Edge™

< Blog Home

PostgreSQL Performance Tuning

Ibrar Ahmed | 31 January 2025

PostgreSQL is already known for its reliability, extensibility, and open-source pedigree and continues to grow and evolve with each release. PostgreSQL 17 introduces several performance improvements and features that make it a powerhouse for OLTP (Online Transaction Processing) and OLAP (Online Analytical Processing) workloads.

Feature	PostgreSQL 15	PostgreSQL 16	PostgreSQL 17
Logical Replication	Basic support	Improved failover recovery	Seamless failover slots
Parallel Query Support	Limited	Better parallel joins	Expanded parallel aggregates
Incremental Sort	Initial implementation	More scenarios supported	Optimized for large datasets
WAL Compression	Introduced	Improved	Faster and more efficient
Indexing	Basic deduplication	BRIN enhancements	Multi-column BRIN, better B-Tree

Autovacuum	Basic thresholds	Smarter activity-based tuning	Adaptive thresholds

This blog will explore advanced performance tuning techniques for PostgreSQL 17 and highlight key improvements compared to versions 15 and 16.

Performance Improvements in PostgreSQL 17

PostgreSQL 17 builds on the features of PostgreSQL 15 and 16, delivering optimizations that address latency, scalability, and manageability. Some of the standout enhancements include:

Logical Replication Enhancements

- **Failover Slot Synchronization:** PostgreSQL 17 introduces failover slots that ensure logical replication can seamlessly continue after failover events. This feature reduces downtime and data loss during failovers, significantly improving PostgreSQL 16's partial support for logical replication recovery.
- **Memory Optimization:** Logical decoding in PostgreSQL 17 now consumes less memory during high transaction loads. This refinement improves stability for replication-heavy systems.

Query Planner Optimizations

- **Incremental Sort Enhancements:** Incremental sort, introduced in PostgreSQL 13, has been further optimized in PostgreSQL 17 to handle larger datasets with minimal memory usage. The planner now evaluates query costs more accurately, reducing execution times for queries with complex sorting requirements.
- **Parallel Query Enhancements:** PostgreSQL 17 expands parallelism for queries involving `FULL OUTER JOIN` and aggregates, offering significant performance boosts compared to PostgreSQL 16.

Storage and Vacuum Improvements

- **Autovacuum Tuning:** PostgreSQL 17 introduces smarter autovacuum thresholds that adapt based on table activity. Compared to PostgreSQL 16, this results in fewer, more efficient vacuum operations, particularly for high-write workloads.
- **Checksum Verification:** Checksum functionality has been optimized to reduce overhead, ensuring data integrity checks without impacting I/O performance.

WAL Compression

- **Enhanced WAL Compression:** While PostgreSQL 15 introduced basic WAL compression, PostgreSQL 17 refines this feature with faster algorithms and better integration with modern storage systems. This reduces the write-ahead log size without compromising write throughput.

Indexing Advancements

- **BRIN Index Improvements:** Block Range Indexes (BRIN) have been further optimized in PostgreSQL 17 to support a broader range of use cases, including multi-column indexes with better update performance than PostgreSQL 16.
- **Improved B-Tree Deduplication:** B-tree deduplication now applies to more scenarios, reducing index bloat and improving lookup speeds.

Tuning PostgreSQL 17 for Maximum Performance

To maximize PostgreSQL 17's potential, you must tune various aspects of your database. Here's a comprehensive guide:

Workload Analysis

Understanding your database workload is crucial for practical tuning. Workloads can be broadly categorized as OLTP or OLAP; each type benefits from different optimizations.

OLTP Workloads

- **Focus Areas:**

- Low-latency transaction processing.
- High concurrency.
- Maximizing transaction throughput.

- **Optimization Steps:**
  - **Connection Pooling:** Use a lightweight pooling tool like PgBouncer to maintain optimal connection levels.
  - **Indexing:** Create indexes tailored for frequently accessed tables to speed up queries.
  - **Vacuuming:** Ensure regular autovacuuming to prevent table bloat, especially in high-update scenarios.
  - **Lock Management:** Monitor lock contention with the pg_stat_activity and pg_locks views and optimize queries that are causing bottlenecks.

OLAP Workloads

- **Focus Areas:**
  - Efficient processing of large, complex queries.
  - Handling of extensive table scans, aggregations, and joins.

- **Optimization Steps:**
  - **Parallel Queries:** Enable parallel query execution and allocate sufficient resources (`parallel_workers_per_gather`).
  - **Partitioning:** Use table partitioning for large datasets to improve query performance.
  - **Incremental Sorts:** Leverage PostgreSQL 17's improved incremental sort functionality for sorting large datasets.

Key Parameters for PostgreSQL Performance Tuning

The following parameters allow you to refine your PostgreSQL database performance.

Memory Parameters

Memory allocation significantly affects database performance. PostgreSQL 17 provides better memory management tools to cater to varying

workloads.

- shared_buffers
  Allocates memory for caching frequently accessed data. Set to 25–40% of total RAM, with a cap of 8GB for large systems unless benchmarking suggests additional benefits. Monitor `pg_stat_activity` and `pg_stat_database` to achieve a cache hit ratio of 99% or more.
- work_mem
  Determines memory for sorting and hash joins. Set to 4–16MB per connection, scaling based on query complexity. Use `EXPLAIN` (`ANALYZE`) to identify operations needing larger memory allocations.
- maintenance_work_mem
  Dedicated to `VACUUM`, indexing, and maintenance tasks. Defaults to 256MB–1GB, but temporary increases during bulk operations like `VACUUM FULL` or `CREATE INDEX` can improve performance.

I/O and WAL Parameters

- wal_buffers
  Adjusts memory allocation for WAL writes. A default value of 16MB is typically sufficient, but increasing this value can improve performance for write-intensive workloads. Monitoring pg_stat_bgwriter can help you determine if higher values are needed.
- max_wal_size
  Controls WAL segment growth. Rather than using vague terms like "large," a geometric increase method should be applied. In tests with HammerDB, performance improvements were observed when increasing beyond 50GB, ensuring no requested checkpoints occurred. Use pg_stat_bgwriter to monitor WAL activity and optimize this setting.
- wal_compression
  Enables WAL compression to reduce disk I/O at the expense of CPU usage. This is beneficial for I/O-bound workloads where reducing disk writes is a priority. Evaluate the trade-off based on system capabilities.

Query and Indexing Parameters

- effective_cache_size
  Estimates the OS-level file system cache available for query planner optimization. Typically, set to 50–75% of total system memory to ensure efficient index usage and avoid unnecessary sequential scans.
- max_parallel_workers_per_gather
  Controls the number of parallel query workers. Increasing this value for complex queries on large datasets helps leverage PostgreSQL 17's improved parallelism.
- autovacuum_vacuum_cost_limit
  Determines how much work the autovacuum can perform before pausing. For high-write workloads, increasing this value ensures timely vacuuming and prevents wraparound failures.
- random_page_cost
  Represents the cost of a non-sequential disk page fetch relative to a sequential one. The default is 4.0, but on modern SSDs or high-performance disks, it is almost universally recommended to reduce it to 1.1–2.0. This adjustment better reflects the random I/O capabilities of current storage technology and encourages index scans where beneficial.

Optimization Strategy

- Monitor `pg_stat_bgwriter` for WAL activity so you know how to size `wal_buffers` and `max_wal_size` properly.
- Use geometric increases for WAL settings rather than vague "large" values.
- Adjust `random_page_cost` appropriately for SSDs to favor index scans.
- Set `effective_cache_size` to reflect available memory for better planner decisions.
- Tune `autovacuum_vacuum_cost_limit` to ensure vacuum processes are completed efficiently in high-write environments.

Connection and Pooling Parameters

- max_connections
  Limits the number of active connections. Overloading leads to performance bottlenecks. Use a connection pooler like PgBouncer to manage high concurrency efficiently.
- idle_in_transaction_session_timeout
  Prevents long-running idle transactions from holding locks. A timeout of 5–15 minutes is recommended for transactional workloads.

WAL Configuration

Write-ahead logging (WAL) ensures data durability and consistency, but improper configuration can impact write-heavy workloads.

- wal_compression
  Compresses WAL records to reduce I/O overhead for I/O-intensive workloads. This parameter reduces disk space usage but slightly increases CPU overhead. Test the impact on your system to confirm net benefits.
- checkpoint_timeout and max_wal_size
  Controls the frequency and size of checkpoints increase `checkpoint_timeout` to 10–15 minutes and `max_wal_size` to a minimum of 1GB–2GB for write-heavy workloads. Longer intervals reduce I/O overhead but increase recovery time after crashes.
- wal_buffers
  Acts as a temporary buffer for WAL records before they're written to disk; set to at least 16MB for high-write workloads.
- checkpoint_completion_target
  This parameter determines how evenly the checkpoint I/O is distributed over the checkpoint interval. A value close to 1.0 (e.g., 0.9) spreads the I/O workload more evenly, reducing I/O spikes and performance degradation during checkpoints.

Autovacuum Settings

PostgreSQL's autovacuum process manages table bloat and transaction wraparound prevention. PostgreSQL 17's smarter thresholds make it even

more efficient.

- autovacuum_vacuum_cost_limit
  This parameter limits the cost of vacuum operations to minimize their impact on active queries. Increase the setting for high-write tables to ensure timely cleanup. Note that this setting is typically set to -1, meaning it defers to vacuum_cost_limit. However, it can be useful in conjunction with `ALTER TABLE` to apply different limits per table.
- autovacuum_freeze_max_age
  This parameter prevents transaction ID wraparound issues. Monitor pg_stat_all_tables for age (relfrozenxid) and adjust the threshold as needed. Higher values reduce the frequency of freezes, spreading out I/O load to prevent stampedes when multiple tables need freezing at the same time.

Monitoring Autovacuum

Use the pg_stat_all_tables view to track autovacuum activity and ensure it keeps up with your workload. Understanding how **freeze max age impacts freeze intervals** can help optimize autovacuum behavior across different tables.

Query Optimization

Efficient query design is crucial for leveraging PostgreSQL 17's advanced features.

Steps for Query Optimization

Address the following key point to optimize your queries:

- **Analyze Execution Plans**
  ○ Use EXPLAIN (ANALYZE) to identify slow queries and bottlenecks.
  ○ Optimize joins, avoid sequential scans for large datasets, and leverage indexes.
- **Leverage Parallel Queries**
  ○ Enable parallel queries for aggregations and joins using `parallel_setup_cost` and `parallel_tuple_cost`.

- Adjust `parallel_workers_per_gather` for optimal parallelism.

- **Incremental Sort**
  - PostgreSQL 17 improves incremental sort performance. Ensure queries with ORDER BY clauses are structured to use this feature.

Using query optimization effectively

- Ensure indexes match the ORDER BY clause to take advantage of sorting optimizations.
- Use EXPLAIN ANALYZE to verify if the query planner is using incremental sort.
- Consider breaking large sorts into smaller groups using LIMIT and OFFSET where applicable.
- Avoid unnecessary sorting operations by leveraging index scan strategies.

Avoid Over-Indexing

- Excessive indexing can lead to higher maintenance overhead; you should only create indexes for frequently queried columns.

Monitoring Index Usage:

- Use `pg_stat_user_indexes` to identify indexes that are rarely or never used.
- To find unused indexes, run the following query:

```
SELECT schemaname, relname, indexrelname, idx_scan
FROM pg_stat_user_indexes
WHERE idx_scan = 0;
```

- This query retrieves indexes that have never been scanned, indicating they might be unused.

- Regularly check index bloat with `pg_stat_all_indexes` and consider reindexing or dropping unused indexes.
- Balance indexing with query performance and maintenance overhead to optimize database efficiency.

Index Optimization

Indexes are pivotal for improving query performance, and PostgreSQL 17's indexing advancements offer additional opportunities for optimization. You should evaluate your index use to ensure that you follow some best practices:

- **Use BRIN Indexes**
  - BRIN (Block Range Indexes) are ideal for large, sequentially stored datasets such as time-series data.
  - BRIN summarizes data blocks rather than indexing each row, reducing storage requirements.
- **Regular Index Maintenance**
  - **Monitor Index Bloat:** Use the `pg_stat_user_indexes` view to identify bloated indexes.

```
SELECT
    schemaname || '.' || relname AS table_name,
    indexrelname AS index_name,
    pg_size_pretty(pg_relation_size(indexrelid)) AS index_size,
    pg_size_pretty(pg_relation_size(relid)) AS table_size,
    (pg_relation_size(indexrelid)::numeric / NULLIF(pg_relation_size(relid), 0)
FROM
    pg_stat_user_indexes
JOIN
    pg_class ON pg_stat_user_indexes.indexrelid = pg_class.oid
WHERE
    pg_relation_size(indexrelid) > 10000000 -- Filter indexes larger than ~10MB
ORDER BY
    index_to_table_ratio DESC;
```

- If an index is disproportionately large compared to its associated table, it may be bloated and should be prioritized for maintenance to improve efficiency. A high index-to-table size ratio, such as values greater than 1.0, can indicate potential inefficiencies caused by index bloat, impacting query performance and storage utilization.
  ○ **Rebuild Indexes:** Run REINDEX periodically to prevent performance degradation from bloated indexes.
- **Multi-Column Indexes**
  ○ PostgreSQL 17 enhances BRIN and B-tree indexes to support multi-column indexing more efficiently. Leverage this for queries with multiple WHERE conditions.

Connection Pooling

Handling high-concurrency workloads efficiently requires connection pooling tools like PgBouncer. pgBouncer minimizes connection establishment overhead by reusing existing connections. To fully realize the benefits of connection pooling you should:

- **Optimize Settings**
  ○ Configure pgBouncer pooling modes (session, transaction, statement) based on your application's behavior.
  ○ Set `max_client_conn` and `default_pool_size` to balance connection limits and application requirements.
- **Regularly Monitor your Application**
  ○ Use PgBouncer's admin console to monitor connection usage and adjust settings dynamically.

Conclusion

PostgreSQL 17 represents a significant leap in database performance, offering a range of optimizations for modern workloads. From seamless failover in logical replication to more brilliant autovacuum mechanisms and improved indexing, PostgreSQL 17 sets a new standard. While PostgreSQL 15 and 16

introduced crucial features, the refinements in PostgreSQL 17 make it a compelling choice for enterprises looking to boost efficiency and scalability.

Whether managing high-concurrency OLTP systems or running complex analytical queries, PostgreSQL 17 has the tools and improvements to meet your needs. This version can unlock unprecedented performance levels with careful tuning, ensuring your database infrastructure is future-ready.

pgEdge supports PostgreSQL 17 and builds on its strong foundation to provide fully distributed Postgres that delivers multi-master capability and the ability to go multi-region and multi-cloud. pgEdge adds essential features such as conflict management, conflict avoidance, automatic DDL replication, and more to cater to the demands of always on, always available, and always responsive global applications. pgEdge distributed Postgres is available for download at https://www.pgedge.com/get-started/platform or accessed in GitHub.

About the Author

Ibrar Ahmed

Principal Engineer

Ibrar Ahmed is a seasoned software professional who began his tech journey with a deep dive into system-level embedded development.

SUBSCRIBE TO BLOG

Email*

SUBMIT

Get started today.

Experience the magic of pgEdge Distributed PostgreSQL now.

SIGN UP FREE

Hot Topics

Distributed Postgres

Postgres Replication

PostgreSQL High Availability

Multi-master

PostgreSQL Download

Products

pgEdge Cloud

pgEdge Platform

Plans & Pricing

Company

About Us

Contact

Support

Solutions

Low Data Latency

Data Residency

PostgreSQL High Availability

Resources

Blog

Webinars

Documentation

Multi-master

Videos

pgEdge™ provides distributed PostgreSQL to reduce data latency and achieve PostgreSQL high availability. Only pgEdge combines multi-master, multi-region using fully managed cloud or self-hosted platform that is 100% standard Postgres and 100% open (source available)

Contact Us
Careers
Terms of Use
Privacy Policy
Community License
Cookie Policy