and user in column names will also come back to bite you later, as you'll constantly have to put quotation marks in your queries, or else return a baffling error message.

3. Make table names simple and descriptive

Use as many words as you need and separate words with underscores (package_deliveries vs. packagedeliveries). If you can say it with one word, however, do that: deliveries is the easiest to read.

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

Don't use prefixes with table names to imply a schema. If you need to group a table into a scope, put it into a schema! Table names with shared prefixes (store_items, store_transactions, etc.) are more trouble than they're worth.

Pro Tip: Use pluralized names for tables (packages vs. package) and both pluralized terms for a join table (packages_users). Singular table names can accidentally collide with reserved keywords and are often less readable in queries.

4. Have an integer primary key

Adding the standard id column with an auto-incrementing integer sequence is a useful habit to get into, even if you're using UUIDs. Using an integer primary key simplifies certain analyses easier (like selecting the first row, etc.).

Additionally, this key allows you to delete specific rows (e.g., if you have an import job that duplicates data):

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

Avoid multi-column primary keys. They are difficult to change and can make extra work for you when trying to write efficient queries. Instead, combine an integer primary key with a multi-column unique constraint and several single-column indexes.

5. Be consistent with foreign keys

Consistency is key: while there are many ways to name your primary and foreign keys, our recommendation is to stick to a primary key called id for any table (foo), and have all foreign keys be called foo_id.

Alternatively, some people prefer using globally unique key names, so if you have a table foo, with a primary key of foo_id, then all foreign keys are also called foo_id. Using abbreviations with this naming convention can be confusing (e.g., uid for the users table), so avoid abbreviations.

The most important thing is to stick to whatever naming convention you choose. (For example, 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)
```

Lastly, be careful with foreign keys that don't obviously match with a table. A column called owner_id could be a foreign key to the users table, but then again maybe not. Calling this column users_id or owner_user_id is much more specific.

6. Use datetimes

When storing Unix timestamps as strings or dates, make sure to use the datetime format. SQL's date math functions leave something to be desired and using SQL to convert a timestamp to a datetime for every query is a hassle you don't want:

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

Also: don't store time information in different columns (day, month, year, etc.). This makes your time series data a chore to write and can hamstring newer users.

7. Stick with UTC

UTC, or Coordinated Universal Time is your friend when it comes to storing datetimes. Your BI tool can easily translate a UTC timestamp to whatever time zone you're in just by adding the appropriate wording to your query (this is for Pacific Time):

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

Mixing UTC and non-UTC datetimes in your database will make time series analysis much harder, so make sure your database's time zone is UTC and all datetime columns are without time zones.

8. Adhere to a single source of truth

Make sure there's only one source of truth for any given piece of data. Label all views and rollups accordingly. This will show consumers of the data that there is a difference between the data that they are using and the raw truth. Example:

```
select *
from daily_usage_rollup
```

Legacy columns like user_id, user_id_old, or user_id_old can create confusion. Make sure that abandoned tables and unused columns are dropped during regular maintenance.

9. Favor tall tables without JSON columns

Avoid super wide tables. Having more than a few dozen columns, with some named sequentially (e.g., answer_1, answer_2, answer_3, etc.) will cause trouble for you later.

Instead, pivot the table into a schema without duplicated columns. This schema shape will be much easier to query. Here's some sample code for 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
```

Extracting data from JSON columns can slow down query performance. There are some reasons to use JSON columns in production, but not for analysis. Instead, schematize JSON columns into simpler data types to simplify and speed up analysis.

10. Don't over-normalize

You don't have to create special tables with foreign key lookups for stuff like dates, zip codes, and countries. Doing that causes every query to have a lot of the same joins, creating duplicated SQL and 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
```

Stick to using tables as first-class objects with lots of their own data and make everything else additional columns on a more important object.

Ready to Run

If you want to get the big things right, you have to get the little things right too. Armed with this slew of SQL tips and tricks, you're ready to make your next table or data warehouse easier to query for yourself and your team members.

The right analytics and BI solution can take your data to the next level. Whether you're looking to empower front-line users to perform ad hoc analyses without help from IT or you want a way to create and embed powerful analytic apps, Sisense can help you get more out of your data and build the future you want to be a part of.

See How Sisense Supercharges Your Company or Product.