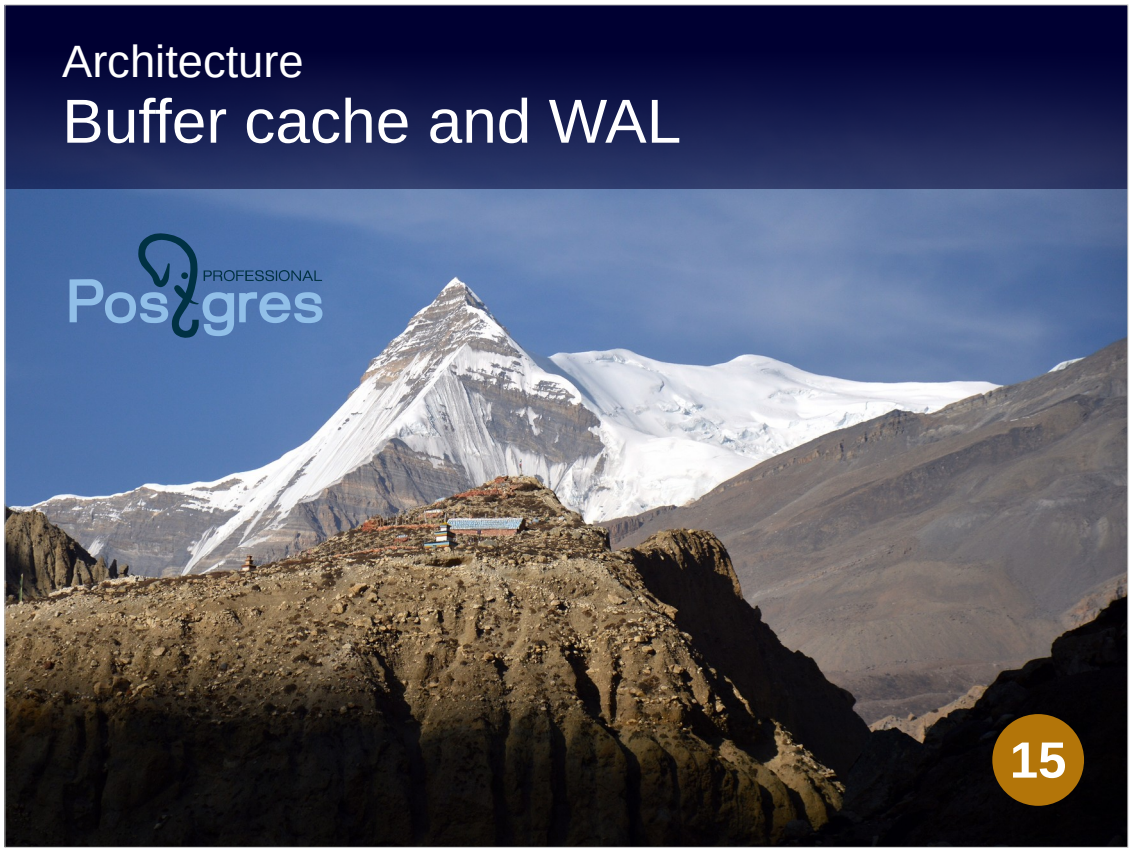


Authors: Egor Rogov, Pavel Luzanov, Ilya Bashtanov
Translated by Alexander Meleshko
Cover photo by Oleg Bartunov (Phu monastery and Bhrikuti peak, Nepal)

Feedback
Please send your feedback, comments and suggestions to:
edu@postgrespro.ru

Agenda

Buffer cache overview

Replacement algorithm

Write-ahead log

Checkpoint

Processes related to the buffer cache and WAL

Buffer cache

Buffer array

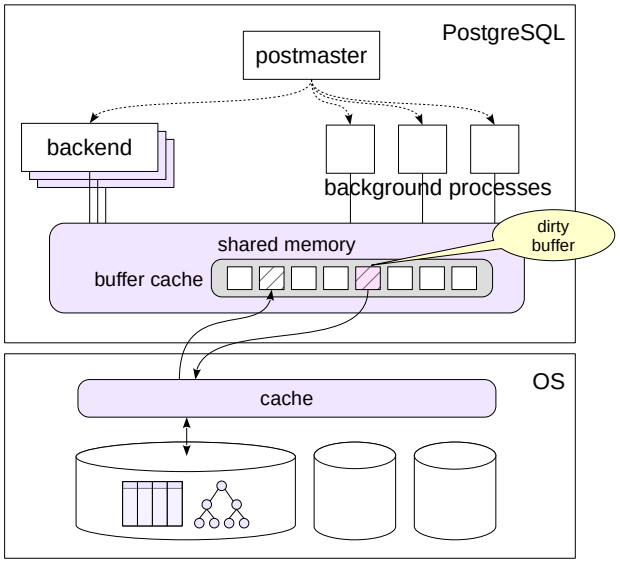
- data page (8 kB)
- additional information

Dirty buffers

- asynchronous write

Locks in memory

- for shared access



The buffer cache is used to smooth out the difference between the RAM and disk 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; the 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. (Note that the OS can read the page either from disk or from its own cache.)

After the page gets into the buffer cache, it can be accessed repeatedly without the overhead of operating system calls.

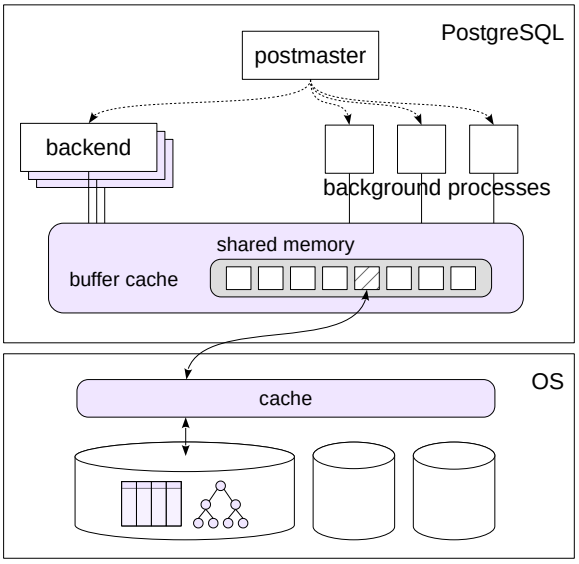
If a process has changed the data in the page, the corresponding buffer is called “dirty”. The modified page must be written on disk, but for performance reasons, the flushing occurs asynchronously and may be delayed.

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 buffer is written on disk
- another page is read into the vacant space



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, sooner or later the buffer cache will run out of space. In this case, *page replacement* occurs.

Page replacement selects a page in the cache that has been used less often than others. If the selected buffer is dirty, the page is written on disk first to store the changes. Then a new page gets into the buffer.

This is called the Least Recently Used replacement, or LRU. It keeps the most frequently accessed data in the cache. Such “hot” blocks of data are not very common, and this approach helps to significantly reduce the number of requests to OS (and disk operations), provided enough cache memory.

The effect of buffer cache on query execution

Create a table:

```
=> CREATE TABLE t(n integer);

CREATE TABLE
```

Populate it with rows:

```
=> INSERT INTO t SELECT id FROM generate_series(1,100000) AS id;

INSERT 0 100000

=> VACUUM ANALYZE t;

VACUUM
```

The shared_buffers parameter indicates the buffer cache size:

```
=> SHOW shared_buffers;

 shared_buffers
----------------
 128MB
(1 row)
```

The default value is too low. In the real world, you should increase it immediately after server installation (it will apply after a restart).

Restart the server to wipe the cache clean.

```
student$ sudo pg_ctlcluster 15 main restart

student$ psql
```

Now, let's compare the behaviour of the system as we run a query once, and then the same query again. Query execution plans are not the topic of this course, but we will peek into them every now and again. The EXPLAIN ANALYZE command used below will execute the query as well as display the execution plan and some extra details:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM t;

                 QUERY PLAN
--------------------------------------------
 Seq Scan on t (actual rows=100000 loops=1)
   Buffers: shared read=443
 Planning:
   Buffers: shared hit=12 read=7 dirtied=1
 Planning Time: 0.171 ms
 Execution Time: 11.163 ms
(6 rows)
```

The "Buffers: shared" line shows the buffer utilization.

- read is the number of buffers that pages from disk were written into.

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
SELECT * FROM t;

                 QUERY PLAN
--------------------------------------------
 Seq Scan on t (actual rows=100000 loops=1)
   Buffers: shared hit=443
 Planning Time: 0.027 ms
 Execution Time: 7.950 ms
(4 rows)
```

- hit is the number of buffers that contained any of the queried pages.

Note that on the second query execution, not only the execution time went down, but the planning time too (because system catalog pages are cached as well).

Write-ahead log (WAL)

Problem: when a crash occurs, data from RAM that is not written on disk is lost

WAL

- a stream of records of the actions being performed; can be used to re-trace the steps lost during the crash
- the records are stored on disk before the actual changes are

The log tracks changes to

- pages in tables, indexes and other objects
- transaction status (clog)

The log does not track changes to

- temporary and unlogged tables

Having a buffer cache (and other RAM buffers) increases performance at the cost of reliability. When a crash happens, all buffer cache content is lost. If the crash occurs on the OS or hardware level, the content of OS buffers will also be lost (the OS may have its own failsafes for this).

To ensure reliability, PostgreSQL uses the Write-ahead log. When performing any operation, the log records minimum necessary information about the operation to be able to perform it again. The record must be written into non-volatile memory before the data modified by the operation is (that's why it's called W*rite-ahead* log).

WAL files are located in the PGDATA/pg_wal directory.

All objects that are being worked on in RAM have their operations logged. These include tables, indexes and other objects, and transaction statuses.

Operations with *temporary tables* (tables which exist only during the scope of a session or a transaction and are only available to the user who has created them) aren't logged. You can also set a regular table to be explicitly *unlogged*. The table will be quicker to work with, but will be wiped on crash.

https://postgrespro.com/docs/postgresql/15/wal-intro

Write-ahead log (WAL)

You can imagine WAL as a continuous stream of records. Each record has a 64-bit Log Sequence Number, or LSN — an offset from the beginning of the log, in bytes.

The current log position can be seen with pg_current_wal_lsn:

```
=> SELECT pg_current_wal_lsn();

 pg_current_wal_lsn
--------------------
 0/47C6890
(1 row)
```

The position is displayed as two 32-bit numbers separated by a slash. Let's save it for future reference.

Now let's perform a bunch of operations and see what's changed.

```
=> UPDATE t SET n = 100001 WHERE n = 1;

UPDATE 1

=> SELECT pg_current_wal_lsn();

 pg_current_wal_lsn
--------------------
 0/47C98A8
(1 row)
```

It's not the absolute values we're interested in, but the distance between them, as it shows the size of generated log entries in bytes:

```
=> SELECT '0/47C98A8'::pg_lsn - '0/47C6890'::pg_lsn AS bytes;

 bytes
-------
 12312
(1 row)
```

The log is stored in files in a separate catalog (PGDATA/pg_wal). By default, the files are 16 MB each, but you can change that during cluster initialization.

In addition to browsing the files by means of the OS, you can also display them by the following command:

```
=> SELECT * FROM pg_ls_waldir() ORDER BY name;

           name            |   size   |      modification
---------------------------+----------+------------------------
 000000010000000000000004 | 16777216 | 2023-05-04 19:45:52+03
 000000010000000000000005 | 16777216 | 2023-05-04 19:45:41+03
 000000010000000000000006 | 16777216 | 2023-05-04 19:45:42+03
(3 rows)
```

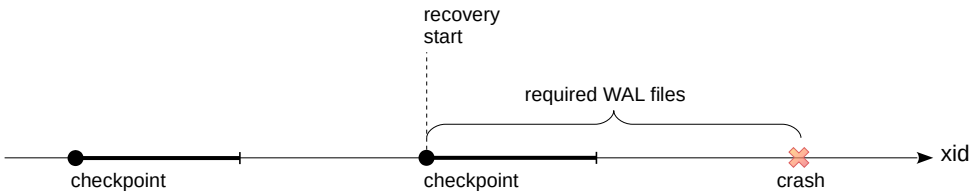
Checkpoint

Regular flushing of all dirty buffers to disk

- ensures that all data changes before the checkpoint get to the disk
- limits the size of the log required for recovery

Crash recovery

- starts from the last checkpoint
- WAL records are replayed one-by-one to restore data consistency



When PostgreSQL crashes, it enters the recovery mode on the next start. The data on disk at this point is inconsistent. Changes to hot pages were in the buffer cache and are now lost, while some of the later changes have been flushed to disk already.

To restore consistency, PostgreSQL reads the WAL log and sequentially reads the records, replaying the changes that did not make it to the disk. This way, the state of all transactions at the time of the crash is restored. Then, any transactions that haven't been logged as committed are aborted.

However, logging all changes throughout a server's lifetime and replaying everything from day 1 after each crash is impractical, if not impossible. Instead, PostgreSQL uses checkpoints. Every now and then, it forces all dirty buffers to disk (including clog buffers with transaction statuses) to ensure that all data changes up to this point are safe in non-volatile memory.

This state is called a checkpoint. The “point” in checkpoint is the moment in time when the flushing of all data to disk is started. However, you only have a valid checkpoint when the flushing is complete, and it may take a bit of time.

Now, when a crash occurs, you can start recovery from the closest checkpoint. Consequently, it's sufficient to store WAL files only as far back as the last checkpoint goes.

WAL and crash recovery

So far, current page changes are in the buffer cache, but not on disk. On a regular shutdown, the server will perform a checkpoint and write all dirty pages to disk. Instead, let's simulate a crash by sending the following command to postmaster:

```
student$ sudo head -n 1 /var/lib/postgresql/15/main/postmaster.pid

7103

student$ sudo kill -9 7103
```

When the server comes back up, it should begin recovery:

```
student$ sudo pg_ctlcluster 15 main start

student$ psql

=> SELECT min(n), max(n) FROM t;

 min |  max
-----+--------
   2 | 100001
(1 row)
```

All the changes have been recovered.

After performing a checkpoint, PostgreSQL automatically deletes log files that are no longer necessary for recovery.

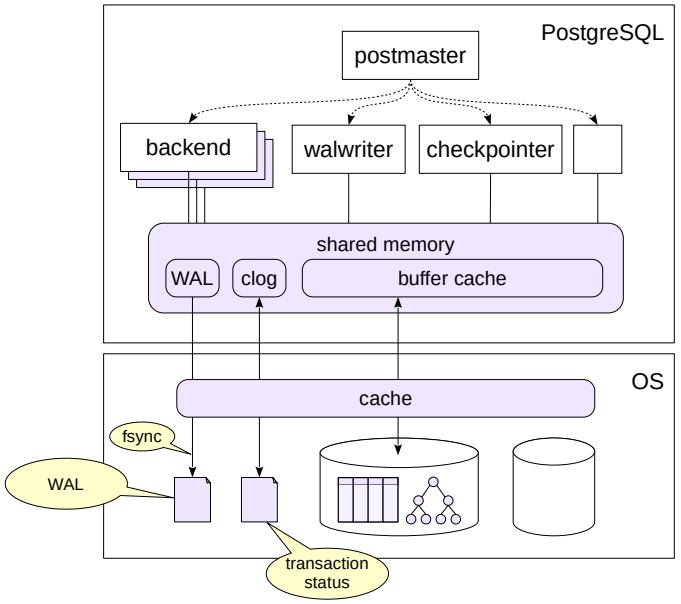
Performance

Synchronous mode

- write on commit
- backend

Asynchronous mode

- write in background
- walwriter



10

The WAL approach is faster than working directly with disk without a buffer cache. Firstly, a WAL record is smaller than an entire page of data. Secondly, the log is written sequentially (and usually not read until a crash occurs), which is better for basic hard disk drives.

Various configurations also affect WAL performance. If the records are stored to disk immediately (synchronous mode), this guarantees that the committed operation will get to disk one way or the other. But recording to disk is expensive, and forces the committing process to wait in line. To prevent log entries from being “stuck” in the OS cache, PostgreSQL calls the fsync function, which forces the data into non-volatile storage.

There is also asynchronous mode, which has a background process (walwriter) constantly sending WAL records to disk with a certain delay. It's more efficient at the cost of some reliability, but still ensures consistency after crash recovery.

In fact, both modes work together. Long transaction log records are written asynchronously (to free up WAL buffers). And if a pages is getting flushed to disk and the corresponding log record isn't there yet, it will immediately be recorded in synchronous mode.

Main processes

WAL Writer

Checkpointer

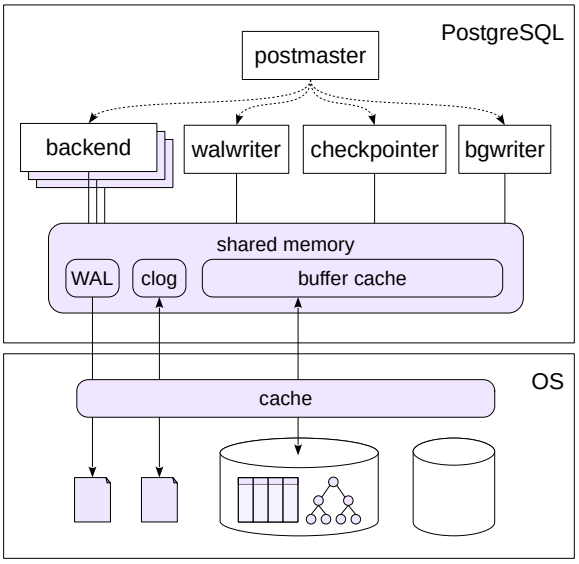
flush all dirty buffers

Background writer

flush some dirty buffers

Backend

flush the replaced dirty buffer



Let's take a step back and look at the processes that maintain the buffer cache and the WAL.

First, there is walwriter. This process writes WAL records to disk in asynchronous mode. In synchronous mode, this job is handled by the process that commits the transaction.

Second, checkpointer, the checkpoint process. It periodically flushes all dirty buffers to disk.

Third, bgwriter (or background writer). It operates similarly to checkpointer, but it only flushes some of the dirty buffers, prioritizing the ones which are at a high risk of being replaced soon. It frees up buffer space so that when a background worker selects a buffer to put a new page in, it doesn't have to flush the old contents of the buffer to disk itself.

Fourth, there are backends that put data into the buffer cache. Whenever a buffer being replaced is still dirty (despite the efforts of checkpointer and bgwriter), the background process will flush it to disk.

Log levels

Minimal

guarantees crash recovery

Replica *(default)*

backup

replication: transfer and replay of the log on another server

Logical

logical replication: information about adding, changing, and deleting table rows

WAL was developed as a data protection tool to mitigate the risk of data loss due to crashes.

However, the WAL mechanism turned out to have other applications, if its records are supplemented with additional info.

The amount of data stored in each WAL record is controlled by the *wal_level* parameter.

- The **minimal** level is sufficient to recover after a crash, and nothing else.

- The **replica** level stores additional information that allows it to be used for backup (see the Backup module) and replication (see the Replication module). During replication, WAL records are streamed to the replica and applied there, creating an exact copy of the original server.

- At the **logical** level, information is added to the log that allows decoding “physical” log entries and forming “logical” records of adding, changing and deleting table rows. This is logical replication (also discussed in the “Replication” module).

Takeaways

Buffer cache increases performance by reducing the number of disk operations

WAL ensures reliability

WAL size is kept in check by checkpoints

WAL has multiple uses

- crash recovery
- backup
- replication

Practice

1. Using the OS tools, find the processes responsible for the buffer cache and the WAL.
2. Stop PostgreSQL in fast mode; start it again.
   Check the server message log.
3. Now stop PostgreSQL in immediate mode; start it again.
   Check the server message log and compare
   with the previous one.

Task 2. To stop in fast mode, use the command

```
pg_ctlcluster 15 main stop
```

This makes the server abort all open connections and perform a checkpoint before shutting down, so that all data is flushed to disk and consistent. In this mode, the shutdown may take some time, but on startup the server will be good to go right away.

Task 3. To stop in immediate mode, use the command

```
pg_ctlcluster 15 main stop -m immediate --skip-systemctl-redirect
```

The server will also abort open connections, but will not perform a checkpoint. Data on disk will be inconsistent, like after a crash. In this mode, the server shuts down quickly, but will enter recovery mode on startup and will use the WAL to reach consistency.

If your PostgreSQL is compiled from source code, the fast stop command will be

```
pg_ctl stop
```

and the immediate stop command will be

```
pg_ctl stop -m immediate
```

Task 1. Operating system processes

First, we need to get the postmaster process ID. It is stored in the first line of the postmaster.pid file. The file is located in the data directory and is created each time the server starts.

```
student$ sudo cat /var/lib/postgresql/15/main/postmaster.pid
```

```
17295
/var/lib/postgresql/15/main
1683219035
5432
/var/run/postgresql
localhost
   655372        62
ready
```

Check all the child processes of postmaster:

```
student$ sudo ps -o pid,command --ppid 17295
```

```
    PID COMMAND
  17296 postgres: 15/main: checkpointer
  17297 postgres: 15/main: background writer
  17299 postgres: 15/main: walwriter
  17301 postgres: 15/main: logical replication launcher
  19390 postgres: 15/main: autovacuum launcher
```

Processes that support the buffer cache and WAL include:

- checkpointer;
- background writer;
- walwriter.

Task 2. Stopping the server in the fast mode

In order to easily separate new log messages from old ones, we will simply delete the log file before we start the server. Of course, this is not a good idea to do in production.

```
student$ sudo rm /var/log/postgresql/postgresql-15-main.log
```

```
student$ sudo pg_ctlcluster 15 main restart
```

Server message log:

```
student$ cat /var/log/postgresql/postgresql-15-main.log
```

```
2023-05-04 19:51:01.222 MSK [19615] LOG:  starting PostgreSQL 15.1 (Ubuntu 15.1-1.pgdg22.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, 64-bit
2023-05-04 19:51:01.223 MSK [19615] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-05-04 19:51:01.225 MSK [19615] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-05-04 19:51:01.232 MSK [19618] LOG:  database system was shut down at 2023-05-04 19:51:00 MSK
2023-05-04 19:51:01.239 MSK [19615] LOG:  database system is ready to accept connections
```

Task 3. Stopping the server in the immediate mode

```
student$ sudo rm /var/log/postgresql/postgresql-15-main.log
```

```
student$ sudo pg_ctlcluster 15 main stop -m immediate --skip-systemctl-redirect
```

```
student$ sudo pg_ctlcluster 15 main start
```

Server message log:

```
student$ cat /var/log/postgresql/postgresql-15-main.log
```

```
2023-05-04 19:51:04.153 MSK [19732] LOG:  starting PostgreSQL 15.1 (Ubuntu 15.1-1.pgdg22.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0, 64-bit
2023-05-04 19:51:04.154 MSK [19732] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2023-05-04 19:51:04.155 MSK [19732] LOG:  listening on Unix socket "/var/run/postgresql/.s.PGSQL.5432"
2023-05-04 19:51:04.162 MSK [19735] LOG:  database system was interrupted; last known up at 2023-05-04 19:51:01 MSK
2023-05-04 19:51:04.993 MSK [19735] LOG:  database system was not properly shut down; automatic recovery in progress
2023-05-04 19:51:04.996 MSK [19735] LOG:  invalid record length at 0/12A43668: wanted 24, got 0
2023-05-04 19:51:04.996 MSK [19735] LOG:  redo is not required
2023-05-04 19:51:04.999 MSK [19733] LOG:  checkpoint starting: end-of-recovery immediate wait
2023-05-04 19:51:05.007 MSK [19733] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.003 s, sync=0.001 s, total=0.009 s; sync file
2023-05-04 19:51:05.011 MSK [19732] LOG:  database system is ready to accept connections
```

Before getting ready to receive queries, the system performed an automatic recovery (automatic recovery in progress).