Tips and Tricks for Optimizing Shared Buffers in PostgreSQL

Optimizing shared buffers in PostgreSQL

Optimizing shared buffers in PostgreSQL is critical for improving performance by efficiently utilizing memory for caching data pages, reducing disk I/O, and enhancing query execution speed. Below are key tips and tricks for tuning shared_buffers in PostgreSQL.

1. Understand the Purpose of shared_buffers

- shared_buffers is the memory allocated to PostgreSQL for caching frequently accessed data pages
- It works alongside the OS page cache but allows PostgreSQL to manage the cached data directly for better query planning and execution

2. Allocate the Right Amount of Memory

A. General Rule of Thumb

- Set shared_buffers to 25-40% of total system memory for most workloads
- Example for a system with 16 GB of memory:

```
shared_buffers = '4GB'
```

- For write-heavy workloads, you might need to increase it further, but avoid going beyond 50% of the system memory to leave room for the OS cache

B. Test and Benchmark

- Start with 25% of RAM and gradually increase it, monitoring query performance and system metrics

- Use tools like pgbench or actual workload testing to benchmark performance after adjustments

3. Monitor Cache Hit Ratios

A. Check PostgreSQL Cache Usage

Query the cache hit ratio to determine how effectively shared_buffers is being used:

```
SELECT
    name,
    setting,
    CASE
        WHEN name = 'shared_buffers' THEN
            ROUND(setting::NUMERIC / 1024 / 1024, 2) || ' GB'
        ELSE setting
    END AS value
FROM pg_settings WHERE name IN ('shared_buffers');
```

Ideal Cache Hit Ratio: Aim for a cache hit ratio > 99%, meaning most data requests are served from memory rather than disk:

```
SELECT
    blks_hit * 100.0 / (blks_read + blks_hit) AS cache_hit_ratio
FROM pg_stat_database
WHERE datname = 'your_database';
```

B. Adjust if Needed

- If the cache hit ratio is low, increase shared_buffers incrementally and retest

4. Leverage the OS Cache

A. Balance PostgreSQL and OS Caches

- Leave enough memory for the OS to handle file system caching

- Use tools like free -h (Linux) or vmstat to monitor OS memory usage

B. Adjust Kernel Settings

On Linux, configure the OS to favor PostgreSQL's shared buffers:

```
sysctl -w vm.swappiness=10  # Reduce OS cache swapping
```

5. Monitor pg_buffercache Usage

A. Install and Use the pg_buffercache Extension

pg_buffercache provides insights into what is stored in shared_buffers:

```
CREATE EXTENSION IF NOT EXISTS pg_buffercache;
SELECT c.relname, count(*) AS buffers, isdirty
FROM pg_buffercache b
JOIN pg_class c ON b.relfilenode = c.relfilenode
GROUP BY c.relname, isdirty
ORDER BY buffers DESC;
```

- This helps identify which tables or indexes dominate the cache and whether they benefit query performance

6. Optimize for Workload Characteristics

A. For Read-Heavy Workloads

- Increase shared_buffers to keep more frequently accessed data in memory
- Monitor and tune indexes to reduce redundant data scans

B. For Write-Heavy Workloads

Tune wal_buffers (often set to 3-16 MB) in conjunction with shared_buffers to handle high write volumes without contention:

```
ALTER SYSTEM SET wal_buffers = '16MB';
```

Increase checkpoint_completion_target to spread write operations more evenly:

```
ALTER SYSTEM SET checkpoint_completion_target = 0.9;
```

7. Monitor Checkpoints

- Frequent checkpoints indicate excessive dirty buffer writes, which could result from a small shared_buffers setting

Check the frequency of checkpoints:

```
SELECT checkpoints_timed, checkpoints_req, buffers_checkpoint
FROM pg_stat_bgwriter;
```

- If checkpoints_req is high, increase shared_buffers or tune checkpoint_timeout and max_wal_size

8. Adjust for Specific Workloads

A. Analytical Workloads

- Analytical queries benefit from a higher shared_buffers setting, as larger working sets are accessed repeatedly
- Combine this with increased work_mem for complex queries

B. Transactional Workloads

Balance shared_buffers and max_connections to avoid memory starvation from too many connections:

```
ALTER SYSTEM SET max_connections = 200;  -- Example adjustment
```

9. Configure PostgreSQL Logging for Shared Buffer Insights

Enable detailed logging to analyze buffer usage and identify bottlenecks:

```
log_checkpoints = on
log_temp_files = 0
```

10. Regularly Vacuum and Analyze

Ensure autovacuum is keeping table statistics and visibility maps up to date for efficient use of shared buffers:

```
VACUUM (VERBOSE, ANALYZE);
```

11. Avoid Over-Provisioning

- If shared_buffers is too large:
  - It can lead to memory pressure on the OS and increased risk of swapping
  - Monitor swapping with vmstat or sar

12. Use Tools to Monitor Performance

A. pg_stat_statements

Enable the pg_stat_statements extension to track which queries are causing high buffer usage:

```
CREATE EXTENSION pg_stat_statements;
SELECT query, calls, total_exec_time, shared_blks_hit, shared_blks_read
FROM pg_stat_statements
ORDER BY shared_blks_hit DESC;
```

B. Performance Insights in AWS RDS

- If running PostgreSQL on RDS, use Performance Insights to analyze memory usage and buffer-related waits

By applying these tips and continuously monitoring your workload, you can fine-tune shared_buffers for optimal PostgreSQL performance. 🚀

Optimizer index caching in PostgreSQL

Optimizer index caching in PostgreSQL is the process by which the database caches the index pages in memory to speed up the execution of queries. This feature helps improve the performance of queries by reducing … Continue reading

The WebScale Database Infrastructure Operations Experts in PostgreSQL, MySQL, MariaDB, MongoDB and ClickHouse

Understanding Shared Buffers Implementation in PostgreSQL

In PostgreSQL, shared buffers serve as the shared memory pool used to cache data pages in memory to improve read performance. Whenever a query needs to access data, it checks if the required pages are … Continue reading

The WebScale Database Infrastructure Operations Experts in PostgreSQL, MySQL, MariaDB, MongoDB and ClickHouse

Tuning PostgreSQL Server – How are data block visits and undo implemented in PostgreSQL?

In PostgreSQL, data block visits and undo are implemented as part of the storage and transaction management systems. In summary, PostgreSQL implements data block visits and undo through a combination of the buffer manager, the … Continue reading

The WebScale Database Infrastructure Operations Experts in PostgreSQL, MySQL, MariaDB, MongoDB and ClickHouse