

| Business Template  **Clusters** |
| --- |
|  |

### 2.1 TASK 1 – CREATE NEW DATABASE

1. Connect to the **postgres** database and create a new one named **“test\_db”**.
2. Run the following query and investigate the result:

SELECT d.oid,

d.datname,

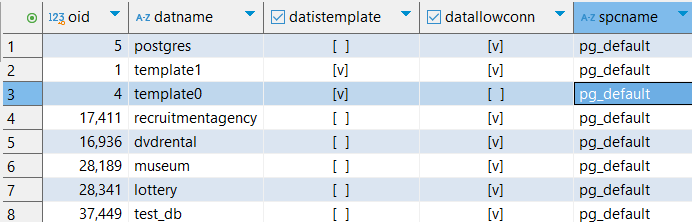
d.datistemplate,

d.datallowconn,

t.spcname

FROM pg\_database d

JOIN pg\_tablespace t ON t.oid = d.dattablespace;



-- Check existing databases and their tablespaces

**SELECT** *d*.oid, *d*.datname, *d*.datistemplate, *d*.datallowconn, *t*.spcname

**FROM** pg\_database *d*

**JOIN** pg\_tablespace *t* **ON** *t*.oid = *d*.dattablespace;

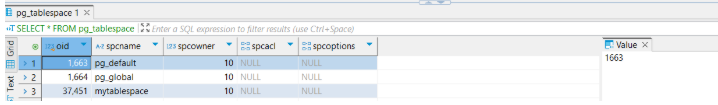
As we examine the results, we can see a cluster of databases, all created under PostgreSQL’s default tablespace, pg\_default.

One important observation is that it's not recommended to modify template1, as it serves as the default template for creating new databases. Any changes made to template1 will be inherited by all future

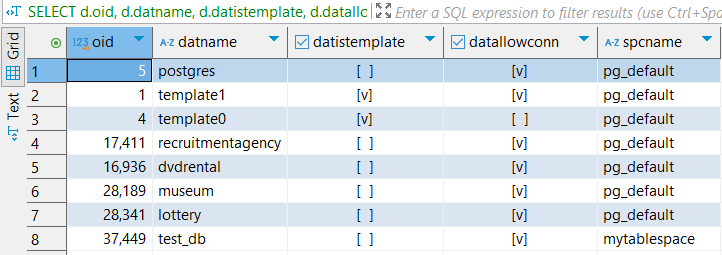
datname: The name of each database.  
datistemplate: Indicates if the database is a template (used to create others)-like the template0 and template .  
datallowconn: Shows if connections are allowed.  
spcname: The tablespace where the database is stored.  
oid: The internal object idnetifier of the database.

### 2.2 TASK 2 – CREATE NEW TABLESPACE

1. Create a new tablespace **“mytablespace”** with location:  
    **[where postgresql located]/data/tblspc\_test/**
2. Check that your tablespace exists in the pg\_tablespace table by running:  
   SELECT \* FROM pg\_tablespace;



1. Move test\_db into the new tablespace:  
   ALTER DATABASE test\_db SET TABLESPACE mytablespace;
2. Run the query from **Task 2.1, step 2** again and check your database.



1. Verify the directory where the tablespace is located.

After running the query, I verified that the spcname (tablespace) for the database test\_db had changed, confirming that the operation was successful.

Since multiple connections to the database were open, I had to first identify and terminate those connections in order to proceed with moving the associated files. Additionally, we needed to locate the appropriate directory to create the new tablespace.

We used a query to list all open connections, and also ensured that the newly created folder had the proper security permissions assigned to it.

-- Step 1: Show the PostgreSQL data directory path

**SHOW** data\_directory;

-- Step 2: Create the tablespace directory manually at the shown path, e.g.,

-- C:/Program Files/PostgreSQL/17/data/tblspc\_test

-- Step 3: Create a new tablespace using that path

**CREATE** **TABLESPACE** mytablespace

**LOCATION** 'C:/Program Files/PostgreSQL/17/data/tblspc\_test';

-- Step 4: Confirm that the new tablespace was created

**SELECT** \*

**FROM** pg\_tablespace;

-- Step 5: Identify active connections to 'test\_db'

**SELECT** pid, usename, application\_name, client\_addr, state

**FROM** pg\_stat\_activity

**WHERE** datname = 'test\_db';

-- Step 6: Terminate all other sessions connected to 'test\_db'

**SELECT** **pg\_terminate\_backend**(pid)

**FROM** pg\_stat\_activity

**WHERE** datname = 'test\_db'

**AND** pid <> **pg\_backend\_pid**();

-- Step 7: Move 'test\_db' into the new tablespace (must be connected to a different DB)

**ALTER** **DATABASE** test\_db

**SET** **TABLESPACE** mytablespace;

### 2.3 TASK 3 – CREATE NEW SCHEMA

1. Connect to the **test\_db** database and create a new schema named **“labs”**.
2. In the new schema, create a table named **“person”**:

CREATE TABLE labs.person (

id INTEGER NOT NULL,

name VARCHAR(15)

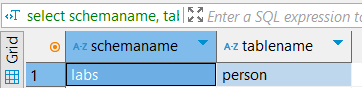
);

1. Check that the table was created:

SELECT schemaname, tablename

FROM pg\_tables

WHERE tablename = 'person';



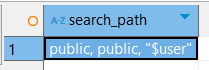
1. Insert values into the **person** table (correct queries if needed):

INSERT INTO person VALUES (1, 'Bob');

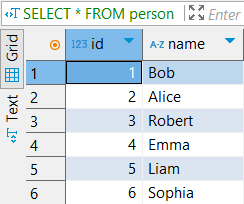
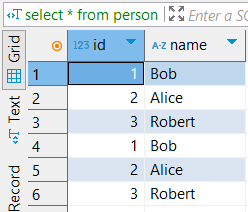
INSERT INTO person VALUES (2, 'Alice');

INSERT INTO person VALUES (3, 'Robert');

1. Use the following commands to **show and set the search\_path** so that inserts can be done **without specifying the schema**:  
   SHOW search\_path;



SET search\_path to perform INSERTS from previous task without any correction



-- Step 1: Create schema

**CREATE** **SCHEMA** **IF** **NOT** **EXISTS** labs;

-- Step 2: Create table within the 'labs' schema

**CREATE** **TABLE** labs.person (

id **INTEGER** **NOT** **NULL**,

name **VARCHAR**(15)

);

-- Step 3: Verify that the table was created

**SELECT** schemaname, tablename

**FROM** pg\_tables

**WHERE** tablename = 'person';

-- Step 4: Insert values using explicit schema reference

**INSERT** **INTO** labs.person **VALUES** (1, 'Bob');

**INSERT** **INTO** labs.person **VALUES** (2, 'Alice');

**INSERT** **INTO** labs.person **VALUES** (3, 'Robert');

-- Step 5: Check current search\_path

**SHOW** search\_path;

-- Step 6: Set search\_path to 'labs' so schema name is no longer needed

**SET** search\_path **TO** labs;

-- Step 7: Insert values without specifying schema

**INSERT** **INTO** person **VALUES** (1, 'Bob');

**INSERT** **INTO** person **VALUES** (2, 'Alice');

**INSERT** **INTO** person **VALUES** (3, 'Robert');

-- Step 8: View all values in the table

**SELECT** \* **FROM** person;

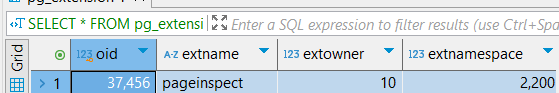
### 3. TRANSACTIONS AND VACUUMING

### 3.1 TASK 4 – INVESTIGATE MVCC

*(Make sure the required extension is installed before starting.)*

**Step 1: Install the required extension:**

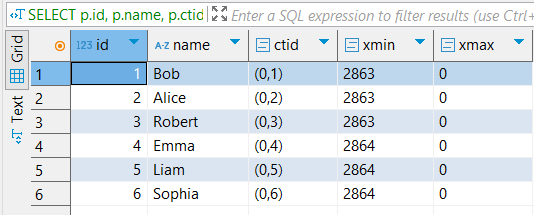
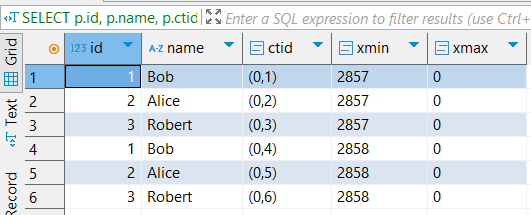
CREATE EXTENSION pageinspect;



**Step 2: Run the following query to view MVCC-related metadata:**

SELECT p.id, p.name, p.ctid, p.xmin, p.xmax

FROM person p;



SELECT

t\_xmin,

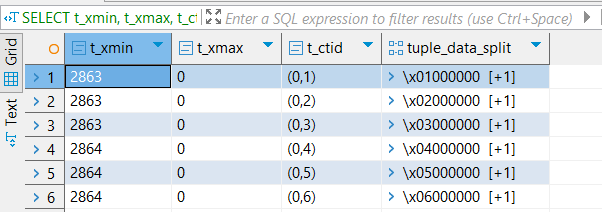
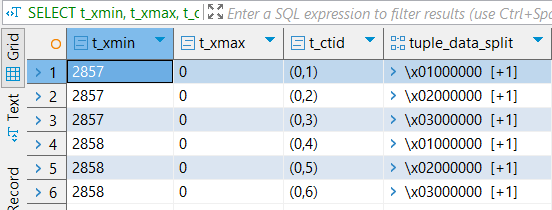
t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::regclass, t\_data, t\_infomask, t\_infomask2, t\_bits)

FROM

heap\_page\_items(get\_raw\_page('labs.person', 0));



**Step 4: Investigate changes in xmin and xmax by executing the following operations in separate transactions:**

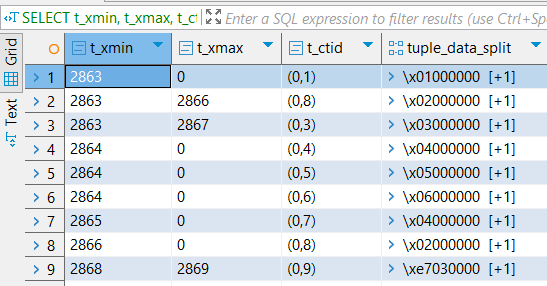
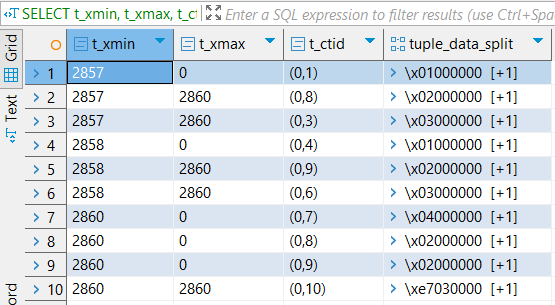
INSERT INTO person VALUES (4, 'John');

UPDATE person SET name = 'Alex' WHERE id = 2;

DELETE FROM person WHERE id = 3;

INSERT INTO person VALUES (999, 'Test');

DELETE FROM person WHERE id = 999;



During these two tasks, I inserted different values into the table for inspection. Initially, as requested, I inserted the same values twice—once using the schema.table format and once using only the table name format after setting the search path to the target schema.

In the second task, I followed a similar process. However, in the second group of insertions, I inserted entirely different rows. When I ran the tuple queries, I made the following observations:

1. **Insert Behavior**:  
    Each time I added new rows, the t\_xmin value updated for every new transaction, even after dropping and recreating the table. Since I maintained the same number of transactions, the overall structure remained largely unchanged.
2. **Transaction Count Discrepancy**:  
    In the second tuple query—after performing a combination of deletes, updates, and inserts—I noticed a difference in transaction counts: 9 transactions appeared in the second table versus 10 in the first. This discrepancy seems to stem from how deletions and updates were handled.
3. **Delete Operation**:  
    For the delete operation (id = 3), only the t\_xmax was updated—changing from 0 to 2867 in the second table. In contrast, in the first table (which had duplicate entries), this update occurred in two rows.
4. **Update Operation**:  
    Similar to the delete, updates also showed differences. In the second task, performing updates one by one created separate transactions for each operation. This resulted in consecutive transaction numbers for each insert, update, or delete. Meanwhile, the first task, executed in a single run, grouped all operations under the same transaction number.
5. **Row Versions**:  
    For updates, a new row is created with a fresh t\_xmin, and the original row’s t\_xmax is updated to match this new t\_xmin. This versioning explains why there's only a one-row difference between the two tables—due to the duplicate updates involving id = 3 in the first task.
6. **Insert Operation**:  
    Inserts simply generate new rows with a unique t\_xmin and a t\_xmax set to 0 in both cases.

-- Step 1: Install the pageinspect extension (if not already installed)

**CREATE** **EXTENSION** **IF** **NOT** **EXISTS** pageinspect;

-- Verify that the extension is installed

**SELECT** \*

**FROM** pg\_extension

**WHERE** extname = 'pageinspect';

-- Step 2: Run a basic MVCC query on the 'person' table

**SELECT**

*p*.id,

*p*.**name**,

*p*.ctid,

*p*.xmin,

*p*.xmax

**FROM** person p;

-- Step 3: Find current database and schema information for pageinspect

**SELECT** **current\_database**();

**SELECT**

extname,

extnamespace::**regnamespace**

**FROM** pg\_extension;

-- Step 4: Set the search path to include the correct schema for pageinspect

**SET** search\_path **TO** public, labs, **"$user"**, public;

-- Step 5: Analyze tuple-level details using pageinspect

**SELECT**

t\_xmin,

t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::**regclass**, t\_data, t\_infomask, t\_infomask2, t\_bits)

**FROM** heap\_page\_items(

get\_raw\_page('labs.person', 0)

);

-- Step 6: Perform DML operations on the 'person' table

**INSERT** **INTO** person **VALUES** (4, 'John');

**UPDATE** person

**SET** **name** = 'Alex'

**WHERE** id = 2;

**DELETE** **FROM** person

**WHERE** id = 3;

**INSERT** **INTO** person **VALUES** (999, 'Test');

**DELETE** **FROM** person

**WHERE** id = 999;

-- Step 7: Inspect the tuples again after DML operations

**SELECT**

t\_xmin,

t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::**regclass**, t\_data, t\_infomask, t\_infomask2, t\_bits)

**FROM** heap\_page\_items(

get\_raw\_page('labs.person', 0)

);

### 3.2 TASK 5 – INVESTIGATE VACUUM

Use the following query to check what happens during vacuum operations:

SELECT

t\_xmin,

t\_xmax,

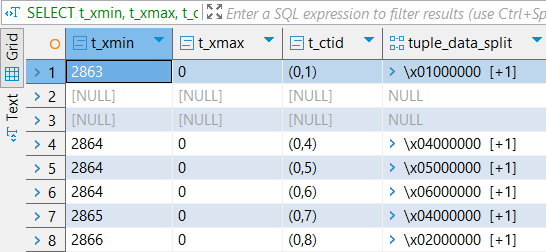
t\_ctid,

tuple\_data\_split('labs.person'::regclass, t\_data, t\_infomask, t\_infomask2, t\_bits)

FROM

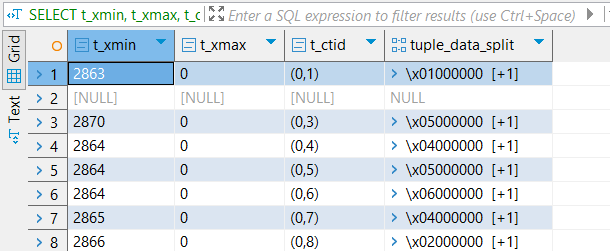
heap\_page\_items(get\_raw\_page('labs.person', 0));

#### **Step-by-Step:**

1. **Run:  
   **

VACUUM labs.person;

* Check and record the results using the query above.

1. **Run:  
   **

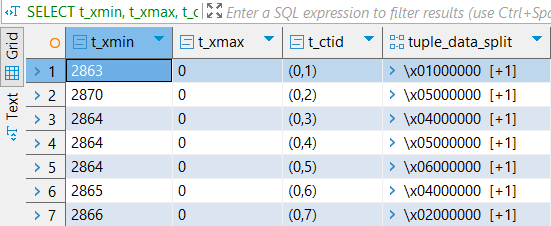
INSERT INTO person VALUES (5, 'Sarah');

* Check the results again using the same query.

1. **Run:**

VACUUM FULL labs.person;

* Check the final state using the query again and compare changes.



The main idea is that the vacuum usually tries to cleans the rows that have been deleted but are still present on the disk. The first time we do the cleaning is a lightweight cleaning so in the first table we can still see the rows where the delete transactions have been recorded but now they are null, which tells the system that this is a reusable row. Therefore next when we do the insert, the insert will simply be done in one of those rows with a new t\_xmin as transaction number. Finally when we do vacuum all that left null row is completely physically removed from the disk.

-- Step 1: Run a regular VACUUM and inspect tuple-level details

**VACUUM** labs.person;

**SELECT**

t\_xmin,

t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::**regclass**, t\_data, t\_infomask, t\_infomask2, t\_bits)

**FROM** heap\_page\_items(

get\_raw\_page('labs.person', 0)

);

-- Step 2: Insert a new row and inspect the changes at the tuple level

**INSERT** **INTO** person **VALUES** (5, 'Sarah');

**SELECT**

t\_xmin,

t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::**regclass**, t\_data, t\_infomask, t\_infomask2, t\_bits)

**FROM** heap\_page\_items(

get\_raw\_page('labs.person', 0)

);

-- Step 3: Run VACUUM FULL and inspect the final state of the heap page

**VACUUM** **FULL** labs.person;

**SELECT**

t\_xmin,

t\_xmax,

t\_ctid,

tuple\_data\_split('labs.person'::**regclass**, t\_data, t\_infomask, t\_infomask2, t\_bits)

**FROM** heap\_page\_items(

get\_raw\_page('labs.person', 0)

);