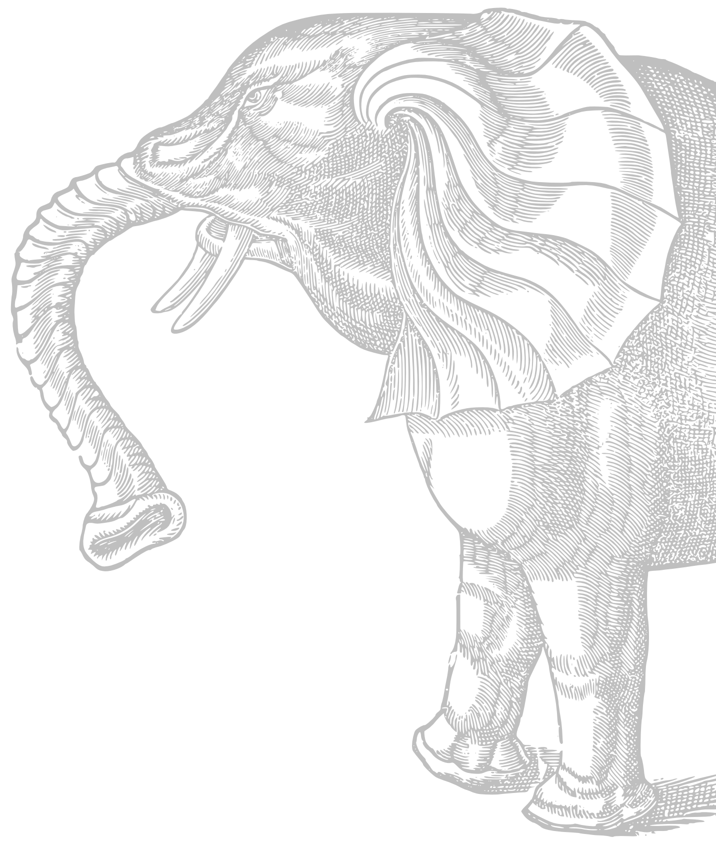
Egor Rogov

PostgreSQL 14 Internals



This is Part I of the book.
The other parts will follow soon:

II. Buffer Cache and WAL

III. Locks

IV. Query Execution

V. Index Types

Postgres Professional
Moscow, 2022

The elephant on the cover is a fragment of an illustration from Edward Topsell's *The History of Four-footed Beasts and Serpents*, published in London in 1658

PostgreSQL 14 Internals
by Egor Rogov

Translated from Russian by Liudmila Mantrova

This book in PDF is available at postgrespro.com/community/books/internals

Contents at a Glance

Table of Contents

About This Book

Books are not made to be believed, but to be subjected to inquiry.
— Umberto Eco, *The Name of the Rose*

For Whom Is This Book?

This book is for those who will not settle for a black-box approach when working with a database. If you are eager to learn, prefer not to take expert advice for granted, and would like to figure out everything yourself, follow along.

I assume that the reader has already tried using PostgreSQL and has at least some general understanding of how it works. Entry-level users may find the text a bit difficult. For example, I will not tell anything about how to install the server, enter psql commands, or set configuration parameters.

I hope that the book will also be useful for those who are familiar with another database system, but switch over to PostgreSQL and would like to understand how they differ. A book like this would have saved me a lot of time several years ago. And that's exactly why I finally wrote it.

What This Book Will Not Provide

This book is not a collection of recipes. You cannot find ready-made solutions for every occasion, but if you understand inner mechanisms of a complex system, you will be able to analyze and critically evaluate other people's experience and come to your own conclusions. For this reason, I explain such details that may at first seem to be of no practical use.

But this book is not a tutorial either. While delving deeply into some fields (in which I am more interested myself), it may say nothing at all about the other.

By no means is this book a reference. I tried to be precise, but I did not aim at replacing documentation, so I could easily leave out some details that I considered insignificant. In any unclear situation read the documentation.

This book will not teach you how to develop the PostgreSQL core. I do not expect any knowledge of the C language, as this book is mainly intended for database administrators and application developers. But I do provide multiple references to the source code, which can give you as many details as you like, and even more.

What This Book Does Provide

In the introductory chapter, I briefly touch upon the main database concepts that will serve as the foundation for all the further narration. I do not expect you to get much new information from this chapter but still include it to complete the big picture. Besides, this overview can be found useful by those who are migrating from other database systems.

Part I is devoted to questions of data consistency and isolation. I first cover them from the user's perspective (you will learn which isolation levels are available and what are the implications) and then dwell on their internals. For this purpose, I have to explain implementation details of multiversion concurrency control and snapshot isolation, paying special attention to cleanup of outdated row versions.

Part II describes buffer cache and WAL, which is used to restore data consistency after a failure.

Part III goes into details about the structure and usage of various types of locks: lightweight locks for RAM, heavyweight locks for relations, and row-level locks.

Part IV explains how the server plans and executes SQL queries. I will tell you which data access methods are available, which join methods can be used, and how the collected statistics are applied.

Part V extends the discussion of indexes from the already covered B-trees to other access methods. I will explain some general principles of extensibility that define the boundaries between the core of the indexing system, index access methods, and data types (which will bring us to the concept of operator classes), and then elaborate on each of the available methods.

PostgreSQL includes multiple “introspective” extensions, which are not used in routine work, but give us an opportunity to peek into the server’s internal behavior. This book uses quite a few of them. Apart from letting us explore the server internals, these extensions can also facilitate troubleshooting in complex usage scenarios.

Conventions

I tried to write this book in a way that would allow reading it page by page, from start to finish. But it is hardly possible to uncover all the truth at once, so I had to get back to one and the same topic several times. Writing that “it will be considered later” over and over again would inevitably make the text much longer, that’s why in such cases I simply put the page number in the margin to refer you to further discussion. A similar number pointing backwards will take you to the page where something has been already said on the subject. *p. 11*

Both the text and all the code examples in this book apply to PostgreSQL 14. Next to some paragraphs, you can see a version number in the page margin. It means that the provided information is relevant starting from the indicated PostgreSQL version, while all the previous versions either did not have the described feature at all, or used a different implementation. Such notes can be useful for those who have not upgraded their systems to the latest release yet. v. 14

I also use the margins to show the default values of the discussed parameters. The names of both regular and storage parameters are printed in italics: *work_mem*. 4MB

In footnotes, I provide multiple links to various sources of information. There are several of them, but first and foremost, I list the PostgreSQL documentation[1], which is a wellspring of knowledge. Being an essential part of the project, it is always kept up-to-date by PostgreSQL developers themselves. However, the primary reference is definitely the source code[2]. It is amazing how many answers you can find by simply reading comments and browsing through README files, even if you do not know C. Sometimes I also refer to commitfest[3] entries: you can always trace the

[1] postgresql.org/docs/14/index.html
[2] git.postgresql.org/gitweb/?p=postgresql.git;a=summary.
[3] commitfest.postgresql.org.

history of all changes and understand the logic of decisions taken by developers if you read the related discussions in the psql-hackers mailing list, but it requires digging through piles of emails.

> Side notes that can lead the discussion astray (which I could not help but include into the book) are printed like this, so they can be easily skipped.

Naturally, the book contains multiple code examples, mainly in SQL. The code is provided with the prompt =>; the server response follows if necessary:

```
=> SELECT now();
              now
-------------------------------
 2022-07-10 18:39:19.763435+03
(1 row)
```

If you carefully repeat all the provided commands in PostgreSQL 14, you should get exactly the same results (down to transaction IDs and other inessential details). Anyway, all the code examples in this book have been generated by the script containing exactly these commands.

When it is required to illustrate concurrent execution of several transactions, the code run in another session is indented and marked off by a vertical line.

```
|   => SHOW server_version;
|    server_version
|   ----------------
|    14.4
|   (1 row)
```

To try out such commands (which is useful for self-study, just like any experimentation), it is convenient to open two psql terminals.

The names of commands and various database objects (such as tables and columns, functions, or extensions) are highlighted in the text using a sans-serif font: UPDATE, pg_class.

If a utility is called from the operating system, it is shown with a prompt that ends with $:

```
postgres$ whoami
postgres
```

I use Linux, but without any technicalities; having some basic understanding of this operating system will be enough.

Acknowledgments

It is impossible to write a book alone, and now I have an excellent opportunity to thank good people.

I am very grateful to Pavel Luzanov who found the right moment and offered me to start doing something really worthwhile.

I am obliged to Postgres Professional for the opportunity to work on this book beyond my free time. But there are actual people behind the company, so I would like to express my gratitude to Oleg Bartunov for sharing ideas and infinite energy, and to Ivan Panchenko for thorough support and $\LaTeX$.

I would like to thank my colleagues from the education team for the creative atmosphere and discussions that shaped the scope and format of our training courses, which also got reflected in the book. Special thanks to Pavel Tolmachev for his meticulous review of the drafts.

Many chapters of this book were first published as articles in the Habr blog[1], and I am grateful to the readers for their comments and feedback. It showed the importance of this work, highlighted some gaps in my knowledge, and helped me improve the text.

I would also like to thank Liudmila Mantrova who has put much effort into polishing this book's language. If you do not stumble over every other sentence, the credit goes to her. Besides, Liudmila took the trouble to translate this book into English, for which I am very grateful too.

[1] habr.com/en/company/postgrespro/blog

I do not provide any names, but each function or feature mentioned in this book has required years of work done by particular people. I admire PostgreSQL developers, and I am very glad to have the honor of calling many of them my colleagues.

1 Introduction

1.1. Data Organization

Databases

PostgreSQL is a program that belongs to the class of database management systems. When this program is running, we call it a PostgreSQL *server*, or *instance*.

Data managed by PostgreSQL is stored in databases[1]. A single PostgreSQL instance can serve several databases at a time; together they are called a *database cluster*.

To be able to use the cluster, you must first *initialize*[2] (create) it. The directory that contains all the files related to the cluster is usually called PGDATA, after the name of the environment variable pointing to this directory.

> Installations from pre-built packages can add their own "abstraction layers" over the regular PostgreSQL mechanism by explicitly setting all the parameters required by utilities. In this case, the database server runs as an operating system service, and you may never come across the PGDATA variable directly. But the term itself is well-established, so I am going to use it.

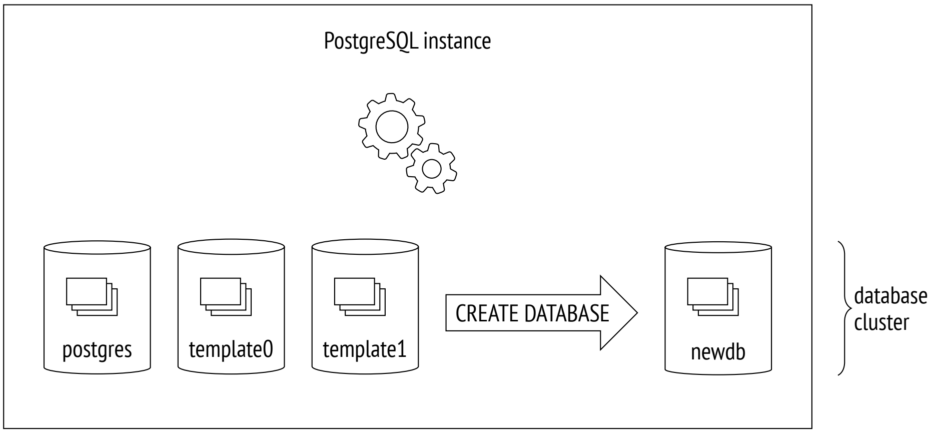
After cluster initialization, PGDATA contains three identical databases:

template0 is used for cases like restoring data from a logical backup or creating a database with a different encoding; it must never be modified.

template1 serves as a template for all the other databases that a user can create in the cluster.

[1] postgresql.org/docs/14/managing-databases.html
[2] postgresql.org/docs/14/app-initdb.html

postgres is a regular database that you can use at your discretion.



System Catalog

Metadata of all cluster objects (such as tables, indexes, data types, or functions) is stored in tables that belong to the *system catalog*[1]. Each database has its own set of tables (and views) that describe the objects of this database. Several system catalog tables are common to the whole cluster; they do not belong to any particular database (technically, a dummy database with a zero ID is used), but can be accessed from all of them.

The system catalog can be viewed using regular SQL queries, while all modifications in it are performed by DDL commands. The psql client also offers a whole range of commands that display the contents of the system catalog.

Names of all system catalog tables begin with pg_, like in pg_database. Column names start with a three-letter prefix that usually corresponds to the table name, like in datname.

In all system catalog tables, the column declared as the primary key is called oid (object identifier); its type, which is also called oid, is a 32-bit integer.

[1] postgresql.org/docs/14/catalogs.html

The implementation of oid object identifiers is virtually the same as that of sequences, but it appeared in PostgreSQL much earlier. What makes it special is that the generated unique IDs issued by a common counter are used in different tables of the system catalog. When an assigned ID exceeds the maximum value, the counter is reset. To ensure that all values in a particular table are unique, the next issued oid is checked by the unique index; if it is already used in this table, the counter is incremented, and the check is repeated[1].

Schemas

Schemas[2] are namespaces that store all objects of a database. Apart from user schemas, PostgreSQL offers several predefined ones:

public is the default schema for user objects unless other settings are specified.

pg_catalog is used for system catalog tables.

information_schema provides an alternative view for the system catalog as defined by the SQL standard.

pg_toast is used for objects related to TOAST. *p. 24*

pg_temp comprises temporary tables. Although different users create temporary tables in different schemas called pg_temp_*N*, everyone refers to their objects using the pg_temp alias.

Each schema is confined to a particular database, and all database objects belong to this or that schema.

If the schema is not specified explicitly when an object is accessed, PostgreSQL selects the first suitable schema from the *search path*. The search path is based on the value of the *search_path* parameter, which is implicitly extended with pg_catalog and (if necessary) pg_temp schemas. It means that different schemas can contain objects with the same names.

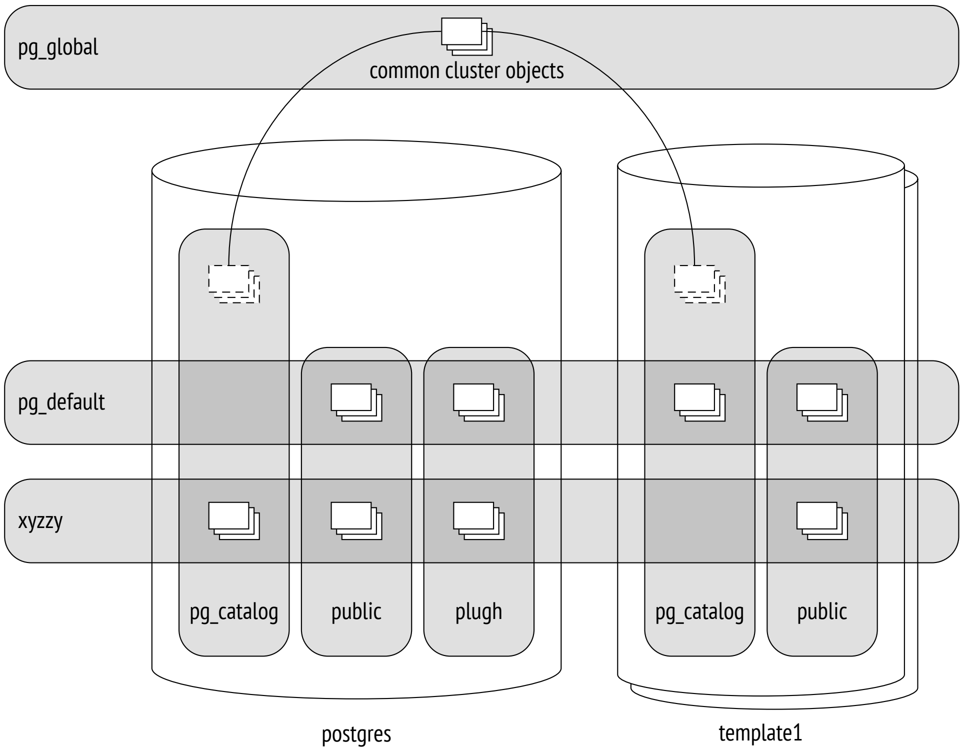
[1] backend/catalog/catalog.c, GetNewOidWithIndex function
[2] postgresql.org/docs/14/ddl-schemas.html

Tablespaces

Unlike databases and schemas, which determine logical distribution of objects, *tablespaces* define physical data layout. A tablespace is virtually a directory in a file system. You can distribute your data between tablespaces in such a way that archive data is stored on slow disks, while the data that is being actively updated goes to fast disks.

One and the same tablespace can be used by different databases, and each database can store data in several tablespaces. It means that logical structure and physical data layout do not depend on each other.

Each database has the so-called *default tablespace*. All database objects are created in this tablespace unless another location is specified. System catalog objects related to this database are also stored there.



During cluster initialization, two tablespaces are created:

pg_default is located in the PGDATA/base directory; it is used as the default tablespace unless another tablespace is explicitly selected for this purpose.

pg_global is located in the PGDATA/global directory; it stores system catalog objects that are common to the whole cluster.

When creating a custom tablespace, you can specify any directory; PostgreSQL will create a symbolic link to this location in the PGDATA/pg_tblspc directory. In fact, all paths used by PostgreSQL are relative to the PGDATA directory, which allows you to move it to a different location (provided that you have stopped the server, of course).

The illustration on the previous page puts together databases, schemas, and tablespaces. Here the postgres database uses tablespace xyzzy as the default one, whereas the template1 database uses pg_default. Various database objects are shown at the intersections of tablespaces and schemas.

Relations

For all of their differences, *tables* and *indexes*—the most important database objects—have one thing in common: they consist of rows. This point is quite self-evident when we think of tables, but it is equally true for B-tree nodes, which contain indexed values and references to other nodes or table rows.

Some other objects also have the same structure; for example, *sequences* (virtually one-row tables) and *materialized views* (which can be thought of as tables that "keep" the corresponding queries). Besides, there are regular *views*, which do not store any data but otherwise are very similar to tables.

In PostgreSQL, all these objects are referred to by the generic term *relation*.

In my opinion, it is not a happy term because it confuses database tables with "genuine" relations defined in the relational theory. Here we can feel the academic legacy of the project and the inclination of its founder, Michael Stonebraker, to see everything as a relation. In one of his works, he even introduced the concept of an "ordered relation" to denote a table in which the order of rows is defined by an index.

The system catalog table for relations was originally called pg_relation, but following the object orientation trend, it was soon renamed to pg_class, which we are now used to. Its columns still have the REL prefix though.

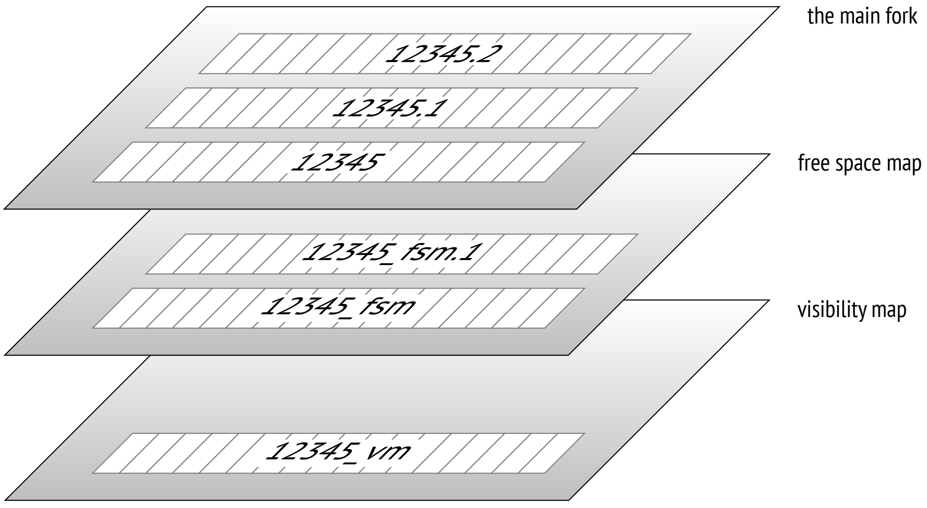
Files and Forks

All information associated with a relation is stored in several different *forks*[1], each containing data of a particular type.

At first, a fork is represented by a single *file*. Its filename consists of a numeric ID (oid), which can be extended by a suffix that corresponds to the fork's type.

The file grows over time, and when its size reaches 1 GB, another file of this fork is created (such files are sometimes called *segments*). The sequence number of the segment is added to the end of its filename.

The file size limit of 1 GB was historically established to support various file systems that could not handle large files. You can change this limit when building PostgreSQL (./configure --with-segsize).



[1] postgresql.org/docs/14/storage-file-layout.html

Thus, a single relation is represented on disk by several files. Even a small table without indexes will have at least three files, by the number of mandatory forks.

Each tablespace directory (except for pg_global) contains separate subdirectories for particular databases. All files of the objects belonging to the same tablespace and database are located in the same subdirectory. You must take it into account because too many files in a single directory may not be handled well by file systems.

There are several standard types of forks.

The main fork represents actual data: table rows or index rows. This fork is available for any relations (except for views, which contain no data).

Files of the main fork are named by their numeric IDs, which are stored as relfilenode values in the pg_class table.

Let's take a look at the path to a file that belongs to a table created in the pg_default tablespace:

```
=> CREATE UNLOGGED TABLE t(
  a integer,
  b numeric,
  c text,
  d json
);
=> INSERT INTO t VALUES (1, 2.0, 'foo', '{}');
=> SELECT pg_relation_filepath('t');
 pg_relation_filepath
----------------------
 base/16384/16385
(1 row)
```

The base directory corresponds to the pg_default tablespace, the next subdirectory is used for the database, and it is here that we find the file we are looking for:

```
=> SELECT oid FROM pg_database WHERE datname = 'internals';
  oid
-------
 16384
(1 row)
=> SELECT relfilenode
```

```
FROM pg_class
WHERE relname = 't';
 relfilenode
-------------
       16385
(1 row)
```

Here is the corresponding file in the file system:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385');
 size
------
 8192
(1 row)
```

The initialization fork[1] is available only for unlogged tables (created with the UNLOGGED clause) and their indexes. Such objects are the same as regular ones, except that any actions performed on them are not written into the write-ahead log. It makes these operations considerably faster, but you will not be able to restore consistent data in case of a failure. Therefore, PostgreSQL simply deletes all forks of such objects during recovery and overwrites the main fork with the initialization fork, thus creating a dummy file.

The t table is created as unlogged, so the initialization fork is present. It has the same name as the main fork, but with the _init suffix:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_init');
 size
------
    0
(1 row)
```

The free space map[2] keeps track of available space within pages. Its volume changes all the time, growing after vacuuming and getting smaller when new row versions appear. The free space map is used to quickly find a page that can accommodate new data being inserted.

backend/storage/freespace/README

All files related to the free space map have the _fsm suffix. Initially, no such files are created; they appear only when necessary. The easiest way to get them is to vacuum a table: *p. 119*

```
=> VACUUM t;
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_fsm');
 size
-------
 24576
(1 row)
```

To speed up search, the free space map is organized as a tree; it takes at least three pages (hence its file size for an almost empty table).

The free space map is provided for both tables and indexes. But since an index row cannot be added into an arbitrary page (for example, B-trees define the place of insertion by the sort order), PostgreSQL tracks only those pages that have been fully emptied and can be reused in the index structure.

The visibility map[1] can quickly show whether a page needs to be vacuumed or frozen. For this purpose, it provides two bits for each table page.

The first bit is set for pages that contain only up-to-date row versions. Vacuum skips such pages because there is nothing to clean up. Besides, when a transaction tries to read a row from such a page, there is no point in checking its visibility, so an index-only scan can be used. *p. 118*

The second bit is set for pages that contain only frozen row versions. I will use the term *freeze map* to refer to this part of the fork. v. 9.6 *p. 137*

Files of the visibility map have the _vm suffix. They are usually the smallest ones:

```
=> SELECT size
FROM pg_stat_file('/usr/local/pgsql/data/base/16384/16385_vm');
 size
------
 8192
(1 row)
```

The visibility map is provided for tables, but not for indexes. *p. 78*

Pages

p. 64 To facilitate I/O, all files are logically split into *pages* (or *blocks*), which represent the minimum amount of data that can be read or written. Consequently, many internal PostgreSQL algorithms are tuned for page processing.

The page size is usually 8 kB. It can be configured to some extent (up to 32 kB), but only at build time (./configure --with-blocksize), and nobody usually does it. Once built and launched, the instance can work only with pages of the same size; it is impossible to create tablespaces that support different page sizes.

Regardless of the fork they belong to, all the files are handled by the server in roughly the same way. Pages are first moved to the buffer cache (where they can be read and updated by processes) and then flushed back to disk as required.

TOAST

Each row must fit a single page: there is no way to continue a row on the next page. To store long rows, PostgreSQL uses a special mechanism called TOAST[1] (The Oversized Attributes Storage Technique).

Toast implies several strategies. You can move long attribute values into a separate service table, having sliced them into smaller "toasts." Another option is to compress a long value in such a way that the row fits the page. Or you can do both: first compress the value, and then slice and move it.

If the main table contains potentially long attributes, a separate TOAST table is created for it right away, one for all the attributes. For example, if a table has a column of the numeric or text type, a TOAST table will be created even if this column will never store any long values.

For indexes, the TOAST mechanism can offer only compression; moving long attributes into a separate table is not supported. It limits the size of the keys that can be indexed (the actual implementation depends on a particular operator class).

[1] postgresql.org/docs/14/storage-toast.html
include/access/heaptoast.h

By default, the TOAST strategy is selected based on the data type of a column. The easiest way to review the used strategies is to run the \d+ command in psql, but I will query the system catalog to get an uncluttered output:

```
=> SELECT attname, atttypid::regtype,
  CASE attstorage
    WHEN 'p' THEN 'plain'
    WHEN 'e' THEN 'external'
    WHEN 'm' THEN 'main'
    WHEN 'x' THEN 'extended'
  END AS storage
FROM pg_attribute
WHERE attrelid = 't'::regclass AND attnum > 0;
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | extended
(4 rows)
```

PostgreSQL supports the following strategies:

plain means that TOAST is not used (this strategy is applied to data types that are known to be "short," such as the integer type).

extended allows both compressing attributes and storing them in a separate TOAST table.

external implies that long attributes are stored in the TOAST table in an uncompressed state.

main requires long attributes to be compressed first; they will be moved to the TOAST table only if compression did not help.

In general terms, the algorithm looks as follows[1]. PostgreSQL aims at having at least four rows in a page. So if the size of the row exceeds one fourth of the page, excluding the header (for a standard-size page it is about 2000 bytes), we must apply the TOAST mechanism to some of the values. Following the workflow described below, we stop as soon as the row length does not exceed the threshold anymore:

[1] backend/access/heap/heaptoast.c

1. First of all, we go through attributes with external and extended strategies, starting from the longest ones. Extended attributes get compressed, and if the resulting value (on its own, without taking other attributes into account) exceeds one fourth of the page, it is moved to the TOAST table right away. External attributes are handled in the same way, except that the compression stage is skipped.

2. If the row still does not fit the page after the first pass, we move the remaining attributes that use external or extended strategies into the TOAST table, one by one.

3. If it did not help either, we try to compress the attributes that use the main strategy, keeping them in the table page.

4. If the row is still not short enough, the main attributes are moved into the TOAST table.

v. 11 The threshold value is 2000 bytes, but it can be redefined at the table level using the *toast_tuple_target* storage parameter.

It may sometimes be useful to change the default strategy for some of the columns. If it is known in advance that the data in a particular column cannot be compressed (for example, the column stores JPEG images), you can set the external strategy for this column; it allows you to avoid futile attempts to compress the data. The strategy can be changed as follows:

```
=> ALTER TABLE t ALTER COLUMN d SET STORAGE external;
```

If we repeat the query, we will get the following result:

```
 attname | atttypid | storage
---------+----------+----------
 a       | integer  | plain
 b       | numeric  | main
 c       | text     | extended
 d       | json     | external
(4 rows)
```

TOAST tables reside in a separate schema called pg_toast; it is not included into the search path, so TOAST tables are usually hidden. For temporary tables, pg_toast_temp_*N* schemas are used, by analogy with pg_temp_*N*.

Let's take a look at the inner mechanics of the process. Suppose table t contains three potentially long attributes; it means that there must be a corresponding TOAST table. Here it is:

```
=> SELECT relnamespace::regnamespace, relname
FROM pg_class
WHERE oid = (
  SELECT reltoastrelid
  FROM pg_class WHERE relname = 't'
);
 relnamespace |    relname
--------------+----------------
 pg_toast     | pg_toast_16385
(1 row)
```

```
=> \d+ pg_toast.pg_toast_16385
TOAST table "pg_toast.pg_toast_16385"
   Column    |  Type   | Storage
------------+---------+---------
 chunk_id   | oid     | plain
 chunk_seq  | integer | plain
 chunk_data | bytea   | plain
Owning table: "public.t"
Indexes:
    "pg_toast_16385_index" PRIMARY KEY, btree (chunk_id, chunk_seq)
Access method: heap
```

It is only logical that the resulting chunks of the toasted row use the plain strategy: there is no second-level TOAST.

Apart from the TOAST table itself, PostgreSQL creates the corresponding index in the same schema. This index is *always* used to access TOAST chunks. The name of the index is displayed in the output, but you can also view it by running the following query:

```
=> SELECT indexrelid::regclass FROM pg_index
WHERE indrelid = (
  SELECT oid
  FROM pg_class WHERE relname = 'pg_toast_16385'
);
          indexrelid
-------------------------------
 pg_toast.pg_toast_16385_index
(1 row)
```

```
=> \d pg_toast.pg_toast_16385_index
Unlogged index "pg_toast.pg_toast_16385_index"
  Column   |  Type   | Key? | Definition
-----------+---------+------+------------
 chunk_id  | oid     | yes  | chunk_id
 chunk_seq | integer | yes  | chunk_seq
primary key, btree, for table "pg_toast.pg_toast_16385"
```

Thus, a TOAST table increases the minimum number of fork files used by the table up to eight: three for the main table, three for the TOAST table, and two for the TOAST index.

Column c uses the extended strategy, so its values will be compressed:

```
=> UPDATE t SET c = repeat('A',5000);
=> SELECT * FROM pg_toast.pg_toast_16385;
 chunk_id | chunk_seq | chunk_data
----------+-----------+------------
(0 rows)
```

The TOAST table is empty: repeated symbols have been compressed by the LZ algorithm, so the value fits the table page.

And now let's construct this value of random symbols:

```
=> UPDATE t SET c = (
  SELECT string_agg( chr(trunc(65+random()*26)::integer), '')
  FROM generate_series(1,5000)
)
RETURNING left(c,10) || '...' || right(c,10);
        ?column?
-------------------------
 TNBAZHGYYR...LHBFWWTUHN
(1 row)
UPDATE 1
```

This sequence cannot be compressed, so it gets into the TOAST table:

```
=> SELECT chunk_id,
  chunk_seq,
  length(chunk_data),
  left(encode(chunk_data,'escape')::text, 10) || '...' ||
  right(encode(chunk_data,'escape')::text, 10)
FROM pg_toast.pg_toast_16385;
```

```
 chunk_id | chunk_seq | length |        ?column?
----------+-----------+--------+------------------------
    16390 |         0 |   1996 | TNBAZHGYYR...HAFBUNMXMY
    16390 |         1 |   1996 | XTIKSELYUD...CKZEQYANGI
    16390 |         2 |   1008 | NOMSLWWXQM...LHBFWWTUHN
(3 rows)
```

We can see that the characters are sliced into chunks. The chunk size is selected in such a way that the page of the TOAST table can accommodate four rows. This value varies a little from version to version depending on the size of the page header.

When a long attribute is accessed, PostgreSQL automatically restores the original value and returns it to the client; it all happens seamlessly for the application. If long attributes do not participate in the query, the TOAST table will not be read at all. It is one of the reasons why you should avoid using the asterisk in production solutions.

If the client queries one of the first chunks of a long value, PostgreSQL will read the required chunks only, even if the value has been compressed. v. 13

Nevertheless, data compression and slicing require a lot of resources; the same goes for restoring the original values. That's why it is not a good idea to keep bulky data in PostgreSQL, especially if this data is being actively used and does not require transactional logic (like scanned accounting documents). A potentially better alternative is to store such data in the file system, keeping in the database only the names of the corresponding files. But then the database system cannot guarantee data consistency.

1.2. Processes and Memory

A PostgreSQL server instance consists of several interacting processes.

The first process launched at the server start is postgres, which is traditionally called postmaster. It spawns all the other processes (Unix-like systems use the fork system call for this purpose) and supervises them: if any process fails, postmaster restarts it (or the whole server if there is a risk that the shared data has been damaged).

Because of its simplicity, the process model has been used in PostgreSQL from the very beginning, and ever since there have been unending discussions about switching over to threads.

The current model has several drawbacks: static shared memory allocation does not allow resizing structures like buffer cache on the fly; parallel algorithms are hard to implement and less efficient than they could be; sessions are tightly bound to processes. Using threads sounds promising, even though it involves some challenges related to isolation, OS compatibility, and resource management. However, their implementation would require a radical code overhaul and years of work, so conservative views prevail for now: no such changes are expected in the near future.

Server operation is maintained by background processes. Here are the main ones:

startup restores the system after a failure.

p. 121 **autovacuum** removes stale data from tables and indexes.

wal writer writes WAL entries to disk.

checkpointer executes checkpoints.

writer flushes dirty pages to disk.

stats collector collects usage statistics for the instance.

wal sender sends WAL entries to a replica.

wal receiver gets WAL entries on a replica.

Some of these processes are terminated once the task is complete, others run in the background all the time, and some can be switched off.

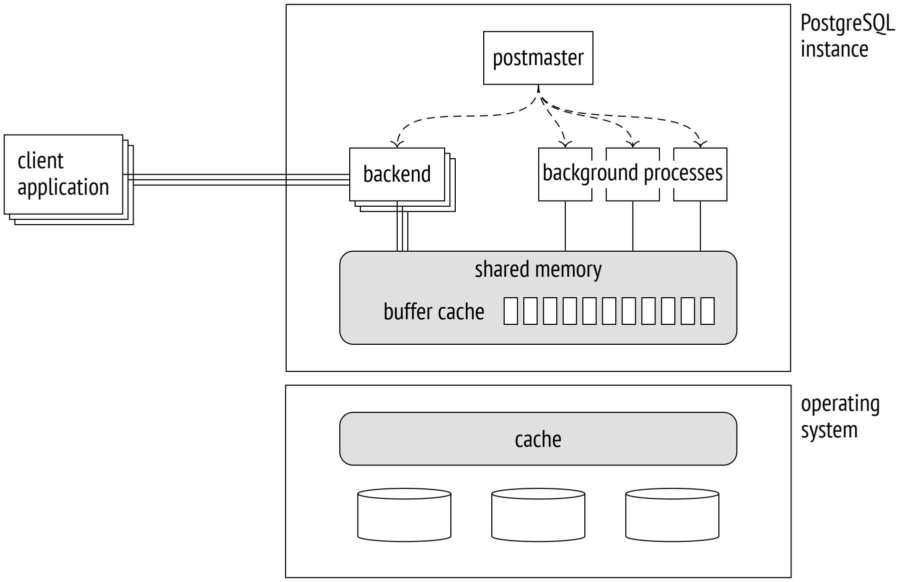
Each process is managed by configuration parameters, sometimes by dozens of them. To set up the server in a comprehensive manner, you have to be aware of its inner workings. But general considerations will only help you select more or less adequate initial values; later on, these settings have to be fine-tuned based on monitoring data.

To enable process interaction, postmaster allocates *shared memory*, which is available to all the processes.

Since disks (especially HDD, but SSD too) are much slower than RAM, PostgreSQL uses caching: some part of the shared RAM is reserved for recently read pages, in hope that they will be needed more than once and the overhead of repeated disk access will be reduced. Modified data is also flushed to disk after some delay, not immediately.

Buffer cache takes the greater part of the shared memory, which also contains other buffers used by the server to speed up disk access.

The operating system has its own cache too. PostgreSQL (almost) never bypasses the operating system mechanisms to use direct I/O, so it results in double caching.



In case of a failure (such as a power outage or an operating system crash), the data kept in RAM is lost, including that of the buffer cache. The files that remain on disk have their pages written at different points in time. To be able to restore data consistency, PostgreSQL maintains the *write-ahead log* (WAL) during its operation, which makes it possible to repeat lost operations when necessary.

1.3. Clients and the Client-Server Protocol

Another task of the postmaster process is to listen for incoming connections. Once a new client appears, postmaster spawns a separate *backend process*[1]. The client

[1] backend/tcop/postgres.c, PostgresMain function

establishes a connection and starts a *session* with this backend. The session continues until the client disconnects or the connection is lost.

The server has to spawn a separate backend for each client. If many clients are trying to connect, it can turn out to be a problem.

- Each process needs RAM to cache catalog tables, prepared statements, intermediate query results, and other data. The more connections are open, the more memory is required.
- If connections are short and frequent (a client performs a small query and disconnects), the cost of establishing a connection, spawning a new process, and performing pointless local caching is unreasonably high.
- The more processes are started, the more time is required to scan their list, and *p. 88* this operation is performed very often. As a result, performance may decline as the number of clients grows.

This problem can be resolved by *connection pooling*, which limits the number of spawned backends. PostgreSQL has no such built-in functionality, so we have to rely on third-party solutions: pooling managers integrated into the application server or external tools (such as PgBouncer[1] or Odyssey[2]). This approach usually means that each server backend can execute transactions of different clients, one after another. It imposes some restrictions on application development since it is only allowed to use resources that are local to a transaction, not to the whole session.

To understand each other, a client and a server must use one and the same interfacing protocol[3]. It is usually based on the standard libpq library, but there are also other custom implementations.

Speaking in the most general terms, the protocol allows clients to connect to the server and execute SQL queries.

A connection is always established to a particular database on behalf of a particular role, or user. Although the server supports a database cluster, it is required to establish a separate connection to each database that you would like to use in your

[1] pgbouncer.org
[2] github.com/yandex/odyssey
[3] postgresql.org/docs/14/protocol.html

application. At this point, *authentication* is performed: the backend process verifies the user's identity (for example, by asking for the password) and checks whether this user has the right to connect to the server and to the specified database.

SQL queries are passed to the backend process as text strings. The process parses the text, optimizes the query, executes it, and returns the result to the client.

Part I

Isolation and MVCC

2

Isolation

2.1. Consistency

The key feature of relational databases is their ability to ensure data *consistency*, that is, data *correctness*.

It is a known fact that at the database level it is possible to create *integrity constraints*, such as NOT NULL or UNIQUE. The database system ensures that these constraints are never broken, so data integrity is never compromised.

If all the required constraints could be formulated at the database level, consistency would be guaranteed. But some conditions are too complex for that, for example, they touch upon several tables at once. And even if a constraint can be defined in the database, but for some reason it is not, it does not mean that this constraint may be violated.

Thus, data consistency is stricter than integrity, but the database system has no idea what "consistency" actually means. If an application breaks it without breaking the integrity, there is no way for the database system to find out. Consequently, it is the application that must lay down the criteria for data consistency, and we have to believe that it is written correctly and will never have any errors.

But if the application always executes only correct sequences of operators, where does the database system come into play?

First of all, a correct sequence of operators can temporarily break data consistency, and—strange as it may seem—it is perfectly normal.

A hackneyed but clear example is a transfer of funds from one account to another. A consistency rule may sound as follows: *a money transfer must never change the total* *balance of the affected accounts*. It is quite difficult (although possible) to formulate this rule as an integrity constraint in SQL, so let's assume that it is defined at the application level and remains opaque to the database system. A transfer consists of two operations: the first one draws some money from one of the accounts, whereas the second one adds this sum to another account. The first operation breaks data consistency, whereas the second one restores it.

If the first operation succeeds, but the second one does not (because of some failure), data consistency will be broken. Such situations are unacceptable, but it takes a great deal of effort to detect and address them at the application level. Luckily it is not required—the problem can be completely solved by the database system itself if it knows that these two operations constitute an indivisible whole, that is, a *transaction*.

But there is also a more subtle aspect here. Being absolutely correct on their own, transactions can start operating incorrectly when run in parallel. That's because operations belonging to different transactions often get intermixed. There would be no such issues if the database system first completed all operations of one transaction and then moved on to the next one, but performance of sequential execution would be implausibly low.

> A truly simultaneous execution of transactions can only be achieved on systems with suitable hardware: a multi-core processor, a disk array, and so on. But the same reasoning is also true for a server that executes commands sequentially in the time-sharing mode. For generalization purposes, both these situations are sometimes referred to as *concurrent execution*.

Correct transactions that behave incorrectly when run together result in concurrency *anomalies*, or *phenomena*.

Here is a simple example. To get consistent data from the database, the application must not see any changes made by other uncommitted transactions, at the very minimum. Otherwise (if some transactions are rolled back), it would see the database state that has never existed. Such an anomaly is called a *dirty read*. There are also many other anomalies, which are more complex.

When running transactions concurrently, the database must guarantee that the result of such execution will be the same as the outcome of one of the possible sequential executions. In other words, it must *isolate* transactions from one another, thus taking care of any possible anomalies.

To sum it up, a transaction is a set of operations that takes the database from one correct state to another correct state (*consistency*), provided that it is executed in full (*atomicity*) and without being affected by other transactions (*isolation*). This definition combines the requirements implied by the first three letters of the ACID acronym. They are so intertwined that it makes sense to discuss them together. In fact, the durability requirement is hardly possible to split off either: after a crash, the system may still contain some changes made by uncommitted transactions, and you have to do something about it to restore data consistency.

Thus, the database system helps the application maintain data consistency by taking transaction boundaries into account, even though it has no idea about the implied consistency rules.

Unfortunately, full isolation is hard to implement and can negatively affect performance. Most real-life systems use weaker isolation levels, which prevent some anomalies, but not all of them. It means that the job of maintaining data consistency partially falls on the application. And that's exactly why it is very important to understand which isolation level is used in the system, what is guaranteed at this level and what is not, and how to ensure that your code will be correct in such conditions.

2.2. Isolation Levels and Anomalies Defined by the SQL Standard

The SQL standard specifies four isolation levels[1]. These levels are defined by the list of anomalies that may or may not occur during concurrent transaction execution. So when talking about isolation levels, we have to start with anomalies.

We should bear in mind that the standard is a theoretical construct: it affects the practice, but the practice still diverges from it in lots of ways. That's why all ex-

[1] postgresql.org/docs/14/transaction-iso.html

amples here are rather hypothetical. Dealing with transactions on bank accounts, these examples are quite self-explanatory, but I have to admit that they have nothing to do with real banking operations.

It is interesting that the actual database theory also diverges from the standard: it was developed after the standard had been adopted, and the practice was already well ahead.

Lost Update

The *lost update* anomaly occurs when two transactions read one and the same table row, then one of the transactions updates this row, and finally the other transaction updates the same row without taking into account any changes made by the first transaction.

Suppose that two transactions are going to increase the balance of one and the same account by $100. The first transaction reads the current value ($1,000), then the second transaction reads the same value. The first transaction increases the balance (making it $1,100) and writes the new value into the database. The second transaction does the same: it gets $1,100 after increasing the balance and writes this value. As a result, the customer loses $100.

Lost updates are forbidden by the standard at all isolation levels.

Dirty Reads and Read Uncommitted

The *dirty read* anomaly occurs when a transaction reads uncommitted changes made by another transaction.

For example, the first transaction transfers $100 to an empty account but does not commit this change. Another transaction reads the account state (which has been updated but not committed) and allows the customer to withdraw the money—even though the first transaction gets interrupted and its changes are rolled back, so the account is empty.

The standard allows dirty reads at the Read Uncommitted level.

Non-Repeatable Reads and Read Committed

The *non-repeatable read* anomaly occurs when a transaction reads one and the same row twice, whereas another transaction updates (or deletes) this row between these reads and commits the change. As a result, the first transaction gets different results.

For example, suppose there is a consistency rule that *forbids having a negative balance in bank accounts*. The first transaction is going to reduce the account balance by $100. It checks the current value, gets $1,000, and decides that this operation is possible. At the same time, another transaction withdraws all the money from this account and commits the changes. If the first transaction checked the balance again at this point, it would get $0 (but the decision to withdraw the money is already taken, and this operation causes an overdraft).

The standard allows non-repeatable reads at the Read Uncommitted and Read Committed levels.

Phantom Reads and Repeatable Read

The *phantom read* anomaly occurs when one and the same transaction executes two identical queries returning a set of rows that satisfy a particular condition, while another transaction adds some other rows satisfying this condition and commits the changes in the time interval between these queries. As a result, the first transaction gets two different sets of rows.

For example, suppose there is a consistency rule that *forbids a customer to have more than three accounts*. The first transaction is going to open a new account, so it checks how many accounts are currently available (let's say there are two of them) and decides that this operation is possible. At this very moment, the second transaction also opens a new account for this client and commits the changes. If the first transaction double-checked the number of open accounts, it would get three (but it is already opening another account, and the client ends up having four of them).

The standard allows phantom reads at the Read Uncommitted, Read Committed, and Repeatable Read isolation levels.

No Anomalies and Serializable

The standard also defines the Serializable level, which does not allow any anomalies. It is not the same as the ban on lost updates and dirty, non-repeatable, and phantom reads. In fact, there is a much higher number of known anomalies than the standard specifies, and an unknown number of still unknown ones.

The Serializable level must prevent *any* anomalies. It means that the application developer does not have to take isolation into account. If transactions execute correct operator sequences when run on their own, concurrent execution cannot break data consistency either.

To illustrate this idea, I will use a well-known table provided in the standard; the last column is added here for clarity:

	lost update	dirty read	non-repeatable read	phantom read	other anomalies
Read Uncommitted	–	yes	yes	yes	yes
Read Committed	–	–	yes	yes	yes
Repeatable Read	–	–	–	yes	yes
Serializable	–	–	–	–	–

Why These Anomalies?

Of all the possible anomalies, why does the standard mentions only some, and why exactly these ones?

No one seems to know it for sure. But it is not unlikely that other anomalies were simply not considered when the first versions of the standard were adopted, as theory was far behind practice at that time.

Besides, it was assumed that isolation had to be based on locks. The widely used *two-phase locking protocol* (2PL) requires transactions to lock the affected rows during execution and release the locks upon completion. In simplistic terms, the more locks a transaction acquires, the better it is isolated from other transactions. And consequently, the worse is the system performance, as transactions start queuing to get access to the same rows instead of running concurrently.

I believe that to a great extent the difference between the standard isolation levels is defined by the number of locks required for their implementation.

If the rows to be updated are locked for writes but not for reads, we get the Read Uncommitted isolation level, which allows reading data before it is committed.

If the rows to be updated are locked for both reads and writes, we get the Read Committed level: it is forbidden to read uncommitted data, but a query can return different values if it is run more than once (non-repeatable reads).

Locking the rows to be read and to be updated for all operations gives us the Repeatable Read level: a repeated query will return the same result.

However, the Serializable level poses a problem: it is impossible to lock a row that does not exist yet. It leaves an opportunity for phantom reads to occur: a transaction can add a row that satisfies the condition of the previous query, and this row will appear in the next query result.

Thus, regular locks cannot provide full isolation: to achieve it, we have to lock conditions (predicates) rather than rows. Such *predicate* locks were introduced as early as 1976 when System R was being developed; however, their practical applicability is limited to simple conditions for which it is clear whether two different predicates may conflict. As far as I know, predicate locks in their intended form have never been implemented in any system.

2.3. Isolation Levels in PostgreSQL

Over time, lock-based protocols for transaction management got replaced with the *Snapshot Isolation* (SI) protocol. The idea behind this approach is that each transaction accesses a consistent snapshot of data as it appeared at a particular point in time. The snapshot includes all the current changes committed before the snapshot was taken.

Snapshot isolation minimizes the number of required locks. In fact, a row will be locked only by concurrent update attempts. In all other cases, operations can be executed concurrently: writes never lock reads, and reads never lock anything.

PostgreSQL uses a *multiversion* flavor of the SI protocol. Multiversion concurrency control implies that at any moment the database system can contain several versions of one and the same row, so PostgreSQL can include an appropriate version into the snapshot rather than abort transactions that attempt to read stale data.

Based on snapshots, PostgreSQL isolation differs from the requirements specified in the standard—in fact, it is even stricter. Dirty reads are forbidden by design. Technically, you can specify the Read Uncommitted level, but its behavior will be the same as that of Read Committed, so I am not going to mention this level anymore. *p. 148* Repeatable Read allows neither non-repeatable nor phantom reads (even though it does not guarantee full isolation). But *in some cases*, there is a risk of losing changes at the Read Committed level.

	lost updates	dirty reads	non-repeatable reads	phantom reads	other anomalies
Read Committed	yes	–	yes	yes	yes
Repeatable Read	–	–	–	–	yes
Serializable	–	–	–	–	–

p. 86 Before exploring the internal mechanisms of isolation, let's discuss each of the three isolation levels from the user's perspective.

For this purpose, we are going to create the accounts table; Alice and Bob will have $1,000 each, but Bob will have two accounts:

```
=> CREATE TABLE accounts(
  id integer PRIMARY KEY GENERATED BY DEFAULT AS IDENTITY,
  client text,
  amount numeric
);
=> INSERT INTO accounts VALUES
  (1, 'alice', 1000.00), (2, 'bob', 100.00), (3, 'bob', 900.00);
```

Read Committed

No dirty reads. It is easy to check that reading dirty data is not allowed. Let's start a transaction. By default, it uses the Read Committed[1] isolation level:

[1] postgresql.org/docs/14/transaction-iso.html#XACT-READ-COMMITTED

```
=> BEGIN;
=> SHOW transaction_isolation;
 transaction_isolation
-----------------------
 read committed
(1 row)
```

To be more exact, the default level is set by the following parameter, which can be changed as required:

```
=> SHOW default_transaction_isolation;
 default_transaction_isolation
-------------------------------
 read committed
(1 row)
```

The opened transaction withdraws some funds from the customer account but does not commit these changes yet. It will see its own changes though, as it is always allowed:

```
=> UPDATE accounts SET amount = amount - 200 WHERE id = 1;
=> SELECT * FROM accounts WHERE client = 'alice';
 id | client | amount
----+--------+--------
  1 | alice  | 800.00
(1 row)
```

In the second session, we start another transaction that will also run at the Read Committed level:

```
    => BEGIN;
    => SELECT * FROM accounts WHERE client = 'alice';
     id | client | amount
    ----+--------+---------
      1 | alice  | 1000.00
    (1 row)
```

Predictably, the second transaction does not see any uncommitted changes—dirty reads are forbidden.

Non-repeatable reads. Now let the first transaction commit the changes. Then the second transaction will repeat the same query:

```
=> COMMIT;
```

```
    => SELECT * FROM accounts WHERE client = 'alice';
     id | client | amount
    ----+--------+--------
      1 | alice  | 800.00
    (1 row)
    => COMMIT;
```

The query receives an updated version of the data—and it is exactly what is understood by the *non-repeatable read* anomaly, which is allowed at the Read Committed level.

A practical insight: in a transaction, you must not take any decisions based on the data read by the previous operator, as everything can change in between. Here is an example whose variations appear in the application code so often that it can be considered a classic anti-pattern:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN
  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

During the time that passes between the check and the update, other transactions can freely change the state of the account, so such a "check" is absolutely useless. For better understanding, you can imagine that random operators of other transactions are "wedged" between the operators of the current transaction. For example, like this:

```
IF (SELECT amount FROM accounts WHERE id = 1) >= 1000 THEN

    UPDATE accounts SET amount = amount - 200 WHERE id = 1;
    COMMIT;

  UPDATE accounts SET amount = amount - 1000 WHERE id = 1;
END IF;
```

If everything goes wrong as soon as the operators are rearranged, then the code is incorrect. Do not delude yourself that you will never get into this trouble: anything that can go wrong will go wrong. Such errors are very hard to reproduce, and consequently, fixing them is a real challenge.

How can you correct this code? There are several options:

- Replace procedural code with declarative one.

  For example, in this particular case it is easy to turn an IF statement into a CHECK constraint:

  ```
  ALTER TABLE accounts
    ADD CHECK amount >= 0;
  ```

  Now you do not need any checks in the code: it is enough to simply run the command and handle the exception that will be raised if an integrity constraint violation is attempted.

- Use a single SQL operator.

  Data consistency can be compromised if a transaction gets committed within the time gap between operators of another transaction, thus changing data visibility. If there is only one operator, there are no such gaps.

  PostgreSQL has enough capabilities to solve complex tasks with a single SQL statement. In particular, it offers common table expressions (CTE) that can contain operators like INSERT, UPDATE, DELETE, as well as the INSERT ON CONFLICT operator that implements the following logic: insert the row if it does not exist, otherwise perform an update.

- Apply explicit locks.

  The last resort is to manually set an exclusive lock on all the required rows (SELECT FOR UPDATE) or even on the whole table (LOCK TABLE). This approach always works, but it nullifies all the advantages of MVCC: some operations that could be executed concurrently will run sequentially.

Read skew. However, it is not all that simple. The PostgreSQL implementation allows other, less known anomalies, which are not regulated by the standard.

Suppose the first transaction has started a money transfer between Bob's accounts:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 2;
```

Meanwhile, the other transaction starts looping through all Bob's accounts to calculate their total balance. It begins with the first account (seeing its previous state, of course):

```
    => BEGIN;
    => SELECT amount FROM accounts WHERE id = 2;
     amount
    --------
     100.00
    (1 row)
```

At this moment, the first transaction completes successfully:

```
=> UPDATE accounts SET amount = amount + 100 WHERE id = 3;
=> COMMIT;
```

The second transaction reads the state of the second account (and sees the already updated value):

```
    => SELECT amount FROM accounts WHERE id = 3;
     amount
    ---------
     1000.00
    (1 row)
    => COMMIT;
```

As a result, the second transaction gets $1,100 because it has read incorrect data. Such an anomaly is called *read skew*.

How can you avoid this anomaly at the Read Committed level? The answer is obvious: use a single operator. For example, like this:

```
SELECT sum(amount) FROM accounts WHERE client = 'bob';
```

I have been stating so far that data visibility can change only between operators, but is it really so? What if the query is running for a long time? Can it see different parts of data in different states in this case?

Let's check it out. A convenient way to do it is to add a delay to an operator by calling the pg_sleep function. Then the first row will be read at once, but the second row will have to wait for two seconds:

```
=> SELECT amount, pg_sleep(2) -- two seconds
FROM accounts WHERE client = 'bob';
```

While this statement is being executed, let's start another transaction to transfer the money back:

```
    => BEGIN;
    => UPDATE accounts SET amount = amount + 100 WHERE id = 2;
    => UPDATE accounts SET amount = amount - 100 WHERE id = 3;
    => COMMIT;
```

The result shows that the operator has seen all the data in the state that corresponds to the beginning of its execution, which is certainly correct:

```
 amount  | pg_sleep
---------+----------
    0.00 |
 1000.00 |
(2 rows)
```

But it is not all that simple either. If the query contains a function that is declared VOLATILE, and this function executes another query, then the data seen by this nested query will not be consistent with the result of the main query.

Let's check the balance in Bob's accounts using the following function:

```
=> CREATE FUNCTION get_amount(id integer) RETURNS numeric
AS $$
  SELECT amount FROM accounts a WHERE a.id = get_amount.id;
$$ VOLATILE LANGUAGE sql;
=> SELECT get_amount(id), pg_sleep(2)
FROM accounts WHERE client = 'bob';
```

We will transfer the money between the accounts once again while our delayed query is being executed:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
=> COMMIT;
```

In this case, we are going to get inconsistent data—$100 has been lost:

```
 get_amount | pg_sleep
------------+----------
     100.00 |
     800.00 |
(2 rows)
```

I would like to emphasize that this effect is possible only at the Read Committed isolation level, and only if the function is VOLATILE. The trouble is that PostgreSQL uses exactly this isolation level and this volatility category by default. So we have to admit that the trap is set in a very cunning way.

Read skew instead of lost updates. The read skew anomaly can also occur within a single operator during an update—even though in a somewhat unexpected way.

Let's see what happens if two transactions try to modify one and the same row. Bob currently has a total of $1,000 in two accounts:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)
```

Start a transaction that will reduce Bob's balance:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100 WHERE id = 3;
```

At the same time, the other transaction will be calculating the interest for all customer accounts with the total balance of $1,000 or more:

```
=> UPDATE accounts SET amount = amount * 1.01
WHERE client IN (
  SELECT client
  FROM accounts
  GROUP BY client
  HAVING sum(amount) >= 1000
);
```

The UPDATE operator execution virtually consists of two stages. First, the rows to be updated are selected based on the provided condition. Since the first transaction is not committed yet, the second transaction cannot see its result, so the selection of rows picked for interest accrual is not affected. Thus, Bob's accounts satisfy the condition, and his balance must be increased by $10 once the UPDATE operation completes.

At the second stage, the selected rows are updated one by one. The second transaction has to wait because the row with id = 3 is locked: it is being updated by the first transaction.

Meanwhile, the first transaction commits its changes:

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client |  amount
----+--------+----------
  2 | bob    | 202.0000
  3 | bob    | 707.0000
(2 rows)
```

On the one hand, the UPDATE command must not see any changes made by the first transaction. But on the other hand, it must not lose any committed changes.

Once the lock is released, the UPDATE operator *re-reads* the row to be updated (but only this row!). As a result, Bob gets $9 of interest, based on the total of $900. But if he had $900, his accounts should not have been included into the query results in the first place.

Thus, our transaction has returned incorrect data: different rows have been read from different snapshots. Instead of a lost update, we observe the read skew anomaly again.

Lost updates. However, the trick of re-reading the locked row will not help against lost updates if the data is modified by different SQL operators.

p. 40 Here is an example that we have already seen. The application reads and registers (outside of the database) the current balance of Alice's account:

```
=> BEGIN;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 800.00
(1 row)
```

Meanwhile, the other transaction does the same:

```
    => BEGIN;
    => SELECT amount FROM accounts WHERE id = 1;
     amount
    --------
     800.00
    (1 row)
```

The first transaction increases the previously registered value by $100 and commits this change:

```
=> UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
RETURNING amount;
 amount
--------
 900.00
(1 row)
UPDATE 1
=> COMMIT;
```

The second transaction does the same:

```
    => UPDATE accounts SET amount = 800.00 + 100 WHERE id = 1
    RETURNING amount;
     amount
    --------
     900.00
    (1 row)
    UPDATE 1
    => COMMIT;
```

Unfortunately, Alice has lost $100. The database system does not know that the registered value of $800 is somehow related to accounts.amount, so it cannot prevent the lost update anomaly. At the Read Committed isolation level, this code is incorrect.

Repeatable Read

No non-repeatable and phantom reads. As its name suggests, the Repeatable Read[1] isolation level must guarantee repeatable reading. Let's check it and make sure that phantom reads cannot occur either. For this purpose, we are going to start a transaction that will revert Bob's accounts to their previous state and create a new account for Charlie:

```
=> BEGIN;

=> UPDATE accounts SET amount = 200.00 WHERE id = 2;
=> UPDATE accounts SET amount = 800.00 WHERE id = 3;
=> INSERT INTO accounts VALUES
  (4, 'charlie', 100.00);

=> SELECT * FROM accounts ORDER BY id;
 id | client  | amount
----+---------+--------
  1 | alice   | 900.00
  2 | bob     | 200.00
  3 | bob     | 800.00
  4 | charlie | 100.00
(4 rows)
```

In the second session, let's start another transaction, with the Repeatable Read level explicitly specified in the BEGIN command (the level of the first transaction is not important):

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SELECT * FROM accounts ORDER BY id;
```

[1] postgresql.org/docs/14/transaction-iso.html#XACT-REPEATABLE-READ

```
     id | client |  amount
    ----+--------+----------
      1 | alice  |   900.00
      2 | bob    | 202.0000
      3 | bob    | 707.0000
    (3 rows)
```

Now the first transaction commits its changes, and the second transaction repeats the same query:

```
=> COMMIT;
```

```
    => SELECT * FROM accounts ORDER BY id;
     id | client |  amount
    ----+--------+----------
      1 | alice  |   900.00
      2 | bob    | 202.0000
      3 | bob    | 707.0000
    (3 rows)
    => COMMIT;
```

The second transaction still sees the same data as before: neither new rows nor row updates are visible. At this isolation level, you do not have to worry that something will change between operators.

p. 50 **Serialization failures instead of lost updates.** As we have already seen, if two transactions update one and the same row at the Read Committed level, it can cause the read skew anomaly: the waiting transaction has to re-read the locked row, so it sees the state of this row at a different point in time as compared to other rows.

Such an anomaly is not allowed at the Repeatable Read isolation level, and if it does happen, the transaction can only be aborted with a serialization failure. Let's check it out by repeating the scenario with interest accrual:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 800.00
(2 rows)
=> BEGIN;
=> UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
```

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => UPDATE accounts SET amount = amount * 1.01
    WHERE client IN (
      SELECT client
      FROM accounts
      GROUP BY client
      HAVING sum(amount) >= 1000
    );
```

```
=> COMMIT;
```

```
    ERROR:  could not serialize access due to concurrent update
    => ROLLBACK;
```

The data remains consistent:

```
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  2 | bob    | 200.00
  3 | bob    | 700.00
(2 rows)
```

The same error will be raised by any concurrent row updates, even if they affect different columns.

We will also get this error if we try to update the balance based on the previously stored value:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT amount FROM accounts WHERE id = 1;
 amount
--------
 900.00
(1 row)
```

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SELECT amount FROM accounts WHERE id = 1;
     amount
    --------
     900.00
    (1 row)
```

```
=> UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
RETURNING amount;
 amount
---------
 1000.00
(1 row)
UPDATE 1
=> COMMIT;
```

```
    => UPDATE accounts SET amount = 900.00 + 100.00 WHERE id = 1
    RETURNING amount;
    ERROR:  could not serialize access due to concurrent update
    => ROLLBACK;
```

A practical insight: if your application is using the Repeatable Read isolation level for writing transactions, it must be ready to retry transactions that have been completed with a serialization failure. For read-only transactions, such an outcome is impossible.

Write skew. As we have seen, the PostgreSQL implementation of the Repeatable Read isolation level prevents all the anomalies described in the standard. But not all possible ones: no one knows how many of them exist. However, one important fact is proved for sure: snapshot isolation does not prevent *only two* anomalies, no matter how many other anomalies are out there.

The first one is *write skew*.

Let's define the following consistency rule: *it is allowed to have a negative balance in some of the customer's accounts as long as the total balance is non-negative.*

The first transaction gets the total balance of Bob's accounts:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
  sum
--------
 900.00
(1 row)
```

The second transaction gets the same sum:

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SELECT sum(amount) FROM accounts WHERE client = 'bob';
      sum
    --------
     900.00
    (1 row)
```

The first transaction fairly assumes that it can debit one of the accounts by $600:

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

The second transaction comes to the same conclusion, but debits the other account:

```
    => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
    => COMMIT;
```

```
=> COMMIT;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+---------
  2 | bob    | -400.00
  3 | bob    |  100.00
(2 rows)
```

Bob's total balance is now negative, although both transactions would have been correct if run separately.

Read-only transaction anomaly. The *read-only transaction* anomaly is the second and the last one allowed at the Repeatable Read isolation level. To observe this anomaly, we have to run three transactions: two of them are going to update the data, while the third one will be read-only.

But first let's restore Bob's balance:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> SELECT * FROM accounts WHERE client = 'bob';
 id | client | amount
----+--------+--------
  3 | bob    | 100.00
  2 | bob    | 900.00
(2 rows)
```

The first transaction calculates the interest to be accrued on Bob's total balance and adds this sum to one of his accounts:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ; -- 1
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

Then the second transaction withdraws some money from Bob's other account and commits this change:

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ; -- 2
    => UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
    => COMMIT;
```

If the first transaction gets committed at this point, there will be no anomalies: we could assume that the first transaction is committed before the second one (but not vice versa—the first transaction had seen the state of account with id = 3 before any updates were made by the second transaction).

But let's imagine that at this very moment we start a ready-only transaction to query an account that is not affected by the first two transactions:

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ; -- 3
    => SELECT * FROM accounts WHERE client = 'alice';
     id | client | amount
    ----+--------+---------
      1 | alice  | 1000.00
    (1 row)
```

And only now will the first transaction get committed:

```
=> COMMIT;
```

Which state should the third transaction see at this point? Having started, it could see the changes made by the second transaction (which had already been committed), but not by the first one (which had not been committed yet). But as we have already established, the second transaction should be treated as if it were started after the first one. Any state seen by the third transaction will be inconsistent—this is exactly what is meant by the read-only transaction anomaly:

```
    => SELECT * FROM accounts WHERE client = 'bob';
     id | client | amount
    ----+--------+--------
      2 | bob    | 900.00
      3 | bob    |   0.00
    (2 rows)
    => COMMIT;
```

Serializable

The Serializable[1] isolation level prevents all possible anomalies. This level is virtually built on top of snapshot isolation. Those anomalies that do not occur at the Repeatable Read isolation level (such as dirty, non-repeatable, or phantom reads) cannot occur at the Serializable level either. And those two anomalies that do occur (write skew and read-only transaction anomalies) get detected in a special way to abort the transaction, causing an already familiar serialization failure.

No anomalies. Let's make sure that our write skew scenario will eventually end with a serialization failure: *p. 56*

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
=> SELECT sum(amount) FROM accounts WHERE client = 'bob';
   sum
----------
 910.0000
(1 row)
```

[1] postgresql.org/docs/14/transaction-iso.html#XACT-SERIALIZABLE

```
    => BEGIN ISOLATION LEVEL SERIALIZABLE;
    => SELECT sum(amount) FROM accounts WHERE client = 'bob';
          sum
    ----------
      910.0000
    (1 row)
```

```
=> UPDATE accounts SET amount = amount - 600.00 WHERE id = 2;
```

```
    => UPDATE accounts SET amount = amount - 600.00 WHERE id = 3;
    => COMMIT;
    COMMIT
```

```
=> COMMIT;
ERROR:  could not serialize access due to read/write dependencies
among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during
commit attempt.
HINT:  The transaction might succeed if retried.
```

The scenario with the read-only transaction anomaly will lead to the same error.

Deferring a read-only transaction. To avoid situations when a read-only transaction can cause an anomaly that compromises data consistency, PostgreSQL offers an interesting solution: this transaction can be deferred until its execution becomes safe. It is the only case when a SELECT statement can be blocked by row updates.

We are going to check it out by repeating the scenario that demonstrated the read-only transaction anomaly:

```
=> UPDATE accounts SET amount = 900.00 WHERE id = 2;
=> UPDATE accounts SET amount = 100.00 WHERE id = 3;
=> SELECT * FROM accounts WHERE client = 'bob' ORDER BY id;
 id | client | amount
----+--------+--------
  2 | bob    | 900.00
  3 | bob    | 100.00
(2 rows)
=> BEGIN ISOLATION LEVEL SERIALIZABLE; -- 1
=> UPDATE accounts SET amount = amount + (
  SELECT sum(amount) FROM accounts WHERE client = 'bob'
) * 0.01
WHERE id = 2;
```

```
    => BEGIN ISOLATION LEVEL SERIALIZABLE; -- 2
    => UPDATE accounts SET amount = amount - 100.00 WHERE id = 3;
    => COMMIT;
```

Let's explicitly declare the third transaction as READ ONLY and DEFERRABLE:

```
    => BEGIN ISOLATION LEVEL SERIALIZABLE READ ONLY DEFERRABLE; -- 3
    => SELECT * FROM accounts WHERE client = 'alice';
```

An attempt to run the query blocks the transaction—otherwise, it would have caused an anomaly.

And only when the first transaction is committed, the third one can continue its execution:

```
=> COMMIT;
```

```
     id | client | amount
    ----+--------+---------
      1 | alice  | 1000.00
    (1 row)
    => SELECT * FROM accounts WHERE client = 'bob';
     id | client |  amount
    ----+--------+----------
      2 | bob    | 910.0000
      3 | bob    |     0.00
    (2 rows)
    => COMMIT;
```

Thus, if an application uses the Serializable isolation level, it must be ready to retry transactions that have ended with a serialization failure. (The Repeatable Read level requires the same approach unless the application is limited to read-only transactions.)

The Serializable isolation level brings ease of programming, but the price you pay is the overhead incurred by anomaly detection and forced termination of a certain fraction of transactions. You can lower this impact by explicitly using the READ ONLY clause when declaring read-only transactions. But the main questions is, of course, how big the fraction of aborted transactions is—since these transactions will have to be retried. It would have been not so bad if PostgreSQL aborted only those transactions that result in data conflicts and are really incompatible. But such an approach would inevitably be too resource-intensive, as it would involve tracking operations on each row.

The current implementation allows false positives: PostgreSQL can abort some absolutely safe transactions that are simply out of luck. Their "luck" depends on many factors, such as the presence of appropriate indexes or the amount of RAM available, so the actual behavior is hard to predict in advance.

If you use the Serializable level, it must be observed by all transactions of the application. When combined with other levels, Serializable behaves as Repeatable Read without any notice. So if you decide to use the Serializable level, it makes sense to modify the *default_transaction_isolation* parameter value accordingly—even though someone can still overwrite it by explicitly setting a different level.

read committed

v. 12 There are also other restrictions; for example, queries run at the Serializable level cannot be executed on replicas. And although the functionality of this level is constantly being improved, the current limitations and overhead make it less attractive.

2.4. Which Isolation Level to Use?

Read Committed is the default isolation level in PostgreSQL, and apparently it is this level that is used in the vast majority of applications. This level can be convenient because it allows aborting transactions only in case of a failure; it does not abort any transactions to preserve data consistency. In other words, serialization failures cannot occur, so you do not have to take care of transaction retries.

The downside of this level is a large number of possible anomalies, which have been discussed in detail above. A developer has to keep them in mind all the time and write the code in a way that prevents their occurrence. If it is impossible to define all the needed actions in a single SQL statement, then you have to resort to explicit locking. The toughest part is that the code is hard to test for errors related to data inconsistency; such errors can appear in unpredictable and barely reproducible ways, so they are very hard to fix too.

The Repeatable Read isolation level eliminates some of the inconsistency problems, but alas, not all of them. Therefore, you must not only remember about the remaining anomalies, but also modify the application to correctly handle serialization failures, which is certainly inconvenient. However, for read-only transactions this level is a perfect complement to the Read Committed level; it can be very useful for cases like building reports that involve multiple SQL queries.

And finally, the Serializable isolation level allows you not to worry about data consistency at all, which simplifies writing the code to a great extent. The only thing required from the application is the ability to retry any transaction that is aborted with a serialization failure. However, the number of aborted transactions and associated overhead can significantly reduce system throughput. You should also keep in mind that the Serializable level is not supported on replicas and cannot be combined with other isolation levels.

3

Pages and Tuples

3.1. Page Structure

Each page has a certain inner layout that usually consists of the following parts[1]:

- page header
- an array of item pointers
- free space
- items (row versions)
- special space

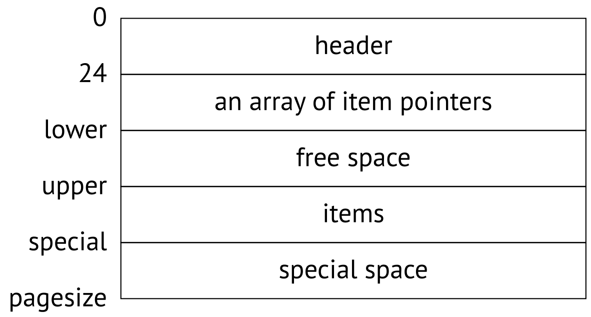
Page Header

The page *header* is located in the lowest addresses and has a fixed size. It stores various information about the page, such as its checksum and the sizes of all the other parts of the page. *p. 114*

These sizes can be easily displayed using the pageinspect[2] extension. Let's take a look at the first page of the table (page numbering is zero-based):

[1] postgresql.org/docs/14/storage-page-layout.html
include/storage/bufpage.h
[2] postgresql.org/docs/14/pageinspect.html

```
=> CREATE EXTENSION pageinspect;
=> SELECT lower, upper, special, pagesize
FROM page_header(get_raw_page('accounts',0));
 lower | upper | special | pagesize
-------+-------+---------+----------
   152 |  6904 |    8192 |     8192
(1 row)
```



Special Space

The *special space* is located in the opposite part of the page, taking its highest addresses. It is used by some indexes to store auxiliary information; in other indexes and table pages this space is zero-sized.

In general, the layout of index pages is quite diverse; their content largely depends on a particular index type. Even one and the same index can have different kinds of pages: for example, B-trees have a metadata page of a special structure (page zero) and regular pages that are very similar to table pages.

Tuples

Rows contain the actual data stored in the database, together with some additional information. They are located just before the special space.

In the case of tables, we have to deal with *row versions* rather than rows because multiversion concurrency control implies having several versions of one and the same row. Indexes do not use this MVCC mechanism; instead, they have to reference all the available row versions, falling back on visibility rules to select the appropriate ones.

Both table row versions and index entries are often referred to as *tuples*. This term is borrowed from the relational theory—it is yet another legacy of PostgreSQL's academic past.

Item Pointers

The *array of pointers* to tuples serves as the page's table of contents. It is located right after the header.

Index entries have to refer to particular heap tuples somehow. PostgreSQL employs six-byte *tuple identifiers* (TIDS) for this purpose. Each TID consists of the page number of the main fork and a reference to a particular row version located in this page. *p. 21*

In theory, tuples could be referred to by their offset from the start of the page. But then it would be impossible to move tuples within pages without breaking these references, which in turn would lead to page fragmentation and other unpleasant consequences.

For this reason, PostgreSQL uses indirect addressing: a tuple identifier refers to the corresponding pointer number, and this pointer specifies the current offset of the tuple. If the tuple is moved within the page, its TID still remains the same; it is enough to modify the pointer, which is also located in this page.

Each pointer takes exactly four bytes and contains the following data:

- tuple offset from the start of the page
- tuple length
- several bits defining the tuple status

Free Space

Pages can have some *free space* left between pointers and tuples (which is reflected in the free space map). There is no page fragmentation: all the free space available is always aggregated into one chunk[1]. *p. 22*

3.2. Row Version Layout

Each row version contains a header followed by actual data. The header consists of multiple fields, including the following:

xmin, xmax represent transaction IDs; they are used to differentiate between this and other versions of one and the same row.

infomask provides a set of information bits that define version properties.

ctid is a pointer to the next updated version of the same row.

null bitmap is an array of bits marking the columns that can contain NULL values.

As a result, the header turns out quite big: it requires at least 23 bytes for each tuple, and this value is often exceeded because of the null bitmap and the mandatory padding used for data alignment. In a "narrow" table, the size of various metadata can easily beat the size of the actual data stored.

Data layout on disk fully coincides with data representation in RAM. The page along with its tuples is read into the buffer cache as is, without any transformations. That's why data files are incompatible between different platforms[2].

One of the sources of incompatibility is the byte order. For example, the x86 architecture is little-endian, z/Architecture is big-endian, and ARM has configurable byte order.

Another reason is data alignment by machine word boundaries, which is required by many architectures. For example, in a 32-bit x86 system, integer numbers (the integer type, takes four bytes) are aligned by the boundary of four-byte words,

[1] backend/storage/page/bufpage.c, PageRepairFragmentation function
[2] include/access/htup_details.h

just like double-precision floating-point numbers (the double precision type, eight bytes). But in a 64-bit system, double values are aligned by the boundary of eight-byte words.

Data alignment makes the size of a tuple dependent on the order of fields in the table. This effect is usually negligible, but in some cases it can lead to a significant size increase. Here is an example:

```
=> CREATE TABLE padding(
  b1 boolean,
  i1 integer,
  b2 boolean,
  i2 integer
);
=> INSERT INTO padding VALUES (true,1,false,2);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
 lp_len
--------
     40
(1 row)
```

I have used the heap_page_items function of the pageinspect extension to display some details about pointers and tuples.

> In PostgreSQL, tables are often referred to as *heap*. This is yet another obscure term that hints at the similarity between space allocation for tuples and dynamic memory allocation. Some analogy can certainly be seen, but tables are managed by completely different algorithms. We can interpret this term in the sense that "everything is piled up into a heap," by contrast with ordered indexes.

The size of the row is 40 bytes. Its header takes 24 bytes, a column of the integer type takes 4 bytes, and boolean columns take 1 byte each. It makes 34 bytes, and 6 bytes are wasted on four-byte alignment of integer columns.

If we rebuild the table, the space will be used more efficiently:

```
=> DROP TABLE padding;
=> CREATE TABLE padding(
  i1 integer,
  i2 integer,
  b1 boolean,
  b2 boolean
);
```

```
=> INSERT INTO padding VALUES (1,2,true,false);
=> SELECT lp_len FROM heap_page_items(get_raw_page('padding', 0));
 lp_len
--------
     34
(1 row)
```

Another possible micro-optimization is to start the table with the fixed-length columns that cannot contain NULL values. Access to such columns will be more efficient because it is possible to cache their offset within the tuple[1].

3.3. Operations on Tuples

To identify different versions of one and the same row, PostgreSQL marks each of them with two values: xmin and xmax. These values define "validity time" of each row version, but instead of the actual time, they rely on ever-increasing transaction IDS. *p. 137*

When a row is created, its xmin value is set to the transaction ID of the INSERT command.

When a row is deleted, the xmax value of its current version is set to the transaction ID of the DELETE command.

With a certain degree of abstraction, the UPDATE command can be regarded as two separate operations: DELETE and INSERT. First, the xmax value of the current row version is set to the transaction ID of the UPDATE command. Then a new version of this row is created; its xmin value will be the same as the xmax value of the previous version.

Now let's get down to some low-level details of different operations on tuples[2].

For these experiments, we will need a two-column table with an index created on one of the columns:

[1] backend/access/common/heaptuple.c, heap_deform_tuple function
[2] backend/access/transam/README

```
=> CREATE TABLE t(
  id integer GENERATED ALWAYS AS IDENTITY,
  s text
);
=> CREATE INDEX ON t(s);
```

Insert

Start a transaction and insert one row:

```
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
```

Here is the current transaction ID:

```
=> --  txid_current() before v.13
SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                776
(1 row)
```

To denote the concept of a transaction, PostgreSQL uses the term xact, which can be found both in SQL function names and in the source code. Consequently, a transaction ID can be called xact ID, TXID, or simply XID. We are going to come across these abbreviations over and over again.

Let's take a look at the page contents. The heap_page_items function can give us all the required information, but it shows the data "as is," so the output format is a bit hard to comprehend:

```
=> SELECT *
FROM heap_page_items(get_raw_page('t',0)) \gx
-[ RECORD 1 ]-------------------
lp          | 1
lp_off      | 8160
lp_flags    | 1
lp_len      | 32
t_xmin      | 776
t_xmax      | 0
t_field3    | 0
t_ctid      | (0,1)
```

```
t_infomask2 | 2
t_infomask  | 2050
t_hoff      | 24
t_bits      |
t_oid       |
t_data      | \x0100000009464f4f
```

To make it more readable, we can leave out some information and expand a few columns:

```
=> SELECT '(0,'||lp||')' AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin as xmin,
     t_xmax as xmax,
     (t_infomask & 256) > 0  AS xmin_committed,
     (t_infomask & 512) > 0  AS xmin_aborted,
     (t_infomask & 1024) > 0 AS xmax_committed,
     (t_infomask & 2048) > 0 AS xmax_aborted
FROM heap_page_items(get_raw_page('t',0)) \gx

-[ RECORD 1 ]--+-------
ctid           | (0,1)
state          | normal
xmin           | 776
xmax           | 0
xmin_committed | f
xmin_aborted   | f
xmax_committed | f
xmax_aborted   | t
```

This is what has been done here:

- The lp pointer is converted to the standard format of a tuple ID: (page number, pointer number).
- The lp_flags state is spelled out. Here it is set to the normal value, which means that it really points to a tuple.
- Of all the information bits, we have singled out just two pairs so far. The xmin_committed and xmin_aborted bits show whether the xmin transaction is

committed or aborted. The xmax_committed and xmax_aborted bits give similar information about the xmax transaction.

v. 13 The pageinspect extension provides the heap_tuple_infomask_flags function that explains all the information bits, but I am going to retrieve only those that are required at the moment, showing them in a more concise form.

Let's get back to our experiment. The INSERT command has added pointer 1 to the heap page; it refers to the first tuple, which is currently the only one.

The xmin field of the tuple is set to the current transaction ID. This transaction is still active, so the xmin_committed and xmin_aborted bits are not set yet.

The xmax field contains 0, which is a dummy number showing that this tuple has not been deleted and represents the current version of the row. Transactions will ignore this number because the xmax_aborted bit is set.

It may seem strange that the bit corresponding to an aborted transaction is set for the transaction that has not happened yet. But there is no difference between such transactions from the isolation standpoint: an aborted transaction leaves no trace, hence it has never existed.

We will use this query more than once, so I am going to wrap it into a function. And while being at it, I will also make the output more concise by hiding the information bit columns and displaying the status of transactions together with their IDs.

```
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(ctid tid, state text, xmin text, xmax text)
AS $$
SELECT (pageno,lp)::text::tid AS ctid,
     CASE lp_flags
       WHEN 0 THEN 'unused'
       WHEN 1 THEN 'normal'
       WHEN 2 THEN 'redirect to '||lp_off
       WHEN 3 THEN 'dead'
     END AS state,
     t_xmin || CASE
       WHEN (t_infomask & 256) > 0 THEN ' c'
       WHEN (t_infomask & 512) > 0 THEN ' a'
       ELSE ''
     END AS xmin,
     t_xmax || CASE
```

```
        WHEN (t_infomask & 1024) > 0 THEN ' c'
        WHEN (t_infomask & 2048) > 0 THEN ' a'
        ELSE ''
      END AS xmax
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Now it is much clearer what is happening in the tuple header:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 776  | 0 a
(1 row)
```

You can get similar but less detailed information from the table itself by querying the xmin and xmax pseudocolumns:

```
=> SELECT xmin, xmax, * FROM t;
 xmin | xmax | id |  s
------+------+----+-----
  776 |    0 |  1 | FOO
(1 row)
```

Commit

Once a transaction has been completed successfully, its status has to be stored somehow—it must be registered that the transaction is *committed*. For this purpose, PostgreSQL employs a special CLOG[1] (commit log) structure. It is stored as files in the PGDATA/pg_xact directory rather than as a system catalog table.

Previously, these files were located in PGDATA/pg_clog, but in version 10 this directory got renamed[2]: it was not uncommon for database administrators unfamiliar with PostgreSQL to delete it in search of free disk space, thinking that a "log" is something unnecessary.

1 include/access/clog.h
backend/access/transam/clog.c
2 commitfest.postgresql.org/13/750.

p. 147 CLOG is split into several files solely for convenience. These files are accessed page by page via buffers in the server's shared memory[1].

Just like a tuple header, CLOG contains two bits for each transaction: committed and aborted.

Once committed, a transaction is marked in CLOG with the committed bit. When any other transaction accesses a heap page, it has to answer the question: has the xmin transaction already finished?

- If not, then the created tuple must not be visible.

  To check whether the transaction is still active, PostgreSQL uses yet another structure located in the shared memory of the instance; it is called ProcArray. This structure contains the list of all the active processes, with the corresponding current (active) transaction specified for each process.

- If yes, was it committed or aborted? In the latter case, the corresponding tuple cannot be visible either.

  It is this check that requires CLOG. But even though the most recent CLOG pages are stored in memory buffers, it is still expensive to perform this check every time. Once determined, the transaction status is written into the tuple header—more specifically, into xmin_committed and xmin_aborted information bits, which are also called *hint bits*. If one of these bits is set, then the xmin transaction status is considered to be already known, and the next transaction will have to access neither CLOG nor ProcArray.

Why aren't these bits set by the transaction that performs row insertion? The problem is that it is not known yet at that time whether this transaction will complete successfully. And when it is committed, it is already unclear which tuples and pages have been changed. If a transaction affects many pages, it may be too expensive to track them. Besides, some of these pages may be not in the cache anymore; reading them again to simply update the hint bits would seriously slow down the commit.

1 backend/access/transam/clog.c

The flip side of this cost reduction is that any transaction (even a read-only SELECT command) can start setting hint bits, thus leaving a trail of dirtied pages in the buffer cache.

Finally, let's commit the transaction started with the INSERT statement:

```
=> COMMIT;
```

Nothing has changed in the page (but we know that the transaction status has already been written into CLOG):

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin | xmax
-------+--------+------+------
 (0,1) | normal | 776  | 0 a
(1 row)
```

Now the first transaction that accesses the page (in a "standard" way, without using pageinspect) has to determine the status of the xmin transaction and update the hint bits:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)
```

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 0 a
(1 row)
```

Delete

When a row is deleted, the xmax field of its current version is set to the transaction ID that performs the deletion, and the xmax_aborted bit is unset.

While this transaction is active, the xmax value serves as a row lock. If another transaction is going to update or delete this row, it will have to wait until the xmax transaction is complete.

Let's delete a row:

```
=> BEGIN;
=> DELETE FROM t;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                777
(1 row)
```

The transaction ID has already been written into the xmax field, but the information bits have not been set yet:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 777
(1 row)
```

Abort

The mechanism of aborting a transaction is similar to that of commit and happens just as fast, but instead of committed it sets the aborted bit in CLOG. Although the corresponding command is called ROLLBACK, no actual data rollback is happening: all the changes made by the aborted transaction in data pages remain in place.

```
=> ROLLBACK;
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 777
(1 row)
```

When the page is accessed, the transaction status is checked, and the tuple receives the xmax_aborted hint bit. The xmax number itself still remains in the page, but no one is going to pay attention to it anymore:

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | FOO
(1 row)
```

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 776 c | 777 a
(1 row)
```

Update

An update is performed in such a way as if the current tuple is deleted, and then a new one is inserted:

```
=> BEGIN;
```

```
=> UPDATE t SET s = 'BAR';
```

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                778
(1 row)
```

The query returns a single row (its new version):

```
=> SELECT * FROM t;
 id |  s
----+-----
  1 | BAR
(1 row)
```

But the page keeps both versions:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 776 c | 778
 (0,2) | normal | 778   | 0 a
(2 rows)
```

The xmax field of the previously deleted version contains the current transaction ID. This value is written on top of the old one because the previous transaction was aborted. The xmax_aborted bit is unset since the status of the current transaction is still unknown.

To complete this experiment, let's commit the transaction.

```
=> COMMIT;
```

3.4. Indexes

Regardless of their type, indexes do not use row versioning; each row is represented by exactly one tuple. In other words, index row headers do not contain xmin and xmax fields. Index entries point to all the versions of the corresponding table row. *p. 104* To figure out which row version is visible, transactions have to access the table (unless the required page appears in the visibility map).

For convenience, let's create a simple function that will use pageinspect to display all the index entries in the page (B-tree index pages store them as a flat list):

```
=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid)
AS $$
SELECT itemoffset,
       htid -- ctid before v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;
```

The page references both heap tuples, the current and the previous one:

```
=> SELECT * FROM index_page('t_s_idx',1);
 itemoffset | htid
------------+-------
          1 | (0,2)
          2 | (0,1)
(2 rows)
```

Since BAR < FOO, the pointer to the second tuple comes first in the index.

3.5. TOAST

A TOAST table is virtually a regular table, and it has its own versioning that does not depend on row versions of the main table. However, rows of TOAST tables are handled in such a way that they are never updated; they can be either added or deleted, so their versioning is somewhat artificial. *p. 24*

Each data modification results in creation of a new tuple in the main table. But if an update does not affect any long values stored in TOAST, the new tuple will reference an existing toasted value. Only when a long value gets updated will PostgreSQL create both a new tuple in the main table and new "toasts."

3.6. Virtual Transactions

To consume transaction IDs sparingly, PostgreSQL offers a special optimization.

If a transaction is read-only, it does not affect row visibility in any way. That's why such a transaction is given a *virtual XID*[1] at first, which consists of the backend process ID and a sequential number. Assigning a virtual XID does not require any synchronization between different processes, so it happens very fast. At this point, the transaction has no real ID yet:

```
=> BEGIN;
```

[1] backend/access/transam/xact.c

```
=> --  txid_current_if_assigned() before v.13
SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−

(1 row)
```

At different points in time, the system can contain some virtual XIDs that have already been used. And it is perfectly normal: virtual XIDs exist only in RAM, and only while the corresponding transactions are active; they are never written into data pages and never get to disk.

Once the transaction starts modifying data, it receives an actual unique ID:

```
=> UPDATE accounts
SET amount = amount - 1.00;

=> SELECT pg_current_xact_id_if_assigned();
 pg_current_xact_id_if_assigned
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
                             780
(1 row)

=> COMMIT;
```

3.7. Subtransactions

Savepoints

SQL supports *savepoints*, which enable canceling some of the operations within a transaction without aborting this transaction as a whole. But such a scenario does not fit the course of action described above: the status of a transaction applies to all its operations, and no physical data rollback is performed.

To implement this functionality, a transaction containing a savepoint is split into several *subtransactions*[1], so their status can be managed separately.

[1] backend/access/transam/subtrans.c

Subtransactions have their own IDs (which are bigger than the ID of the main transaction). The status of a subtransaction is written into CLOG in the usual manner; however, committed subtransactions receive both the committed and the aborted bits at once. The final decision depends on the status of the main transaction: if it is aborted, all its subtransactions will be considered aborted too.

The information about subtransactions is stored under the PGDATA/pg_subtrans directory. File access is arranged via buffers that are located in the instance's shared memory and have the same structure as CLOG buffers[1].

Do not confuse subtransactions with autonomous ones. Unlike subtransactions, the latter do not depend on each other in any way. Vanilla PostgreSQL does not support autonomous transactions, and it is probably for the best: they are required in very rare cases, but their availability in other database systems often provokes misuse, which can cause a lot of trouble.

Let's truncate the table, start a new transaction, and insert a row:

```
=> TRUNCATE TABLE t;
=> BEGIN;
=> INSERT INTO t(s) VALUES ('FOO');
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                782
(1 row)
```

Now create a savepoint and insert another row:

```
=> SAVEPOINT sp;
=> INSERT INTO t(s) VALUES ('XYZ');
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                782
(1 row)
```

Note that the pg_current_xact_id function returns the ID of the main transaction, not that of a subtransaction.

[1] backend/access/transam/slru.c

```
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 782  | 0 a  |  2 | FOO
 (0,2) | normal | 783  | 0 a  |  3 | XYZ
(2 rows)
```

Let's roll back to the savepoint and insert the third row:

```
=> ROLLBACK TO sp;

=> INSERT INTO t(s) VALUES ('BAR');
=> SELECT *
FROM heap_page('t',0) p
  LEFT JOIN t ON p.ctid = t.ctid;
 ctid  | state  | xmin | xmax | id |  s
-------+--------+------+------+----+-----
 (0,1) | normal | 782  | 0 a  |  2 | FOO
 (0,2) | normal | 783  | 0 a  |    |
 (0,3) | normal | 784  | 0 a  |  4 | BAR
(3 rows)
```

The page still contains the row added by the aborted subtransaction.

Commit the changes:

```
=> COMMIT;

=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 782 c | 0 a
 (0,2) | normal | 783 a | 0 a
 (0,3) | normal | 784 c | 0 a
(3 rows)
```

Now we can clearly see that each subtransaction has its own status.

SQL does not allow using subtransactions directly, that is, you cannot start a new transaction before completing the current one:

```
=> BEGIN;
BEGIN
=> BEGIN;
WARNING:  there is already a transaction in progress
BEGIN
=> COMMIT;
COMMIT
=> COMMIT;
WARNING:  there is no transaction in progress
COMMIT
```

Subtransactions are employed implicitly: to implement savepoints, handle exceptions in PL/pgSQL, and in some other, more exotic cases.

Errors and Atomicity

What happens if an error occurs during execution of a statement?

```
=> BEGIN;
=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)
=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero
```

After a failure, the whole transaction is considered aborted and cannot perform any further operations:

```
=> SELECT * FROM t;
ERROR:  current transaction is aborted, commands ignored until end
of transaction block
```

And even if you try to commit the changes, PostgreSQL will report that the transaction is rolled back:

```
=> COMMIT;
ROLLBACK
```

Why is it forbidden to continue transaction execution after a failure? Since the already executed operations are never rolled back, we would get access to some changes made before the error—it would break the atomicity of the statement, and hence that of the transaction itself.

For example, in our experiment the operator has managed to update one of the two rows before the failure:

```
=> SELECT * FROM heap_page('t',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | normal | 782 c | 785
 (0,2) | normal | 783 a | 0 a
 (0,3) | normal | 784 c | 0 a
 (0,4) | normal | 785   | 0 a
(4 rows)
```

On a side note, psql provides a special mode that allows you to continue a transaction after a failure as if the erroneous statement were rolled back:

```
=> \set ON_ERROR_ROLLBACK on

=> BEGIN;

=> UPDATE t SET s = repeat('X', 1/(id-4));
ERROR:  division by zero

=> SELECT * FROM t;
 id |  s
----+-----
  2 | FOO
  4 | BAR
(2 rows)

=> COMMIT;
COMMIT
```

As you can guess, psql simply adds an implicit savepoint before each command when run in this mode; in case of a failure, a rollback is initiated. This mode is not used by default because issuing savepoints (even if they are not rolled back to) incurs significant overhead.

4 Snapshots

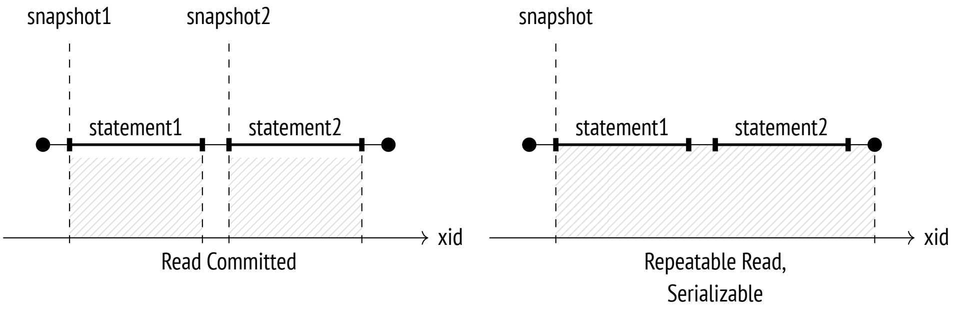
4.1. What is a Snapshot?

A data page can contain several versions of one and the same row, although each transaction must see only one of them at the most. Together, visible versions of all the different rows constitute a *snapshot*. A snapshot includes only the current data committed by the time it was taken, thus providing a consistent (in the ACID sense) view of the data for this particular moment. *p. 43*

To ensure isolation, each transaction uses its own snapshot. It means that different transactions can see different snapshots taken at different points in time, which are nevertheless consistent.

At the Read Committed isolation level, a snapshot is taken at the beginning of each statement, and it remains active only for the duration of this statement.

At the Repeatable Read and Serializable levels, a snapshot is taken at the beginning of the first statement of a transaction, and it remains active until the whole transaction is complete.



4.2. Row Version Visibility

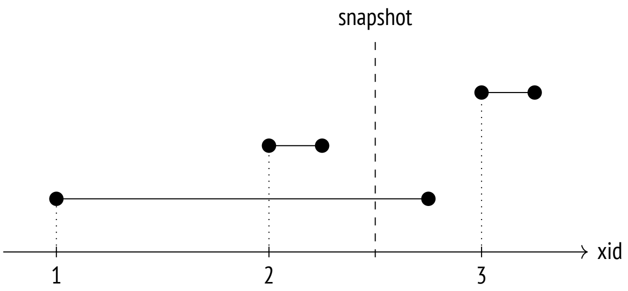
A snapshot is not a physical copy of all the required tuples. Instead, it is defined by several numbers, while tuple visibility is determined by certain rules.

Tuple visibility is defined by xmin and xmax fields of the tuple header (that is, IDs of transactions that perform insertion and deletion) and the corresponding hint bits. Since xmin–xmax intervals do not intersect, each row is represented in any snapshot by only one of its versions.

The exact visibility rules[1] are quite complex, as they take into account a variety of different scenarios and corner cases. Very roughly, we can describe them as follows: a tuple is visible in a snapshot that includes xmin transaction changes but excludes xmax transaction changes (in other words, the tuple has already appeared and has not been deleted yet).

In their turn, transaction changes are visible in a snapshot if this transaction was committed before the snapshot creation. As an exception, transactions can see their own uncommitted changes. If a transaction is aborted, its changes will not be visible in any snapshot.

Let's take a look at a simple example. In this illustration line segments represent transactions (from their start time till commit time):



Here visibility rules are applied to transactions as follows:

- Transaction 2 was committed before the snapshot creation, so its changes are visible.

[1] backend/access/heap/heapam_visibility.c

- Transaction 1 was active at the time of the snapshot creation, so its changes are not visible.
- Transaction 3 was started after the snapshot creation, so its changes are not visible either (it makes no difference whether this transaction is completed or not).

4.3. Snapshot Structure

Unfortunately, the previous illustration has nothing to do with the way PostgreSQL actually sees this picture[1]. The problem is that the system does not know when transactions got committed. It is only known when they were started (this moment is defined by the transaction ID), while their completion is not registered anywhere.

off

Commit times can be tracked[2] if you enable the *track_commit_timestamp* parameter, but they do not participate in visibility checks in any way (although it can still be useful to track them for other purposes, for example, to apply in external replication solutions).

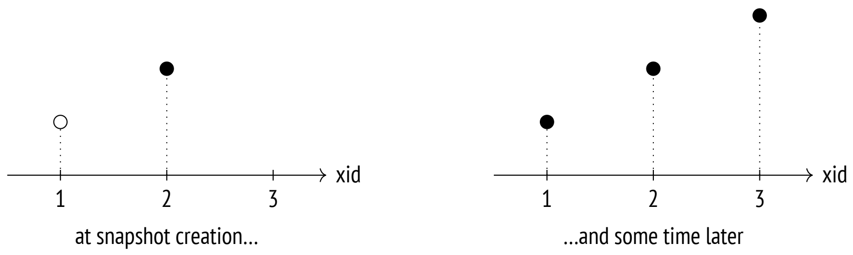
Besides, PostgreSQL always logs commit and rollback times in the corresponding WAL entries, but this information is used only for point-in-time recovery.

It is only the *current* status of a transaction that we can learn. This information is available in the server's shared memory: the ProcArray structure contains the list of all the active sessions and their transactions. Once a transaction is complete, it is impossible to find out whether it was active at the time of the snapshot creation.

So to create a snapshot, it is not enough to register the moment when it was taken: it is also necessary to collect the status of all the transactions at that moment. Otherwise, later it will be impossible to understand which tuples must be visible in the snapshot, and which must be excluded.

Take a look at the information available to the system when the snapshot was taken and some time afterwards (the white circle denotes an active transaction, whereas the black circles stand for completed ones):

[1] include/utils/snapshot.h
backend/utils/time/snapmgr.c
[2] backend/access/transam/commit_ts.c



Suppose we did not know that at the time the snapshot was taken the first transaction was still being executed and the third transaction had not started yet. Then it would seem that they were just like the second transaction (which was committed at that time), and it would be impossible to filter them out.

For this reason, PostgreSQL cannot create a snapshot that shows a consistent state of data at some arbitrary point in the past, even if all the required tuples are present in heap pages. Consequently, it is impossible to implement retrospective queries (which are sometimes also called temporal or flashback queries).

> Intriguingly, such functionality was declared as one of the objectives of Postgres and was implemented at the very start, but it was removed from the database system when the project support was passed on to the community[1].

Thus, a snapshot consists of several values saved at the time of its creation[2]:

xmin is the snapshot's lower boundary, which is represented by the ID of the oldest active transaction.

All the transactions with smaller IDs are either committed (so their changes are included into the snapshot) or aborted (so their changes are ignored). *p. 137*

xmax is the snapshot's upper boundary, which is represented by the value that exceeds the ID of the latest committed transaction by one. The upper boundary defines the moment when the snapshot was taken.

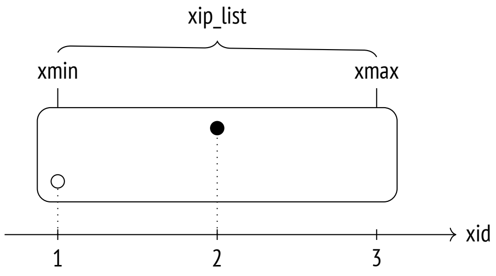
All the transactions whose IDs are equal to or greater than xmax are either still running or do not exist, so their changes cannot be visible.

xip_list is the list of IDs of all the active transactions except for virtual ones, which do not affect visibility in any way. *p. 79*

[1] *Joseph M. Hellerstein*, Looking Back at Postgres. https://arxiv.org/pdf/1901.01973.pdf
[2] backend/storage/ipc/procarray.c, GetSnapshotData function

Snapshots also include several other parameters, but we will ignore them for now.

In a graphical form, a snapshot can be represented as a rectangle that comprises transactions from xmin to xmax:



To understand how visibility rules are defined by the snapshot, we are going to reproduce the above scenario on the accounts table.

```
=> TRUNCATE TABLE accounts;
```

The first transaction inserts the first row into the table and remains open:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (1, 'alice', 1000.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                790
(1 row)
```

The second transaction inserts the second row and commits this change immediately:

```
    => BEGIN;
    => INSERT INTO accounts VALUES (2, 'bob', 100.00);
    => SELECT pg_current_xact_id();
     pg_current_xact_id
    --------------------
                    791
    (1 row)
    => COMMIT;
```

At this point, let's create a new snapshot in another session. We could simply run any query for this purpose, but we will use a special function to take a look at this snapshot right away:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> --  txid_current_snapshot() before v.13
SELECT pg_current_snapshot();
 pg_current_snapshot
---------------------
 790:792:790
(1 row)
```

This function displays the following snapshot components, separated by colons: xmin, xmax, and xip_list (the list of active transactions; in this particular case it consists of a single item).

Once the snapshot is taken, commit the first transaction:

```
=> COMMIT;
```

The third transaction is started after the snapshot creation. It modifies the second row, so a new tuple appears:

```
=> BEGIN;
=> UPDATE accounts SET amount = amount + 100 WHERE id = 2;
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                792
(1 row)
=> COMMIT;
```

Our snapshot sees only one tuple:

```
=> SELECT ctid, * FROM accounts;
 ctid  | id | client | amount
-------+----+--------+--------
 (0,2) |  2 | bob    | 100.00
(1 row)
```

But the table contains three of them:

```
=> SELECT * FROM heap_page('accounts',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 790 c | 0 a
 (0,2) | normal | 791 c | 792 c
 (0,3) | normal | 792 c | 0 a
(3 rows)
```

So how does PostgreSQL choose which versions to show? By the above rules, changes are included into a snapshot only if they are made by committed transactions that satisfy the following criteria:

- If xid < xmin, changes are shown unconditionally (like in the case of the transaction that created the accounts table).
- If xmin ⩽ xid < xmax, changes are shown only if the corresponding transaction IDs are not in xip_list.

The first row (0,1) is invisible because it is inserted by a transaction that appears in xip_list (even though this transaction falls into the snapshot range).

The latest version of the second row (0,3) is invisible because the corresponding transaction ID is above the upper boundary of the snapshot.

But the first version of the second row (0,2) is visible: row insertion was performed by a transaction that falls into the snapshot range and does not appear in xip_list (the insertion is visible), while row deletion was performed by a transaction whose ID is above the upper boundary of the snapshot (the deletion is invisible).

```
=> COMMIT;
```

4.4. Visibility of Transactions' Own Changes

Things get a bit more complicated when it comes to defining visibility rules for transactions' own changes: in some cases, only part of such changes must be visible. For example, a cursor that was opened at a particular point in time must not see any changes that happened later, regardless of the isolation level.

To address such situations, tuple headers provide a special field (displayed as cmin and cmax pseudocolumns) that shows the sequence number of the operation within the transaction. The cmin column identifies insertion, while cmax is used for deletion operations. To save space, these values are stored in a single field of the tuple header rather than in two different ones. It is assumed that one and the same row almost never gets both inserted and deleted within a single transaction. (If it does happen, PostgreSQL writes a special combo identifier into this field, and the actual cmin and cmax[1] values are stored by the backend in this case.)

As an illustration, let's start a transaction and insert a row into the table:

```
=> BEGIN;
=> INSERT INTO accounts VALUES (3, 'charlie', 100.00);
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                793
(1 row)
```

Open a cursor to run the query that returns the number of rows in this table:

```
=> DECLARE c CURSOR FOR SELECT count(*) FROM accounts;
```

Insert one more row:

```
=> INSERT INTO accounts VALUES (4, 'charlie', 200.00);
```

Now extend the output by another column to display the cmin value for the rows inserted by our transaction (it makes no sense for other rows):

```
=> SELECT xmin, CASE WHEN xmin = 793 THEN cmin END cmin, *
FROM accounts;
 xmin | cmin | id | client  | amount
------+------+----+---------+---------
  790 |      |  1 | alice   | 1000.00
  792 |      |  2 | bob     |  200.00
  793 |    0 |  3 | charlie |  100.00
  793 |    1 |  4 | charlie |  200.00
(4 rows)
```

[1] backend/utils/time/combocid.c

The cursor query gets only three rows; the row inserted when the cursor was already open does not make it into the snapshot because the cmin < 1 condition is not satisfied:

```
=> FETCH c;
 count
-------
     3
(1 row)
```

Naturally, this cmin number is also stored in the snapshot, but it is impossible to display it using any SQL means.

4.5. Transaction Horizon

As mentioned earlier, the lower boundary of the snapshot is represented by xmin, which is the ID of the oldest transaction that was active at the moment of the snapshot creation. This value is very important because it defines the *horizon* of the transaction that uses this snapshot.

If a transaction has no active snapshot (for example, at the Read Committed isolation level between statement execution), its horizon is defined by its own ID if it is assigned.

All the transactions that are beyond the horizon (those with xid < xmin) are guranteed to be committed. It means that a transaction can see only the current row versions beyond its horizon.

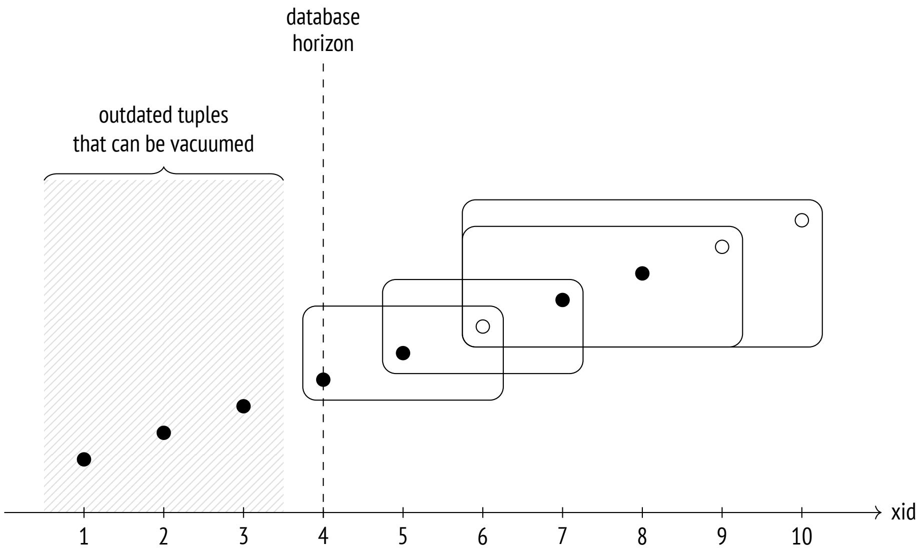
As you can guess, this term is inspired by the concept of *event horizon* in physics.

PostgreSQL tracks the current horizons of all its processes; transactions can see their own horizons in the pg_stat_activity table:

```
=> BEGIN;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 row)
```

Virtual transactions have no real IDs, but they still use snapshots just like regular transactions, so they have their own horizons. The only exception is virtual transactions without an active snapshot: the concept of the horizon makes no sense for them, and they are fully “transparent” to the system when it comes to snapshots and visibility (even though pg_stat_activity.backend_xmin may still contain an xmin of an old snapshot).

We can also define the *database horizon* in a similar manner. For this purpose, we should take the horizons of all the transactions in this database and select the most remote one, which has the oldest xmin[1]. Beyond this horizon, outdated heap tuples will never be visible to any transaction in this database. *Such tuples can be safely cleaned up by vacuum*—this is exactly why the concept of the horizon is so important from a practical standpoint.



Let’s draw some conclusions:

- If a transaction (no matter whether it is real or virtual) at the Repeatable Read or Serializable isolation level is running for a long time, it thereby holds the database horizon and defers vacuuming.

[1] backend/storage/ipc/procarray.c, ComputeXidHorizons function

- A real transaction at the Read Committed isolation level holds the database horizon in the same way, even if it is not executing any operators (being in the "idle in transaction" state).
- A virtual transaction at the Read Committed isolation level holds the horizon only while executing operators.

There is only one horizon for the whole database, so if it is being held by a transaction, it is impossible to vacuum any data within this horizon—even if this data has not been accessed by this transaction.

Cluster-wide tables of the system catalog have a separate horizon that takes into account all transactions in all databases. Temporary tables, on the contrary, do not have to pay attention to any transactions except those that are being executed by the current process.

Let's get back to our current experiment. The active transaction of the first session still holds the database horizon; we can see it by incrementing the transaction counter:

```
=> SELECT pg_current_xact_id();
 pg_current_xact_id
--------------------
                794
(1 row)
```

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          793
(1 row)
```

And only when this transaction is complete, the horizon moves forward, and outdated tuples can be vacuumed:

```
=> COMMIT;
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          795
(1 row)
```

In a perfect world, you should avoid combining long transactions with frequent updates (that spawn new row versions), as it will lead to table and index bloating. *p. 157*

4.6. System Catalog Snapshots

Although the system catalog consists of regular tables, they cannot be accessed via a snapshot used by a transaction or an operator. The snapshot must be "fresh" enough to include all the latest changes, otherwise transactions could see outdated definitions of table columns or miss newly added integrity constraints.

Here is a simple example:

```
=> BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;

=> SELECT 1; -- a snapshot for the transaction is taken

  |  => ALTER TABLE accounts
  |    ALTER amount SET NOT NULL;

=> INSERT INTO accounts(client, amount)
  VALUES ('alice', NULL);
ERROR:  null value in column "amount" of relation "accounts"
violates not-null constraint
DETAIL:  Failing row contains (1, alice, null).

=> ROLLBACK;
```

The integrity constraint that appeared after the snapshot creation was visible to the INSERT command. It may seem that such behavior breaks isolation, but if the inserting transaction had accessed the accounts table, the ALTER TABLE command would have been blocked until this transaction completion.

In general, the server behaves as if a separate snapshot is created for each system catalog query. But the implementation[1] is, of course, much more complex since frequent snapshot creation would negatively affect performance; besides, many system catalog objects get cached, and it must also be taken into account.

[1] backend/utils/time/snapmgr.c, GetCatalogSnapshot function

4.7. Exporting Snapshots

In some situations, concurrent transactions must see one and the same snapshot by all means. For example, if the pg_dump utility is run in the parallel mode, all its processes must see the same database state to produce a consistent backup.

We cannot assume that snapshots will be identical simply because transactions were started "simultaneously." To ensure that all the transactions see the same data, we must employ the snapshot export mechanism.

The pg_export_snapshot function returns a snapshot ID, which can be passed to another transaction (outside of the database system):

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)


=> SELECT pg_export_snapshot();
 pg_export_snapshot
---------------------
 00000004-0000006E-1
(1 row)
```

Before executing the first statement, the other transaction can import the snapshot by running the SET TRANSACTION SNAPSHOT command. The isolation level must be set to Repeatable Read or Serializable because operators use their own snapshots at the Read Committed level:

```
    => DELETE FROM accounts;
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SET TRANSACTION SNAPSHOT '00000004-0000006E-1';
```

Now the second transaction is going to use the snapshot of the first transaction, and consequently, it will see four rows (instead of zero):

```
=> SELECT count(*) FROM accounts;
 count
-------
     4
(1 row)
```

Clearly, the second transaction will not see any changes made by the first transaction after the snapshot export (and vice versa): regular visibility rules still apply.

The exported snapshot's lifetime is the same as that of the exporting transaction.

```
=> COMMIT;
```

```
=> COMMIT;
```

5

Page Pruning and HOT Updates

5.1. Page Pruning

While a heap page is being read or updated, PostgreSQL can perform some quick page cleanup, or *pruning*[1]. It happens in the following cases:

- The previous UPDATE operation did not find enough space to place a new tuple into the same page. This event is reflected in the page header.
- 100 The heap page contains more data than allowed by the *fillfactor* storage parameter.

  An INSERT operation can add a new row into the page only if this page is filled for less than *fillfactor* percent. The rest of the space is kept for UPDATE operations (no such space is reserved by default).

Page pruning removes the tuples that cannot be visible in any snapshot anymore *p. 95* (that is, that are beyond the database horizon). It never goes beyond a single heap page, but in return it is performed very fast. Pointers to pruned tuples remain in place since they may be referenced from an index—which is already a different page.

For the same reason, neither the visibility map nor the free space map is refreshed (so the recovered space is set aside for updates, not for insertions).

Since a page can be pruned during reads, any SELECT statement can cause page modifications. This is yet another such case in addition to deferred setting of in- *p. 74* formation bits.

[1] backend/access/heap/pruneheap.c, heap_page_prune_opt function

Let's take a look at how page pruning actually works. We are going to create a two-column table and build an index on each of the columns:

```
=> CREATE TABLE hot(id integer, s char(2000)) WITH (fillfactor = 75);
=> CREATE INDEX hot_id ON hot(id);
=> CREATE INDEX hot_s ON hot(s);
```

If the s column contains only Latin letters, each heap tuple will have a fixed size of 2004 bytes, plus 24 bytes of the header. The *fillfactor* storage parameter is set to 75 %. It means that the page has enough free space for four tuples, but we can insert only three.

Let's insert a new row and update it several times:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
```

Now the page contains four tuples:

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+-------
 (0,1) | normal | 801 c | 802 c
 (0,2) | normal | 802 c | 803 c
 (0,3) | normal | 803 c | 804
 (0,4) | normal | 804   | 0 a
(4 rows)
```

Expectedly, we have just exceeded the *fillfactor* threshold. You can tell it by the difference between the pagesize and upper values—it is bigger than 75 % of the page size, which is 6144 bytes: *p. 64*

```
=> SELECT upper, pagesize FROM page_header(get_raw_page('hot',0));
 upper | pagesize
-------+----------
    64 |     8192
(1 row)
```

The next page access triggers page pruning that removes all the outdated tuples. Then a new tuple (0,5) is added into the freed space:

```
=> UPDATE hot SET s = 'E';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax
-------+--------+-------+------
 (0,1) | dead   |       |
 (0,2) | dead   |       |
 (0,3) | dead   |       |
 (0,4) | normal | 804 c | 805
 (0,5) | normal | 805   | 0 a
(5 rows)
```

The remaining heap tuples are physically moved towards the highest addresses so that all the free space is aggregated into a single continuous chunk. The tuple pointers are also modified accordingly. As a result, there is no free space fragmentation in the page.

The pointers to the pruned tuples cannot be removed yet because they are still referenced from the indexes; PostgreSQL changes their status from normal to dead. Let's take a look at the first page of the hot_s index (the zero page is used for metadata):

```
=> SELECT * FROM index_page('hot_s',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

We can see the same picture in the other index too:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid
------------+-------
          1 | (0,1)
          2 | (0,2)
          3 | (0,3)
          4 | (0,4)
          5 | (0,5)
(5 rows)
```

An index scan can return (0,1), (0,2), and (0,3) as tuple identifiers. The server tries to read the corresponding heap tuple but sees that the pointer has the dead status; it means that this tuple does not exist anymore and should be ignored. And while being at it, the server also changes the pointer status in the index page to avoid repeated heap page access[1].

Let's extend the function displaying index pages so that it also shows whether the pointer is dead: v. 13

```
=> DROP FUNCTION index_page(text, integer);

=> CREATE FUNCTION index_page(relname text, pageno integer)
RETURNS TABLE(itemoffset smallint, htid tid, dead boolean)
AS $$
SELECT itemoffset,
       htid,
       dead -- starting from v.13
FROM bt_page_items(relname,pageno);
$$ LANGUAGE sql;

=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
          4 | (0,4) | f
          5 | (0,5) | f
(5 rows)
```

All the pointers in the index page are active so far. But as soon as the first index scan occurs, their status changes:

```
=> EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM hot WHERE id = 1;
                        QUERY PLAN
--------------------------------------------------------
 Index Scan using hot_id on hot (actual rows=1 loops=1)
   Index Cond: (id = 1)
(2 rows)
```

[1] backend/access/index/indexam.c, index_fetch_heap function

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | t
          2 | (0,2) | t
          3 | (0,3) | t
          4 | (0,4) | t
          5 | (0,5) | f
(5 rows)
```

Although the heap tuple referenced by the fourth pointer is still unpruned and has the normal status, it is already beyond the database horizon. That's why this pointer is also marked as dead in the index.

5.2. HOT Updates

It would be very inefficient to keep references to all heap tuples in an index.

To begin with, each row modification triggers updates of *all* the indexes created on the table: once a new heap tuple appears, each index must include a reference to this tuple, even if the modified fields are not indexed.

Furthermore, indexes accumulate references to historic heap tuples, so they have to be pruned together with these tuples. *p. 112*

Things get worse as you create more indexes on a table.

But if the updated column is not a part of *any* index, there is no point in creating another index entry that contains the same key value. To avoid such redundancies, PostgreSQL provides an optimization called *Heap-Only Tuple updates*[1].

If such an update is performed, an index page contains only one entry for each row. This entry points to the very first row version; all the subsequent versions located in the same page are bound into a chain by ctid pointers in the tuple headers.

Row versions that are not referenced from any index are tagged with the Heap-Only Tuple bit. If a version is included into the HOT chain, it is tagged with the Heap Hot Updated bit.

[1] backend/access/heap/README.HOT

If an index scan accesses a heap page and finds a row version marked as Heap Hot Updated, it means that the scan should continue, so it goes further along the chain of HOT updates. Obviously, all the fetched row versions are checked for visibility before the result is returned to the client.

To take a look at how HOT updates are performed, let's delete one of the indexes and truncate the table.

```
=> DROP INDEX hot_s;
=> TRUNCATE TABLE hot;
```

For convenience, we will redefine the heap_page function so that its output includes three more fields: ctid and the two bits related to HOT updates:

```
=> DROP FUNCTION heap_page(text,integer);
=> CREATE FUNCTION heap_page(relname text, pageno integer)
RETURNS TABLE(
  ctid tid, state text,
  xmin text, xmax text,
  hhu text, hot text, t_ctid tid
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax,
       CASE WHEN (t_infomask2 & 16384) > 0 THEN 't' END AS hhu,
       CASE WHEN (t_infomask2 & 32768) > 0 THEN 't' END AS hot,
       t_ctid
FROM heap_page_items(get_raw_page(relname,pageno))
ORDER BY lp;
$$ LANGUAGE sql;
```

Let's repeat the insert and update operations:

```
=> INSERT INTO hot VALUES (1, 'A');
=> UPDATE hot SET s = 'B';
```

The page now contains a chain of HOT updates:

- The Heap Hot Updated bit shows that the executor should follow the CTID chain.
- The Heap Only Tuple bit indicates that this tuple is not referenced from any indexes.

```
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | normal | 812 c | 813  | t   |     | (0,2)
 (0,2) | normal | 813   | 0 a  |     | t   | (0,2)
(2 rows)
```

As we make further updates, the chain will grow—but only within the page limits:

```
=> UPDATE hot SET s = 'C';
=> UPDATE hot SET s = 'D';
=> SELECT * FROM heap_page('hot',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 812 c | 813 c | t   |     | (0,2)
 (0,2) | normal | 813 c | 814 c | t   | t   | (0,3)
 (0,3) | normal | 814 c | 815   | t   | t   | (0,4)
 (0,4) | normal | 815   | 0 a   |     | t   | (0,4)
(4 rows)
```

The index still contains only one reference, which points to the head of this chain:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
(1 row)
```

A HOT update is possible if the modified fields are not a part of *any* index. Otherwise, some of the indexes would contain a reference to a heap tuple that appears in the middle of the chain, which contradicts the idea of this optimization. Since a HOT chain can grow only within a single page, traversing the whole chain never requires access to other pages and thus does not hamper performance.

5.3. Page Pruning for HOT Updates

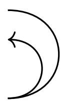
A special case of page pruning—which is nevertheless important—is pruning of HOT update chains.

In the example above, the *fillfactor* threshold is already exceeded, so the next update should trigger page pruning. But this time the page contains a chain of HOT updates. The head of this chain must always remain in its place since it is referenced from the index, but other pointers can be released because they are sure to have no external references.

To avoid moving the head, PostgreSQL uses dual addressing: the pointer referenced from the index (which is (0,1) in this case) receives the redirect status since it points to the tuple that currently starts the chain:

```
=> UPDATE hot SET s = 'E';

=> SELECT * FROM heap_page('hot',0);
 ctid  |      state      | xmin  | xmax | hhu | hot | t_ctid
-------+-----------------+-------+------+-----+-----+--------
 (0,1) | redirect to 4 |       |      |     |     |
 (0,2) | normal        | 816   | 0 a  |     | t   | (0,2)
 (0,3) | unused        |       |      |     |     |
 (0,4) | normal        | 815 c | 816  | t   | t   | (0,2)
(4 rows)
```



The tuples (0,1), (0,2), and (0,3) have been pruned; the head pointer 1 remains for redirection purposes, while pointers 2 and 3 have been deallocated (received the unused status) since they are guaranteed to have no references from indexes. The new tuple is written into the freed space as tuple (0,2).

Let’s perform some more updates:

```
=> UPDATE hot SET s = 'F';
```

```
=> UPDATE hot SET s = 'G';
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state      | xmin  | xmax  | hhu | hot | t_ctid
-------+----------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 4  |       |       |     |     |
 (0,2) | normal         | 816 c | 817 c | t   | t   | (0,3)
 (0,3) | normal         | 817 c | 818   | t   | t   | (0,5)
 (0,4) | normal         | 815 c | 816 c | t   | t   | (0,2)
 (0,5) | normal         | 818   | 0 a   |     | t   | (0,5)
(5 rows)
```

The next update is going to trigger page pruning:

```
=> UPDATE hot SET s = 'H';
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state      | xmin  | xmax | hhu | hot | t_ctid
-------+----------------+-------+------+-----+-----+--------
 (0,1) | redirect to 5  |       |      |     |     |
 (0,2) | normal         | 819   | 0 a  |     | t   | (0,2)
 (0,3) | unused         |       |      |     |     |
 (0,4) | unused         |       |      |     |     |
 (0,5) | normal         | 818 c | 819  | t   | t   | (0,2)
(5 rows)
```

Again, some of the tuples are pruned, and the pointer to the head of the chain is shifted accordingly.

If unindexed columns are modified frequently, it makes sense to reduce the *fillfactor* value, thus reserving some space in the page for updates. Obviously, you have to keep in mind that the lower the *fillfactor* value is, the more free space is left in the page, so the physical size of the table grows.

5.4. HOT Chain Splits

If the page has no more space to accommodate a new tuple, the chain will be cut off. PostgreSQL will have to add a separate index entry to refer to the tuple located in another page.

To observe this situation, let's start a concurrent transaction with a snapshot that blocks page pruning:

```
    => BEGIN ISOLATION LEVEL REPEATABLE READ;
    => SELECT 1;
```

Now we are going to perform some updates in the first session:

```
=> UPDATE hot SET s = 'I';

=> UPDATE hot SET s = 'J';

=> UPDATE hot SET s = 'K';

=> SELECT * FROM heap_page('hot',0);
 ctid  |     state      | xmin  | xmax  | hhu | hot | t_ctid
-------+----------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2  |       |       |     |     |
 (0,2) | normal         | 819 c | 820 c | t   | t   | (0,3)
 (0,3) | normal         | 820 c | 821 c | t   | t   | (0,4)
 (0,4) | normal         | 821 c | 822   | t   | t   | (0,5)
 (0,5) | normal         | 822   | 0 a   |     | t   | (0,5)
(5 rows)
```

When the next update happens, this page will not be able to accommodate another tuple, and page pruning will not manage to free any space:

```
=> UPDATE hot SET s = 'L';
```

```
    => COMMIT; -- the snapshot is not required anymore
```

```
=> SELECT * FROM heap_page('hot',0);
 ctid  |     state      | xmin  | xmax  | hhu | hot | t_ctid
-------+----------------+-------+-------+-----+-----+--------
 (0,1) | redirect to 2  |       |       |     |     |
 (0,2) | normal         | 819 c | 820 c | t   | t   | (0,3)
 (0,3) | normal         | 820 c | 821 c | t   | t   | (0,4)
 (0,4) | normal         | 821 c | 822 c | t   | t   | (0,5)
 (0,5) | normal         | 822 c | 823   |     | t   | (1,1)
(5 rows)
```

Tuple (0,5) contains the (1,1) reference that goes to page 1:

```
=> SELECT * FROM heap_page('hot',1);
 ctid  | state  | xmin | xmax | hhu | hot | t_ctid
-------+--------+------+------+-----+-----+--------
 (1,1) | normal | 823  | 0 a  |     |     | (1,1)
(1 row)
```

However, this reference is not used: the Heap Hot Updated bit is not set for tuple (0,5). As for tuple (1,1), it can be accessed from the index that now has two entries. Each of them points to the head of their own HOT chain:

```
=> SELECT * FROM index_page('hot_id',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (1,1) | f
(2 rows)
```

5.5. Page Pruning for Indexes

I have declared that page pruning is confined to a single heap page and does not affect indexes. However, indexes have their own pruning[1], which also cleans up a single page—an index one in this case.

Index pruning happens when an insertion into a B-tree is about to split the page into two, as the original page does not have enough space anymore. The problem is that even if some index entries are deleted later, two separate index pages will not

[1] postgresql.org/docs/14/btree-implementation.html#BTREE-DELETION

be merged into one. It leads to index bloating, and once bloated, the index cannot shrink even if a large part of the data is deleted. But if pruning can remove some of the tuples, a page split may be deferred.

There are two types of tuples that can be pruned from an index.

First of all, PostgreSQL prunes those tuples that have been tagged as dead[1]. As I have already said, PostgreSQL sets such a tag during an index scan if it detects an index entry pointing to a tuple that is not visible in any snapshot anymore or simply does not exist.

If no tuples are known to be dead, PostgreSQL checks those index entries that reference different versions of one and the same table row[2]. Because of MVCC, update operations may generate a large number of row versions, and many of them are soon likely to disappear behind the database horizon. HOT updates cushion this effect, but they are not always applicable: if the column to update is a part of an index, the corresponding references are propagated to all the indexes. Before splitting the page, it makes sense to search for the rows that are not tagged as dead yet but can already be pruned. To achieve this, PostgreSQL has to check visibility of heap tuples. Such checks require table access, so they are performed only for "promising" index tuples, which have been created as copies of the existing ones for MVCC purposes. It is cheaper to perform such a check than to allow an extra page split. v. 14

[1] backend/access/nbtree/README, Simple deletion section
[2] backend/access/nbtree/README, Bottom-Up deletion section
include/access/tableam.h

6

Vacuum and Autovacuum

6.1. Vacuum

Page pruning happens very fast, but it frees only part of the space that can be potentially reclaimed. Operating within a single heap page, it does not touch upon indexes (or vice versa, it cleans up an index page without affecting the table).

Routine vacuuming[1], which is the main vacuuming procedure, is performed by the VACUUM [2] command. It processes the whole table and eliminates both outdated heap tuples and all the corresponding index entries.

Vacuuming is performed in parallel with other processes in the database system. While being vacuumed, tables and indexes can be used in the usual manner, both for read and write operations (but concurrent execution of such commands as CREATE INDEX, ALTER TABLE, and some others is not allowed).

p. 23 To avoid scanning extra pages, PostgreSQL uses a visibility map. Pages tracked in this map are skipped since they are sure to contain only the current tuples, so a page will only be vacuumed if it does not appear in this map. If all the tuples remaining in a page after vacuuming are beyond the database horizon, the visibility map is refreshed to include this page.

The free space map also gets updated to reflect the space that has been cleared.

Let's create a table with an index on it:

[1] postgresql.org/docs/14/routine-vacuuming.html
[2] postgresql.org/docs/14/sql-vacuum.html
backend/commands/vacuum.c

```
=> CREATE TABLE vac(
  id integer,
  s char(100)
)
WITH (autovacuum_enabled = off);
```

```
=> CREATE INDEX vac_s ON vac(s);
```

The *autovacuum_enabled* storage parameter turns off autovacuum; we are doing it here solely for the purpose of experimentation to precisely control vacuuming start time.

Let's insert a row and make a couple of updates:

```
=> INSERT INTO vac(id,s) VALUES (1,'A');
```

```
=> UPDATE vac SET s = 'B';
```

```
=> UPDATE vac SET s = 'C';
```

Now the table contains three tuples:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | normal | 826 c | 827 c |     |     | (0,2)
 (0,2) | normal | 827 c | 828   |     |     | (0,3)
 (0,3) | normal | 828   | 0 a   |     |     | (0,3)
(3 rows)
```

Each tuple is referenced from the index:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,1) | f
          2 | (0,2) | f
          3 | (0,3) | f
(3 rows)
```

Vacuuming has removed all the dead tuples, leaving only the current one:

```
=> VACUUM vac;
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 828 c | 0 a  |     |     | (0,3)
(3 rows)
```

In the case of page pruning, the first two pointers would be considered dead, but here they have the unused status since no index entries are referring to them now:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

Pointers with the unused status are treated as free and can be reused by new row versions.

Now the heap page appears in the visibility map; we can check it using the pg_visibility extension:

```
=> CREATE EXTENSION pg_visibility;
=> SELECT all_visible
FROM pg_visibility_map('vac',0);
 all_visible
-------------
 t
(1 row)
```

The page header has also received an attribute showing that all its tuples are visible in all snapshots:

```
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

6.2. Database Horizon Revisited

Vacuuming detects dead tuples based on the database horizon. This concept is so fundamental that it makes sense to get back to it once again.

Let's restart our experiment from the very beginning:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s) VALUES (1,'A');
=> UPDATE vac SET s = 'B';
```

But this time, before updating the row, we are going to open another transaction that will hold the database horizon (it can be almost any transaction, except for a virtual one executed at the Read Committed isolation level). For example, this transaction can modify some rows in *another* table. *p. 95*

```
    => BEGIN;
    => UPDATE accounts SET amount = 0;
```

```
=> UPDATE vac SET s = 'C';
```

Now our table contains three tuples, and the index contains three references. Let's vacuum the table and see what changes:

```
=> VACUUM vac;
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax  | hhu | hot | t_ctid
-------+--------+-------+-------+-----+-----+--------
 (0,1) | unused |       |       |     |     |
 (0,2) | normal | 833 c | 835 c |     |     | (0,3)
 (0,3) | normal | 835 c | 0 a   |     |     | (0,3)
(3 rows)
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,2) | f
          2 | (0,3) | f
(2 rows)
```

While the previous run left only one tuple in the page, now we have two of them: VACUUM has decided that version (0,2) cannot be removed yet. The reason is the database horizon, which is defined by an unfinished transaction in this case:

```
=> SELECT backend_xmin FROM pg_stat_activity
WHERE pid = pg_backend_pid();
 backend_xmin
--------------
          834
(1 row)
```

We can use the VERBOSE clause when calling VACUUM to observe what is going on:

```
=> VACUUM VERBOSE vac;
INFO:  vacuuming "public.vac"
INFO:  table "vac": found 0 removable, 2 nonremovable row versions
in 1 out of 1 pages
DETAIL:  1 dead row versions cannot be removed yet, oldest xmin: 834
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

The output shows the following information:

- VACUUM has detected no tuples that can be removed (0 REMOVABLE).
- Two tuples must not be removed (2 NONREMOVABLE).
- One of the nonremovable tuples is dead (1 DEAD), the other is in use.
- The current horizon respected by VACUUM (OLDEST XMIN) is the horizon of the active transaction.

Once the active transaction completes, the database horizon moves forward, and vacuuming can continue:

```
=> COMMIT;
=> VACUUM VERBOSE vac;
INFO:  vacuuming "public.vac"
INFO:  scanned index "vac_s" to remove 1 row versions
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  table "vac": removed 1 dead item identifiers in 1 pages
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
INFO:  index "vac_s" now contains 1 row versions in 2 pages
DETAIL:  1 index row versions were removed.
0 index pages were newly deleted.
0 index pages are currently deleted, of which 0 are currently
reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  table "vac": found 1 removable, 1 nonremovable row versions
in 1 out of 1 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 836
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

VACUUM has detected and removed a dead tuple beyond the new database horizon.

Now the page contains no outdated row versions; the only version remaining is the current one:

```
=> SELECT * FROM heap_page('vac',0);
 ctid  | state  | xmin  | xmax | hhu | hot | t_ctid
-------+--------+-------+------+-----+-----+--------
 (0,1) | unused |       |      |     |     |
 (0,2) | unused |       |      |     |     |
 (0,3) | normal | 835 c | 0 a  |     |     | (0,3)
(3 rows)
```

The index also contains only one entry:

```
=> SELECT * FROM index_page('vac_s',1);
 itemoffset | htid  | dead
------------+-------+------
          1 | (0,3) | f
(1 row)
```

6.3. Vacuum Phases

The mechanism of vacuuming seems quite simple, but this impression is misleading. After all, both tables and indexes have to be processed concurrently, without blocking other processes. To enable such operation, vacuuming of each table is carried out in several phases[1].

It all starts with scanning a table in search of dead tuples; if found, they are first removed from indexes and then from the table itself. If too many dead tuples have to be vacuumed in one go, this process is repeated. Eventually, heap truncation may be performed.

Heap Scan

In the first phase, a *heap scan*[2] is performed. The scanning process takes the visibility map into account: all pages tracked in this map are skipped because they are sure to contain no outdated tuples. If a tuple is beyond the horizon and is not required anymore, its ID is added to a special tid array. Such tuples cannot be removed yet because they may still be referenced from indexes.

The tid array resides in the local memory of the VACUUM process; the size of the allocated memory chunk is defined by the *maintenance_work_mem* parameter. The whole chunk is allocated at once rather than on demand. However, the allocated memory never exceeds the volume required in the worst-case scenario, so if the table is small, vacuuming may use less memory than specified in this parameter.

64MB

Index Vacuuming

The first phase can have two outcomes: either the table is scanned in full, or the memory allocated for the tid array is filled up before this operation completes. In any case, *index vacuuming*[3] begins. In this phase, *each* of the indexes created on

[1] backend/access/heap/vacuumlazy.c, heap_vacuum_rel function
[2] backend/access/heap/vacuumlazy.c, lazy_scan_heap function
[3] backend/access/heap/vacuumlazy.c, lazy_vacuum_all_indexes function

the table is *fully scanned* to find all the entries that refer to the tuples registered in the tid array. These entries are removed from index pages.

> An index can help you quickly get to a heap tuple by its index key, but there is no way to quickly find an index entry by the corresponding tuple ID. This functionality is currently being implemented for B-trees[1], but this work is not completed yet.

If there are several indexes bigger than the *min_parallel_index_scan_size* value, they can be vacuumed by background workers running in parallel. Unless the level of parallelism is explicitly defined by the parallel *N* clause, VACUUM launches one worker per suitable index (within the general limits imposed on the number of background workers)[2]. One index cannot be processed by several workers. 512kB v. 13

During the index vacuuming phase, PostgreSQL updates the free space map and calculates statistics on vacuuming. However, this phase is skipped if rows are only inserted (and are neither deleted nor updated) because the table contains no dead tuples in this case. Then an index scan will be forced only once at the very end, as part of a separate phase of *index cleanup*[3].

The index vacuuming phase leaves no references to outdated heap tuples in indexes, but the tuples themselves are still present in the table. It is perfectly normal: index scans cannot find any dead tuples, while sequential scans of the table rely on visibility rules to filter them out.

Heap Vacuuming

Then the *heap vacuuming*[4] phase begins. The table is scanned again to remove the tuples registered in the tid array and free the corresponding pointers. Now that all the related index references have been removed, it can be done safely.

The space recovered by VACUUM is reflected in the free space map, while the pages that now contain only the current tuples visible in all snapshots are tagged in the visibility map.

[1] commitfest.postgresql.org/21/1802
[2] postgresql.org/docs/14/bgworker.html
[3] backend/access/heap/vacuumlazy.c, lazy_cleanup_all_indexes function
backend/access/nbtree/nbtree.c, btvacuumcleanup function
[4] backend/access/heap/vacuumlazy.c, lazy_vacuum_heap function

If the table was not read in full during the heap scan phase, the tid array is cleared, and the heap scan is resumed from where it left off last time.

Heap Truncation

Vacuumed heap pages contain some free space; occasionally, you may be lucky to clear the whole page. If you get several empty pages at the end of the file, vacuuming can “bite off” this tail and return the reclaimed space to the operating system. It happens during *heap truncation*[1], which is the final vacuum phase.

Heap truncation requires a short exclusive lock on the table. To avoid holding other processes for too long, attempts to acquire a lock do not exceed five seconds.

Since the table has to be locked, truncation is only performed if the empty tail takes at least $\frac{1}{16}$ of the table or has reached the length of 1,000 pages. These thresholds are hardcoded and cannot be configured.

v. 12 If, despite all these precautions, table locks still cause any issues, truncation can be disabled altogether using the *vacuum_truncate* and *toast.vacuum_truncate* storage parameters.

6.4. Analysis

When talking about vacuuming, we have to mention yet another task that is closely related to it, even though there is no formal connection between them. It is *analysis*[2], or gathering statistical information for the query planner. The collected statistics include the number of rows (pg_class.reltuples) and pages (pg_class.relpages) in relations, data distribution within columns, and some other information.

You can run the analysis manually using the ANALYZE[3] command, or combine it with vacuuming by calling VACUUM ANALYZE. However, these two tasks are still performed sequentially, so there is no difference in terms of performance.

1 backend/access/heap/vacuumlazy.c, lazy_truncate_heap function
2 postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-STATISTICS
3 backend/commands/analyze.c

Historically, VACUUM ANALYZE appeared first, in version 6.1, while a separate ANALYZE command was not implemented until version 7.2. In earlier versions, statistics were collected by a TCL script.

Automatic vacuum and analysis are set up in a similar way, so it makes sense to discuss them together.

6.5. Automatic Vacuum and Analysis

Unless the database horizon is held up for a long time, routine vacuuming should cope with its work. But how often do we need to call the VACUUM command?

If a frequently updated table is vacuumed too seldom, it will grow bigger than desired. Besides, it may accumulate too many changes, and then the next VACUUM run will have to make several passes over the indexes.

If the table is vacuumed too often, the server will be busy with maintenance instead of useful work.

Furthermore, typical workloads may change over time, so having a fixed vacuuming schedule will not help anyway: the more often the table is updated, the more often it has to be vacuumed.

This problem is solved by *autovacuum*[1], which launches vacuum and analysis processes based on the intensity of table updates.

About the Autovacuum Mechanism

When autovacuum is enabled (*autovacuum* configuration parameter is on), the autovacuum launcher process is always running in the system. This process defines the autovacuum schedule and maintains the list of "active" databases based on usage statistics. Such statistics are collected if the *track_counts* parameter is enabled. Do not switch off these parameters, otherwise autovacuum will not work.

on

on

[1] postgresql.org/docs/14/routine-vacuuming.html#AUTOVACUUM

1min Once in *autovacuum_naptime*, the autovacuum launcher starts an autovacuum worker[1] for each active database in the list (these workers are spawned by postmaster, as usual). Consequently, if there are *N* active databases in the cluster, *N* workers are spawned within the *autovacuum_naptime* interval. But the total number of autovacuum workers running in parallel cannot exceed the threshold defined
3 by the *autovacuum_max_workers* parameter.

> Autovacuum workers are very similar to regular background workers, but they appeared much earlier than this general mechanism of task management. It was decided to leave the autovacuum implementation unchanged, so autovacuum workers do not use *max_worker_processes* slots.

Once started, the background worker connects to the specified database and builds two lists:

- the list of all tables, materialized views, and TOAST tables to be vacuumed
- the list of all tables and materialized views to be analyzed (TOAST tables are not analyzed because they are always accessed via an index)

Then the selected objects are vacuumed or analyzed one by one (or undergo both operations), and once the job is complete, the worker is terminated.

Automatic vacuuming works similar to the manual one initiated by the VACUUM command, but there are some nuances:

- Manual vacuuming accumulates tuple IDs in a memory chunk of the *maintenance_work_mem* size. However, using the same limit for autovacuum is undesirable, as it can result in excessive memory consumption: there may be several autovacuum workers running in parallel, and each of them will get *maintenance_work_mem* of memory at once. Instead, PostgreSQL provides a separate memory limit for autovacuum processes, which is defined by the *autovacuum_work_mem* parameter.

  -1 By default, the *autovacuum_work_mem* parameter falls back on the regular *maintenance_work_mem* limit, so if the *autovacuum_max_workers* value is high, you may have to adjust the *autovacuum_work_mem* value accordingly.

[1] backend/postmaster/autovacuum.c

- Concurrent processing of several indexes created on one table can be performed only by manual vacuuming; using autovacuum for this purpose would result in a large number of parallel processes, so it is not allowed.

If a worker fails to complete all the scheduled tasks within the *autovacuum_naptime* interval, the autovacuum launcher spawns another worker to be run in parallel in that database. The second worker will build its own lists of objects to be vacuumed and analyzed and will start processing them. There is no parallelism at the table level; only *different* tables can be processed concurrently.

Which Tables Need to be Vacuumed?

You can disable autovacuum at the table level—although it is hard to imagine why it could be necessary. There are two storage parameters provided for this purpose, one for regular tables and the other for TOAST tables:

- *autovacuum_enabled*
- *toast.autovacuum_enabled*

In usual circumstances, autovacuum is triggered either by accumulation of dead tuples or by insertion of new rows. *p. 143*

Dead tuple accumulation. Dead tuples are constantly being counted by the statistics collector; their current number is shown in the system catalog table called pg_stat_all_tables.

It is assumed that dead tuples have to be vacuumed if they exceed the threshold defined by the following two parameters:

- *autovacuum_vacuum_threshold*, which specifies the number of dead tuples (an absolute value) 50
- *autovacuum_vacuum_scale_factor*, which sets the fraction of dead tuples in a table 0.2

Vacuuming is required if the following condition is satisfied:

> pg_stat_all_tables.n_dead_tup >
> *autovacuum_vacuum_threshold* +
> *autovacuum_vacuum_scale_factor* × pg_class.reltuples

The main parameter here is of course *autovacuum_vacuum_scale_factor*: its value is important for large tables (and it is large tables that are likely to cause the majority of issues). The default value of 20 % seems too big and may have to be significantly reduced.

For different tables, optimal parameter values may vary: they largely depend on the table size and workload type. It makes sense to set more or less adequate initial values and then override them for particular tables using storage parameters:

- *autovacuum_vacuum_threshold* and *toast.autovacuum_vacuum_threshold*
- *autovacuum_vacuum_scale_factor* and *toast.autovacuum_vacuum_scale_factor*

v. 13 **Row insertions.** If rows are only inserted and are neither deleted nor updated, the table contains no dead tuples. But such tables should also be vacuumed to freeze *p. 137* heap tuples in advance and update the visibility map (thus enabling index-only scans).

A table will be vacuumed if the number of rows inserted since the previous vacuuming exceeds the threshold defined by another similar pair of parameters:

- 1000 *autovacuum_vacuum_insert_threshold*
- 0.2 *autovacuum_vacuum_insert_scale_factor*

The formula is as follows:

> pg_stat_all_tables.n_ins_since_vacuum >
> *autovacuum_vacuum_insert_threshold* +
> *autovacuum_vacuum_insert_scale_factor* × pg_class.reltuples

Like in the previous example, you can override these values at the table level using storage parameters:

- *autovacuum_vacuum_insert_threshold* and its TOAST counterpart
- *autovacuum_vacuum_insert_scale_factor* and its TOAST counterpart

Which Tables Need to Be Analyzed?

Automatic analysis needs to process only modified rows, so the calculations are a bit simpler than those for autovacuum.

It is assumed that a table has to be analyzed if the number of rows modified since the previous analysis exceeds the threshold defined by the following two configuration parameters:

- *autovacuum_analyze_threshold* 50
- *autovacuum_analyze_scale_factor* 0.1

Autoanalysis is triggered if the following condition is met:

> pg_stat_all_tables.n_mod_since_analyze >
> *autovacuum_analyze_threshold* +
> *autovacuum_analyze_scale_factor* × pg_class.reltuples

To override autoanalysis settings for particular tables, you can use the same-name storage parameters:

- *autovacuum_analyze_threshold*
- *autovacuum_analyze_scale_factor*

Since TOAST tables are not analyzed, they have no corresponding parameters.

Autovacuum in Action

To formalize everything said in this section, let's create two views that show which tables currently need to be vacuumed and analyzed[1]. The function used in these views returns the current value of the passed parameter, taking into account that this value can be redefined at the table level:

```
=> CREATE FUNCTION p(param text, c pg_class) RETURNS float
AS $$
  SELECT coalesce(
    -- use storage parameter if set
    (SELECT option_value
     FROM   pg_options_to_table(c.reloptions)
     WHERE  option_name = CASE
              -- for TOAST tables the parameter name is different
              WHEN c.relkind = 't' THEN 'toast.' ELSE ''
            END || param
    ),
    -- else take the configuration parameter value
    current_setting(param)
  )::float;
$$ LANGUAGE sql;
```

This is how a vacuum-related view can look like:

```
=> CREATE VIEW need_vacuum AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) reltuples,
    p('autovacuum_vacuum_threshold', c) threshold,
    p('autovacuum_vacuum_scale_factor', c) scale_factor,
    p('autovacuum_vacuum_insert_threshold', c) ins_threshold,
    p('autovacuum_vacuum_insert_scale_factor', c) ins_scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m','t')
)
SELECT st.schemaname || '.' || st.relname AS tablename,
  st.n_dead_tup AS dead_tup,
  c.threshold + c.scale_factor * c.reltuples AS max_dead_tup,
  st.n_ins_since_vacuum AS ins_tup,
  c.ins_threshold + c.ins_scale_factor * c.reltuples AS max_ins_tup,
  st.last_autovacuum
FROM pg_stat_all_tables st
  JOIN c ON c.oid = st.relid;
```

[1] backend/postmaster/autovacuum.c, relation_needs_vacanalyze function

The max_dead_tup column shows the number of dead tuples that will trigger autovacuum, whereas the max_ins_tup column shows the threshold value related to insertion.

Here is a similar view for analysis:

```
=> CREATE VIEW need_analyze AS
WITH c AS (
  SELECT c.oid,
    greatest(c.reltuples, 0) reltuples,
    p('autovacuum_analyze_threshold', c) threshold,
    p('autovacuum_analyze_scale_factor', c) scale_factor
  FROM pg_class c
  WHERE c.relkind IN ('r','m')
)
SELECT st.schemaname || '.' || st.relname AS tablename,
  st.n_mod_since_analyze AS mod_tup,
  c.threshold + c.scale_factor * c.reltuples AS max_mod_tup,
  st.last_autoanalyze
FROM pg_stat_all_tables st
  JOIN c ON c.oid = st.relid;
```

The max_mod_tup column shows the threshold value for autoanalysis.

To speed up the experiment, we will be starting autovacuum every second:

```
=> ALTER SYSTEM SET autovacuum_naptime = '1s';
=> SELECT pg_reload_conf();
```

Let's truncate the vac table and then insert 1,000 rows. Note that autovacuum is turned off at the table level.

```
=> TRUNCATE TABLE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,1000) id;
```

Here is what our vacuum-related view will show:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 50
ins_tup         | 1000
max_ins_tup     | 1000
last_autovacuum |
```

The actual threshold value is max_dead_tup = 50, although the formula listed above suggests that it should be $50 + 0.2 \times 1000 = 250$. The thing is that statistics on this table are not available yet since the INSERT command does not update it:

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
        -1
(1 row)
```

v. 14 The pg_class.reltuples value is set to -1; this special constant is used instead of zero to differentiate between a table without any statistics and a really empty table that has already been analyzed. For the purpose of calculation, the negative value is taken as zero, which gives us $50 + 0.2 \times 0 = 50$.

The value of max_ins_tup = 1000 differs from the projected value of 1,200 for the same reason.

Let's have a look at the analysis view:

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+-----------
tablename        | public.vac
mod_tup          | 1006
max_mod_tup      | 50
last_autoanalyze |
```

We have updated (inserted in this case) 1,000 rows; as a result, the threshold is exceeded: since the table size is unknown, it is currently set to 50. It means that autoanalysis will be triggered immediately when we turn it on:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
```

Once the table analysis completes, the threshold is reset to an adequate value of 150 rows.

```
=> SELECT reltuples FROM pg_class WHERE relname = 'vac';
 reltuples
-----------
      1000
(1 row)
```

```
=> SELECT * FROM need_analyze WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]----+------------------------------
tablename        | public.vac
mod_tup          | 0
max_mod_tup      | 150
last_autoanalyze | 2022-07-10 18:39:54.149651+03
```

Let's get back to autovacuum:

```
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
```

The max_dead_tup and max_ins_tup values have also been updated based on the actual table size discovered by the analysis.

Vacuuming will be started if at least one of the following conditions is met:

- More than 250 dead tuples are accumulated.
- More than 200 rows are inserted into the table. v. 13

Let's turn off autovacuum again and update 251 rows so that the threshold value is exceeded by one:

```
=> ALTER TABLE vac SET (autovacuum_enabled = off);
=> UPDATE vac SET s = 'B' WHERE id <= 251;
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+-----------
tablename       | public.vac
dead_tup        | 251
max_dead_tup    | 250
ins_tup         | 1000
max_ins_tup     | 1200
last_autovacuum |
```

Now the trigger condition is satisfied. Let's enable autovacuum; after a while, we will see that the table has been processed, and its usage statistics has been reset:

```
=> ALTER TABLE vac SET (autovacuum_enabled = on);
=> SELECT * FROM need_vacuum WHERE tablename = 'public.vac' \gx
-[ RECORD 1 ]---+------------------------------
tablename       | public.vac
dead_tup        | 0
max_dead_tup    | 250
ins_tup         | 0
max_ins_tup     | 1200
last_autovacuum | 2022-07-10 18:40:00.252794+03
```

6.6. Managing the Load

Operating at the page level, vacuuming does not block other processes; but nevertheless, it increases the system load and can have a noticeable impact on performance.

Vacuum Throttling

To control vacuuming intensity, PostgreSQL makes regular pauses in table pro-
200 cessing. After completing about *vacuum_cost_limit* units of work, the process falls
0 asleep and remains idle for the *vacuum_cost_delay* time interval.

The default zero value of *vacuum_cost_delay* means that routine vacuuming actually never sleeps, so the exact *vacuum_cost_limit* value makes no difference. It is assumed that if administrators have to resort to manual vacuuming, they are likely to expect its completion as soon as possible.

If the sleep time is set, then the process will pause each time it has spent *vac-
uum_cost_limit* units of work on page processing in the buffer cache. The cost of
1 each page read is estimated at *vacuum_cost_page_hit* units if the page is found in
2 the buffer cache, or *vacuum_cost_page_miss* units otherwise[1]. If a clean page is dirt-
20 ied by vacuum, it adds another *vacuum_cost_page_dirty* units[2].

[1] backend/storage/buffer/bufmgr.c, ReadBuffer_common function
[2] backend/storage/buffer/bufmgr.c, MarkBufferDirty function

If you keep the default value of the *vacuum_cost_limit* parameter, VACUUM can process up to 200 pages per cycle in the best-case scenario (if all the pages are cached, and no pages are dirtied by VACUUM) and only nine pages in the worst case (if all the pages are read from disk and become dirty).

Autovacuum Throttling

Throttling for autovacuum[1] is quite similar to VACUUM throttling. However, autovacuum can be run with a different intensity as it has its own set of parameters:

- *autovacuum_vacuum_cost_limit* −1
- *autovacuum_vacuum_cost_delay* 2ms

If any of these parameters is set to -1, it falls back on the corresponding parameter for regular VACUUM. Thus, the *autovacuum_vacuum_cost_limit* parameter relies on the *vacuum_cost_limit* value by default.

Prior to version 12, the default value of *autovacuum_vacuum_cost_delay* was 20 ms, and it led to very poor performance on modern hardware.

Autovacuum work units are limited to *autovacuum_vacuum_cost_limit* per cycle, and since they are shared between all the workers, the overall impact on the system remains roughly the same, regardless of their number. So if you need to speed up autovacuum, both the *autovacuum_max_workers* and *autovacuum_vacuum_cost_limit* values should be increased proportionally.

If required, you can override these settings for particular tables by setting the following storage parameters:

- *autovacuum_vacuum_cost_delay* and *toast.autovacuum_vacuum_cost_delay*
- *autovacuum_vacuum_cost_limit* and *toast.autovacuum_vacuum_cost_limit*

[1] backend/postmaster/autovacuum.c, autovac_balance_cost function

6.7. Monitoring

If vacuuming is monitored, you can detect situations when dead tuples cannot be removed in one go, as references to them do not fit the *maintenance_work_mem* memory chunk. In this case, all the indexes will have to be fully scanned several times. It can take a substantial amount of time for large tables, thus creating a significant load on the system. Even though queries will not be blocked, extra I/O operations can seriously limit system throughput.

Such issues can be corrected either by vacuuming the table more often (so that each run cleans up fewer tuples) or by allocating more memory.

Monitoring Vacuum

When run with the VERBOSE clause, the VACUUM command performs the cleanup and displays the status report, whereas the pg_stat_progress_vacuum view shows the current state of the started process. v. 9.6

v. 13 There is also a similar view for analysis (pg_stat_progress_analyze), even though it is usually performed very fast and is unlikely to cause any issues.

Let's insert more rows into the table and update them all so that VACUUM has to run for a noticeable period of time:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
  SELECT id, 'A' FROM generate_series(1,500000) id;
=> UPDATE vac SET s  = 'B';
```

For the purpose of this demonstration, we will limit the amount of memory allocated for the tid array by 1 MB:

```
=> ALTER SYSTEM SET maintenance_work_mem = '1MB';
=> SELECT pg_reload_conf();
```

Launch the VACUUM command and query the pg_stat_progress_vacuum view several times while it is running:

```
=> VACUUM VERBOSE vac;
```

```
    => SELECT * FROM pg_stat_progress_vacuum \gx
    -[ RECORD 1 ]------+------------------
    pid                | 14503
    datid              | 16391
    datname            | internals
    relid              | 16479
    phase              | vacuuming indexes
    heap_blks_total    | 17242
    heap_blks_scanned  | 3009
    heap_blks_vacuumed | 0
    index_vacuum_count | 0
    max_dead_tuples    | 174761
    num_dead_tuples    | 174522
    => SELECT * FROM pg_stat_progress_vacuum \gx
    -[ RECORD 1 ]------+------------------
    pid                | 14503
    datid              | 16391
    datname            | internals
    relid              | 16479
    phase              | vacuuming indexes
    heap_blks_total    | 17242
    heap_blks_scanned  | 17242
    heap_blks_vacuumed | 6017
    index_vacuum_count | 2
    max_dead_tuples    | 174761
    num_dead_tuples    | 150956
```

In particular, this view shows:

- phase—the name of the current vacuum phase (I have described the main ones, but there are actually more of them[1])
- heap_blks_total—the total number of pages in a table
- heap_blks_scanned—the number of scanned pages
- heap_blks_vacuumed—the number of vacuumed pages
- index_vacuum_count—the number of index scans

[1] postgresql.org/docs/14/progress-reporting.html#VACUUM-PHASES

The overall vacuuming progress is defined by the ratio of heap_blks_vacuumed to heap_blks_total, but you have to keep in mind that it changes in spurts because of index scans. In fact, it is more important to pay attention to the number of vacuum cycles: if this value is greater than one, it means that the allocated memory was not enough to complete vacuuming in one go.

You can see the whole picture in the output of the VACUUM VERBOSE command, which has already finished by this time:

```
INFO:  vacuuming "public.vac"
INFO:  scanned index "vac_s" to remove 174522 row versions      } index
DETAIL:  CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.06 s     } vacuum
INFO:  table "vac": removed 174522 dead item identifiers in     }
3009 pages                                                      } table
DETAIL:  CPU: user: 0.01 s, system: 0.00 s, elapsed: 0.06 s     } vacuum
INFO:  scanned index "vac_s" to remove 174522 row versions      } index
DETAIL:  CPU: user: 0.03 s, system: 0.00 s, elapsed: 0.06 s     } vacuum
INFO:  table "vac": removed 174522 dead item identifiers in     }
3009 pages                                                      } table
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.01 s     } vacuum
INFO:  scanned index "vac_s" to remove 150956 row versions      } index
DETAIL:  CPU: user: 0.02 s, system: 0.00 s, elapsed: 0.05 s     } vacuum
INFO:  table "vac": removed 150956 dead item identifiers in     }
2603 pages                                                      } table
DETAIL:  CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s     } vacuum
INFO:  index "vac_s" now contains 500000 row versions in
932 pages
DETAIL:  500000 index row versions were removed.
433 index pages were newly deleted.
433 index pages are currently deleted, of which 0 are
currently reusable.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
INFO:  table "vac": found 500000 removable, 500000
nonremovable row versions in 17242 out of 17242 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest
xmin: 851
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.21 s, system: 0.01 s, elapsed: 0.49 s.
VACUUM
```

All in all, there have been three index scans; each scan has removed 174,522 pointers to dead tuples at the most. This value is defined by the number of tid pointers (each of them takes 6 bytes) that can fit into an array of the *maintenance_work_mem* size. The maximum size possible is shown by pg_stat_progress_vacuum.max_dead_tuples, but the actually used space is always a bit smaller. It guarantees that when the next page is read, all its pointers to dead tuples, no matter how many of them are located in this page, will fit into the remaining memory.

Monitoring Autovacuum

The main approach to monitoring autovacuum is to print its status information (which is similar to the output of the VACUUM VERBOSE command) into the server log for further analysis. If the *log_autovacuum_min_duration* parameter is set to zero, all autovacuum runs are logged: -1

```
=> ALTER SYSTEM SET log_autovacuum_min_duration = 0;

=> SELECT pg_reload_conf();

=> UPDATE vac SET s = 'C';
UPDATE 500000

postgres$ tail -n 13 /home/postgres/logfile
2022-07-10 18:40:19.165 MSK [17329] LOG:  automatic vacuum of table
"internals.public.vac": index scans: 3
pages: 0 removed, 17242 remain, 0 skipped due to pins, 0
skipped frozen
tuples: 500000 removed, 500000 remain, 0 are dead but not
yet removable, oldest xmin: 853
index scan needed: 8622 pages from table (50.01% of total)
had 500000 dead item identifiers removed
index "vac_s": pages: 1428 in total, 496 newly deleted, 929
currently deleted, 433 reusable
avg read rate: 12.169 MB/s, avg write rate: 16.846 MB/s
buffer usage: 46041 hits, 5667 misses, 7845 dirtied
WAL usage: 41028 records, 14964 full page images, 93292501
bytes
system usage: CPU: user: 0.37 s, system: 0.25 s, elapsed:
3.63 s
2022-07-10 18:40:19.520 MSK [17329] LOG:  automatic analyze of table
"internals.public.vac"
avg read rate: 45.188 MB/s, avg write rate: 0.022 MB/s
buffer usage: 15354 hits, 2036 misses, 1 dirtied
system usage: CPU: user: 0.09 s, system: 0.02 s, elapsed:
0.35 s
```

To track the list of tables that have to be vacuumed and analyzed, you can use the need_vacuum and need_analyze views, which we have already reviewed. If this list grows, it means that autovacuum does not cope with the load and has to be sped up by either reducing the gap (*autovacuum_vacuum_cost_delay*) or increasing the amount of work done between the gaps (*autovacuum_vacuum_cost_limit*). It is not unlikely that the degree of parallelism will also have to be increased (*autovacuum_max_workers*).

7

Freezing

7.1. Transaction ID Wraparound

In PostgreSQL, a transaction ID takes 32 bits. Four billions seems to be quite a big number, but it can be exhausted very fast if the system is being actively used. For example, for an average load of 1,000 transactions per second (excluding virtual ones), it will happen in about six weeks of continuous operation.

Once all the numbers are used up, the counter has to be reset to start the next round (this situation is called a "wraparound"). But a transaction with a smaller ID can only be considered older than another transaction with a bigger ID if the assigned numbers are always increasing. So the counter cannot simply start using the same numbers anew after being reset.

Allocating 64 bits for transaction IDs would have eliminated this problem altogether, so why doesn't PostgreSQL take advantage of it? The thing is that each tuple header has to store IDs for two transactions: xmin and xmax. The header is quite big already (at least 24 bytes if data alignment is taken into account), and adding more bits would have given another 8 bytes. *p. 64*

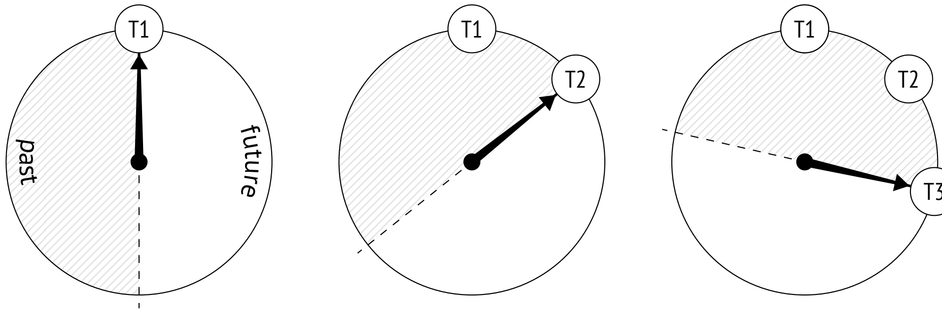
PostgreSQL does implement 64-bit transaction IDs that extend a regular ID[1] by a 32-bit epoch, but they are used only internally and never get into data pages.

To correctly handle wraparound, PostgreSQL has to compare the age of transactions (defined as the number of subsequent transactions that have appeared since the start of this transaction) rather than transaction IDs. Thus, instead of the terms *less than* and *greater than* we should use the concepts of *older* (precedes) and *younger* (follows).

[1] include/access/transam.h, FullTransactionId type

In the code, this comparison is implemented by simply using the 32-bit arithmetic: first the difference between 32-bit transaction IDs is found, and then this result is compared to zero[1].

To better visualize this idea, you can imagine a sequence of transaction IDs as a clock face. For each transaction, half of the circle in the clockwise direction will be in the future, while the other half will be in the past.



However, this visualization has an unpleasant catch. An old transaction (T1) is in the remote past as compared to more recent transactions. But sooner or later a new transaction will see it in the half of the circle related to the future. If it were really so, it would have a catastrophic impact: from now on, all newer transactions would not see the changes made by transaction T1.

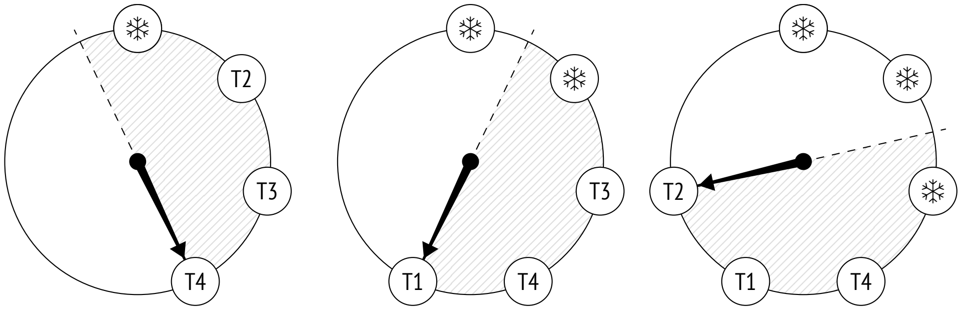
7.2. Tuple Freezing and Visibility Rules

To prevent such "time travel," vacuuming performs one more task (in addition to page cleanup)[2]: it searches for tuples that are beyond the database horizon (so they are visible in all snapshots) and tags them in a special way, that is, *freezes* them.

For frozen tuples, visibility rules do not have to take xmin into account since such tuples are known to be visible in all snapshots, so this transaction ID can be safely reused.

[1] backend/access/transam/transam.c, TransactionIdPrecedes function
[2] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-WRAPAROUND

You can imagine that the xmin transaction ID is replaced in frozen tuples by a hypothetical “minus infinity” (pictured as a snowflake below); it is a sign that this tuple is created by a transaction that is so far in the past that its actual ID does not matter anymore. Yet in reality xmin remains unchanged, whereas the freezing attribute is defined by a combination of two hint bits: committed and aborted.



Many sources (including the documentation) mention FrozenTransactionId = 2. It is the “minus infinity” that I have already referred to—this value used to replace xmin in versions prior to 9.4, but now hint bits are employed instead. As a result, the original transaction ID remains in the tuple, which is convenient for both debugging and support. Old systems can still contain the obsolete FrozenTransactionId, even if they have been upgraded to higher versions.

The xmax transaction ID does not participate in freezing in any way. It is only present in outdated tuples, and once such tuples stop being visible in all snapshots (which means that the xmax ID is beyond the database horizon), they will be vacuumed away.

Let’s create a new table for our experiments. The *fillfactor* parameter should be set to the lowest value so that each page can accommodate only two tuples—it will be easier to track the progress this way. We will also disable autovacuum to make sure that the table is only cleaned up on demand.

```
=> CREATE TABLE tfreeze(
  id integer,
  s char(300)
)
WITH (fillfactor = 10, autovacuum_enabled = off);
```

We are going to create yet another flavor of the function that displays heap pages using pageinspect. Dealing with a range of pages, it will show the values of the freezing attribute (f) and the xmin transaction age for each tuple (it will have to call the age system function—the age itself is not stored in heap pages, of course):

```
=> CREATE FUNCTION heap_page(
  relname text, pageno_from integer, pageno_to integer
)
RETURNS TABLE(
  ctid tid, state text,
  xmin text, xmin_age integer, xmax text
) AS $$
SELECT (pageno,lp)::text::tid AS ctid,
       CASE lp_flags
         WHEN 0 THEN 'unused'
         WHEN 1 THEN 'normal'
         WHEN 2 THEN 'redirect to '||lp_off
         WHEN 3 THEN 'dead'
       END AS state,
       t_xmin || CASE
         WHEN (t_infomask & 256+512) = 256+512 THEN ' f'
         WHEN (t_infomask & 256) > 0 THEN ' c'
         WHEN (t_infomask & 512) > 0 THEN ' a'
         ELSE ''
       END AS xmin,
       age(t_xmin) AS xmin_age,
       t_xmax || CASE
         WHEN (t_infomask & 1024) > 0 THEN ' c'
         WHEN (t_infomask & 2048) > 0 THEN ' a'
         ELSE ''
       END AS xmax
FROM generate_series(pageno_from, pageno_to) p(pageno),
     heap_page_items(get_raw_page(relname, pageno))
ORDER BY pageno, lp;
$$ LANGUAGE sql;
```

Now let's insert some rows into the table and run the VACUUM command that will immediately create the visibility map.

```
=> INSERT INTO tfreeze(id, s)
  SELECT id, 'FOO'||id FROM generate_series(1,100) id;
INSERT 0 100
```

```
=> VACUUM tfreeze;
```

We are going to observe the first two heap pages using the pg_visibility extension. When vacuuming completes, both pages get tagged in the visibility map (all_visible) but not in the freeze map (all_frozen), as they still contain some unfrozen tuples: v. 9.6

```
=> CREATE EXTENSION pg_visibility;
=> SELECT *
FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 rows)
```

The xmin_age of the transaction that has created the rows equals 1 because it is the latest transaction performed in the system:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 856 c |        1 | 0 a
 (0,2) | normal | 856 c |        1 | 0 a
 (1,1) | normal | 856 c |        1 | 0 a
 (1,2) | normal | 856 c |        1 | 0 a
(4 rows)
```

7.3. Managing Freezing

There are four main parameters that control freezing. All of them represent transaction age and define when the following events happen:

- Freezing starts (*vacuum_freeze_min_age*).
- Aggressive freezing is performed (*vacuum_freeze_table_age*).
- Freezing is forced (*autovacuum_freeze_max_age*).
- Freezing receives priority (*vacuum_failsafe_age*). v. 14

Minimal Freezing Age

50 million The *vacuum_freeze_min_age* parameter defines the minimal freezing age of xmin transactions. The lower its value, the higher the overhead: if a row is “hot” and is actively being changed, then freezing all its newly created versions will be a wasted effort. Setting this parameter to a relatively high value allows you to wait for a while.

To observe the freezing process, let’s reduce this parameter value to one:

```
=> ALTER SYSTEM SET vacuum_freeze_min_age = 1;
=> SELECT pg_reload_conf();
```

Now update one row in the zero page. The new row version will get into the same page because the *fillfactor* value is quite small:

```
=> UPDATE tfreeze SET s = 'BAR' WHERE id = 1;
```

The age of all transactions has been increased by one, and the heap pages now look as follows:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 856 c |        2 | 857
 (0,2) | normal | 856 c |        2 | 0 a
 (0,3) | normal | 857   |        1 | 0 a
 (1,1) | normal | 856 c |        2 | 0 a
 (1,2) | normal | 856 c |        2 | 0 a
(5 rows)
```

At this point, the tuples that are older than *vacuum_freeze_min_age* = 1 are subject to freezing. But vacuum will not process any pages tagged in the visibility map: *p. 118*

```
=> SELECT * FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | f           | f
     1 | t           | f
(2 rows)
```

The previous UPDATE command has removed the visibility bit of the zero page, so the tuple that has an appropriate xmin age in this page will be frozen. But the first page will be skipped altogether:

```
=> VACUUM tfreeze;
```

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state     | xmin  | xmin_age | xmax
-------+---------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal        | 856 f |        2 | 0 a
 (0,3) | normal        | 857 c |        1 | 0 a
 (1,1) | normal        | 856 c |        2 | 0 a
 (1,2) | normal        | 856 c |        2 | 0 a
(5 rows)
```

Now the zero page appears in the visibility map again, and if nothing changes in it, vacuuming will not return to this page anymore:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
     pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | f
(2 rows)
```

Age for Aggressive Freezing

As we have just seen, if a page contains only the current tuples that are visible in all snapshots, vacuuming will not freeze them. To overcome this constraint, PostgreSQL provides the *vacuum_freeze_table_age* parameter. It defines the transaction age that allows vacuuming to ignore the visibility map, so any heap page can be frozen. 150 million

For each table, the system catalog keeps a transaction ID for which it is known that all the older transactions are sure to be frozen. It is stored as relfrozenid:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class
WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          854 |   4
(1 row)
```

It is the age of this transaction that is compared to the *vacuum_freeze_table_age* value to decide whether the time has come for aggressive freezing.

v. 9.6 Thanks to the freeze map, there is no need to perform a full table scan during vacuuming: it is enough to check only those pages that do not appear in the map. Apart from this important optimization, the freeze map also brings fault tolerance: if vacuuming is interrupted, its next run will not have to get back to the pages that have already been processed and are tagged in the map.

PostgreSQL performs aggressive freezing of all pages in a table each time when the number of transactions in the system reaches the *vacuum_freeze_table_age* − *vacuum_freeze_min_age* limit (if the default values are used, it happens after each 100 million transactions). Thus, if the *vacuum_freeze_min_age* value is too big, it can lead to excessive freezing and increased overhead.

To freeze the whole table, let's reduce the *vacuum_freeze_table_age* value to four; then the condition for aggressive freezing will be satisfied:

```
=> ALTER SYSTEM SET vacuum_freeze_table_age = 4;
```

```
=> SELECT pg_reload_conf();
```

Run the VACUUM command:

```
=> VACUUM VERBOSE tfreeze;
INFO:  aggressively vacuuming "public.tfreeze"
INFO:  table "tfreeze": found 0 removable, 100 nonremovable row
versions in 50 out of 50 pages
DETAIL:  0 dead row versions cannot be removed yet, oldest xmin: 858
Skipped 0 pages due to buffer pins, 0 frozen pages.
CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s.
VACUUM
```

Now that the whole table has been analyzed, the relfrozenid value can be advanced—heap pages are guaranteed to have no older unfrozen xmin transactions:

```
=> SELECT relfrozenxid, age(relfrozenxid)
FROM pg_class
WHERE relname = 'tfreeze';
 relfrozenxid | age
--------------+-----
          857 |   1
(1 row)
```

The first page now contains only frozen tuples:

```
=> SELECT * FROM heap_page('tfreeze',0,1);
 ctid  |     state     | xmin  | xmin_age | xmax
-------+---------------+-------+----------+------
 (0,1) | redirect to 3 |       |          |
 (0,2) | normal        | 856 f |        2 | 0 a
 (0,3) | normal        | 857 c |        1 | 0 a
 (1,1) | normal        | 856 f |        2 | 0 a
 (1,2) | normal        | 856 f |        2 | 0 a
(5 rows)
```

Besides, this page is tagged in the freeze map:

```
=> SELECT * FROM generate_series(0,1) g(blkno),
  pg_visibility_map('tfreeze',g.blkno)
ORDER BY g.blkno;
 blkno | all_visible | all_frozen
-------+-------------+------------
     0 | t           | f
     1 | t           | t
(2 rows)
```

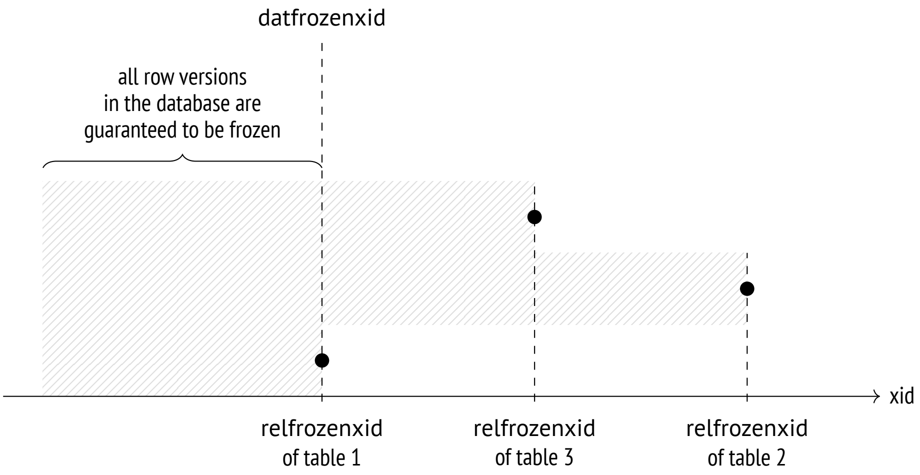
Age for Forced Autovacuum

Sometimes it is not enough to configure the two parameters discussed above to timely freeze tuples. Autovacuum might be switched off, while regular VACUUM is not being called at all (it is a very bad idea, but technically it is possible). Besides,

p. 121 some inactive databases (like template0) may not be vacuumed. PostgreSQL can handle such situations by *forcing* autovacuum in the aggressive mode.

Autovacuum is forced[1] (even if it is switched off) when there is a risk that the age of some unfrozen transaction IDs in the database will exceed the *autovacuum_freeze_max_age* value. The decision is taken based on the age of the oldest pg_class.relfrozenxid transaction in all the tables, as all the older transactions are guaranteed to be frozen. The ID of this transaction is stored in the system catalog: 200 million

```
=> SELECT datname, datfrozenxid, age(datfrozenxid) FROM pg_database;
  datname  | datfrozenxid | age
-----------+--------------+-----
 postgres  |          726 | 132
 template1 |          726 | 132
 template0 |          726 | 132
 internals |          726 | 132
(4 rows)
```



The *autovacuum_freeze_max_age* limit is set to 2 billion transactions (a bit less than half of the circle), while the default value is 10 times smaller. It is done for good reason: a big value increases the risk of transaction ID wraparound, as PostgreSQL may fail to timely freeze all the required tuples. In this case, the server must stop immediately to prevent possible issues and will have to be restarted by an administrator.

[1] backend/access/transam/varsup.c, SetTransactionIdLimit function

The *autovacuum_freeze_max_age* value also affects the size of CLOG. There is no need to keep the status of frozen transactions, and all the transactions that precede the one with the oldest datfrozenxid in the cluster are sure to be frozen. Those CLOG files that are not required anymore are removed by autovacuum[1]. *p. 73*

Changing the *autovacuum_freeze_max_age* parameter requires a server restart. However, all the freezing settings discussed above can also be adjusted at the table level via the corresponding storage parameters. Note that the names of all these parameters start with "auto":

- *autovacuum_freeze_min_age* and *toast.autovacuum_freeze_min_age*
- *autovacuum_freeze_table_age* and *toast.autovacuum_freeze_table_age*
- *autovacuum_freeze_max_age* and *toast.autovacuum_freeze_max_age*

Age for Failsafe Freezing

v. 14

If autovacuum is already struggling to prevent transaction ID wraparound and it is clearly a race against time, a safety switch is pulled: autovacuum will ignore the *autovacuum_vacuum_cost_delay* (*vacuum_cost_delay*) value and will stop vacuuming indexes to freeze heap tuples as soon as possible.

A failsafe freezing mode is enabled[2] if there is a risk that the age of an unfrozen transaction in the database will exceed the *vacuum_failsafe_age* value. It is assumed that this value must be higher than *autovacuum_freeze_max_age*. 1.6 billion

7.4. Manual Freezing

It is sometimes more convenient to manage freezing manually rather than rely on autovacuum.

[1] backend/commands/vacuum.c, vac_truncate_clog function
[2] backend/access/heap/vacuumlazy.c, lazy_check_wraparound_failsafe function

Freezing by Vacuum

You can initiate freezing by calling the VACUUM command with the FREEZE option. It will freeze all the heap tuples regardless of their transaction age, as if *vacuum_freeze_min_age* = 0.

v. 12 If the purpose of such a call is to freeze heap tuples as soon as possible, it makes sense to disable index vacuuming, like it is done in the failsafe mode. You can do it either explicitly, by running the VACUUM (freeze, index_cleanup false) command, or via the *vacuum_index_cleanup* storage parameter. It is rather obvious that it should not be done on a regular basis since in this case VACUUM will not be coping well with its main task of page cleanup.

Freezing Data at the Initial Loading

The data that is not expected to change can be frozen at once, while it is being loaded into the database. It is done by running the COPY command with the FREEZE option.

Tuples can be frozen during the initial loading only if the resulting table has been created or truncated within the same transaction, as both these operations acquire an exclusive lock on the table. This restriction is necessary because frozen tuples are expected to be visible in all snapshots, regardless of the isolation level; otherwise, transactions would suddenly see freshly-frozen tuples right as they are being uploaded. But if the lock is acquired, other transactions will not be able to get access to this table.

Nevertheless, it is still technically possible to break isolation. Let's start a new transaction at the Repeatable Read isolation level in a separate session:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
=> SELECT 1; -- the shapshot is built
```

Truncate the tfreeze table and insert new rows into this table within the same transaction. (If the reading transaction had already accessed the tfreeze table, the TRUNCATE command will be blocked.)

```
=> BEGIN;
=> TRUNCATE tfreeze;
=> COPY tfreeze FROM stdin WITH FREEZE;
1 FOO
2 BAR
3 BAZ
\.
=> COMMIT;
```

Now the reading transaction sees the new data as well:

```
    => SELECT count(*) FROM tfreeze;
     count
    -------
         3
    (1 row)
    => COMMIT;
```

It does break isolation, but since data loading is unlikely to happen regularly, in most cases it will not cause any issues.

If you load data with freezing, the visibility map is created at once, and page headers receive the visibility attribute: v. 14 *p. 114*

```
=> SELECT * FROM pg_visibility_map('tfreeze',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('tfreeze',0));
 all_visible
-------------
 t
(1 row)
```

Thus, if the data has been loaded with freezing, the table will not be processed by vacuum (as long as the data remains unchanged). Unfortunately, this functionality is not supported for TOAST tables yet: if an oversized value is loaded, vacuum will have to rewrite the whole TOAST table to set visibility attributes in all page headers. v. 14

8

Rebuilding Tables and Indexes

8.1. Full Vacuuming

Why is Routine Vacuuming not Enough?

Routine vacuuming can free more space than page pruning, but sometimes it may still be not enough.

If table or index files have grown in size, VACUUM can clean up some space within pages, but it can rarely reduce the number of pages. The reclaimed space can only be returned to the operating system if several empty pages appear at the very end of the file, which does not happen too often.

An excessive size can lead to unpleasant consequences:

- Full table (or index) scan will take longer.
- A bigger buffer cache may be required (pages are cached as a whole, so data density decreases).
- B-trees can get an extra level, which slows down index access.
- Files take up extra space on disk and in backups.

If the fraction of useful data in a file has dropped below some reasonable level, an administrator can perform *full vacuuming*[1] by running the VACUUM FULL command. In this case, the table and all its indexes are rebuilt from scratch, and the data is packed as densely as possible (taking the *fillfactor* parameter into account). *p. 100*

[1] postgresql.org/docs/14/routine-vacuuming.html#VACUUM-FOR-SPACE-RECOVERY

When full vacuuming is performed, PostgreSQL first fully rebuilds the table and then each of its indexes. While an object is being rebuilt, both old and new files have to be stored on disk[1], so this process may require a lot of free space.

You should also keep in mind that this operation fully blocks access to the table, both for reads and writes.

Estimating Data Density

For the purpose of illustration, let's insert some rows into the table:

```
=> TRUNCATE vac;
=> INSERT INTO vac(id,s)
  SELECT id, id::text FROM generate_series(1,500000) id;
```

Storage density can be estimated using the pgstattuple extension:

```
=> CREATE EXTENSION pgstattuple;
=> SELECT * FROM pgstattuple('vac') \gx
-[ RECORD 1 ]------+---------
table_len          | 70623232
tuple_count        | 500000
tuple_len          | 64500000
tuple_percent      | 91.33
dead_tuple_count   | 0
dead_tuple_len     | 0
dead_tuple_percent | 0
free_space         | 381844
free_percent       | 0.54
```

The function reads the whole table and displays statistics on space distribution in its files. The tuple_percent field shows the percentage of space taken up by useful data (heap tuples). This value is inevitably less than 100 % because of various metadata within pages, but in this example it is still quite high.

For indexes, the displayed information differs a bit, but the avg_leaf_density field has the same meaning: it shows the percentage of useful data (in B-tree leaf pages).

[1] backend/commands/cluster.c

```
=> SELECT * FROM pgstatindex('vac_s') \gx
-[ RECORD 1 ]------+----------
version            | 4
tree_level         | 3
index_size         | 114302976
root_block_no      | 2825
internal_pages     | 376
leaf_pages         | 13576
empty_pages        | 0
deleted_pages      | 0
avg_leaf_density   | 53.88
leaf_fragmentation | 10.59
```

The previously used pgstattuple functions read the table or index in full to get the precise statistics. For large objects, it can turn out to be too expensive, so the extension also provides another function called pgstattuple_approx, which skips the pages tracked in the visibility map to show approximate figures.

A faster but even less accurate method is to roughly estimate the ratio between the data volume and the file size using the system catalog[1].

Here are the current sizes of the table and its index:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

Now let's delete 90 % of all the rows:

```
=> DELETE FROM vac WHERE id % 10 != 0;
DELETE 450000
```

Routine vacuuming does not affect the file size because there are no empty pages at the end of the file:

```
=> VACUUM vac;
```

[1] wiki.postgresql.org/wiki/Show_database_bloat.

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 67 MB      | 109 MB
(1 row)
```

However, data density has dropped about 10 times:

```
=> SELECT vac.tuple_percent, vac_s.avg_leaf_density
FROM pgstattuple('vac') vac, pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
          9.13 |             6.71
(1 row)
```

The table and the index are currently located in the following files:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
          pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16514
vac_s_filepath | base/16391/16515
```

Let's check what we will get after VACUUM FULL. While the command is running, its progress can be tracked in the pg_stat_progress_cluster view (which is similar to the pg_stat_progress_vacuum view provided for VACUUM): v. 12

```
=> VACUUM FULL vac;
```

```
    => SELECT * FROM pg_stat_progress_cluster \gx
    -[ RECORD 1 ]-------+-----------------
    pid                 | 19452
    datid               | 16391
    datname             | internals
    relid               | 16479
    command             | VACUUM FULL
    phase               | rebuilding index
    cluster_index_relid | 0
    heap_tuples_scanned | 50000
    heap_tuples_written | 50000
    heap_blks_total     | 8621
    heap_blks_scanned   | 8621
    index_rebuild_count | 0
```

Expectedly, VACUUM FULL phases[1] differ from those of routine vacuuming.

Full vacuuming has replaced old files with new ones:

```
=> SELECT pg_relation_filepath('vac') AS vac_filepath,
          pg_relation_filepath('vac_s') AS vac_s_filepath \gx
-[ RECORD 1 ]--+-----------------
vac_filepath   | base/16391/16526
vac_s_filepath | base/16391/16529
```

Both index and table sizes are much smaller now:

```
=> SELECT pg_size_pretty(pg_table_size('vac')) AS table_size,
          pg_size_pretty(pg_indexes_size('vac')) AS index_size;
 table_size | index_size
------------+------------
 6904 kB    | 6504 kB
(1 row)
```

As a result, data density has increased. For the index, it is even higher than the original one: it is more efficient to create a B-tree from scratch based on the available data than to insert entries row by row into an already existing index:

```
=> SELECT vac.tuple_percent,
          vac_s.avg_leaf_density
FROM pgstattuple('vac') vac,
     pgstatindex('vac_s') vac_s;
 tuple_percent | avg_leaf_density
---------------+------------------
         91.23 |            91.08
(1 row)
```

Freezing

When the table is being rebuilt, PostgreSQL freezes its tuples because this operation costs almost nothing compared to the rest of the work:

[1] postgresql.org/docs/14/progress-reporting.html#CLUSTER-PHASES

```
=> SELECT * FROM heap_page('vac',0,0) LIMIT 5;
 ctid  | state  | xmin  | xmin_age | xmax
-------+--------+-------+----------+------
 (0,1) | normal | 861 f |        5 | 0 a
 (0,2) | normal | 861 f |        5 | 0 a
 (0,3) | normal | 861 f |        5 | 0 a
 (0,4) | normal | 861 f |        5 | 0 a
 (0,5) | normal | 861 f |        5 | 0 a
(5 rows)
```

But pages are registered neither in the visibility map nor in the freeze map, and the page header does not receive the visibility attribute (as it happens when the COPY command is executed with the FREEZE option): p. 148

```
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 f           | f
(1 row)
=> SELECT flags & 4 > 0 all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 f
(1 row)
```

The situation improves only after VACUUM is called (or autovacuum is triggered):

```
=> VACUUM vac;
=> SELECT * FROM pg_visibility_map('vac',0);
 all_visible | all_frozen
-------------+------------
 t           | t
(1 row)
=> SELECT flags & 4 > 0 AS all_visible
FROM page_header(get_raw_page('vac',0));
 all_visible
-------------
 t
(1 row)
```

It essentially means that even if all tuples in a page are beyond the database horizon, such a page will still have to be rewritten.

8.2. Other Rebuilding Methods

Alternatives to Full Vacuuming

In addition to VACUUM FULL, there are several other commands that can fully rebuild tables and indexes. All of them exclusively lock the table, all of them delete old data files and recreate them anew.

The CLUSTER command is fully analogous to VACUUM FULL, but it also reorders tuples in files based on one of the available indexes. In some cases, it can help the planner use index scans more efficiently. But you should bear in mind that clusterization is not supported: all further table updates will be breaking the physical order of tuples.

Programmatically, VACUUM FULL is simply a special instance of the CLUSTER command that does not require tuple reordering[1].

The REINDEX command rebuilds one or more indexes[2]. In fact, VACUUM FULL and CLUSTER use this command under the hood when rebuilding indexes.

The TRUNCATE [3] command deletes all table rows; it is a logical equivalent of DELETE run without the WHERE clause. But while DELETE simply marks heap tuples as deleted (so they still have to be vacuumed), TRUNCATE creates a new empty file, which is usually faster.

p. 75

Reducing Downtime during Rebuilding

VACUUM FULL is not meant to be run regularly, as it exclusively locks the table (even for queries) for the whole duration of its operation. This is usually not an option for highly available systems.

1 backend/commands/cluster.c
2 backend/commands/indexcmds.c
3 backend/commands/tablecmds.c, ExecuteTruncate function

There are several extensions (such as pg_repack[1]) that can rebuild tables and indexes with almost zero downtime. An exclusive lock is still required, but only at the beginning and at the end of this process, and only for a short time. It is achieved by a more complex implementation: all the changes made on the original table while it is being rebuilt are saved by a trigger and then applied to the new table. To complete the operation, the utility replaces one table with the other in the system catalog.

An unconventional solution is offered by the pgcompacttable[2] utility. It performs multiple fake row updates (that do not change any data) so that current row versions gradually move towards the start of the file.

Between these update series, vacuuming removes outdated tuples and truncates the file little by little. This approach takes much more time and resources, but it requires no extra space for rebuilding the table and does not lead to load spikes. Short-time exclusive locks are still acquired while the table is being truncated, but vacuuming handles them rather smoothly. *p. 120*

8.3. Preventive Measures

Read-Only Queries

One of the reasons for file bloating is executing long-running transactions that hold the database horizon alongside intensive data updates. *p. 95*

As such, long-running (read-only) transactions do not cause any issues. So a common approach is to split the load between different systems: keep fast OLTP queries on the primary server and direct all OLAP transactions to a replica. Although it makes the solution more expensive and complicated, such measures may turn out to be indispensable.

In some cases, long transactions are the result of application or driver bugs rather than a necessity. If an issue cannot be resolved in a civilized way, the administrator can resort to the following two parameters:

[1] github.com/reorg/pg_repack.
[2] github.com/dataegret/pgcompacttable.

v. 9.6 • The *old_snapshot_threshold* parameter defines the maximum lifetime of a snapshot. Once this time is up, the server has the right to remove outdated tuples; if a long-running transaction still requires them, it will get an error (“snapshot too old”).

v. 9.6 • The *idle_in_transaction_session_timeout* parameter limits the lifetime of an idle transaction. The transaction is aborted upon reaching this threshold.

Data Updates

Another reason for bloating is simultaneous modification of a large number of tuples. If all table rows get updated, the number of tuples can double, and vacuuming will not have enough time to interfere. Page pruning can reduce this problem, but not resolve it entirely.

Let’s extend the output with another column to keep track of the processed rows:

```
=> ALTER TABLE vac ADD processed boolean DEFAULT false;
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 6936 kB
(1 row)
```

Once all the rows are updated, the table gets almost two times bigger:

```
=> UPDATE vac SET processed = true;
UPDATE 50000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 14 MB
(1 row)
```

To address this situation, you can reduce the number of changes performed by a single transaction, spreading them out over time; then vacuuming can delete outdated tuples and free some space for new ones within the already existing pages. Assuming that each row update can be committed separately, we can use the following query that selects a batch of rows of the specified size as a template:

```
SELECT ID
FROM table
WHERE filtering the already processed rows
LIMIT batch size
FOR UPDATE SKIP LOCKED
```

This code snippet selects and immediately locks a set of rows that does not exceed the specified size. The rows that are already locked by other transactions are skipped: they will get into another batch next time. It is a rather flexible and convenient solution that allows you to easily change the batch size and restart the operation in case of a failure. Let's unset the processed attribute and perform full vacuuming to restore the original size of the table:

```
=> UPDATE vac SET processed = false;
=> VACUUM FULL vac;
```

Once the first batch is updated, the table size grows a bit:

```
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7064 kB
(1 row)
```

But from now on, the size remains almost the same because new tuples replace the removed ones:

```
=> VACUUM vac;
=> WITH batch AS (
  SELECT id FROM vac WHERE NOT processed LIMIT 1000
  FOR UPDATE SKIP LOCKED
)
UPDATE vac SET processed = true
WHERE id IN (SELECT id FROM batch);
UPDATE 1000
```

```
=> SELECT pg_size_pretty(pg_table_size('vac'));
 pg_size_pretty
----------------
 7072 kB
(1 row)
```

Index