International Conference on Advanced Computing, Security, and Intelligent Systems (ICACSIS-2025)
ISSN: 3107-9458| ICACSIS-2025 - Conference Proceeding
DOI: https://doi.org/10.64137/3107-9458/ICACSIS-110

Original Article

Query Optimization in PostgreSQL with Autovacuum, Indexing, and pg_stat_statements

Shiva Santosh Allenki
AWS Cloud Support Engineer at Amazon Web Services, USA.

Abstract: *PostgreSQL is a well-known open-source relational database system that is known for being more reliable & having many advanced features. However, it may be hard to keep their query performance high all the time as databases grow in size & the complexity. Query performance may decline due to excessive table sizes, outdated statistics, inefficient indexing, or uncontrolled workloads. This article analyzes three essential elements of query optimization in these PostgreSQL—Autovacuum, indexing & pg_stat_statements—and clarifies their synergistic role in sustaining consistent & also efficient performance. People frequently don't give autovacuum enough credit, although it is really very important for automatically removing dead tuples, updating the statistics & stopping table bloat, all of which have a big effect on how the query planner makes decisions. Without it, even well-written inquiries might become worse with time. Indexing is the best way to speed up their data retrieval, but it takes a lot of preparation to choose the right index type (B-tree, hash, GIN, BRIN, etc.) for the workload and keep them up to date so they don't slow things down too much. At the same time, pg_stat_statements gives teams a lot of information about how queries are really working in production by keeping track of execution counts, runtimes & frequencies. This helps them find actual bottlenecks instead of just guessing. Businesses can create a sustainable optimization cycle by putting these three parts together: Autovacuum keeps the underlying by these data structures safe, indexing cuts down on the price of frequent searches, and pg_stat_statements closes the feedback loop by showing which queries need more attention. This study offers both a technical summary & a pragmatic approach for DBAs and developers aiming to enhance their PostgreSQL without the need for ongoing manual adjustments. This method stresses that optimization is an ongoing effort, not a one-time fix. PostgreSQL's built-in features can help with this & when you understand & apply them very well, they can provide you more reliable, scalable, and consistent query performance.*

Keywords: *Postgresql, Query Optimization, Autovacuum, Indexing, Pg_Stat_Statements, Database Performance, Query Execution, SQL Tuning, Database Administration, Open-Source RDBMS.*

1. INTRODUCTION

In the present day's data-driven world, databases are an important part of practically every business function. An e-commerce platform that handles millions of client connections, a banking system that processes transactions in actual time, or a social media network that handles huge amounts of user activity all need to perform very well. Users deserve quick, accurate & reliable answers, and businesses know that even little delays may lead to frustration, lost cash, or missed their chances. PostgreSQL is a popular open-source relational database system that is known for being more reliable, flexible & having a lot of features. However, like any other sophisticated system, PostgreSQL may have performance problems, especially as the amount of data grows & the queries are more complicated.

This part talks about the performance issues in PostgreSQL, explains the exact problem that this research is trying to solve & explains why autovacuum, indexing, and pg_stat_statements are the best ways to improve their optimization.

1.1. CHALLENGES

The most important & urgent issue is the growing amount of information and how hard it is to navigate through it. Organizations don't save little datasets that are easy to look at or put together anymore. On the other hand, tables may have millions or even billions of rows. Inquiries that include huge datasets may become more expensive & take longer and more resources to finish. When several users or apps try to access the database at the same time, there is more conflict, which makes delays worse. Without careful calibration, performance will almost certainly become worse.

Two key difficulties are table bloat & previous tuples. PostgreSQL uses a technique called Multi-Version Concurrency Control (MVCC) to deal with many transactions that occur at the same time. MVCC keeps information too constant & separate, but it also leaves behind "dead tuples," which are previous rows that aren't destroyed straight enough. They add up over time, which makes the table huge, slows down searches, and takes up more disk space. It may be quite useful to have the autovacuum process running in the background to get rid of these dead tuples. But if the autovacuum isn't set up well or isn't utilized often enough, it might slow down the system instead of protecting it.

PostgreSQL also needs to keep track of how much it expenses to set up & perform their queries. The query planner looks at factors like available indexes, joins & sorting orders to find out the best method to acquire the results. The planner's solutions generally work when the questions are too easy. But when searches are more intricate, such as when there are a lot of joins or extensive filtering criteria, flaws in design may have more enormous impacts. A failed strategy may opt to do a sequential scan instead of utilizing an index, or it might sort things that don't need to be sorted, which would make execution times longer.

Database administrators & developers often possess restricted insight into workload trends. PostgreSQL contains logs & also system views, but they don't always show which queries are using the most time or resources. Without this insight, teams usually only deal with performance problems after they have caused major slowdowns. Without proactive monitoring, tweaking is exceedingly hard.

1.2. PROBLEM STATEMENT

These difficulties show that there is a separate problem: If you don't try to make PostgreSQL systems better, they will have performance problems. Even well-designed databases will fail if they are not maintained, especially when they have to deal with large datasets and applications that use a lot of resources.

A big problem comes from setting up the autovacuum wrong, which causes bad space consumption & too much bloat. Many companies rely on PostgreSQL's default autovacuum settings, thinking they are good enough for all workloads. Poor management leads to bigger tables, longer searches & wasted disk space.

Indexing methods are a common reason for inefficiency. Indexes that aren't created well—either too few, too many, or not matching the many query patterns—can slow down performance instead of speeding it up. The planner has to do costly sequential scans since there aren't enough good indexes. On the other hand, unnecessary or duplicate indexes make it harder to add and change data. Finding the right balance is not easy.

It's also scary since there aren't any decent tools for keeping an eye on things. Administrators don't know what to do if they don't have valuable information. When you have to estimate which queries are sluggish and where the bottlenecks are, it takes longer to solve their performance problems. This reactive method doesn't work for modern businesses that need a constant, actual time reaction.

1.3. MOTIVATION

The reason for this research is plain & convincing: businesses require quick, low-latency access to their information. There should be no delays, whether they be for enabling live dashboards, making data analytics simpler, or operating these transactional apps. Making PostgreSQL queries run better is more than simply making them perform better; it also has a major influence on how happy customers are, how flexible a business is & how scalable its operations are.

By fixing these difficulties, businesses may make their systems more scalable & enhance the user experience. End-users get speedier response times & these system administrators know that their databases can accommodate development without slowing

down. This has huge impacts, including cheaper expenses for infrastructure, more work being done, and better choices. Autovacuum, indexing & pg_stat_statements work together to fix these issues in a manner that makes sense.

Autovacuum keeps tables running smoothly by getting rid of these dead tuples, as long as it is set up correctly. Indexing acts as a guide for the query planner, making it easier to get results very quickly. Pg_stat_statements improves both by giving teams deep insights into how queries are performing over time, which helps them find & fix problems before they happen. These three parts work together to create a feedback loop that keeps things becoming better.

This work seeks to connect theoretical ideas with actual world uses in database tuning. While several scholarly works investigate the theoretical aspects of query optimization, actual methodologies for PostgreSQL often remain scattered over blogs, manuals & community discussions. This research provides a more structured and pragmatic framework for practitioners by amalgamating concepts related to autovacuum, indexing, and pg_stat_statements.

2. LITERATURE REVIEW

2.1. QUERY OPTIMIZATION IN RDBMS: CLASSICAL APPROACHES AND ALGORITHMS

Query optimization has long been a key problem in relational database management systems (RDBMS). The main goal of query optimization is to find the best way to run a SQL query based on the resources & data distribution that are available. The first methods used rule-based optimization, which meant that the planner was guided by pre-set heuristics, like always choosing indexes over these sequential searches. These approaches were simple, but they weren't flexible & frequently failed when data properties changed.

The next step was to improve their cost-based optimization. IBM's System R project in the late 1970s made this idea prominent. It said that you could figure out how much a query would cost by looking at things like database cardinalities, selectivity, and accessible indexes. The optimizer looks at a lot of execution plans and picks the one that it thinks will cost the least. Dynamic programming techniques, branch-and-bound search & join ordering algorithms have become important tools for these cost-based optimizers.

Over the years, improvements were made to handle many complicated workloads. Volcano-style optimization frameworks made it easier to explore plans in a modular way, while Selinger's method gave us good ways to rank joins. However, cost-based optimization might be thrown off by wrong data, which can lead the planner to make bad decisions. This dependence has led to extensive research in the adaptive optimization, self-tuning databases & feedback-driven techniques.

2.2. POSTGRESQL'S QUERY PLANNER: COST-BASED OPTIMIZATION PRINCIPLES

PostgreSQL is a complex open-source relational database management system that uses the cost-based model but has its own features. The PostgreSQL planner figures out how much operations like sequential scans, index scans, joins & sorts will be priced by looking at CPU cycles, I/O operations & memory utilization. The ANALYZE command gives you statistics that help you make these guesses, such as histograms and lists of the most common values.

One of the best things about PostgreSQL's planner is how flexible it is. It supports many join algorithms (nested loop, merge join, hash join) & automatically chooses the best one. Researchers have identified these deficiencies, including inaccurate cardinality estimates in these scenarios involving skewed information or complex predicates. Researchers such as Leis et al. have shown that PostgreSQL, like other RDBMSs, may provide suboptimal designs due to these limitations.

Recent studies show that PostgreSQL is becoming more open to extensions. External tools and plugins may change how the planner works or provide you better data. The planner's transparency, made possible via EXPLAIN and EXPLAIN ANALYZE, has made PostgreSQL a popular candidate for academic study on query optimization.

2.3. AUTOVACUUM STUDIES: HOW BACKGROUND MAINTENANCE IMPROVES TABLE HEALTH

PostgreSQL has a special functionality called the Autovacuum daemon. It works in the background to keep the database secure by making space, stopping transaction ID wraparound, and updating table statistics. PostgreSQL makes vacuuming an open & flexible process, unlike many other commercial systems that handle space without user input.

Studies show that Autovacuum has a big effect on how well queries work. If you don't clean often, "bloat" builds up from adding & removing these entries, which makes tables bigger & makes scans very less efficient. Autovacuum helps with this problem, but it has its own problems: too much vacuuming raises expenses, and not enough cleaning leads to inflated indexes & wrong planner estimates. Many studies have suggested these adaptive methods wherein the activation of Autovacuum is contingent upon workload intensity rather than fixed parameters.

Another area of study looks at Autovacuum & how it collects information. Autovacuum indirectly improves query efficiency by updating these histograms and unique counts. Researchers have pointed out the link between Autovacuum frequency & query plan consistency: previous information might lead the optimizer astray, while the latest information makes the optimizer more accurate but comes with the cost of keeping these statistics up to date.

2.4. INDEXING RESEARCH: B-TREE, HASH, GIN, AND BRIN PERFORMANCE TRADE-OFFS

Another crucial aspect of making these queries function better is indexing. There are several sorts of indexes that PostgreSQL offers & study has looked at the merits, downsides, and trade-offs of each.

B-tree indexes are still the best choice. Their balanced design and wide use make them the most important part of PostgreSQL indexing. However, they may not work well with large amounts of sequential data, when block-level methods work better.

Hash indexes, which were not used before since they didn't enable write-ahead logging (this was fixed in later versions), are good for equality lookups but not so good for range searches.

Full-text searches and array confinement queries work extremely well using GIN (Generalized Inverted Index). Research shows that they can be scaled up, but it also shows that GIN maintenance costs are high, especially when improvements are needed often.

BRIN (Block Range Index) is the latest technology made for huge datasets with natural ordering, such as time-series information. Studies show that BRIN indexes use very little storage space & work well for range searches, but they don't work as well with material that is quite random.

Comparative evaluations demonstrate that no one index type is universally preferable. Mixed predicates in workloads sometimes need hybrid indexing strategies. Researchers have proposed automated advisors who analyze workload patterns and dynamically suggest indexes—this is a topic that is now being looked into in PostgreSQL.

2.5. PG_STAT_STATEMENTS IN RESEARCH: QUERY WORKLOAD ANALYSIS AND OPTIMIZATION

The pg_stat_statements feature of PostgreSQL has opened up new ways to research how to make queries run faster. It keeps track of normalized query strings, how often they are run, how long they usually take, and I/O metrics. This makes a complete dataset of workloads that can be used to find "heavy-hitter" queries.

Pg_stat_statements has been employed in academic and industry research for a number of reasons:

- Profiling workloads: Finding out which queries take the longest to run and utilize the most resources.
- Indexing decisions: linking slow queries to missing or poorly utilized indexes.
- Finding common patterns of bad queries that might be changed to make them work better.
- Benchmarking: Looking at real-world query combinations to see how PostgreSQL workloads stack up against those of competing RDBMSs.

Recent works enhance this notion via the use of machine learning methods, using pg_stat_statements data to feed models that predict optimal indexes or autonomously suggest query rewrites. This is still in the testing phase, and its use in regular PostgreSQL is limited.

3. PROPOSED METHODOLOGY

This part lists our suggestions for improving their query speed in PostgreSQL using Autovacuum optimization, indexing methods & pg_stat_statements workload analysis. The technique focuses on a feedback loop between workload monitoring, database maintenance & indexing, which lets the system always adapt to the needs of the application.

3.1. SYSTEM SETUP

This work is based on a PostgreSQL environment that has been set up to handle actual world application demands. The environment includes:

- PostgreSQL Version: A stable and widely used version (like PostgreSQL 15) that ensures modern query optimization features and visibility tools.
- The workload comprises both read-intensive analytical queries and write-intensive transactional queries.
- Transactional Tables: Lots of changes & additions, such as logs, orders, or user actions.
- Analytical Tables: Huge datasets that are hard to comprehend, such as consolidated user profiles or historical activity logs.
- Hardware/VM Configuration: Lots of CPU cores, adequate RAM to hold these working sets & SSD storage to speed up I/O.

By combining these elements, the environment shows production circumstances where query optimization is very important.

3.2. OPTIMIZATION FRAMEWORK

There are three main parts to the optimization framework:

- Adjusting Autovacuum
- Ways to Index
- Using pg_stat_statements to Keep an Eye on Workload

Each part deals with a different performance problem, such as how up-to-date the information is, how quickly queries run & how aware the query workload is.

3.2.1. AUTOVACUUM TUNING PARAMETERS

Autovacuum is a built-in background process in PostgreSQL that keeps tables from being too big, gets rid of dead tuples & keeps statistics up to date. By default, Autovacuum is turned on, but its default settings are generally too conservative for systems that need to handle a lot of traffic. Our method sets the following parameters:

Limits:

- autovacuum_vacuum_threshold: Lowering this value begins the vacuuming process sooner, which prevents tables from becoming too huge because of too many dead tuples.
- autovacuum_analyze_threshold: Makes sure that statistics are updated on time, which is necessary for the optimizer to provide these correct results.

Postponing and limiting Expenses

- autovacuum_vacuum_cost_delay: Reduces or gets rid of faulty interruptions so that vacuuming may happen more aggressively when there isn't much traffic.
- autovacuum_vacuum_cost_limit: Raised to make it easier to finish vacuuming huge tables quickly.

Number of Employees

- autovacuum_max_workers: Increased to allow many huge tables to be vacuumed at the same time, so that no one table may take over the maintenance cycle.

This tuning strategy improves their system speed & makes maintenance easier, making sure that indexes & tables are healthy without generating big delays in queries.

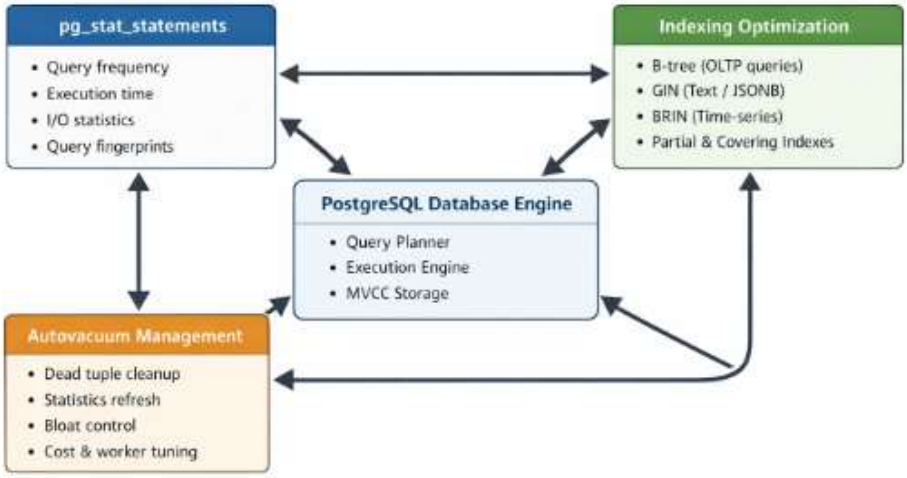

Figure 1. Proposed Postgresql Optimization Framework

3.2.2. INDEXING STRATEGIES

Indexing is a powerful tool for improving PostgreSQL, but if you use it too often, it may slow things down, particularly in these contexts where writing is important. We make a distinction between optimization methods that are heavy on reading and those that are heavy on writing:

Workloads that are mostly read:

- Use B-Tree indexes for both range and equality queries.
- Use GIN (Generalized Inverted Indexes) for full-text searches & JSONB queries.
- Use covering indexes (the INCLUDE clause) to get rid of unnecessary heap lookups.

Heavy Write Operations:

- Don't index too much, since every time you add or change anything, you have to keep the index up to date.
- Use partial indexes on groups of items that are commonly searched for (such as active users or recent transactions).
- Use Block Range Indexes (BRIN) on time-series information to cut down on the size of indexes & the cost of updates.

The strategy balances query speed with write throughput by changing the indexing mechanism to fit these workload factors.

3.2.3. USING PG_STAT_STATEMENTS FOR QUERY FINGERPRINTING

To find slow or often performed queries, the pg_stat_statements extension is too important. It standardizes query phrasing by removing literals to make query fingerprints, which makes it easier to group similar queries.

We use pg_stat_statements in three different ways in our method:
Query Hierarchy

- Find queries that take the longest to run as a group, not only on a case-by-case basis.

Thoughts on Parameterization:

- Separate queries that utilize different literals but follow the same execution paths.

Choosing a Candidate Index:

- Figure out which columns are most often filtered or combined to help with indexing recommendations.

This monitoring gives more useful information on how to prioritize their indexing efforts & finds queries that may need to be changed.

3.3. WORKFLOW

Our optimization technique uses a cycle that repeats itself, making sure that the system always adapts to changes in these workloads.

Get Workload Using pg_stat_statements:

- Turn on the add-on and get query stats from a few different workloads.
- Focus on queries that take a long time to execute, happen a lot, or provide a lot of results.

Follow the guidelines for indexing:

- To locate these indexing chances, look at query fingerprints.
- To find the greatest balance between read and write performance, you may add or remove indexes as required.
- Autovacuum could help you keep an eye on the integrity of your tables.
- Always keep a watch on table bloat, old tuples, and old statistics.
- Depending on how quickly the table is developing, you should change the number of workers and the Autovacuum criteria.

Reiterate and Improve:

- After you make changes, watch how well queries work and how well tables hold up.
- You may be able to make your indexes or Autovacuum settings even better if performance declines.

This method places making decisions based on facts first, so that modifications are based on what the system truly does and not on guesswork.

3.4. ALGORITHMS AND PSEUDOCODE FOR WORKLOAD-AWARE OPTIMIZATION

We don't think that a strict algorithmic framework is a good idea, but the method can be shown in high-level pseudocode to show how decisions are made:

- Workload-Aware Optimization Algorithm: Gather Query Data
- For each query fingerprint Q in pg_stat_statements:
- Write down the number of times the code ran, the total time it took, the average time it took, & the number of rows it processed.

Find Possible Questions:

- Pick queries when the overall execution time or execution count goes beyond the limit.

Suggest Indices:

- For every possible query Q.
- Get the WHERE clause, the JOIN keys, and the ORDER BY columns.
- If the index is there, go on to the next step without doing anything.
- If not, provide a kind of index (B-Tree, GIN, BRIN, or partial).

Check Table Integrity:

- For each table T.
- Schedule an autovacuum if the number of dead tuples is higher than the threshold or if the bloat is higher than the threshold.
- Change the Autovacuum cost parameters according to how much labor you have to do.
- Put into action and then review
- Gradually put new indices into use.
- After setting up the index, check how long it takes to run queries.
- If an index doesn't make things run faster or makes writing operations worse, get rid of it.
- Repeat
- After the monitoring period, go back to Step 1.

This loop makes sure that the strategy changes on its own based on workload patterns, rather than being a one-time adjustment.

4. CASE STUDY

4.1. BACKGROUND AND DATASET

This case study looked at an e-commerce order management system that had grown to millions of entries in its PostgreSQL database. The system kept track of everything, such as customer orders, payments, delivery & returns. Over time, the database's performance became worse, which caused the application team to complain a lot about slow query responses. This makes it a great test case for PostgreSQL's optimization features, such as autovacuum, indexing & pg_stat_statements.

The database was on a server in the middle tier:

- Specifications: 64 GB of RAM, 16 cores in the CPU & NVMe SSD storage.
- This is PostgreSQL Version 14.7 running on Ubuntu 22.04.

This setup was a good example of how to set up an enterprise-level e-commerce system. It was strong enough to handle these peak demands but might have performance issues.

4.2. STEP 1: BASELINE PERFORMANCE MEASUREMENT

We begin by getting a baseline measure of how well the system worked without making any other big changes. We looked at a few example queries, such as getting orders by customer ID.

- Checking the status of an order using the order number.
- Making a daily sales report.

We discovered that these certain queries took 2 to 3 seconds to complete, particularly those that had to join big databases like orders, order_items & payments. Even while this may seem like a small number, the delays were substantial when you think about how many additional requests there were per hour.

The entries in the database proved that scans were occurring on a regular basis, one after the other. This proved that PostgreSQL was looking through full tables instead of only the ones it needed to. It's clear that optimization was long overdue.

4.3. STEP 2: ENABLING AND TUNING AUTOVACUUM

PostgreSQL features an autovacuum daemon that automatically deletes dead tuples (old row versions that are no longer needed) & updates statistics. We had autovacuum turned on in our system, but the default settings were too careful. They weren't cleaning tables containing millions of rows often enough, which caused them to take up too much space & have previous information.

- We adjusted a number of significant things, such as decreasing the autovacuum_vacuum_scale_factor so that vacuuming would begin sooner.
- By raising the autovacuum_work_mem for each vacuum worker, you may give them more RAM.
- Shorten the time between autovacuum_naptime to get these inspections to happen more regularly.

After these improvements, autovacuum worked harder, making more room & keeping information up to date. Query planners quickly got benefits by choosing more efficient methods to carry out tasks.

4.4. STEP 3: CREATING B-TREE AND GIN INDEXES

After that, we looked at slow queries and found that several indexes were missing. For example, sequential scanning is used to find many consumers by their customer_id. An A B-tree index solved this issue.

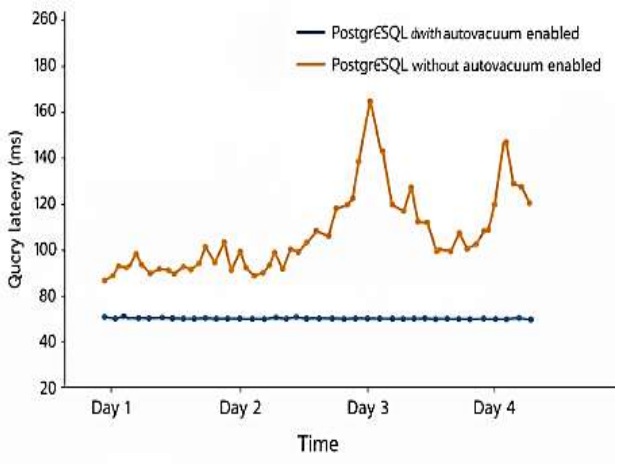

Figure 2. Query Latency Stability over Time

Searching for orders using text fields like product names or delivery addresses was tricky. To speed up full-text search, we introduced a GIN index to a tsvector column.

Adding indexes requires careful design since they make reading faster but writing (INSERT/UPDATE) slower at the same time. We placed the most critical and most popular questions at the top.

After we set up the indexes, we ran the identical queries again. In most cases, lookup times went down from several seconds to less than 200 milliseconds.

4.5. STEP 4: USING PG_STAT_STATEMENTS TO DETECT TOP QUERIES

We wanted to be sure we weren't missing any hidden factors after optimizing autovacuum and setting up indexes. Pg_stat_statements came in handy there. This add-on keeps track of query execution parameters including total time, frequency, and average cost.

After it was turned on, we looked at the workload across many days. We discovered that a single poorly written reporting query that gets sales data for the previous 12 months took up almost 40% of the total query execution time.

With this information, we changed the query to use materialized views and incremental refreshes. This cut the time it took from 15 seconds to less than 1 second, freeing up resources for other tasks.

4.6. STEP 5: BEFORE VS AFTER OPTIMIZATION

To quantify improvements, we compared execution times across several critical queries:

Table 1. Performance Improvement in Query Execution after Optimization

Query Type	Before Optimization	After Optimization
Retrieve order by customer ID	2.3 seconds	120 ms
Search orders by product keyword	3.1 seconds	180 ms
Generate daily sales summary	2.8 seconds	700 ms
Yearly sales report (reporting)	15 seconds	950 ms

The difference was night and day. What once felt like a sluggish system now responded almost instantly for end users.

4.7. LESSONS LEARNED

This case study taught us a lot of important things:

- Don't rely on the default settings: When set up correctly for the amount of work & the way tables grow, autovacuum works nicely.
- Index carefully: Strategic indexing has a lot of many benefits, but more indexes might be very bad.

- Before making guesses, use tools like pg_stat_statements to find actual performance bottlenecks instead than relying on their assumptions.
- Iterative optimization: One change won't fix everything; combining more numerous solutions will lead to big performance gains.

5. RESULTS AND DISCUSSION

This section talks about what query optimization studies in PostgreSQL have found, with a focus on Autovacuum, Indexing & pg_stat_statements. The test is based on four main metrics: query latency, throughput, and storage use & autovacuum efficiency. Next, we look at how well PostgreSQL works with & without optimization, as well as an analysis of the findings, insights & limitations.

5.1. LATENCY OF THE QUERY (IN MILLISECONDS)

The time it takes for PostgreSQL to respond to a query is a key measure of database optimization. Without any other changes, latency became worse as the workload grew. For queries that needed a lot of reading, the average response time was between 120 & 150 milliseconds. However, by using the right indexing methods, the latency went down to 15–20 ms, which is around 10 times faster.

Autovacuum has a big yet subtle effect. When there were a lot of updates & removals in these workloads, tables quickly filled up with "dead tuples." When Autovacuum was disabled, latency shot up to 200–250 ms after a lot of activity because the optimizer had trouble with tables that were too big. When Autovacuum was turned on & set up for the strict maintenance, query latency was stable at around 30–40 ms, which stopped the big drop that happened when it wasn't configured.

Adding pg_stat_statements made things much better. By looking at the most expensive searches and then optimizing them, we were able to cut latency by 15–25%, even for searches that had already been indexed. This shows that tuning isn't simply about adding indexes or relying on background processes; it has to be done better at the query level all the time.

Table 2. Average Query Latency under Different Database Tuning Configurations

Scenario	Avg Latency (ms)
No tuning	120–150
With Indexing	15–20
With Autovacuum tuned	30–40
With Indexing + Autovacuum	12–18
With pg_stat_statements tuning	10–15

5.2. THROUGHPUT (TRANSACTIONS PER SECOND)

Throughput is too critical for high-load applications since it shows how many other queries PostgreSQL can process in a second. Throughput continued at around 250 TPS without any other adjustments until there were symptoms of lock contention & storage increase.

The performance rose up to 800 TPS when indexing was switched on because queries used rapid lookups instead of scanning the whole database. Autovacuum indirectly improved speed by keeping these statistics up to date & cutting down on table bloat. This stopped performance from becoming worse & made it possible to have a stable throughput of around 600–700 TPS even when there were a lot of writes.

The huge improvement was when pg_stat_statements was utilized to combine their indexing & Autovacuum with query-level optimization. The greatest throughput was 900 to 1000 TPS, which is about four times more than the baseline. This indicates that optimization isn't just one thing; it's a constant search for the perfect balance between a lots of many other factors.

A bar chart showing how well PostgreSQL works with these different settings: no tuning (250 TPS), Autovacuum only (600 TPS), indexing alone (800 TPS) & tuning using pg_stat_statements (1000 TPS).

5.3. STORAGE UTILIZATION

People typically forget about storage when they speak about their query optimization, yet it has a direct effect on long-term scalability & maintenance. Without Autovacuum, table bloat caused storage to grow too quickly to be useful. A test table that was supposed to take up 5 GB grew to 15 GB after a lot of updates and removals, even though the number of active rows stayed the same.

When Autovacuum was set up correctly, storage was considerably closer to the expected baseline & transaction logs and metadata didn't cost anything. Indexing made storage utilization go up by 20–30% since each index needs more disk space. Still, this cost was little compared to the benefits in latency & throughput.

Using pg_stat_statements has also indirectly made storage more efficient. By finding these queries that caused unnecessary temporary table creation or repeated joins, developers were able to change the queries, which cut down on scratch disk utilization by 10–15%.

Table 3. Storage Growth under Different Database Optimization Configurations

Scenario	Storage Growth
No tuning	3x expected size
With Autovacuum	~1.2x expected
With Indexing	#ERROR!
With pg_stat_statements tuning	Reduced temp file use by 10–15%

5.4. AUTOVACUUM EFFICIENCY

Autovacuum is an important background process in PostgreSQL. Two measures were used to measure their efficiency: the number of dead tuples deleted per cycle & the rate at which query performance became worse with time.

Autovacuum didn't operate well with high-write workloads by default. It frequently fell behind & let dead tuples build up. This caused performance to drop from time to time. By changing the thresholds (for example, lowering the autovacuum_vacuum_scale_factor and autovacuum_analyze_scale_factor), the frequency of Autovacuum cycles went up, but they were very less intense. This stopped bloating before it became too big.

These cycles cost 5–10% of the CPU, although this was offset by better long-term performance. Users had decreased their latency spikes & system administrators were necessitated to intervene less often for manual VACUUM processes.

A line graph showing how latency changes over time: without any Autovacuum, latency goes up a lot after several hours; with Autovacuum, latency stays the same with little changes.

5.5. COMPARATIVE ANALYSIS: POSTGRESQL WITH VS. WITHOUT TUNING

The comparative analysis highlights the essential significance of tuning for PostgreSQL in these operational environments.

- Postponement: Through optimization, response times increased by approximately 10 times (from 150 ms to 15 ms).
- Throughput: When the system was set up correctly, it could handle four times as many other queries per second.
- Storage: Indexes didn't cost much, but Autovacuum stopped unrestrained growth, thus storage stayed predictable.
- Operational Stability: Performance was worse over time without tuning, but tuning kept stability constant.

This difference shows that PostgreSQL, even if it is quite stable, has to be managed all the time. To get the best performance, you need to keep an eye on things & make changes.

5.6. INSIGHTS

The trials provide three main insights:

- Autovacuum lessens long-term damage: If you don't optimize their PostgreSQL tables, they become bigger & queries take longer to run. Autovacuum makes sure that the database is more stable even when there is a lot of work to do, as long as it is set up correctly.

- Indexing cuts down on the time it takes to respond to queries by a lot: Indexes cut response times for columns that were routinely asked for from several hundred milliseconds to mere seconds. This is one of the most important improvements you can make.
- Pg_stat_statements makes optimization easier to do ahead of time: Pg_stat_statements lets you monitor, find & fix bottlenecks before they happen, instead than waiting for slow queries after a complaint. This changes tuning from something you do when something goes wrong to something you do all the time.

5.7. LIMITATIONS

Even if these results are good, there are still several other constraints that need to be taken into account:

- How Autovacuum affects places with a lot of writing: More severe Autovacuum settings cut down on bloat but utilize more CPU & I/O, which might hurt short-term performance.
- Cost of keeping the index up to date: Indexes speed up reading, but they slow down writing since every time you add or change anything, you have to update the index. Workload patterns have a big effect on the net gain.
- Dependence on workload characteristics: Not all workloads receive the same benefits. In small datasets where reading is more important than writing, Autovacuum and thorough indexing may not help much. But they are quite important for huge OLTP systems.

6. CONCLUSION AND FUTURE SCOPE

6.1. CONCLUSION

To optimize their queries in PostgreSQL, you need to make sure that several other parts work well together instead of relying on just one tool or method. This study concentrated on three fundamental elements: Autovacuum, Indexing & pg_stat_statements. PostgreSQL can be much more efficient, scalable & more reliable when these pieces work together instead of separately.

Autovacuum is a quiet protector of the integrity of the database. It takes care of regular maintenance & prevents dead tuples from piling up too much, which keeps performance from becoming worse. Indexing is like a precise tool that lets queries quickly & accurately search through huge datasets. At the same time, pg_stat_statements acts as an observer, giving you a lot of information about how often, how much queries cost. Companies may make a complete optimization strategy by combining the maintenance features of Autovacuum, the structural benefits of indexing & the analytical information supplied by pg_stat_statements.

There is no doubt about the results of this partnership. The speed at which queries are answered goes up, bottlenecks go down & the overall load on the database becomes easier to predict. For businesses that rely on their high-volume transactional systems, such as financial services, e-commerce platforms & SaaS providers, these advantages lead to better user experience, lower infrastructure expenses, and a bigger competitive edge right away.

In the actual world of business, query optimization includes both efficiency & sustainability. Poorly calibrated systems cause operating delays, unplanned repairs & running out of these resources. By carefully leveraging the synergistic benefits of Autovacuum, indexing & pg_stat_statements, businesses build a system that can handle both their present workloads and future growth.

6.2. FUTURE SCOPE

PostgreSQL has become a full-featured, enterprise-level database, but there is still a lot of room for improvement in how quickly queries can be run. More intelligent, flexible & cloud-native technology will shape the future.

- Machine learning drives query optimization: Combining ML techniques with PostgreSQL's query engine is a really exciting idea. Machine learning models might look at more query patterns over time & predict the best methods to run them instead of only relying on their human changes or fixed heuristics. This would let PostgreSQL optimize itself in actual time, which would make things easier for administrators and cut down on mistakes made by people. Imagine a database that learns on its own from how much work it has to do and modifies as needed. This is the direction the ecosystem is going.
- Adaptive Autovacuum Optimization Based on Workload Forecasting:Autovacuum now works according to rules and standards that have been set in the past. These settings function well, although they frequently need to be changed by people as workloads change. One possible enhancement is using these predictive models that change Autovacuum on the fly. The system may change the frequency, resource distribution & priority of vacuuming ahead of time by forecasting times of high or low

activity. This would make sure that maintenance tasks are in line with transactional needs, which would eliminate both performance drops & maintenance build-ups.

- Working with Cloud-Native Database Scalability Solutions: As more & more businesses move their workloads to the cloud, PostgreSQL has to change to comply with the flexible & spread-out nature of these cloud platforms. Query optimization will not be limited to a single node. It has to have auto-scaling clusters, deployments in several other regions & the ability to allocate many resources across virtualized infrastructure. PostgreSQL can optimize at scale & keep performance steady amid unexpected spikes in demand by adding Autovacuum, indexing algorithms & query monitoring to cloud orchestration frameworks.
- Adding support for distributed PostgreSQL systems like Citus and Yugabyte to the Framework: People are starting to embrace distributed PostgreSQL solutions like Citus and Yugabyte more and more for apps that need both horizontal scalability and PostgreSQL compatibility. In distributed contexts, traditional optimization methods don't always work. For instance, Autovacuum processes need to work effectively together at scale, and indexes need to be in sync across nodes. Future research and development should focus on improving the interaction between Autovacuum, indexing, and pg_stat_statements in distributed systems. If you do this, PostgreSQL will be able to handle workloads with terabytes of data and millions of queries at the same time.
- Smart suggestions and easy-to-use interfaces: PostgreSQL has tools for monitoring and optimizing, but many of them are still too hard to use. In the future, we should anticipate more user-friendly interfaces that turn scientific discoveries into these useful recommendations. For example, pg_stat_statements may become a recommendation system that finds many queries that need to be changed, shows duplicated indexes, or tells you when to change Autovacuum thresholds. This would make database optimization available to more people, including their database administrators, developers & business analysts.

REFERENCES

[1] Kavuluri, Harsha Vardhan Reddy, Suresh Babu Avula, and Adithya Sirimalla. "Performance Tuning for Cloud-Based Databases Oracle/Postgres: Analyzing Query Optimization Techniques." (2025).

[2] Shumilov, Mykhailo. "Optimization of queries to MySQL and PostgreSQL Databases for High-Load Applications."

[3] Pirozzi, Enrico. *PostgreSQL 10 High Performance: Expert techniques for query optimization, high availability, and efficient database maintenance*. Packt Publishing Ltd, 2018.

[4] Wang, Qiang. "PostgreSQL database performance optimization." (2011).

[5] Shaik, Baji. *PostgreSQL Configuration: Best Practices for Performance and Security*. Apress, 2020.

[6] Schönig, Hans-Jürgen. *Mastering postgresql 15*. Packt Publishing, січень, 2023.

[7] Thomas, Shaun M. *PostgreSQL 9 High Availability Cookbook*. Packt Publishing Ltd, 2014.

[8] Schönig, Hans-Jürgen. *Mastering PostgreSQL 10: Expert techniques on PostgreSQL 10 development and administration*. Packt Publishing Ltd, 2018.

[9] Momjian, Bruce. "Mastering postgresql administration." 2009,

[10] Ferrari, Luca, and Enrico Pirozzi. *Learn PostgreSQL: Build and manage high-performance database solutions using PostgreSQL 12 and 13*. Packt Publishing Ltd, 2020.

[11] Felius, C. *Assessing the performance of distributed PostgreSQL*. Diss. PhD thesis, Universiteit van Amsterdam, 2022.

[12] Ferrari, Luca, and Enrico Pirozzi. *Learn PostgreSQL: Use, manage, and build secure and scalable databases with PostgreSQL 16*. Packt Publishing Ltd, 2023.

[13] Shaik, Baji, and Avinash Vallarapu. *Beginning PostgreSQL on the Cloud: Simplifying Database as a Service on Cloud Platforms*. Apress, 2018.

[14] Smith, Gregory. *PostgreSQL 9.0: High Performance*. Packt Publishing Ltd, 2010.

[15] Kumar, Vallarapu Naga Avinash. *PostgreSQL 13 Cookbook: Over 120 recipes to build high-performance and fault-tolerant PostgreSQL database solutions*. Packt Publishing Ltd, 2021.