Removing Bloat with pg_repack Extension

Introduction

Regular updates and deletions within PostgreSQL tables can lead to various issues such as bloat, fragmentation, and a decline in performance over time. These challenges can significantly impact the efficiency and reliability of the database, potentially affecting critical operations.

To address these concerns, PostgreSQL introduced the **pg_repack** extension, which provides a robust solution for managing table maintenance without disrupting the production environment. By allowing tables to be rebuilt online, pg_repack tackles bloat and fragmentation issues, ensuring that database storage remains optimized and performance is consistently maintained.

Problems with VACUUM FULL

PostgreSQL has built in **VACUUM** process to reclaim storage occupied by obsolete data to ensure optimal performance by freeing up space for future use and efficiency. However, the **VACUUM FULL** operation, a variant of the standard VACUUM process, presents distinct challenges that must be addressed.

In contrast to typical VACUUM operations, which can proceed alongside other database DML activities, VACUUM FULL begins by obtaining an exclusive lock on the specified table. This reconstructs the table from active rows and arranges them in a compact manner to reclaim filesystem space. While this lock is in place, no reads, writes, or schema alterations can occur on the table, which can potentially lead to notable downtime and service interruptions.

pg_repack to the rescue

pg_repack allows you to clean up unnecessary space from tables and indexes and, if needed, reorder clustered indexes physically. It operates online, meaning it doesn't need to lock tables exclusively while running. It's also efficient, offering performance similar to directly using CLUSTER.

The following options are available:

- Online CLUSTER
- Table Ordering by specific columns
- Online VACUUM FULL
- Rebuild or relocate only the indexes of a table

Requirements

- The utility can only be used by superusers.
- The target table must have either a PRIMARY KEY or at least one UNIQUE index on a NOT NULL column.
- Performing a full-table repack requires free disk space approximately double the size of the target table(s) and its indexes.

How does pg_repack work?

During a full-table repack operation using pg_repack, the following steps are undertaken:

1. **Log Table Creation**: Initially, pg_repack creates a log table to record any new changes made to the original table. This ensures tracking of INSERTs, UPDATEs, and DELETEs.
2. **Trigger Addition**: Next, a trigger is created onto the original table, enabling the logging of INSERTs, UPDATEs, and DELETEs into the designated log table. To accomplish this, pg_repack secures an access-exclusive lock on the original table to create and enable triggers.

3. **Shadow Table Creation**: Subsequently, a new shadow table is created to copy the initial data from the source table.
4. **Index Building**: Indexes are built concurrently on the shadow table.
5. **Application of Changes**: All modifications in the log table are applied to the shadow table.
6. **Table Swap**: The shadow table is switched with the original one. During this transition, the original table is locked in an access-exclusive mode to prevent any interference. This swap entails updating references in pg_class which also takes care of toast tables and associated indexes.
7. **Original Table Deletion**: Once the swap is completed, the original table is dropped from the database.

Throughout most of the procedure, pg_repack only uses ACCESS SHARE lock on the original table to facilitate concurrent data operations such as INSERTs, UPDATEs, and DELETEs. Exclusive locks are temporarily held during the initial setup (steps 1 and 2) and the final swap-and-drop phase (steps 6 and 7).

Important Options

–jobs=num_jobs

Specify the desired number of additional connections to parallelize the index rebuilding process for each table during a full-table repack. This feature is not compatible with the –index or –only-indexes options.

-n, –no-order

Execute an online VACUUM FULL operation. Since version 1.2, this behavior is the default for non-clustered tables.

-T secs –wait-timeout=secs

Determines the duration pg_repack will wait to obtain an exclusive lock required for the reorganization's conclusion. If the lock exceeds the specified time, pg_repack will forcibly terminate conflicting queries. The default timeout is set to 60 seconds.

-D, –no-kill-backend

Prevents the termination of other backends when a **–wait-timeout** occurs.

-k, –no-superuser-check

Bypass the superuser validation within the client. This setting is beneficial for running pg_repack on platforms that permit its execution by non-superusers, such as AWS or Azure PostgreSQL Dbaas offerings. More information: https://github.com/reorg/pg_repack/issues/223

Restrictions

pg_repack comes with the following restrictions.

- pg_repack cannot reorganize temp tables.
- pg_repack cannot cluster tables by GiST indexes.
- You will not be able to perform DDL commands of the target table(s) **except** VACUUM or ANALYZE while pg_repack is working. pg_repack will hold an ACCESS SHARE lock on the target table during a full-table repack, to enforce this restriction.
- If there is other pg_repack process in progress, it will throw out the following error and exit:

```
ERROR: Another pg_repack command may be running on the table. Please try again later.Copy to Clipboard
```

Conclusion

pg_repack is a valuable extension for PostgreSQL, widely used and appreciated for its effectiveness but it's important to avoid excessive usage. Excessive data movement between the original and temporary tables can lead to high WAL generation, which may have unintended consequences. For optimal results, it's advisable to schedule pg_repack during low activity periods to minimize disruptions. In critical applications where downtime is unacceptable, it's best to refrain from using pg_repack, especially during peak usage times.

An alternative approach is to implement aggressive vacuuming by fine-tuning the autovacuum configuration. You can find more detailed information in the referenced blog post: Vacuum in PostgreSQL.