PERISCOPE DATA PRESENTS

Speed up Your SQL

Your SQL Queries Are Holding You Back

As an analyst, your work is important! Slow queries can keep you from efficiently surfacing and acting on insights. This could be delaying better business outcomes, as well as your professional advancement.

Our customers frequently ask us how they can speed up their queries; as a result, we've built up a lot of expertise around optimizing SQL for faster queries.

We've written this guide to help the data analyst community sharpen their skills and speed up their analyses. We'll be focusing on four main opportunities: pre-aggregated data, avoiding joins, avoiding table scans, and approximations. Tactics vary from beginner to advanced to make sure any reader can benefit.

Beyond this guide, we write about SQL—a lot. Check out our blog for the latest articles. Want us to cover something? Give us a shout at hello@periscope.com.

Pre-Aggregating Data

When running reports, it's common to need to combine data from different tables for a query. Depending upon where you do this in your process, you could be looking at a very slow query. In the simplest terms, an **aggregate** is a simple summary table that can be derived by performing a GROUP BY SQL query.

Aggregations are usually precomputed, partially summarized data, stored in new aggregated tables. To speed up your queries, you should aggregate as early in the query as possible.

In this section, we'll look at a few tactics you can use to speed up your aggregations, including:

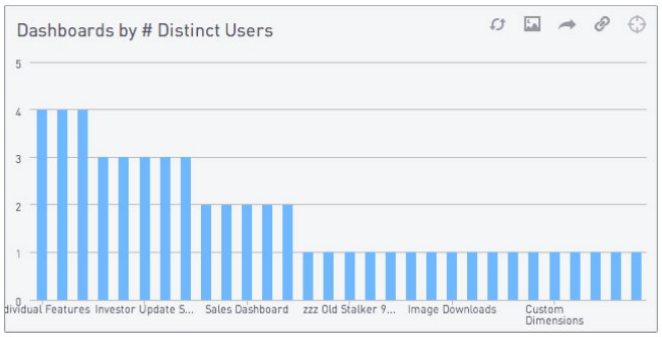
- Aggregating data
- Reducing the data set
- Working with materialized views

Speeding up Count Distinct

Count distinct is well known for slowing down queries, so it's an obvious choice for demonstrating aggregating. Let's start with a simple query we run all the time: Which dashboards do most users visit?

```
select
  dashboards.name
  , count(distinct time_on_site_logs.user_id) ct
from time_on_site_logs
join dashboards
  on time_on_site_logs.dashboard_id = dashboards.id
group by name
order by ct desc
```

In Periscope Data, this would give you a graph like this:



For the sake of example, let's assume the indices on user_id and dashboard_id are in place, and there are lots more log lines than dashboards and users.

Even with just 10 million rows, this query takes 48 seconds. Let's take a closer look to understand why.

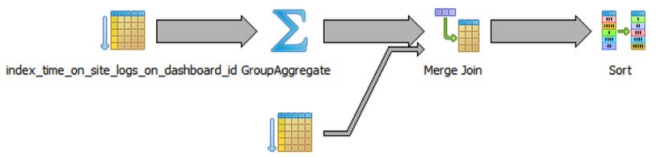
The database is iterating over all the logs and all the dashboards, then joining them, then sorting them—all before getting down to the real work of grouping and aggregating!

Aggregate, Then Join.

Anything that comes after grouping and aggregating will be a lot faster, because the data size is much smaller. Since we don't need dashboards.name in the group-and-aggregate, we can have the database do the aggregation before the join.

```
select
  dashboards.name
  , log_counts.ct
from dashboards
join (
  select
    dashboard_id
    , count(distinct user_id) as ct
  from time_on_site_logs
  group by dashboard_id
) as log_counts
  on log_counts.dashboard_id = dashboards.id
order by log_counts.ct desc
```

This query runs in 20 seconds—a 2.4x improvement! Let's see why:



As you can see, the group-and-aggregate comes before the join. As a bonus, we can take advantage of the index on the time_on_site_logs table.

Reduce the Data Set

We can do better. By doing the group-and-aggregate over the whole logs table, we made our database unnecessarily process a lot of data. Count distinct builds a hash set for each group—in this case, each dashboard_id—to keep track of which values have been seen in which buckets.

```
select
  dashboards.name
  , log_counts.ct
from dashboards
join (
  select
    distinct_logs.dashboard_id
    , count(1) as ct
  from (
    select
      distinct dashboard_id
      , user_id
    from
      time_on_site_logs
  ) as distinct_logs
  group by distinct_logs.dashboard_id
) as log_counts
  on log_counts.dashboard_id = dashboards.id
order by log_counts.ct desc
```

We've taken the inner count distinct and group and broken it up into two pieces. The inner piece computes distinct pairs (i.e., dashboard_id, user_id). The second piece runs a simple group and count over them. And, as always, the join is last.

This query clocks in at 0.7 seconds! That's a 28x increase over the previous query, and 68x increase over the original query.

As you know, data size and shape matters a lot. These examples benefit a lot from a relatively low cardinality. There are a small number of distinct pairs compared to the total amount of data. With more **unique pairs**—which is to say, the more data rows that must be grouped and counted separately—the opportunity for easy speedups decreases.

Next time count distinct is taking all day, try a few subqueries to lighten the load.

Working With Materialized Views

The best way to make your SQL queries run faster is to have them do less work. And a great way to do less work is to query a materialized view that's already done the heavy lifting.

Materialized views are particularly nice for analytics queries for a few reasons:

1. Many queries do math on the same basic atoms.
2. The data changes infrequently—often as part of daily ETLs.
3. ETL jobs provide a convenient home for view creation and maintenance.

Redshift doesn't yet support materialized views out of the box; however, with a few extra lines in your import script—or with a solution like Periscope Data—creating and maintaining materialized views as tables is a breeze.

Example: Calculating Lifetime Daily ARPU

Average revenue per user (ARPU) is a common metric that shows the changes in how much money you're making per user over the lifetime of your product.

$$\text{Lifetime ARPU } (\textit{Date}) = \frac{\text{Sum of purchases up to } \textit{Date}}{\text{Unique user count up to } \textit{Date}}$$

To calculate this, we'll need a purchases table and gameplays table, along with the lifetime accumulated values for each date.

Here's the SQL for calculating lifetime gameplays:

```
with
lifetime_gameplays as (
  select
    dates.d
    , count(distinct gameplays.user_id) as count_users
  from (
    select distinct date(created_at) as d
    from gameplays
  ) as dates
  inner join gameplays
    on date(gameplays.created_at) <= dates.d
  group by d
),
```

The range join in the correlated subquery lets us recalculate the distinct number of users for each date.

Here's the SQL for lifetime purchases in the same format:

```
lifetime_purchases as (
  select
    dates.d
    , sum(price) as sum_purchases
  from (
    select
      distinct date(created_at) as d
    from purchases
  ) as dates
  inner join purchases
    on date(purchases.created_at) <= dates.d
  group by d
)
```

Now that the setup is done, we can calculate lifetime daily ARPU:

```
with
lifetime_gameplays as (...),
lifetime_purchases as (...)

select
  lifetime_gameplays.d as date
  , round(
      lifetime_purchases.sum_purchases /
      lifetime_gameplays.count_users
    , 2) as arpu
from lifetime_purchases
inner join lifetime_gameplays
  on lifetime_purchases.d = lifetime_gameplays.d
order by lifetime_gameplays.d
```

That's a huge query, taking minutes to run on a database with two billion gameplays and three billion purchases. That's way too slow, especially if we want to quickly slice by dimensions like what platform the game was played on. Plus, similar lifetime metrics will need to recalculate the same data over and over again!

Easy View Materialization in Redshift

Fortunately, we wrote our query in a format that makes it clear which parts can be extracted into materialized views: lifetime_gameplays and lifetime_purchases.

We'll fake view materialization in Redshift by creating tables, and Redshift makes it easy to create tables from snippets of SQL:

```
create table lifetime_purchases as (
  select
    dates.d,
    sum(price) as sum_purchases
  from (
    select distinct date(created_at) as d
    from purchases
  ) as dates
  inner join purchases
    on date(purchases.created_at} <= dates.d
  group by d
)
```

Do the same thing for lifetime_gameplays, and and calculating lifetime daily ARPU now takes less than a second to complete!

Remember that in order to keep you data fresh, you'll need to drop and recreate these tables every time you upload data to your Redshift cluster.

With Periscope Data, your dashboards and charts are automatically kept up-to-date.

Avoiding Joins

Depending on how your database scheme is structures, you're likely to have data required for common analysis queries in different tables. **Joins** are typically used to combine data from disparate tables. They're powerful, but not without their downside.

Your database has to scan each table that's joined and figure out how each row matches up. This makes joins expensive when it comes to query performance. This can be mitigated through smart join usage, but your queries would be even faster if you could completely avoid joins.

In this section, we'll cover two ways you can avoid joins:

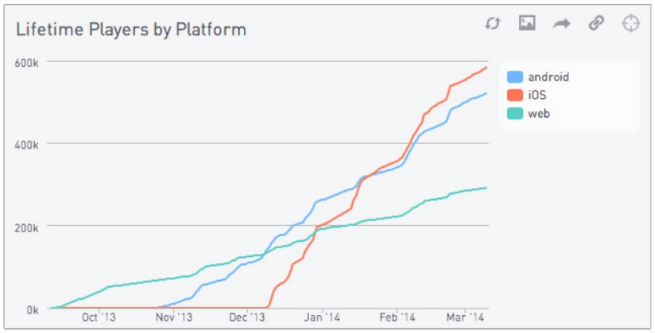
1. Generate series
2. Window functions

Generate Series

Calculating Lifetime Metrics

Lifetime metrics are a great way to get a long-term sense of the health of business. They're particularly useful for seeing how healthy a segment is by comparing it to others over the long term.

For example, here's a fictional graph of lifetime game players by platform:



We can see that iOS is our fastest-growing platform, quickly surpassing the others in lifetime players despite being launched months later!

Metrics like these can be incredibly slow, since they're typically calculated with distinct counts over asymmetric joins. Let's look at how we can calculate these metrics in milliseconds—not minutes.

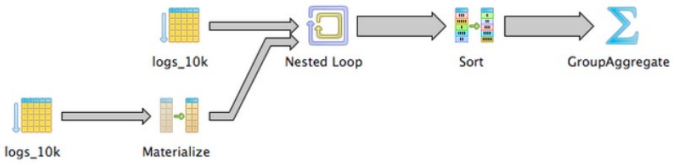
Self-Joining

The easiest way to write the query is to join gameplays onto itself. The first gameplays, which we name dates, provides the dates that will go on our x-axis. The second gameplays, called plays, supplies every game play that happened on or before each date.

The SQL used here is nice and simple.

```
select
  date(dates.created_at)
  , plays.platform
  , count(distinct plays.user_id)
from gameplays dates
join gameplays plays
  on plays.created_at <= dates.created_at
group by 1, 2
```

However, the query has miserable performance: 37 minutes on a 10,000-row table! Let's take a closer look to understand why:



By asymmetrically joining gameplays to itself, we're bogging down in an n^2 loop, which then leads to sorting the whole blown-out table!

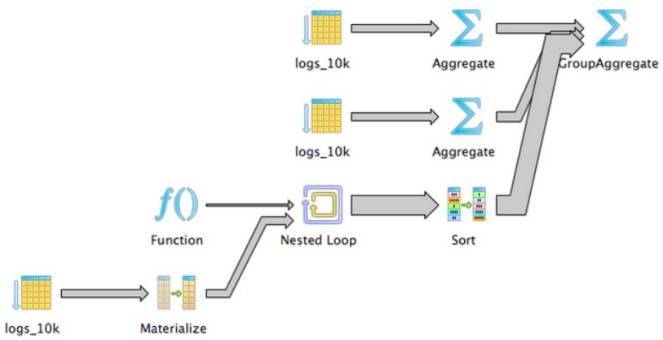
Generate Series

Astute readers will notice that we're only using the first gameplays table in the join for its dates. It conveniently has the exact same dates we need for our graph—the dates of every gameplay—but we can select those separately and then drop them into a generate_series.

The resulting SQL is still pretty simple.

```
select d
  , platform
  , count(distinct user_id)
from generate_series(
  (select date(min(created_at)) from gameplays),
  (select date(max(created_at)) from gameplays),
  '1 day'
)
join gameplays
  on created_at <= d
group by 1, 2
```

And now we go from 37 minutes to 26 seconds, an 86x speedup over the previous query! Here's why:



Don't be fooled by the scans at the top: Those are just to get the single min and max values for generate_series.

Notice that—instead of joining and nested-looping over gameplays multiplied by gameplays—we are joining with gameplays and a Postgres function, generate_series, in this case. Since the series has only one row per date, the result is a lot faster to loop over and sort.

But there's still room for improvement.

Window Functions

Every user started playing on a particular platform on one day. Once they've started, they count forever. Let's look at the first gameplay for each user on each platform.

```
select
  date(min(created_at)) dt
  , platform
  , user_id
from gameplays
group by platform, user_id
```

Now we can aggregate it into how many first gameplays there were on each platform each day.

```
with first_gameplays as (
  select
    date(min(created_at)) dt
    , platform
    , user_id
  from gameplays
  group by platform, user_id
)

select
  dt
  , platform
  , count(1) user_ct
from first_gameplays
group by dt, platform
```

We've included each user's first gameplay per platform in a with clause, and then simply counted the number of users for each date and platform.

Now that we have the data organized this way, we can simply sum the values for each platform over time to get the lifetime numbers. We need each date's sum to include previous dates, but not new dates. And we need the sums to be separated by platform. Given this, our best option is to use a window function.

Here's the full query:

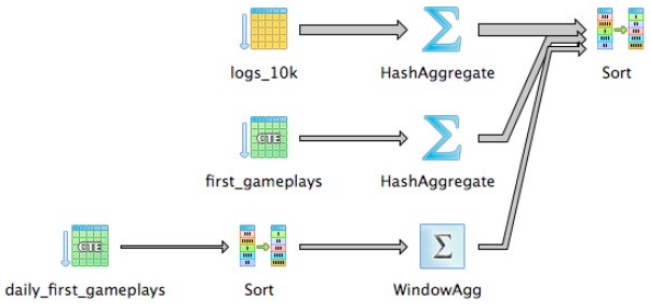
```
with
first_gameplays as (
  select
    date(min(created_at)) dt
    , platform
    , user_id
  from gameplays
  group by platform, user_id
),
daily_first_gameplays as (
  select
    dt
    , platform
    , count(1) user_ct
  from first_gameplays
  group by dt, platform
)

select
  dt
  , platform
  , sum(user_ct) over (
      partition by platform
      order by dt
      rows between unbounded preceding and current row
  ) users
from daily_first_gameplays
```

This window function is worth a closer look.

```
sum(user_ct) over (
  partition by platform
  order by dt
  rows between unbounded preceding and current row
)
```

Partition by platform means we want a separate sum for each platform. Order by dt specifies the order of the rows—in this case, by date. That's important, because rows between unbounded preceding and current row specified that each row's sum will include all previous rows (i.e., all previous dates), but no future rows. Thus it becomes a rolling sum, which is exactly what we want.

Let's examine how this query operates.



You can see we've been able to completely avoid joins. And when we're sorting, it's only over the relatively small daily_first_gameplays with clause, and the final aggregated result.

So, what effect did this have on performance? This version runs in 28 milliseconds, a 928x speedup over the previous versions, and an unbelievable 79,000x speedup over the original self-join!

Avoiding Table Scans

Most SQL queries that are written for analysis without performance in mind will cause the database engine to look at every row in the table. After all, if your goal is to aggregate rows that meet certain conditions, it makes sense to inspect each row.

But as datasets get bigger, looking at every single role just won't work in a reasonable amount of time. This is where it helps to give the query planner the tools it needs to only look at a subset of the data. In this section, we'll review a few tips you can use in queries that are scanning more rows than they should.

The Power of Indices

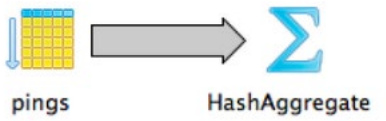
Most conversations about SQL performance begin with indices—and rightly so! They possess a remarkable combination of versatility, simplicity, and power. Before proceeding with many of the more sophisticated techniques in this book, it's worth understanding what they are and how they work.

Table Scans

Unless otherwise specified, the table is stored in the order the rows were inserted. Unfortunately, this means that queries that only use a small subset of the data—for example, queries about today's data—must still scan through the entire table looking for today's rows.

```
select
  date_trunc('hour', created_at)
from pings
where created_at > date(now())
group by 1
```

This query takes 74.7 seconds. Let's see why.



The operation on the left-hand side of the arrow is a full scan of the pings table! Because the table is not sorted by created_at, we still have to look at every row in the table to decide whether the ping happened on the selected day or not.

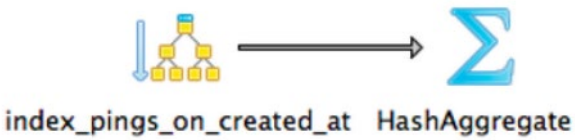
Indexing the Data

Let's improve thing by adding an index on created_at.

```
create index "index_pings_on_created_at"
  on "pings" ("created_at")
```

This will create a pointer to every row, and sort those points by the columns we're indexing on—in this case, created_at. The exact structure used to store those pointers depends on your index type, but binary trees are very popular for their versatility and log(n) search times.

Now that we have an index, let's take a closer look at how the query works:



Instead of scanning the whole table, we just search the index! By ignoring the pings that didn't happen today, the query runs in 1.7 seconds—a 44x improvement!

Caveats

Creating a lot of indices on a table will slow down inserts on that table. This is because every index has to be updated each time data is inserted. To use indices without taking a hit on speed, take a look at your most common queries and create indices designed to maximize their efficiency.

Moving Math in Where Clause

A Deceptively Simple Query

One of the most-viewed charts on our Periscope Data dashboard is a question we ask all the time: How much usage has there been today? To answer it, we've written a seemingly simple query:

```
select sum(seconds)
from time_on_site_logs
where created_at - interval '8 hour'
      > date(now() - interval '8 hour')
```

Notice the "- interval '8 hour'" operations in the where clause. Times are stored in Coordinated Universal Time (UTC), but we want "today" to mean today in Pacific Standard Time (PST).

The time_on_site_logs table has grown to 22 million rows—even with an index on created_at—slowing our query down to an average of 4.45 minutes!

Ignoring the Index

Let's run explain to see what's slowing this query down: It turns out our database is ignoring the index and doing a sequential scan!

The problem is that we're doing math on created_at, our indexed column. This causes the database to look at each row in the table, compute created_at - interval '8 hour' for that row, and compare the result to date(now() - interval '8 hour').

Moving Math to the Right

Our goal is to compute one value in advance, and let the database search the created_at index for that value. To do that, we can just move all the math to the right-hand side of the comparison.

```
select
  sum(seconds)
from
  time_on_site_logs
where
  created_at > date(now() - interval '8 hour')
  + interval '8 hour'
```

This query runs in a blazing 85 milliseconds! With a simple change, we've achieved a 3,000x speedup!

With a single value computed in advance, the database can search its index.

Of course, sometimes the math can't be refactored quite so easily. But when writing queries with restrictions on indexed columns, avoid math on the indexed column and you can get some pretty massive speedups.

Approximations

For truly large data sets, **approximations** can be one of the most powerful performance tools you have. In certain contexts, an answer that is accurate to +/-5% is just as useful for making a decision—and multiple orders of magnitude faster to get.

However, this technique comes with a few caveats. The first is making sure the customers of your data understand the limitations. Error bars are a useful visualization and are well-understood by data consumers.

The second caveat is to understand the assumptions your technique makes about the distribution of your data; for example, if you data isn't normally distributed, sampling will produce incorrect results.

Let's dig into various techniques and their uses!

HyperLogLog

We'll optimize a very simple query, which calculates the daily distinct sessions for five million gameplays (~150,000/day):

```
select
  date(created_at)
  , count(distinct session_id)
from gameplays
```

The original query takes 162.2 seconds. The HyperLogLog version is 5.1x faster (31.5 seconds) with a 3.7% error, and uses a small fraction of the RAM.

Why HyperLogLog?

Databases often implement count(distinct) in two ways:

1. When there are few distinct elements, the database makes a hashset in RAM and then counts the keys.
2. When there are too many elements to fit in RAM, the database writes them to disk, sorts the file, and then counts the number of element groups.

The second case—writing the intermediate data to disk—is very slow. Probabilistic counters are designed to use as little RAM as possible, making them ideal for large data sets that would otherwise page to disk.

The HyperLogLog probabilistic counter is an algorithm for determining the approximate number of distinct elements in a set using minimal RAM. Selecting distincts from a set that has 10M unique, 100-character strings can take more than a gigabyte of RAM using a hash table—meanwhile, HyperLogLog uses less than a megabyte. Since the probabilistic counter can stay entirely in RAM during the process, it's much faster than any alternative that has to write to disk, and usually faster than alternatives using a lot more RAM.

Hashing

The core of the HyperLogLog algorithm relies on one simple property of uniform hashes: The probability of the position of the leftmost set bit in a random hash is 1/2n, where n is the position. We call the position of the leftmost set bit the **most significant bit**, or **MSB**.

Here are some hash patterns and the positions of their MSBs:

Hash	MSB Position	Hashes like this
1xxxxx	1	50%
01xxxx	2	25%
001xxx	3	12.5%
0001xx	4	6.25%

Hashtext is an undocumented hashing function in Postgres. It hashes strings to 32-bit numbers. We could use MD5 and convert it from a hex string to an integer, but this is faster.

We use ~(1<<31) to clear the leftmost bit of the hashed number. Postgres uses that bit to determine if the number is positive or negative, and we only want to deal with positive numbers when taking the logarithm.

The floor(log(2,...)) does the heavy lifting: The integer part of the base-2 logarithm tells us the position from the right of the MSB. Subtracting that from 31 gives us the position of the MSB from the left, starting at 1.

With that line, we've got our MSB per-hash of the session_id field!

Bucketing

The maximum MSB for our elements is capable of crudely estimating the number of distinct elements in the set. If the maximum MSB we've seen is 3, given the probabilities above we'd expect around 8 (i.e., 23) distinct elements in our set. Of course, this is a terrible estimate to make as there are many ways to skew the data.

The HyperLogLog algorithm divides the data into evenly sized buckets and takes the harmonic mean of the maximum MSBs of those buckets. The harmonic mean is better here since it discounts the outliers, reducing the bias in our count.

Using more buckets reduces the error in the distinct count calculation, at the expense of time and space. The function for determining the number of buckets needed given a desired error is:

$$2^{\left\lceil \log_2\left(\frac{1.04}{error\_rate}\right)^2 \right\rceil}$$

We'll aim for for a +/-5% error, so plugging in 0.05 for the error rate gives us 512 buckets. Here's the SQL for grouping MSBs by date and bucket:

```
select
  date(created_at) as created_date
  , hashtext(session_id) & (512 - 1) as bucket_num
  , 31 - floor(log(2, min(hashtext(session_id) & ~(1 <<
31))))
   as bucket_hash
from sessions
group by 1, 2
order by 1, 2
```

The hashtext(...) & (512 - 1) gives us the rightmost 9 bits, 511 in binary is 111111111), and we're using that for the bucket number.

The bucket_hash line uses a min inside the logarithm instead of something like this max(31 - floor(log(...))) so that we can compute the logarithm once—greatly speeding up the calculation.

Now we've got 512 rows for each date—one for each bucket—and the maximum MSB for the hashes that fell into that bucket. In future examples we'll call this select bucketed_data.

Counting

It's time to put together the buckets and the MSBs. The new variables are m (the number of buckets: 512, in our case) and M (the list of buckets indexed by j: the rows of SQL, in our case). The denominator of this equation is the harmonic mean mentioned earlier:
.

$$\frac{\left(\frac{0.7213}{1+\frac{1.079}{m}}\right) \times m^2}{\sum_{j=1}^{m} 2^{-M[j]}}$$

In SQL, it looks like this:

```
select
  created_date
  , ((pow(512, 2) * (0.7213 / (1 + 1.079 / 512))) /
    ((512 - count(1)) +
    sum(pow(2, -1 * bucket_hash))))::int
    as num_uniques
  , 512 - count(1) as num_zero_buckets
from bucketed_data
group by 1
order by 1
```

We add in (512 - count(1)) to account for missing rows. If no hashes fell into a bucket it won't be present in the SQL, but by adding 1 per missing row to the result of the sum we achieve the same effect.

The num_zero_buckets is pulled out for the next step where we account for sparse data.

We now have distinct counts that will be right most of the time—now we need to correct for the extremes. In future examples we'll call this select counted_data.

Correcting

The results above work great when most of the buckets have data. When a lot of the buckets are zeros (missing rows), the the counts get a heavy bias. To correct for that, we apply the formula below only when the estimate is likely biased.

$$\left(\frac{0.7213}{1+\frac{1.079}{m}}\right) \times m \times \log\left(\frac{m}{num\_zero\_buckets}\right)$$

The SQL for that looks like this:

```
select
  counted_data.created_date
  , case when num_uniques < 2.5 * 512
      and num_zero_buckets > 0
    then ((0.7213 / (1 + 1.079 / 512)) *
      (512 * log(2, (512::numeric) /
      num_zero_buckets)))::int
    else num_uniques end as approx_distinct_count
from counted_data
order by 1
```

Let's put it all together:

```
select
  counted_data.created_date,
  case
    when num_uniques < 2.5 * 512 and num_zero_buckets
> 0 then
      ((0.7213 / (1 + 1.079 / 512)) * (512 *
        log(2, (512::numeric) / num_zero_
buckets)))::int
  else num_uniques end as approx_distinct_count
from (
  select
    created_date
    , ((pow(512, 2) * (0.7213 / (1 + 1.079 / 512))) /
        ((512 - count(1)) +
        sum(pow(2, -1 * bucket_hash))))::int
      as num_uniques
    , 512 - count(1) as num_zero_buckets
  from (
    select
      date(created_at) as created_date
      , hashtext(session_id) & (512 - 1) as bucket_num
      , 31 - floor(log(2, min(hashtext(session_id) &
~(1 << 31))))
        as bucket_hash
    from gameplays
    group by 1, 2
  ) as bucketed_data
  group by 1
  order by 1
) as counted_data order by 1
```

And that's the HyperLogLog probabilistic counter in pure SQL!

Bonus: Parallelizing

The HyperLogLog algorithm really shines when you're in an environment where you can count (distinct) in parallel. The results of the bucketed_data step can be combined from multiple nodes into one superset of data, greatly reducing the cross-node overhead usually required when counting distinct elements across a cluster of nodes. You can also preserve the results of the bucketed_data step for later, making it nearly free to update the distinct count of a set on the fly!

Sampling

Sampling is an incredibly powerful tool to speed up analyses at scale. While it's not appropriate for all datasets or all analyses, when it works, it really works. At Periscope Data, we've realized several orders of magnitude in speedups on large datasets with judicious use of sampling.

However, when sampling from databases, it's easy to lose all your speedups by using inefficient methods to select the sample itself. Let's look at how to select random samples in fractions of a second.

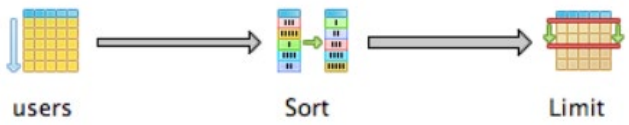
The Obvious, Correct—and Slow—Solution

Let's say we want to send a coupon to a random 100 users as an experiment. The naive approach sorts the entire table randomly and selects N results. It's slow, but it's simple and it works even when there are gaps in the primary keys.

Here's the query for selecting a random row:

```
select
  *
from
  users
order by
  random()
limit 1
```

On a Postgres database with 20M rows in the users table, this query takes 17.51 seconds! Let's see why:



The database is sorting the entire table before selecting our 100 rows! This is an O(n log n) operation, which can easily take minutes—or longer—on a 100M+ row table. Even on medium-sized tables, a full table sort is unacceptably slow in a production environment.

Query Faster by Sorting Only a Subset of the Table

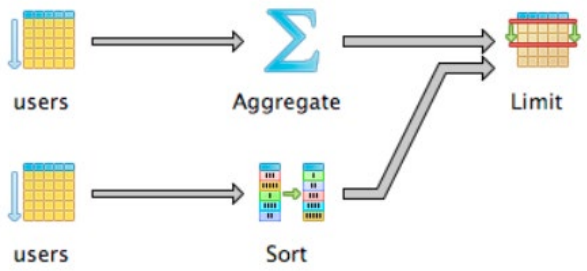
The most obvious way to speed this up is to filter down the dataset before doing the expensive sort.

We'll select a larger sample than we need and then limit it, because we might get randomly fewer than the expected numbers of rows in the table.

Here's our new query:

```
select * from users
where
  random() < 200 / (select count(1) from logs)::float
order by random()
limit 100
```

Now the query runs in 7.97 seconds—twice as fast!



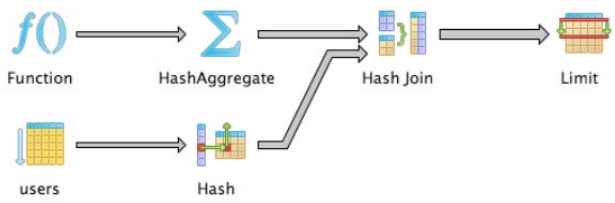
This is pretty good, but we can do better. You'll notice we're still scanning the table, albeit after the restriction. Our next step will be to avoid scans of any kind.

Generate Random Indices in the ID Range

Ideally we wouldn't use any scans at all, and rely entirely on index lookups. If we have an upper bound on table size, we can generate random numbers in the ID range and then look up the rows with those IDs.

```
select * from users
where id in (
  select round(random() * 21e6)::integer as id
  from generate_series(1, 110)
  group by id -- Discard duplicates
)
limit 100
```

This query runs in 0.064s, a 273x speedup over the naive query!



Counting the table itself takes almost 8 seconds, so we'll just pick a constant beyond the end of the ID range, sample a few extra numbers to be sure we don't lose any, and then select the 100 we actually want.

Bonus: Random Sampling With Replacement

Imagine you want to flip a coin 100 times. If you flip a heads, you need to be able to flip another heads. This is called **sampling with replacement**. All of our previous methods couldn't return a single row twice, but the last method was close. If we remove the inner group by id, then the selected IDs can be duplicated.

```
select * from users
where id in (
  select round(random() * 21e6)::integer as id
  from generate_series(1, 110) -- Preserve duplicates
)
limit 100
```

Sampling is an incredibly powerful tool for speeding up statistical analyses at scale, but only if the mechanism for getting the sample doesn't take too long. Next time you need to do it, generate random numbers first, then select those records.

Conclusion

A Complete Platform to Support Your Analytics Lifecycle.

We built our platform for the entire data team — supporting them from database connection to transformation to C-suite-ready visualization. Periscope Data begins where the data resides and makes it simple to bring all your data together in one place. Layer on Periscope Data's intelligent SQL-based analytics editor and data analysts can move faster than ever to spot insights and create shareable, interactive dashboards.

About Periscope Data

Periscope Data is the analytics system of record for professional data teams. The platform enables data leaders, analysts and engineers to unlock the transformative power of data using SQL and surface actionable business insights in seconds—not hours or days. Periscope Data gives them control over the full data analysis lifecycle, from data ingestion, storage and management through analysis, visualization and sharing. The company serves nearly 900 customers including Adobe, New Relic, EY, ZipRecruiter, Tinder and Flexport, with analysts spending the equivalent of two business days per week in the platform to support data-driven decision-making. Periscope Data is headquartered in San Francisco, CA. For more information, visit www.periscopedata.com.