9.24. System Administration Functions

The functions shown in Table 9-56 assist in making on-line backups. Use of the first three functions is restricted to superusers.

Table 9-56. Backup Control Functions

Name	Return Type	Description
`pg_start_backup(label text [, fast boolean ])`	`text`	Prepare for performing on-line backup
`pg_stop_backup()`	`text`	Finish performing on-line backup
`pg_switch_xlog()`	`text`	Force switch to a new transaction log file
`pg_current_xlog_location()`	`text`	Get current transaction log write location
`pg_current_xlog_insert_location()`	`text`	Get current transaction log insert location
`pg_xlogfile_name_offset(location text)`	`text, integer`	Convert transaction log location string to file name and decimal byte offset within file
`pg_xlogfile_name(location text)`	`text`	Convert transaction log location string to file name

`pg_start_backup` accepts an arbitrary user-defined label for the backup. (Typically this would be the name under which the backup dump file will be stored.) The function writes a backup label file (`backup_label`) into the database cluster's data directory, performs a checkpoint, and then returns the backup's starting transaction log location as text. The user can ignore this result value, but it is provided in case it is useful.

```
postgres=# select pg_start_backup('label_goes_here');
 pg_start_backup
-----------------
 0/D4445B8
(1 row)
```

There is an optional boolean second parameter. If `true`, it specifies executing `pg_start_backup` as quickly as possible. This forces an immediate checkpoint which will cause a spike in I/O operations, slowing any concurrently executing queries.

`pg_stop_backup` removes the label file created by `pg_start_backup`, and creates a backup history file in the transaction log archive area. The history file includes the label given to `pg_start_backup`, the starting and ending transaction log locations for the backup, and the starting and ending times of the backup. The return value is the backup's ending transaction log location (which again can be ignored). After recording the ending location, the current transaction log insertion point is automatically advanced to the next transaction log file, so that the ending transaction log file can be archived immediately to complete the backup.

`pg_switch_xlog` moves to the next transaction log file, allowing the current file to be archived (assuming you are using continuous archiving). The return value is the ending transaction log location + 1 within the just-completed transaction log file. If there has been no transaction log activity since the last transaction log switch, `pg_switch_xlog` does nothing and returns the start location of the transaction log file currently in use.

`pg_current_xlog_location` displays the current transaction log write location in the same format used by the above functions. Similarly, `pg_current_xlog_insert_location` displays the current transaction log insertion point. The insertion point is the "logical" end of the transaction log at any instant, while the write location is the end of what has actually been written out from the server's internal buffers. The write location is the end of what can be examined from outside the server, and is usually what you want if you are interested in archiving partially-complete transaction log files. The insertion point is made available primarily for server debugging purposes. These are both read-only operations and do not require superuser permissions.

You can use `pg_xlogfile_name_offset` to extract the corresponding transaction log file name and byte offset from the results of any of the above functions. For example:

```
postgres=# SELECT * FROM pg_xlogfile_name_offset(pg_stop_backup());
        file_name         | file_offset
--------------------------+-------------
 00000001000000000000000D |     4039624
(1 row)
```

Similarly, `pg_xlogfile_name` extracts just the transaction log file name. When the given transaction log location is exactly at a transaction log file boundary, both these functions return the name of the preceding transaction log file. This is usually the desired behavior for managing transaction log archiving behavior, since the preceding file is the last one that currently needs to be archived.

For details about proper usage of these functions, see Section 24.3.

The functions shown in Table 9-57 provide information about the current status of Hot Standby. These functions may be executed during both recovery and in normal running.

Table 9-57. Recovery Information Functions

Name	Return Type	Description
`pg_is_in_recovery()`	`bool`	True if recovery is still in progress. If you wish to know more detailed status information use `pg_current_recovery_target`.
`pg_last_recovered_xid()`	`integer`	Returns the transaction id (32-bit) of the last completed transaction in the current recovery. Later numbered transaction ids may already have completed, so the value could in some cases be lower than the last time this function executed. If recovery has completed then the return value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without a recovery then the return value will be InvalidXid (zero).
`pg_last_recovered_xact_timestamp()`	`timestamp with time zone`	Returns the original completion timestamp with timezone of the last recovered transaction. If recovery is still in progress this will increase monotonically, while if recovery has completed then this value will remain static at the value of the last transaction applied during that recovery. When the server has been started normally without a recovery then the return value will be a default value.
`pg_last_recovered_xlog_location()`	`text`	Returns the transaction log location of the last recovered transaction in the current recovery. This value is updated only when transaction completion records (commit or abort) arrive, so WAL records beyond this value may also have been recovered. If recovery is still in progress this will increase monotonically. If recovery has completed then this value will remain static at the value of the last WAL record applied during that recovery. When the server has been started normally without a recovery then the return value will be InvalidXLogRecPtr (0/0).

The functions shown in Table 9-58 can be used to control archive recovery when executed in Hot Standby mode. These functions can only be executed during recovery. Their use is restricted to superusers only.

Table 9-58. Recovery Control Functions

Name	Return Type	Description
`pg_recovery_paus`	`void`	Pause recovery processing, unconditionally.

Name	Return Type	Description
e()		
pg_recovery_continue()	void	If recovery is paused, continue processing.
pg_recovery_stop()	void	End recovery and begin normal processing.
pg_recovery_pause_xid(xid integer)	void	Continue recovery until specified xid completes, if it is ever seen, then pause recovery.
pg_recovery_pause_timestamp(endtime timestamp)	void	Continue recovery until a transaction with specified timestamp completes, if one is ever seen, then pause recovery.
pg_recovery_pause_location(location text)	void	Continue recovery until a transaction with an LSN higher than the specified WAL location completes, if one is ever seen, then pause recovery. The location is specified as a string of the same form output by pg_current_xlog_location(), e.g. pg_recovery_pause_location('0/D4445B8')
pg_recovery_advance(num_records integer)	void	Advance recovery specified number of records then pause.
pg_current_recovery_target()	text	Returns details of the server's current recovery target, if any. If recovery is paused then the return value is 'Recovery paused'.
pg_recovery_max_standby_delay(delay integer)	void	Set the max_standby_delay for recovery conflict processing (in seconds).

pg_recovery_pause and pg_recovery_continue allow a superuser to control the progress of recovery on the database server. Once recovery is paused it will stay paused until you release it, even if the server falls further behind than max_standby_delay. Recovery can be paused, continued, paused, continued, etc. as many times as required. If the superuser wishes recovery to complete and normal processing mode to start, execute pg_recovery_stop.

The paused state provides a stable, unchanging database that can be queried to determine how far forwards recovery has progressed. Recovery can never go backwards because previous data may have been overwritten, so some care must be taken to recover to a specific point. pg_recovery_pause_xid and

`pg_recovery_pause_timestamp`, allow the specification of a trial recovery target, similarly to Recovery Settings. Recovery will then progress to the specified point and then pause. This allows the superuser to assess whether this is a desirable stopping point for recovery, or a good place to copy data that is known to be deleted later in the recovery. `pg_recovery_pause_location` can also be used to pause recovery after a transaction completion record arrives that has a higher LSN.

`pg_recovery_advance` allows recovery to progress record by record, for very careful analysis or debugging. Step size can be 1 or more records. If recovery is not yet paused then `pg_recovery_advance` will process the specified number of records then pause. If recovery is already paused, recovery will continue for another N records before pausing again.

If you pause recovery while the server is waiting for a WAL file when operating in standby mode it will have apparently no effect until the file arrives. Once the server begins processing WAL records again it will notice the pause request and will act upon it. This is not a bug.

You can see if recovery is paused by checking the process title, or by using `pg_current_recovery_target`.