Physical Structure of Database

Case of PostgreSQL

Source: https://www.interdb.jp/pg/

Database physical structure

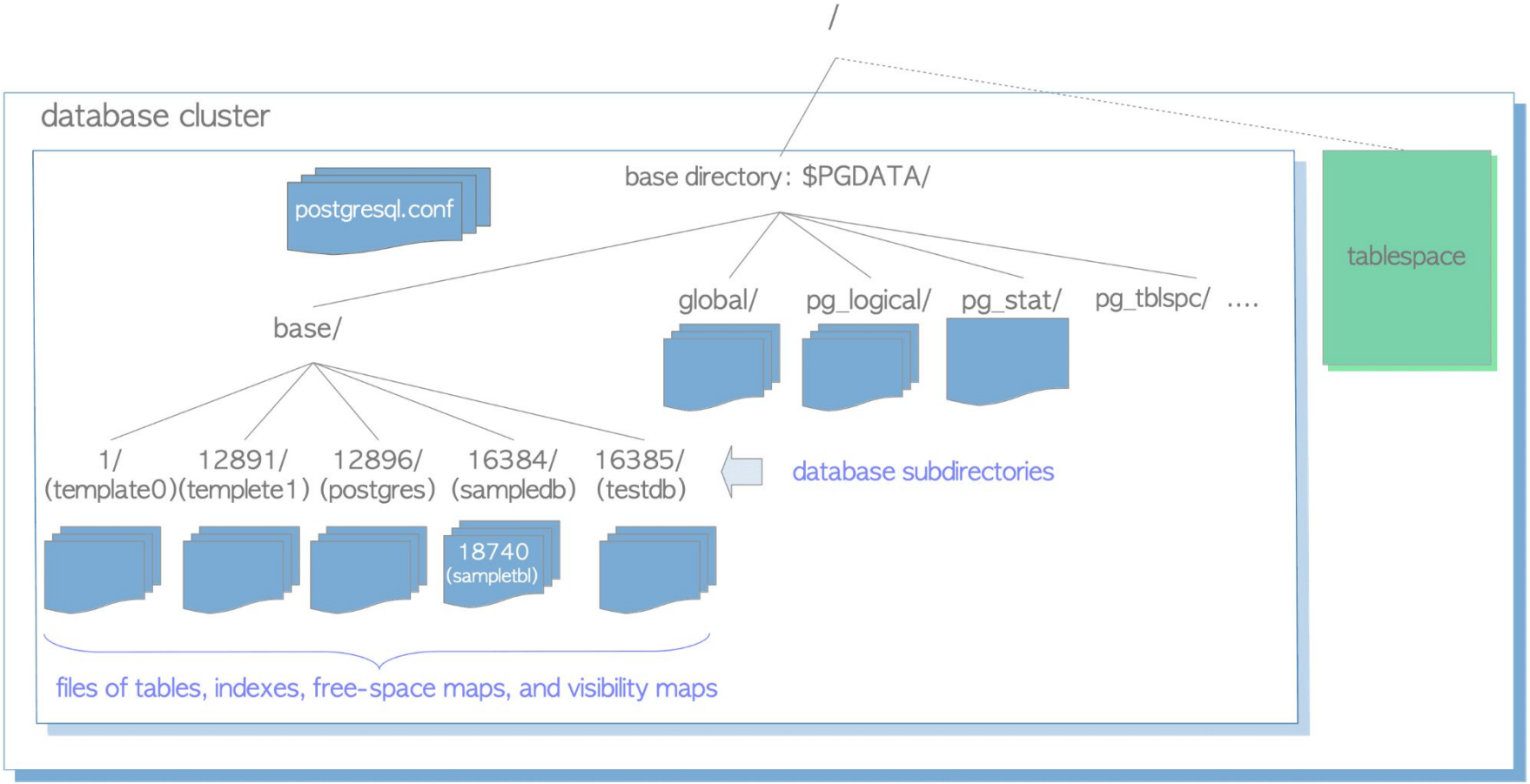
Each table or index whose size is less than 1GB is a single file stored under the database directory it belongs to. Tables and indexes as database objects are internally managed by individual OIDs, while those data files are managed by the variable, *relfilenode*.

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 has been filled up, next new file named like relfilenode.2 will be created, and so on.

each table has two associated files suffixed respectively with

'_fsm' **free space map** and Each FSM stores the information about the free space capacity of each page within the corresponding table or index file.

'_vm'**visibility map** the **Visibility Map (VM)** was introduced to improve the efficiency of removing dead tuples.


/
database cluster
base directory: $PGDATA/
postgresql.conf
tablespace
base/
global/
pg_logical/
pg_stat/
pg_tblspc/
....
1/
(template0)
12891/
(templete1)
12896/
(postgres)
16384/
(sampledb)
16385/
(testdb)
database subdirectories
18740
(sampletbl)
files of tables, indexes, free-space maps, and visibility maps

```
# pwd
/var/lib/postgresql/data
# ls
base          pg_dynshmem    pg_logical    pg_replslot   pg_stat       pg_tblspc    pg_wal                postgresql.conf
global        pg_hba.conf    pg_multixact  pg_serial     pg_stat_tmp   pg_twophase  pg_xact               postmaster.opts
pg_commit_ts  pg_ident.conf  pg_notify     pg_snapshots  pg_subtrans   PG_VERSION   postgresql.auto.conf  postmaster.pid
```

SELECT datname, oid **FROM** pg_database

```
# pwd
/var/lib/postgresql/data/base
# ls
1  13756  13757  16384  16385  pgsql_tmp
#
```

	datname name	oid [PK] oid
1	postgres	13757
2	flights	16384
3	template1	1
4	template0	13756
5	demo	16385

'Flights' is a table in the database ‘demo”

SELECT relname, oid, relfilenode **FROM** pg_class
WHERE relname = 'flights';

```
# pwd
/var/lib/postgresql/data/base/16385
# ls 16414*
16414  16414_fsm  16414_vm
```

	relname name	oid [PK] oid	relfilenode oid
1	flights	16414	16414

Internal Layout of a Heap Table File

PostgreSQL use fixed page size (commonly 8 kB), and does not allow tuples to span multiple pages.

s aA heap file is an unordered collection of pages where tuples that are stored in random order.

Inside the data file (heap table and index, as well as the free space map and visibility map), it is divided into **pages** (or **blocks**) of fixed length, the default is 8192 byte (8 KB). Those pages within each file are numbered sequentially from 0, and such numbers are called as **block numbers**. If the file has been filled up, PostgreSQL adds a new empty page to the end of the file to increase the file size.

The data are loaded per page or block by: page number (page#) and page size.

The pointer of the page, called page offset, is calculated by

Offset = Page# × PageSize

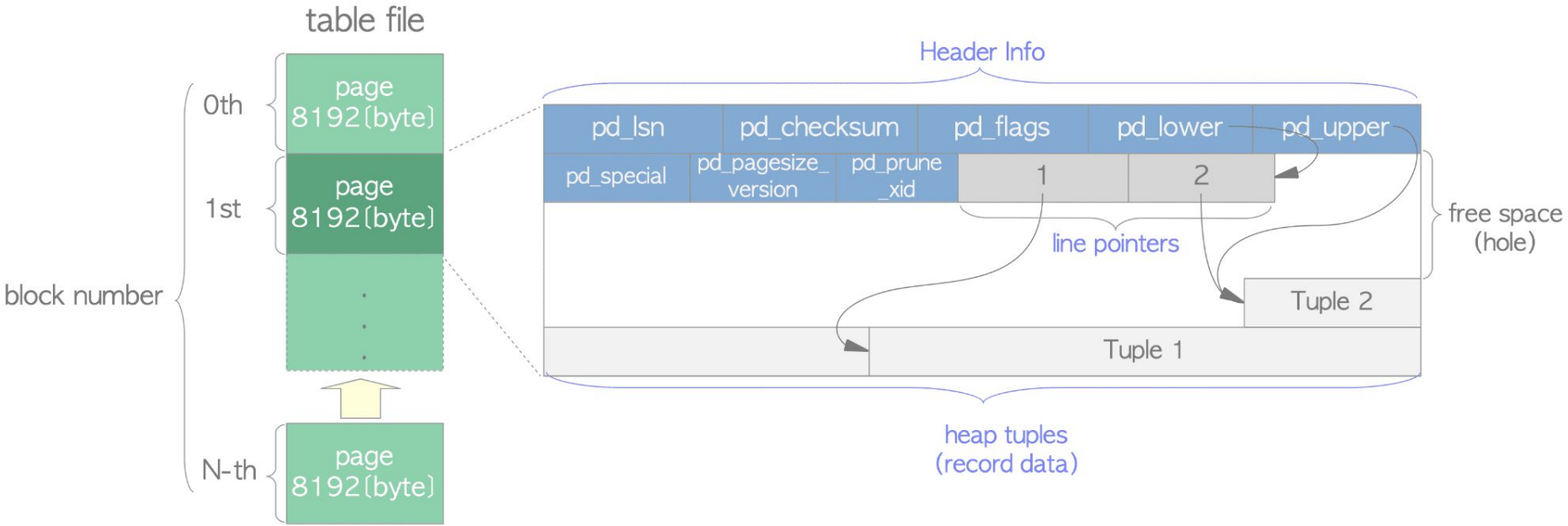


table file
block number
0th
page
8192(byte)
1st
page
8192(byte)
N-th
page
8192(byte)
Header Info
pd_lsn
pd_checksum
pd_flags
pd_lower
pd_upper
pd_special
pd_pagesize_
version
pd_prune
_xid
1
2
line pointers
free space
(hole)
Tuple 2
Tuple 1
heap tuples
(record data)

heap tuple(s) – A heap tuple is a record data itself. They are stacked in order from the bottom of the page.

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, related to the WAL (Write-Ahead Logging) mechanism.
- *pd_checksum* – This variable stores the checksum value of this page. (Note that this variable is supported in version 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, **tuple identifier (TID)** is internally used. 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.

```
typedef struct PageHeaderData @src/include/storage/bufpage.h
{
  /* XXX LSN is member of *any* block, not only page-organized ones */
  XLogRecPtr    pd_lsn;      /* LSN: next byte after last byte of xlog
                              * record for last change to this page */
  uint16        pd_checksum; /* checksum */
  uint16        pd_flags;    /* flag bits, see below */
  LocationIndex pd_lower;    /* offset to start of free space */
  LocationIndex pd_upper;    /* offset to end of free space */
  LocationIndex pd_special;  /* offset to start of special space */
  uint16        pd_pagesize_version;
  TransactionId pd_prune_xid;/* oldest prunable XID, or zero if none */
  ItemIdData    pd_linp[1];  /* beginning of line pointer array */
} PageHeaderData;

typedef PageHeaderData *PageHeader;

typedef uint64 XLogRecPtr;
```

Tuple structure

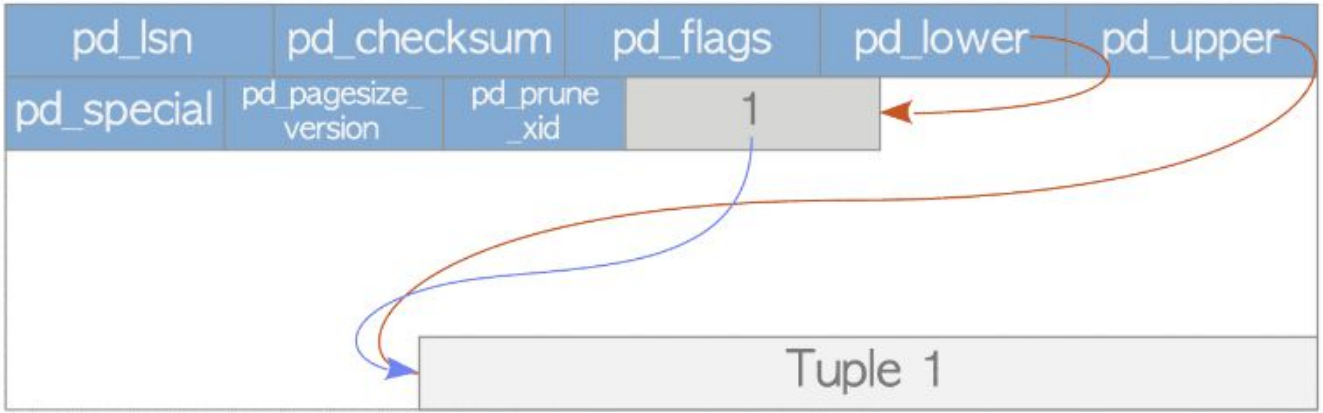
A heap tuple comprises three parts, i.e. the HeapTupleHeaderData structure, NULL bitmap, and user data

- **t_xmin** holds the txid of the transaction that inserted this tuple.
- **t_xmax** holds the txid of the transaction that deleted or updated this tuple. If this tuple has not been deleted or updated, t_xmax is set to *0*, which means INVALID.
- **t_cid** holds the command id (cid), which means how many SQL commands were executed before this command was executed within the current transaction beginning from 0. For example, assume that we execute three INSERT commands within a single transaction: 'BEGIN; INSERT; INSERT; INSERT; COMMIT;'. If the first command inserts this tuple, t_cid is set to 0. If the second command inserts this, t_cid is set to 1, and so on.
- **t_ctid** holds the tuple identifier (tid) that points to itself or a new tuple. TID, is used to identify a tuple within a table. When this tuple is updated, the t_ctid of this tuple points to the new tuple; otherwise, the t_ctid points to itself.

Writing Heap Tuples

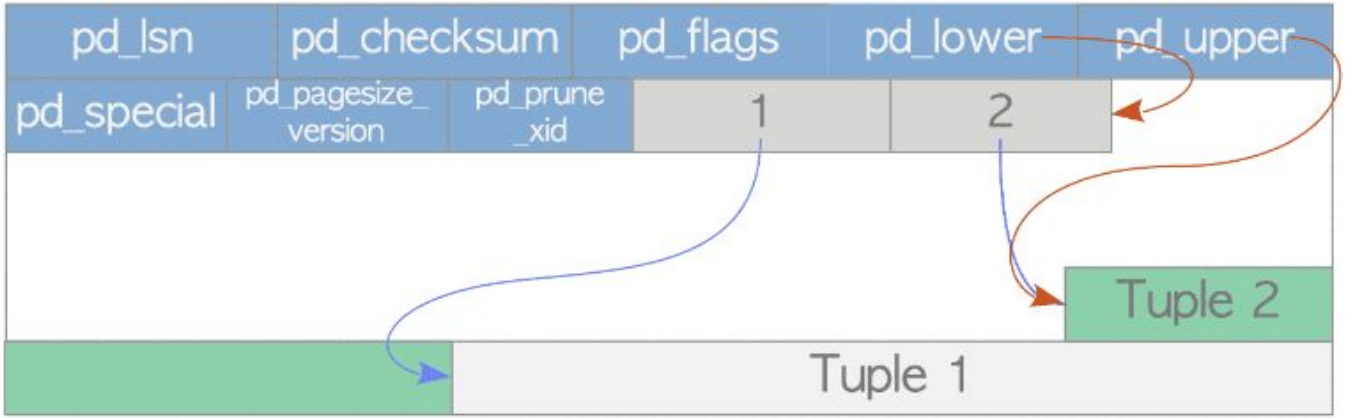
Suppose a table composed of one page which contains just one heap tuple. The pd_lower of this page points to the first line pointer, and both the line pointer and the pd_upper point to the first heap tuple.

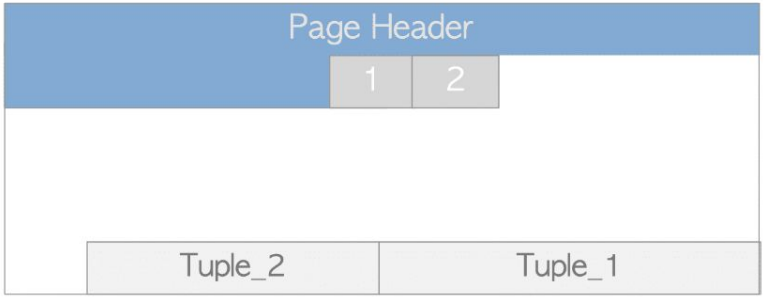
(a) Before insertion of Tuple 2



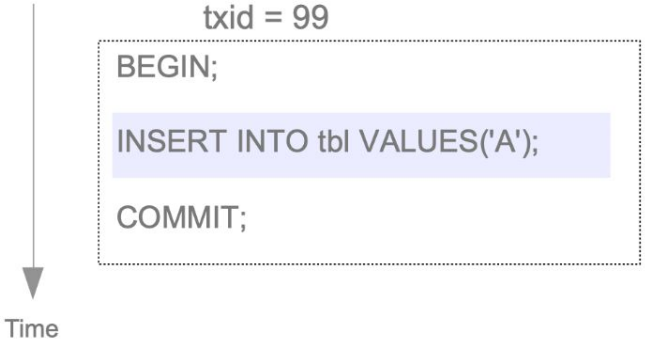
When the second tuple is inserted, it is placed after the first one. The second line pointer is pushed onto the first one, and it points to the second tuple. The pd_lower changes to point to the second line pointer, and the pd_upper to the second heap tuple. Other header data within this page (e.g., pd_lsn, pg_checksum, pg_flag) are also rewritten to appropriate values;

(b) After insertion of Tuple 2

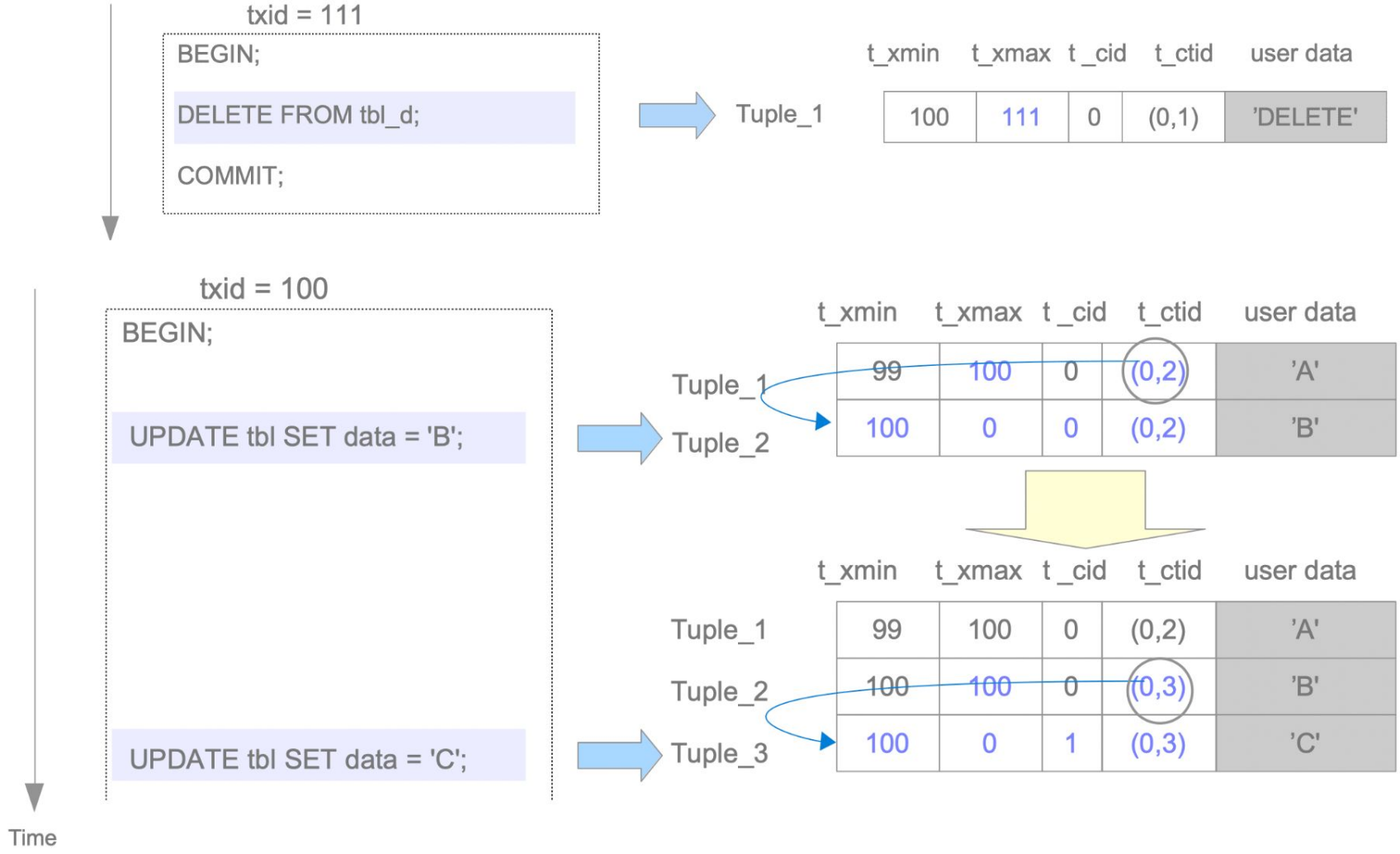




	t_xmin	t_xmax	t_cid	t_ctid	user data
Tuple_1	99	100	0	(0,2)	'A'
Tuple_2	100	0	0	(0,2)	'B'



	t_xmin	t_xmax	t_cid	t_ctid	user data
Tuple_1	99	0	0	(0,1)	'A'


txid = 111
BEGIN;
DELETE FROM tbl_d;
COMMIT;
t_xmin
t_xmax
t _cid
t_ctid
user data
Tuple_1
100
111
0
(0,1)
'DELETE'
txid = 100
BEGIN;
UPDATE tbl SET data = 'B';
UPDATE tbl SET data = 'C';
t_xmin
t_xmax
t _cid
t_ctid
user data
Tuple_1
Tuple_2
99
100
0
(0,2)
'A'
100
0
0
(0,2)
'B'
t_xmin
t_xmax
t _cid
t_ctid
user data
Tuple_1
Tuple_2
Tuple_3
99
100
0
(0,2)
'A'
100
100
0
(0,3)
'B'
100
0
1
(0,3)
'C'
Time

Reading Heap Tuples

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

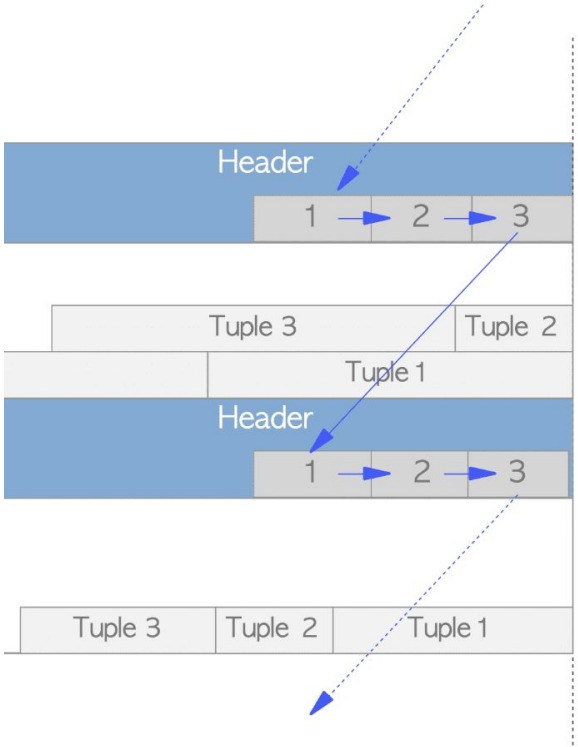
Sequential scan – All tuples in all pages are sequentially read by scanning all line pointers in each page.

B-tree index scan – An index file contains index tuples, each of which is composed of an index key and a TID pointing 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.

For example, in the Figure below, TID value of the obtained index tuple is ‘(block = 7, Offset = 2)’. It means that the target heap tuple is 2nd tuple in the 7th page within the table, so PostgreSQL can read the desired heap tuple without unnecessary scanning in the pages.

) Sequential Scan

SELECT * FROM tbl;



(b) Index Scan

SELECT * FROM tbl WHERE col = 'Queen';