Architecture

Buffer Cache and Logging

11

Topics

Buffer cache

Replacement algorithm

Write-Ahead Log

Checkpoint

The processes related to the buffer cache and WAL

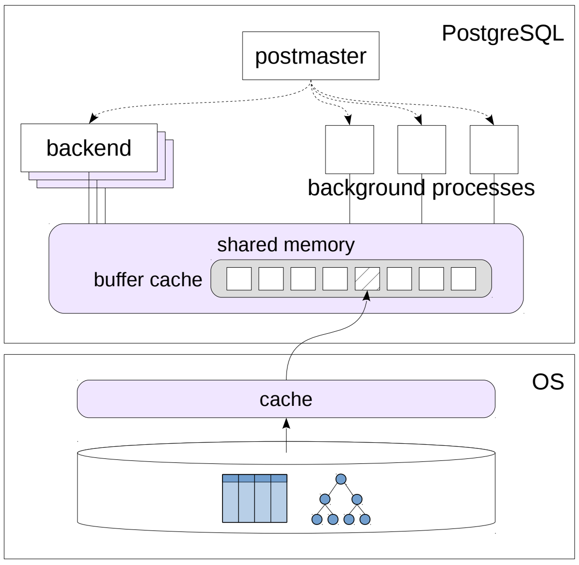
Buffer cache

Buffers array

- data page (8 KB)
- additional information

Locks in memory

- for shared access



The buffer cache is used to level the difference between the RAM and disks speed. It consists of an array of buffers that contain data pages and some additional information (for example, the file name and the position of the page inside this file).

The page size is usually 8 KB; size can only be changed when building PostgreSQL.

Any work with data pages goes through the buffer cache. If any process is going to work with the page, it first tries to find it in the cache. If the page does not exist, the process requests the operating system to read this page and places it in the buffer cache. Please note that the OS can read the page either from disk (which is slow) or from its own cache (which is fast).

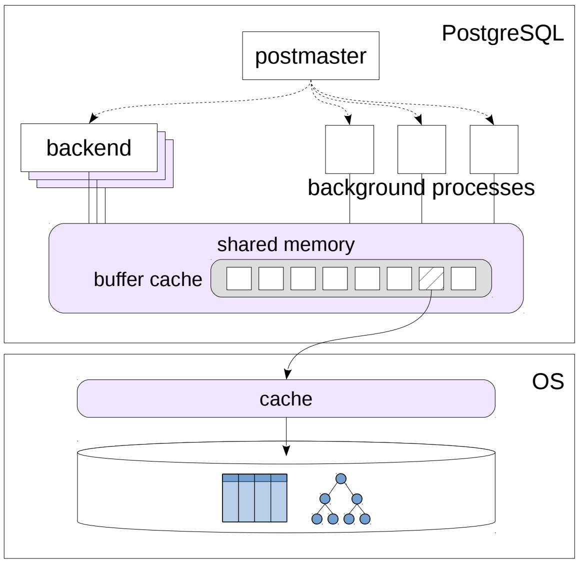
After the page is get to the buffer cache, it can be accessed many times without the overhead of operating system calls.

The buffer cache, like other shared memory structures, is protected by locks to control concurrent access. Although locks are implemented effectively, access to the buffer cache is not nearly as fast as simply accessing RAM. Therefore, in general, the less data a query reads and modifies, the faster it will work.

Replacement

Least Recently Used replacement

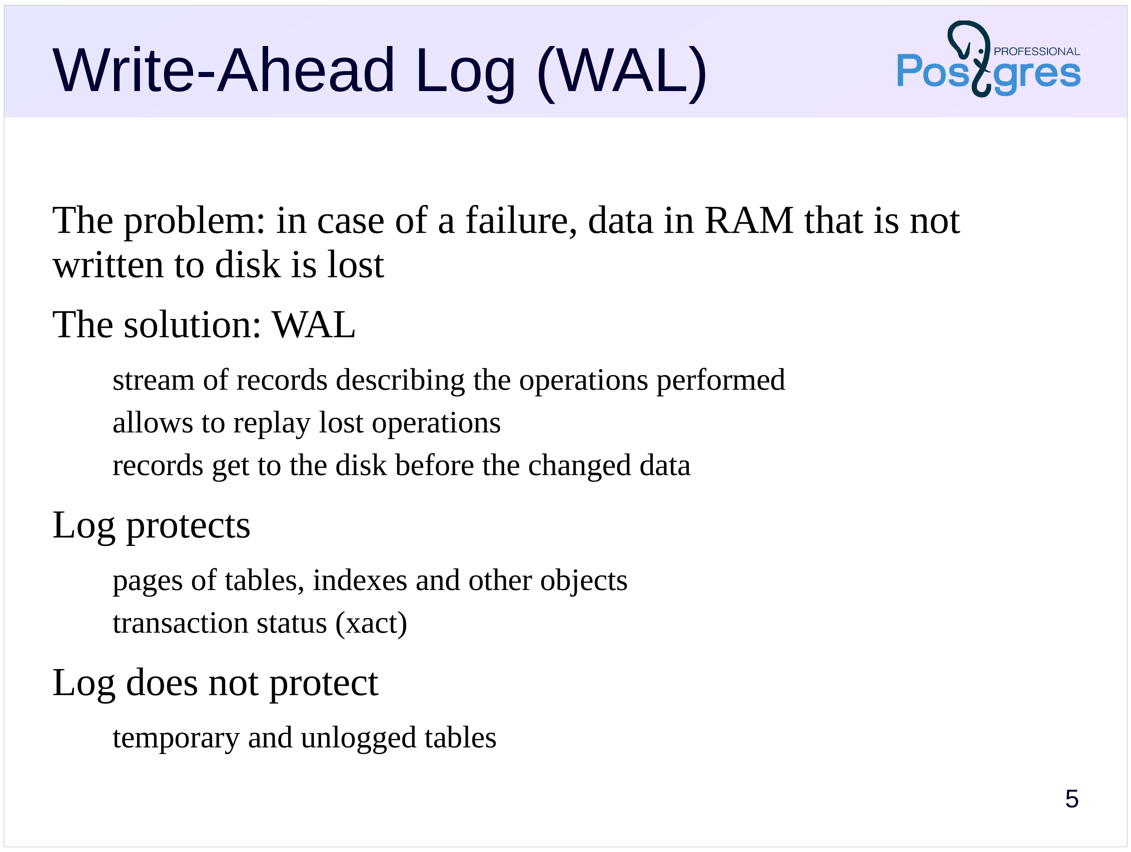
- «dirty» page is written to disk first
- page is replaced with another page



The buffer cache size is usually not so large as to fit the entire database. It is limited by the available RAM, and also the larger the buffer cache, the greater the overhead. Therefore, when reading the next page into the cache, sooner or later it turns out that the free buffers are over. In this case, page replacement is applied.

The replacement algorithm chooses a page in the cache that has recently been used less frequently than others, and replaces it with a new one. If the choosen page has been changed, then it must first be written to disk in order not to lose the changes (the buffer containing the modified page is called «dirty»).

This is called the Least Recently Used (LRU) algorithm. It ensures that «hot» data is cached. There are usually not that many hot pages even in a big database. Large enough buffer cache allows to significantly reduce the number of calls to the OS (and subsequent disk operations).



Write-Ahead Log (WAL)

The problem: in case of a failure, data in RAM that is not written to disk is lost

The solution: WAL

- stream of records describing the operations performed
- allows to replay lost operations
- records get to the disk before the changed data

Log protects

- pages of tables, indexes and other objects
- transaction status (xact)

Log does not protect

- temporary and unlogged tables

The buffer cache (and other buffers in RAM) increases performance, but decreases reliability. In the event of a failure in the DBMS, the contents of the buffer cache is lost. If a crash occurs in the operating system or at the hardware level, the contents of the OS buffers is also lost (but the operating system itself copes with this).

PostgreSQL uses write-ahead logging to ensure reliability. When performing any operation, a record is created containing the minimum necessary information so that the operation can be performed again. Such a record must get on the disk (or another non-volatile storage) before the data modified by the operation will be written (which is why it is called the *write-ahead* log).

Log files have traditionally been located in `PGDATA/pg_xlog` directory; since version 10, the directory has been renamed to `pg_wal`.

WAL protects all objects that are handled in RAM buffers: tables, indexes and other objects, transactions status.

The log does not contain data on temporary tables (such tables are accessible only to the user who created it and only for the duration of the session or transaction) and unlogged tables (such tables are no different from regular tables, except that they are not protected by the log). In case of a failure, such tables are simply cleared. The meaning of their existence is that work with them is faster.

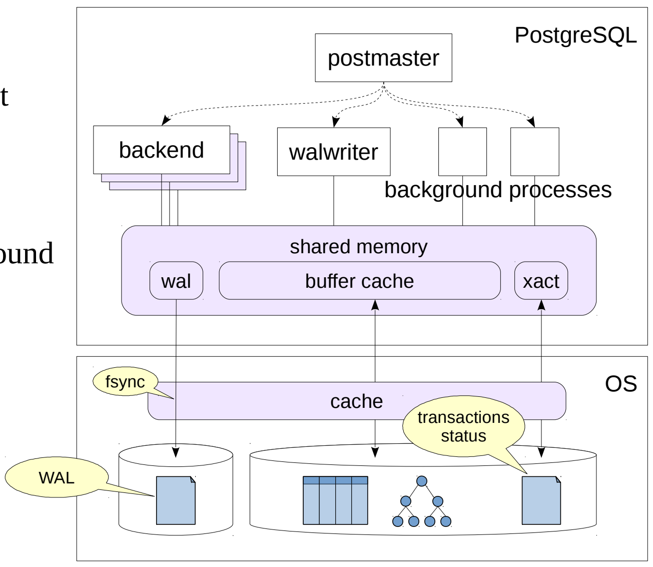
Performance

Synchronous mode

- write and sync on commit
- backend

Asynchronous mode

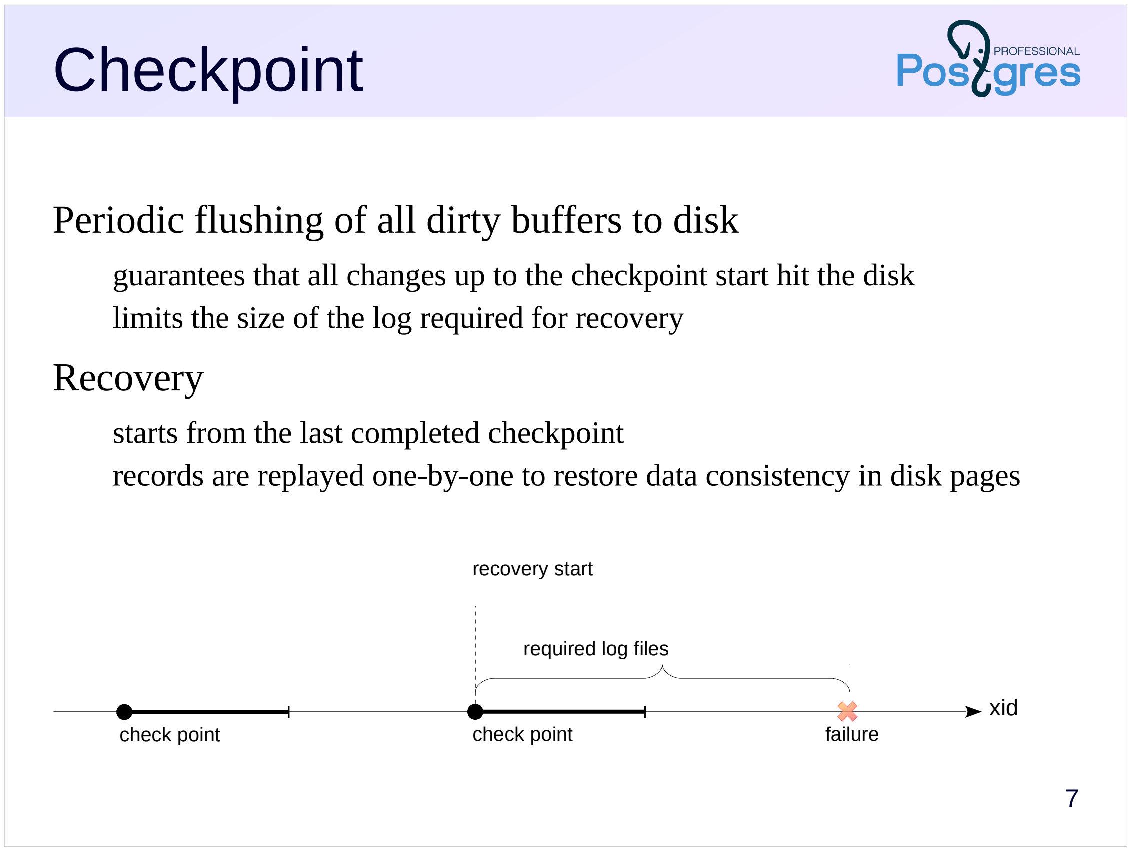
- write and sync in background
- walwriter



The logging mechanism is more efficient than working directly with a disk without a buffer cache. First, the size of the log entries is smaller than the size of the whole page of data. Second, the log is written strictly sequentially (and usually is read only in case of recovery), which even HDDs do quite well.

Performance can also be influenced by tuning. If the WAL records are written immediately (synchronously), then it is guaranteed that the committed data will not be lost. But writing is a rather expensive operation, during which the backend process that performs COMMIT has to wait. To ensure that the log entry is not «stuck» in the operating system cache, the `fsync` call is made: PostgreSQL relies on this call to make sure that the data hit a non-volatile media.

Therefore, there is a delayed (asynchronous) mode. In this case, records are written by the `walwriter` background process gradually, with a slight delay. This somewhat decreases reliability, but greatly increases performance. Even in this case data consistency after recovery is guaranteed.



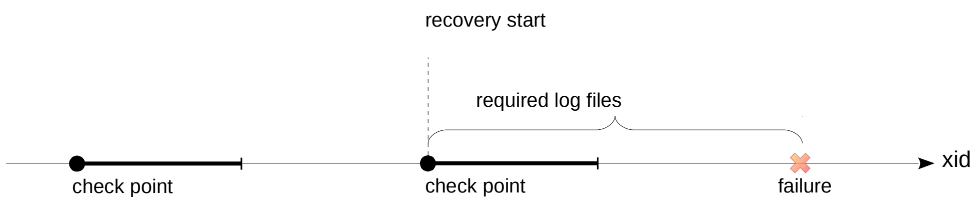
Checkpoint

Periodic flushing of all dirty buffers to disk

- guarantees that all changes up to the checkpoint start hit the disk
- limits the size of the log required for recovery

Recovery

- starts from the last completed checkpoint
- records are replayed one-by-one to restore data consistency in disk pages



When PostgreSQL starts after a crash, the server enters a recovery mode. The disk after a crash is known to contain inconsistent information: data pages were written in different moments of time and contain data in different states.

To restore consistency, PostgreSQL reads the write-ahead log and replays log records one-by-one if the corresponding change is not yet on the disk. This way, changes of all transactions are restored, except for those whose commit record did not get into the log.

However, the log volume during server operation can reach gigantic proportions. It is absolutely impossible to keep the entire log and read it from the very beginning in case of a failure. Therefore, the DBMS periodically performs a *checkpoint*: forcibly flushes all dirty buffers (including the state of transactions) to disk. This ensures that changes of all pages done before the checkpoint are on disk.

The checkpoint can take a lot of time, and this is normal. Actually, the «point» of time we refer to is the beginning of the process. But the checkpoint is considered to be completed only after all the dirty buffers that presented at the time of the process start are flushed.

Recovery begins at the nearest completed checkpoint, which allows PostgreSQL to store only log files written since the last completed checkpoint. Previous log files are discarded automatically to free up disk space.

WAL-related processes

WAL writer

Checkpointer

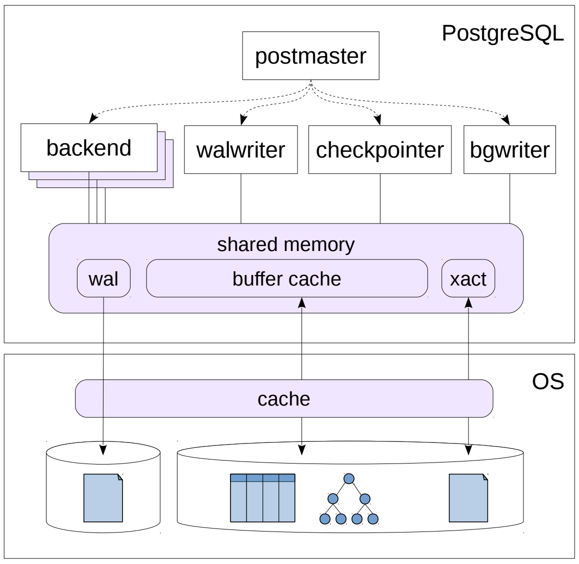
flushes all dirty buffers

Background writer

flushes some of dirty buffers

Backends

flush when forced
to replace a dirty buffer



Let us return to the illustration and clarify the remaining processes related to the maintenance of the buffer cache and log.

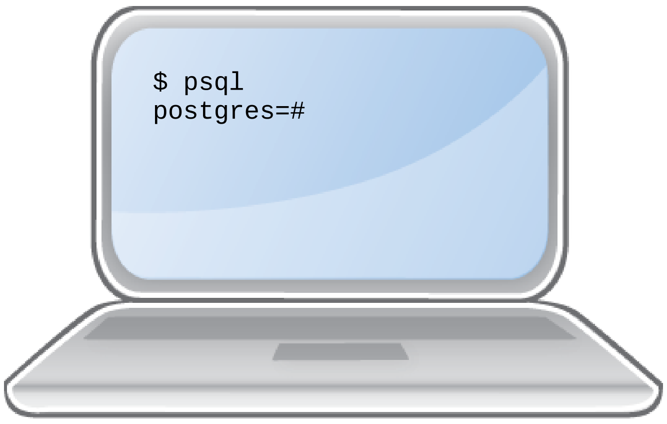
First, it is a **walwriter** process that asynchronously writes log records to disk. In the synchronous mode, the log records are written by the process that commits the transaction.

Second, the **checkpointer** process, periodically flushing all dirty buffers to disk.

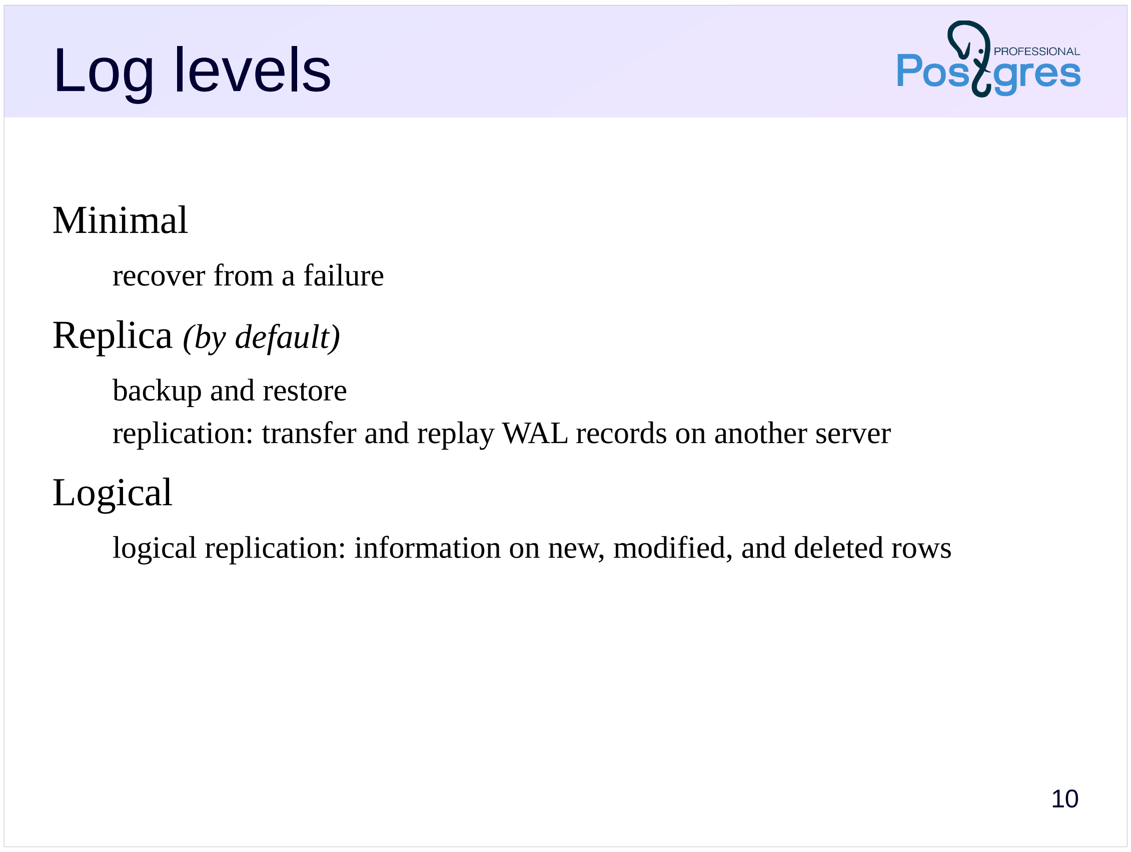
Third, the **bgwriter** (or simply «writer») background process. This process is similar to the checkpoint process, but it records only part of the dirty buffers, and those that are likely to be replaced in the nearest future. Thus, when a backend needs a buffer, it will most likely find it not dirty and will not waste time to flush the buffer to disk.

And fourth, **backend** processes that read the data into the buffer cache. If, despite the work of the checkpoint and background processes, the backend finds the page to be dirty, the backend will have to write it to disk itself.

Demonstration



```
$ psql
postgres=#
```



Log levels

Minimal

recover from a failure

Replica *(by default)*

backup and restore

replication: transfer and replay WAL records on another server

Logical

logical replication: information on new, modified, and deleted rows

10

As already mentioned, the reason for the existing of the log is the need to protect information from failures due to the loss of the contents of RAM.

However, the log is a mechanism that has turned out to be convenient for other purposes, if we add more information to it.

The amount of data that goes into the log is determined by the *wal_level* parameter.

- At the level of `minimal` the log provides only recovery from a failure. Before version 10, this level was set by default.

- At the `replica` level, information is added to the log, allowing it to be used for creating and restoring backups (see the «Backup» module) and replication (the «Replication» module). During replication, log records are transferred to another server and applied there; this creates and maintains an exact binary copy (replica) of the primary server.

Before version 9.6, there were two separate levels (`archive` and `hot_standby`), but they were merged into one common level.

- At the `logical` level, some more information is added that allow for logical replication, that is replicating table rows instead of page changes (also discussed in the module «Replication»).

Summary

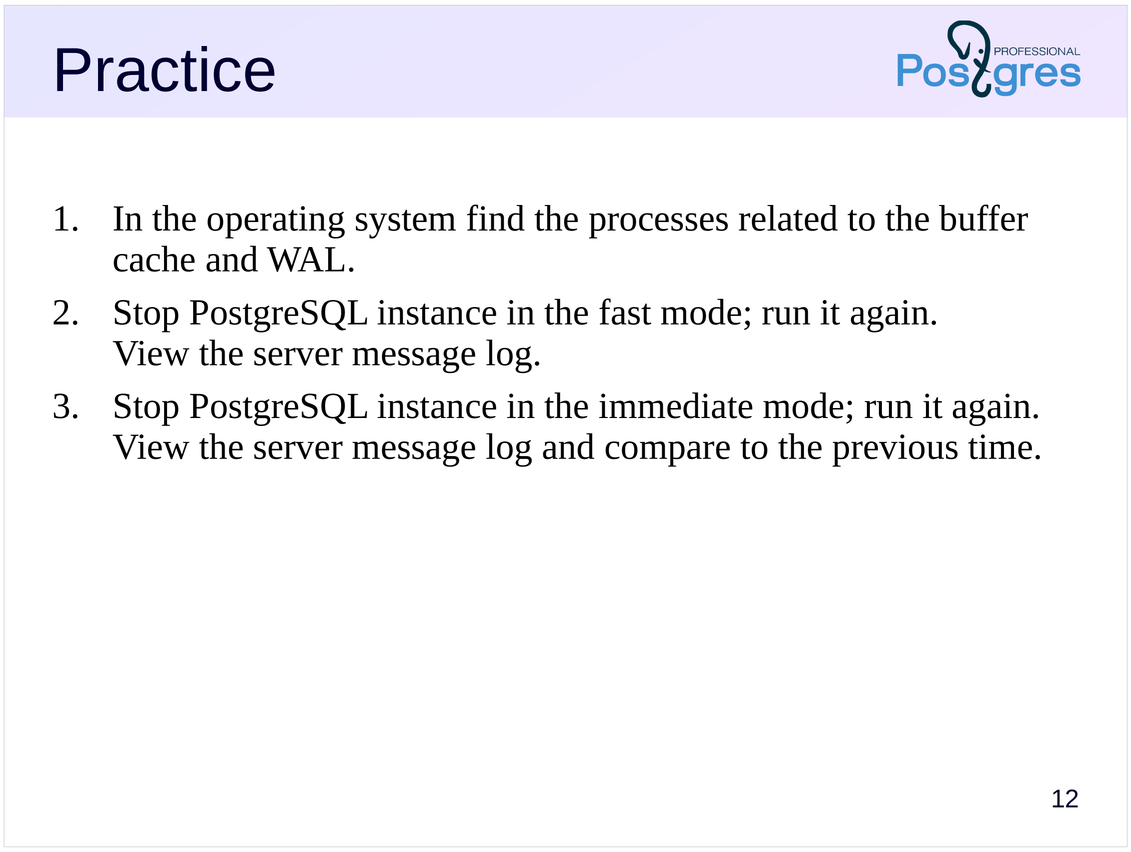
Buffer cache significantly improves performance, reducing the number of disk operations

Reliability is provided by logging

Log size is limited due to checkpoints

WAL is convenient in many cases:

- recovery
- backup
- replication



Practice

1. In the operating system find the processes related to the buffer cache and WAL.
2. Stop PostgreSQL instance in the fast mode; run it again. View the server message log.
3. Stop PostgreSQL instance in the immediate mode; run it again. View the server message log and compare to the previous time.

2. Use the command

```
pg_ctlcluster 11 alpha stop
```

In this mode the server terminates all open connections and performs a checkpoint before shutting down so that consistent data is written to the disk. Thus, shutdown may take a relatively long time, but at startup the server will immediately be ready for operation.

3. Use the command

```
pg_ctlcluster 11 alpha stop -m immediate \
--skip-systemctl-redirect
```

In this mode the server also terminates open connections, but does not execute a shutdown checkpoint. This leaves inconsistent data on disk, as after a failure. Thus, the shutdown occurs quickly, but at startup the server will have to recover data consistency using the log.