Routine Database Maintenance Tasks

- Routine Vacuuming
  - Vacuuming Basics
  - Recovering Disk Space
  - Updating Planner Statistics
  - Updating the Visibility Map
  - Preventing Transaction ID Wraparound Failures
    - Multixacts and Wraparound
  - The Autovacuum Daemon
    - Autovacuum Prioritization
  - Parallel Vacuum
- Routine Reindexing
- Log File Maintenance

maintenance

routine maintenance

PostgreSQL, like any database software, requires that certain tasks be performed regularly to achieve optimum performance. The tasks discussed here are *required*, but they are repetitive in nature and can easily be automated using standard tools such as cron scripts or Windows' Task Scheduler. It is the database administrator's responsibility to set up appropriate scripts, and to check that they execute successfully.

One obvious maintenance task is the creation of backup copies of the data on a regular schedule. Without a recent backup, you have no chance of recovery after a catastrophe (disk failure, fire, mistakenly dropping a critical table, etc.). The backup and recovery mechanisms available in PostgreSQL are discussed at length in ???.

The other main category of maintenance task is periodic “vacuuming” of the database. This activity is discussed in Routine Vacuuming. Closely related to this is updating the statistics that will be used

by the query planner, as discussed in Updating Planner Statistics.

Another task that might need periodic attention is log file management. This is discussed in Log File Maintenance.

check_postgres is available for monitoring database health and reporting unusual conditions. check_postgres integrates with Nagios and MRTG, but can be run standalone too.

PostgreSQL is low-maintenance compared to some other database management systems. Nonetheless, appropriate attention to these tasks will go far towards ensuring a pleasant and productive experience with the system.

Routine Vacuuming

vacuum

PostgreSQL databases require periodic maintenance known as vacuuming. For many installations, it is sufficient to let vacuuming be performed by the autovacuum daemon, which is described in The Autovacuum Daemon. You might need to adjust the autovacuuming parameters described there to obtain best results for your situation. Some database administrators will want to supplement or replace the daemon's activities with manually-managed `VACUUM` commands, which typically are executed according to a schedule by cron or Task Scheduler scripts. To set up manually-managed vacuuming properly, it is essential to understand the issues discussed in the next few subsections. Administrators who rely on autovacuuming may still wish to skim this material to help them understand and adjust autovacuuming.

Vacuuming Basics

PostgreSQL's `VACUUM` command has to process each table on a regular basis for several reasons:

1. To recover or reuse disk space occupied by updated or deleted rows.
2. To update data statistics used by the PostgreSQL query planner.
3. To update the visibility map, which speeds up index-only scans.
4. To protect against loss of very old data due to transaction ID wraparound or multixact ID wraparound.

Each of these reasons dictates performing `VACUUM` operations of varying frequency and scope, as explained in the following subsections.

There are two variants of `VACUUM`: standard `VACUUM` and `VACUUM FULL`. `VACUUM FULL` can reclaim more disk space but runs much more slowly. Also, the standard form of `VACUUM` can run in parallel with

production database operations. (Commands such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` will continue to function normally, though you will not be able to modify the definition of a table with commands such as `ALTER TABLE` while it is being vacuumed.) `VACUUM FULL` requires an `ACCESS EXCLUSIVE` lock on the table it is working on, and therefore cannot be done in parallel with other use of the table. Generally, therefore, administrators should strive to use standard `VACUUM` and avoid `VACUUM FULL`.

`VACUUM` creates a substantial amount of I/O traffic, which can cause poor performance for other active sessions. There are configuration parameters that can be adjusted to reduce the performance impact of background vacuuming — see ???.

Recovering Disk Space

disk space

In PostgreSQL, an `UPDATE` or `DELETE` of a row does not immediately remove the old version of the row. This approach is necessary to gain the benefits of multiversion concurrency control (MVCC, see ???): the row version must not be deleted while it is still potentially visible to other transactions. But eventually, an outdated or deleted row version is no longer of interest to any transaction. The space it occupies must then be reclaimed for reuse by new rows, to avoid unbounded growth of disk space requirements. This is done by running `VACUUM`.

The standard form of `VACUUM` removes dead row versions in tables and indexes and marks the space available for future reuse. However, it will not return the space to the operating system, except in the special case where one or more pages at the end of a table become entirely free and an exclusive table lock can be easily obtained. In contrast, table-rewriting commands such as `VACUUM FULL`, `CLUSTER`, and `REPACK` actively compact tables by writing a complete new version of the table file without the dead space left by old row versions. This minimizes the size of the table, but can take a long time. It also requires extra disk space for the new copy of the table, until the operation completes.

The usual goal of routine vacuuming is to do standard `VACUUM`s often enough to avoid needing `VACUUM FULL`. The autovacuum daemon attempts to work this way, and in fact will never issue `VACUUM FULL`. In this approach, the idea is not to keep tables at their minimum size, but to maintain steady-state usage of disk space: each table occupies space equivalent to its minimum size plus however much space gets used up between vacuum runs. Although table-rewriting compaction operations such as `VACUUM FULL`, `CLUSTER`, and `REPACK` can be used to shrink a table back to its minimum size and return the disk space to the operating system, there is not much point in this if the table will just grow again in the future. Thus, moderately-frequent standard `VACUUM` runs are a better approach than infrequent use of such operations for maintaining heavily-updated tables.

Some administrators prefer to schedule vacuuming themselves, for example doing all the work at night when load is low. The difficulty with doing vacuuming according to a fixed schedule is that if a table has an unexpected spike in update activity, it may get bloated to the point that a table-

rewriting compaction operation such as `VACUUM FULL` or `REPACK` may be needed to reclaim space. Using the autovacuum daemon alleviates this problem, since the daemon schedules vacuuming dynamically in response to update activity. It is unwise to disable the daemon completely unless you have an extremely predictable workload. One possible compromise is to set the daemon's parameters so that it will only react to unusually heavy update activity, thus keeping things from getting out of hand, while scheduled `VACUUM`s are expected to do the bulk of the work when the load is typical.

For those not using autovacuum, a typical approach is to schedule a database-wide `VACUUM` once a day during a low-usage period, supplemented by more frequent vacuuming of heavily-updated tables as necessary. (Some installations with extremely high update rates vacuum their busiest tables as often as once every few minutes.) If you have multiple databases in a cluster, don't forget to `VACUUM` each one; the program ??? might be helpful.

> Plain `VACUUM` may not be satisfactory when a table contains large numbers of dead row versions as a result of massive update or delete activity. If you have such a table and you need to reclaim the excess disk space it occupies, you will need to use `VACUUM FULL`, `CLUSTER`, `REPACK`, or one of the table-rewriting variants of `ALTER TABLE`. These commands rewrite an entire new copy of the table and build new indexes for it. All these options require an `ACCESS EXCLUSIVE` lock, except that `REPACK` can be run with `CONCURRENTLY`, in which case the `ACCESS EXCLUSIVE` lock is held only while swapping the table and index files. Note that they also temporarily use extra disk space approximately equal to the size of the table, since the old copies of the table and indexes can't be released until the new ones are complete.

> If you have a table whose entire contents are deleted on a periodic basis, consider doing it with `TRUNCATE` rather than using `DELETE` followed by `VACUUM`. `TRUNCATE` removes the entire content of the table immediately, without requiring a subsequent `VACUUM` or `VACUUM FULL` to reclaim the now-unused disk space. The disadvantage is that strict MVCC semantics are violated.

Updating Planner Statistics

statistics

of the planner

ANALYZE

The PostgreSQL query planner relies on statistical information about the contents of tables in order to generate good plans for queries. These statistics are gathered by the `ANALYZE` command, which can be invoked by itself or as an optional step in `VACUUM`. It is important to have reasonably

accurate statistics, otherwise poor choices of plans might degrade database performance.

The autovacuum daemon, if enabled, will automatically issue `ANALYZE` commands whenever the content of a table has changed sufficiently. However, administrators might prefer to rely on manually-scheduled `ANALYZE` operations, particularly if it is known that update activity on a table will not affect the statistics of “interesting” columns. The daemon schedules `ANALYZE` strictly as a function of the number of rows inserted or updated; it has no knowledge of whether that will lead to meaningful statistical changes.

Tuples changed in partitions and inheritance children do not trigger analyze on the parent table. If the parent table is empty or rarely changed, it may never be processed by autovacuum, and the statistics for the inheritance tree as a whole won't be collected. It is necessary to run `ANALYZE` on the parent table manually in order to keep the statistics up to date.

As with vacuuming for space recovery, frequent updates of statistics are more useful for heavily-updated tables than for seldom-updated ones. But even for a heavily-updated table, there might be no need for statistics updates if the statistical distribution of the data is not changing much. A simple rule of thumb is to think about how much the minimum and maximum values of the columns in the table change. For example, a `timestamp` column that contains the time of row update will have a constantly-increasing maximum value as rows are added and updated; such a column will probably need more frequent statistics updates than, say, a column containing URLs for pages accessed on a website. The URL column might receive changes just as often, but the statistical distribution of its values probably changes relatively slowly.

It is possible to run `ANALYZE` on specific tables and even just specific columns of a table, so the flexibility exists to update some statistics more frequently than others if your application requires it. In practice, however, it is usually best to just analyze the entire database, because it is a fast operation. `ANALYZE` uses a statistically random sampling of the rows of a table rather than reading every single row.

> Although per-column tweaking of `ANALYZE` frequency might not be very productive, you might find it worthwhile to do per-column adjustment of the level of detail of the statistics collected by `ANALYZE`. Columns that are heavily used in `WHERE` clauses and have highly irregular data distributions might require a finer-grain data histogram than other columns.
>
> See `ALTER TABLE SET STATISTICS`, or change the database-wide default using the ??? configuration parameter.
>
> Also, by default there is limited information available about the selectivity of functions. However, if you create a statistics object or an expression index that uses a function call, useful statistics will be gathered about the function, which can greatly improve query plans that use the expression index.

> The autovacuum daemon does not issue `ANALYZE` commands for foreign tables, since it has no means of determining how often that might be useful. If your queries require statistics on foreign tables for proper planning, it's a good idea to run manually-managed `ANALYZE` commands on those tables on a suitable schedule.

> The autovacuum daemon does not issue `ANALYZE` commands for partitioned tables. Inheritance parents will only be analyzed if the parent itself is changed - changes to child tables do not trigger autoanalyze on the parent table. If your queries require statistics on parent tables for proper planning, it is necessary to periodically run a manual `ANALYZE` on those tables to keep the statistics up to date.

Updating the Visibility Map

Vacuum maintains a visibility map for each table to keep track of which pages contain only tuples that are known to be visible to all active transactions (and all future transactions, until the page is again modified). This has two purposes. First, vacuum itself can skip such pages on the next run, since there is nothing to clean up.

Second, it allows PostgreSQL to answer some queries using only the index, without reference to the underlying table. Since PostgreSQL indexes don't contain tuple visibility information, a normal index scan fetches the heap tuple for each matching index entry, to check whether it should be seen by the current transaction. An index-only scan, on the other hand, checks the visibility map first. If it's known that all tuples on the page are visible, the heap fetch can be skipped. This is most useful on large data sets where the visibility map can prevent disk accesses. The visibility map is vastly smaller than the heap, so it can easily be cached even when the heap is very large.

Preventing Transaction ID Wraparound Failures

transaction ID

wraparound

wraparound

of transaction IDs

PostgreSQL's MVCC transaction semantics depend on being able to compare transaction ID (XID) numbers: a row version with an insertion XID greater than the current transaction's XID is “in the

future” and should not be visible to the current transaction. But since transaction IDs have limited size (32 bits) a cluster that runs for a long time (more than 4 billion transactions) would suffer transaction ID wraparound: the XID counter wraps around to zero, and all of a sudden transactions that were in the past appear to be in the future — which means their output become invisible. In short, catastrophic data loss. (Actually the data is still there, but that's cold comfort if you cannot get at it.) To avoid this, it is necessary to vacuum every table in every database at least once every two billion transactions.

The reason that periodic vacuuming solves the problem is that `VACUUM` will mark rows as *frozen*, indicating that they were inserted by a transaction that committed sufficiently far in the past that the effects of the inserting transaction are certain to be visible to all current and future transactions. Normal XIDs are compared using modulo-2^{32} arithmetic. This means that for every normal XID, there are two billion XIDs that are “older” and two billion that are “newer”; another way to say it is that the normal XID space is circular with no endpoint. Therefore, once a row version has been created with a particular normal XID, the row version will appear to be “in the past” for the next two billion transactions, no matter which normal XID we are talking about. If the row version still exists after more than two billion transactions, it will suddenly appear to be in the future. To prevent this, PostgreSQL reserves a special XID, `FrozenTransactionId`, which does not follow the normal XID comparison rules and is always considered older than every normal XID. Frozen row versions are treated as if the inserting XID were `FrozenTransactionId`, so that they will appear to be “in the past” to all normal transactions regardless of wraparound issues, and so such row versions will be valid until deleted, no matter how long that is.

> In PostgreSQL versions before 9.4, freezing was implemented by actually replacing a row's insertion XID with `FrozenTransactionId`, which was visible in the row's xmin system column. Newer versions just set a flag bit, preserving the row's original xmin for possible forensic use. However, rows with xmin equal to `FrozenTransactionId` (2) may still be found in databases pg_upgrade'd from pre-9.4 versions.
>
> Also, system catalogs may contain rows with xmin equal to `BootstrapTransactionId` (1), indicating that they were inserted during the first phase of initdb. Like `FrozenTransactionId`, this special XID is treated as older than every normal XID.

??? controls how old an XID value has to be before rows bearing that XID will be frozen. Increasing this setting may avoid unnecessary work if the rows that would otherwise be frozen will soon be modified again, but decreasing this setting increases the number of transactions that can elapse before the table must be vacuumed again.

`VACUUM` uses the visibility map to determine which pages of a table must be scanned. Normally, it will skip pages that don't have any dead row versions even if those pages might still have row versions with old XID values. Therefore, normal `VACUUM`s won't always freeze every old row version in the table. When that happens, `VACUUM` will eventually need to perform an aggressive vacuum, which will freeze all eligible unfrozen XID and MXID values, including those from all-visible but not all-frozen pages.

If a table is building up a backlog of all-visible but not all-frozen pages, a normal vacuum may choose to scan skippable pages in an effort to freeze them. Doing so decreases the number of pages the next aggressive vacuum must scan. These are referred to as eagerly scanned pages.

Eager scanning can be tuned to attempt to freeze more all-visible pages by increasing ???. Even if eager scanning has kept the number of all-visible but not all-frozen pages to a minimum, most tables still require periodic aggressive vacuuming. However, any pages successfully eager frozen may be skipped during an aggressive vacuum, so eager freezing may minimize the overhead of aggressive vacuums.

??? controls when a table is aggressively vacuumed. All all-visible but not all-frozen pages are scanned if the number of transactions that have passed since the last such scan is greater than `vacuum_freeze_table_age` minus `vacuum_freeze_min_age`. Setting `vacuum_freeze_table_age` to 0 forces `VACUUM` to always use its aggressive strategy.

The maximum time that a table can go unvacuumed is two billion transactions minus the `vacuum_freeze_min_age` value at the time of the last aggressive vacuum. If it were to go unvacuumed for longer than that, data loss could result. To ensure that this does not happen, autovacuum is invoked on any table that might contain unfrozen rows with XIDs older than the age specified by the configuration parameter ???. (This will happen even if autovacuum is disabled.)

This implies that if a table is not otherwise vacuumed, autovacuum will be invoked on it approximately once every `autovacuum_freeze_max_age` minus `vacuum_freeze_min_age` transactions. For tables that are regularly vacuumed for space reclamation purposes, this is of little importance. However, for static tables (including tables that receive inserts, but no updates or deletes), there is no need to vacuum for space reclamation, so it can be useful to try to maximize the interval between forced autovacuums on very large static tables. Obviously one can do this either by increasing `autovacuum_freeze_max_age` or decreasing `vacuum_freeze_min_age`.

The effective maximum for `vacuum_freeze_table_age` is 0.95 * `autovacuum_freeze_max_age`; a setting higher than that will be capped to the maximum. A value higher than `autovacuum_freeze_max_age` wouldn't make sense because an anti-wraparound autovacuum would be triggered at that point anyway, and the 0.95 multiplier leaves some breathing room to run a manual `VACUUM` before that happens. As a rule of thumb, `vacuum_freeze_table_age` should be set to a value somewhat below `autovacuum_freeze_max_age`, leaving enough gap so that a regularly scheduled `VACUUM` or an autovacuum triggered by normal delete and update activity is run in that window. Setting it too close could lead to anti-wraparound autovacuums, even though the table was recently vacuumed to reclaim space, whereas lower values lead to more frequent aggressive vacuuming.

The sole disadvantage of increasing `autovacuum_freeze_max_age` (and `vacuum_freeze_table_age` along with it) is that the `pg_xact` and `pg_commit_ts` subdirectories of the database cluster will take more space, because it must store the commit status and (if `track_commit_timestamp` is enabled) timestamp of all transactions back to the `autovacuum_freeze_max_age` horizon. The commit status uses two bits per transaction, so if `autovacuum_freeze_max_age` is set to its maximum allowed value of two billion, `pg_xact` can be expected to grow to about half a gigabyte and `pg_commit_ts` to about 20GB. If this is trivial compared to your total database size, setting `autovacuum_freeze_max_age` to its

maximum allowed value is recommended. Otherwise, set it depending on what you are willing to allow for `pg_xact` and `pg_commit_ts` storage. (The default, 200 million transactions, translates to about 50MB of `pg_xact` storage and about 2GB of `pg_commit_ts` storage.)

One disadvantage of decreasing `vacuum_freeze_min_age` is that it might cause `VACUUM` to do useless work: freezing a row version is a waste of time if the row is modified soon thereafter (causing it to acquire a new XID). So the setting should be large enough that rows are not frozen until they are unlikely to change any more.

To track the age of the oldest unfrozen XIDs in a database, `VACUUM` stores XID statistics in the system tables pg_class and pg_database. In particular, the relfrozenxid column of a table's pg_class row contains the oldest remaining unfrozen XID at the end of the most recent `VACUUM` that successfully advanced relfrozenxid (typically the most recent aggressive VACUUM). Similarly, the datfrozenxid column of a database's pg_database row is a lower bound on the unfrozen XIDs appearing in that database — it is just the minimum of the per-table relfrozenxid values within the database. A convenient way to examine this information is to execute queries such as:

```
SELECT c.oid::regclass AS table_name,
       greatest(age(c.relfrozenxid), age(t.relfrozenxid)) AS age
FROM pg_class c
LEFT JOIN pg_class t ON c.reltoastrelid = t.oid
WHERE c.relkind IN ('r', 'm');

SELECT datname, age(datfrozenxid) FROM pg_database;
```

The `age` column measures the number of transactions from the cutoff XID to the current transaction's XID.

> When the `VACUUM` command's `VERBOSE` parameter is specified, `VACUUM` prints various statistics about the table. This includes information about how relfrozenxid and relminmxid advanced, and the number of newly frozen pages. The same details appear in the server log when autovacuum logging (controlled by ???) reports on a `VACUUM` operation executed by autovacuum.

While `VACUUM` scans mostly pages that have been modified since the last vacuum, it may also eagerly scan some all-visible but not all-frozen pages in an attempt to freeze them, but the relfrozenxid will only be advanced when every page of the table that might contain unfrozen XIDs is scanned. This happens when relfrozenxid is more than `vacuum_freeze_table_age` transactions old, when `VACUUM`'s `FREEZE` option is used, or when all pages that are not already all-frozen happen to require vacuuming to remove dead row versions. When `VACUUM` scans every page in the table that is not already all-frozen, it should set `age(relfrozenxid)` to a value just a little more than the `vacuum_freeze_min_age` setting that was used (more by the number of transactions started since the `VACUUM` started). `VACUUM` will set relfrozenxid to the oldest XID that remains in the table, so it's

possible that the final value will be much more recent than strictly required. If no relfrozenxid-advancing `VACUUM` is issued on the table until `autovacuum_freeze_max_age` is reached, an autovacuum will soon be forced for the table.

If for some reason autovacuum fails to clear old XIDs from a table, the system will begin to emit warning messages like this when the database's oldest XIDs reach one hundred million transactions from the wraparound point:

```
WARNING:  database "mydb" must be vacuumed within 99985967 transactions
DETAIL:  Approximately 4.66% of transaction ID space remains before wraparound.
HINT:  To avoid transaction ID assignment failures, execute a database-wide VACUUM in that
database.
```

(A manual `VACUUM` should fix the problem, as suggested by the hint; but note that the `VACUUM` should be performed by a superuser, else it will fail to process system catalogs, which prevent it from being able to advance the database's datfrozenxid.) If these warnings are ignored, the system will refuse to assign new XIDs once there are fewer than three million transactions left until wraparound:

```
ERROR:  database is not accepting commands that assign new transaction IDs to avoid wraparound
data loss in database "mydb"
HINT:  Execute a database-wide VACUUM in that database.
```

In this condition any transactions already in progress can continue, but only read-only transactions can be started. Operations that modify database records or truncate relations will fail. The `VACUUM` command can still be run normally. Note that, contrary to what was sometimes recommended in earlier releases, it is not necessary or desirable to stop the postmaster or enter single user-mode in order to restore normal operation. Instead, follow these steps:

1. Resolve old prepared transactions. You can find these by checking pg_prepared_xacts for rows where `age(transactionid)` is large. Such transactions should be committed or rolled back.
2. End long-running open transactions. You can find these by checking pg_stat_activity for rows where `age(backend_xid)` or `age(backend_xmin)` is large. Such transactions should be committed or rolled back, or the session can be terminated using `pg_terminate_backend`.
3. Drop any old replication slots. Use pg_replication_slots to find slots where `age(xmin)` or `age(catalog_xmin)` is large. In many cases, such slots were created for replication to servers that no longer exist, or that have been down for a long time. If you drop a slot for a server that still exists and might still try to connect to that slot, that replica may need to be rebuilt.
4. Execute `VACUUM` in the target database. A database-wide `VACUUM` is simplest; to reduce the time required, it is also possible to issue manual `VACUUM` commands on the tables where relminxid is oldest. Do not use `VACUUM FULL` in this scenario, because it requires an XID and will therefore fail, except in super-user mode, where it will instead consume an XID

and thus increase the risk of transaction ID wraparound. Do not use `VACUUM FREEZE` either, because it will do more than the minimum amount of work required to restore normal operation.

5. Once normal operation is restored, ensure that autovacuum is properly configured in the target database in order to avoid future problems.

> In earlier versions, it was sometimes necessary to stop the postmaster and `VACUUM` the database in a single-user mode. In typical scenarios, this is no longer necessary, and should be avoided whenever possible, since it involves taking the system down. It is also riskier, since it disables transaction ID wraparound safeguards that are designed to prevent data loss. The only reason to use single-user mode in this scenario is if you wish to `TRUNCATE` or `DROP` unneeded tables to avoid needing to `VACUUM` them. The three-million-transaction safety margin exists to let the administrator do this. See the ??? reference page for details about using single-user mode.

Multixacts and Wraparound

MultiXactId

wraparound

of multixact IDs

Multixact IDs (MXIDs) are used to support row locking by multiple transactions. Since there is only limited space in a tuple header to store lock information, that information is encoded as a “multiple transaction ID”, or multixact ID for short, whenever there is more than one transaction concurrently locking a row. Information about which transaction IDs are included in any particular multixact ID is stored separately in the `pg_multixact` subdirectory, and only the multixact ID appears in the xmax field in the tuple header. Like transaction IDs, multixact IDs are implemented as a 32-bit counter and corresponding storage, all of which requires careful aging management, storage cleanup, and wraparound handling. There is a separate storage area which holds the list of members in each multixact, which cannot wrap around but must also be managed to keep disk usage under control.

The system function `pg_get_multixact_members()` described in ??? can be used to examine the transaction IDs associated with a multixact ID.

Whenever `VACUUM` scans any part of a table, it will replace any multixact ID it encounters which is older than ??? by a different value, which can be the zero value, a single transaction ID, or a newer multixact ID. For each table, pg_class.relminmxid stores the oldest possible multixact ID still appearing in any tuple of that table. If this value is older than ???, an aggressive vacuum is forced. As discussed in the previous section, an aggressive vacuum means that only those pages which are known to be all-frozen will be skipped. `mxid_age()` can be used on pg_class.relminmxid to find its age.

Aggressive `VACUUM`s, regardless of what causes them, are *guaranteed* to be able to advance the table's relminmxid. Eventually, as all tables in all databases are scanned and their oldest multixact values are advanced, on-disk storage for older multixacts can be removed.

As a safety device, an aggressive vacuum scan will occur for any table whose multixact-age is greater than ???. The aggressive scans will occur even if autovacuum is nominally disabled. Also, if the number of multixact member entries created exceeds approximately 2 billion entries (occupying roughly 10GB in the `pg_multixact/members` directory), aggressive vacuum scans will occur more often for all tables, starting with those that have the oldest multixact-age.

The `pg_get_multixact_stats()` function described in ??? provides a way to monitor multixact allocation and usage patterns in real time, for example:

```
=# SELECT *, pg_size_pretty(members_size) members_size_pretty
     FROM pg_catalog.pg_get_multixact_stats();
 num_mxids | num_members | members_size | oldest_multixact | members_size_pretty
-----------+-------------+--------------+------------------+--------------------
 311740299 |  2785241176 |  13926205880 |                2 | 13 GB
(1 row)
```

This output shows a system with significant multixact activity: about 312 million multixact IDs and about 2.8 billion member entries consuming 13 GB of storage space. A spike in `num_mxids` might indicate multiple sessions running `UPDATE` statements with foreign key checks, concurrent `SELECT FOR SHARE` operations, or frequent use of savepoints causing lock contention. If `oldest_multixact` value remains unchanged while `num_members` grows, it could indicate that long-running transactions are preventing cleanup, or autovacuum is not keeping up with the workload.

Similar to the XID case, if autovacuum fails to clear old MXIDs from a table, the system will begin to emit warning messages when the database's oldest MXIDs reach one hundred million transactions from the wraparound point. And, just as in the XID case, if these warnings are ignored, the system will refuse to generate new MXIDs once there are fewer than three million left until wraparound.

Normal operation when MXIDs are exhausted can be restored in much the same way as when XIDs are exhausted. Follow the same steps in the previous section, but with the following differences:

1. Running transactions and prepared transactions can be ignored if there is no chance that they might appear in a multixact.
2. Unlike transaction ID wraparound, replication slots do not directly hold back multixact cleanup. Dropping stale replication slots is therefore not usually relevant to resolving multixact ID wraparound problems.
3. MXID information is not directly visible in system views such as `pg_stat_activity`; however, looking for old XIDs is still a good way of determining which transactions are causing MXID wraparound problems.

4. XID exhaustion will block all write transactions, but MXID exhaustion will only block a subset of write transactions, specifically those that involve row locks that require an MXID.

The Autovacuum Daemon

autovacuum

general information

PostgreSQL has an optional but highly recommended feature called autovacuum, whose purpose is to automate the execution of `VACUUM` and `ANALYZE` commands. When enabled, autovacuum checks for tables that have had a large number of inserted, updated or deleted tuples. These checks use the statistics collection facility; therefore, autovacuum cannot be used unless ??? is set to `true`. In the default configuration, autovacuuming is enabled and the related configuration parameters are appropriately set.

The “autovacuum daemon” actually consists of multiple processes. There is a persistent daemon process, called the autovacuum launcher, which is in charge of starting autovacuum worker processes for all databases. The launcher will distribute the work across time, attempting to start one worker within each database every ??? seconds. (Therefore, if the installation has <N> databases, a new worker will be launched every `autovacuum_naptime`/<N> seconds.) A maximum of ??? worker processes are allowed to run at the same time. If there are more than `autovacuum_max_workers` databases to be processed, the next database will be processed as soon as the first worker finishes. Each worker process will check each table within its database and execute `VACUUM` and/or `ANALYZE` as needed. ??? and ??? can be set to monitor autovacuum workers' activity.

If several large tables all become eligible for vacuuming in a short amount of time, all autovacuum workers might become occupied with vacuuming those tables for a long period. This would result in other tables and databases not being vacuumed until a worker becomes available. There is no limit on how many workers might be in a single database, but workers do try to avoid repeating work that has already been done by other workers. Note that the number of running workers does not count towards ??? or ??? limits.

Tables whose relfrozenxid value is more than ??? transactions old are always vacuumed (this also applies to those tables whose freeze max age has been modified via storage parameters; see below). Otherwise, if the number of tuples obsoleted since the last `VACUUM` exceeds the “vacuum threshold”, the table is vacuumed. The vacuum threshold is defined as:

```
vacuum threshold = Minimum(vacuum max threshold, vacuum base threshold + vacuum scale factor *
number of tuples)
```

where the vacuum max threshold is ???, the vacuum base threshold is ???, the vacuum scale factor is ???, and the number of tuples is pg_class.reltuples.

The table is also vacuumed if the number of tuples inserted since the last vacuum has exceeded the defined insert threshold, which is defined as:

```
vacuum insert threshold = vacuum base insert threshold + vacuum insert scale factor * number of tuples * percent of table not frozen
```

where the vacuum insert base threshold is ???, the vacuum insert scale factor is ???, the number of tuples is pg_class.reltuples, and the percent of the table not frozen is `1 - pg_class.relallfrozen / pg_class.relpages`. Such vacuums may allow portions of the table to be marked as all visible and also allow tuples to be frozen, which can reduce the work required in subsequent vacuums. For tables which receive `INSERT` operations but no or almost no `UPDATE`/`DELETE` operations, it may be beneficial to lower the table's ??? as this may allow tuples to be frozen by earlier vacuums. The number of obsolete tuples and the number of inserted tuples are obtained from the cumulative statistics system; it is an eventually-consistent count updated by each `UPDATE`, `DELETE` and `INSERT` operation. If the relfrozenxid value of the table is more than `vacuum_freeze_table_age` transactions old, an aggressive vacuum is performed to freeze old tuples and advance relfrozenxid.

For analyze, a similar condition is used: the threshold, defined as:

```
analyze threshold = analyze base threshold + analyze scale factor * number of tuples
```

is compared to the total number of tuples inserted, updated, or deleted since the last `ANALYZE`.

Partitioned tables do not directly store tuples and consequently are not processed by autovacuum. (Autovacuum does process table partitions just like other tables.) Unfortunately, this means that autovacuum does not run `ANALYZE` on partitioned tables, and this can cause suboptimal plans for queries that reference partitioned table statistics. You can work around this problem by manually running `ANALYZE` on partitioned tables when they are first populated, and again whenever the distribution of data in their partitions changes significantly.

Temporary tables cannot be accessed by autovacuum. Therefore, appropriate vacuum and analyze operations should be performed via session SQL commands.

The default thresholds and scale factors are taken from `postgresql.conf`, but it is possible to override them (and many other autovacuum control parameters) on a per-table basis; see ??? for more information. If a setting has been changed via a table's storage parameters, that value is used when processing that table; otherwise the global settings are used. See ??? for more details on the global settings.

When multiple workers are running, the autovacuum cost delay parameters (see ???) are "balanced" among all the running workers, so that the total I/O impact on the system is the same regardless of the number of workers actually running. However, any workers processing tables whose per-table `autovacuum_vacuum_cost_delay` or `autovacuum_vacuum_cost_limit` storage parameters have been set are not considered in the balancing algorithm. Parallel workers launched for Parallel Vacuum are using the same cost delay parameters as the leader worker. If any of these parameters are changed in the leader worker, it will propagate the new parameter values to all of its parallel workers.

Autovacuum workers generally don't block other commands. If a process attempts to acquire a lock that conflicts with the `SHARE UPDATE EXCLUSIVE` lock held by autovacuum, lock acquisition will interrupt the autovacuum. For conflicting lock modes, see ???. However, if the autovacuum is running to prevent transaction ID wraparound (i.e., the autovacuum query name in the pg_stat_activity view ends with `(to prevent wraparound)` or the started_by column in the pg_stat_progress_vacuum view shows `autovacuum_wraparound` value), the autovacuum is not automatically interrupted.

> Regularly running commands that acquire locks conflicting with a `SHARE UPDATE EXCLUSIVE` lock (e.g., ANALYZE) can effectively prevent autovacuums from ever completing.

Autovacuum Prioritization

Autovacuum decides what to process in two steps: first it chooses a database, then it chooses the tables within that database. The autovacuum launcher process prioritizes databases at risk of transaction ID or multixact ID wraparound, else it chooses the database processed least recently. As an exception, it skips databases with no connections or no activity since the last statistics reset, unless at risk of wraparound.

Within a database, the autovacuum worker process builds a list of tables that require vacuum or analyze and sorts them using a scoring system. It scores each table by taking the maximum value of several component scores representing various criteria important to vacuum or analyze. Those components are as follows:

- The *transaction ID* component measures the age in transactions of the table's pg_class.relfrozenxid field as compared to ???. Furthermore, this component increases greatly once the age surpasses ???. The final value for this component can be adjusted via ???. Note that increasing this parameter's value also lowers the age at which this component begins scaling aggressively, i.e., the scaling age is divided by its value if greater than `1.0`.
- The *multixact ID* component measures the age in multixacts of the table's pg_class.relminmxid field as compared to ???. Furthermore, this component increases

greatly once the age surpasses ??? or when the number of multixact member entries created exceeds approximately 2 billion entries (see Multixacts and Wraparound). The final value for this component can be adjusted via ???. Note that increasing this parameter's value also lowers the age at which this component begins scaling aggressively, i.e., the scaling age is divided by its value if greater than `1.0`.

- The *vacuum* component measures the number of updated or deleted tuples as compared to the threshold calculated with ???, ???, and ???. The final value for this component can be adjusted via ???.
- The *vacuum insert* component measures the number of inserted tuples as compared to the threshold calculated with ??? and ???. The final value for this component can be adjusted via ???.
- The *analyze* component measures the number of inserted, updated, or deleted tuples as compared to the threshold calculated with ??? and ???. The final value for this component can be adjusted via ???.

To revert to the prioritization strategy used before PostgreSQL 19 (i.e., the order the tables are listed in the `pg_class` system catalog), set all of the aforementioned "weight" parameters to `0.0`. Otherwise, these "weight" parameters are multiplied to their respective component scores. For example, raising ??? to `2.0` effectively doubles the *analyze* component score.

The `pg_stat_autovacuum_scores` view shows the current scores of all tables in the current database.

Parallel Vacuum

`VACUUM` can perform index vacuuming and index cleanup phases in parallel using background workers (for the details of each vacuum phase, please refer to ???). The degree of parallelism is determined by the number of indexes on the relation that support parallel vacuum. For manual `VACUUM`, this is limited by the `PARALLEL` option if specified, which is further capped by ???. For autovacuum, it is limited by the table's ??? if specified, which is further capped by ??? parameter. Please note that it is not guaranteed that the number of parallel workers that was calculated will be used during execution. It is possible for a vacuum to run with fewer workers than specified, or even with no workers at all.

An index can participate in parallel vacuum if and only if the size of the index is more than ???. Only one worker can be used per index. So parallel workers are launched only when there are at least `2` indexes in the table. Workers for vacuum are launched before the start of each phase and exit at the end of the phase. These behaviors might change in a future release.

Routine Reindexing

reindex

In some situations it is worthwhile to rebuild indexes periodically with the ??? command or a series of individual rebuilding steps.

B-tree index pages that have become completely empty are reclaimed for re-use. However, there is still a possibility of inefficient use of space: if all but a few index keys on a page have been deleted, the page remains allocated. Therefore, a usage pattern in which most, but not all, keys in each range are eventually deleted will see poor use of space. For such usage patterns, periodic reindexing is recommended.

The potential for bloat in non-B-tree indexes has not been well researched. It is a good idea to periodically monitor the index's physical size when using any non-B-tree index type.

Also, for B-tree indexes, a freshly-constructed index is slightly faster to access than one that has been updated many times because logically adjacent pages are usually also physically adjacent in a newly built index. (This consideration does not apply to non-B-tree indexes.) It might be worthwhile to reindex periodically just to improve access speed.

??? can be used safely and easily in all cases. This command requires an `ACCESS EXCLUSIVE` lock by default, hence it is often preferable to execute it with its `CONCURRENTLY` option, which requires only a `SHARE UPDATE EXCLUSIVE` lock.

Log File Maintenance

server log

log file maintenance

It is a good idea to save the database server's log output somewhere, rather than just discarding it via `/dev/null`. The log output is invaluable when diagnosing problems.

> The server log can contain sensitive information and needs to be protected, no matter how or where it is stored, or the destination to which it is routed. For example, some DDL statements might contain plaintext passwords or other authentication details. Logged statements at the `ERROR` level might show the SQL source code for applications and might also contain some parts of data rows. Recording data, events and related information is the intended function of this facility, so this is not a leakage or a bug. Please ensure the server logs are visible only to appropriately authorized people.

Log output tends to be voluminous (especially at higher debug levels) so you won't want to save it indefinitely. You need to *rotate* the log files so that new log files are started and old ones removed after a reasonable period of time.

If you simply direct the `stderr` of `postgres` into a file, you will have log output, but the only way to truncate the log file is to stop and restart the server. This might be acceptable if you are using PostgreSQL in a development environment, but few production servers would find this behavior acceptable.

A better approach is to send the server's `stderr` output to some type of log rotation program. There is a built-in log rotation facility, which you can use by setting the configuration parameter `logging_collector` to `true` in `postgresql.conf`. The control parameters for this program are described in ???. You can also use this approach to capture the log data in machine readable CSV (comma-separated values) format.

Alternatively, you might prefer to use an external log rotation program if you have one that you are already using with other server software. For example, the rotatelogs tool included in the Apache distribution can be used with PostgreSQL. One way to do this is to pipe the server's `stderr` output to the desired program. If you start the server with `pg_ctl`, then `stderr` is already redirected to `stdout`, so you just need a pipe command, for example:

```
pg_ctl start | rotatelogs /var/log/pgsql_log 86400
```

You can combine these approaches by setting up logrotate to collect log files produced by PostgreSQL built-in logging collector. In this case, the logging collector defines the names and location of the log files, while logrotate periodically archives these files. When initiating log rotation, logrotate must ensure that the application sends further output to the new file. This is commonly done with a `postrotate` script that sends a `SIGHUP` signal to the application, which then reopens the log file. In PostgreSQL, you can run `pg_ctl` with the `logrotate` option instead. When the server receives this command, the server either switches to a new log file or reopens the existing file, depending on the logging configuration (see ???).

> When using static log file names, the server might fail to reopen the log file if the max open file limit is reached or a file table overflow occurs. In this case, log messages are sent to the old log file until a successful log rotation. If logrotate is configured to compress the log file and delete it, the server may lose the messages logged in this time frame. To avoid this issue, you can configure the logging collector to dynamically assign log file names and use a `prerotate` script to ignore open log files.

Another production-grade approach to managing log output is to send it to syslog and let syslog deal with file rotation. To do this, set the configuration parameter `log_destination` to `syslog` (to log to syslog only) in `postgresql.conf`. Then you can send a `SIGHUP` signal to the syslog daemon whenever you want to force it to start writing a new log file. If you want to automate log rotation, the logrotate program can be configured to work with log files from syslog.

On many systems, however, syslog is not very reliable, particularly with large log messages; it might truncate or drop messages just when you need them the most. Also, on Linux, syslog will flush each message to disk, yielding poor performance. (You can use a “`-`” at the start of the file name in the syslog configuration file to disable syncing.)

Note that all the solutions described above take care of starting new log files at configurable intervals, but they do not handle deletion of old, no-longer-useful log files. You will probably want to set up a batch job to periodically delete old log files. Another possibility is to configure the rotation program so that old log files are overwritten cyclically.

pgBadger is an external project that does sophisticated log file analysis. check_postgres provides Nagios alerts when important messages appear in the log files, as well as detection of many other extraordinary conditions.

□□□ #2
□ □□□ □□□ 2026-10-31 14:06:47 UTC
□ □□□ □□□ 2026-10-31 14:06:48 UTC