

Architecture Buffer Cache and WAL



Copyright

© Postgres Professional, 2017–2025

Authors Egor Rogov, Pavel Luzanov, Ilya Bashtanov, Igor Gnatyuk

Translated by: Liudmila Mantrova, Alexander Meleshko, Elena Sharafutdinova

Photo by: Oleg Bartunov (Phu monastery, Bhrikuti Peak, Nepal)

Use of Course Materials

Non-commercial use of course materials (presentations, demonstrations) is allowed without restrictions. Commercial use is possible only with the written permission of Postgres Professional. It is prohibited to make changes to the course materials.

Feedback

Please send your feedback, comments and suggestions to:

edu@postgrespro.ru

Disclaimer

Postgres Professional assumes no responsibility for any damages and losses, including loss of income, caused by direct or indirect, intentional or accidental use of course materials. Postgres Professional company specifically disclaims any warranties on course materials. Course materials are provided “as is,” and Postgres Professional company has no obligations to provide maintenance, support, updates, enhancements, or modifications.

Buffer Cache

Replacement Algorithm

Write-Ahead Log

Checkpoint

Processes Related to the Buffer Cache and WAL

Buffer Cache

Buffer array

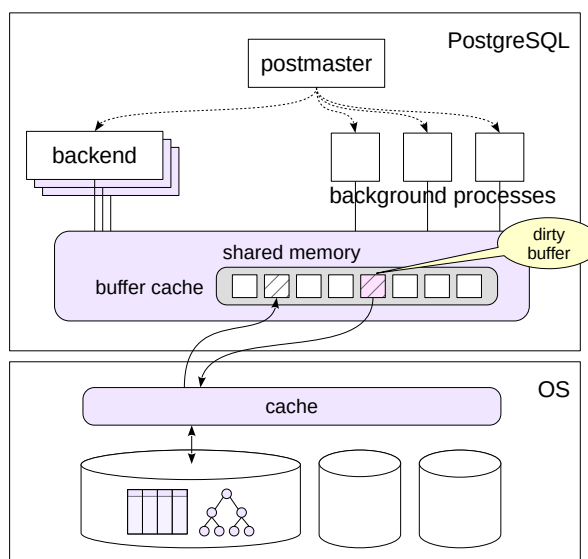
data page (8 KB)
additional information

“Dirty” buffers

asynchronous write

Locks in memory

for shared access



3

The buffer cache is used to smooth out the difference between the RAM and disk speed. It consists of an array of buffers which 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 is written to the buffer cache, it can be accessed repeatedly without the overhead of operating system calls.

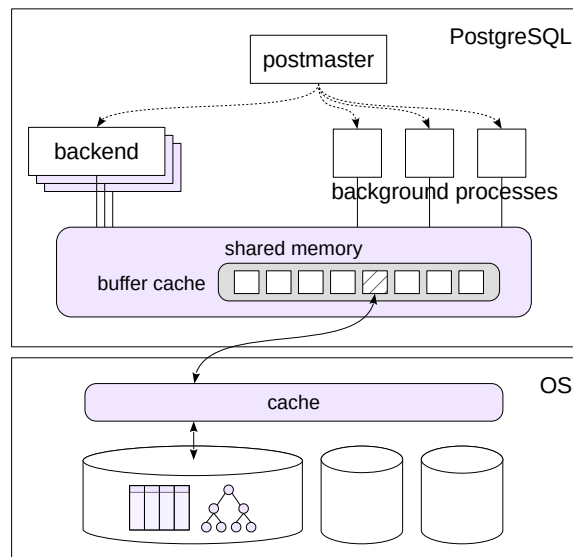
If a process has changed the data in the page, the corresponding buffer becomes “dirty”. The modified page must be written on disk, but for performance reasons, the recording occurs asynchronously and may be delayed.

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. Also, the larger the buffer cache, the greater the overhead. Therefore, when reading the next page, sooner or later the buffer cache has to run out of space. In this case, *page replacement* happens.

The replacement algorithm 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 made to it. Then, a new page is written into the buffer.

This replacement is called LRU (Least Recently Used). 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 Impact of Buffer Cache on Query Execution

Creating a new database in the cluster and connecting to it (for more details about databases, see the Data Organization module):

```
=> CREATE DATABASE arch_wal_overview;
```

```
CREATE DATABASE
```

```
=> \c arch_wal_overview
```

You are now connected to database "arch_wal_overview" as user "student".

Create a table:

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

```
CREATE TABLE
```

Populate it with rows:

```
=> INSERT INTO t SELECT id FROM generate_series(1,100_000) AS id;
```

```
INSERT 0 100000
```

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 be applied after restart).

Restart the server to wipe the cache clean.

```
=> \q
```

```
student$ sudo pg_ctlcluster 16 main restart
```

```
student$ psql arch_wal_overview
```

Now, let's compare the behaviour of the system as we run a query once, and then the same query again. Query plans is 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=8 dirtied=1
Planning Time: 0.212 ms
Execution Time: 22.889 ms
(6 rows)
```

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

- read — the number of buffers where pages had to be read from disk.

```
=> 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: 0.039 ms
Execution Time: 11.580 ms
(4 rows)
```

- hit — the number of buffers where requested pages were found.

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

- stream of records of the actions being performed;
- can be used to redo the steps lost during the crash
- records are written to disk earlier than the changed data

WAL tracks changes to

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

WAL does not track changes to

- temporary and unlogged tables

6

Having a buffer cache (and other buffers in RAM) 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 increase reliability, PostgreSQL uses the Write-ahead log. When performing any operation, the WAL records minimum necessary information about the operation to be able to perform it again. The record must be written into the disk (or another persistent storage) before the data modified by the operation is (that is why it is called *Write-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, as well as 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) are not 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/16/wal-intro>

Write-Ahead Log

The WAL can be considered as a continuous stream of records. Each record has a unique ID called an LSN (Log Sequence Number). This 64-bit number represents the record's byte offset from the start of the WAL.

The current WAL position can be seen with `pg_current_wal_lsn` function:

```
=> SELECT pg_current_wal_lsn();
```

```
pg_current_wal_lsn
-----
0/2378028
(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 some operations and see what's changed.

```
=> UPDATE t SET n = 100_001 WHERE n = 1;
```

```
UPDATE 1
```

```
=> SELECT pg_current_wal_lsn();
```

```
pg_current_wal_lsn
-----
0/237B040
(1 row)
```

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

```
=> SELECT '0/237B040'::pg_lsn - '0/2378028'::pg_lsn AS bytes;
```

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

The WAL 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 LIMIT 10;
```

name	size	modification
00000001000000000000000002	16777216	2025-09-24 16:58:26+03
00000001000000000000000003	16777216	2025-09-24 16:58:18+03

```
(2 rows)
```

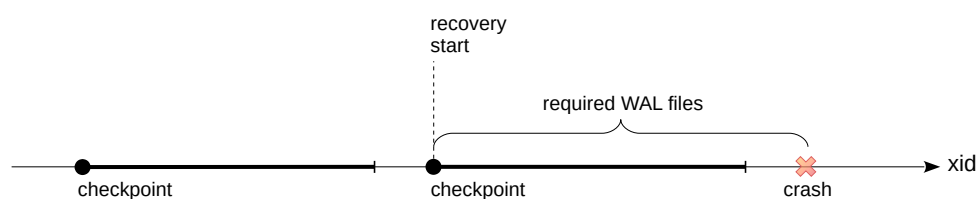

Checkpoint

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 WAL required for recovery

Crash recovery

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



8

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 sequentially reads the WAL 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 have not been logged as committed are considered aborted.

However, logging all changes throughout a server's lifetime and replaying everything from day one 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, which store transaction statuses).

A 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 of all such buffers is complete. It ensures that all data changes up to this point are safe in persistent memory.

In production environments with a large buffer cache, a checkpoint can flush many dirty buffers, so the server spreads this flushing over time to smooth out the I/O load.

When a crash occurs, recovery is started from the last completed checkpoint. Consequently, it is sufficient to store WAL files only as far back as the last completed checkpoint goes.

Crash Recovery Using WAL

Modified table pages exist in the buffer cache but haven't been written to disk yet. During normal shutdown, the server performs a checkpoint to flush all dirty pages to disk. However, we'll simulate a system crash by sending a signal to the postmaster process.

```
student$ sudo kill -QUIT $(sudo head -n 1 /var/lib/postgresql/16/main/postmaster.pid)
```

When the server comes back up, it should begin the recovery. Let's try:

```
student$ sudo pg_ctlcluster 16 main start
```

```
student$ psql arch_wal_overview
```

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

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

All the changes have been recovered.

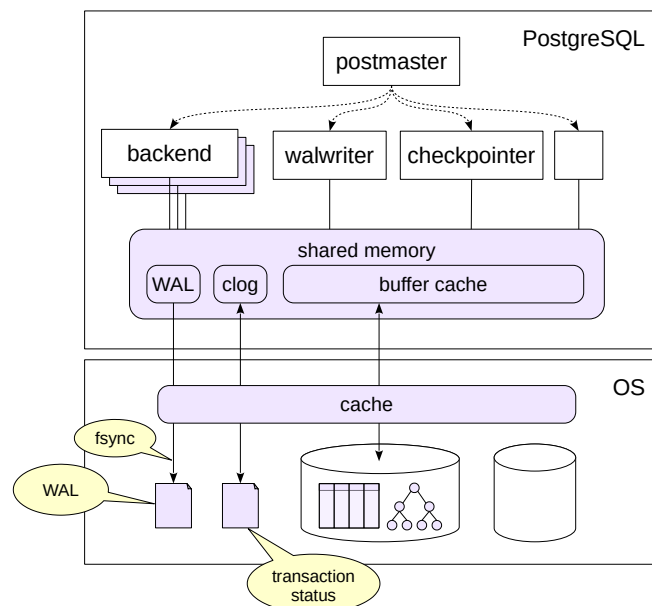
After performing a checkpoint, PostgreSQL automatically deletes WAL files that are no longer necessary for recovery.

Synchronous mode

write on commit
backend

Asynchronous mode

background write
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 WAL is written sequentially (and usually not read until a crash occurs), which is better for basic hard disk drives.

Performance can also be managed via configuration settings. If the records are stored to disk immediately (synchronous mode), this guarantees that the committed transaction will not be lost. But recording to disk is expensive and forces the committing backend process to wait in line. To prevent WAL records from getting “stuck” in the OS cache, PostgreSQL relies on call of the *fsync* function, which forces the data into persistent storage.

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

In fact, both modes work together. WAL records of a long transaction are written asynchronously (to free up WAL buffers). And if a page is getting flushed to disk and the corresponding WAL record is not there yet, it will immediately be flushed in synchronous mode.

Main Processes

WAL writer

Checkpointner

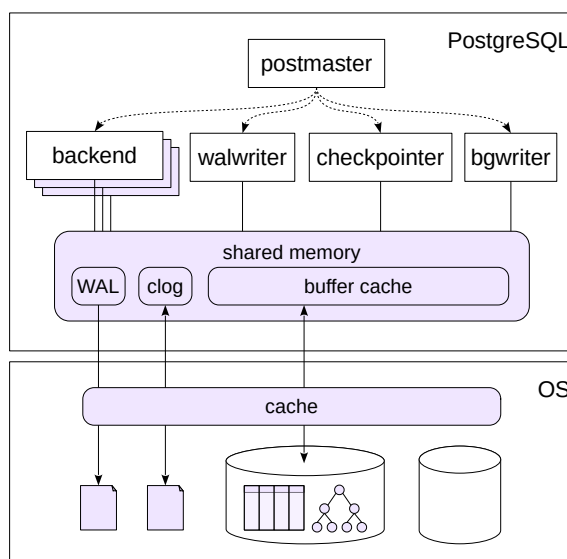
flush all
dirty buffers

Background writer

flush some
dirty buffers

Backends

flush replaced
dirty buffer



11

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 backend that commits the transaction.

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

Third, *bgwriter* (or background writer). It operates similarly to checkpointner, but it only flushes some of the dirty buffers, specifically, those that are most likely to be replaced soon. It frees up buffer space so that when backend selects a buffer to put a new page in, it does not 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 checkpointner and bgwriter), the backend will flush it to disk.

Minimal

guarantees crash recovery

Replica (*default*)

backup

replication: transmit the WAL on another server and replay it there

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 it is supplemented with additional information.

The data stored in the WAL 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 WAL to be used for backup and replication. During replication, WAL records are transmitted to another server and applied there, creating an exact copy (replica) of the original server.
- At the **logical** level, information is added to the WAL that allows decoding “physical” WAL records and forming “logical” records of adding, changing and deleting table rows. This enables logical replication (see corresponding lessons of DEV2 and DBA3 courses for details).

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

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.

2. To stop in fast mode, use the command

```
pg_ctlcluster 16 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.

3. To stop in immediate mode, use the command

```
pg_ctlcluster 16 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 have to restore data consistency using WAL on startup.

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
```

1. Operating System Processes

First, get the postmaster process ID. It is stored in the first line of `postmaster.pid`. This file is located in the data directory and is created every time the server starts.

```
student$ sudo cat /var/lib/postgresql/16/main/postmaster.pid
```

```
32548
/var/lib/postgresql/16/main
1758722835
5432
/var/run/postgresql
localhost
    1342266      32770
ready
```

Now, find all the processes spawned by postmaster:

```
student$ sudo ps -o pid,command --ppid 32548
```

```
    PID COMMAND
  32549 postgres: 16/main: checkpointer
  32550 postgres: 16/main: background writer
  32552 postgres: 16/main: walwriter
  32553 postgres: 16/main: autovacuum launcher
  32554 postgres: 16/main: logical replication launcher
  32597 postgres: 16/main: student student [local] idle
```

The processes that serve the buffer cache and the WAL include:

- checkpointer
- background writer
- walwriter

2. Stop in Fast Mode

To easily separate old messages from new ones, clear the message log before restarting the server. Do not do this on a real production server.

```
student$ sudo -u postgres truncate -cs0 /var/log/postgresql/postgresql-16-main.log
```

```
student$ sudo pg_ctlcluster 16 main stop
```

```
student$ sudo pg_ctlcluster 16 main start
```

Server message log:

```
student$ cat /var/log/postgresql/postgresql-16-main.log
```

```
2025-09-24 17:07:19.702 MSK [32548] LOG:  received fast shutdown request
2025-09-24 17:07:19.718 MSK [32548] LOG:  aborting any active transactions
2025-09-24 17:07:19.719 MSK [32597] student@student FATAL:  terminating connection due to
administrator command
2025-09-24 17:07:19.733 MSK [32548] LOG:  background worker "logical replication
launcher" (PID 32554) exited with exit code 1
2025-09-24 17:07:19.735 MSK [32549] LOG:  shutting down
2025-09-24 17:07:19.751 MSK [32549] LOG:  checkpoint starting: shutdown immediate
2025-09-24 17:07:20.141 MSK [32549] LOG:  checkpoint complete: wrote 5 buffers (0.0%); 0
WAL file(s) added, 0 removed, 0 recycled; write=0.141 s, sync=0.043 s, total=0.406 s;
sync files=4, longest=0.015 s, average=0.011 s; distance=0 kB, estimate=0 kB;
lsn=0/1921710, redo lsn=0/1921710
2025-09-24 17:07:20.149 MSK [32548] LOG:  database system is shut down
2025-09-24 17:07:20.631 MSK [32826] LOG:  starting PostgreSQL 16.10 (Ubuntu
16.10-1.pgdg24.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
13.3.0-6ubuntu2~24.04) 13.3.0, 64-bit
2025-09-24 17:07:20.632 MSK [32826] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2025-09-24 17:07:20.646 MSK [32826] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2025-09-24 17:07:20.681 MSK [32829] LOG:  database system was shut down at 2025-09-24
17:07:20 MSK
2025-09-24 17:07:20.706 MSK [32826] LOG:  database system is ready to accept connections
```

3. Stop in Immediate Mode

```
student$ sudo -u postgres truncate -cs0 /var/log/postgresql/postgresql-16-main.log
```



```
student$ sudo pg_ctlcluster 16 main stop -m immediate --skip-systemctl-redirect
```

```
student$ sudo pg_ctlcluster 16 main start
```

Server message log:

```
student$ cat /var/log/postgresql/postgresql-16-main.log
```

```
2025-09-24 17:07:23.372 MSK [32826] LOG:  received immediate shutdown request
2025-09-24 17:07:23.394 MSK [32826] LOG:  database system is shut down
2025-09-24 17:07:23.898 MSK [32999] LOG:  starting PostgreSQL 16.10 (Ubuntu
16.10-1.pgdg24.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu
13.3.0-6ubuntu2-24.04) 13.3.0, 64-bit
2025-09-24 17:07:23.899 MSK [32999] LOG:  listening on IPv4 address "127.0.0.1", port 5432
2025-09-24 17:07:23.919 MSK [32999] LOG:  listening on Unix socket
"/var/run/postgresql/.s.PGSQL.5432"
2025-09-24 17:07:23.953 MSK [33002] LOG:  database system was interrupted; last known up
at 2025-09-24 17:07:20 MSK
2025-09-24 17:07:34.001 MSK [33002] LOG:  syncing data directory (fsync), elapsed time:
10.02 s, current path: ./base/1/1249_vm
2025-09-24 17:07:44.007 MSK [33002] LOG:  syncing data directory (fsync), elapsed time:
20.02 s, current path: ./base/16385/1247_fsm
2025-09-24 17:07:53.989 MSK [33002] LOG:  syncing data directory (fsync), elapsed time:
30.00 s, current path: ./base/5/2667
2025-09-24 17:08:01.219 MSK [33002] LOG:  database system was not properly shut down;
automatic recovery in progress
2025-09-24 17:08:01.239 MSK [33002] LOG:  invalid record length at 0/1921788: expected at
least 24, got 0
2025-09-24 17:08:01.239 MSK [33002] LOG:  redo is not required
2025-09-24 17:08:01.285 MSK [33000] LOG:  checkpoint starting: end-of-recovery immediate
wait
2025-09-24 17:08:01.388 MSK [33000] LOG:  checkpoint complete: wrote 3 buffers (0.0%); 0
WAL file(s) added, 0 removed, 0 recycled; write=0.029 s, sync=0.015 s, total=0.122 s;
sync files=2, longest=0.008 s, average=0.008 s; distance=0 kB, estimate=0 kB;
lsn=0/1921788, redo lsn=0/1921788
2025-09-24 17:08:01.410 MSK [32999] LOG:  database system is ready to accept connections
```

Before starting to accept connections, the DBMS performed an automatic recovery.