MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

DAY – 6

TOPIC: Postgres Performance Tuning & Query Optimization

Mind Map

⚙️ 1. Configuration Tuning

🧠 Memory Parameters

Parameter	Purpose	Example Script
shared_buffers	Memory for caching data pages	ALTER SYSTEM SET shared_buffers = '4GB';
work_mem	Memory per sort/hash operation	ALTER SYSTEM SET work_mem = '64MB';
maintenance_work_mem	Memory for maintenance tasks (e.g., VACUUM)	ALTER SYSTEM SET maintenance_work_mem = '512MB';
effective_cache_size	Estimate of OS-level cache available	ALTER SYSTEM SET effective_cache_size = '12GB';

SELECT name, setting FROM pg_settings

WHERE name IN ('shared_buffers', 'work_mem', 'maintenance_work_mem', 'effective_cache_size');

💡 **Tip**: Use pg_reload_conf(); after changes.

💾 I/O Behavior

Parameter	Purpose	Example Script
wal_buffers	Memory for WAL before writing to disk	ALTER SYSTEM SET wal_buffers = '16MB';
checkpoint_timeout	Time between checkpoints	ALTER SYSTEM SET checkpoint_timeout = '15min';

Parameter	Purpose	Example Script
checkpoint_completion_target	Spread checkpoint I/O load	ALTER SYSTEM SET checkpoint_completion_target = 0.9;

SELECT name, setting FROM pg_settings

WHERE name LIKE 'checkpoint%' OR name = 'wal_buffers';

👥 Concurrency Controls

Parameter	Purpose	Example Script
max_connections	Total allowed connections	ALTER SYSTEM SET max_connections = 200;
max_worker_processes	Background workers for parallel tasks	ALTER SYSTEM SET max_worker_processes = 16;

SELECT name, setting FROM pg_settings

WHERE name IN ('max_connections', 'max_worker_processes');

🔍 2. Query Optimization

🐢 Identify Slow Queries

EXPLAIN ANALYZE SELECT * FROM orders WHERE customer_id = 123;

💡 **Insight**: Look for sequential scans, high cost estimates, and nested loops.

✍️ Rewrite Queries

Before:

SELECT * FROM orders WHERE customer_id IN (SELECT id FROM customers WHERE region = 'APAC');

After:

```
SELECT o.* FROM orders o
JOIN customers c ON o.customer_id = c.id
WHERE c.region = 'APAC';
```

💡 **Insight**: Joins often outperform subqueries when indexed properly.

🧱 3. Index Tuning

🏗️ Create Appropriate Indexes

```
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
CREATE INDEX idx_customers_region ON customers(region);
```

💡 Use GIN for full-text search, GiST for geometric data.

🩹 Maintain Indexes

```
REINDEX INDEX idx_orders_customer_id;
-- Or concurrently:
REINDEX INDEX CONCURRENTLY idx_orders_customer_id;
```

📊 Update Statistics

```
ANALYZE orders;
```

💡 Run after bulk inserts or index changes.

🖥️ 4. Hardware Optimization

🧮 Resource Sizing

- Use htop, vmstat, and iostat to monitor CPU/memory/disk.
- Ensure shared_buffers ≈ 25–40% of RAM.
- SSDs drastically reduce I/O latency.

💽 Storage Performance

- Prefer RAID 10 for balance of speed and redundancy.
- Use pg_test_fsync to benchmark disk sync performance.

pg_test_fsync

🔄 **5. Regular Maintenance**

🧹 **VACUUM and ANALYZE**

VACUUM ANALYZE orders;

-- Or for aggressive cleanup:

VACUUM FULL orders;

💡 Prevents table bloat and keeps planner stats fresh.

🧠 **Bonus: Monitoring Slow Queries**

Enable logging:

ALTER SYSTEM SET log_min_duration_statement = 1000; -- in ms

SELECT pg_reload_conf();

Then check logs:

tail -f /var/lib/pg/data/log/postgre.log

PostgreSQL performance tuning is a multifaceted process that involves optimizing various layers of your database system, from the initial design to the underlying hardware and operating system. The goal is to maximize efficiency, reduce latency, and ensure your database can handle its workload effectively. Unlike some other databases, PostgreSQL offers a high degree of configurability, allowing you to tailor its behavior for specific use cases, whether you prioritize frequent reads, writes, or a balanced workload.

Let's delve into the details of PostgreSQL performance tuning with real-time examples and practical scripts.

How to Tune PostgreSQL Performance: A Detailed Approach

1. Database Design: The Foundation of Performance

Database design is paramount for performance. A well-structured schema can dramatically reduce the resources needed for queries.

a. Data Partitioning

Instead of a single, monolithic table, partitioning divides a large table into smaller, more manageable pieces. This can significantly improve query performance by allowing the database to scan only the relevant partitions, reducing the amount of data processed.

Real-time Example: Imagine a sales table with billions of records, storing sales transactions from multiple years.

```
-- Original large table
CREATE TABLE sales (
  sale_id SERIAL PRIMARY KEY,
  product_id INT,
  customer_id INT,
  sale_date DATE,
  amount DECIMAL(10, 2)
);
```

As this table grows, queries filtering by sale_date will become slow. We can partition it by sale_date for annual or monthly partitioning.

PostgreSQL Script for Range Partitioning (by year):

```
-- Create the master partitioned table
```

```
CREATE TABLE sales_partitioned (
    sale_id SERIAL,
    product_id INT,
    customer_id INT,
    sale_date DATE,
    amount DECIMAL(10, 2)
) PARTITION BY RANGE (sale_date);

-- Create partitions for specific years
CREATE TABLE sales_2023 PARTITION OF sales_partitioned
FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE sales_2024 PARTITION OF sales_partitioned
FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');

CREATE TABLE sales_2025 PARTITION OF sales_partitioned
FOR VALUES FROM ('2025-01-01') TO ('2026-01-01');

-- Add data (PostgreSQL automatically routes to the correct partition)
INSERT INTO sales_partitioned (product_id, customer_id, sale_date, amount) VALUES
(101, 501, '2023-03-15', 150.00),
(102, 502, '2024-07-20', 200.50),
(103, 503, '2025-01-10', 75.25);
```

How to Deal with This Issue: When querying for sales within a specific year, PostgreSQL's query planner will only scan the relevant partition, significantly reducing I/O and CPU usage.

```
-- Querying sales for a specific year
EXPLAIN ANALYZE
```

```
SELECT SUM(amount)
FROM sales_partitioned
WHERE sale_date BETWEEN '2024-01-01' AND '2024-12-31';
```

You'll observe that the EXPLAIN ANALYZE output will show a "Partition Pruning" step, indicating that only the sales_2024 partition was scanned.

b. Indexing

Indexes are crucial for speeding up data retrieval by providing a quick lookup mechanism for specific columns. However, overuse can be detrimental.

Real-time Example: Consider a users table where you frequently search for users by their email address.

```
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  username VARCHAR(50) NOT NULL,
  email VARCHAR(100) UNIQUE NOT NULL,
  registration_date DATE
);
```

Without an index on email, a query like SELECT * FROM users WHERE email = 'test@example.com' would perform a full table scan.

PostgreSQL Script to Create an Index:

```
-- Create an index on the email column
CREATE INDEX idx_users_email ON users (email);
```

How to Deal with This Issue: After creating the index, the same query will utilize the index for faster lookup.

```
EXPLAIN ANALYZE
SELECT *
FROM users
WHERE email = 'test@example.com';
```

You should see an "Index Scan" or "Index Only Scan" in the EXPLAIN ANALYZE output.

Caution on Over-indexing: Every index adds overhead to INSERT, UPDATE, and DELETE operations because the index also needs to be updated. Too many indexes on a frequently modified table can slow down write operations. Regularly analyze your query patterns and remove unused or redundant indexes.

PostgreSQL Script to Check Index Usage:

```
SELECT
  schemaname,
  relname AS table_name,
  indexrelname AS index_name,
  idx_scan,
  idx_tup_read,
  idx_tup_fetch
FROM
  pg_stat_user_indexes
ORDER BY
  idx_scan DESC;
```

This query shows how often an index has been used (idx_scan). Low idx_scan values for a long period might indicate an unused index.

2. Hardware Tuning: Optimizing the Physical Layer

The underlying hardware directly impacts PostgreSQL's performance.

a. CPU

Complex queries involving aggregations, joins, sorting, and hashing are CPU-intensive.

Real-time Example: A complex analytical query calculating average sales per product category over a large dataset.

```
SELECT
  pc.category_name,
  AVG(s.amount) AS average_sale_amount
FROM
  sales_partitioned s
```

```
JOIN
  products p ON s.product_id = p.product_id
JOIN
  product_categories pc ON p.category_id = pc.category_id
WHERE
  s.sale_date BETWEEN '2024-01-01' AND '2024-12-31'
GROUP BY
  pc.category_name
ORDER BY
  average_sale_amount DESC;
```

How to Deal with This Issue:

- **Query Optimization:** Before considering a CPU upgrade, ensure your queries are optimized. Use EXPLAIN ANALYZE to identify bottlenecks. Can you reduce the amount of data processed? Can you use more efficient joins or aggregations?
- **CPU Upgrade:** If query optimization isn't enough and CPU utilization remains consistently high during peak loads, a CPU upgrade (more cores or faster clock speed) might be necessary.
- **Parallel Queries:** PostgreSQL can utilize multiple CPU cores for parallel query execution for certain operations (e.g., large sequential scans, joins, aggregations). Adjust max_worker_processes and max_parallel_workers_per_gather in postgresql.conf.

PostgreSQL Script to Check CPU Usage (indirectly through query execution time):

```
-- To understand where CPU time is spent, analyze query plans.
EXPLAIN (ANALYZE, BUFFERS)
SELECT ... -- Your CPU-intensive query here;
```

Look for CPU Time in the EXPLAIN ANALYZE output.

b. RAM

RAM is critical for caching data and executing queries. More RAM means more data can be held in memory, reducing expensive disk I/O.

Real-time Example: Running a large query that requires sorting a huge result set. If there isn't enough memory, PostgreSQL will "spill" to disk, leading to slower performance.

How to Deal with This Issue:

- **shared_buffers:** This is the most important memory parameter. It defines the amount of memory PostgreSQL uses for caching data blocks. A common recommendation is to set it to 25% of your total system RAM, but it can be higher for dedicated database servers (up to 40-50%).
    - **To check current value:** SHOW shared_buffers;
    - **To set in postgresql.conf:** shared_buffers = 2GB (or an appropriate value)
- **work_mem:** This parameter specifies the amount of memory to be used by internal sort operations and hash tables before writing to temporary disk files. If you frequently see "external sort" or "hash aggregate" operations spilling to disk in EXPLAIN ANALYZE, increasing work_mem can help.
    - **To check current value:** SHOW work_mem;
    - **To set in postgresql.conf:** work_mem = 64MB (or higher for large sorts)
    - **Important:** work_mem is allocated *per operation*, so a query with multiple sort operations will use work_mem multiple times. Be cautious not to set it too high, as it can lead to OOM errors if many concurrent queries require large work_mem allocations.
- **maintenance_work_mem:** Used for maintenance operations like VACUUM, CREATE INDEX, and ALTER TABLE ADD FOREIGN KEY. Increasing this can speed up these operations.
    - **To check current value:** SHOW maintenance_work_mem;
    - **To set in postgresql.conf:** maintenance_work_mem = 512MB

PostgreSQL Script to Identify Queries Spilling to Disk: This requires monitoring pg_stat_statements if enabled, or analyzing EXPLAIN ANALYZE output manually for "disk" in the plan.

-- You need to have pg_stat_statements enabled in postgresql.conf

-- (shared_preload_libraries = 'pg_stat_statements')

-- Then run:

SELECT query, total_time, calls, temp_blks_read, temp_blks_written

```
FROM pg_stat_statements

ORDER BY temp_blks_written DESC

LIMIT 10;
```

Look for queries with high temp_blks_written, indicating they are writing temporary data to disk.

c. Disk I/O

Disk I/O is often the slowest component in a database system. Fast disks are essential.

Real-time Example: A database with a high volume of writes (e.g., an OLTP system) or frequently accessed large tables.

How to Deal with This Issue:

- **SSDs (Solid State Drives):** Always prefer SSDs over traditional HDDs for database storage due to their significantly faster random read/write speeds.
- **RAID Configurations:** Use appropriate RAID levels (e.g., RAID 10) for both performance and redundancy.
- **Separate Tablespaces:** For very high I/O workloads, consider distributing tablespaces across different physical disk drives. This allows simultaneous I/O operations on different parts of your data.

PostgreSQL Script to Check I/O Activity:

```
SELECT
  relname AS table_name,
  heap_blks_read,
  heap_blks_hit,
  toast_blks_read,
  toast_blks_hit,
  idx_blks_read,
  idx_blks_hit
FROM
  pg_statio_user_tables
ORDER BY
```

heap_blks_read DESC;

heap_blks_read and idx_blks_read indicate how many data blocks were read from disk. High numbers here suggest I/O is a bottleneck. _hit values indicate blocks served from cache (memory), which is good.

d. Network

Network latency and bandwidth can become a bottleneck, especially in distributed systems or when clients are geographically distant.

Real-time Example: An application server in one data center connecting to a PostgreSQL database in another, or heavy replication traffic between master and standby servers.

How to Deal with This Issue:

- **High-Speed Network Cards:** Ensure servers have adequate network interface cards (NICs) capable of high bandwidth (e.g., 10Gbps, 25Gbps, or higher).
- **Low Latency Connectivity:** Minimize network hops and ensure low latency between your application servers and database servers. Co-locating them in the same data center is ideal.
- **Network Hardware:** Invest in high-performance switches and routers.
- **Compression:** For certain types of traffic (e.g., replication), consider using network compression if bandwidth is severely limited, though this adds CPU overhead.
- **Connection Pooling:** Using a connection pooler like PgBouncer can reduce the overhead of establishing new connections and improve overall network efficiency by reusing existing connections.

PostgreSQL Script to Monitor Network Activity (indirectly): While PostgreSQL doesn't have direct network monitoring views, you can observe connection activity and query latency.

```
SELECT
  client_addr,
  usename,
  state,
  backend_start,
  query_start,
```

```
  state_change,
  waiting,
  query
FROM
  pg_stat_activity
WHERE
  state = 'active'
ORDER BY
  query_start;
```

Long query_start to state_change times might indicate network latency if the database itself is not busy.

3. Operating System Optimization: Bridging Software and Hardware

The OS acts as the intermediary between PostgreSQL and the hardware. Proper OS configuration can significantly impact performance.

Real-time Example: A long-running idle connection between an application and the database getting unexpectedly terminated.

How to Deal with This Issue:

a. TCP Keepalives: PostgreSQL relies on the OS to maintain TCP connections. If a connection is idle for too long, the OS might terminate it, leading to "Connection reset by peer" or "server closed the connection unexpectedly" errors. TCP keepalives prevent this by periodically sending small packets to keep the connection alive.

How to Deal with This Issue (Linux): You can configure these parameters in /etc/sysctl.conf and apply them with sudo sysctl -p.

- net.ipv4.tcp_keepalive_time: The number of seconds a connection must be idle before TCP begins sending keepalive probes. Default is usually 7200 seconds (2 hours).
    - **Recommendation:** Reduce this to a more practical value, e.g., 300 (5 minutes) or 600 (10 minutes), especially if you have firewalls or load balancers with aggressive idle timeouts.
- net.ipv4.tcp_keepalive_time = 300

- net.ipv4.tcp_keepalive_intvl: The number of seconds between successive keepalive probes. Default is usually 75 seconds.
    - **Recommendation:** Keep it low, e.g., 15-30 seconds.
- net.ipv4.tcp_keepalive_intvl = 15
- net.ipv4.tcp_keepalive_probes: The number of probes TCP should send before giving up and closing the connection. Default is usually 9.
    - **Recommendation:** Increase if needed, but the _time and _intvl are usually more important.
- net.ipv4.tcp_keepalive_probes = 5

How to Deal with This Issue (Windows): These settings are in the Windows Registry under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters.

- KeepAliveTime: The interval in milliseconds after which an idle connection starts sending keep-alive probes. Default is 7,200,000 milliseconds (2 hours).
    - **Recommendation:** Set to 300,000 (5 minutes).
- KeepAliveInterval: The interval in milliseconds between successive keep-alive retransmissions until a response is received. Default is 1,000 milliseconds (1 second).
    - **Recommendation:** Set to 15,000 (15 seconds).

b. File System and I/O Scheduler:

- **File System Choice:** For Linux, XFS or ext4 are generally good choices for PostgreSQL. Ensure they are mounted with appropriate options (e.g., noatime to avoid updating access times on every read).
- **I/O Scheduler:** For SSDs, noop or deadline schedulers are often recommended on Linux. For HDDs, cfq or deadline might be more suitable. You can check and set this via /sys/block/<disk_name>/queue/scheduler.

c. Swappiness: vm.swappiness controls how aggressively the Linux kernel swaps out inactive memory pages to disk. A high value (default 60) can lead to the kernel swapping out PostgreSQL's buffer cache, which is detrimental to performance.

- **Recommendation:** For a dedicated PostgreSQL server, set vm.swappiness to a low value, like 1 or 10.
- -- In /etc/sysctl.conf
- vm.swappiness = 10

d. Huge Pages (Linux): Huge pages can reduce TLB (Translation Lookaside Buffer) miss rates, especially for large memory allocations like shared_buffers, potentially improving performance.

PostgreSQL Script (for Checking Huge Pages - indirect): PostgreSQL logs will indicate if huge pages are being used. You can configure them in postgresql.conf using huge_pages = on (requires OS-level configuration).

-- Check if huge pages are enabled in PostgreSQL config

SHOW huge_pages;

Conclusion

PostgreSQL performance tuning is an ongoing process that requires a holistic approach. It's not a one-time fix but a continuous cycle of:

1. **Monitoring:** Use pg_stat_statements, pg_stat_activity, and OS-level tools (e.g., top, iostat, vmstat) to identify bottlenecks.
2. **Analysis:** Use EXPLAIN ANALYZE to understand query plans and resource consumption.
3. **Tuning:** Apply changes to database design, PostgreSQL configuration, hardware, or OS settings based on your analysis.
4. **Testing:** Thoroughly test changes in a staging environment before deploying to production.
5. **Iteration:** Performance needs evolve with your application and data, so regular review and tuning are crucial.

PostgreSQL's strength lies in its extensive configurability. Tuning its configuration parameters is a critical aspect of optimizing performance for your specific workload. These parameters, typically managed in the postgresql.conf file, dictate how PostgreSQL utilizes system resources, manages connections, handles data integrity, and plans queries.

Let's break down these configuration parameters in detail, along with real-time examples and PostgreSQL scripts for checking and dealing with them.

Database Configuration Parameters Tuning: In-Depth Analysis

1. Connection Configuration Parameters

max_connections: This parameter defines the maximum number of concurrent client connections that your PostgreSQL server will accept. Setting it too high without sufficient hardware resources can lead to severe performance degradation due to context switching overhead and memory exhaustion. Setting it too low can result in "too many connections" errors for your applications.

- **Formula for Calculation:** max_connections = max(4 * number of CPU cores, 100)
    - This formula serves as a *starting point*. It aims to prevent the CPU from being overloaded but doesn't account for application-level connection pooling or specific workload patterns (e.g., many idle connections vs. many active, CPU-intensive connections).
    - For a server with 8 CPU cores, the formula suggests max(4 * 8, 100) = max(32, 100) = 100.

Real-time Example: An e-commerce website experiences high traffic during a flash sale. Many users simultaneously try to access the database. If max_connections is too low, new user requests will be rejected, leading to "Service Unavailable" errors for the users.

PostgreSQL Script to Check and Monitor:

```
-- Check the current max_connections setting

SHOW max_connections;


-- Check current active connections

SELECT

  datname,

  usename,

  client_addr,

  state,

  backend_start,

  query_start,

  state_change,

  waiting,
```

query

FROM pg_stat_activity

WHERE state != 'idle';

-- Count current connections by state

SELECT state, COUNT(*) FROM pg_stat_activity GROUP BY state;

-- Count connections by application name (if set by clients)

SELECT application_name, COUNT(*) FROM pg_stat_activity GROUP BY application_name;

How to Deal with This Issue:

1. **Monitor pg_stat_activity:** Regularly observe the number of active connections. If you frequently see connections in active state approaching max_connections, it's a sign that your application is using many concurrent connections.
2. **Application-Level Connection Pooling:** This is the **most recommended solution**. Tools like PgBouncer (a PostgreSQL-specific connection pooler) or built-in connection pooling in application frameworks (e.g., HikariCP for Java, SQLAlchemy for Python) dramatically reduce the number of direct database connections. The application connects to the pooler, which maintains a smaller, fixed number of connections to the database.
    - **Benefit:** Reduces overhead of connection establishment, prevents database from being overwhelmed, and allows max_connections to be set to a lower, more manageable value.
3. **Increase max_connections:** Only as a last resort, or if you have a dedicated database server with ample RAM and CPU and genuinely high concurrent activity. If you do increase it, ensure you also scale up other resources (especially RAM) accordingly, as each connection consumes some memory.
    - **To set in postgresql.conf:**
    - max_connections = 200 # Example value for a server with good resources
    - **Requires Restart:** Changing max_connections requires a PostgreSQL server restart.

2. Memory Configuration Parameters

These parameters dictate how PostgreSQL uses system memory. The sum of memory allocated by all active sessions (including shared_buffers, temp_buffers *per session*, and work_mem *per operation*) should not exceed available RAM, or you risk swapping, OOM errors, or system instability.

a. shared_buffers: The amount of memory PostgreSQL uses for caching data blocks, indexes, and other frequently accessed information. This is global memory shared by all backend processes.

- **Rule of thumb:** 25% of total system RAM for a dedicated DB server. For systems also running other applications, adjust downwards. In modern systems with abundant RAM (e.g., 64GB+), you might go up to 40-50% if the database is highly read-intensive, but be careful not to starve the OS disk cache.

Real-time Example: A read-heavy analytics database where the same reports are generated frequently. A large shared_buffers value means more data can be cached, leading to faster report generation as data is served from memory rather than disk.

PostgreSQL Script to Check and Deal:

```
-- Check current shared_buffers setting

SHOW shared_buffers;


-- Check buffer hit ratio (indicates how effectively shared_buffers is used)

-- A high ratio (e.g., > 99%) is generally good.

SELECT

 sum(blks_hit) * 100 / (sum(blks_hit) + sum(blks_read)) AS hit_ratio

FROM pg_stat_database;
```

How to Deal with This Issue:

- **Increase shared_buffers:** If your buffer hit ratio is consistently low (e.g., below 90%) and you have available RAM, increasing shared_buffers is often the first and most impactful memory tuning step.
    - **To set in postgresql.conf:**
    - shared_buffers = 8GB # Example for a server with 32GB RAM
    - **Requires Restart:** Changes to shared_buffers require a server restart.

b. temp_buffers: Amount of memory used for temporary tables and temporary indexes created *per database session*. This memory is local to each session and is released when the session ends.

- **Default:** 8MB.

Real-time Example: A complex query involving a large WITH clause that materializes a temporary result set, or a query that creates temporary indexes during execution.

PostgreSQL Script to Check:

-- Check current temp_buffers setting

SHOW temp_buffers;

How to Deal with This Issue:

- **Increase temp_buffers:** Only if EXPLAIN ANALYZE shows that temporary tables or indexes are spilling to disk (Buffers: temp read/written values). This is less common to tune than work_mem.
    - **To set in postgresql.conf:**
    - temp_buffers = 32MB # Adjust based on specific query needs
    - **No Restart:** Can be set per session or in postgresql.conf. Changing in postgresql.conf applies to new sessions.

c. wal_buffers: Amount of shared memory used to buffer Write-Ahead Log (WAL) data before it's written to disk.

- **Default:** -1 (meaning 3% of shared_buffers, capped at 16MB).
- **Range:** Minimum 64KB, Maximum 1 WAL segment (typically 16MB).

Real-time Example: A high-transaction OLTP system with frequent INSERT, UPDATE, DELETE operations. Larger wal_buffers can batch more WAL records before flushing to disk, reducing disk I/O.

PostgreSQL Script to Check:

-- Check current wal_buffers setting

SHOW wal_buffers;

How to Deal with This Issue:

- **Increase wal_buffers:** For write-intensive workloads, increasing wal_buffers to its maximum of 16MB can reduce the frequency of WAL flushes, improving write performance.

- **To set in postgresql.conf:**
- wal_buffers = 16MB
- **Requires Restart:** Changes to wal_buffers require a server restart.

d. work_mem: The maximum amount of memory used by a query operation (like sorting, hashing, or joining) before it starts spilling data to temporary disk files. This is allocated *per operation* for *each backend process*.

- **Default:** 4MB.
- **Crucial Note:** If a single complex query has multiple sort or hash operations, it can use work_mem multiple times concurrently. So, max_connections * average_active_sessions * work_mem can be a rough estimate of potential peak memory usage *just for query operations*, which can quickly consume all RAM if work_mem is set too high.

Real-time Example: A query performing a large ORDER BY clause or a complex JOIN operation on tables that don't fit into memory.

PostgreSQL Script to Check and Deal:

```
-- Check current work_mem setting

SHOW work_mem;


-- Analyze a query to see if it's spilling to disk (using temp files)

EXPLAIN (ANALYZE, BUFFERS)

SELECT

  c.customer_name,

  SUM(o.total_amount) AS total_orders

FROM

  customers c

JOIN

  orders o ON c.customer_id = o.customer_id

WHERE

  o.order_date >= '2025-01-01'

GROUP BY
```

```
    c.customer_name
ORDER BY
    total_orders DESC
LIMIT 100;
```

In the EXPLAIN ANALYZE output, look for lines indicating "Sort Method: external merge Disk" or "HashAggregate Disk". Also, check Buffers: temp read/written. These indicate work_mem was insufficient.

How to Deal with This Issue:

- **Increase work_mem:** If you observe frequent disk spills in EXPLAIN ANALYZE for common, performance-critical queries.
    - **To set in postgresql.conf:**
    - work_mem = 64MB # A common starting point for larger queries
    - **No Restart:** Can be set per session or in postgresql.conf. Be cautious; a too-high global work_mem can lead to OOM errors with many concurrent complex queries. Consider setting it at the user or database level if only specific workloads need more.

e. maintenance_work_mem: Amount of memory dedicated to maintenance operations such as VACUUM, CREATE INDEX, ALTER TABLE ADD FOREIGN KEY, and COPY (for data loading).

- **Default:** 256MB.
- **Recommendation:** 1GB or higher for large databases, up to 10-20% of RAM. This memory is only used by one maintenance process at a time.

Real-time Example: Rebuilding a large index on a multi-terabyte table. With sufficient maintenance_work_mem, the index build can largely happen in memory, significantly reducing I/O and speeding up the operation.

PostgreSQL Script to Check:

```
-- Check current maintenance_work_mem setting
SHOW maintenance_work_mem;
```

How to Deal with This Issue:

- **Increase maintenance_work_mem:** If you have large tables and indexes and find VACUUM or CREATE INDEX operations taking a very long time.

- **To set in postgresql.conf:**
- maintenance_work_mem = 1GB # Or more, depending on your largest tables
- **No Restart:** Can be set per session or in postgresql.conf.

3. Write-Ahead Log (WAL) Configuration Parameters

WAL ensures data durability and enables point-in-time recovery. Tuning WAL parameters involves balancing performance (reducing write overhead) with recovery time objective (RTO).

a. fsync: Controls whether PostgreSQL forces writes to disk (syncs) for WAL records and data files.

- **Default:** on.
- **Impact:** If fsync is off, data can be lost in case of an OS crash or power failure. Performance is faster, but data integrity is compromised.

Real-time Example: A development or testing environment where data loss is acceptable, and you prioritize raw speed for large data loads.

PostgreSQL Script to Check:

-- Check current fsync setting

SHOW fsync;

How to Deal with This Issue:

- **DO NOT DISABLE IN PRODUCTION:** Leave fsync = on for any production environment where data integrity is paramount.
- **Consider for Testing/Dev:** You might temporarily set fsync = off for bulk loading data into a development database that can be easily recreated.
- **To set in postgresql.conf:**
- fsync = off # ONLY FOR NON-PRODUCTION ENVIRONMENTS
- **Requires Restart:** Changes to fsync require a server restart.

b. commit_delay: Sets the delay in microseconds after a commit before flushing WAL to disk. This allows other transactions to group their commits, leading to fewer, larger flushes.

- **Default:** 0 (no delay).

- **Range:** 0 to 100000 (100 milliseconds).
- **Caution:** Setting too high increases the risk of data loss in a crash, as more un-flushed transactions could be lost.

Real-time Example: An OLTP application with many small, frequent transactions. Batching commits can reduce disk I/O.

PostgreSQL Script to Check:

-- Check current commit_delay setting

SHOW commit_delay;

How to Deal with This Issue:

- **Experiment for Write-Heavy Workloads:** If your storage system is slow or you have many concurrent small transactions, a small commit_delay (e.g., 1000-5000 microseconds, i.e., 1-5ms) can sometimes improve performance by batching WAL flushes.
    - **To set in postgresql.conf:**
    - commit_delay = 1000 # 1 millisecond
    - **No Restart:** Can be set per session or in postgresql.conf.

c. checkpoint_timeout: The maximum time interval between WAL checkpoints. Checkpoints ensure that all dirty data pages in shared_buffers are written to disk, minimizing recovery time after a crash.

- **Default:** 5 minutes.
- **Range:** 30 seconds to 1 day.

Real-time Example: A database that experiences high write activity. Frequent checkpoints (low checkpoint_timeout) can lead to I/O spikes during checkpointing, while infrequent checkpoints (high checkpoint_timeout) can lead to longer recovery times.

PostgreSQL Script to Check:

-- Check current checkpoint_timeout setting

SHOW checkpoint_timeout;

How to Deal with This Issue:

- **Balance Performance and Recovery:**

- **Increase for Performance (with caution):** If you see frequent "checkpoint starting" messages in your logs and/or high I/O spikes during checkpoints, increasing checkpoint_timeout (e.g., to 15-30 minutes) can smooth out disk writes. However, this increases recovery time after a crash.
- **To set in postgresql.conf:**
- checkpoint_timeout = 15min
- **Requires Restart:** Changes to checkpoint_timeout require a server restart.

d. checkpoint_completion_target: Dictates how much of the checkpoint_timeout interval should be used to complete the dirty page write-out during a checkpoint. This helps spread the I/O load more evenly.

- **Default:** 0.9 (90%).

Real-time Example: With checkpoint_completion_target = 0.9, PostgreSQL aims to finish writing all dirty buffers within 90% of the checkpoint_timeout period. This leaves 10% of the period for any unexpected delays or bursts of activity.

PostgreSQL Script to Check:

-- Check current checkpoint_completion_target setting

SHOW checkpoint_completion_target;

How to Deal with This Issue:

- **Generally Leave Default:** The default of 0.9 is usually optimal. It provides a good balance. Only advanced users with very specific I/O patterns might consider adjusting this, but it's rarely necessary.

e. min_wal_size and max_wal_size: These parameters, introduced in PostgreSQL 9.5+, control the minimum and maximum amount of WAL disk space. They effectively replace checkpoint_segments.

- **min_wal_size**: PostgreSQL will delete WAL segments older than max_wal_size *unless* doing so would reduce the total WAL size below min_wal_size. This ensures a certain amount of WAL is always retained for recovery or standby servers.

  - **Default:** 80MB.

- **max_wal_size**: The maximum size of the WAL directory. When the total size of WAL segments exceeds this, a checkpoint is forced. This parameter works in conjunction with checkpoint_timeout.
  - **Default:** 1GB.
  - **Formula (approx. equivalent to old checkpoint_segments):** max_wal_size = (3 * checkpoint_segments_old_value) * 16MB

Real-time Example: A very busy database generating a lot of WAL. If max_wal_size is too low, it will trigger frequent checkpoints, causing I/O spikes. If min_wal_size is too low, old WAL files might be removed prematurely, causing issues for standby servers that are lagging.

PostgreSQL Script to Check:

```
-- Check current WAL size settings

SHOW min_wal_size;

SHOW max_wal_size;


-- Check actual WAL directory size (OS level)

-- For Linux: du -sh $PGDATA/pg_wal/
```

How to Deal with This Issue:

- **Increase max_wal_size:** If your database is write-heavy and you observe frequent checkpoints due to WAL volume rather than checkpoint_timeout. This allows more WAL to accumulate before a checkpoint is forced, spreading I/O over a longer period.
  - **To set in postgresql.conf:**
  - max_wal_size = 4GB # Example for a busy database
- **Increase min_wal_size:** Useful for replication scenarios where standby servers might occasionally lag. A larger min_wal_size ensures more WAL is retained on the primary, reducing the chance of a standby falling too far behind and requiring a full base backup.
  - **To set in postgresql.conf:**
  - min_wal_size = 1GB

- **Requires Restart:** Changes to min_wal_size and max_wal_size require a server restart.

4. Query Cost Configuration Parameters

These parameters influence the PostgreSQL query planner's choices by assigning "costs" to different operations. By adjusting them, you can guide the planner towards strategies that perform better on your specific hardware and workload.

a. random_page_cost: The planner's estimated cost of fetching a disk page that is not sequentially next to the previously fetched page. A lower value indicates faster random access (e.g., from SSDs), making index scans more attractive.

- **Default:** 4.0.
- **Context:** Sequential page access is always assumed to cost 1.0. So, random_page_cost = 4.0 means a random disk read is 4 times more expensive than a sequential one.

Real-time Example: Your database is running on fast SSDs. The default random_page_cost might make the planner prefer sequential scans even when an index scan would be faster.

PostgreSQL Script to Check and Deal:

```
-- Check current random_page_cost setting

SHOW random_page_cost;


-- Compare query plans with different random_page_cost values

-- First, with default:

EXPLAIN (ANALYZE)

SELECT * FROM large_table WHERE indexed_column = 'value';


-- Then, temporarily set a lower value and re-run (in a new session or with SET):

SET random_page_cost = 1.1; -- For fast SSDs, close to sequential

EXPLAIN (ANALYZE)

SELECT * FROM large_table WHERE indexed_column = 'value';

RESET random_page_cost; -- Reset after testing
```

Look for changes in the EXPLAIN plan, particularly if it switches from a "Seq Scan" to an "Index Scan" and if the execution time improves.

How to Deal with This Issue:

- **Reduce for SSDs:** If your data resides on fast SSDs, reducing random_page_cost (e.g., to 1.1 - 2.0) can encourage the planner to use indexes more often, as random access is relatively cheaper.
- **Increase for HDDs (Rare):** If on very slow HDDs, you might increase it, but generally, HDDs struggle with random access anyway, so this is less common.
    - **To set in postgresql.conf:**
    - random_page_cost = 1.5 # For SSDs
    - **No Restart:** Can be set per session or in postgresql.conf.

b. effective_cache_size: An estimate of the total amount of disk cache (including the OS file system cache and shared_buffers) available to PostgreSQL. This *does not* allocate memory; it merely informs the query planner about how much data it *expects* to find in memory. A larger value encourages the planner to use indexes more, as it assumes more data will be cached and random access will be cheap.

- **Default:** 4GB.

Real-time Example: A server with 64GB of RAM, where a large portion is used by the OS disk cache and PostgreSQL's shared_buffers. The default effective_cache_size of 4GB would significantly underestimate the available cache, potentially leading the planner to prefer sequential scans even when indexes would be beneficial.

PostgreSQL Script to Check:

-- Check current effective_cache_size setting

SHOW effective_cache_size;

How to Deal with This Issue:

- **Estimate Real Cache Size:** Set this parameter to a realistic estimate of your total system cache, typically shared_buffers + what you expect the OS to cache (e.g., 50-75% of remaining RAM). This is a crucial parameter for guiding the planner.
    - **To set in postgresql.conf:**
    - effective_cache_size = 32GB # For a system with 64GB RAM and 8GB shared_buffers

- **No Restart:** Can be set per session or in postgresql.conf.

General Tuning Best Practices:

- **Start Small, Test Iteratively:** Don't change too many parameters at once. Change one or a few related parameters, then monitor performance (pg_stat_activity, pg_stat_statements, OS metrics like CPU, I/O, memory) and analyze query plans (EXPLAIN ANALYZE).
- **Benchmark Your Workload:** Use tools like pgbench or actual application load testing to simulate your typical workload and measure the impact of changes.
- **Monitor, Monitor, Monitor:** Performance tuning is impossible without good monitoring. Set up alerts for high CPU, I/O, memory usage, and slow queries.
- **Document Changes:** Keep a log of all configuration changes, why they were made, and their observed effects.
- **Consult Experts:** For complex or persistent performance issues, consider consulting a PostgreSQL expert.

Beyond hardware and fundamental database configuration, PostgreSQL offers powerful features for logging, managing data bloat (Vacuum), and analyzing query execution. Leveraging these capabilities is crucial for ongoing performance monitoring, troubleshooting, and proactive optimization.

Let's explore these aspects in detail with real-time examples and practical PostgreSQL scripts.

Logging: The Database's Storyteller

PostgreSQL's logging mechanisms provide invaluable insights into database activity, errors, and performance bottlenecks. Proper log configuration is essential for effective maintenance and troubleshooting.

a. logging_collector: This parameter enables PostgreSQL to redirect log messages from standard error to a dedicated log file. This is highly recommended for production environments as it ensures all messages are captured and can be easily managed and rotated.

- **Default**: off (meaning logs go to stderr, which typically means they go to the terminal where postgres was started or to syslog if configured).

Real-time Example: You notice intermittent "connection reset by peer" errors from your application, but they don't appear in your system's syslog. If logging_collector is on, these errors will be reliably captured in PostgreSQL's dedicated log files.

PostgreSQL Script to Check and Deal:

```
-- Check current logging_collector setting

SHOW logging_collector;


-- Check the log directory (where logs are stored if collector is on)

SHOW log_directory;


-- Check the log filename pattern (for log rotation)

SHOW log_filename;
```

How to Deal with This Issue:

- **Enable in Production:** Always set logging_collector = on in production.
    - **To set in postgresql.conf:**
    - logging_collector = on
    - log_directory = 'log' # Relative to PGDATA, or an absolute path
    - log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log' # Daily log files
    - log_rotation_age = 1d # Rotate daily
    - log_rotation_size = 0 # No size-based rotation (unless needed)
    - **Requires Restart:** Changes to logging_collector require a PostgreSQL server restart.
- **Configure Log Rotation:** Use log_rotation_age and log_rotation_size to prevent log files from growing too large and consuming excessive disk space.

b. log_statement: Controls which SQL statement types are logged. This is powerful for auditing and performance analysis.

- **Options:**
    - none: (Default) No SQL statements are logged.

- ddl: Logs all Data Definition Language (DDL) statements (e.g., CREATE TABLE, ALTER INDEX, DROP DATABASE).
- mod: Logs all DDL and Data Manipulation Language (DML) statements that modify data (e.g., INSERT, UPDATE, DELETE, TRUNCATE).
- all: Logs all statements, including SELECT queries.

Real-time Example: You're tracking down an intermittent performance issue. Setting log_statement = all for a short period can help identify specific slow queries or unexpected query patterns being executed by your application.

PostgreSQL Script to Check and Deal:

-- Check current log_statement setting

SHOW log_statement;

How to Deal with This Issue:

- **Production Recommendations:**
  - log_statement = ddl: Generally safe for production; helps track schema changes.
  - log_statement = mod: Useful for auditing changes, but can generate significant log volume on write-heavy systems.
  - log_statement = all: **Use with extreme caution in production.** This generates massive log files and has a performance overhead. Only enable for short, targeted troubleshooting sessions.
- **To set in postgresql.conf:**
- log_statement = 'ddl'
- **No Restart:** Can be set per session or in postgresql.conf.

c. log_min_error_statement: Sets the minimum error severity level at which the *entire SQL query* that caused the error (or a higher severity message) is logged.

- **Values (ordered by severity):** DEBUG5, DEBUG4, DEBUG3, DEBUG2, DEBUG1, INFO, NOTICE, WARNING, ERROR, LOG, FATAL, PANIC.
- **Default:** ERROR.

Real-time Example: You want to see not just the "division by zero" error message, but also the exact SQL query that caused it, for debugging purposes.

PostgreSQL Script to Check:

-- Check current log_min_error_statement setting

SHOW log_min_error_statement;

How to Deal with This Issue:

- **Troubleshooting:** Temporarily set this to WARNING or NOTICE during debugging sessions to capture more context around potential issues. For regular production, ERROR is usually sufficient to avoid excessive log volume.
    - **To set in postgresql.conf:**
    - log_min_error_statement = 'WARNING'
    - **No Restart:** Can be set per session or in postgresql.conf.

d. log_line_prefix: A printf-style string added to the beginning of each log line, providing context like timestamp, process ID, session ID, etc.

- **Useful Escape Sequences:**
    - %m: Timestamp
    - %p: Process ID
    - %l: Log line number
    - %u: User name
    - %d: Database name
    - %r: Remote host and port
    - %q: No-quotes log_line_prefix (useful when log_statement is all)
    - %L: Session line number
    - %s: Session start time
    - %i: Command tag (e.g., SELECT, INSERT)
    - %c: Session ID (unique across all sessions)

Real-time Example: You're analyzing logs to understand concurrent activity and want to easily correlate log entries from the same session or process.

PostgreSQL Script to Check:

-- Check current log_line_prefix setting

SHOW log_line_prefix;

How to Deal with This Issue:

- **Recommended Production Prefix:** A good prefix includes critical identifiers for debugging.
    - **To set in postgresql.conf:**
    - log_line_prefix = '%m %q%u@%d %r [%p] %l ' # Example
    - # This will produce logs like:
    - # 2025-08-03 20:24:01.123 AEST myuser@mydb [12345] 123456 LOG: statement: SELECT 1;
    - **Requires Restart:** Changes to log_line_prefix require a server restart.

e. log_lock_waits: Logs messages when a session waits longer than deadlock_timeout for a lock. Essential for diagnosing performance issues caused by contention.

- **Default:** off.
- **deadlock_timeout Default:** 1 second.

Real-time Example: Your application occasionally "hangs" on certain operations. Enabling log_lock_waits can reveal if these hangs are due to transactions waiting on locks held by other sessions.

PostgreSQL Script to Check:

-- Check current log_lock_waits setting

SHOW log_lock_waits;

SHOW deadlock_timeout;

How to Deal with This Issue:

- **Enable in Production:** Set log_lock_waits = on to get insights into locking contention.
    - **To set in postgresql.conf:**
    - log_lock_waits = on
    - **Requires Restart:** Changes to log_lock_waits require a server restart.
- **Adjust deadlock_timeout:** If you have short transactions, you might reduce deadlock_timeout (e.g., to 500ms) to detect lock waits sooner. However, be careful not to make it too low, as it can cause false positives for short, normal lock acquisitions.

f. log_checkpoints: Logs details about checkpoints and restart points, including buffer write statistics and time taken. Crucial for understanding I/O patterns and optimizing WAL settings.

- **Default:** off.

Real-time Example: You notice intermittent I/O spikes. Enabling log_checkpoints can help you correlate these spikes with checkpoint activity, providing data to tune checkpoint_timeout and max_wal_size.

PostgreSQL Script to Check:

-- Check current log_checkpoints setting

SHOW log_checkpoints;

How to Deal with This Issue:

- **Enable in Production:** Set log_checkpoints = on to gain valuable insights into checkpoint performance.
    - **To set in postgresql.conf:**
    - log_checkpoints = on
    - **Requires Restart:** Changes to log_checkpoints require a server restart.

Vacuum: Reclaiming Space and Refreshing Stats

PostgreSQL uses a Multi-Version Concurrency Control (MVCC) architecture. When data is updated or deleted, the old versions (tuples) are not immediately removed. This leads to "dead tuples" that occupy disk space and can degrade query performance. VACUUM is the process of cleaning up these dead tuples.

a. The VACUUM Command:

- **Purpose:** Reclaims space occupied by dead tuples. The space is then made available for reuse within the same table. It does *not* return space to the operating system.
- **Concurrency:** Can be run concurrently with SELECT, INSERT, UPDATE, and DELETE operations.

Real-time Example: A logs table that receives millions of INSERTs and DELETEs daily. Over time, the table file size on disk grows, and queries become slower as they have to scan through dead tuples.

PostgreSQL Script for VACUUM:

```
-- Vacuum a specific table (recommended for targeted cleanup)

VACUUM public.my_large_table;


-- Vacuum all tables in the current database

VACUUM;
```

How to Deal with This Issue:

- **Regular Execution (Autovacuum):** Rely primarily on autovacuum for routine maintenance.
- **Ad-Hoc for Specific Tables:** Manually run VACUUM on tables known to have high update/delete activity or after a large DELETE operation.

b. VACUUM ANALYZE: Combines the VACUUM process with ANALYZE. After cleaning up dead tuples, it updates the statistics for the table, which the query planner uses to generate efficient execution plans.

Real-time Example: After a major data import or bulk update, the table's statistics might be outdated. Running VACUUM ANALYZE ensures that future queries on that table are planned using fresh, accurate statistics.

PostgreSQL Script for VACUUM ANALYZE:

```
-- Vacuum and analyze a specific table

VACUUM ANALYZE public.my_order_history;


-- Vacuum and analyze all tables in the current database

VACUUM ANALYZE;
```

How to Deal with This Issue:

- autovacuum typically handles ANALYZE automatically, but manual VACUUM ANALYZE is useful after significant data changes that might not immediately trigger autovacuum.

c. VACUUM FULL: Rewrites the entire table to a new disk file, reclaiming all freed space and returning it to the operating system. It also compacts the table, potentially making it smaller.

- **Caution:** VACUUM FULL requires an ACCESS EXCLUSIVE lock on the table. This means *no other operations (reads or writes)* can occur on the table during the VACUUM FULL process, leading to significant downtime for that table.
- **Use Cases:** Only for tables that have shrunk significantly after major deletions and where the freed space needs to be returned to the OS, and downtime is acceptable.

Real-time Example: A table was used for temporary data and then had 90% of its rows deleted. VACUUM alone won't return the disk space. VACUUM FULL would reclaim the space.

PostgreSQL Script for VACUUM FULL:

```
-- Vacuum FULL a specific table (BE CAREFUL, it locks the table!)

VACUUM FULL public.my_temp_table;
```

How to Deal with This Issue:

- **Avoid in Production (Generally):** Due to the exclusive lock, VACUUM FULL is rarely used in high-availability production environments.
- **Alternative for Online Compaction:** Consider tools like pg_repack or using logical replication for zero-downtime table rewrites if table bloat is a severe issue requiring compaction.

d. Autovacuum: The most critical component for maintaining PostgreSQL health. It's a set of background processes that automatically run VACUUM and ANALYZE operations based on predefined thresholds.

- **Configuration:** Many parameters in postgresql.conf control autovacuum's behavior (e.g., autovacuum, autovacuum_max_workers, autovacuum_vacuum_scale_factor, autovacuum_vacuum_threshold).

Real-time Example: A busy orders table has frequent INSERTs and UPDATEs. Autovacuum continuously cleans up dead tuples and updates statistics in the background, preventing bloat and ensuring the query planner always has up-to-date information.

PostgreSQL Script to Check Autovacuum Status and Configuration:

```
-- Check if autovacuum is enabled

SHOW autovacuum;
```

```
-- Check autovacuum workers
SHOW autovacuum_max_workers;


-- Check table-specific autovacuum settings (e.g., for 'my_table')
SELECT
  relname,
  autovacuum_enabled,
  autovacuum_vacuum_threshold,
  autovacuum_vacuum_scale_factor,
  autovacuum_analyze_threshold,
  autovacuum_analyze_scale_factor
FROM pg_class c
LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
WHERE n.nspname = 'public' AND c.relname = 'my_table';


-- Check autovacuum activity
SELECT
  datname,
  relname,
  phase,
  backend_type,
  state,
  query
FROM pg_stat_activity
WHERE backend_type = 'autovacuum worker';


-- Check last vacuum/analyze times and counts for tables
```

```
SELECT
  relname,
  last_vacuum,
  last_autovacuum,
  vacuum_count,
  autovacuum_count,
  last_analyze,
  last_autoanalyze,
  analyze_count,
  autoanalyze_count
FROM pg_stat_user_tables
ORDER BY last_autovacuum ASC NULLS FIRST;
```

How to Deal with This Issue:

- **Enable and Tune:** Ensure autovacuum = on in postgresql.conf.
- **Adjust Thresholds:** For very busy tables, you might need to lower autovacuum_vacuum_scale_factor or increase autovacuum_vacuum_threshold (or autovacuum_analyze_scale_factor/autovacuum_analyze_threshold) at the table level (using ALTER TABLE SET (autovacuum_vacuum_scale_factor = 0.05);) to make autovacuum run more frequently.
- **Increase Workers:** If many tables are accumulating bloat, increase autovacuum_max_workers.
- **Monitor:** Regularly check pg_stat_user_tables to see if autovacuum is running frequently enough and if tables are getting vacuumed and analyzed.

Query Performance Tuning: Unveiling the Execution Plan

Understanding how PostgreSQL executes your queries is fundamental to optimizing them.

a. ANALYZE: Collects statistics about the contents of tables and indexes, which the query planner then uses to make informed decisions about the most efficient way to execute queries.

Real-time Example: After loading a large amount of new data into a table, the distribution of values in columns might change significantly. If ANALYZE isn't run, the planner will rely on old, inaccurate statistics, potentially choosing inefficient query plans.

PostgreSQL Script for ANALYZE:

-- Analyze a specific table

ANALYZE public.my_sales_data;

-- Analyze all tables in the current database

ANALYZE;

How to Deal with This Issue:

- **Rely on Autovacuum (Mostly):** autovacuum automatically runs ANALYZE when certain thresholds are met.
- **Manual ANALYZE for Bulk Changes:** After a large data import, TRUNCATE and INSERT, or a major UPDATE that affects many rows, a manual ANALYZE can provide immediate, updated statistics to the planner, improving subsequent query performance.

b. EXPLAIN: Shows the execution plan that the PostgreSQL planner generates for a given SQL query *without actually running the query*. It's your primary tool for understanding how a query will be executed.

- **EXPLAIN ANALYZE**: Executes the query and shows the actual run-time statistics in addition to the plan. This is invaluable for identifying bottlenecks (e.g., which steps took the longest, how many rows were processed, disk I/O vs. CPU time).

Real-time Example: A report query is consistently slow. Using EXPLAIN ANALYZE, you can see if it's performing a full table scan when an index could be used, or if a specific JOIN or SORT operation is taking an excessive amount of time.

PostgreSQL Script for EXPLAIN and EXPLAIN ANALYZE:

-- Show the planned execution plan (does not run the query)

EXPLAIN SELECT product_name, SUM(quantity) FROM order_items GROUP BY product_name;

-- Show the actual execution plan and runtime statistics (runs the query)

EXPLAIN ANALYZE SELECT product_name, SUM(quantity) FROM order_items GROUP BY product_name;

-- More detailed output (buffers and timings)

EXPLAIN (ANALYZE, BUFFERS, TIMING) SELECT customer_name FROM customers WHERE city = 'Sydney';

How to Deal with This Issue:

- **Interpret EXPLAIN ANALYZE Output:**
    - **Scan Types:** Look for "Seq Scan" on large tables when you expect an "Index Scan." This often indicates missing or unused indexes, or stale statistics.
    - **Costs/Times:** Compare estimated costs with actual run times. Discrepancies might point to inaccurate statistics or assumptions.
    - **Rows:** Check the number of rows processed at each step. If a step processes many more rows than expected, it might indicate a filtering issue.
    - **Memory Spills:** Look for "Sort Method: external merge Disk" or "HashAggregate Disk," which indicate work_mem was insufficient.
    - **Parallelism:** Observe if parallel workers are being used as expected for large queries.
- **Iterative Optimization:** Based on EXPLAIN ANALYZE output, you might:
    - Add or modify indexes.
    - Rewrite the query (e.g., using different JOIN types, subqueries vs. CTEs).
    - Adjust postgresql.conf parameters (e.g., work_mem, random_page_cost, effective_cache_size).
    - Run ANALYZE manually if statistics are suspected to be stale.

Indexing Performance: Precision Tools for Data Access

Indexes are essential for fast data retrieval, but they come with trade-offs.

a. Impact of Indexing:

- **Reads:** Dramatically speeds up SELECT queries that filter on indexed columns (e.g., WHERE, ORDER BY, JOIN conditions).

- **Writes:** Adds overhead to INSERT, UPDATE, and DELETE operations because the indexes themselves must also be updated.

Real-time Example: A users table with an index on email. A query SELECT * FROM users WHERE email = 'some@email.com' will be very fast. However, inserting new users will take slightly longer because the email index needs to be updated.

PostgreSQL Script to Check Indexes and Their Usage:

```
-- List all indexes on a table
\d my_table


-- Check index usage statistics (useful for finding unused indexes)
SELECT
  schemaname,
  relname AS table_name,
  indexrelname AS index_name,
  idx_scan,      -- Number of times this index has been scanned
  idx_tup_read,  -- Number of index entries returned by index scans
  idx_tup_fetch  -- Number of table rows fetched via this index
FROM
  pg_stat_user_indexes
ORDER BY
  idx_scan DESC;


-- Identify columns that are frequently filtered in queries (to consider for indexing)
-- This requires pg_stat_statements to be enabled
SELECT
  query,
  calls,
  total_time
```

```
FROM
  pg_stat_statements
WHERE
  query ILIKE '%WHERE%'
ORDER BY
  total_time DESC
LIMIT 10;
```

How to Deal with This Issue:

- **Identify Critical Queries:** Focus on indexing columns used in WHERE, JOIN ON, ORDER BY, and GROUP BY clauses of your most frequently executed and slowest queries.
- **Consider Data Access Patterns:**
    - **High Selectivity:** Indexes are most effective on columns with high cardinality (many unique values), like email or user_id, where the index can filter out most rows.
    - **Low Selectivity:** Indexes on columns with low cardinality (few unique values, e.g., gender, status) are often less effective because the planner might still prefer a sequential scan.
- **Indexing Algorithms (B-tree, Hash, GIN, GiST, BRIN):**
    - **B-tree:** Default and most common. Good for equality and range queries (=, <, >).
    - **Hash:** Only for equality (=) queries, less commonly used due to lack of crash safety and other limitations compared to B-tree.
    - **GIN (Generalized Inverted Index):** For columns containing multiple values per row (e.g., ARRAY, jsonb, tsvector for full-text search).
    - **GiST (Generalized Search Tree):** For complex data types and specialized queries (e.g., geometric data, full-text search, k-nearest neighbor).
    - **BRIN (Block Range Index):** For very large tables where data is naturally ordered (e.g., timestamp in log tables). Very small and efficient for range queries.
- **Avoid Over-indexing:**

- **Performance Overhead:** Each index adds to storage space and slows down INSERT/UPDATE/DELETE.
- **Planner Confusion:** Too many indexes can sometimes confuse the query planner, leading it to choose suboptimal plans.
- **Redundant Indexes:** Remove indexes that are never used (check pg_stat_user_indexes).

- **Composite Indexes:** For queries with multiple WHERE conditions (e.g., WHERE col1 = 'A' AND col2 = 'B'), a composite index on (col1, col2) can be more efficient than two separate indexes. The order of columns in a composite index matters.

🧰 **PostgreSQL Performance Tuning Tools & Diagnostic Questions**

Tool Name	Key Focus Area	Diagnostic Questions It Helps Answer
pg_stat_statements	Query profiling	🔍 Which queries are consuming the most time or resources? 🔍 Are there repetitive queries that could be cached or optimized?
auto_explain	Execution plan logging	🔍 What execution plans are being used for slow queries? 🔍 Are sequential scans happening where indexes should be used?
pgBadger	Log analysis & reporting	🔍 What are the top slow queries? 🔍 Are there spikes in connection time or lock contention?
pganalyze	Full-stack observability	🔍 Do I need more indexes? 🔍 Is my buffer cache hit ratio optimal? 🔍 Are autovacuum settings tuned correctly?
Prometheus + Grafana	OS + DB metrics visualization	🔍 Is CPU or disk I/O a bottleneck? 🔍 Is memory usage aligned with PostgreSQL config? 🔍 Are there spikes in connections or WAL activity?

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

Tool Name	Key Focus Area	Diagnostic Questions It Helps Answer
pgTune	Config recommendation engine	🔍 Are my memory and connection settings aligned with hardware? 🔍 What should my shared_buffers, work_mem, and effective_cache_size be?
pgbench	Load testing	🔍 How does PostgreSQL perform under simulated load? 🔍 Can my current setup handle peak traffic?
check_postgres	Health checks & alerts	🔍 Are there bloated tables or indexes? 🔍 Is replication lag increasing? 🔍 Are autovacuum stats healthy?
HypoPG	Hypothetical indexing	🔍 Would a new index improve query performance? 🔍 What is the impact of multi-column indexes before creating them?
pgmetrics	Snapshot-based metrics	🔍 What are the current stats across databases, tables, and indexes? 🔍 Are there anomalies in locks or cache usage?

🧠 **How These Tools Help You Answer:**

- **Resource Planning**: Prometheus + Grafana, pgTune
- **Query Optimization**: pg_stat_statements, auto_explain, pgBadger, pganalyze
- **Index Strategy**: HypoPG, pganalyze Index Advisor
- **Configuration Tuning**: pgTune, pganalyze Config Advisor
- **Maintenance Insight**: check_postgres, pgmetrics

Different database types and how their characteristics influence PostgreSQL performance tuning. 🚀

Web Application (Web) 🌐

A **web application database** typically serves a user-facing application, such as a blog, a small e-commerce site, or a social media platform. Its primary role is to handle many concurrent, short-lived requests, often involving simple data retrieval and updates.

Characteristics

- **Typically CPU-bound**: The bottleneck often lies in the CPU due to the overhead of managing numerous connections and executing many small queries.
- **DB much smaller than RAM**: The active dataset often fits comfortably within the available system RAM, allowing for extensive caching.
- **90% or more simple queries**: Most queries are straightforward SELECT statements, often with simple WHERE clauses, or small INSERT/UPDATE/DELETE operations.

Typical Workload & Performance Considerations

Web applications prioritize **low latency** for individual requests and **high concurrency** to serve many users simultaneously. The workload consists of many transactions, each touching a small amount of data. Performance is crucial for a smooth user experience.

PostgreSQL Tuning Focus

For web applications, the focus is on optimizing connection handling, effective caching, and efficient execution of simple queries.

- **max_connections**: Needs to be high enough to accommodate concurrent web requests, but balanced with available memory.

SHOW max_connections;

- **shared_buffers**: Crucial for caching frequently accessed data. Set this to a significant portion of available RAM (e.g., 25-40%).

SHOW shared_buffers;

- **work_mem**: Can often remain at its default or a slightly increased value, as queries are typically simple and don't require large sorts.

SHOW work_mem;

- **effective_cache_size**: Inform the planner about the large amount of available cache to encourage index usage.

SHOW effective_cache_size;

- **random_page_cost**: If using SSDs (highly recommended for web apps), reduce this to encourage index scans.

SHOW random_page_cost;

- **Connection Pooling**: Crucial for managing connections efficiently. While not a postgresql.conf parameter, using an external pooler like **PgBouncer** is highly recommended to reduce database overhead.

-- Example of a typical simple query

EXPLAIN ANALYZE SELECT username, email FROM users WHERE user_id = 123;

Example Scenario

A popular online news portal. Users frequently browse articles, leave comments, and log in. Each action translates to a few quick database queries. The database needs to handle thousands of such requests per second without slowing down.

Online Transaction Processing (OLTP) 📊

OLTP databases are designed for transactional workloads, emphasizing speed, accuracy, and data integrity for high volumes of short, atomic transactions. Think banking systems, inventory management, or order processing.

Characteristics

- **Typically CPU- or I/O-bound**: Can be CPU-bound due to complex logic or high concurrency, or I/O-bound due to the sheer volume of reads and writes.
- **DB slightly larger than RAM to 1TB**: The active dataset might not fit entirely in RAM, requiring efficient disk access.
- **20-40% small data write queries**: A significant portion of queries are INSERT, UPDATE, and DELETE operations, alongside read queries.
- **Some long transactions and complex read queries**: While mostly short, there might be occasional longer transactions or more complex analytical reads.

Typical Workload & Performance Considerations

OLTP systems demand **high throughput** (transactions per second), **low latency** for individual transactions, and **strong ACID properties** (Atomicity, Consistency, Isolation, Durability). Data consistency is paramount.

PostgreSQL Tuning Focus

Tuning for OLTP involves balancing read and write performance, optimizing WAL (Write-Ahead Log) activity, and managing contention.

- **shared_buffers**: Still important for caching, but perhaps not as high a percentage as web apps if the active dataset is much larger than RAM.

SHOW shared_buffers;

- **wal_buffers**: Increase to 16MB for write-heavy workloads to batch more WAL records before flushing to disk.

SHOW wal_buffers;

- **checkpoint_timeout & max_wal_size**: Adjust to smooth out I/O spikes during checkpoints, balancing recovery time with write performance.

SHOW checkpoint_timeout;

SHOW max_wal_size;

- **work_mem**: May need to be slightly increased if there are occasional complex queries with large sorts or hashes.

SHOW work_mem;

- **fsync**: **Must be on** for data integrity.

SHOW fsync;

- **commit_delay**: Can be experimented with (e.g., 1000-5000 microseconds) for very high commit rates to batch WAL flushes.

SHOW commit_delay;

- **Autovacuum Tuning**: Crucial to prevent bloat from high UPDATE/DELETE activity. Adjust autovacuum_vacuum_scale_factor and autovacuum_max_workers.

SHOW autovacuum_vacuum_scale_factor;

SHOW autovacuum_max_workers;

- **Indexing**: Critical for fast lookups in both read and write queries.

```
-- Example of an OLTP query (order creation)

EXPLAIN ANALYZE INSERT INTO orders (customer_id, order_date, total_amount)
VALUES (123, NOW(), 99.99) RETURNING order_id;
```

Example Scenario

A banking system processing thousands of customer transactions (deposits, withdrawals, transfers) every second. Each transaction must be recorded accurately and immediately, and the system needs to handle high concurrency without data loss or significant delays.

Data Warehouse (DW) 📦

A **Data Warehouse (DW)**, also known as **Decision Support** or **Business Intelligence (BI)**, is optimized for analytical queries over large datasets. It's used for reporting, trend analysis, and strategic decision-making, rather than day-to-day operations.

Characteristics

- **Typically I/O- or RAM-bound**: Performance is often limited by how quickly data can be read from disk or how much data can be held in memory for processing.
- **Large bulk loads of data**: Data is typically loaded in large batches (ETL/ELT processes) rather than individual transactions.
- **Large complex reporting queries**: Queries often involve aggregations, joins across many large tables, and full table scans.

Typical Workload & Performance Considerations

DW systems prioritize **query execution speed** for complex analytical queries, even if individual query latency is higher than OLTP. Data freshness is less critical than data consistency over time.

PostgreSQL Tuning Focus

Tuning for DW focuses on maximizing memory usage for large operations, optimizing sequential reads, and leveraging parallel processing.

- **work_mem**: **Crucial** for large sorts, hashes, and joins. Set this significantly higher (e.g., 256MB, 512MB, or even GBs) to prevent spilling to disk.

SHOW work_mem;

- **maintenance_work_mem**: Important for speeding up bulk loads (e.g., COPY) and index creation after loads. Set to 1GB or more.

SHOW maintenance_work_mem;

- **shared_buffers**: Still relevant, but a smaller percentage might be acceptable if the working set is truly massive and won't fit entirely.

SHOW shared_buffers;

- **random_page_cost**: Often set higher (e.g., 2.0-4.0) to encourage sequential scans over index scans for large queries, as most data will be read anyway.

SHOW random_page_cost;

- **effective_cache_size**: Set to a very high value (e.g., 75% of total RAM) to inform the planner that a lot of data *could* be cached, even if it's not all in shared_buffers.

SHOW effective_cache_size;

- **Parallel Query Execution**: Leverage multiple CPU cores for large scans, joins, and aggregations.
    - max_worker_processes
    - max_parallel_workers
    - max_parallel_workers_per_gather

SHOW max_worker_processes;

SHOW max_parallel_workers_per_gather;

- **Partitioning**: Very beneficial for large fact tables to limit scan scope.

```
-- Example of a DW query (complex aggregation)
EXPLAIN ANALYZE SELECT
  product_category,
  EXTRACT(YEAR FROM sale_date) AS sale_year,
  SUM(amount) AS total_sales,
  COUNT(DISTINCT customer_id) AS unique_customers
FROM sales_fact
```

JOIN products DIM ON sales_fact.product_id = products_dim.product_id

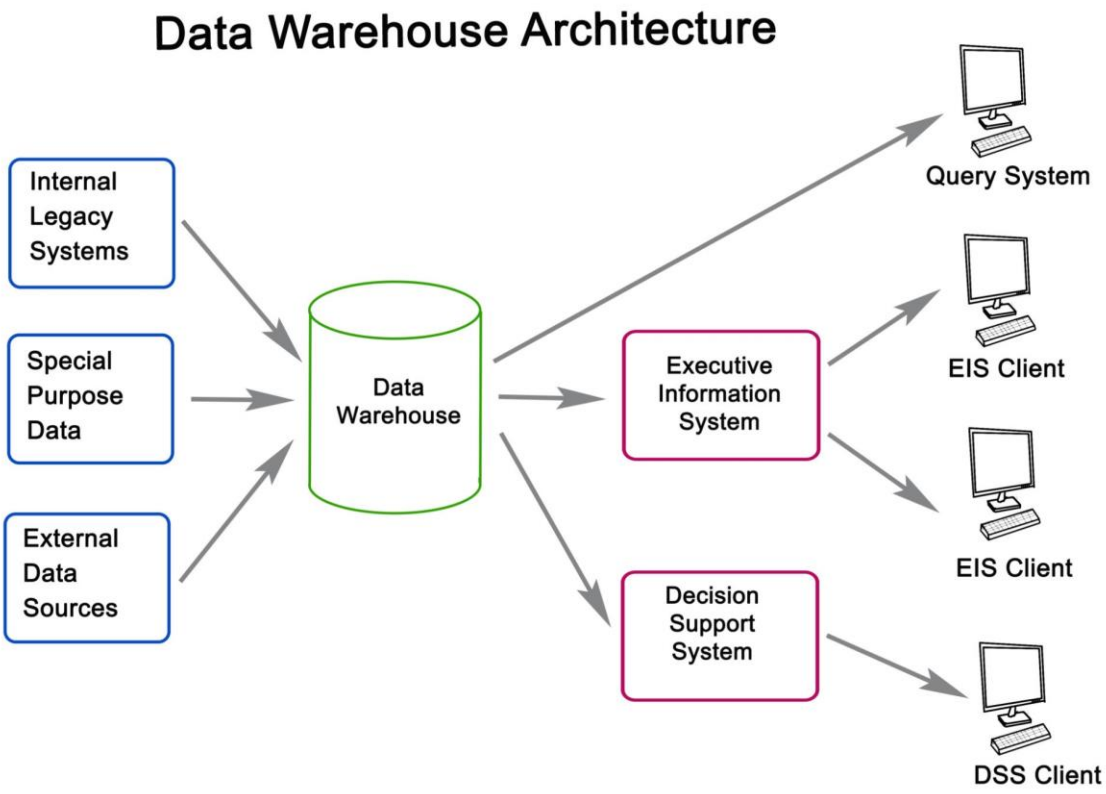
WHERE sale_date BETWEEN '2020-01-01' AND '2024-12-31'

GROUP BY product_category, sale_year

ORDER BY total_sales DESC;

Example Scenario

A retail company's data warehouse, storing years of sales, customer, and product data. Analysts run complex queries to identify top-selling products by region, customer segments, and seasonal trends. These queries might scan billions of rows.



Desktop Application 💻

A **desktop application database** refers to PostgreSQL running on a general workstation, typically for a single user or a small number of local users. This isn't a dedicated database server but rather a component of a local application (e.g., a developer's local environment, a GIS application, or a personal data management tool).

Characteristics

- **Not a dedicated database**: Shares resources with other applications running on the same machine.
- **A general workstation, perhaps for a developer**: Resource allocation needs to be balanced with other user activities.
- **Mixed type of application**: Workload can vary widely depending on the application, from simple CRUD operations to occasional complex data analysis.

Typical Workload & Performance Considerations

Performance is important for responsiveness of the desktop application, but resource consumption must be kept in check to avoid impacting the overall workstation experience. Data integrity is still important, but high availability might not be a primary concern.

PostgreSQL Tuning Focus

The tuning here is conservative, ensuring PostgreSQL doesn't hog resources.

- **max_connections**: Keep this low (e.g., 20-50) as there are few concurrent users.

SHOW max_connections;

- **shared_buffers**: Allocate a smaller percentage of RAM (e.g., 10-15%) to leave plenty for other desktop applications.

SHOW shared_buffers;

- **work_mem**: Default 4MB is often fine, or a slight increase if the desktop application performs local analytical tasks.

SHOW work_mem;

- **maintenance_work_mem**: Default 256MB is usually sufficient.

SHOW maintenance_work_mem;

- **autovacuum**: Ensure it's enabled to prevent bloat, but its parameters might be left at defaults or slightly adjusted to be less aggressive during active user hours.

SHOW autovacuum;

- **Logging**: Keep log_statement at none or ddl to avoid excessive log file growth on a personal machine.

SHOW log_statement;

Example Scenario

A developer running a local instance of their web application for testing and development. Or, a GIS application on a user's laptop that stores spatial data in a local PostgreSQL database. The database's performance is tied directly to the application's responsiveness.

Mixed Type of Application 🔄

A **mixed-type application** combines characteristics of both OLTP and Data Warehouse workloads. This is a common scenario in modern applications where users perform daily operational tasks *and* also run reports or analytics from the same database.

Characteristics

- **Mixed DW and OLTP characteristics**: Handles both high-volume transactional operations and complex analytical queries.
- **A wide mixture of queries**: Expect a blend of simple SELECT/INSERT/UPDATE/DELETE and resource-intensive aggregations/joins.

Typical Workload & Performance Considerations

This is the most challenging type to tune, as you need to find a balance. You must ensure that heavy analytical queries don't starve the transactional workload, and vice-versa. This often involves trade-offs.

PostgreSQL Tuning Focus

Tuning for a mixed workload requires careful balancing of parameters to accommodate both short, fast transactions and long, complex analytical queries.

- **shared_buffers**: A good balance is crucial. Often around 25% of RAM.

SHOW shared_buffers;

- **work_mem**: Set higher than OLTP (e.g., 64MB-256MB) to support analytical queries, but not so high that many concurrent queries exhaust RAM.

SHOW work_mem;

- **maintenance_work_mem**: Set generously (e.g., 512MB-1GB) to handle maintenance tasks for larger tables.

SHOW maintenance_work_mem;

- **WAL Parameters (wal_buffers, checkpoint_timeout, max_wal_size)**: Tune for the transactional part of the workload, balancing write performance and recovery time.

SHOW wal_buffers;

SHOW checkpoint_timeout;

SHOW max_wal_size;

- **Query Planner Costs (random_page_cost, effective_cache_size)**: Adjust carefully. You might lean slightly towards SSD-friendly settings if your storage is fast, but keep in mind that analytical queries might still benefit from sequential scans.

SHOW random_page_cost;

SHOW effective_cache_size;

- **Parallel Query Execution**: Enable and tune for analytical queries, but monitor its impact on transactional workload.

SHOW max_parallel_workers_per_gather;

- **Indexing Strategy**: A hybrid approach is needed. Create indexes for fast transactional lookups, but also consider partial indexes or expression indexes for specific analytical filters.

-- Example of a mixed workload:

-- OLTP part (customer order)

EXPLAIN ANALYZE INSERT INTO orders (customer_id, product_id, quantity) VALUES (456, 789, 2);

-- DW part (daily sales summary)

EXPLAIN ANALYZE SELECT product_id, SUM(quantity) FROM orders WHERE order_date = CURRENT_DATE GROUP BY product_id;

- **Resource Queues/Workload Management**: In more advanced setups, consider using connection pooling (like PgBouncer) with multiple pools or even separate database instances for OLTP and DW workloads if contention becomes too high.

Example Scenario

A SaaS platform that allows users to manage their projects (OLTP) and also provides dashboards and reports on project progress and team performance (DW). The database needs to efficiently handle both real-time project updates and complex, resource-intensive report generation.

Slow Query Problem

Here's a detailed guide on how to troubleshoot slow queries in PostgreSQL, including step-by-step solutions using psql scripts and GUI methods. 🛠️

1. Initial Diagnosis: Finding the Slow Query

The first step is to identify which queries are performing poorly.

Using pg_stat_statements (psql)

pg_stat_statements is a powerful extension that tracks planning and execution statistics of all SQL statements executed by a server.

1. **Install and Enable**: First, ensure the extension is enabled in your postgresql.conf file by adding pg_stat_statements to shared_preload_libraries. Then, restart PostgreSQL.

SQL

-- In postgresql.conf

shared_preload_libraries = 'pg_stat_statements'

2. **Create the Extension**: Once the server is restarted, connect to your database and create the extension.

SQL

CREATE EXTENSION pg_stat_statements;

3. **Find Slow Queries**: You can now query pg_stat_statements to find the most time-consuming queries.

SQL

SELECT

  query,

  calls,

  total_time,

  mean_time,

  rows

FROM

  pg_stat_statements

```
ORDER BY
  total_time DESC
LIMIT 10;
```

- query: The SQL statement itself.
- calls: The number of times the query was executed.
- total_time: The total time spent in milliseconds on this query across all executions.
- mean_time: The average execution time in milliseconds.
- rows: The total number of rows returned or affected.

Using a GUI (e.g., DBeaver or pgAdmin)

Modern GUI tools often have built-in dashboards to show performance stats. In pgAdmin's "Dashboard" tab for a database, you'll find "Sessions" and "Statistics" panels that can help. For DBeaver, you might need to run the pg_stat_statements query manually in a SQL editor.

---

2. Deep Dive: Understanding the Query Plan

Once you've identified a slow query, you need to understand *why* it's slow. This is where EXPLAIN ANALYZE comes in.

Using EXPLAIN ANALYZE (psql)

EXPLAIN ANALYZE shows the query planner's chosen execution plan and actually runs the query, providing real-world performance metrics.

1. **Run the command**: Prefix your slow query with EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON). Using FORMAT JSON is helpful for tools that can visualize the plan, and BUFFERS shows disk I/O.

SQL

```
EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON)
SELECT
  *
FROM
  my_table
```

```
WHERE
  created_at > '2023-01-01'
ORDER BY
  created_at DESC;
```

2. **Analyze the Output**: The output is a tree-like structure of nodes, each representing an operation (e.g., Seq Scan, Index Scan, Hash Join). Key metrics to look for include:
    - actual time: The time spent on this node (in milliseconds).
    - rows: The number of rows processed by this node.
    - cost: The planner's estimated cost (a unitless measure of work). A high cost doesn't always mean a slow query, but it's a good indicator.
    - Heap Fetches: Indicates table bloat, where multiple versions of the same row need to be read.

Using a GUI (e.g., DBeaver or pgAdmin)

Most GUI clients have a "visualize plan" feature. In pgAdmin, you can select your query and press the "Explain Analyze" button, which will display the plan in a graphical, easy-to-read format with costs and timing for each node. DBeaver offers similar functionality.

---

3. Common Bottlenecks and Solutions

Based on the EXPLAIN output, you can identify the root cause. Here's a diagram illustrating the troubleshooting workflow:

A. Sequential Scans on Large Tables 🔍

If EXPLAIN shows a **Seq Scan** on a large table, PostgreSQL is reading the entire table instead of using an index.

- **Cause**: Missing or unused index. The WHERE clause might not have an index, or the index might not be selective enough.
- **Solution**: **Create an index** on the columns used in the WHERE, ORDER BY, or JOIN clauses.

SQL

```
-- Example for a query with a WHERE clause
```

CREATE INDEX idx_my_table_created_at ON my_table (created_at);

B. Outdated Statistics 📈

The query planner uses table statistics to make decisions. If stats are old, the planner might choose a suboptimal plan.

- **Cause**: Tables have changed significantly (many INSERTs, UPDATEs, DELETEs) since the last ANALYZE.
- **Solution**: **Run ANALYZE** on the relevant tables.

SQL

ANALYZE my_table;

This should be done for parent tables when child tables change significantly.

C. High Sort Cost ⏳

If EXPLAIN shows a **Sort** node with a high actual time, the query is spending a lot of time sorting data, possibly spilling to disk.

- **Cause**: The amount of memory allocated for sorting (work_mem) is too low.
- **Solution**: **Increase work_mem** for your session (or globally, with caution).

SQL

SET work_mem TO '256MB';

-- Then run your query

You can also try creating an index that matches the ORDER BY clause to avoid the sort entirely.

D. Bloated Tables or Indexes 🧹

Bloat can cause Seq Scan to be inefficient and increase Heap Fetches.

- **Cause**: UPDATE and DELETE operations leave behind dead tuples. If VACUUM isn't running frequently enough, these dead tuples accumulate.
- **Solution**: **VACUUM FULL** (which is locking) or use a non-locking method like pg_repack to rebuild the table. In modern versions of PostgreSQL, REINDEX CONCURRENTLY can also help with bloated indexes without blocking.

SQL

-- Rebuilds an index without blocking writes

REINDEX CONCURRENTLY my_index;

E. Suboptimal Join Order 🤝

The planner might choose a poor order to join tables.

- **Cause**: Incorrect table statistics or complex queries where the planner makes a wrong assumption.
- **Solution**: You can use SET join_collapse_limit = 1; or hints (using the pg_hint_plan extension) to force a specific join order, but this should be a last resort. **Improving statistics is the better long-term solution.**

---

4. System-Wide Tuning ⚙️

These settings affect all queries on your server and can have a significant impact.

- **shared_buffers**: Memory allocated for PostgreSQL's shared buffer cache. A good starting point is **25% of your total RAM**.
- **effective_cache_size**: The planner's estimate of the total cache available to the database (including OS cache). Set this to **75% of your total RAM**.
- **wal settings**: For write-heavy workloads (INSERT, UPDATE, DELETE), adjusting max_wal_size and placing pg_wal on a separate, fast disk can significantly improve performance.

OR

🧠 **1. Postgres Version**

- Use: SELECT version();
- Why it matters:
  - PostgreSQL query planner evolves with each release.
  - Features like parallelism, JIT compilation, extended statistics, and planner hints vary across versions.
- Tip: Always include the full version string (e.g., PostgreSQL 15.3 on x86_64-pc-linux-gnu).

💻 **2. Operating System + Version**

- Use: tail /etc/*release (Linux)

- Why it matters:
    - Kernel I/O behavior, filesystem type (e.g., ext4 vs. xfs), and memory management affect query performance.
    - Helps determine compatibility with tuning tools and extensions.

📐 3. Full Table and Index Schema

- Use: \d table_name in psql
- Why it matters:
    - Reveals table structure, data types, constraints, and indexes.
    - Essential for understanding scan types (sequential vs. index), join strategies, and filter conditions.
- Include:
    - Table definitions
    - Index definitions
    - View or function definitions if referenced

📊 4. Table Metadata

- Use:

sql

SELECT relname, relpages, reltuples, relallvisible, relkind, relnatts, relhassubclass, reloptions, pg_table_size(oid)

FROM pg_class WHERE relname='your_table';

- Why it matters:
    - Helps estimate table size, visibility map coverage, and row count.
    - Reveals if autovacuum is keeping up or if bloat is accumulating.

📑 5. Table Characteristics

- Important flags:
    - Large objects (TOAST usage)
    - High NULL density
    - Frequent UPDATE/DELETE

- Rapid growth
- Many indexes
- Triggers or function calls

- Why it matters:
  - These factors influence planner decisions, index usage, and vacuum behavior.

🔍 6. EXPLAIN (ANALYZE, BUFFERS, SETTINGS)

- Use:

sql

SET track_io_timing = on;

EXPLAIN (ANALYZE, BUFFERS, SETTINGS) SELECT ...;

- Why it matters:
  - Shows actual vs. estimated row counts
  - Reveals I/O cost, memory usage, and planner settings
  - Essential for diagnosing misestimates, bad join orders, or missing indexes
- Tip: Share via explain.depesz.com

📈 7. Query History

- Questions to ask:
  - Was the query always slow?
  - Has data volume changed?
  - Have indexes or schema changed?
  - Do you have older plans for comparison?
- Why it matters:
  - Helps identify regressions or planner instability
  - Useful for pinpointing when performance degraded

🎛️ 8. Hardware

- Include:

- CPU model and core count
- RAM size
- Disk type (SSD vs. HDD)
- RAID configuration

- Why it matters:
  - Determines baseline performance expectations
  - Helps assess whether bottlenecks are hardware-related

🧪 9. Hardware Benchmark

- Use:

bash

bonnie++ -f -n0 -x4 -d /var/lib/pgsql

time dd if=/dev/sdX2 of=/dev/null bs=1M count=32K skip=$((128*$RANDOM/32))

- Why it matters:
  - Measures sequential read/write speed
  - Identifies disk I/O bottlenecks
  - Validates whether WAL or data files are on slow storage

🧹 10. Maintenance Setup

- Use:

sql

SELECT * FROM pg_stat_user_tables WHERE relname='your_table';

- Why it matters:
  - Autovacuum prevents bloat and keeps statistics fresh
  - Poor vacuum settings lead to slow queries due to outdated stats or bloated indexes
- Tip: Monitor autovacuum logs and tune thresholds for large or frequently updated tables

📦 11. WAL Configuration

- Key settings:

- wal_level, wal_buffers, checkpoint_timeout, max_wal_size, min_wal_size

- Why it matters:
  - WAL tuning affects write throughput and checkpoint frequency
  - Poor WAL configuration can cause I/O spikes and slow down INSERT/UPDATE-heavy workloads

⚙️ 12. GUC Settings

- Use:

sql

```
SELECT name, setting FROM pg_settings WHERE source != 'default';
```

- Key parameters:
  - shared_buffers, work_mem, effective_cache_size, maintenance_work_mem, random_page_cost, enable_seq_scan
- Why it matters:
  - These settings directly influence planner decisions and memory usage
  - Misconfigured values can lead to poor plans or excessive disk I/O

📊 13. Statistics: n_distinct, MCV, histogram

- Use:

sql

```
SELECT (SELECT sum(x) FROM unnest(most_common_freqs) x) AS frac_MCV,
    tablename, attname, inherited, null_frac, n_distinct,
    array_length(most_common_vals,1) AS n_mcv,
    array_length(histogram_bounds,1) AS n_hist,
    correlation
FROM pg_stats
WHERE attname='your_column' AND tablename='your_table'
ORDER BY 1 DESC;
```

- Why it matters:
  - Helps detect skewed distributions or poor cardinality estimates

- Useful for diagnosing bad join plans or filter selectivity issues

📝 14. Enable Logging

- Suggested settings:

conf

log_min_duration_statement = '2222ms'

log_autovacuum_min_duration = '9s'

log_checkpoints = on

log_lock_waits = on

log_temp_files = 0

log_destination = 'stderr,csvlog'

log_rotation_age = '2min'

log_rotation_size = '32MB'

logging_collector = on

- Why it matters:
  - Captures slow queries, lock waits, temp file usage, and checkpoint behavior
  - Essential for long-term performance monitoring and forensic analysis

---

Most Important Links to find out the Details information about Postgres troubleshooting and Optimization

1. Performance Tunning for Postgres
2. Performance Tunning by Josh Berkus
3. Postgres SQL Hardware Performance Tunning
4. The effect of data fragmentation
5. Index only scan performance
6. Optimize Postgres SQL Server Performance Through Configuration

Database Hardware Section and Setup

1. Database Hardware Solution
2. Reliable write

MS SQL DATABASE ADMINISTRATOR 30 DAYS COURSE

BY: Mukesh Chaurasia

Critical Maintenance Performance:

1. Planner Statics
2. Log in Deferent queries
3. Log in Check point
4. Bulk Loading and restores
5. Performance Analysis Tools

Database Architeecture

1. **Limitation and prioritizing user/query/database use**
2. **Prioritizing database by separating into multiple CLuster**
3. **Clustering**
4. **Shared Storage**