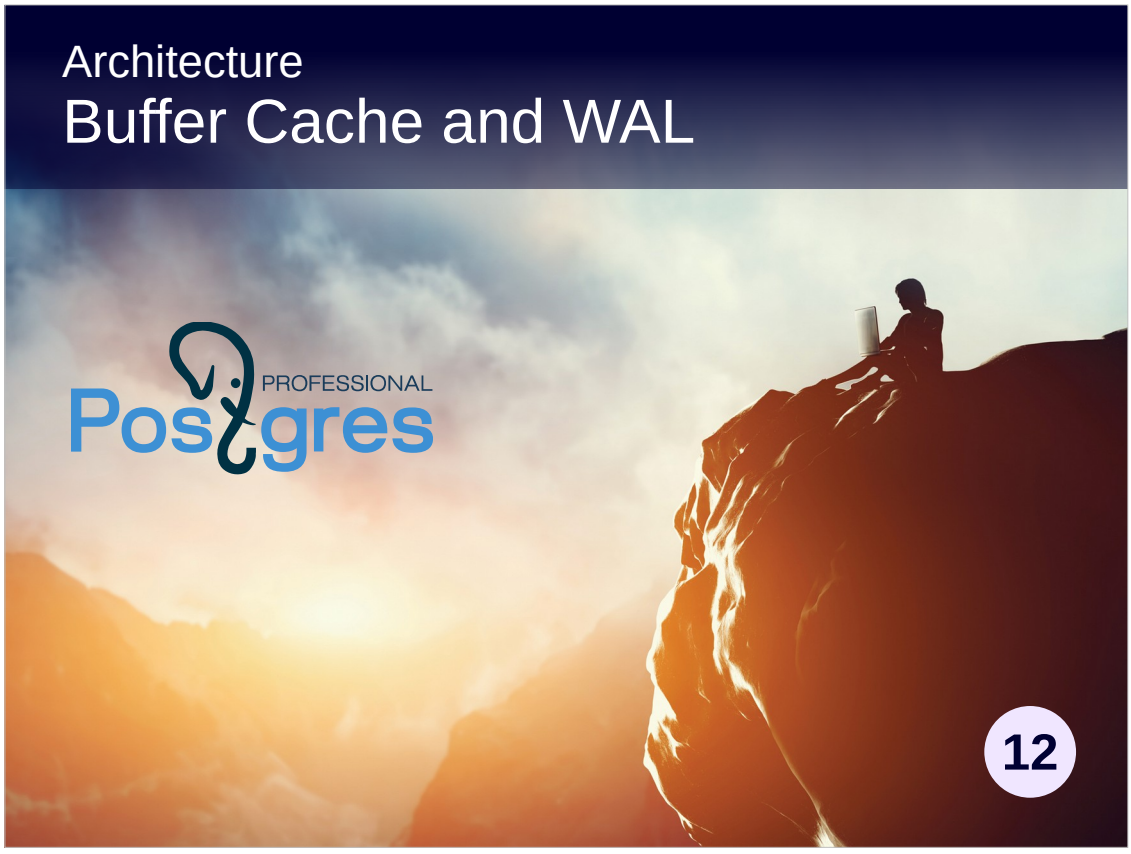


Authors: Egor Rogov, Pavel Luzanov
Translated by Liudmila Mantrova

Contact Us
Please send your feedback to: edu@postgrespro.ru

Agenda

Buffer cache overview

Eviction algorithm

Write-ahead log

Checkpoints

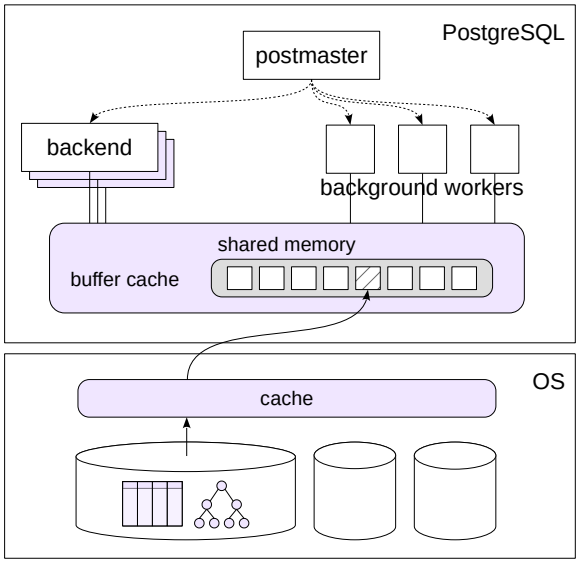
Buffer Cache

An array of buffers

- data page (8 kB)
- additional metadata

Memory locks

- for shared access



Buffer cache is used to even out the speed difference between disk and RAM access. It consists of an array of buffers that store data pages and additional metadata (such as the name of the file and page location within this file).

Page size is usually 8 kB (it can be configured at build time, but there is usually no point in doing it).

All data page access operations go via buffer cache. If any process needs to access a page, it first tries to find it in cache. If the page is not found there, the process asks the operating system to read this page and loads it into buffer cache. (Note that the OS can read the page from disk or find it in its own cache.)

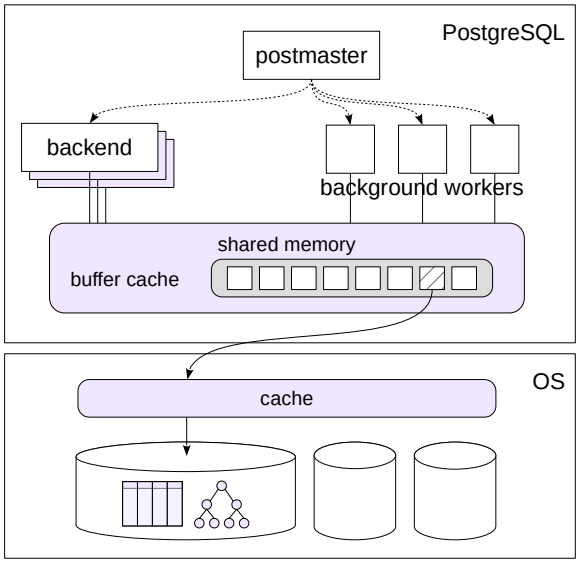
Once the page is in buffer cache, it can be accessed multiple times without any overhead caused by system calls.

However, buffer cache (just like other shared memory structures) is protected by locks to handle concurrent access. Even though locks are implemented quite efficiently, buffer cache access is not as fast as direct access to RAM. So in most cases, the less data is read and updated by a query, the faster this query completes.

Eviction

Eviction of rarely accessed pages

- a “dirty” buffer is flushed to disk
- another page is loaded into the freed space



Buffer cache size is usually not big enough to hold the whole database. It is limited both by the available RAM and by additional overhead that arises when we try to increase it. Sooner or later, buffer cache will become completely full while a new page is being read. In this case, page *eviction* is applied.

An eviction algorithm selects a page in cache that has been recently used less often than others, and replaces it with the new page. If the selected page has been updated, it must be flushed to disk first not to lose these changes (a buffer that contains an updated page is called “dirty”).

This eviction algorithm is called LRU (Least Recently Used). It ensures that actively used data is kept in cache. Such “hot” data is usually not abundant, so having just enough buffer cache allows to significantly reduce the number of system calls (and disk operations).

Buffer Cache and Query Execution

Let's create a database and a table in it:

```
=> CREATE DATABASE arch_wal_overview;

CREATE DATABASE

=> \c arch_wal_overview

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

=> CREATE TABLE t(n integer);

CREATE TABLE
```

Fill the table with some rows:

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

INSERT 0 100000

=> VACUUM ANALYZE t;

VACUUM
```

Now restart the server to clear buffer cache.

```
=> \q

student$ sudo pg_ctlcluster 12 main restart

student$ psql arch_wal_overview
```

Let's compare what happens in two subsequent runs of the same query. In this course, we will not discuss query plans in detail, but we are going to take a look at some of them from time to time. Now we'll run the EXPLAIN ANALYZE command, which executes a query and displays both the query plan and some additional information:

```
=> 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 Time: 2.249 ms
 Execution Time: 8.169 ms
(4 rows)
```

The "Buffers: shared" row shows how buffer cache is used.

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

```
=> 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: 0.030 ms
 Execution Time: 8.132 ms
(4 rows)
```

- hit — the number of buffers that already contained the queried pages.

Note that in addition to faster execution, the second run also shows shorter planning time (that's because system catalog tables are also cached).

Write-Ahead Log (WAL)

Problem: in case of a failure, all data from RAM will be lost if not flushed to disk

WAL

- a stream of information about all performed operations, which allows repeating them if they are lost because of a failure
- WAL entry is written to disk before the modified data

WAL protects

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

WAL does not protect

- temporary and unlogged tables

Buffer cache (like other buffers in RAM) boosts performance, but negatively affects reliability. If the database system fails, all buffer cache contents will be lost. In case of operating system or hardware failures, OS buffer contents will also be cleared (but the operating system copes with it on its own).

To ensure fault tolerance, PostgreSQL uses write-ahead logging. For each operation, a WAL entry is generated; it contains the essential information required to repeat this operation. Such entry must be written to disk (or any other persistent storage) before the actual data update (hence the *"write-ahead" log*).

WAL protects all objects that are handled in RAM, such as tables or indexes, as well as transaction status metadata.

WAL files do not store any data about *temporary tables* (such tables can be accessed only by their owner within the current session or transaction) and *unlogged tables* (such tables are just like regular ones, except that they are not protected by WAL). In case of a failure, such tables are simply cleared. They are used to speed up access to data that can be recovered by other means.

https://postgrespro.com/docs/postgresql/12/wal-intro

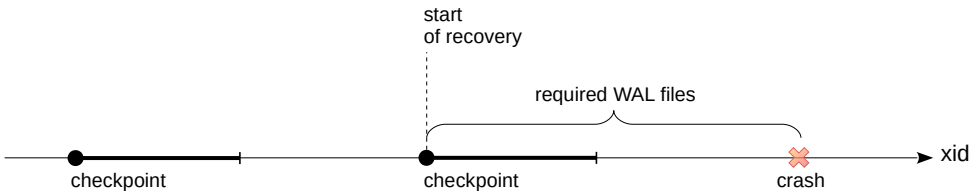
Checkpoints

Periodic flushing of all dirty buffers to disk

- guarantees that all changes before the checkpoint are saved on disk
- reduces the size of WAL required for recovery

Crash recovery

- starts from the latest checkpoint
- replays WAL entries one by one if the corresponding changes are missing



When PostgreSQL server is started after a failure, it enters the recovery mode. At this moment, the data stored on disk is inconsistent: some "hot" pages may not have been flushed yet, even though they got updated before other pages already written to disk.

To restore consistency, PostgreSQL reads WAL and replays all WAL entries one by one if the corresponding changes have not been flushed to disk. Thus, it restarts all transactions and then aborts those that were not registered in WAL as committed.

However, WAL size could become huge during server operation. It is absolutely impossible to store all WAL entries and replay them all after a failure. That's why the database system periodically performs a *checkpoint*: all dirty buffers are forced to disk (including xact buffers that store transaction status metadata). It guarantees that all transaction changes that had happened before the checkpoint are saved on disk.

A checkpoint can take a lot of time, and that's OK. The "point" itself in the sense of a particular moment marks the beginning of the process. But the checkpoint is considered complete only after *all* dirty buffers that were present at that moment are flushed to disk.

Crash recovery starts from the latest checkpoint, which allows PostgreSQL to store only those WAL files that were written after the last completed checkpoint.

Using WAL for Recovery

WAL files are stored in a separate directory; they are not a part of any database. You can access this directory via the file system, or display its contents using the following query:

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

```
           name           |   size   |      modification
--------------------------+----------+------------------------
 000000010000000000000001 | 16777216 | 2021-10-19 17:00:41+03
 000000010000000000000002 | 16777216 | 2021-10-19 17:00:41+03
(2 rows)
```

Once the checkpoint is complete, PostgreSQL can delete redundant files.

Let's make some changes:

```
=> DELETE FROM t;
```

```
DELETE 100000
```

```
=> INSERT INTO t(n) VALUES (0);
```

```
INSERT 0 1
```

All the modified table pages are available in buffer cache, but are not yet flushed to disk. Let's simulate a system failure by stopping the server in the immediate mode. When stopped normally, the server executes the checkpoint to flush all dirty pages to disk, but it doesn't happen in the immediate mode.

```
student$ sudo pg_ctlcluster 12 main stop -m immediate --skip-systemctl-redirect
```

```
student$ sudo pg_ctlcluster 12 main start
```

At the server start, data consistency is restored using WAL. Let's check the result:

```
student$ psql arch_wal_overview
```

```
=> SELECT * FROM t;
```

```
 n
---
 0
(1 row)
```

All the changes have been restored.

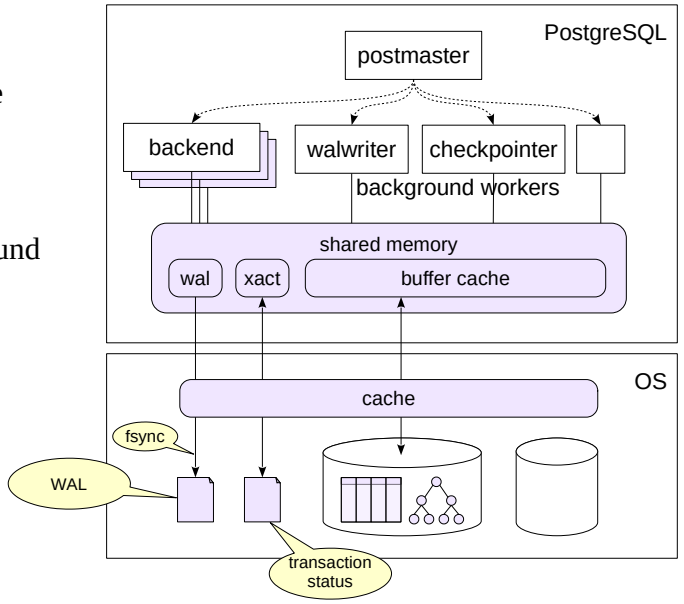
Performance

Synchronous mode

flushed at commit time
by backend

Asynchronous mode

flushed in the background
by walwriter



Using write-ahead logging is more efficient than direct unbuffered disk writes. First of all, the size of WAL entries is smaller than the size of the whole data page. Second, WAL entries are written sequentially (and are usually not read until a failure), so this process can be easily handled even by HDDs.

You can also tune performance by changing some settings. If the data is flushed at once (synchronously), it is guaranteed that a committed transaction won't disappear. But flushing is quite expensive, and the backend performing the commit has to wait for its completion. For WAL entries not to get stuck in the operating system cache, fsync is called: PostgreSQL assumes it is enough to ensure that the data reaches a persistent storage.

That's why PostgreSQL also provides a delayed (asynchronous) commit mode. In this case, commits are gradually flushed to disk by the walwriter background worker (with some delay). It is less reliable, but provides better performance in return. In case of a failure, a consistent recovery is still guaranteed, but some of the recently committed transactions may be lost.

Summary

Buffer cache boosts performance by reducing the number of disk access operations

Reliability is ensured by write-ahead logging

WAL size is reduced by using checkpoints

WAL is a convenient mechanism used in many scenarios

- to perform recovery after a failure
- during backup
- for replicating data between servers

Practice

1. Check how buffer cache is used when updating a single row in a regular table and in a *temporary* table. Try to explain the difference.
2. Create an *unlogged* table and insert several rows into it. Simulate a system failure by stopping the server in the immediate mode as shown in the demo.
   Start the server and check the table state.
   In the server log, find the entries related to recovery.

Task 1. Temporary tables look just like regular ones, but their lifetime is limited to the current session. Likewise, such tables are visible only in the current session.

Use the following command as shown in the demo:

```
EXPLAIN (analyze, buffers, costs off, timing off)
```

Task 2. To stop the server in the immediate mode, run:

```
sudo pg_ctlcluster 12 main stop -m immediate –skip-systemctl-redirect
```

The --skip-systemctl-redirect parameter is required because PostgreSQL has been installed on Ubuntu from a package. The server is managed by the pg_ctlcluster command that calls the systemctl utility, and the specified mode gets lost by the time pg_ctl is started. This parameter allows us to do without systemctl and pass the command to pg_ctl directly.

Task 1. Using Cache with Regular and Temporary Tables

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

```
CREATE DATABASE
```

```
=> \c arch_wal_overview
```

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

Let's create a regular table with one row...

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

```
CREATE TABLE
```

```
=> INSERT INTO t(n) VALUES (1);
```

```
INSERT 0 1
```

...and a temporary table that looks exactly the same.

```
=> CREATE TEMPORARY TABLE tt(n integer);
```

```
CREATE TABLE
```

```
=> INSERT INTO tt(n) VALUES (1);
```

```
INSERT 0 1
```

Update the row in the regular table:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
UPDATE t SET n = n + 1;
```

```
                 QUERY PLAN
--------------------------------------------
 Update on t (actual rows=0 loops=1)
   Buffers: shared hit=2
   ->  Seq Scan on t (actual rows=1 loops=1)
         Buffers: shared hit=1
 Planning Time: 0.158 ms
 Execution Time: 0.075 ms
(6 rows)
```

- The page was found in buffer cache ("shared hit=1") during the table scan ("Seq Scan").
- It was required to read two pages while performing the update ("shared hit=2"). The second page belongs to a visibility map (we are going to discuss visibility maps in the "Data Organization" module).

Let's update the row in our temporary table:

```
=> EXPLAIN (analyze, buffers, costs off, timing off)
UPDATE tt SET n = n + 1;
```

```
                  QUERY PLAN
---------------------------------------------
 Update on tt (actual rows=0 loops=1)
   Buffers: local hit=2
   ->  Seq Scan on tt (actual rows=1 loops=1)
         Buffers: local hit=1
 Planning Time: 0.062 ms
 Execution Time: 0.044 ms
(6 rows)
```

Instead of the shared buffer cache, which is located in the server's shared memory, this operation uses local cache of the current ("local") session.

Task 2. Unlogged Tables and Failures

Let's create an unlogged table:

```
=> CREATE UNLOGGED TABLE u(s text);
```

```
CREATE TABLE
```

```
=> INSERT INTO u VALUES ('Hello!');
```

```
INSERT 0 1
```

Simulate a failure:

```
student$ sudo pg_ctlcluster 12 main stop -m immediate --skip-systemctl-redirect
```

Start the server:

```
student$ sudo pg_ctlcluster 12 main start
```

```
student$ psql arch_wal_overview
```

```
=> SELECT * FROM u;
```

```
 s
---
(0 rows)
```

The table is here, but it is empty. The contents of unlogged tables is not restored after a failure; such tables are simply cleared instead.

Let’s check the server log:

```
student$ tail -n 5 /var/log/postgresql/postgresql-12-main.log
```

```
2021-10-19 17:05:08.739 MSK [35603] LOG:  database system was not properly shut down; automatic recovery in progress
2021-10-19 17:05:08.742 MSK [35603] LOG:  redo starts at 0/382470A0
2021-10-19 17:05:08.743 MSK [35603] LOG:  invalid record length at 0/382671E0: wanted 24, got 0
2021-10-19 17:05:08.743 MSK [35603] LOG:  redo done at 0/382671B8
2021-10-19 17:05:08.781 MSK [35602] LOG:  database system is ready to accept connections
```

It shows that an abnormal termination was detected at the server start, and an automatic recovery was performed.