24.5. Hot Standby

Hot Standby is the term used to describe the ability to connect to the server and run queries while the server is in archive recovery. This is useful for both log shipping replication and for restoring a backup to an exact state with great precision. The term Hot Standby also refers to the ability of the server to move from recovery through to normal running while users continue running queries and/or continue their connections.

Running queries in recovery is in many ways the same as normal running though there are a large number of usage and administrative points to note.

24.5.1. User's Overview

Users can connect to the database while the server is in recovery and perform read-only queries. Read-only access to catalogs and views will also occur as normal.

The data on the standby takes some time to arrive from the primary server so there will be a measurable delay between primary and standby. Queries executed on the standby will be correct as of the data that had been recovered at the start of the query (or start of first statement, in the case of Serializable transactions). Running the same query nearly simultaneously on both primary and standby might therefore return differing results. We say that data on the standby is eventually consistent with the primary.

When a connection is made in recovery, the parameter default_transaction_read_only will be forced to be true, whatever its setting in postgresql.conf. As a result, all transactions started during this time will be limited to read-only actions only. In all other ways, connected sessions will appear identical to sessions initiated during normal processing mode. There are no special commands required to initiate a connection at this time, so all interfaces will work normally without change.

Read-only here means "no writes to the permanent database tables". So there are no problems with queries that make use of temporary sort and work files will be used. Temporary tables cannot be created and therefore cannot be used at all in recovery mode.

The following actions are allowed

- Query access - SELECT, COPY TO including views and SELECT RULEs
- Cursor commands - DECLARE, FETCH, CLOSE,

- Parameters - SHOW, SET, RESET
- Transaction management commands
    - BEGIN, END, ABORT, START TRANSACTION
    - SAVEPOINT, RELEASE, ROLLBACK TO SAVEPOINT
    - EXCEPTION blocks and other internal subtransactions
- LOCK, with restrictions, see later
- Plans and resources - PREPARE, EXECUTE, DEALLOCATE, DISCARD
- Plugins and extensions - LOAD

These actions will produce error messages

- DML - Insert, Update, Delete, COPY FROM, Truncate which all write data. Any RULE which generates DML will throw error messages as a result. Note that there is no action possible that can result in a trigger being executed.
- DDL - Create, Drop, Alter, Comment (even for temporary tables because currently these cause writes to catalog tables)
- SELECT ... FOR SHARE | UPDATE which cause row locks to be written
- Transaction management commands that explicitly set non-read only state
    - BEGIN READ WRITE, START TRANSACTION READ WRITE
    - SET TRANSACTION READ WRITE, SET SESSION CHARACTERISTICS AS TRANSACTION READ WRITE
    - SET transaction_read_only = off; or SET default_transaction_read_only = off;
- Two-phase commit commands - PREPARE TRANSACTION, COMMIT PREPARED, ROLLBACK PREPARED because even read-only transactions need to write WAL in the prepare phase (the first phase of two phase commit).
- sequence update - nextval()
- LISTEN, UNLISTEN, NOTIFY since they currently write to system tables

Note that current behaviour of read only transactions when not in recovery is to allow the last two actions, so there is a small and subtle difference in behaviour between standby read-only transactions and read only transactions during normal running. It is possible that the restrictions on LISTEN, UNLISTEN, NOTIFY and temporary tables may be lifted in a future release, if their internal implementation is altered to make this possible.

If failover or switchover occurs the database will switch to normal processing mode. Sessions will remain connected while the server changes mode. Current

transactions will continue, though will remain read-only. After this, it will be possible to initiate read-write transactions, though users must *manually* reset their default_transaction_read_only setting first, if they want that behaviour.

Users will be able to tell whether their session is read-only by issuing SHOW default_transaction_read_only. In addition a set of functions Table 9-57 allow users to access information about Hot Standby. These allow you to write functions that are aware of the current state of the database. These can be used to monitor the progress of recovery, or to allow you to write complex programs that restore the database to particular states.

In recovery, transactions will not be permitted to take any lock higher other than AccessShareLock or AccessExclusiveLock. In addition, transactions may never assign a TransactionId and may never write WAL. The LOCK TABLE command by default applies an AccessExclusiveLock. Any LOCK TABLE command that runs on the standby and requests a specific lock type other than AccessShareLock will be rejected.

During recovery database changes are applied using full MVCC rules. In general this means that queries will not experience lock conflicts with writes, just like normal Postgres concurrency control (MVCC).

24.5.2. Handling query conflicts

There is some potential for conflict between standby queries and WAL redo from the primary node. The user is provided with a number of optional ways to handle these conflicts, though we must first understand the possible reasons behind a conflict.

- Access Exclusive Locks from primary node, including both explicit LOCK commands and various kinds of DDL action
- Early cleanup of data still visible to the current query's snapshot
- Dropping tablespaces on the primary while standby queries are using those tablespace for temporary work files (work_mem overflow)
- Dropping databases on the primary while that role is connected on standby.
- Waiting to acquire buffer cleanup locks (for which there is no time out)

Some WAL redo actions will be for DDL actions. These DDL actions are repeating actions that have already committed on the primary node, so they must not fail on the standby node. These DDL locks take priority and will automatically *cancel* any read-only transactions that get in their way, after a grace period. This is similar to the possibility of being canceled by the deadlock detector, but in this case the standby process always wins, since the replayed actions must not fail. This also ensures that replication doesn't fall behind while we wait for a query to complete. Again, we assume that the standby is there for high availability purposes primarily.

An example of the above would be an Administrator on Primary server runs a DROP TABLE command that refers to a table currently in use by a User query on the standby server.

Clearly the query cannot continue if we let the DROP TABLE proceed. If this situation occurred on the primary, the DROP TABLE would wait until the query has finished. When the query is on the standby and the DROP TABLE is on the primary, the primary doesn't have information about what the standby is running and so does not wait on the primary. The WAL change records come through to the standby while the query is still running, causing a conflict.

The second reason for conflict between standby queries and WAL redo is "early cleanup". Normally, PostgreSQL allows cleanup of old row versions when there are no users who may need to see them to ensure correct visibility of data (known as MVCC). If there is a standby query that has been running for longer than any query on the primary then it is possible for old row versions to be removed by either VACUUM or HOT. This will then generate WAL records that, if applied, would remove data on the standby that might *potentially* be required by the standby query. In more technical language, the Primary's xmin horizon is later than the Standby's xmin horizon, allowing dead rows to be removed.

We have a number of choices for resolving query conflicts. The default is that we wait and hope the query completes. If the recovery is not paused, then the server will wait automatically until the server the lag between primary and standby is at most max_standby_delay seconds. Once that grace period expires, we then take one of the following actions:

- If the conflict is caused by a lock, we cancel the standby transaction immediately, even if it is idle-in-transaction.

- If the conflict is caused by cleanup records we tell the standby query that a conflict has occurred and that it must cancel itself to avoid the risk that it attempts to silently fails to read relevant data because that data has been removed. (This is very similar to the much feared error message "snapshot too old").

  Note also that this means that idle-in-transaction sessions are never canceled except by locks. Users should be clear that tables that are regularly and heavily updated on primary server will quickly cause cancellation of any longer running queries made against those tables.

  If cancellation does occur, the query and/or transaction can always be re-executed. The error is dynamic and will not necessarily occur the same way if the query is executed again.

Other remdial actions exist if the number of cancelations is unacceptable. The first option is to connect to primary server and keep a query active for as long as we need to run queries on the standby. This guarantees that a WAL cleanup record is never generated and we don't ever get query conflicts as described above. This could be done using contrib/dblink and pg_sleep(), or via other mechanisms.

A second option is to pause recovery using recovery control functions. These can pause WAL apply completely and allows queries to proceed to completion. We can issue pg_recovery_continue() at any time, so the pause can be held for long or short periods, as the administrator allows. This method of conflict resolution may mean that there is a build up of WAL records waiting to be applied and this will progressively increase the failover delay. If there is regular arrival of WAL records this would quickly prevent the use of the standby as a high availability failover

target. Some users may wish to use multiple standby servers for various purposes. Pauses in recovery stay until explicitly released, so that pauses override the setting of max_standby_delay.

Note that max_standby_delay is set in recovery.conf. It applies to the server as a whole, so once used it may not be available for other users. They will have to wait for the server to catch up again before the grace period is available again. So max_standby_delay is a configuration parameter set by the administrator which controls the maximum acceptable failover delay and is not a user-settable parameter to specify how long their query needs to run in.

Waits for buffer cleanup locks do not currently result in query cancelation. Long waits are uncommon, though can happen in some cases with long running nested loop joins.

Dropping tablespaces or databases is discussed in the administrator's section since they are not typical user situations.

24.5.3. Administrator's Overview

If there is a recovery.conf file present then the will start in Hot Standby mode by default, though this can be disabled by setting "recovery_connections = off" in recovery.conf. The server may take some time to enable recovery connections since the server must first complete sufficient recovery to provide a consistent state against which queries can run before enabling read only connections. Look for these messages in the server logs

```
LOG:  consistent recovery state reached
LOG:  database system is ready to accept read only connections
```

If you are running file-based log shipping ("warm standby"), you may need to wait until the next WAL file arrives, which could be as long as the archive_timeout setting on the primary. This is because consistency information is recorded once per checkpoint on the primary. The consistent state can also be delayed in the presence of both transactions that contain large numbers of subtransactions and long-lived transactions.

The setting of max_connections on the standby should be equal to or greater than the setting of max_connections on the primary. This is to ensure that standby has sufficient resources to manage incoming transactions.

It is important that the administrator consider the appropriate setting of "max_standby_delay", set in recovery,conf. The default is 60 seconds, though there is no optimal setting and it should be set according to business priorities. For example if the server is primarily tasked as a High Availability server, then you may wish to lower max_standby_delay or even set it to zero. If the standby server is tasked as an additional server for decision support queries then it may be acceptable to set this to a value of many hours, e.g. max_standby_delay = 43200 (12 hours). It is also possible to set max_standby_delay to -1 which means "always wait" if there are conflicts, which will be useful when performing an archive recovery from a backup.

A set of functions allow superusers to control the flow of recovery are described in

Table 9-58. These functions allow you to pause and continue recovery, as well as dynamically set new recovery targets wile recovery progresses. Note that when a server is paused the apparent delay between primary and standby will continue to increase.

Transaction status "hint bits" written on primary are not WAL-logged, so data on standby will likely re-write the hints again on the standby. Thus the main database blocks will produce write I/Os even though all users are read-only; no changes have occurred to the data values themselves. Users will be able to write large sort temp files and re-generate relcache info files, so there is no part of the database that is truly read-only during hot standby mode. There is no restriction on use of set returning functions, or other users of tuplestore/tuplesort code. Note also that writes to remote databases will still be possible, even though the transaction is read-only locally.

Failover can be initiated at any time by allowing the startup process to reach the end of WAL, or by issuing the function pg_recovery_stop() as superuser.

The following types of administrator command will not be accepted during recovery mode

- Data Definition Language (DDL) - e.g. CREATE INDEX
- Privilege and Ownership - GRANT, REVOKE, REASSIGN
- Maintenance commands - ANALYZE, VACUUM, CLUSTER, REINDEX

Note again that some of these commands are actually allowed during "read only" mode transactions on the primary.

As a result, you cannot create additional indexes that exist solely on the standby, nor can statistics that exist solely on the standby.

pg_cancel_backend() will work on user backends, but not the Startup process, which performs recovery. pg_locks will show locks held by backends as normal. pg_locks also shows a virtual transaction managed by the Startup process that owns all AccessExclusiveLocks held by transactions being replayed by recovery. pg_stat_activity does not show an entry for the Startup process, nor do recovering transactions show as active.

check_pgsql will work, but it is very simple. check_postgres will also work, though many some actions could give different or confusing results. e.g. last vacuum time will not be maintained for example, since no vacuum occurs on the standby (though vacuums running on the primary do send their changes to the standby).

WAL file control commands will not work during recovery e.g. pg_start_backup(), pg_switch_xlog() etc..

Dynamically loadable modules work, including the pg_stat_statements.

Advisory locks work normally in recovery, including deadlock detection. Note that advisory locks are never WAL logged, so it is not possible for an advisory lock on either the primary or the standby to conflict with WAL replay. Nor is it possible to acquire an advisory lock on the primary and have it initiate a similar advisory lock

on the standby. Advisory locks relate only to a single server on which they are acquired.

Trigger-based replication systems (Slony, Londiste, Bucardo etc) won't run on the standby at all, though they will run happily on the primary server. WAL replay is not trigger-based so you cannot relay from the standby to any system that requires additional database writes or relies on the use of triggers.

New oids cannot be assigned, though some UUID generators may still work as long as they do not rely on writing new status to the database.

Currently, creating temp tables is not allowed during read only transactions, so in some cases existing scripts will not run correctly. It is possible we may relax that restriction in a later release. This is both a SQL Standard compliance issue and a technical issue, so will not be resolved in this release.

DROP TABLESPACE can only succeed if the tablespace is empty. Some standby users may be actively using the tablespace via their temp_tablespaces parameter. If there are temp files in the tablespace we currently cancel all active queries to ensure that temp files are removed, so that we can remove the tablespace and continue with WAL replay.

Running DROP DATABASE, ALTER DATABASE SET TABLESPACE, or ALTER DATABASE RENAME on primary will cause all users connected to that database on the standby to be forcibly disconnected, once max_standby_delay has been reached.

In normal running, if you issue DROP USER or DROP ROLE for a role with login capability while that user is still connected then nothing happens to the connected user - they remain connected. The user cannot reconnect however. This behaviour applies in recovery also, so a DROP USER on the primary does not disconnect that user on the standby.

Stats collector is active during recovery. All scans, reads, blocks, index usage etc will all be recorded normally on the standby. Replayed actions will not duplicate their effects on primary, so replaying an insert will not increment the Inserts column of pg_stat_user_tables. The stats file is deleted at start of recovery, so stats from primary and standby will differ; this is considered a feature not a bug.

Autovacuum is not active during recovery, though will start normally at the end of recovery.

Background writer is active during recovery and will perform restartpoints (similar to checkpoints on primary) and normal block cleaning activities. The CHECKPOINT command is accepted during recovery, though performs a restartpoint rather than a new checkpoint.

24.5.4. Hot Standby Parameter Reference

The following additional parameters are supported/provided within the `recovery.conf`.

`recovery_connections` (`boolean`)

Specifies whether you would like to connect during recovery, or not. The default is on, though you may wish to disable it to avoid software problems, should they occur. Parameter can only be changed be stopping and restarting the server.

`recovery_starts_paused` (`boolean`)

Allows the Administrator to start recovery in paused mode. The default is to start recovery so that it will continue processing all available records.

`max_standby_delay` (`string`)

This parameter allows the Administrator to set a wait policy for queries that conflict with incoming data changes. Valid settings are -1, meaning wait forever, or a wait time of 0 or more seconds. If a conflict should occur the server will delay up to this amount before it begins trying to resolve things less amicably, described in Section 24.5.2. The `max_standby_delay` may be set at server start or it may be dynamically adjusted using `pg_recovery_max_standby_delay` described in Table 9-58. start

24.5.5. Caveats

At this writing, there are several limitations of Hot Standby. These can and probably will be fixed in future releases:

- Operations on hash indexes are not presently WAL-logged, so replay will not update these indexes. Hash indexes will not be available for use when running queries during recovery.
