AWS RDS Performance Tuning Activities

**Benchmarking and Baseline Performance**

Benchmarking and establishing baseline performance in AWS RDS for PostgreSQL involves a series of steps to ensure that your database performance is understood and can be measured against future changes. Here's a structured approach:

1. Initial Setup

* Instance Selection: Choose an appropriate RDS instance type based on your workload. Ensure it meets your requirements in terms of CPU, memory, and storage.
* Parameter Group: Configure a custom parameter group for PostgreSQL settings that might impact performance (e.g., work\_mem, maintenance\_work\_mem, checkpoint\_timeout).

2. Monitoring Tools

* Amazon CloudWatch: Use CloudWatch to monitor metrics such as CPU utilization, memory, disk I/O, and network throughput.
* RDS Performance Insights: Enable Performance Insights to get detailed performance data and identify bottlenecks.
* pg\_stat\_statements: Enable the pg\_stat\_statements extension to capture SQL query performance data.

3. Benchmarking Tools

* pgbench: A built-in PostgreSQL benchmarking tool that can simulate various workloads.
* sysbench: A versatile benchmarking tool that can be used for database performance testing.

4. Baseline Performance Testing

1. Install pgbench:

sudo yum install postgresql-contrib

for windows, pgbench in included in pgadmin installation

1. Initialize the pgbench Database:

pgbench -i -s 50 -U postgres database1

-i initializes the database.

-s 50 specifies the scale factor. Adjust based on the size of your dataset.

1. Run the Benchmark:

pgbench -c 10 -j 2 -T 30 -U postgres database1

-c 10: Number of clients.

-j 2: Number of threads.

-T 600: Duration of the test in seconds.

1. Install sysbench:

sudo yum install sysbench

for windows: <https://github.com/akopytov/sysbench/releases>

1. Prepare the Database:

sysbench

--db-driver=pgsql

--pgsql-host=localhost

--pgsql-port=5432

--pgsql-user=sysbench\_user

--pgsql-password=password

--pgsql-db=sysbench\_db

--tables=10

--table-size=1000000

/usr/share/sysbench/oltp\_read\_write.lua

prepare

--db-driver=pgsql: Specifies the database driver.

--pgsql-host=localhost: PostgreSQL host (replace with your PostgreSQL host if different).

--pgsql-port=5432: PostgreSQL port (default is 5432).

--pgsql-user=sysbench\_user: PostgreSQL user.

--pgsql-password=password: PostgreSQL user password.

--pgsql-db=sysbench\_db: Database name.

--tables=10: Number of tables to create.

--table-size=1000000: Number of rows per table.

/usr/share/sysbench/oltp\_read\_write.lua: Path to the sysbench OLTP script (adjust the path if needed).

1. Run the Benchmark:

sysbench

--db-driver=pgsql

--pgsql-host=localhost

--pgsql-port=5432

--pgsql-user=sysbench\_user

--pgsql-password=password

--pgsql-db=sysbench\_db

--threads=16

--time=600

/usr/share/sysbench/oltp\_read\_write.lua

run

--threads=16: Number of threads to use.

--time=600: Duration of the test in seconds.

Adjust the parameters as needed based on your requirements.

1. Cleanup after test

sysbench

--db-driver=pgsql

--pgsql-host=localhost

--pgsql-port=5432

--pgsql-user=sysbench\_user

--pgsql-password=password

--pgsql-db=sysbench\_db

/usr/share/sysbench/oltp\_read\_write.lua

cleanup

* After the benchmark completes, sysbench will display results including transactions per second, latency statistics, and other metrics.
* Use this information to evaluate the performance of your PostgreSQL database.

5. Monitoring and Analyzing Results

* CloudWatch Metrics: Review CPU, memory, disk I/O, and network throughput metrics.
* Performance Insights: Analyze wait events, top SQL queries, and resource consumption.
* pg\_stat\_statements: Query this extension to identify slow queries and resource-heavy operations.

SELECT \* FROM pg\_stat\_statements ORDER BY total\_time DESC LIMIT 10;

6. Documentation and Baseline Creation

* Document Results: Record the results of your benchmarks, including configuration details, metrics, and any anomalies.
* Create Baseline: Establish a performance baseline from the collected data. This baseline will serve as a reference point for future performance comparisons.

7. Regular Monitoring and Re-Benchmarking

* Regular Intervals: Perform benchmarks at regular intervals or after significant changes (e.g., schema changes, instance type changes).
* Comparative Analysis: Compare new results with the baseline to identify performance degradation or improvements.

Additional Tips

* Isolation: Perform benchmarks in a controlled environment to minimize external influences.
* Automation: Use scripts or tools like AWS CloudFormation to automate benchmarking tasks.
* Review RDS Limits: Ensure you're aware of any RDS-specific limits that might affect your benchmarking results.

**Indexing**

Indexing is a critical technique to improve the performance of a PostgreSQL database by speeding up query execution times. Here are the steps and best practices for performing indexing to increase performance:

1. Understand Your Queries

* Analyze the queries that are run most frequently or take the longest time to execute.
* Use tools like pg\_stat\_statements to identify slow queries:

SELECT \* FROM pg\_stat\_statements ORDER BY total\_time DESC LIMIT 10;

2. Types of Indexes in PostgreSQL

* B-tree Index: The default and most common type of index. Suitable for equality and range queries.
* Hash Index: Optimized for equality comparisons. Not as commonly used as B-tree.
* GIN (Generalized Inverted Index): Useful for indexing array values, full-text search, and JSONB data.
* GiST (Generalized Search Tree): Suitable for complex data types such as geometric data and full-text search.
* BRIN (Block Range INdex): Efficient for large tables where values are naturally clustered.

3. Creating Indexes

Single-Column Index:

CREATE INDEX idx\_column\_name ON table\_name(column\_name);

Example:

CREATE INDEX idx\_customer\_name ON customers(customer\_name);

Multi-Column (Composite) Index:

CREATE INDEX idx\_columns ON table\_name(column1, column2);

Example:

CREATE INDEX idx\_customer\_order\_date ON orders(customer\_id, order\_date);

Unique Index:

CREATE UNIQUE INDEX idx\_unique\_column ON table\_name(column\_name);

Example:

CREATE UNIQUE INDEX idx\_unique\_email ON users(email);

Partial Index:

CREATE INDEX idx\_partial ON table\_name(column\_name) WHERE condition;

Example:

CREATE INDEX idx\_active\_customers ON customers(customer\_id) WHERE active = true;

Expression Index:

CREATE INDEX idx\_expression ON table\_name((expression));

Example:

CREATE INDEX idx\_lower\_email ON users((lower(email)));

Full-Text Search Index (GIN):

CREATE INDEX idx\_fts ON table\_name USING gin(to\_tsvector('english', column\_name));

Example:

CREATE INDEX idx\_fts\_content ON documents USING gin(to\_tsvector('english', content));

4. Maintaining Indexes

Reindexing: Periodically reindex to rebuild indexes and improve performance.

REINDEX TABLE table\_name;

Monitoring Index Usage: Use pg\_stat\_user\_indexes to monitor index usage and identify unused indexes.

SELECT indexrelname, idx\_scan, idx\_tup\_read, idx\_tup\_fetch

FROM pg\_stat\_user\_indexes

WHERE schemaname = 'public';

5. Best Practices

* Index Selective Columns: Create indexes on columns that are frequently used in WHERE clauses and have high selectivity.
* Avoid Over-Indexing: Too many indexes can degrade performance on write operations (INSERT, UPDATE, DELETE).
* Use Composite Indexes Wisely: Ensure the order of columns in composite indexes matches the order in query predicates.
* Regular Maintenance: Perform regular maintenance tasks like vacuuming and analyzing to keep indexes efficient.

VACUUM ANALYZE;

6. Example Scenario

Suppose you have an orders table and you want to optimize queries that filter by customer\_id and order\_date:

Identify the Query:

SELECT \* FROM orders WHERE customer\_id = 123 AND order\_date >= '2023-01-01';

Create a Composite Index:

CREATE INDEX idx\_customer\_order\_date ON orders(customer\_id, order\_date);

Analyze the Query Performance:

Use EXPLAIN to see how the query plan changes:

EXPLAIN SELECT \* FROM orders WHERE customer\_id = 123 AND order\_date >= '2023-01-01';

By carefully analyzing your queries and applying the appropriate indexing strategies, you can significantly improve the performance of your PostgreSQL database.

**Query Optimization**

Query optimization in PostgreSQL involves various techniques and best practices to improve the performance and efficiency of your SQL queries. Here’s a comprehensive guide on how to perform query optimization:

1. Understand the Query Execution Plan

Use EXPLAIN:

EXPLAIN SELECT \* FROM orders WHERE customer\_id = 123 AND order\_date >= '2023-01-01';

This command provides a query execution plan, showing how PostgreSQL intends to execute the query.

Use EXPLAIN ANALYZE:

EXPLAIN ANALYZE SELECT \* FROM orders WHERE customer\_id = 123 AND order\_date >= '2023-01-01';

This not only provides the execution plan but also runs the query and shows the actual execution time and rows processed.

2. Indexing

Single-Column Index:

CREATE INDEX idx\_customer\_id ON orders(customer\_id);

Composite Index:

CREATE INDEX idx\_customer\_order\_date ON orders(customer\_id, order\_date);

Partial Index:

CREATE INDEX idx\_active\_customers ON customers(customer\_id) WHERE active = true;

Full-Text Search Index:

CREATE INDEX idx\_fts\_content ON documents USING gin(to\_tsvector('english', content));

3. Query Rewriting

SELECT Specific Columns:

Instead of SELECT \*, specify the columns needed:

SELECT customer\_id, order\_date, total\_amount FROM orders WHERE customer\_id = 123;

Use Proper Joins:

Prefer using explicit JOIN syntax:

SELECT o.order\_id, c.customer\_name

FROM orders o

JOIN customers c ON o.customer\_id = c.customer\_id

WHERE c.active = true;

Avoid Functions on Indexed Columns:

Avoid applying functions to indexed columns in the WHERE clause:

SELECT \* FROM users WHERE LOWER(email) = 'example@example.com';

Instead, create an index on the expression:

CREATE INDEX idx\_lower\_email ON users((LOWER(email)));

4. Optimization Techniques

Use LIMIT for Pagination:

Use LIMIT and OFFSET for efficient pagination:

SELECT \* FROM orders WHERE customer\_id = 123 ORDER BY order\_date DESC LIMIT 10 OFFSET 20;

Use EXISTS Instead of IN:

Prefer EXISTS for subqueries:

SELECT \* FROM orders o

WHERE EXISTS (SELECT 1 FROM customers c WHERE c.customer\_id = o.customer\_id AND c.active = true);

Avoid SELECT DISTINCT When Not Necessary:

Ensure that DISTINCT is needed to eliminate duplicates:

SELECT DISTINCT customer\_id FROM orders;

5. Database Configuration

Adjust Work\_mem:

Increase work\_mem for queries requiring large sort operations or hash joins:

SET work\_mem = '64MB';

Analyze and Vacuum:

acuum your tables to update statistics and reclaim storage:

VACUUM ANALYZE;

Use pg\_stat\_statements:

Enable and use the pg\_stat\_statements extension to track execution statistics of all SQL statements:

CREATE EXTENSION pg\_stat\_statements;

SELECT \* FROM pg\_stat\_statements ORDER BY total\_time DESC LIMIT 10;

**Partitioning**

Partitioning involves dividing a large table into smaller, more manageable pieces called partitions. Each partition holds a subset of the table’s data.

**Steps to Implement Partitioning**

Create a Partitioned Table

CREATE TABLE orders (

order\_id SERIAL PRIMARY KEY,

customer\_id INT NOT NULL,

order\_date DATE NOT NULL,

total\_amount NUMERIC

) PARTITION BY RANGE (order\_date);

Create Partitions

CREATE TABLE orders\_2023 PARTITION OF orders

FOR VALUES FROM ('2023-01-01') TO ('2023-12-31');

CREATE TABLE orders\_2024 PARTITION OF orders

FOR VALUES FROM ('2024-01-01') TO ('2024-12-31');

Insert Data

INSERT INTO orders (customer\_id, order\_date, total\_amount) VALUES (1, '2023-06-15', 100.00);

Query Data

SELECT \* FROM orders WHERE order\_date BETWEEN '2023-01-01' AND '2023-12-31';

Indexing Partitions

Indexes should be created on individual partitions:

CREATE INDEX idx\_orders\_2023\_customer\_id ON orders\_2023 (customer\_id);

CREATE INDEX idx\_orders\_2024\_customer\_id ON orders\_2024 (customer\_id);

Benefits of Partitioning

* Improved query performance for large datasets.
* Simplified maintenance and management.
* Efficient data archiving and purging.

**Connection and Concurrency Management**

1. Connection Management

**Connection Pooling**

Connection pooling is essential for managing database connections efficiently, reducing the overhead of establishing new connections.

**Using RDS Proxy**

AWS RDS Proxy is a managed database proxy for RDS that helps improve application availability, performance, and security.

Create an RDS Proxy:

1. Go to the RDS Management Console.
2. Navigate to "Proxies" and click "Create proxy."

Configure the proxy settings, including:

* Proxy Identifier: Name for the proxy.
* Engine Compatibility: Select PostgreSQL.
* Database: Select your RDS instance.
* Secrets: Select the AWS Secrets Manager secret for database credentials.
* IAM Role: Create or select an IAM role with the required permissions.

Configure Proxy Target:

* Add the RDS instance as the proxy target.

Connect to RDS Proxy:

* Modify your application to connect to the RDS Proxy endpoint instead of the RDS instance endpoint.

**Using PgBouncer**

Install PgBouncer on an EC2 Instance:

1. Launch an EC2 instance and install PgBouncer.
2. Follow the same steps as the non-RDS PgBouncer installation (described earlier).

Configure PgBouncer to Connect to RDS:

* Modify pgbouncer.ini to point to your RDS instance:

ini

[databases]

mydb = host=<RDS-Endpoint> port=5432 dbname=mydb user=<username> password=<password>

[pgbouncer]

listen\_addr = 0.0.0.0

listen\_port = 6432

auth\_type = md5

auth\_file = /etc/pgbouncer/userlist.txt

pool\_mode = transaction

max\_client\_conn = 100

default\_pool\_size = 20

Start PgBouncer:

pgbouncer -d /etc/pgbouncer/pgbouncer.ini

**Connection Limits**

Set appropriate connection limits to prevent resource exhaustion.

Modify RDS Instance Parameters:

* In the RDS Management Console, modify the parameter group associated with your RDS instance.
* Set max\_connections to a reasonable number based on your instance class and workload.

Database User Limits:

Set limits for specific users or roles:

ALTER ROLE myuser CONNECTION LIMIT 10;

2. Concurrency Management

Concurrency management involves controlling how multiple transactions interact to ensure data consistency and performance.

Isolation Levels

PostgreSQL supports different isolation levels that control the visibility of changes made by concurrent transactions.

Set Isolation Level:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

**Locking**

Locks are mechanisms to control concurrent access to data. PostgreSQL uses different types of locks:

Row-Level Locks: Prevents concurrent updates to the same row.

SELECT \* FROM orders WHERE order\_id = 1 FOR UPDATE;

Table-Level Locks: Prevents concurrent modifications to a table.

LOCK TABLE orders IN SHARE MODE;

**Deadlock Detection**

PostgreSQL automatically detects deadlocks, but proper transaction design can minimize their occurrence.

* Avoid Long-Running Transactions: Keep transactions short and efficient.
* Consistent Lock Ordering: Ensure transactions acquire locks in a consistent order.
* Monitoring Concurrency

**RDS Performance Insights:**

Enable and use RDS Performance Insights to monitor database load and identify bottlenecks.

In the RDS Management Console, select your DB instance and enable Performance Insights.

**CloudWatch Metrics:**

* Monitor key metrics like CPUUtilization, DatabaseConnections, and ReadIOPS using Amazon CloudWatch.
* Set up CloudWatch Alarms to alert you when thresholds are breached.
* pg\_stat\_activity: View active connections and their states.

SELECT \* FROM pg\_stat\_activity;

* pg\_locks: View lock information and detect potential issues.

SELECT \* FROM pg\_locks;

* pg\_stat\_user\_tables: Monitor table-level statistics.

SELECT \* FROM pg\_stat\_user\_tables;

**Example Scenario**

Suppose you have an application with high concurrency and you want to optimize connection and concurrency management on AWS RDS:

1. Setup RDS Proxy:

Create and configure an RDS Proxy as described above.

1. Configure RDS Instance Parameters:

Modify the parameter group to set max\_connections to a reasonable number based on your instance class and workload.

1. Implement Connection Limits:

Limit connections for specific roles:

ALTER ROLE appuser CONNECTION LIMIT 50;

1. Optimize Transaction Design:

Use appropriate isolation levels:

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

1. Monitor and Adjust:

* Regularly monitor Performance Insights, CloudWatch metrics, and PostgreSQL system views like pg\_stat\_activity and pg\_locks to detect and resolve issues.

**Maintaining long-term performance**

1. Regular Maintenance Tasks

**Automated Backups**

AWS RDS provides automated backups to restore your database to a specific point in time.

Enable Automated Backups:

* In the RDS Management Console, ensure that automated backups are enabled for your instance.
* Configure the retention period according to your needs.

Manual Snapshots:

* Regularly take manual snapshots, especially before major changes or maintenance activities.

**Routine Vacuuming and Analyzing**

To maintain optimal performance, regularly vacuum and analyze your tables.

AutoVacuum:

Ensure autovacuum is enabled in the RDS parameter group.

SELECT name, setting FROM pg\_settings WHERE name = 'autovacuum';

Adjust autovacuum settings if necessary to handle your workload.

Manual VACUUM and ANALYZE:

Regularly run VACUUM and ANALYZE to reclaim storage and update statistics.

VACUUM (VERBOSE, ANALYZE);

Reindexing

Reindexing helps maintain the efficiency of indexes.

Identify Fragmented Indexes:

SELECT indexrelid::regclass AS index,

pg\_size\_pretty(pg\_relation\_size(indexrelid)) AS index\_size,

pg\_size\_pretty(pg\_relation\_size(indrelid)) AS table\_size

FROM pg\_stat\_user\_indexes

JOIN pg\_index

ON pg\_stat\_user\_indexes.indexrelid = pg\_index.indexrelid;

Reindex Tables or Indexes:

REINDEX TABLE mytable;

REINDEX INDEX myindex;

2. Monitoring and Performance Tuning

Amazon CloudWatch

Use CloudWatch to monitor RDS performance metrics.

Key Metrics to Monitor:

* CPUUtilization: High CPU usage may indicate the need for optimization or scaling.
* FreeableMemory: Low memory can lead to performance issues.
* DatabaseConnections: Monitor the number of active connections.
* ReadIOPS and WriteIOPS: High I/O operations can indicate a need for better indexing or more efficient queries.
* Set Alarms:

Configure CloudWatch Alarms to alert you when metrics exceed predefined thresholds.

aws cloudwatch put-metric-alarm --alarm-name HighCPUUsage --metric-name CPUUtilization --namespace AWS/RDS --statistic Average --period 300 --threshold 80 --comparison-operator GreaterThanOrEqualToThreshold --dimensions Name=DBInstanceIdentifier,Value=mydbinstance --evaluation-periods 2 --alarm-actions arn:aws:sns:region:account-id:sns-topic

Performance Insights

Enable and use Amazon RDS Performance Insights to get deeper visibility into database performance.

* Enable Performance Insights:

In the RDS Management Console, enable Performance Insights for your instance.

* Analyze Performance:

Use Performance Insights to identify slow queries and database bottlenecks.

Optimize slow queries by adding indexes, rewriting queries, or adjusting database parameters.

PostgreSQL System Views

Use PostgreSQL system views for detailed monitoring.

pg\_stat\_activity:

Monitor active sessions and their state.

SELECT \* FROM pg\_stat\_activity;

pg\_stat\_user\_tables:

Monitor table-level statistics.

SELECT \* FROM pg\_stat\_user\_tables;

pg\_locks:

View lock information and detect potential issues.

SELECT \* FROM pg\_locks;

3. Scaling and High Availability

Scaling

Scale your RDS instance based on workload.

Vertical Scaling:

* Modify your RDS instance class to a larger size if you need more CPU, memory, or I/O capacity.
* In the RDS Management Console, select your instance, click "Modify," and choose a larger instance class.

Horizontal Scaling:

* Use Read Replicas to offload read traffic from the primary instance.

**High Availability**

Ensure high availability and fault tolerance.

Multi-AZ Deployments:

* Enable Multi-AZ deployment for automatic failover.
* In the RDS Management Console, select your instance, click "Modify," and enable Multi-AZ deployment.

Automated Backups and Snapshots:

* Ensure automated backups are enabled and regularly take manual snapshots.

4. Security and Compliance

Regular Security Patches

* Ensure your RDS instance is running the latest security patches.

Automatic Minor Version Upgrades:

* Enable automatic minor version upgrades in the RDS Management Console.

Access Control

* Implement robust access controls.
* IAM Policies:

Use IAM policies to control access to your RDS instance.

Security Groups:

* Configure security groups to allow only trusted IP addresses and applications to connect to your RDS instance.

Encryption:

* Enable encryption at rest and in transit for sensitive data.