EDB
Postgres for the AI Generation

Navigating Challenging Customer Issues in PostgreSQL:
Strategies for Resolution and Prevention

Dilip Kumar and Rushabh Lathia
06-03-2025

Content

- Understanding critical PostgreSQL Issues
- Key challenges
- Why immediate action is necessary
- Critical PostgreSQL Issues
- PostgreSQL database good practices

Understanding Critical PostgreSQL Issues

- Severe database issues can impact performance, data integrity, and system availability.
- Delayed resolution can lead to downtime, financial loss, and customer dissatisfaction.
- These issues can escalate quickly and cause cascading failures.

Key Challenges

- Identifying root causes amidst complex interdependencies.
- Prioritizing urgent issues while maintaining service availability.
- Coordinating across teams (DBAs, DevOps, and application engineers) for rapid mitigation.

Why immediate action is necessary

- Prevents minor issues from turning into major system failures.
- Protects data integrity and ensures business continuity.
- Reduces operational risks and improves customer trust.

PostgreSQL Critical issues & EDB.

- Certified PostgreSQL Experts response for any type of issues.
  - Reduce risk. Protecting revenu and reputation.
- Support always available for virtual assistance
  - Engineers available via **virtual calls** for real-time troubleshooting.
- EDB bring PostgreSQL internal expertise and operational expertise.
  - Helps in providing resolution faster
  - Hot-fix
- It offers proactive support, including fundamental source code assistance and unlimited access to a customer support portal.

Critical PostgreSQL Issues

- Case 1: Subtransaction cache overflow
- Case 2: Database corruption
- Case 3: Autovacuum failing, causing transaction wraparound risks
- Case 4: Streaming replication lag

Case 1: Subtransaction cache overflow

- A high number of nested subtransactions exceeded the subtransaction cache.
- PostgreSQL had to look up subtransactions in SLRU cache and disk.
- Performance degradation due to excessive disk I/O and lock contention.
- Increased query latency and new connection failures.
- Applications experiencing slow performance and timeouts.

Detecting the issue

- High number of wait event are observed on SubtransSLRU and SubtransBuffers
- Those wait event indicates that we are accessing the subtransaction to parent mapping from SLRU
- Which will cause high contention on SLRU related lock

Cause of the issue

- Frequent buffer replacement in cache
  - This mainly occurs if wider range of transaction data are getting accessed

- High contention on centralized control lock
  - When a lot of readers are accessing the SLRU along with writers.
  - This will also be caused by frequent SLRU buffer eviction

Mitigation plan

- Review application why you need more than 64 subtransaction
  - Is it intentional or by mistake
  - Look for nested procedure with exception block
  - Check for savepoint count under a transaction
- Can we modify the application
- Short term relief for customer
  - https://www.enterprisedb.com/blog/life-bug-customer-escalation-postgresql-commit
- Fixed in open source PG
  - Provided GUC to increase cache size
  - Also optimized and reduced lock contention

Case 2: Database Corruption

- What is Database Corruption?

  Database corruption occurs when the data stored in a database becomes inconsistent, unreadable, or lost due to various factors.

- Common Causes:
  - Hardware Failures – Disk corruption, memory errors, power failure
  - Software Bugs – Issues in PostgreSQL, extensions, database tools, OS, Kernel, filesystem
  - Admin Errors – Incorrect system reboots, unsafe manual modifications
  - User Error - Faulty backup and recovery procedure, upgrade, etc

- Corruption is rare but serious! Prevention & quick diagnosis are crucial.

How to detect database corruption

- Symptoms of Corruption
  - Unexpected errors like:
    - ERROR: could not read block X in relation Y
    - ERROR: invalid page header in block
    - ERROR: cache lookup failed
  - Inconsistent query results
  - PostgreSQL startup failures
  - Random server crash

- Detection Methods
  - Check PostgreSQL Logs: Look for I/O errors or corrupt page messages
  - Use pg_catcheck to identify PG catalog corruption
  - Run pg_dump or pg_verify_checksums: may fail if corruption exists
  - Use amcheck extension: Checks index and table integrity
  - Tools like pg_waldump can be used to check for issues in WAL files.
    - Look for **missing sequences, broken records, or unusual transaction behavior**.

Recovering from Database Corruption

- Identify the Cause
  - Check logs and errors (`dmesg`, `journalctl`, PostgreSQL logs)
- Run `fsck`, check memory (`memtest`), verify disk health (`smartctl`)
- Use pg_surgery module, provides various functions to perform surgery on a damaged relation
- If possible, use a recent physical or logical backup
- Use `pg_dump --clean` (if only part of the DB is corrupt)
- Rebuild indexes (`REINDEX`), vacuum tables (`VACUUM FULL`)

Prevention Tips

- Enable checksum: Use initdb --data-checksums to catch corruption early.
- fsync = ON: PostgreSQL ensures that data is flushed to disk before transaction commits
- full_page_writes = ON: writes the full content of each modified database page to the WAL the first time it is modified after a checkpoint.
- Enable full backups or incremental backups and WAL archiving
- Monitor PostgreSQL Logs
- Keep a standby replica for failover in case of corruption.
- Avoid kill -9 and ensure proper system shutdowns to prevent data loss.
- Run Integrity Checks Regularly (like pg_catcheck, pg_amcheck and pg_checksums)

Case 3: Autovacuum failing & wraparound risks

- What is Autovacuum Failing:
  - Autovacuum in PostgreSQL is a background process responsible for cleaning up dead tuples and preventing transaction ID (XID) wraparound.
  - If autovacuum fails, it means that PostgreSQL is unable to perform vacuuming operations effectively, leading to table bloat, degraded performance.

- Common reasons why autovacuum fails:
  - Long-running transactions blocking autovacuum.
  - High system load causing autovacuum to lag.
  - Disk I/O or memory bottlenecks.
  - Insufficient `autovacuum_max_workers` or `autovacuum_naptime` settings.
  - Autovacuum disabled manually (`autovacuum = off`).

Case 3: Autovacuum failing & wraparound risks

- What is Transaction ID Wraparound Risk:
  - PostgreSQL uses a 32-bit transaction ID (XID), which means it can handle approximately **2 billion transactions** before it must "wrap around."
  - If a table is not vacuumed in time, older XIDs are frozen, and when the wraparound limit is reached, in order to prevent data corruption - PostgreSQL will stop accepting new transactions.

- Signs of Wraparound Risk:
  - PostgreSQL logs show warnings like:
    - database "your_database" must be vacuumed within 200000 transactions
    - ERROR: uncommitted xmin 338162537 from before xid cutoff 2428844533 needs to be frozen
  - Check for autovacuum failures: SELECT relname, last_autovacuum, last_autoanalyze, n_dead_tup FROM **pg_stat_user_tables** ORDER BY n_dead_tup DESC;
  - Queries slow down due to excessive transaction ID lookups.
  -

Prevent Autovacuum Failing & Wraparound Risks

- Tune autovacuum setting.
  - Change postgres.conf setting to make vacuum more aggressive, Example:

```
autovacuum_naptime = 10s              # Reduce time between autovacuum runs
autovacuum_vacuum_threshold = 50      # Lower threshold for vacuuming small tables
autovacuum_analyze_threshold = 50
autovacuum_vacuum_cost_limit = 2000 # Increase cost limit to allow more work per cycle
autovacuum_max_workers = 5            # Increase workers for parallel processing
```

- autovacuum_freeze_max_age setting
  - Specifies the maximum age (in transactions) that a table's pg_class.relfrozenxid field can attain before a VACUUM operation is forced to prevent transaction ID wraparound within the table.
  - Keep watch on “cutoff for removing and freezing tuples is far in the past” WARININGs in log file
- Keep a watch on log running transactions:
  - SELECT pid, age(query_start), query FROM pg_stat_activity WHERE state = 'active' ORDER BY age(query_start) DESC;

Prevent Autovacuum Failing & Wraparound Risks

- Check Transaction Age
  - SELECT datname, age(datfrozenxid) as xid_age, 2000000000 - age(datfrozenxid) AS remaining_xids FROM pg_database ORDER BY xid_age DESC;
- Warning Levels
  - Above 1.5 billion XIDs –> High risk
  - Above 2 billion XIDs –> Database stop generating new transaction IDs
- Check Individual Table Risks
  - SELECT relname, age(relfrozenxid) AS xid_age FROM pg_class WHERE relkind = 'r' ORDER BY xid_age DESC LIMIT 10;
  - Tables with high `xid_age` need **freezing**!

Resolution and best practices

- Resolution
  - Increased autovacuum_vacuum_cost_limit to run more aggressive vacuuming.
  - Manually vacuumed high-risk tables (VACUUM FREEZE).
  - Enforced shorter transaction lifespans to allow vacuuming.
  - Tune Autovacuum Settings
  - Kill long transactions (if needed) to prevent autovacuum blocking.
  - In worst cases, use single-user mode to recover PostgreSQL from shutdown.
- Best Practices
  - Regularly monitor pg_stat_all_tables for autovacuum activity.
  - Set up alerts for age of transactions.
  - Schedule proactive vacuums for high-write tables.

Case 4: Streaming Replication Lag

Streaming replication continuously sends WAL (Write-Ahead Log) changes from the primary to one or more replica servers. The replica applies the changes sequentially, as they're received.

- Replication lag is the delay between the time when data is written to the primary database and the time when it is replicated to the standby databases
- Replication lag can impact performance and system availability.
- It can result in data loss when failover occurs.

What causes replication lag

This lag occurs when the rate at which changes are generated on the primary exceeds the rate at which they can be processed and applied to the replica server.

- Network latency between the nodes or network congestion
- Slow disk I/O can delay writing data to the standby databases.
- Limited CPU cores or slow processors on replica.
- High concurrency workloads on the primary.

Identifying the Replication Lag

- Query to run on Primary

```
SELECT pid,application_name,client_addr,client_hostname,state,sync_state,replay_lag   FROM
pg_stat_replication ;
```

- Query to run on standby

```
SELECT pg_is_in_recovery(),pg_is_wal_replay_paused(), pg_last_wal_receive_lsn(),
pg_last_wal_replay_lsn(), pg_last_xact_replay_timestamp();
```

Mitigating replication lag

- Monitoring replication lag
  - Monitoring replication lag can help identify the cause of the lag and take appropriate actions to mitigate it.

- Increasing the network bandwidth

- Tuning PostgreSQL configuration parameters
  - Tuning wal_buffers, wal_writer_delay, etc., can reduce lag.
  - Enable wal_compression in order to reduce the traffic.
  - Add extra processing power on replica if needed

PostgreSQL database good practices

- Monitor database performance
  - Check running queries in pg_stat_activity
  - Identify long-running queries that might cause performance issues.

- Check locks
  - Detect queries causing deadlocks or blocking others pg_locks

- Monitor slow queries (enable logging)
  - log_min_duration_statement
  - Identify the unused (pg_stat_user_indexes)/missing index.

PostgreSQL database good practices

- Regular maintenance
  - Prevent table bloat and update statistics for the query planner: VACUUM ANALYZE
- Autovacuum monitor
  - SELECT * FROM pg_stat_all_tables WHERE last_autovacuum IS NULL;
  - List of tables where autovacuum has never run.
- Reindex tables to optimize performance
- Monitor Database Size (pg_database)

PostgreSQL database good practices

- Backup & disaster recovery
  - Take regular backups
  - Check replication status (pg stat replication)
  - Test your recovery plans.
- Resource usage & hardware monitoring
  - Check CPU and memory usage
  - Check disk space.
  - Monitor active connections
- Upgrade
  - Upgrade the minor release.
  - Upgrade the extensions.
  - Upgrade to major release.

EDB
Postgres for the AI Generation

Thank You!

References: PGConf India 2020 - Avoiding, Detecting, and Recovering From Data Corruption - Robert Haas - EDB

Various PostgreSQL Blogs and document.