A concise way to put it is that *standard sampling theory* (in particular, the **Central Limit Theorem**–based formula for estimating a proportion) tells us that once the population is large, the required sample size for a desired margin of error depends very little on the total population NNN. In other words, the **margin of error** in a proportion estimate is primarily a function of:

1. The **sample size** nnn,
2. The **confidence level** (via a z‐score), and
3. The underlying proportion ppp,

—and *not* directly on how large NNN is, so long as NNN is “large enough” to treat the sample as a small fraction of the population (where the finite population correction factor is negligible).

**Why Population Size Hardly Matters for Large NNN**

* **Central Limit Theorem (CLT)** for proportions:  
  When you take a simple random sample and compute the sample proportion p^\hat{p}p^​, its sampling distribution (for large nnn) is approximately normal with standard error p(1−p)n\sqrt{\frac{p(1-p)}{n}}np(1−p)​​. Notice NNN doesn’t appear in that basic form—hence the sample size needed for a given margin of error doesn’t blow up just because NNN is in the millions.
* **Finite Population Correction (FPC)**:  
  Formally, when the sample is more than a “small fraction” of the population, statisticians multiply by a correction factor N−nN−1\sqrt{\frac{N - n}{N - 1}}N−1N−n​​. But if nnn is in the hundreds or a few thousands and NNN is in the millions, this factor is extremely close to 1—so the effect of NNN practically vanishes.

**Bottom Line**

This result (that a few hundred samples can suffice for a 30-million-person population) follows directly from the **classical margin-of-error formula** for proportions, itself derived from the **Central Limit Theorem** and basic sampling theory. Once NNN is large, *the confidence interval width is controlled mainly by how large your sample nnn is, not how large NNN is.*

o1

**You said:**

how do we know the propotion ?

When planning a sample size for estimating a proportion (let’s call it ppp), we generally **don’t** know the exact value of ppp in advance, because that’s what we want to find out. Here are the usual approaches to deal with this:

**1. Use a Conservative “Worst-Case” Assumption (p=0.5p=0.5p=0.5)**

If you have **no idea** what ppp might be:

1. Set p=0.5p = 0.5p=0.5 in your margin‐of‐error formula.
2. This choice maximizes p(1−p)p(1-p)p(1−p), which is largest at p=0.5p=0.5p=0.5.
3. Hence, using p=0.5p=0.5p=0.5 ensures you pick a **large enough** sample.

For instance, if you want a 95% confidence interval with ±5% margin of error, the standard formula gives:

n=z2×p(1−p)E2=(1.96)2×0.5×0.5(0.05)2≈384.n = \frac{z^2 \times p(1-p)}{E^2} = \frac{(1.96)^2 \times 0.5 \times 0.5}{(0.05)^2} \approx 384.n=E2z2×p(1−p)​=(0.05)2(1.96)2×0.5×0.5​≈384.

By plugging in p=0.5p=0.5p=0.5, you’re making a **conservative** assumption. If the **actual** proportion is, say, 0.20 or 0.80, then your required sample would be smaller—but at least you won’t under-sample.

**2. Use a Prior or Pilot Estimate**

If you **do** have some information about ppp—for example, from historical data, a pilot study, or industry benchmarks—then:

1. Use that estimate of ppp in the formula: n=z2 p (1−p)E2.n = \frac{z^2 \,p\,(1-p)}{E^2}.n=E2z2p(1−p)​.
2. This usually yields a smaller required sample size if ppp is far from 0.5 (e.g., 0.10 or 0.90).

**Example**

If you have past data suggesting that 10% of customers “pass a rule” (p=0.1p=0.1p=0.1), and you want ±5% margin of error,

n=(1.96)2×0.1×0.9(0.05)2≈139.n = \frac{(1.96)^2 \times 0.1 \times 0.9}{(0.05)^2} \approx 139.n=(0.05)2(1.96)2×0.1×0.9​≈139.

This is quite a bit smaller than the 384 for the worst‐case p=0.5p=0.5p=0.5.

**3. Refine via Pilot Testing**

Sometimes you can:

1. **Conduct a small pilot** (e.g., draw 50–100 observations, measure how many pass).
2. Estimate p^\hat{p}p^​ from the pilot data.
3. Plug p^\hat{p}p^​ into the formula to get a more precise sample size for the main study.

This approach lets you avoid overshooting or undershooting if you have **no** prior knowledge.

**4. Use an Iterative or Sequential Approach**

If your resources or timelines allow it:

1. Sample a small batch and compute the proportion p^\hat{p}p^​.
2. Recalculate how many more samples you need for the desired margin of error.
3. Continue until you reach your target precision.

This is common in **adaptive sampling** or **sequential sampling** designs.

**Key Point**

To **plan** a sample size for estimating a proportion, you typically **don’t** know the true ppp in advance—**that’s the point of the study**. So you rely on:

* Worst-case guess p=0.5p=0.5p=0.5 if you truly have no idea, or
* A prior estimate or pilot data, if available.

Either way, the formula ensures you pick enough subjects to achieve your desired margin of error at the chosen confidence level.

o1

**You said:**

so 1 sample with 385 customers is enough ?

Reasoned for a few seconds

In **purely statistical terms**, if your goal is to estimate *one proportion* (e.g., “the percentage of 30 million customers who pass a certain rule”) **within ±5%** at **95% confidence**, then yes—**one simple random sample of about 385** respondents (assuming a near 100% response rate or evaluation rate) *is sufficient* in the **worst-case scenario** (p=0.5p = 0.5p=0.5).

However, there are practical and methodological considerations to keep in mind:

1. **Nonresponse / Response Rate**
   * If you are conducting a survey and worry that not all selected customers will respond, you might need to *invite more people* (e.g., 1,000 invites) to ensure you end up with ~385 *actual* completed responses.
2. **Subgroups / Segmentation**
   * If you plan to *break down* your estimate into subgroups (e.g., by region, customer type), each subgroup needs enough samples to get its own margin of error. This can multiply your required sample size.
3. **More Precision or Higher Confidence**
   * If you want *tighter* precision than ±5% (say ±3%) or a higher confidence level (say 99%), you’ll need a larger sample—easily 1,000+ for ±3%, and 1,600+ for ±2%, etc.
4. **Pilot or Prior Estimate**
   * If you know, for example, that the actual proportion is closer to 10% or 90%, then the 385 estimate can be reduced. Conversely, if you are *completely unsure*, using p=0.5p=0.5p=0.5 ensures you won’t undershoot.

**Short Answer**

* **Statistically**: Yes, **385 in one simple random sample** typically suffices for a ±5% margin of error at 95% confidence—even if the population is 30 million.
* **Practically**: Ensure you have a high response/evaluation rate, and consider whether you need more samples for subgroups or narrower margins of error.

o1

**You said:**

what is it required for 99% confi level

Reasoned for 5 seconds

If you want a **±5% margin of error** but at a **99% confidence level**, you need a bigger sample size than for 95%. Using the standard proportion sample-size formula and the worst-case assumption p=0.5p = 0.5p=0.5:

n  =  z2⋅p (1−p)E2n \;=\; \frac{z^2 \cdot p \,(1-p)}{E^2}n=E2z2⋅p(1−p)​

* **Confidence Level**: 99%
* **zzz‐score** for 99% confidence: approximately **2.58**
* **ppp** (worst‐case): **0.5**
* **Margin of error** (EEE): **0.05**

Plugging in:

n  =  (2.58)2×0.5×0.5(0.05)2  ≈  6.6564×0.250.0025  =  665.64n \;=\; \frac{(2.58)^2 \times 0.5 \times 0.5}{(0.05)^2} \;\approx\; \frac{6.6564 \times 0.25}{0.0025} \;=\; 665.64n=(0.05)2(2.58)2×0.5×0.5​≈0.00256.6564×0.25​=665.64

So you need **about 666 respondents** in one random sample (assuming a near‐100% response rate).

**Note on Finite Population Correction**

With a population of 30 million, the **finite population correction** factor would be negligible, so nnn remains right around **666**.

o1

**You said:**

can you give a write on this for a busienss presentation for discovering customer eligibility using sampling methods

Reasoned for 4 seconds

Below is a succinct, business-oriented explanation of how to use sampling methods to estimate the proportion of customers eligible for a given program or rule. It highlights why sample sizes remain relatively small—even with millions of customers—and how to apply the results.

**1. Why Sampling?**

When you have millions of customers, testing *all* of them for eligibility can be time-consuming and costly. **Statistical sampling** lets you check a *small, random subset*—yet still make accurate estimates about the broader population’s eligibility rate.

**2. Key Concept: Margin of Error and Confidence Level**

1. **Margin of Error (MoE)**: Defines how close your sample results will be to the *true* proportion of eligible customers (e.g., ±5%).
2. **Confidence Level**: The probability that your estimate will fall within the margin of error from the true value (e.g., 95% or 99%).

**Example**: “With 95% confidence, the eligibility rate is 40% ±5%.”  
This means there’s a 95% chance the true rate is between 35% and 45%.

**3. Determining Sample Size**

**Formula for a Proportion** (worst-case scenario p=0.5p=0.5p=0.5):

n  =  z2  p  (1−p)E2,n \;=\; \frac{z^2 \; p \;(1-p)}{E^2},n=E2z2p(1−p)​,

* p=0.5p = 0.5p=0.5 if you have *no prior estimate*.
* zzz is the z‐score (about 1.96 for 95% confidence, 2.58 for 99%).
* EEE is the desired margin of error (e.g., 0.05 for ±5%).

| **Confidence** | **Margin of Error** | **zzz-Score** | **Approx. Sample Size** |
| --- | --- | --- | --- |
| 95% | ±5% | 1.96 | ~385 |
| 99% | ±5% | 2.58 | ~666 |

Despite having a **30-million** customer base, **only a few hundred** randomly selected customers are needed to achieve ±5% precision in estimating eligibility.

**Why?**

* Once your population is sufficiently large, the total size doesn’t greatly affect the sample size. Statistical theory (Central Limit Theorem) shows the required sample mostly depends on *desired accuracy*, *confidence level*, and *the proportion itself*—not the total population.

**4. Practical Steps**

1. **Define Objectives**:
   * What margin of error (MoE) can your business tolerate? ±5% is common.
   * Do you want 95% or 99% confidence?
2. **Determine the Sample Size**
   * If you need ±5% at 95% confidence, plan for ~385 completed evaluations.
3. **Draw a Truly Random Sample**
   * Randomly pick customers across your entire population (e.g., using a random number generator or a robust sampling feature in your CRM).
   * Avoid biases (e.g., only sampling frequent users).
4. **Ensure Adequate Response/Evaluation Rate**
   * If you expect some customers not to respond or if you can’t successfully evaluate everyone, invite or test more than 385 so you net the required *final* count.
5. **Conduct the Evaluation**
   * Apply the eligibility test/rule to each sample member.
   * Record whether they pass.
6. **Estimate Overall Eligibility**
   * The proportion passing the rule in your sample approximates the proportion passing in the entire population, within ±MoE.

**5. Interpreting & Communicating Results**

* **Headline**: “We estimate X% of our 30M customers are eligible, with a ±5% margin of error at 95% confidence.”
* If you need to break results into subgroups (e.g., by region), each subgroup needs its own sufficient sample size to maintain the same precision.

**6. Example Business Scenario**

1. **Goal**: A bank wants to see how many existing customers qualify for a new loan product.
2. **Chosen Criteria**: ±5% margin of error, 95% confidence.
3. **Plan**: Randomly select 385 customers. Apply the standard eligibility check. Suppose 150 out of 385 are eligible → 39% sample pass rate.
4. **Conclusion**: About 39% of all 30M customers likely qualify, but the true value could be between 34% and 44% (±5%).
5. **Next Steps**:
   * Decide if that potential range is business-feasible.
   * If you want narrower precision (e.g., ±2–3%), increase the sample size accordingly.

**7. Takeaways for Management**

* **Cost & Efficiency**: You don’t need thousands or millions tested. A carefully chosen random sample of a few hundred can yield statistically reliable insights about an entire population.
* **Confidence in Action**: Once you have the sample’s pass rate (plus the margin of error), you can make data-driven decisions—like budgeting for the program, planning resources, or setting marketing strategies.

**Bottom Line**

**Sampling** is a powerful, cost-effective way to discover customer eligibility. Even with **30 million** customers, **one random sample of a few hundred** (well-chosen and well-executed) can provide a robust estimate—saving time, money, and administrative overhead while maintaining a statistically valid margin of error.

o1

**You said:**

how to take 600 customers from a postgres database table randomly for this sample

Reasoned for 12 seconds

Below are a few ways to randomly select 600 rows from a large PostgreSQL table. The simplest method is often to use ORDER BY RANDOM(), but it can be expensive for very large tables. I'll explain both the straightforward approach and alternatives that can help with performance or precision.

**1. Easiest Method (Might Be Slow for Very Large Tables)**

sql

CopyEdit

SELECT \*

FROM customers

ORDER BY RANDOM()

LIMIT 600;

* **Pros**: Very straightforward; yields a uniformly random sample of 600 rows.
* **Cons**: ORDER BY RANDOM() forces PostgreSQL to generate a random value for every row and then sort them. On a table with tens of millions of rows, this can be **quite slow**.

Despite the performance hit, many teams still use it for one‐off samples. If you just need to do it once (or infrequently) and you can tolerate the runtime, this is the simplest.

**2. Using TABLESAMPLE (PostgreSQL 9.5+)**

PostgreSQL supports the SQL standard TABLESAMPLE clause, which can quickly pull a **rough** random subset of rows. Then you can limit to 600. For example:

sql

CopyEdit

SELECT \*

FROM customers TABLESAMPLE SYSTEM (0.1) -- Try a small percentage

LIMIT 600;

* **SYSTEM** sampling reads a fraction of pages (blocks) at the disk level.
* The fraction (0.1 here means ~0.1% of the table pages) should be chosen experimentally so you can roughly gather more than 600 rows, then apply LIMIT 600.
* **Pros**: Much faster than ORDER BY RANDOM() for huge tables.
* **Cons**: The distribution may not be perfectly uniform at the row level, since SYSTEM sampling occurs at the page/block level. Often good enough for many practical purposes, but not a mathematically perfect uniform sample.

If you want a more row-level approach, you can try TABLESAMPLE BERNOULLI (0.1), but be aware it still must scan the table and test each row.

**3. Random ID Selection (If You Have a Sequential ID)**

If your customers table has a sequential or numeric primary key (e.g., integer IDs running from 1 to some maximum), you can:

1. **Find the min and max ID**:

sql

CopyEdit

SELECT MIN(id) AS min\_id, MAX(id) AS max\_id

FROM customers;

Suppose you get min\_id = 1 and max\_id = 30,000,000.

1. **Generate ~600 random IDs in that range**, ensuring uniqueness. One way is to do it outside the database in a script. Within Postgres, you could attempt something like:

sql

CopyEdit

WITH bounds AS (

SELECT MIN(id) AS min\_id, MAX(id) AS max\_id FROM customers

),

random\_ids AS (

-- Generate more than 600, then take DISTINCT

SELECT DISTINCT (floor(random()\*(bounds.max\_id - bounds.min\_id + 1)

+ bounds.min\_id)::bigint) AS id

FROM bounds

CROSS JOIN generate\_series(1, 2000) gs

-- Enough times to get 600 distinct IDs (some duplicates are possible)

)

SELECT c.\*

FROM random\_ids r

JOIN customers c USING (id)

LIMIT 600;

* + **Pros**: Very fast, because you’re only fetching 600 rows by primary key lookup.
  + **Cons**: Slight chance you won’t get all 600 if many duplicates appear. That’s why we generate more (e.g., 2,000) and then pick distinct. Also, if your ID column has big “gaps” (deleted IDs, etc.), some random IDs may not exist, leading to fewer matched rows—again, over-generating helps.

This approach is often used in **production** settings for large tables because it’s far more efficient than ORDER BY RANDOM().

**4. Summary: Choosing an Approach**

* **One‐off / Not performance-critical**:
  + ORDER BY RANDOM() LIMIT 600. Easiest to write, guaranteed uniform distribution, but can be slow on huge tables.
* **Faster rough sampling**:
  + TABLESAMPLE SYSTEM (X) ... LIMIT 600; picks randomly at the page level. For many analytics tasks, it’s “random enough.”
* **Truly uniform, high performance** (assuming a numeric primary key):
  + Randomly generate ~600 unique IDs in the valid range, then join back to the table.

**Practical Tips for a Business Setting**

1. **Check if you have time/budget** to let ORDER BY RANDOM() run on 30 million+ rows. For a single run with robust hardware, it might be acceptable.
2. **Test TABLESAMPLE SYSTEM(0.1)** (or some fraction) if speed is essential and you only need approximate uniformity. Adjust the fraction so you end up with at least 600 rows, then use LIMIT 600.
3. **If you have a numeric ID with minimal gaps**, the random ID method can be both *very fast* and *precise*.

In any case, once you have your **600 randomly selected** customers, you can proceed with your sampling analysis—checking eligibility, verifying results, or any other test you need for your business presentation.

o1

**You said:**

why should TABLESAMPLE BERNOULLI (0.1) take time and resources ?

TABLESAMPLE BERNOULLI (p) in PostgreSQL scans the entire table and applies a probabilistic test to **each row** to decide whether to include it. Specifically, for each row, PostgreSQL compares a random number to the specified sampling probability ppp, and if random()≤p\text{random}() \leq prandom()≤p, that row is included. Because this process must touch every row—even those ultimately not included—it can be time-consuming and I/O-intensive on large tables.

Below are the key points that explain why:

1. **Full Table Scan**
   * **Bernoulli sampling** is a *row-level* process. PostgreSQL must read every row on disk (or in memory) in order to determine whether it makes it into the sample.
2. **Random Check Per Row**
   * For each row, PostgreSQL generates a random value and tests it against ppp.
   * This is *more computationally expensive* than skipping entire data pages (blocks) at once.
3. **Volume of Data**
   * When your table has tens or hundreds of millions of rows, scanning every row can be expensive from both I/O (reading from disk) and CPU (generating random numbers) perspectives.

**Comparison with SYSTEM Sampling**

* TABLESAMPLE SYSTEM (p) skips entire disk pages, which can be faster because:
  + It only decides once per page whether that page’s rows are included, instead of generating a random check per row.
  + If a page is “selected,” all or most of its rows get sampled (depending on the storage engine’s behavior).
  + If a page is “skipped,” none of its rows are read at all—less I/O compared to reading every row.

**Bottom Line**

**Bernoulli** sampling is more uniform at the row level but can be costly because it examines *each* row. **SYSTEM** sampling works at the page level, skipping large chunks of data at once. The choice depends on your **speed vs. uniformity** needs:

* If you prioritize **perfectly uniform row sampling** and can handle a full table scan, Bernoulli sampling is fine.
* If performance and speed are more important, SYSTEM sampling is typically faster for large tables.

o1

**You said:**

can you give different ways to select data from table and its pros and cons

Reasoned for a few seconds

Below are several common ways to **select subsets of data** from a PostgreSQL table—especially for random sampling, but also for other practical scenarios. Each method has its pros and cons in terms of speed, randomness, and simplicity.

**1. ORDER BY RANDOM() LIMIT X**

sql

CopyEdit

SELECT \*

FROM mytable

ORDER BY RANDOM()

LIMIT 600;

* **How it works**: PostgreSQL assigns a random value to each row (random()), sorts all rows by that random value, then returns the first 600.
* **Pros**:
  + Very easy to write and understand.
  + Yields a truly uniform random sample of size X.
* **Cons**:
  + Potentially **slow** for large tables because it must generate a random number for *every* row and then sort all rows.
  + High memory and CPU usage on very large datasets.

**2. TABLESAMPLE SYSTEM (p)**

sql

CopyEdit

SELECT \*

FROM mytable TABLESAMPLE SYSTEM (0.2)

LIMIT 600;

* **How it works**: Reads only a fraction of pages (blocks) based on the sampling rate (0.2 = 20%). Any rows on the selected pages are returned, from which you can then LIMIT or filter further.
* **Pros**:
  + Faster than ORDER BY RANDOM() on large tables because it can skip many disk pages entirely.
  + Useful when approximate randomness is acceptable.
* **Cons**:
  + **Page-level** sampling: not perfectly uniform at the row level. Some pages may have more rows than others, leading to some bias.
  + If your data is clustered (e.g., by time), you might inadvertently oversample certain ranges of data.

**3. TABLESAMPLE BERNOULLI (p)**

sql

CopyEdit

SELECT \*

FROM mytable TABLESAMPLE BERNOULLI (0.2)

LIMIT 600;

* **How it works**: For each row, PostgreSQL compares a random number to 0.2; if it’s below that threshold, the row is included.
* **Pros**:
  + More uniform *row-level* sampling than SYSTEM because it decides per row rather than per block.
* **Cons**:
  + Still scans **all rows**, meaning it can be slow and I/O-heavy for big tables.
  + Not as fast as SYSTEM (p) because you can’t skip entire pages at once; each row needs a probabilistic check.

**4. Random ID Selection (If You Have Sequential or Numeric IDs)**

**a) Generating Random IDs Then Joining**

1. Find the min and max ID:

Below are several common ways to **select subsets of data** from a PostgreSQL table—especially for random sampling, but also for other practical scenarios. Each method has its pros and cons in terms of speed, randomness, and simplicity.

**1. ORDER BY RANDOM() LIMIT X**

sql

CopyEdit

SELECT \*

FROM mytable

ORDER BY RANDOM()

LIMIT 600;

* **How it works**: PostgreSQL assigns a random value to each row (random()), sorts all rows by that random value, then returns the first 600.
* **Pros**:
  + Very easy to write and understand.
  + Yields a truly uniform random sample of size X.
* **Cons**:
  + Potentially **slow** for large tables because it must generate a random number for *every* row and then sort all rows.
  + High memory and CPU usage on very large datasets.

**2. TABLESAMPLE SYSTEM (p)**

sql

CopyEdit

SELECT \*

FROM mytable TABLESAMPLE SYSTEM (0.2)

LIMIT 600;

* **How it works**: Reads only a fraction of pages (blocks) based on the sampling rate (0.2 = 20%). Any rows on the selected pages are returned, from which you can then LIMIT or filter further.
* **Pros**:
  + Faster than ORDER BY RANDOM() on large tables because it can skip many disk pages entirely.
  + Useful when approximate randomness is acceptable.
* **Cons**:
  + **Page-level** sampling: not perfectly uniform at the row level. Some pages may have more rows than others, leading to some bias.
  + If your data is clustered (e.g., by time), you might inadvertently oversample certain ranges of data.

**3. TABLESAMPLE BERNOULLI (p)**

sql

CopyEdit

SELECT \*

FROM mytable TABLESAMPLE BERNOULLI (0.2)

LIMIT 600;

* **How it works**: For each row, PostgreSQL compares a random number to 0.2; if it’s below that threshold, the row is included.
* **Pros**:
  + More uniform *row-level* sampling than SYSTEM because it decides per row rather than per block.
* **Cons**:
  + Still scans **all rows**, meaning it can be slow and I/O-heavy for big tables.
  + Not as fast as SYSTEM (p) because you can’t skip entire pages at once; each row needs a probabilistic check.

**4. Random ID Selection (If You Have Sequential or Numeric IDs)**

**a) Generating Random IDs Then Joining**

1. Find the min and max ID:

sql

CopyEdit

SELECT MIN(id) AS min\_id, MAX(id) AS max\_id

FROM mytable;

Suppose min\_id = 1 and max\_id = 30,000,000.

1. Generate random IDs within that range (e.g., in a CTE or a separate script):

sql

CopyEdit

WITH bounds AS (

SELECT MIN(id) AS min\_id, MAX(id) AS max\_id

FROM mytable

),

random\_ids AS (

-- Generate more random IDs than needed to account for duplicates or gaps

SELECT DISTINCT (floor(random()\*(bounds.max\_id - bounds.min\_id + 1)

+ bounds.min\_id)::bigint) AS id

FROM bounds

CROSS JOIN generate\_series(1, 2000) -- generate some large number

)

SELECT t.\*

FROM mytable t

JOIN random\_ids r USING (id)

LIMIT 600;

* **Pros**:
  + Fast lookups if you have an index on id.
  + Avoids scanning the entire table or sorting all rows.
  + True random selection of IDs (assuming minimal ID “gaps”).
* **Cons**:
  + Requires a numeric or sequential-like ID covering the full range (few or no large gaps).
  + Must over-generate random IDs to ensure you get 600 valid, existing IDs.
  + If your ID space is sparse or has many gaps, it’s trickier to get the desired sample size.

**b) Using an External Script**

* You can also generate random IDs in a programming language (e.g., Python) and query them in batches:

python

CopyEdit

import random

min\_id, max\_id = ... # from a query

random\_ids = set()

while len(random\_ids) < 600:

random\_ids.add(random.randint(min\_id, max\_id))

# Then pass random\_ids into a query with an IN clause (possibly in chunks).

* **Pros**:
  + Similar benefits—fewer duplicates with good logic.
  + Control the generation process externally.
* **Cons**:
  + Have to handle the ID sets, chunk them if needed (WHERE id IN (...)) so as not to exceed parameter limits.
  + Still must handle the possibility that some IDs might not exist if there are gaps.

**5. Hashing or Adding a “Random” Column**

Another trick is to **add a column** with a random hash or random value (once), then sample based on that column:

1. **Add a random value** (one-time assignment) to each row:

sql

CopyEdit

ALTER TABLE mytable ADD COLUMN rand\_val float;

UPDATE mytable

SET rand\_val = random();

1. Then you can:

sql

CopyEdit

SELECT \*

FROM mytable

WHERE rand\_val < 0.001 -- ~0.1% of rows

LIMIT 600;

* **Pros**:
  + Once you’ve stored a random value, sampling again is *fast*—no need to recalculate each time.
  + You can index rand\_val if needed, though typically you just do a sequential scan if the table is large.
* **Cons**:
  + Requires an extra column.
  + Not truly dynamic if you keep the same random values (the sample is the same every time unless you update them again).
  + The initial assignment is effectively scanning the entire table once.

**6. LIMIT with OFFSET + Random Offsets**

Some might consider:

sql

CopyEdit

SELECT \*

FROM mytable

OFFSET <some random number>

LIMIT 600;

* **This is *not* a uniform random sample** of the *entire table*; it just starts at a random row. If the table is not fully shuffled, it’s biased.
* Could be useful for quick demos, but for serious random sampling, it’s typically avoided.

**Summary Table**

| **Method** | **Query Example** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **ORDER BY RANDOM()** | SELECT ... ORDER BY RANDOM() LIMIT 600; | - Very easy  - Truly uniform at row level | - Slow for large tables  - High CPU and memory |
| **TABLESAMPLE SYSTEM (p)** | SELECT ... FROM tbl TABLESAMPLE SYSTEM(p) | - Fast for big tables (page-level skip)  - Good for “approx” random sample at scale | - Not perfectly uniform at row level  - Some pages might have more or fewer rows |
| **TABLESAMPLE BERNOULLI (p)** | SELECT ... FROM tbl TABLESAMPLE BERNOULLI(p) | - Uniform row-level probability  - Standard SQL | - Must read/test each row (slow for huge tables) |
| **Random ID Approach** | Generate random IDs, then join or WHERE id IN (...) | - Very fast if id is indexed  - Minimal overhead if numeric IDs are mostly consecutive | - Requires numeric or sequential IDs with few large gaps  - Need to handle duplicates or missing IDs (over-generate) |
| **Pre-stored Random Column** | ALTER TABLE ... ADD COLUMN rand\_val then SELECT ... WHERE rand\_val < x | - Once stored, sampling is quick  - Doesn’t re-scan entire table for each sample | - Requires extra column  - Need to reassign if you want a *different* sample later |
| **OFFSET + LIMIT** | SELECT ... OFFSET random LIMIT 600; | - Very simple to code for small uses | - Not truly random if table is unsorted  - Biased sampling unless data is randomly ordered |

**Final Thoughts**

* **Performance** vs. **Uniform Randomness** is a major trade-off.
* For **one-off** usage on moderate data, ORDER BY RANDOM() is the simplest.
* For **huge** datasets, consider TABLESAMPLE SYSTEM or the **Random ID** approach.
* If you need repeated random draws, adding a **persistent random column** might save re-sorting overhead later on.

By choosing the method that balances your **speed**, **uniformity**, and **operational needs**, you can get the sample you want with minimal overhead.

o1

**You said:**

can you put this information a table format

Below is a **Markdown table** summarizing different ways to select (especially random) subsets of data from a PostgreSQL table. Each row outlines **what the method is**, a **sample query**, **advantages**, and **disadvantages**.

| **Method** | **Query Example** | **Pros** | **Cons** |
| --- | --- | --- | --- |
| **1. ORDER BY RANDOM()** | sql<br>SELECT \*<br>FROM mytable<br>ORDER BY RANDOM()<br>LIMIT 600;<br> | - Very easy to write and understand - Yields uniform random sample of size X | - Potentially slow for large tables (must generate and sort random values for every row) - High CPU and memory usage |
| **2. TABLESAMPLE SYSTEM (p)** | sql<br>SELECT \*<br>FROM mytable TABLESAMPLE SYSTEM(0.2)<br>LIMIT 600;<br> | - Fast for big tables (skips entire pages) - Good for approximate random sampling at scale | - Page-level sampling: not perfectly uniform at the row level - Rows in selected pages might not be representative if data is clustered |
| **3. TABLESAMPLE BERNOULLI (p)** | sql<br>SELECT \*<br>FROM mytable TABLESAMPLE BERNOULLI(0.2)<br>LIMIT 600;<br> | - More uniform at the row level than SYSTEM - Standard SQL approach | - Must read/test each row (scans entire table) - Can be slow for very large datasets |
| **4. Random ID Selection** | 1. Get min & max ID 2. Generate random IDs in that range 3. Query by those IDs | - Very fast if you have an indexed, mostly sequential ID - Avoids sorting or scanning the entire table | - Requires a numeric or sequential ID with few gaps - Might need to generate more IDs than needed to avoid duplicates or gaps |
| **5. Pre-stored Random Column** | 1. Add rand\_val column with UPDATE 2. Query using WHERE rand\_val < x or ORDER BY rand\_val LIMIT ... | - Once stored, repeated random sampling is quick (no need to re-calculate each time) - Can index if needed | - Requires adding and populating an extra column - You get the same sample each time unless you periodically reassign rand\_val |
| **6. OFFSET + LIMIT** | sql<br>SELECT \*<br>FROM mytable<br>OFFSET <random offset><br>LIMIT 600;<br> | - Simple to code, might be okay for small/quick demos | - Not truly random unless data is already in random order - Can introduce bias if the table is sorted or if offsets skip large chunks of the table |

**Notes**

1. **Which approach to use?**
   * For **small‐to‐medium** tables (and one‐off needs): ORDER BY RANDOM() is simplest.
   * For **very large** tables:
     + **TABLESAMPLE SYSTEM(p)** is faster (page-level sampling).
     + **Random ID method** is good if you have a numeric, relatively gap-free key.
   * **Repeated Sampling**: Consider **pre-stored random columns** to avoid recalculating each time.
2. **Sampling Percentage vs. Exact Count**
   * TABLESAMPLE methods select a *fraction* of rows, so you often need to guess a fraction that yields enough rows, then do LIMIT.
   * With ORDER BY RANDOM(), LIMIT X directly gives you the desired sample size (though it sorts the entire table).
3. **Uniform vs. Approximate Randomness**
   * ORDER BY RANDOM() and TABLESAMPLE BERNOULLI(p) are closer to true row-level uniform.
   * TABLESAMPLE SYSTEM(p) is approximate at page level. Often “random enough” for analytics but not purely uniform.