Chapter 1

Database Cluster, Databases, and Tables

This chapter and the next chapter summarize the basic knowledge of PostgreSQL to help to read the subsequent chapters. This chapter describes the following topics:

- The logical structure of a database cluster
- The physical structure of a database cluster
- The internal layout of a heap table file
- The methods of writing and reading data to a table

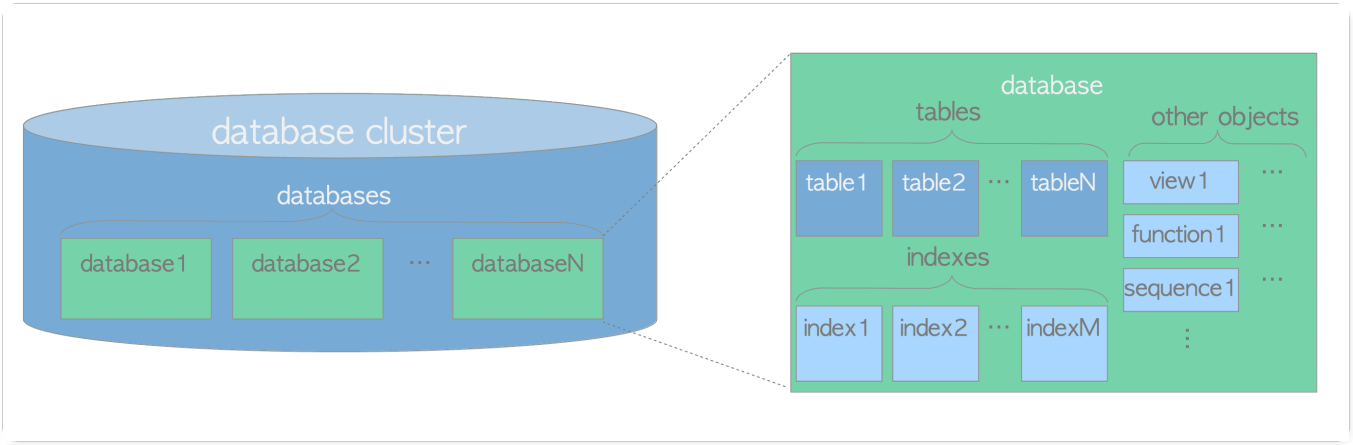
If you are already familiar with these topics, you may skip over this chapter.

1.1. Logical Structure of Database Cluster

A **database cluster** is a collection of *databases* managed by a PostgreSQL server. If you are hearing this definition for the first time, you might be wondering what it means. The term 'database cluster' in PostgreSQL does **not** mean 'a group of database servers'. A PostgreSQL server runs on a single host and manages a single database cluster.

Figure 1.1 shows the logical structure of a database cluster. A *database* is a collection of *database objects*. In the relational database theory, a *database object* is a data structure used to store or reference data. A (heap) *table* is a typical example, and there are many others, such as indexes, sequences, views, functions. In PostgreSQL, databases themselves are also database objects and are logically separated from each other. All other database objects (e.g., tables, indexes, etc) belong to their respective databases.

Fig. 1.1. Logical structure of a database cluster.



All the database objects in PostgreSQL are internally managed by respective **object identifiers (OIDs)**, which are unsigned 4-byte integers. The relations between database objects and their respective OIDs are stored in appropriate system catalogs, depending on the type of objects. For example, OIDs of databases and heap tables are stored in *pg_database* and *pg_class* respectively. You can find out the OIDs you want to know by issuing the queries such as the following:

```
sampledb=# SELECT datname, oid FROM pg_database WHERE datname = 'sampledb';
 datname  |  oid
----------+-------
 sampledb | 16384
(1 row)

sampledb=# SELECT relname, oid FROM pg_class WHERE relname = 'sampletbl';
  relname  |  oid
-----------+-------
 sampletbl | 18740
(1 row)
```

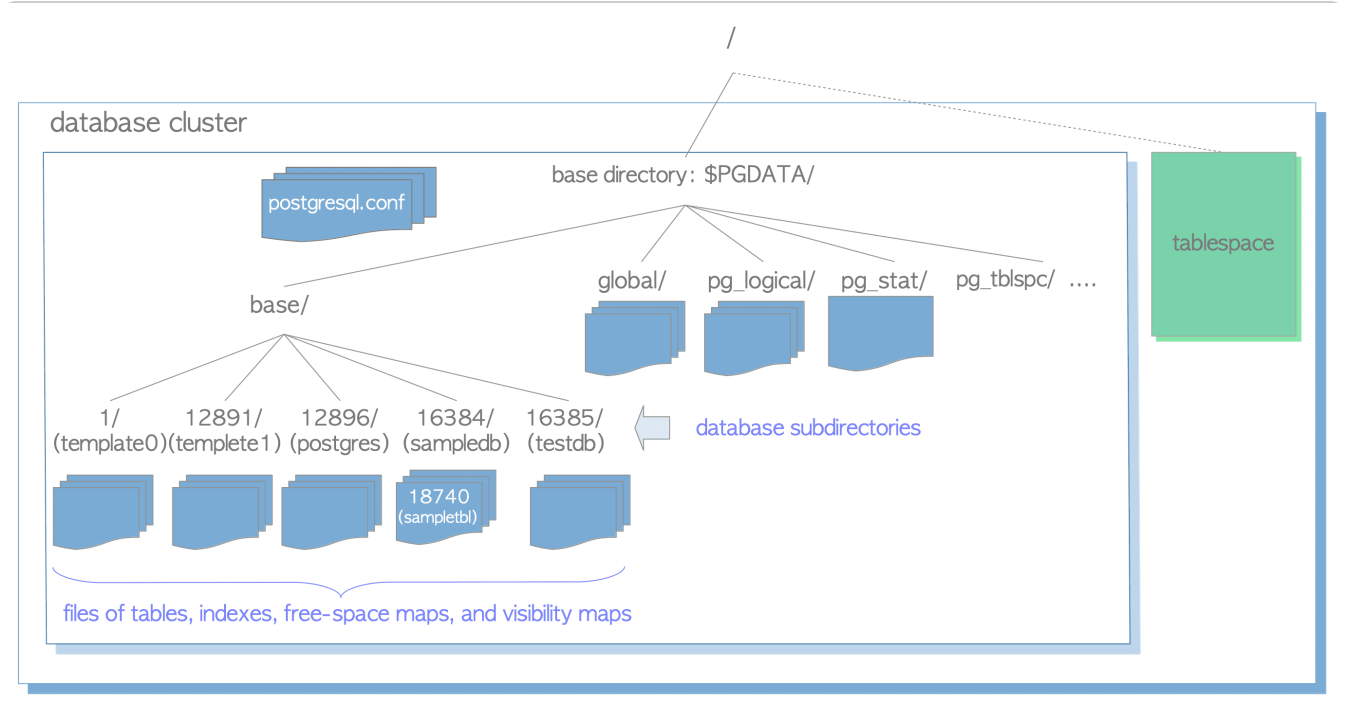
1.2. Physical Structure of Database Cluster

A *database cluster* is basically a single directory, referred to as **base directory**. It contains some subdirectories and many files. When you execute the initdb utility to initialize a new database cluster, a base directory will be created under the specified directory. The path of the base directory is usually set to the environment variable *PGDATA*.

Figure 1.2 shows an example of database cluster in PostgreSQL. A database is a subdirectory under the *base* subdirectory, and each of the tables and indexes is (at least) one file stored under the subdirectory of the database to which it belongs. There are several subdirectories containing particular data and configuration files.

While PostgreSQL supports *tablespaces*, the meaning of the term is different from other RDBMSs. A tablespace in PostgreSQL is a single directory that contains some data outside of the base directory.

Fig. 1.2. An example of database cluster.



In the following subsections, the layout of a database cluster, databases, files associated with tables and indexes, and tablespaces in PostgreSQL are described.

1.2.1. Layout of a Database Cluster

The layout of database cluster has been described in the official document. Main files and subdirectories in a part of the document have been listed in Table 1.1:

table 1.1: Layout of files and subdirectories under the base directory (From the official document)

files	description
PG_VERSION	A file containing the major version number of PostgreSQL.
pg_hba.conf	A file to control PosgreSQL's client authentication.
pg_ident.conf	A file to control PostgreSQL's user name mapping.
postgresql.conf	A file to set configuration parameters.
postgresql.auto.conf	A file used for storing configuration parameters that are set in ALTER SYSTEM. (versions 9.4 or later)
postmaster.opts	A file recording the command line options the server was last started with.
subdirectories	**description**
base/	Subdirectory containing per-database subdirectories.
global/	Subdirectory containing cluster-wide tables, such as pg_database and pg_control.
pg_commit_ts/	Subdirectory containing transaction commit timestamp data. (versions 9.5 or later)
pg_clog/ (versions 9.6 or earlier)	Subdirectory containing transaction commit state data. It is renamed to *pg_xact* in version 10. CLOG will be described in Section 5.4.
pg_dynshmem/	Subdirectory containing files used by the dynamic shared memory subsystem. (versions 9.4 or later)
pg_logical/	Subdirectory containing status data for logical decoding. (versions 9.4 or later)
pg_multixact/	Subdirectory containing multitransaction status data. (used for shared row locks)
pg_notify/	Subdirectory containing LISTEN/NOTIFY status data.
pg_repslot/	Subdirectory containing replication slot data. (versions 9.4 or later)
pg_serial/	Subdirectory containing information about committed serializable transactions. (versions 9.1 or later)
pg_snapshots/	Subdirectory containing exported snapshots. The PostgreSQL's function pg_export_snapshot creates a snapshot information file in this subdirectory. (versions 9.2 or later)
pg_stat/	Subdirectory containing permanent files for the statistics subsystem.
pg_stat_tmp/	Subdirectory containing temporary files for the statistics subsystem.
pg_subtrans/	Subdirectory containing subtransaction status data.

pg_tblspc/	Subdirectory containing symbolic links to tablespaces.
pg_twophase/	Subdirectory containing state files for prepared transactions.
pg_wal/ (versions 10 or later)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed from *pg_xlog* in Version 10.
pg_xact/ (versions 10 or later)	Subdirectory containing transaction commit state data. It is renamed from *pg_clog* in Version 10. CLOG will be described in Section 5.4.
pg_xlog/ (versions 9.6 or earlier)	Subdirectory containing WAL (Write Ahead Logging) segment files. It is renamed to *pg_wal* in Version 10.

1.2.2. Layout of Databases

A database is a subdirectory under the *base* subdirectory. The database directory names are identical to the respective OIDs. For example, when the OID of the database *sampledb* is 16384, its subdirectory name is 16384.

```
$ cd $PGDATA
$ ls -ld base/16384
drwx------  213 postgres postgres  7242  8 26 16:33 16384
```

1.2.3. Layout of Files Associated with Tables and Indexes

Each table or index whose size is less than 1GB is stored in a single file under the database directory to which it belongs. Tables and indexes are internally managed by individual OIDs, while their data files are managed by the variable, *relfilenode*. The relfilenode values of tables and indexes basically but **not** always match the respective OIDs, the details are described below.

For example, let's show the OID and relfilenode of the table *sampletbl*:

```
sampledb=# SELECT relname, oid, relfilenode FROM pg_class WHERE relname = 'sampletbl';
  relname  |  oid  | relfilenode
-----------+-------+-------------
 sampletbl | 18740 |       18740
(1 row)
```

As you can see, the oid and relfilenode values are equal in this case. You can also see that the data file path of the table *sampletbl* is *'base/16384/18740'*.

```
$ cd $PGDATA
$ ls -la base/16384/18740
-rw------- 1 postgres postgres 8192 Apr 21 10:21 base/16384/18740
```

The relfilenode values of tables and indexes can be changed by issuing certain commands, such as TRUNCATE, REINDEX, CLUSTER. For example, if we truncate the table *sampletbl*, PostgreSQL will assign a new relfilenode (18812) to the table, removes the old data file (18740), and creates a new one (18812).

```
sampledb=# TRUNCATE sampletbl;
TRUNCATE TABLE

sampledb=# SELECT relname, oid, relfilenode FROM pg_class WHERE relname = 'sampletbl';
  relname  |  oid  | relfilenode
-----------+-------+-------------
 sampletbl | 18740 |       18812
(1 row)
```

In versions 9.0 or later, the built-in function *pg_relation_filepath* is useful as this function returns the file path name of the relation with the specified OID or name.

```
sampledb=# SELECT pg_relation_filepath('sampletbl');
 pg_relation_filepath
----------------------
 base/16384/18812
(1 row)
```

When the file size of tables and indexes exceeds 1GB, PostgreSQL creates a new file named like relfilenode.1 and uses it. If the new file is filled up, PostgreSQL will create another new file named like relfilenode.2 and so on.

```
$ cd $PGDATA
$ ls -la -h base/16384/19427*
-rw------- 1 postgres postgres 1.0G  Apr  21 11:16 data/base/16384/19427
-rw------- 1 postgres postgres  45M  Apr  21 11:20 data/base/16384/19427.1
...
```

The maximum file size of tables and indexes can be changed using the configuration, option --with-segsize when building PostgreSQL.

If you Look carefully at the database subdirectories, you will find that each table has two associated files, suffixed with '_fsm' and '_vm'. Those are called the **free space map** and **visibility map**, respectively.

The free space map stores information about the free space capacity on each page within the table file, and the visibility map stores information about the visibility of each page within the table file. (More details can be found in Sections 5.3.4 and 6.2.)

Indexes only have individual free space maps and do not have visibility map.

A specific example is shown below:

```
$ cd $PGDATA
$ ls -la base/16384/18751*
-rw------- 1 postgres postgres  8192 Apr 21 10:21 base/16384/18751
-rw------- 1 postgres postgres 24576 Apr 21 10:18 base/16384/18751_fsm
-rw------- 1 postgres postgres  8192 Apr 21 10:18 base/16384/18751_vm
```

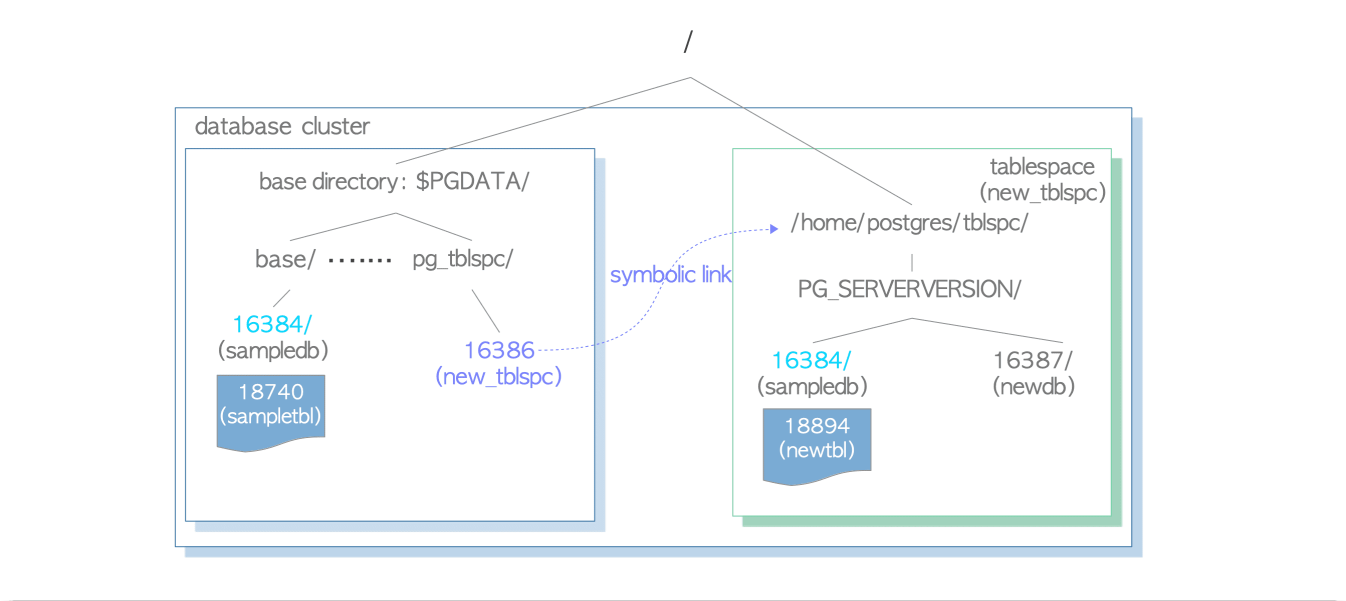
The free space map and visibility map may also be internally referred to as the **forks** of each relation. the free space map is the first fork of the table/index data file (the fork number is 1), the visibility map the second fork of the table's data file (the fork number is 2). The fork number of the data file is 0.

1.2.4. Tablespaces

A *tablespace* in PostgreSQL is an additional data area outside the base directory. This functionality was implemented in version 8.0.

Figure 1.3 shows the internal layout of a tablespace and its relationship with the main data area.

Fig. 1.3. A Tablespace in the Database Cluster.



A tablespace is created under the directory that is specified when you issue the CREATE TABLESPACE statement. Under that directory, a version-specific subdirectory (e.g., PG_14_202011044) will be created. The naming convention for the version-specific subdirectory is shown below.

```
PG _ 'Major version' _ 'Catalogue version number'
```

For example, if you create a tablespace *'new_tblspc'* at *'/home/postgres/tblspc'*, with an OID of 16386, a subdirectory named *'PG_14_202011044'* will be created under the tablespace.

```
$ ls -l /home/postgres/tblspc/
total 4
drwx------ 2 postgres postgres 4096 Apr 21 10:08 PG_14_202011044
```

The tablespace directory is addressed by a symbolic link from the *pg_tblspc* subdirectory. The link name is the same as the OID value of tablespace.

```
$ ls -l $PGDATA/pg_tblspc/
total 0
lrwxrwxrwx 1 postgres postgres 21 Apr 21 10:08 16386 -> /home/postgres/tblspc
```

If you create a new database (OID 16387) under the tablespace, its directory is created under the version-specific subdirectory.

```
$ ls -l /home/postgres/tblspc/PG_14_202011044/
total 4
drwx------ 2 postgres postgres 4096 Apr 21 10:10 16387
```

If you create a new table that belongs to the database created under the base directory, first, a new directory is created under the version-specific subdirectory. The name of the new directory is the same as the OID of the existing database. Then, the new table file is placed under the created directory.

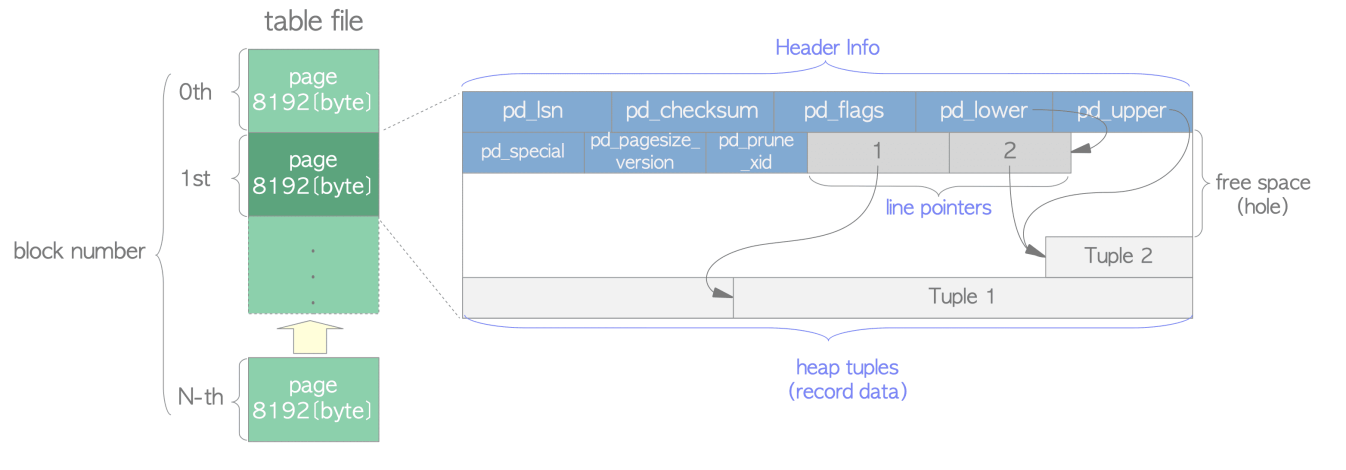
```
sampledb=# CREATE TABLE newtbl (.....) TABLESPACE new_tblspc;

sampledb=# SELECT pg_relation_filepath('newtbl');
             pg_relation_filepath
-----------------------------------------
 pg_tblspc/16386/PG_14_202011044/16384/18894
```

1.3. Internal Layout of a Heap Table File

Inside a data file (heap table, index, free space map, and visibility map), it is divided into **pages** (or **blocks**) of fixed length, which is 8192 bytes (8 KB) by default. The pages within each file are numbered sequentially from 0, and these numbers are called **block numbers**. If the file is full, PostgreSQL adds a new empty page to the end of the file to increase the file size.

The internal layout of pages depends on the data file type. In this section, the table layout is described, as this information will be required in the following chapters.

Fig. 1.4. Page layout of a heap table file.



A page within a table contains three kinds of data:

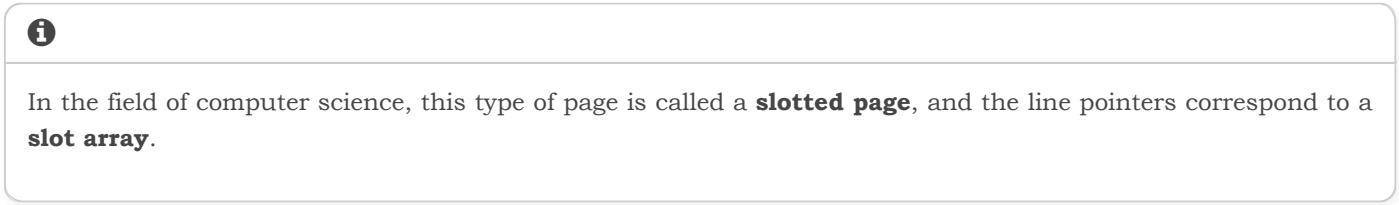
1. **heap tuple(s)** – A heap tuple is a record data itself. Heap tuples are stacked in order from the bottom of the page. The internal structure of tuple is described in Section 5.2 and Chapter 9, as it requires knowledge of both concurrency control (CC) and write-ahead logging (WAL) in PostgreSQL.
2. **line pointer(s)** – A line pointer is 4 bytes long and holds a pointer to each heap tuple. It is also called an **item pointer**.
   Line pointers form a simple array that plays the role of an index to the tuples. Each index is numbered sequentially from 1, and called **offset number**. When a new tuple is added to the page, a new line pointer is also pushed onto the array to point to the new tuple.
3. **header data** – A header data defined by the structure PageHeaderData is allocated in the beginning of the page. It is 24 byte long and contains general information about the page. The major variables of the structure are described below.
   - *pd_lsn* – This variable stores the LSN of XLOG record written by the last change of this page. It is an 8-byte unsigned integer, and is related to the WAL (Write-Ahead Logging) mechanism. The details are described in Chapter 9.
   - *pd_checksum* – This variable stores the checksum value of this page. (Note that this variable is supported in versions 9.3 or later; in earlier versions, this part had stored the timelineId of the page.)

- *pd_lower, pd_upper* – pd_lower points to the end of line pointers, and pd_upper to the beginning of the newest heap tuple.
- *pd_special* – This variable is for indexes. In the page within tables, it points to the end of the page. (In the page within indexes, it points to the beginning of special space, which is the data area held only by indexes and contains the particular data according to the kind of index types such as B-tree, GiST, GiN, etc.)

An empty space between the end of line pointers and the beginning of the newest tuple is referred to as **free space** or **hole**.

To identify a tuple within the table, a **tuple identifier (TID)** is used internally. A TID comprises a pair of values: the *block number* of the page that contains the tuple, and the *offset number* of the line pointer that points to the tuple. A typical example of its usage is index. See more detail in Section 1.4.2.

The structure PageHeaderData is defined in src/include/storage/bufpage.h.



In the field of computer science, this type of page is called a **slotted page**, and the line pointers correspond to a **slot array**.

In addition, heap tuple whose size is greater than about 2 KB (about 1/4 of 8 KB) is stored and managed using a method called **TOAST** (The Oversized-Attribute Storage Technique). Refer PostgreSQL documentation for details.

1.4. The Methods of Writing and Reading Tuples

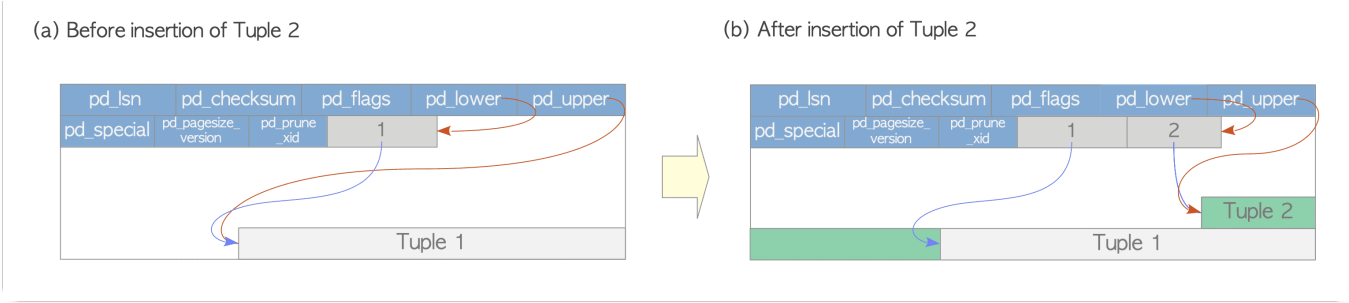
In the end of this chapter, the methods of writing and reading heap tuples are described.

1.4.1. Writing Heap Tuples

Suppose a table composed of one page that contains just one heap tuple. The pd_lower of this page points to the first line pointer, and both the line pointer and the pd_upper point to the first heap tuple. See Fig. 1.5(a).

When the second tuple is inserted, it is placed after the first one. The second line pointer is appended to the first one, and it points to the second tuple. The pd_lower changes to point to the second line pointer, and the pd_upper to the second heap tuple. See Fig. 1.5(b). Other header data within this page (e.g., pd_lsn, pg_checksum, pg_flag) are also updated to appropriate values; more details are described in Section 5.3 and Chapter 9.

Fig. 1.5. Writing of a heap tuple.

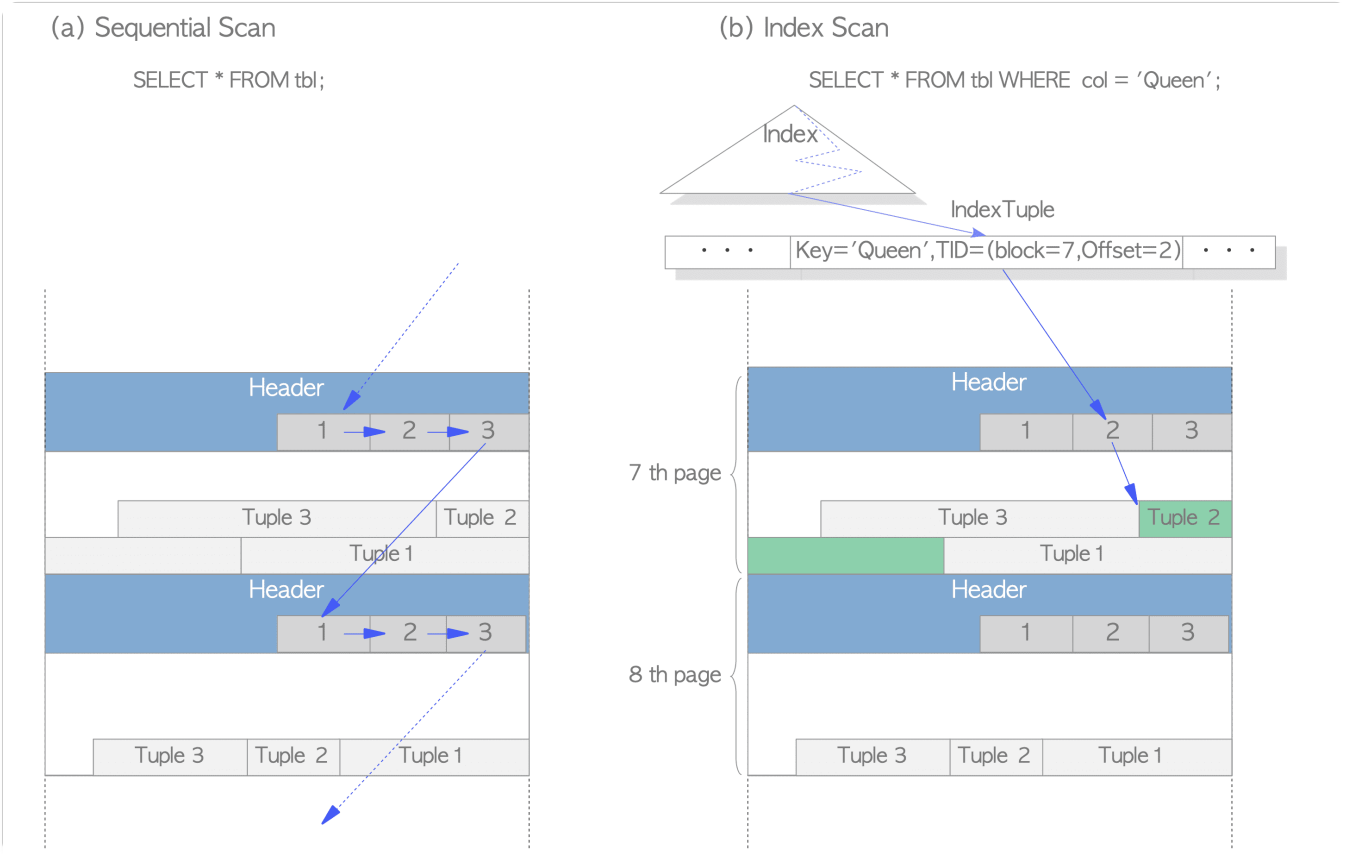


1.4.2. Reading Heap Tuples

Two typical access methods, sequential scan and B-tree index scan, are outlined here:

- **Sequential scan** – It reads all tuples in all pages sequentially by scanning all line pointers in each page. See Fig. 1.6(a).
- **B-tree index scan** – It reads an index file that contains index tuples, each of which is composed of an index key and a TID that points to the target heap tuple. If the index tuple with the key that you are looking for has been found, PostgreSQL reads the desired heap tuple using the obtained TID value. (The description of how to find the index tuples in a B-tree index is not explained here, as it is very common and the space here is limited. See the relevant materials.) For example, in Fig. 1.6(b), the TID value of the obtained index tuple is '(block = 7, Offset = 2)' This means that the target heap tuple is the 2nd tuple in the 7th page within the table, so PostgreSQL can read the desired heap tuple without unnecessary scanning in the pages.

Fig. 1.6. Sequential scan and index scan.



Indexes Internals

This document does not explain indexes in details. To understand them, I recommend to read the valuable posts shown below:

- Indexes in PostgreSQL — 1
- Indexes in PostgreSQL — 2
- Indexes in PostgreSQL — 3 (Hash)
- Indexes in PostgreSQL — 4 (Btree)
- Indexes in PostgreSQL — 5 (GiST)
- Indexes in PostgreSQL — 6 (SP-GiST)
- Indexes in PostgreSQL — 7 (GIN)
- Indexes in PostgreSQL — 9 (BRIN)

PostgreSQL also supports TID-Scan, Bitmap-Scan, and Index-Only-Scan.

TID-Scan is a method that accesses a tuple directly by using TID of the desired tuple. For example, to find the 1st tuple in the 0-th page within the table, issue the following query:

```
sampledb=# SELECT ctid, data FROM sampletbl WHERE ctid = '(0,1)';
 ctid  |   data
-------+----------
 (0,1) | AAAAAAAAA
(1 row)
```

Index-Only-Scan will be described in details in Chapter 7.