Chapter 5

Concurrency Control

Concurrency Control is a mechanism that maintains atomicity and isolation, which are two properties of the ACID, when multiple transactions run concurrently in a database.

There are three broad concurrency control techniques: *Multi-version Concurrency Control* (MVCC), *Strict Two-Phase Locking* (S2PL), and *Optimistic Concurrency Control* (OCC). Each technique has many variations. In MVCC, each write operation creates a new version of a data item while retaining the old version. When a transaction reads a data item, the system selects one of the versions to ensure isolation of the individual transaction. The main advantage of MVCC is that '*readers don't block writers, and writers don't block readers*', in contrast, for example, an S2PL-based system must block readers when a writer writes an item because the writer acquires an exclusive lock for the item. PostgreSQL and some RDBMSs use a variation of MVCC called **Snapshot Isolation (SI)**.

To implement SI, some RDBMSs, such as Oracle, use rollback segments. When writing a new data item, the old version of the item is written to the rollback segment, and subsequently the new item is overwritten to the data area. PostgreSQL uses a simpler method. A new data item is inserted directly into the relevant table page. When reading items, PostgreSQL selects the appropriate version of an item in response to an individual transaction by applying **visibility check rules**.

SI does not allow the three anomalies defined in the ANSI SQL-92 standard: *Dirty Reads*, *Non-Repeatable Reads*, and *Phantom Reads*. However, SI cannot achieve true serializability because it allows serialization anomalies, such as *Write Skew* and *Read-only Transaction Skew*. Note that the ANSI SQL-92 standard based on the classical serializability definition is **not** equivalent to the definition in modern theory. To deal with this issue, **Serializable Snapshot Isolation (SSI)** has been added as of version 9.1. SSI can detect the serialization anomalies and can resolve the conflicts caused by such anomalies. Thus, PostgreSQL versions 9.1 or later provide a true SERIALIZABLE isolation level. (In addition, SQL Server also uses SSI; Oracle still uses only SI.)

This chapter comprises the following four parts:

Part 1: Sections 5.1. — 5.3.
This part provides basic information required for understanding the subsequent parts.
Sections 5.1 and 5.2 describe transaction ids and tuple structure, respectively. Section 5.3 exhibits how tuples are inserted, deleted, and updated.

Part 2: Sections 5.4. — 5.6.
This part illustrates the key features required for implementing the concurrency control mechanism.
Sections 5.4, 5.5, and 5.6 describe the commit log (clog), which holds all transaction states, transaction snapshots, and the visibility check rules, respectively.

Part 3: Sections 5.7. — 5.9.
This part describes the concurrency control in PostgreSQL using specific examples.

Section 5.7 describes the visibility check. This section also shows how the three anomalies defined in the ANSI SQL standard are prevented. Section 5.8 describes preventing *Lost Updates*, and Section 5.9 briefly describes SSI.

Part 4: Section 5.10.

This part describes several maintenance process required to permanently running the concurrency control mechanism. The maintenance processes are performed by vacuum processing, which is described in Chapter 6.

This chapter focuses on the topics that are unique to PostgreSQL, although there are many concurrency control-related topics. Note that descriptions of deadlock prevention and lock modes are omitted. (For more information, refer to the official documentation.)

Transaction Isolation Level in PostgreSQL

PostgreSQL-implemented transaction isolation levels are described in the following table:

Isolation Level	Dirty Reads	Non-repeatable Read	Phantom Read	Serialization Anomaly
READ COMMITTED	Not possible	Possible	Possible	Possible
REPEATABLE READ*1	Not possible	Not possible	Not possible in PG; See Section 5.7.2. (Possible in ANSI SQL)	Possible
SERIALIZABLE	Not possible	Not possible	Not possible	Not possible

*1 : In versions 9.0 and earlier, this level had been used as 'SERIALIZABLE' because it does not allow the three anomalies defined in the ANSI SQL-92 standard. However, with the implementation of SSI in version 9.1, this level has changed to 'REPEATABLE READ' and a true SERIALIZABLE level was introduced.

PostgreSQL uses SSI for DML (Data Manipulation Language, e.g, SELECT, UPDATE, INSERT, DELETE), and 2PL for DDL (Data Definition Language, e.g., CREATE TABLE, etc).

5.1. Transaction ID

Whenever a transaction begins, a unique identifier, referred to as a **transaction id (txid)**, is assigned by the transaction manager. PostgreSQL's txid is a 32-bit unsigned integer, approximately 4.2 billion (thousand millions). If you execute the built-in *txid_current()* function after a transaction starts, the function returns the current txid as follows:

```
testdb=# BEGIN;
BEGIN
testdb=# SELECT txid_current();
 txid_current
--------------
```

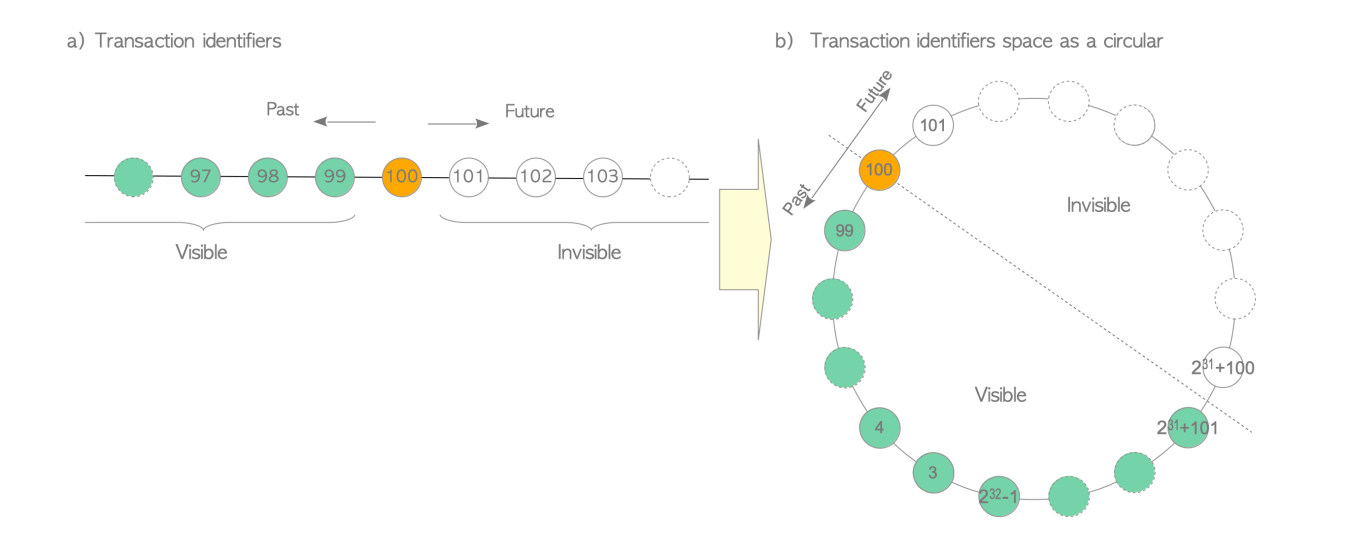
```
          100
(1 row)
```

PostgreSQL reserves the following three special txids:

0 means **Invalid** txid.
1 means **Bootstrap** txid, which is only used in the initialization of the database cluster.
2 means **Frozen** txid, which is described in Section 5.10.1.

Txids can be compared with each other. For example, at the viewpoint of txid 100, txids that are greater than 100 are '*in the future*' and are *invisible* from the txid 100; txids that are less than 100 are '*in the past*' and are *visible* (Fig. 5.1 a)).

Fig. 5.1. Transaction ids in PostgreSQL.



Since the txid space is insufficient in practical systems, PostgreSQL treats the txid space as a circle. The previous 2.1 billion txids are 'in the past', and the next 2.1 billion txids are 'in the future' (Fig. 5.1 b).

Note that the so-called *txid wraparound problem* is described in Section 5.10.1.

Note that BEGIN command does not be assigned a txid. In PostgreSQL, when the first command is executed after a BEGIN command executed, a tixd is assigned by the transaction manager, and then the transaction starts.

5.2. Tuple Structure

Heap tuples in table pages are classified into two types: usual data tuples and TOAST tuples. This section describes only the usual tuple.

A heap tuple comprises three parts: the HeapTupleHeaderData structure, NULL bitmap, and user data (Fig. 5.2).

Fig. 5.2. Tuple structure.

The HeapTupleHeaderData structure is defined in src/include/access/htup_details.h.

The HeapTupleHeaderData structure contains seven fields, but only four of them are required in the subsequent sections:

- **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 is the number of SQL commands that were executed before this command was executed within the current transaction, starting 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 tuple, 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, described in Section 1.3, 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.

```
typedef struct HeapTupleFields
{
        TransactionId t_xmin;           /* inserting xact ID */
        TransactionId t_xmax;            /* deleting or locking xact ID */

        union
        {
                CommandId       t_cid;     /* inserting or deleting command ID, or
both */
                TransactionId         t_xvac;    /* old-style VACUUM FULL xact ID
*/
        } t_field3;
} HeapTupleFields;

typedef struct DatumTupleFields
{
        int32           datum_len_;          /* varlena header (do not touch directl
y!) */
        int32           datum_typmod;        /* -1, or identifier of a record type */
        Oid             datum_typeid;        /* composite type OID, or RECORDOID */

        /*
         * Note: field ordering is chosen with thought that Oid might someday
         * widen to 64 bits.
         */
} DatumTupleFields;
```

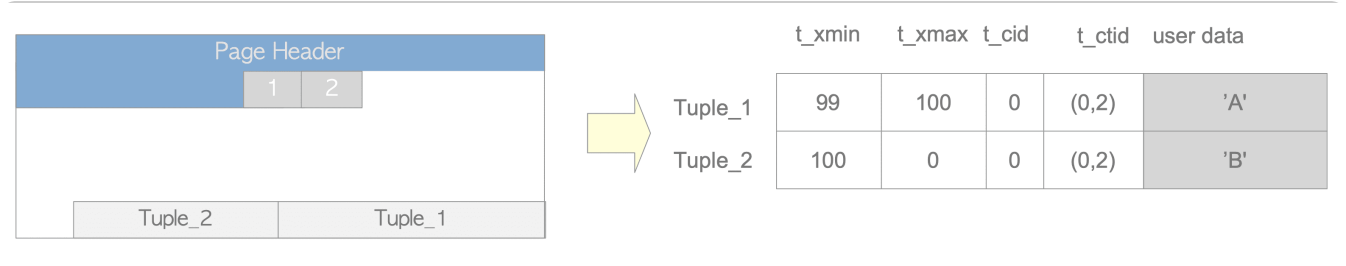
```
typedef struct HeapTupleHeaderData
{
        union
        {
                HeapTupleFields t_heap;
                DatumTupleFields t_datum;
        } t_choice;

        ItemPointerData t_ctid;         /* current TID of this or newer tuple */

        /* Fields below here must match MinimalTupleData! */
        uint16          t_infomask2;    /* number of attributes + various flags */
        uint16          t_infomask;     /* various flag bits, see below */
        uint8           t_hoff;         /* sizeof header incl. bitmap, padding */
        /* ^ - 23 bytes - ^ */
        bits8           t_bits[1];      /* bitmap of NULLs -- VARIABLE LENGTH */

        /* MORE DATA FOLLOWS AT END OF STRUCT */
} HeapTupleHeaderData;

typedef HeapTupleHeaderData *HeapTupleHeader;
```

5.3. Inserting, Deleting and Updating Tuples

This section describes how tuples are inserted, deleted, and updated. Then, the *Free Space Map (FSM)*, which is used to insert and update tuples, is briefly described.

To focus on tuples, page headers and line pointers are not represented in the following. Figure 5.3 shows an example of how tuples are represented.

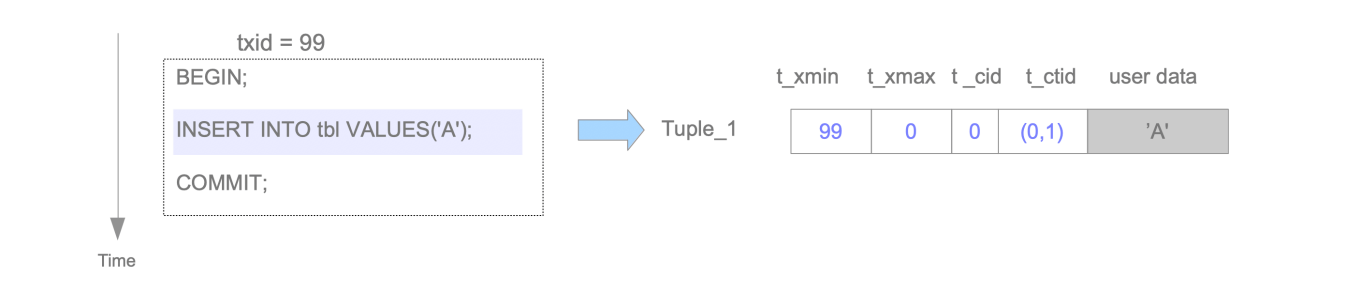
Fig. 5.3. Representation of tuples.



5.3.1. Insertion

With the insertion operation, a new tuple is inserted directly into a page of the target table (Fig. 5.4).

Fig. 5.4. Tuple insertion.



Suppose that a tuple is inserted in a page by a transaction whose txid is 99. In this case, the header fields of the inserted tuple are set as follows.

Tuple_1:

t_xmin is set to 99 because this tuple is inserted by txid 99.
t_xmax is set to 0 because this tuple has not been deleted or updated.
t_cid is set to 0 because this tuple is the first tuple inserted by txid 99.
t_ctid is set to (0,1), which points to itself, because this is the latest tuple.

pageinspect

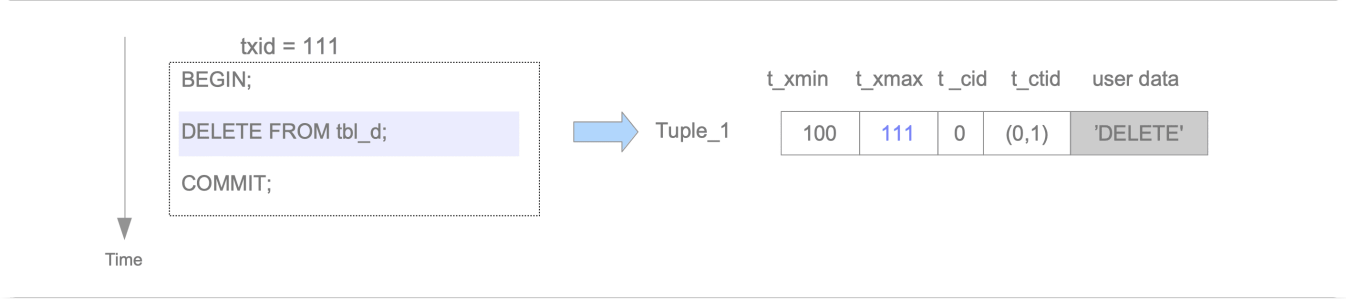
PostgreSQL provides an extension *pageinspect*, which is a contribution module, to show the contents of the database pages.

```
testdb=# CREATE EXTENSION pageinspect;
CREATE EXTENSION
testdb=# CREATE TABLE tbl (data text);
CREATE TABLE
testdb=# INSERT INTO tbl VALUES('A');
INSERT 0 1
testdb=# SELECT lp as tuple, t_xmin, t_xmax, t_field3 as t_cid, t_ctid
               FROM heap_page_items(get_raw_page('tbl', 0));
 tuple | t_xmin | t_xmax | t_cid | t_ctid
-------+--------+--------+-------+--------
     1 |     99 |      0 |     0 | (0,1)
(1 row)
```

5.3.2. Deletion

In the deletion operation, the target tuple is deleted logically. The value of the txid that executes the DELETE command is set to the t_xmax of the tuple (Fig. 5.5).

Fig. 5.5. Tuple deletion.



Suppose that tuple Tuple_1 is deleted by txid 111. In this case, the header fields of Tuple_1 are set as follows:

Tuple_1:

t_xmax is set to 111.

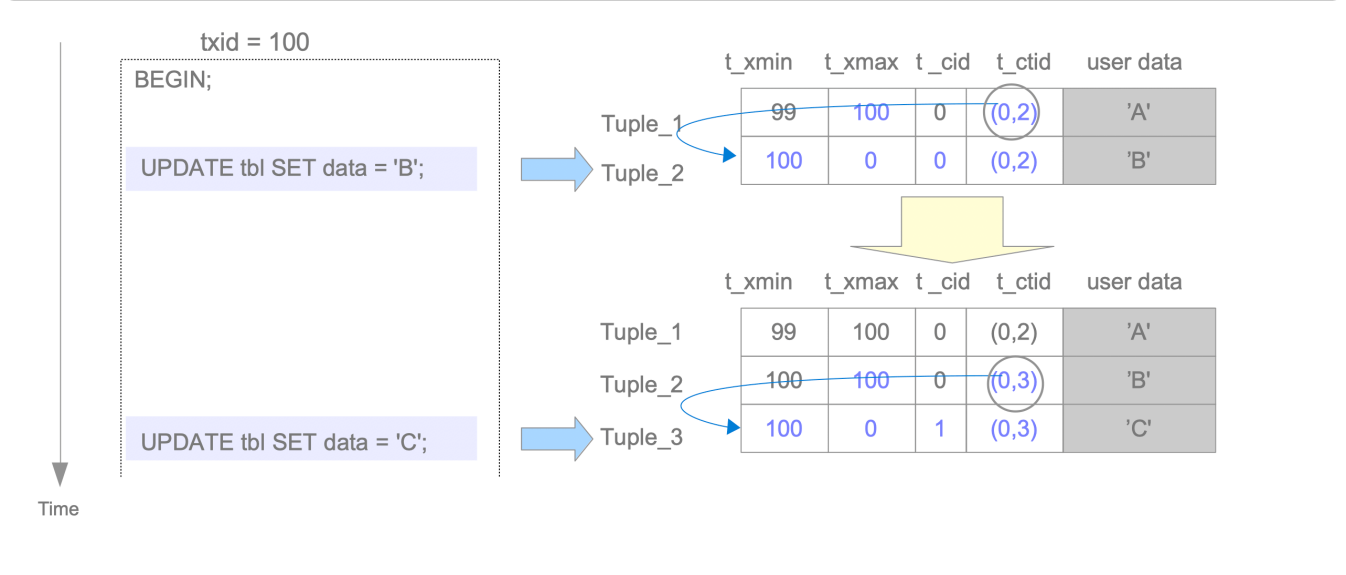
If txid 111 is committed, Tuple_1 is no longer required. Generally, unneeded tuples are referred to as **dead tuples** in PostgreSQL.

Dead tuples should eventually be removed from pages. Cleaning dead tuples is referred to as **VACUUM** processing, which is described in Chapter 6.

5.3.3. Update

In the update operation, PostgreSQL logically deletes the latest tuple and inserts a new one (Fig. 5.6).

Fig. 5.6. Update the row twice.



Suppose that the row, which has been inserted by txid 99, is updated twice by txid 100.

When the first UPDATE command is executed, Tuple_1 is logically deleted by setting txid 100 to the *t_xmax*, and then Tuple_2 is inserted. Then, the *t_ctid* of Tuple_1 is rewritten to point to Tuple_2. The header fields of both Tuple_1 and Tuple_2 are as follows:

Tuple_1:
- **t_xmax** is set to 100.
- **t_ctid** is rewritten from (0, 1) to (0, 2).

Tuple_2:
- **t_xmin** is set to 100.
- **t_xmax** is set to 0.
- **t_cid** is set to 0.
- **t_ctid** is set to (0,2).

When the second UPDATE command is executed, as in the first UPDATE command, Tuple_2 is logically deleted and Tuple_3 is inserted. The header fields of both Tuple_2 and Tuple_3 are as follows:

Tuple_2:
- **t_xmax** is set to 100.
- **t_ctid** is rewritten from (0, 2) to (0, 3).

Tuple_3:
- **t_xmin** is set to 100.
- **t_xmax** is set to 0.
- **t_cid** is set to 1.
- **t_ctid** is set to (0,3).

As with the delete operation, if txid 100 is committed, Tuple_1 and Tuple_2 will be dead tuples, and, if txid 100 is aborted, Tuple_2 and Tuple_3 will be dead tuples.

5.3.4. Free Space Map

When inserting a heap or an index tuple, PostgreSQL uses the **FSM** of the corresponding table or index to select the page which can be inserted into.

As mentioned in Section 1.2.3, all tables and indexes have respective FSMs. Each FSM stores the information about the free space capacity of each page within the corresponding table or index file.

All FSMs are stored with the suffix 'fsm', and they are loaded into shared memory if necessary.

pg_freespacemap

The extension pg_freespacemap provides the freespace of the specified table/index. The following query shows the freespace ratio of each page in the specified table.

```
testdb=# CREATE EXTENSION pg_freespacemap;
CREATE EXTENSION

testdb=# SELECT *, round(100 * avail/8192 ,2) as "freespace ratio"
                FROM pg_freespace('accounts');
 blkno | avail | freespace ratio
-------+-------+-----------------
     0 |  7904 |           96.00
     1 |  7520 |           91.00
     2 |  7136 |           87.00
     3 |  7136 |           87.00
     4 |  7136 |           87.00
     5 |  7136 |           87.00
....
```

5.4. Commit Log (clog)

PostgreSQL holds the statuses of transactions in the **Commit Log**. The Commit Log, often called the **clog**, is allocated to shared memory and is used throughout transaction processing.

This section describes the the status of transactions in PostgreSQL, how the clog operates, and maintenance of the clog.

5.4.1 Transaction Status

PostgreSQL defines four transaction states: IN_PROGRESS, COMMITTED, ABORTED, and SUB_COMMITTED.

The first three statuses are self-explanatory. For example, when a transaction is in progress, its status is IN_PROGRESS.

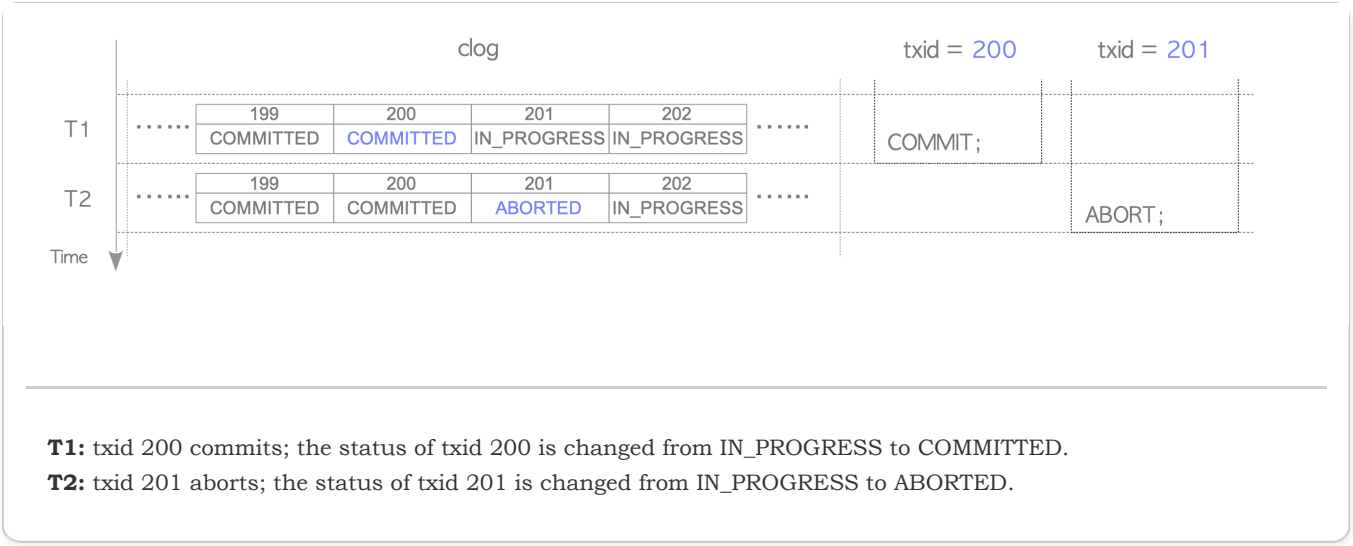
SUB_COMMITTED is for sub-transactions, and its description is omitted in this document.

5.4.2. How Clog Performs

The clog comprises one or more 8 KB pages in shared memory. It logically forms an array, where the indices of the array correspond to the respective transaction ids, and each item in the array

holds the status of the corresponding transaction id. Figure 5.7 shows the clog and how it operates.

Fig. 5.7. How the clog operates.



T1: txid 200 commits; the status of txid 200 is changed from IN_PROGRESS to COMMITTED.
T2: txid 201 aborts; the status of txid 201 is changed from IN_PROGRESS to ABORTED.

When the current txid advances and the clog can no longer store it, a new page is appended.

When the status of a transaction is needed, internal functions are invoked. Those functions read the clog and return the status of the requested transaction. (See also ℹ 'Hint Bits' in Section 5.7.1.)

5.4.3. Maintenance of the Clog

When PostgreSQL shuts down or whenever the checkpoint process runs, the data of the clog are written into files stored in the **pg_xact** subdirectory. (Note that pg_xact was called *pg_clog* in versions 9.6 or earlier.) These files are named *0000*, *0001*, and so on. The maximum file size is 256 KB. For example, if the clog uses eight pages (the first page to the eighth page, the total size is 64 KB), its data are written into 0000 (64 KB). If the clog uses 37 pages (296 KB), its data are written into 0000 and 0001, which are 256 KB and 40 KB in size, respectively.

When PostgreSQL starts up, the data stored in the pg_xact files are loaded to initialize the clog.

The size of the clog continuously increases because a new page is appended whenever the clog is filled up. However, not all data in the clog are necessary. Vacuum processing, described in Chapter 6, regularly removes such old data (both the clog pages and files). Details about removing the clog data are described in Section 6.4.

5.5. Transaction Snapshot

A **transaction snapshot** is a dataset that stores information about whether all transactions are active at a certain point in time for an individual transaction. Here an active transaction means it is in progress or has not yet started.

PostgreSQL internally defines the textual representation format of transaction snapshots as '100:100:'. For example, '100:100:' means 'txids that are less than 99 are not active, and txids that are equal or greater than 100 are active'. In the following descriptions, this convenient representation form is used. If you are not familiar with it, see ℹ below.

The built-in function pg_current_snapshot and its textual representation format

The function pg_current_snapshot shows a snapshot of the current transaction.

```
testdb=# SELECT pg_current_snapshot();
 pg_current_snapshot
---------------------
 100:104:100,102
(1 row)
```

The textual representation of the txid_current_snapshot is **'xmin:xmax:xip_list'**, and the components are described as follows:

- xmin

  (earliest txid that is still active): All earlier transactions will either be committed and visible, or rolled back and dead.

- xmax

  (first as-yet-unassigned txid): All txids greater than or equal to this are not yet started as of the time of the snapshot, and thus invisible.

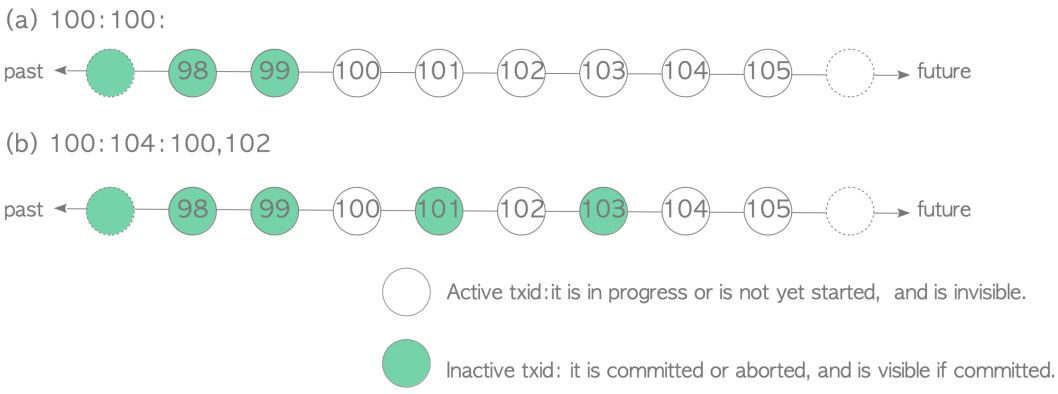
- xip_list

  (list of active transaction ids at the time of the snapshot): The list includes only active txids between xmin and xmax.

For example, in the snapshot '100:104:100,102', xmin is '100', xmax '104', and xip_list '100,102'.

Here are two specific examples:

Fig. 5.8. Examples of transaction snapshot representation.



The first example is '**100:100:**'. This snapshot means the following (Fig. 5.8(a)):

- txids equal or less than 99 are **not** *active* because xmin is 100.
- txids equal or greater than 100 are *active* because xmax is 100.

The second example is '**100:104:100,102**'. This snapshot means the following (Fig. 5.8(b)):

- txids equal or less than 99 are **not** *active*.
- txids equal or greater than 104 are *active*.
- txids 100 and 102 are *active* since they exist in the xip list, whereas txids 101 and 103 are **not** *active*.

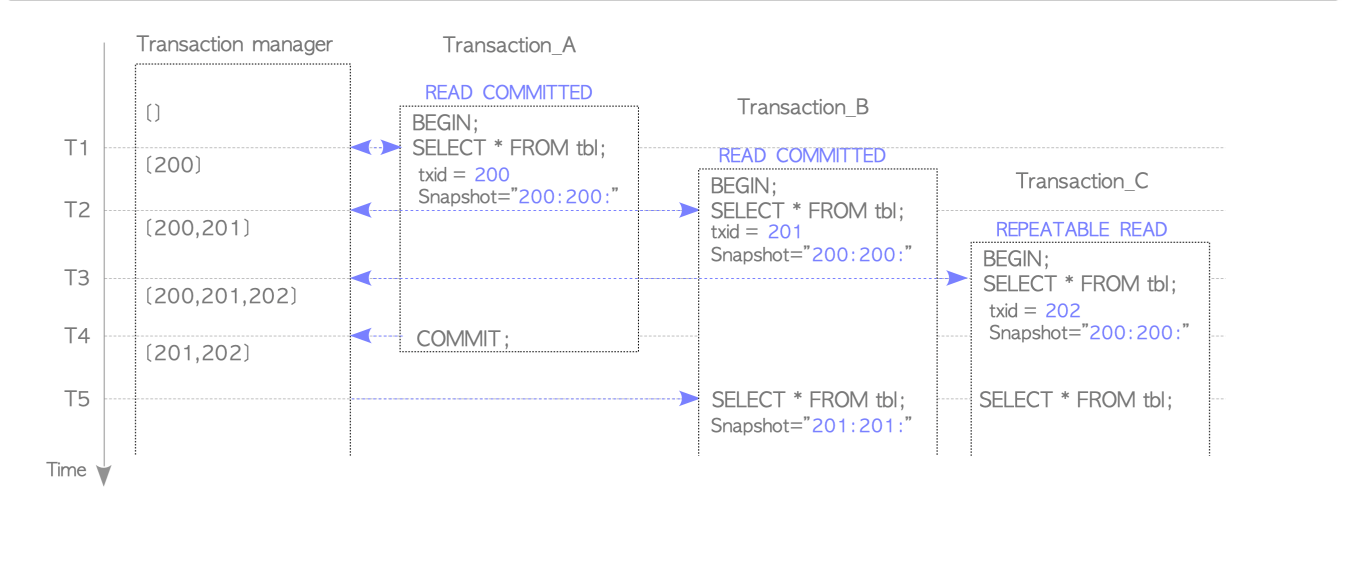
Transaction snapshots are provided by the transaction manager. In the READ COMMITTED isolation level, the transaction obtains a snapshot whenever an SQL command is executed;

otherwise (REPEATABLE READ or SERIALIZABLE), the transaction only gets a snapshot when the first SQL command is executed. The obtained transaction snapshot is used for a visibility check of tuples, which is described in Section 5.7.

When using the obtained snapshot for the visibility check, *active* transactions in the snapshot must be treated as *in progress* even if they have actually been committed or aborted. This rule is important because it causes the difference in the behavior between READ COMMITTED and REPEATABLE READ (or SERIALIZABLE). We refer to this rule repeatedly in the following sections.

In the remainder of this section, the transaction manager and transactions are described using a specific scenario Fig. 5.9.

Fig. 5.9. Transaction manager and transactions.



The transaction manager always holds information about currently running transactions. Suppose that three transactions start one after another, and the isolation level of Transaction_A and Transaction_B are READ COMMITTED, and that of Transaction_C is REPEATABLE READ.

T1:

Transaction_A starts and executes the first SELECT command. When executing the first command, Transaction_A requests the txid and snapshot of this moment. In this scenario, the transaction manager assigns txid 200, and returns the transaction snapshot '200:200:'.

T2:

Transaction_B starts and executes the first SELECT command. The transaction manager assigns txid 201, and returns the transaction snapshot '200:200:' because Transaction_A (txid 200) is in progress. Thus, Transaction_A cannot be seen from Transaction_B.

T3:

Transaction_C starts and executes the first SELECT command. The transaction manager assigns txid 202, and returns the transaction snapshot '200:200:', thus, Transaction_A and Transaction_B cannot be seen from Transaction_C.

T4:

Transaction_A has been committed. The transaction manager removes the information about this transaction.

T5:

Transaction_B and Transaction_C execute their respective SELECT commands.

Transaction_B requires a transaction snapshot because it is in the READ COMMITTED level. In this scenario, Transaction_B obtains a new snapshot '201:201:' because Transaction_A (txid

200) is committed. Thus, Transaction_A is no longer invisible from Transaction_B.
Transaction_C does not require a transaction snapshot because it is in the REPEATABLE READ level and uses the obtained snapshot, i.e. '200:200:'. Thus, Transaction_A is still invisible from Transaction_C.

5.6. Visibility Check Rules

Visibility check rules are a set of rules used to determine whether each tuple is visible or invisible using both the t_xmin and t_xmax of the tuple, the clog, and the obtained transaction snapshot. These rules are too complicated to explain in detail. Therefore this document shows the minimal rules required for the subsequent descriptions. In the following, we omit the rules related to sub-transactions and ignore discussion about t_ctid, i.e. we do not consider tuples that have been updated more than twice within a transaction.

The number of selected rules is ten, and they can be classified into three cases.

5.6.1. Status of t_xmin is ABORTED

A tuple whose t_xmin status is ABORTED is always *invisible* (Rule 1) because the transaction that inserted this tuple has been aborted.

```
/* t_xmin status == ABORTED */
Rule 1: IF t_xmin status is 'ABORTED' THEN
            RETURN 'Invisible'
        END IF
```

This rule is explicitly expressed as the following mathematical expression.

Rule 1: If Status(t_xmin) = ABORTED ⇒ Invisible

5.6.2. Status of t_xmin is IN_PROGRESS

A tuple whose t_xmin status is IN_PROGRESS is essentially *invisible* (Rules 3 and 4), except under one condition.

```
/* t_xmin status == IN_PROGRESS */
        IF t_xmin status is 'IN_PROGRESS' THEN
            IF t_xmin = current_txid THEN
Rule 2:         IF t_xmax = INVALID THEN
                    RETURN 'Visible'
Rule 3:         ELSE  /* this tuple has been deleted or updated by the current transaction itself. */
                    RETURN 'Invisible'
                END IF
Rule 4:     ELSE   /* t_xmin ≠ current_txid */
                RETURN 'Invisible'
            END IF
        END IF
```

If this tuple is inserted by another transaction and the status of t_xmin is IN_PROGRESS, then this tuple is obviously *invisible* (Rule 4).

If t_xmin is equal to the current txid (i.e., this tuple is inserted by the current transaction) and t_xmax is **not** INVALID, then this tuple is *invisible* because it has been updated or deleted by the current transaction (Rule 3).

The exception condition is the case where this tuple is inserted by the current transaction and t_xmax is INVALID. In this case, this tuple must be *visible* from the current transaction (Rule 2)

because this tuple is the tuple inserted by the current transaction itself.

Rule 2: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax = INVAILD ⇒ Visible
Rule 3: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin = current_txid ∧ t_xmax ≠ INVAILD ⇒ Invisible
Rule 4: If Status(t_xmin) = IN_PROGRESS ∧ t_xmin ≠ current_txid ⇒ Invisible

5.6.3. Status of t_xmin is COMMITTED

A tuple whose t_xmin status is COMMITTED is *visible* (Rules 6,8, and 9), except under three conditions.

```
/* t_xmin status == COMMITTED */
        IF t_xmin status is 'COMMITTED' THEN
Rule 5:     IF t_xmin is active in the obtained transaction snapshot THEN
                RETURN 'Invisible'
Rule 6:     ELSE IF t_xmax = INVALID OR status of t_xmax is 'ABORTED' THEN
                RETURN 'Visible'
            ELSE IF t_xmax status is 'IN_PROGRESS' THEN
Rule 7:         IF t_xmax =  current_txid THEN
                    RETURN 'Invisible'
Rule 8:         ELSE  /* t_xmax ≠ current_txid */
                    RETURN 'Visible'
                END IF
            ELSE IF t_xmax status is 'COMMITTED' THEN
Rule 9:         IF t_xmax is active in the obtained transaction snapshot THEN
                    RETURN 'Visible'
Rule 10:        ELSE
                    RETURN 'Invisible'
                END IF
            END IF
        END IF
```

Rule 6 is obvious because t_xmax is INVALID or ABORTED. Three exception conditions and both Rules 8 and 9 are described as follows:

The first exception condition is that t_xmin is *active* in the obtained transaction snapshot (Rule 5). Under this condition, this tuple is *invisible* because t_xmin should be treated as in progress.

The second exception condition is that t_xmax is the current txid (Rule 7). Under this condition, as with Rule 3, this tuple is *invisible* because it has been updated or deleted by this transaction itself.

In contrast, if the status of t_xmax is IN_PROGRESS and t_xmax is not the current txid (Rule 8), the tuple is *visible* because it has not been deleted.

The third exception condition is that the status of t_xmax is COMMITTED and t_xmax is **not** active in the obtained transaction snapshot (Rule 10). Under this condition, this tuple is *invisible* because it has been updated or deleted by another transaction.

In contrast, if the status of t_xmax is COMMITTED but t_xmax is active in the obtained transaction snapshot (Rule 9), the tuple is *visible* because t_xmax should be treated as in progress.

Rule 5: If Status(t_xmin) = COMMITTED ∧ Snapshot(t_xmin) = active ⇒ Invisible
Rule 6: If Status(t_xmin) = COMMITTED ∧ (t_xmax = INVALID ∨ Status(t_xmax) = ABORTED) ⇒ Visible
Rule 7: If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = IN_PROGRESS ∧ t_xmax = current_txid ⇒ Invisible
Rule 8: If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = IN_PROGRESS ∧ t_xmax ≠ current_txid ⇒ Visible

Rule 9: If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = COMMITTED ∧ Snapshot(t_xmax) = active ⇒ Visible

Rule 10: If Status(t_xmin) = COMMITTED ∧ Status(t_xmax) = COMMITTED ∧ Snapshot(t_xmax) ≠ active ⇒ Invisible

---

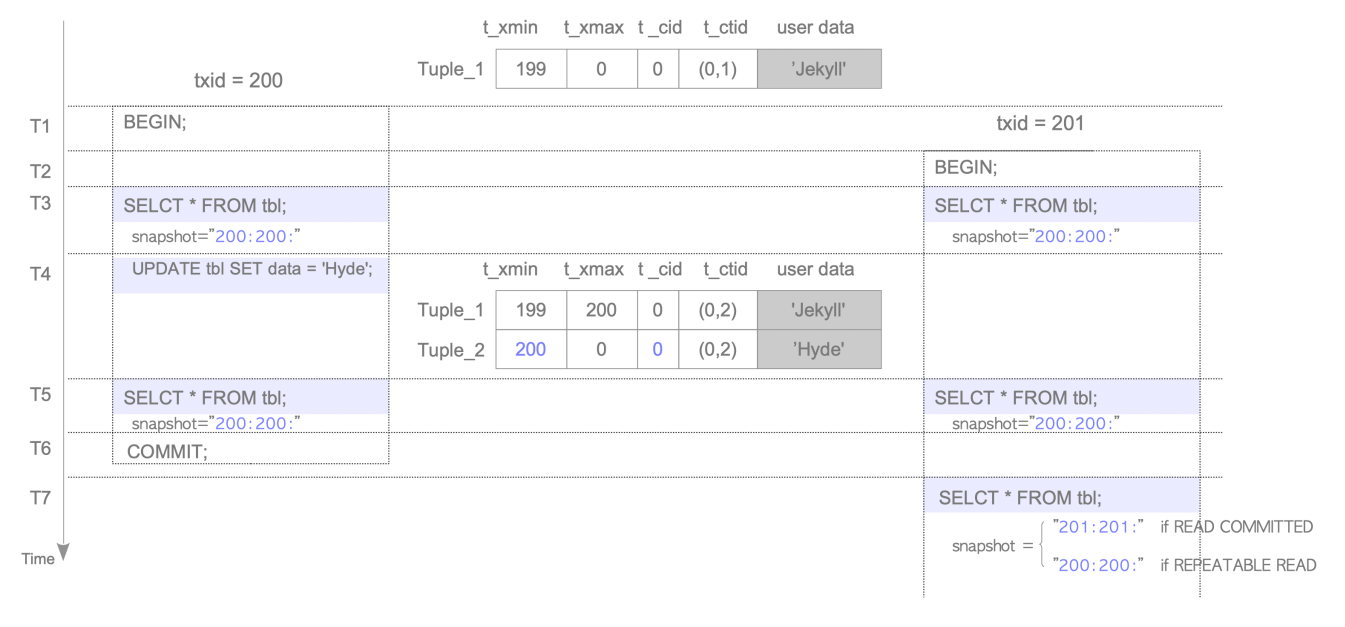
5.7. Visibility Check

This section describes how PostgreSQL performs a visibility check, which is the process of selecting heap tuples of the appropriate versions in a given transaction. This section also describes how PostgreSQL prevents the anomalies defined in the ANSI SQL-92 Standard: Dirty Reads, Repeatable Reads and Phantom Reads.

5.7.1. Visibility Check

Figure 5.10 shows a scenario to describe the visibility check.

Fig. 5.10. Scenario to describe visibility check.



In the scenario shown in Fig. 5.10, SQL commands are executed in the following time sequence.

T1: Start transaction (txid 200)
T2: Start transaction (txid 201)
T3: Execute SELECT commands of txid 200 and 201
T4: Execute UPDATE command of txid 200
T5: Execute SELECT commands of txid 200 and 201
T6: Commit txid 200
T7: Execute SELECT command of txid 201

To simplify the description, assume that there are only two transactions, i.e. txid 200 and 201. The isolation level of txid 200 is READ COMMITTED, and the isolation level of txid 201 is either READ COMMITTED or REPEATABLE READ.

We explore how SELECT commands perform a visibility check for each tuple.

SELECT commands of T3:

At T3, there is only Tuple_1 in the table *tbl* and it is *visible* by **Rule 6**. Therefore, SELECT commands in both transactions return 'Jekyll'.

- Rule6(Tuple_1) ⇒ Status(t_xmin:199) = COMMITTED ∧ t_xmax = INVALID ⇒ Visible

```
testdb=# -- txid 200
testdb=# SELECT * FROM tbl;
 name
--------
 Jekyll
(1 row)
```

```
testdb=# -- txid 201
testdb=# SELECT * FROM tbl;
 name
--------
 Jekyll
(1 row)
```

SELECT commands of T5:

First, we explore the SELECT command executed by txid 200. Tuple_1 is invisible by **Rule 7** and Tuple_2 is visible by **Rule 2**. Therefore, this SELECT command returns 'Hyde'.

- Rule7(Tuple_1): Status(t_xmin:199) = COMMITTED ∧ Status(t_xmax:200) = IN_PROGRESS ∧ t_xmax:200 = current_txid:200 ⇒ Invisible
- Rule2(Tuple_2): Status(t_xmin:200) = IN_PROGRESS ∧ t_xmin:200 = current_txid:200 ∧ t_xmax = INVAILD ⇒ Visible

```
testdb=# -- txid 200
testdb=# SELECT * FROM tbl;
 name
------
 Hyde
(1 row)
```

On the other hand, in the SELECT command executed by txid 201, Tuple_1 is visible by **Rule 8** and Tuple_2 is invisible by **Rule 4**. Therefore, this SELECT command returns 'Jekyll'.

- Rule8(Tuple_1): Status(t_xmin:199) = COMMITTED ∧ Status(t_xmax:200) = IN_PROGRESS ∧ t_xmax:200 ≠ current_txid:201 ⇒ Visible
- Rule4(Tuple_2): Status(t_xmin:200) = IN_PROGRESS ∧ t_xmin:200 ≠ current_txid:201 ⇒ Invisible

```
testdb=# -- txid 201
testdb=# SELECT * FROM tbl;
 name
--------
 Jekyll
(1 row)
```

If the updated tuples are visible from other transactions before they are committed, this is known as **Dirty Reads**, also known as **wr-conflicts**. However, as shown above, Dirty Reads do not occur in any isolation levels in PostgreSQL.

SELECT command of T7:

In the following, the behaviors of SELECT commands of T7 in both isolation levels are described.

When txid 201 is in the READ COMMITTED level, txid 200 is treated as COMMITTED because the transaction snapshot is *'201:201:'*. Therefore, Tuple_1 is *invisible* by **Rule 10** and Tuple_2 is *visible* by **Rule 6**. The SELECT command returns 'Hyde'.

- Rule10(Tuple_1): Status(t_xmin:199) = COMMITTED ∧ Status(t_xmax:200) = COMMITTED ∧ Snapshot(t_xmax:200) ≠ active ⇒ Invisible

- Rule6(Tuple_2): Status(t_xmin:200) = COMMITTED ∧ t_xmax = INVALID ⇒ Visible

```
testdb=# -- txid 201 (READ COMMITTED)
testdb=# SELECT * FROM tbl;
 name
------
 Hyde
(1 row)
```

Note that the results of the SELECT commands, which are executed before and after txid 200 is committed, differ. This is generally known as **Non-Repeatable Reads**.

In contrast, when txid 201 is in the REPEATABLE READ level, txid 200 must be treated as IN_PROGRESS because the transaction snapshot is '200:200:'. Therefore, Tuple_1 is *visible* by **Rule 9** and Tuple_2 is *invisible* by **Rule 5**. The SELECT command returns 'Jekyll'. Note that Non-Repeatable Reads do not occur in the REPEATABLE READ (and SERIALIZABLE) level.

- Rule9(Tuple_1): Status(t_xmin:199) = COMMITTED ∧ Status(t_xmax:200) = COMMITTED ∧ Snapshot(t_xmax:200) = active ⇒ Visible
- Rule5(Tuple_2): Status(t_xmin:200) = COMMITTED ∧ Snapshot(t_xmin:200) = active ⇒ Invisible

```
testdb=# -- txid 201 (REPEATABLE READ)
testdb=# SELECT * FROM tbl;
  name
--------
 Jekyll
(1 row)
```

Hint Bits

To obtain the status of a transaction, PostgreSQL internally provides three functions: TransactionIdIsInProgress, TransactionIdDidCommit, and TransactionIdDidAbort. These functions are implemented to reduce frequent access to the clog, such as caches. However, bottlenecks will occur if they are executed whenever each tuple is checked.

To deal with this issue, PostgreSQL uses *hint bits*, which are shown blow:

```
#define HEAP_XMIN_COMMITTED     0x0100  /* t_xmin committed */
#define HEAP_XMIN_INVALID       0x0200  /* t_xmin invalid/aborted */
#define HEAP_XMAX_COMMITTED     0x0400  /* t_xmax committed */
#define HEAP_XMAX_INVALID       0x0800  /* t_xmax invalid/aborted */
```

When reading or writing a tuple, PostgreSQL sets hint bits to the t_informask of the tuple if possible. For example, assume that PostgreSQL checks the status of the t_xmin of a tuple and obtains the status COMMITTED. In this case, PostgreSQL sets a hint bit HEAP_XMIN_COMMITTED to the t_infomask of the tuple. If hint bits are already set, TransactionIdDidCommit and TransactionIdDidAbort are no longer needed. Therefore, PostgreSQL can efficiently check the statuses of both t_xmin and t_xmax of each tuple.

5.7.2. Phantom Reads in PostgreSQL's REPEATABLE READ Level

REPEATABLE READ as defined in the ANSI SQL-92 standard allows **Phantom Reads**. However, PostgreSQL's implementation does not allow them. In principle, SI does not allow Phantom Reads.

Assume that two transactions, i.e. Tx_A and Tx_B, are running concurrently. Their isolation levels are READ COMMITTED and REPEATABLE READ, and their txids are 100 and 101, respectively. First, Tx_A inserts a tuple. Then, it is committed. The t_xmin of the inserted tuple is 100. Next,

Tx_B executes a SELECT command; however, the tuple inserted by Tx_A is *invisible* by **Rule 5**. Thus, Phantom Reads do not occur.

- Rule5(new tuple): Status(t_xmin:100) = COMMITTED ∧ Snapshot(t_xmin:100) = active ⇒ Invisible

```
testdb=# -- Tx_A: txid 100
testdb=# START TRANSACTION
testdb-#  ISOLATION LEVEL READ COMMITTED;
START TRANSACTION


testdb=# INSERT tbl(id, data)
            VALUES (1,'phantom');
INSERT 1

testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B: txid 101
testdb=# START TRANSACTION
testdb-#  ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
testdb=# SELECT txid_current();
 txid_current
--------------
        101
(1 row)

testdb=# SELECT * FROM tbl WHERE id=1;
 id | data
----+------
(0 rows)
```

5.8. Preventing Lost Updates

A **Lost Update**, also known as a **ww-conflict**, is an anomaly that occurs when concurrent transactions update the same rows, and it must be prevented in both the REPEATABLE READ and SERIALIZABLE levels. (Note that the READ COMMITTED level does not need to prevent Lost Updates.) This section describes how PostgreSQL prevents Lost Updates and shows examples.

5.8.1. Behavior of Concurrent UPDATE Commands

When an UPDATE command is executed, the function ExecUpdate is internally invoked. The pseudocode of the ExecUpdate is shown below:

</> Pseudocode: ExecUpdate

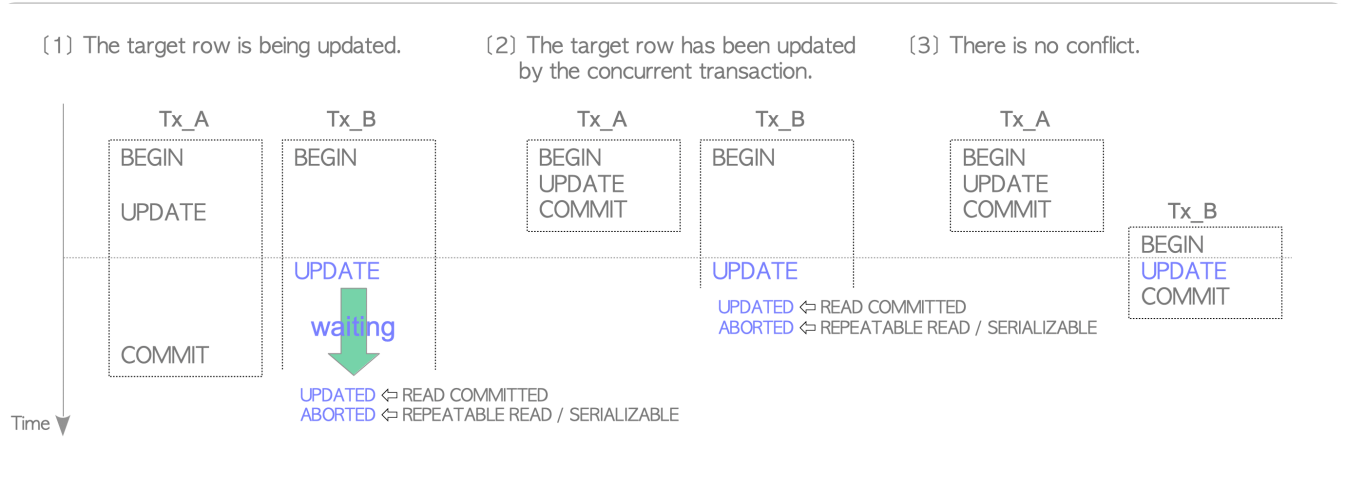
```
(1)  FOR each row that will be updated by this UPDATE command
(2)      WHILE true

           /* The First Block */
(3)         IF the target row is being updated THEN
(4)             WAIT for the termination of the transaction that updated the target row

(5)             IF (the status of the terminated transaction is COMMITTED)
                    AND (the isolation level of this transaction is REPEATABLE READ or SERIALIZABLE) TH
EN
```

```
(6)                 ABORT this transaction  /* First-Updater-Win */
                ELSE
(7)                 GOTO step (2)
                END IF

            /* The Second Block */
(8)          ELSE IF the target row has been updated by another concurrent transaction THEN
(9)             IF (the isolation level of this transaction is READ COMMITTED THEN
(10)                UPDATE the target row
                ELSE
(11)                ABORT this transaction  /* First-Updater-Win */
                END IF

            /* The Third Block */
             ELSE  /* The target row is not yet modified or has been updated by a terminated transaction.
*/
(12)            UPDATE the target row
            END IF
        END WHILE
    END FOR
```

(1) Get each row that will be updated by this UPDATE command.
(2) Repeat the following process until the target row has been updated (or this transaction is aborted).
(3) If the target row *is being updated*, go to step (3); otherwise, go to step (8).
(4) Wait for the termination of the transaction that updated the target row because PostgreSQL uses *first-updater-win* scheme in SI.
(5) If the status of the transaction that updated the target row is COMMITTED and the isolation level of this transaction is REPEATABLE READ (or SERIALIZABLE), go to step (6); otherwise, go to step (7).
(6) Abort this transaction to prevent Lost Updates.
(7) Go to step (2) and attempt to update the target row in the next round.
(8) If the target row has been updated by another concurrent transaction, go to step (9); otherwise, go to step (12).
(9) If the isolation level of this transaction is READ COMMITTED, go to step (10); otherwise, go to step (11).
(10) UPDATE the target row, and go to step (1).
(11) Abort this transaction to prevent Lost Updates.
(12) UPDATE the target row, and go to step (1) because the target row is not yet modified or has been updated by a terminated transaction, i.e. there is ww-conflict.

This function performs update operations for each of the target rows. It has a while loop to update each row, and the inside of the while loop branches to three blocks according to the conditions shown in Fig. 5.11.

Fig. 5.11. Three internal blocks in ExecUpdate.



[1] The target row is being updated (Fig. 5.11[1])

'Being updated' means that the row is being updated by another concurrent transaction and its transaction has not terminated. In this case, the current transaction must wait for termination of the transaction that updated the target row because PostgreSQL's SI uses the **first-updater-win** scheme. For example, assume that transactions Tx_A and Tx_B run concurrently, and Tx_B attempts to update a row; however, Tx_A has updated it and is still in progress. In this case, Tx_B waits for the termination of Tx_A.

After the transaction that updated the target row commits, the update operation of the current transaction proceeds. If the current transaction is in the READ COMMITTED level, the target row will be updated; otherwise (REPEATABLE READ or SERIALIZABLE), the current transaction is aborted immediately to prevent lost updates.

[2] The target row has been updated by the concurrent transaction (Fig. 5.11[2])

The current transaction attempts to update the target tuple; however, the other concurrent transaction has updated the target row and has already been committed. In this case, if the current transaction is in the READ COMMITTED level, the target row will be updated; otherwise, the current transaction is aborted immediately to prevent lost updates.

[3] There is no conflict (Fig. 5.11[3])

When there is no conflict, the current transaction can update the target row.

> **first-updater-win / first-commiter-win**
>
> As mentioned in this section, PostgreSQL's concurrency control based on SI uses the *first-updater-win* scheme to avoid lost update anomalies. In contrast, as explained in the next section, PostgreSQL's SSI uses the *first-committer-win* scheme to avoid serialization anomalies.

5.8.2. Examples

Three examples are shown in the following. The first and second examples show behaviours when the target row is being updated, and the third example shows the behaviour when the target row has been updated.

Example 1:

Transactions Tx_A and Tx_B update the same row in the same table, and their isolation level is READ COMMITTED.

```
testdb=# -- Tx_A
testdb=# START TRANSACTION
testdb-#    ISOLATION LEVEL READ COMMITTED;
START TRANSACTION


testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1



testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb-#    ISOLATION LEVEL READ COMMITTED;
START TRANSACTION
```

```
testdb=# UPDATE tbl SET name = 'Utterson';
   ↓
   ↓ this transaction is being blocked
   ↓
UPDATE 1
```

Tx_B is executed as follows.

1) After executing the UPDATE command, Tx_B should wait for the termination of Tx_A, because the target tuple is being updated by Tx_A (Step (4) in ExecUpdate).
2) After Tx_A is committed, Tx_B attempts to update the target row (Step (7) in ExecUpdate).
3) In the second round of ExecUpdate, the target row is updated again by Tx_B (Steps (2),(8),(9), (10) in ExecUpdate).

Example 2:

Tx_A and Tx_B update the same row in the same table, and their isolation levels are READ COMMITTED and REPEATABLE READ, respectively.

```
testdb=# -- Tx_A
testdb=# START TRANSACTION
testdb-#    ISOLATION LEVEL READ COMMITTED;
START TRANSACTION


testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1



testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb-#    ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION


testdb=# UPDATE tbl SET name = 'Utterson';
   ↓
   ↓ this transaction is being blocked
   ↓
ERROR:couldn't serialize access due to concurrent update
```

The behaviour of Tx_B is described as follows.

1) After executing the UPDATE command, Tx_B should wait for the termination of Tx_A (Step (4) in ExecUpdate).
2) After Tx_A is committed, Tx_B is aborted to resolve conflict because the target row has been updated and the isolation level of this transaction is REPEATABLE READ (Steps (5) and (6) in ExecUpdate).

Example 3:

Tx_B (REPEATABLE READ) attempts to update the target row that has been updated by the committed Tx_A. In this case, Tx_B is aborted (Steps (2),(8),(9), and (11) in ExecUpdate).

```
testdb=# -- Tx_A
testdb=# START TRANSACTION
```

```
testdb-#    ISOLATION LEVEL READ COMMITTED;
START TRANSACTION


testdb=# UPDATE tbl SET name = 'Hyde';
UPDATE 1


testdb=# COMMIT;
COMMIT
```

```
testdb=# -- Tx_B
testdb=# START TRANSACTION
testdb-#    ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION
testdb=# SELECT * FROM tbl;
  name
--------
 Jekyll
(1 row)
testdb=# UPDATE tbl SET name = 'Utterson';
ERROR:couldn't serialize access due to concurrent update
```

5.9. Serializable Snapshot Isolation

Serializable Snapshot Isolation (SSI) has been embedded in SI since version 9.1 to realize a true SERIALIZABLE isolation level. Since the explanation of SSI is not simple, only an outline is explained. For details, see [2].

In the following, the technical terms shown in below are used without definitions. If you are unfamiliar with these terms, see [1, 3].

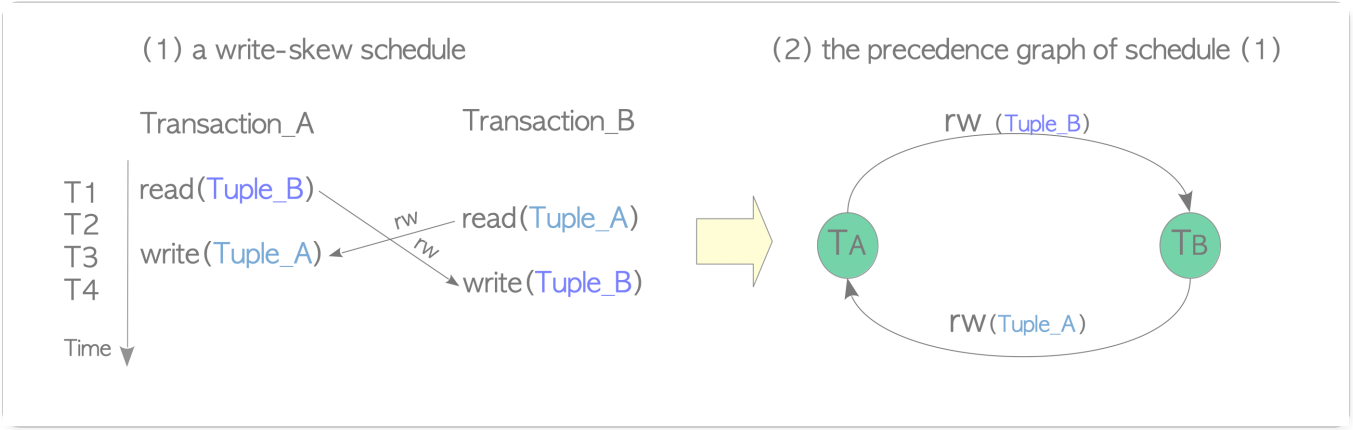
- *precedence graph* (also known as *dependency graph* and *serialization graph)*
- *serialization anomalies* (e.g. *Write-Skew)*

5.9.1. Basic Strategy for SSI Implementation

If a cycle is present in the precedence graph, there will be a serialization anomaly. This can be explained using the simplest anomaly, write-skew.

Figure 5.12(1) shows a schedule. Here, Transaction_A reads Tuple_B and Transaction_B reads Tuple_A. Then, Transaction_A writes Tuple_A and Transaction_B writes Tuple_B. In this case, there are two rw-conflicts, and they make a cycle in the precedence graph of this schedule, as shown in Fig. 5.12(2). Thus, this schedule has a serialization anomaly, Write-Skew.

Fig. 5.12. Write-Skew schedule and its precedence graph.



Conceptually, there are three types of conflicts: wr-conflicts (Dirty Reads), ww-conflicts (Lost Updates), and rw-conflicts. However, wr- and ww-conflicts do not need to be considered because PostgreSQL prevents such conflicts, as shown in the previous sections. Thus, SSI implementation in PostgreSQL only needs to consider rw-conflicts.

PostgreSQL takes the following strategy for the SSI implementation:

1. Record all objects (tuples, pages, relations) accessed by transactions as SIREAD locks.
2. Detect rw-conflicts using SIREAD locks whenever any heap or index tuple is written.
3. Abort the transaction if a serialization anomaly is detected by checking detected rw-conflicts.

5.9.2. Implementing SSI in PostgreSQL

To realize the strategy described above, PostgreSQL has implemented many functions and data structures. However, here we uses only two data structures: **SIREAD locks** and **rw-conflicts**, to describe the SSI mechanism. They are stored in shared memory.

For simplicity, some important data structures, such as SERIALIZABLEXACT, are omitted in this document. Thus, the explanations of the functions, i.e. CheckForSerializableConflictOut, CheckForSerializableConflictIn, and PreCommit_CheckForSerializationFailure, are also extremely simplified. For example, we indicate which functions detect conflicts; however, how the conflicts are detected is not explained in detail. If you want to know the details, refer to the source code: src/backend/storage/lmgr/predicate.c.

SIREAD locks:

An SIREAD lock, internally called a predicate lock, is a pair of an object and (virtual) txids that store information about who has accessed which object. Note that the description of virtual txid is omitted. The term txid is used rather than virtual txid to simplify the following explanation.

SIREAD locks are created by the CheckForSerializableConflictOut function whenever a DML command is executed in SERIALIZABLE mode. For example, if txid 100 reads Tuple_1 of the given table, an SIREAD lock {Tuple_1, {100}} is created. If another transaction, e.g. txid 101, reads Tuple_1, the SIREAD lock is updated to {Tuple_1, {100,101}}. Note that a SIREAD lock is also created when an index page is read because an index page is only read without reading the table page when the Index-Only Scans feature, which is described in Section 7.2, is applied.

SIREAD lock has three levels: tuple, page, and relation. If the SIREAD locks of all tuples within a single page are created, they are aggregated into a single SIREAD lock for that page, and all SIREAD locks of the associated tuples are released (removed), to reduce memory space. The same is true for all pages that are read.

When using sequential scan, a relation level SIREAD lock is created from the beginning regardless of the presence of indexes and/or WHERE clauses. Note that, in certain situations, this implementation can cause false-positive detections of serialization anomalies. The details are described in Section 5.9.4.

rw-conflicts:

A rw-conflict is a triplet of an SIREAD lock and two txids that reads and writes the SIREAD lock.

The CheckForSerializableConflictIn function is invoked whenever either an INSERT, UPDATE, or DELETE command is executed in SERIALIZABLE mode, and it creates rw-conflicts when detecting conflicts by checking SIREAD locks.

For example, assume that txid 100 reads Tuple_1 and then txid 101 updates Tuple_1. In this case, the CheckForSerializableConflictIn function, invoked by the UPDATE command in txid 101, detects a rw-conflict with Tuple_1 between txid 100 and 101 and then creates a rw-conflict {r=100, w=101, {Tuple_1}}.

Both the CheckForSerializableConflictOut and CheckForSerializableConflictIn functions, as well as the PreCommit_CheckForSerializationFailure function, which is invoked when the COMMIT command is executed in SERIALIZABLE mode, check serialization anomalies using the created rw-conflicts. If they detect anomalies, only the first-committed transaction is committed and the other transactions are aborted (by the **first-committer-win** scheme).

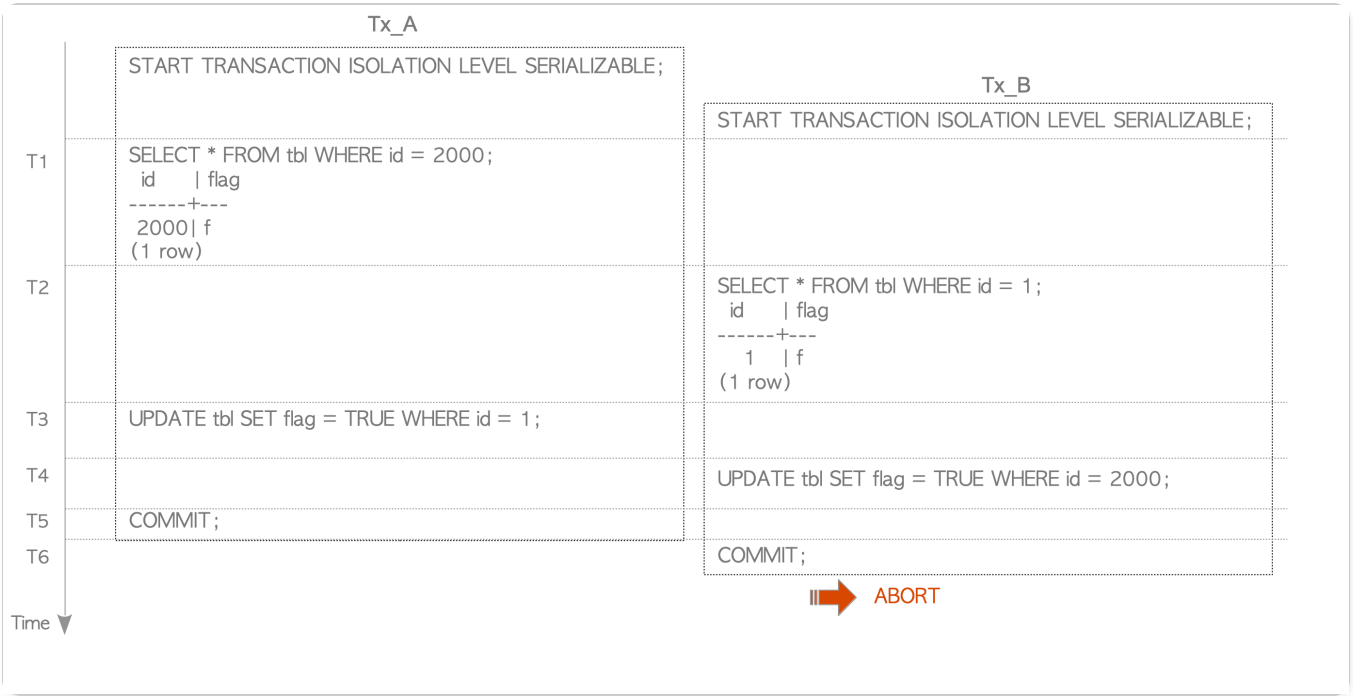
5.9.3. How SSI Performs

Here, we describe how SSI resolves Write-Skew anomalies. We use a simple table *tbl* shown below:

```
testdb=# CREATE TABLE tbl (id INT primary key, flag bool DEFAULT false);
testdb=# INSERT INTO tbl (id) SELECT generate_series(1,2000);
testdb=# ANALYZE tbl;
```

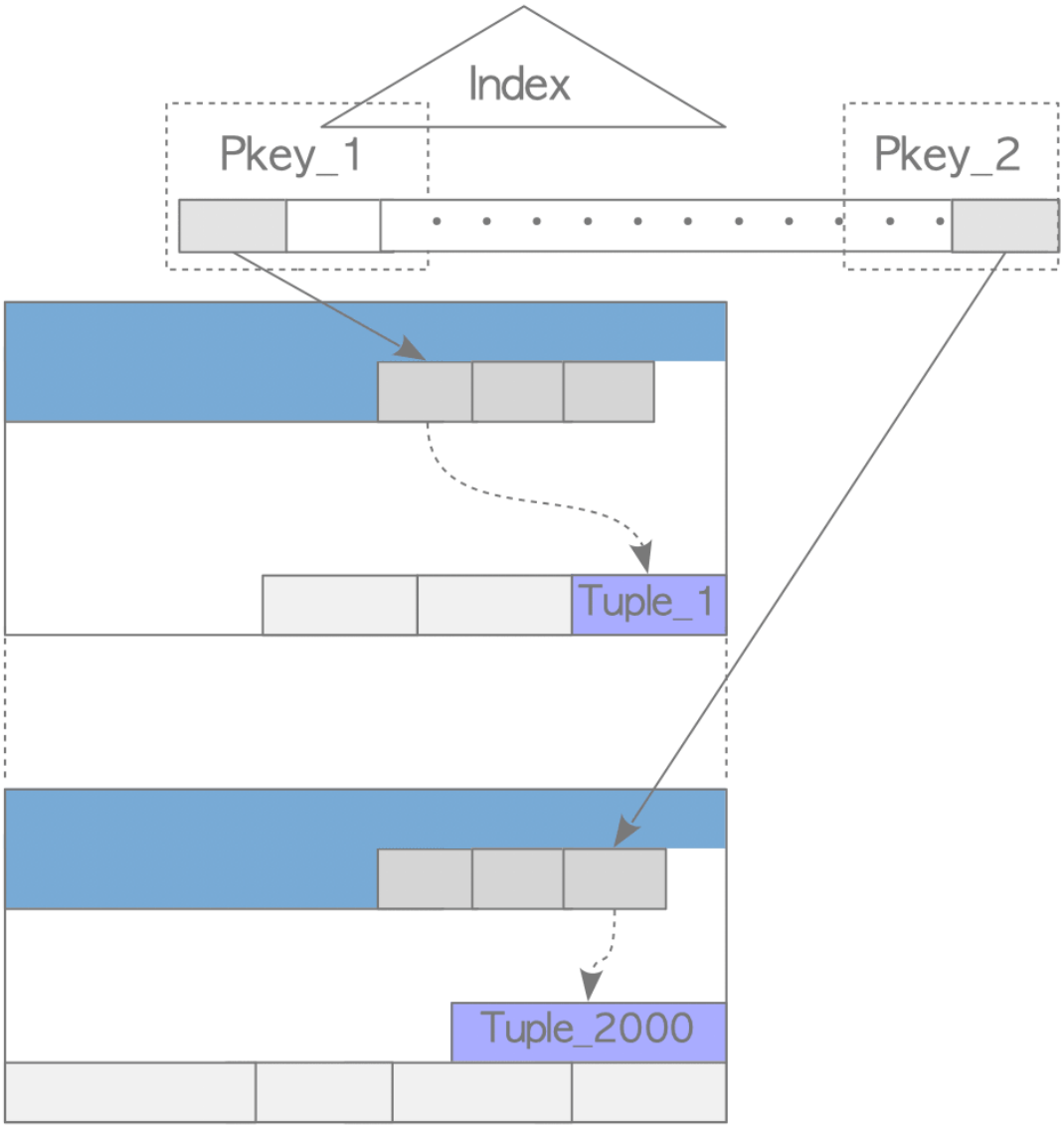
Transactions Tx_A and Tx_B execute the following commands (Fig. 5.13).

Fig. 5.13. Write-Skew scenario.



Assume that all commands use index scan. Therefore, when the commands are executed, they read both heap tuples and index pages, each of which contains the index tuple that points to the corresponding heap tuple. See Fig. 5.14.

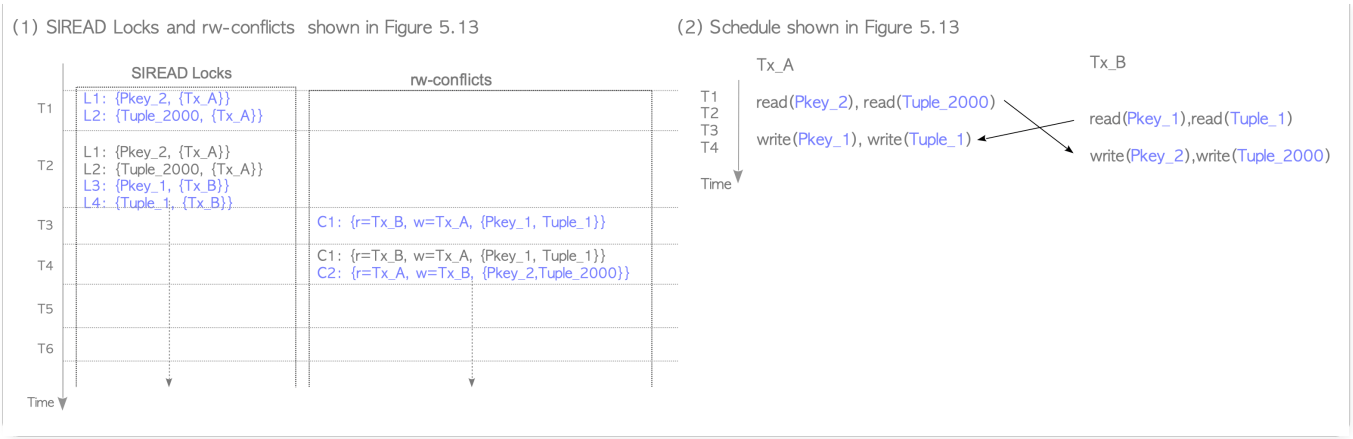
Fig. 5.14. Relationship between the index and table in the scenario shown in Fig. 5.13.



T1: Tx_A executes a SELECT command. This command reads a heap tuple (Tuple_2000) and one page of the primary key (Pkey_2).
T2: Tx_B executes a SELECT command. This command reads a heap tuple (Tuple_1) and one page of the primary key (Pkey_1).
T3: Tx_A executes an UPDATE command to update Tuple_1.
T4: Tx_B executes an UPDATE command to update Tuple_2000.
T5: Tx_A commits.
T6: Tx_B commits; however, it is aborted due to a Write-Skew anomaly.

Figure 5.15 shows how PostgreSQL detects and resolves the Write-Skew anomaly described in the above scenario.

Fig. 5.15. SIREAD locks and rw-conflicts, and schedule of the scenario shown in Fig. 5.13.



T1:

When executing the SELECT command of Tx_A, CheckForSerializableConflictOut creates SIREAD locks. In this scenario, the function creates two SIREAD locks: L1 and L2.
L1 and L2 are associated with Pkey_2 and Tuple_2000, respectively.

T2:

When executing the SELECT command of Tx_B, CheckForSerializableConflictOut creates two SIREAD locks: L3 and L4.
L3 and L4 are associated with Pkey_1 and Tuple_1, respectively.

T3:

When executing the UPDATE command of Tx_A, both CheckForSerializableConflictOut and CheckTargetForConflictsIN are invoked before and after ExecUpdate.
In this scenario, CheckForSerializableConflictOut does nothing.
CheckForSerializableConflictIn creates rw-conflict C1, which is the conflict of both Pkey_1 and Tuple_1 between Tx_B and Tx_A, because both Pkey_1 and Tuple_1 were read by Tx_B and written by Tx_A.

T4:

When executing the UPDATE command of Tx_B, CheckForSerializableConflictIn creates rw-conflict C2, which is the conflict of both Pkey_2 and Tuple_2000 between Tx_A and Tx_B.
In this scenario, C1 and C2 create a cycle in the precedence graph; thus, Tx_A and Tx_B are in a non-serializable state. However, both transactions Tx_A and Tx_B have not been committed, therefore CheckForSerializableConflictIn does not abort Tx_B. Note that this occurs because PostgreSQL's SSI implementation is based on the *first-committer-win* scheme.

T5:

When Tx_A attempts to commit, PreCommit_CheckForSerializationFailure is invoked. This function can detect serialization anomalies and can execute a commit action if possible. In this scenario, Tx_A is committed because Tx_B is still in progress.

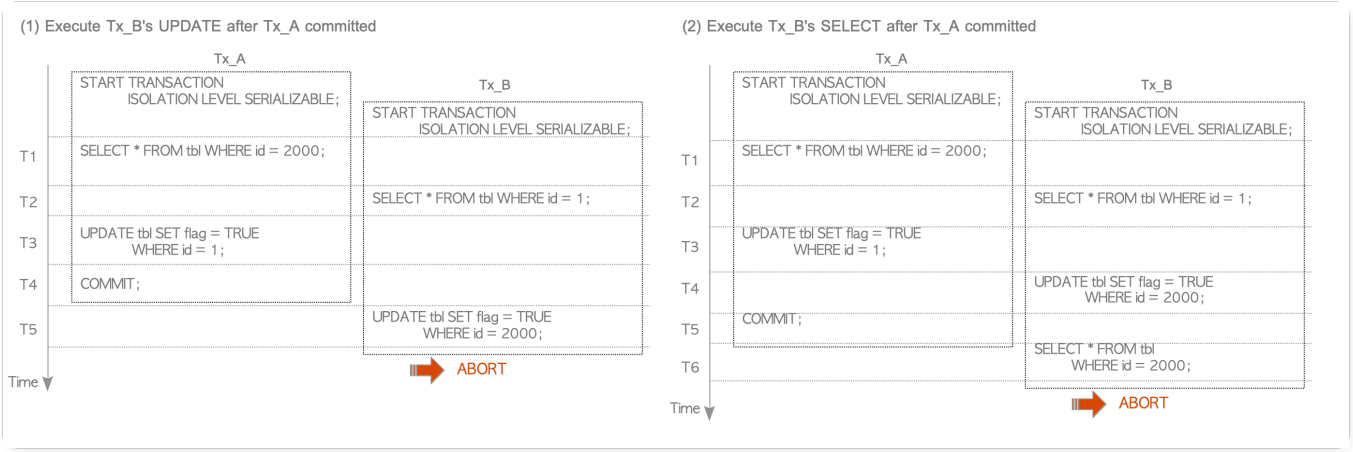
T6:

When Tx_B attempts to commit, PreCommit_CheckForSerializationFailure detects a serialization anomaly and Tx_A has already been committed; thus, Tx_B is aborted.

In addition, if the UPDATE command is executed by Tx_B after Tx_A has been committed (at **T5**), Tx_B is immediately aborted because CheckForSerializableConflictIn invoked by Tx_B's UPDATE command detects a serialization anomaly (Fig. 5.16(1)).

If the SELECT command is executed instead of COMMIT at **T6**, Tx_B is immediately aborted because CheckForSerializableConflictOut invoked by Tx_B's SELECT command detects a serialization anomaly (Fig. 5.16(2)).

Fig. 5.16. Other Write-Skew scenarios.



This Wiki explains several more complex anomalies.

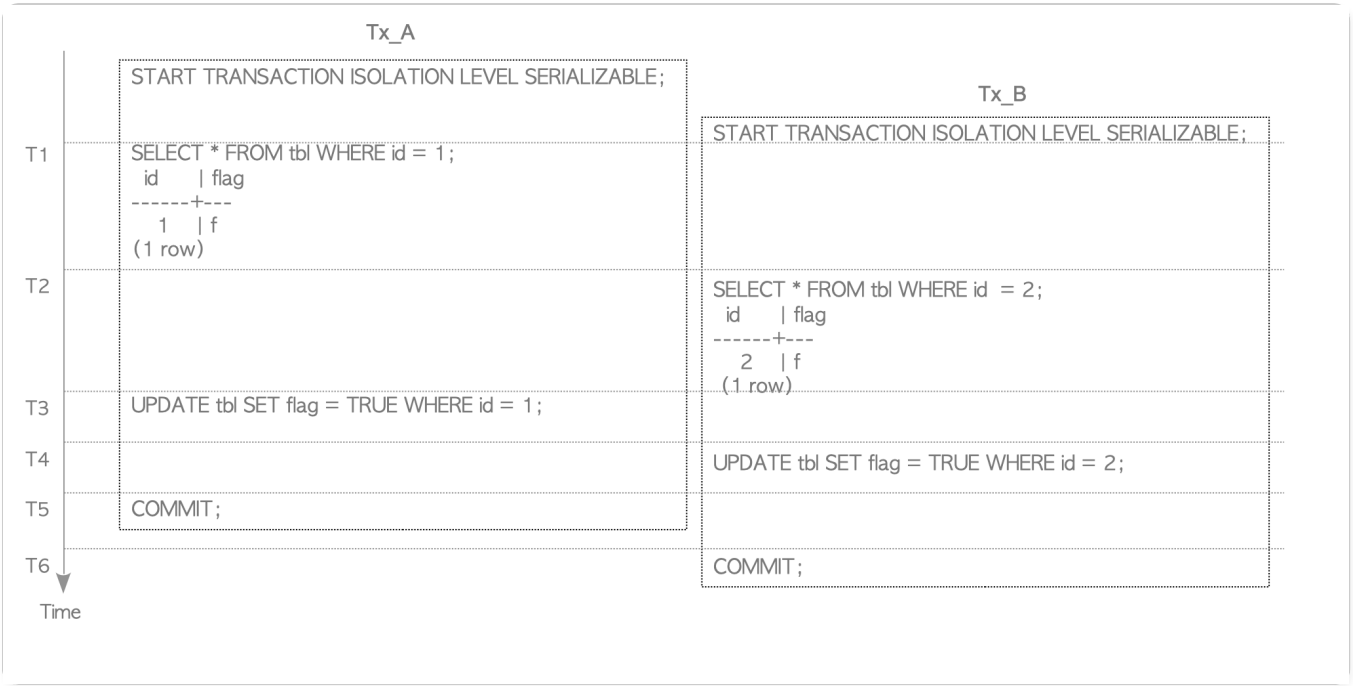
5.9.4. False-Positive Serialization Anomalies

In SERIALIZABLE mode, the serializability of concurrent transactions is always fully guaranteed because false-negative serialization anomalies are never detected. However, under some circumstances, false-positive anomalies can be detected; therefore, users should keep this in mind when using SERIALIZABLE mode. In the following, the situations in which PostgreSQL detects false-positive anomalies are described.

5.9.4.1. False-Positive Scenario 1.

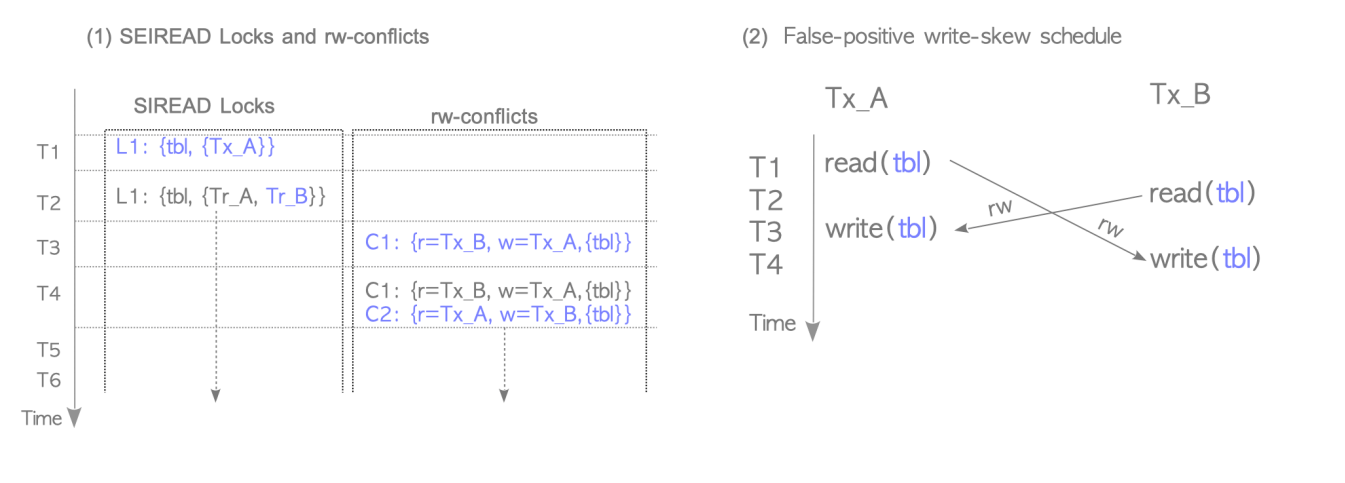
Figure 5.17 shows a scenario where a false-positive serialization anomaly occurs.

Fig. 5.17. Scenario where false-positive serialization anomaly occurs.



When using sequential scan, as mentioned in the explanation of SIREAD locks, PostgreSQL creates a relation level SIREAD lock. Figure 5.18(1) shows SIREAD locks and rw-conflicts when PostgreSQL uses sequential scan. In this case, rw-conflicts C1 and C2, which are associated with the tbl's SIREAD lock, are created, and they create a cycle in the precedence graph. Thus, a false-positive Write-Skew anomaly is detected (and either Tx_A or Tx_B will be aborted even though there is no conflict).

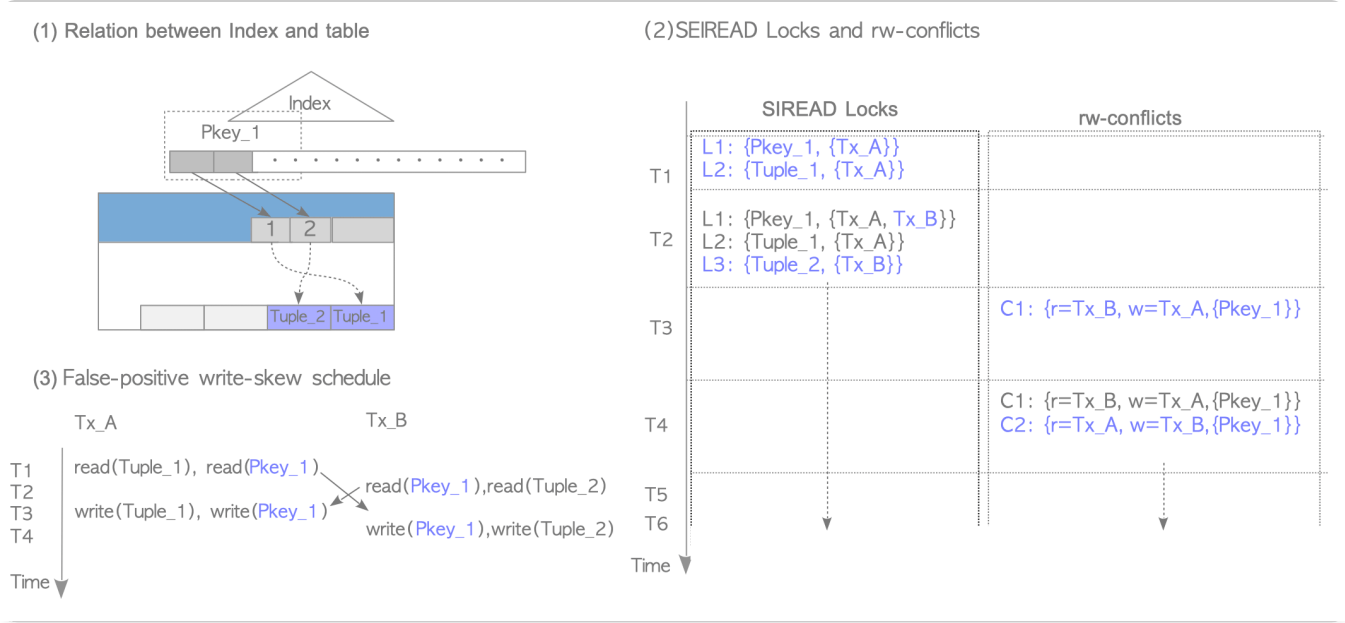
Fig. 5.18. False-positive anomaly (1) – Using sequential scan.



5.9.4.2. False-Positive Scenario 2.

Even when using index scan, if both transactions Tx_A and Tx_B get the same index SIREAD lock, PostgreSQL detects a false-positive anomaly. Figure 5.19 shows this situation.

Fig. 5.19. False-positive anomaly (2) – Index scan using the same index page.



Assume that the index page Pkey_1 contains two index items, one of which points to Tuple_1 and the other points to Tuple_2.

When Tx_A and Tx_B execute respective SELECT and UPDATE commands, Pkey_1 is read and written by both Tx_A and Tx_B. In this case, rw-conflicts C1 and C2, both of which are associated with Pkey_1, create a cycle in the precedence graph; thus, a false-positive Write-Skew anomaly is detected.

(If Tx_A and Tx_B get the SIREAD locks of different index pages, a false-positive is not detected and both transactions can be committed.)

5.10. Required Maintenance Processes

PostgreSQL's concurrency control mechanism requires the following maintenance processes:

1. Remove dead tuples and index tuples that point to corresponding dead tuples
2. Remove unnecessary parts of the clog
3. Freeze old txids
4. Update FSM, VM, and the statistics

The need for the first and second processes have been explained in Sections 5.3.2 and 5.4.3, respectively. The third process is related to the transaction id wraparound problem, which is briefly described in the following subsection.

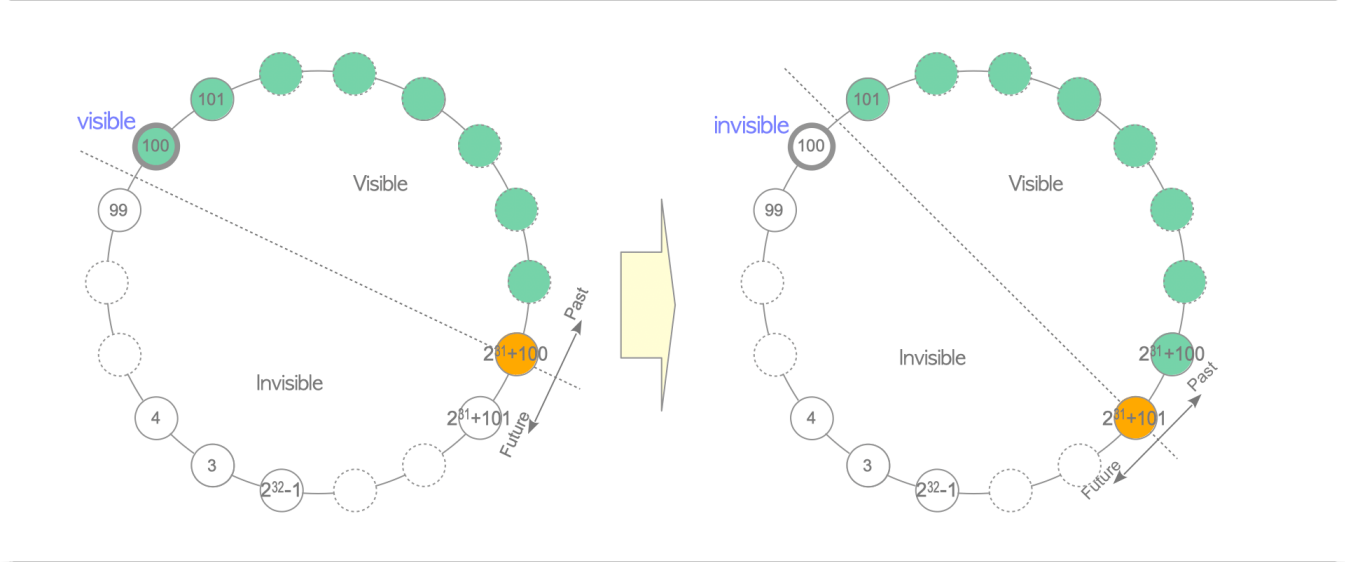
In PostgreSQL, **VACUUM** processing is responsible for these processes, and it is described in Chapter 6.

5.10.1. FREEZE Processing

Here, I describe the txid wraparound problem.

Assume that tuple Tuple_1 is inserted with a txid of 100, i.e. the t_xmin of Tuple_1 is 100. The server has been running for a very long period and Tuple_1 has not been modified. The current txid is 2.1 billion + 100 and a SELECT command is executed. At this time, Tuple_1 is *visible* because txid 100 is *in the past*. Then, the same SELECT command is executed; thus, the current txid is 2.1 billion + 101. However, Tuple_1 is *no longer visible* because txid 100 is *in the future* (Fig. 5.20). This is the so called *transaction wraparound problem* in PostgreSQL.

Fig. 5.20. Wraparound problem.



To deal with this problem, PostgreSQL introduced a concept called *frozen txid*, and implemented a process called *FREEZE*.

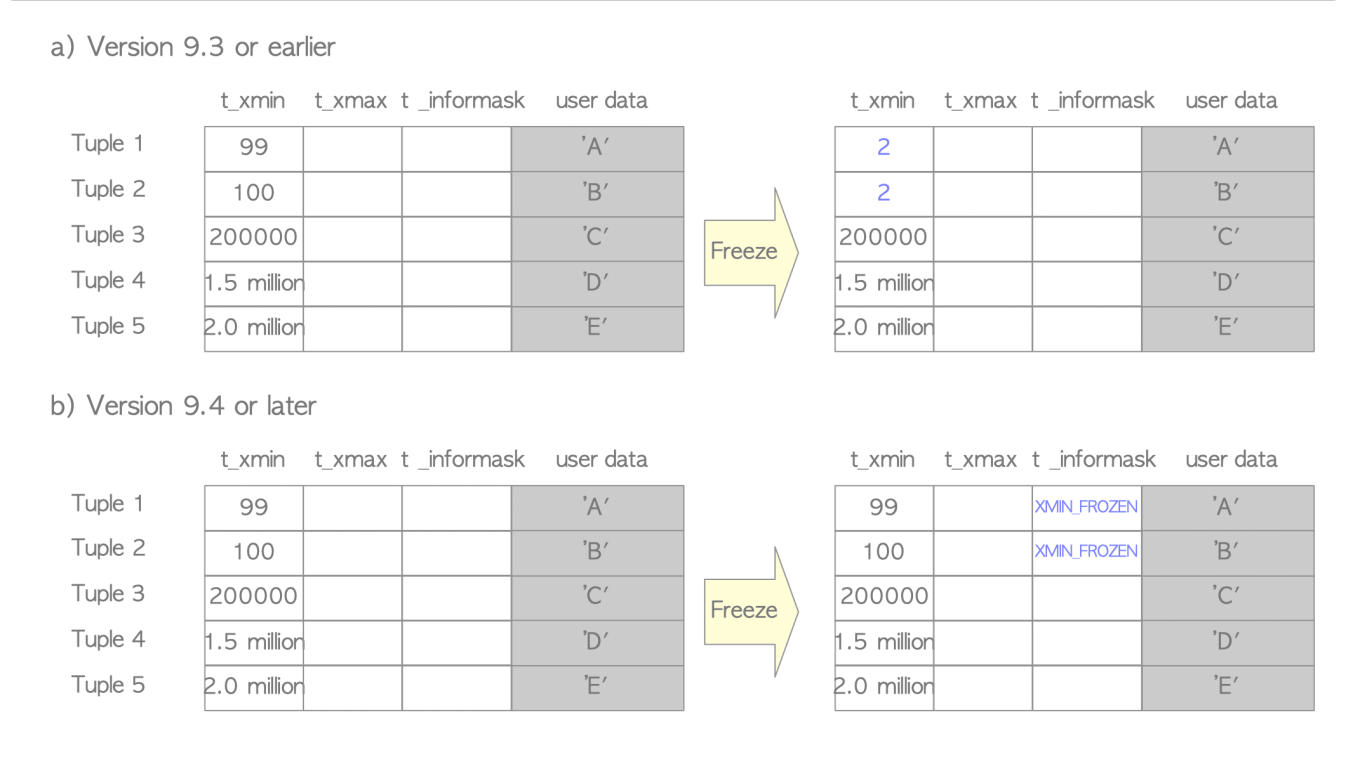
In PostgreSQL, a frozen txid, which is a special reserved txid 2, is defined such that it is always older than all other txids. In other words, the frozen txid is always inactive and visible.

The freeze process is invoked by the vacuum process. The freeze process scans all table files and rewrites the t_xmin of tuples to the frozen txid(2) if the t_xmin value is older than the current txid minus the vacuum_freeze_min_age (the default is 50 million). This is explained in more detail in Chapter 6.

For example, as can be seen in Fig. 5.21 a), the current txid is 50 million and the freeze process is invoked by the VACUUM command. In this case, the t_xmin of both Tuple_1 and Tuple_2 are rewritten to 2.

In versions 9.4 or later, the XMIN_FROZEN bit is set to the t_infomask field of tuples rather than rewriting the t_xmin of tuples to the frozen txid (Fig. 5.21 b).

Fig. 5.21. Freeze process.



References

[1] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan, "Database System Concepts", McGraw-Hill Education, ISBN-13: 978-0073523323

[2] Dan R. K. Ports, and Kevin Grittner, "Serializable Snapshot Isolation in PostgreSQL", VDBL 2012

[3] Thomas M. Connolly, and Carolyn E. Begg, "Database Systems", Pearson, ISBN-13: 978-0321523068