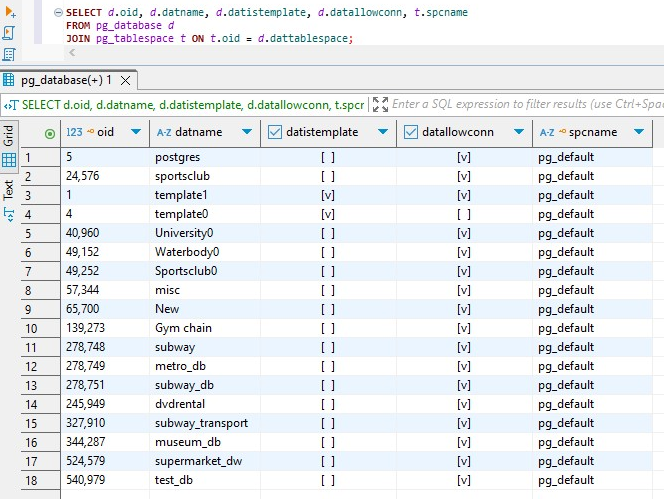
# 2. UNDERSTANDING DATABASE AND TABLESPACES

## 2.1 TASK 1 CREATE NEW DATABASE

1. Connect to the postgres Database and create new one named “test\_db”.

**CREATE** database test\_db;

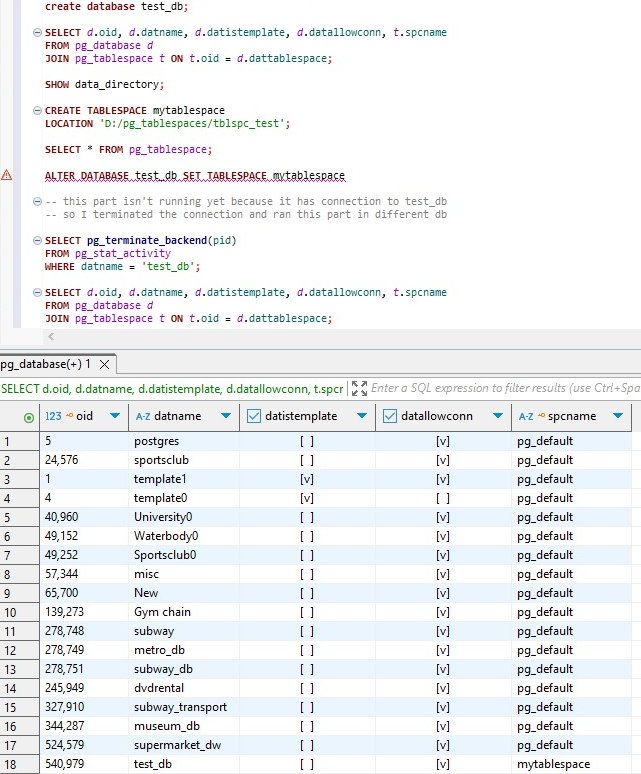
### 2. Run query, investigate result:



This query showed **all 18 databases** currently present in your PostgreSQL instance and that all of them are using the **default tablespace** (pg\_default), including the newly created one: test\_db. The [v] in datallowconn shows all these databases are **open for connections**. Some databases (like template1, template0, University0, etc.) have [v] in datistemplate, meaning they can be used as a **source template** when creating new databases.

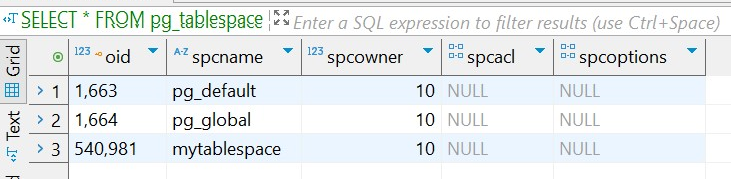
2.2 TASK 2 CREATE NEW TABLESPACE

### 1. Create new tablespace “mytablespace” with location “[where postgresql located]/data/tblspc\_test/”.



2. Check your tablespace exists in pg\_tablespace table:

**SELECT** \* **FROM** pg\_tablespace;



It exists

### 3. Move test\_db into new tablespace, Run again 2.1.2 query and check your database. Check your directory where tablespace is located.

**ALTER** **DATABASE** test\_db **SET** **TABLESPACE** mytablespace

-- this part isn't running yet because it has connection to test\_db

-- so I terminated the connection and ran this part in different db

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

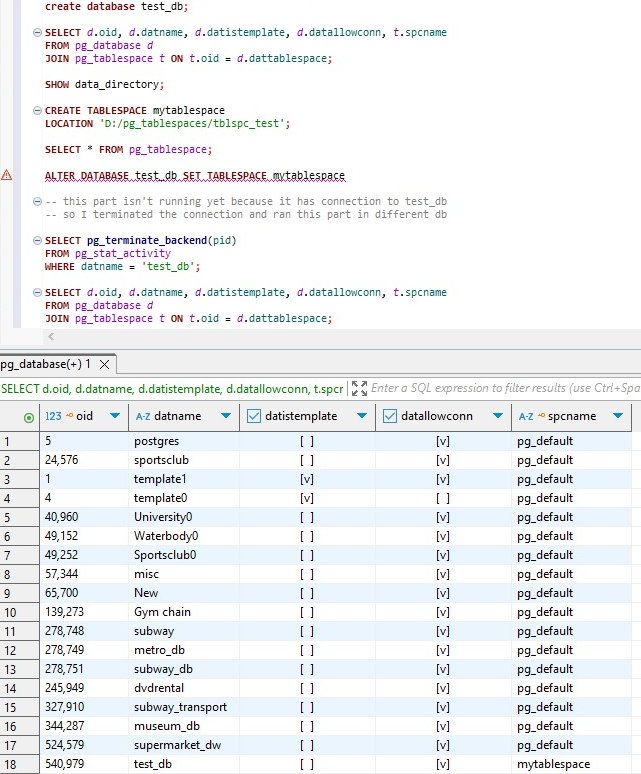
**FROM** pg\_stat\_activity

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

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

**FROM** pg\_database *d*

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



## 2.3 TASK 3 CREATE NEW SCHEMA

### 1. Connect to test\_db and create new schema named “labs”.

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

### 2. In new schema create table named “person”:

**CREATE** **TABLE** **IF** **NOT** **EXISTS** labs.person (

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

name **VARCHAR**(15)

);

### 3. Insert into person table values (correct queries if needed):

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

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

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

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

### 4. Use SHOW search\_path and SET search\_path to perform INSERTS from previous task without any correction.

Already done

# 3. TRANSACTION AND VACUUMING

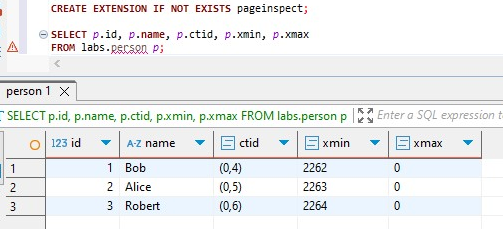
## 3.1 TASK 4 INVESTIGATE MVCC\*

1) you need to install extension before: CREATE EXTENSION pageinspect;

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

### 2) Use queries:

### 1)



This shows each row's:

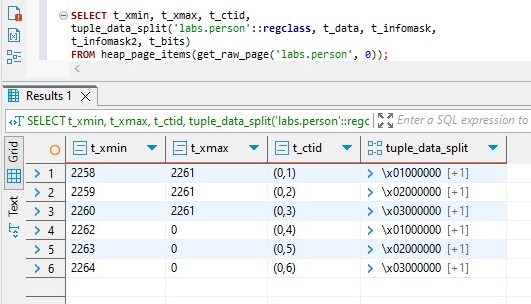
1. ctid: physical tuple location (like (page, row))
2. xmin: transaction that created the row
3. xmax: transaction that deleted the row

**At this point**, all visible rows are active, meaning:

xmax = 0 (not deleted), xmin reflects the transaction ID that inserted them (2262, 2263, 2264)

At first I inserted rows by calling the schema and table and I deleted it with 2261st transaction.

### 2)



### 3) And investigate what is happening with xmin and xmax while performing following in different transactions:

1)After running

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

PostgreSQL **appends** a new tuple to the labs.person heap file.

The new row has: fresh ctid, xmin = 2265, xmax = 0. Now the row is **visible to all future transactions.**

2) After running

**UPDATE labs.person SET name = 'Alex' WHERE id = 2;**

**UPDATE is implemented as DELETE + INSERT**. Old version (id=2, name='Alice') is marked with:

xmin = 2263 and xmax = 2266 as the previous row was deleted by 2266th transaction.

New version (id=2, name='Alex') is created with: new ctid (e.g., (0,6)) xmin = 2266, xmax = 0

3) After running

**DELETE FROM labs.person WHERE id = 3;**

In this process third row where xmin = 2264 is deleted to xmax is updated to 2267 and we have new live row.

4) After running

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

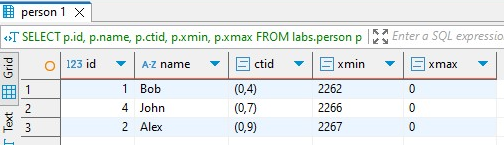
**DELETE FROM labs.person WHERE id = 999;**

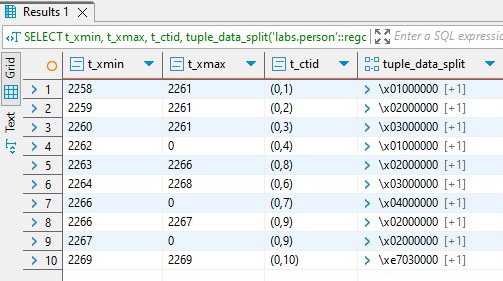
PostgreSQL creates a tuple with:

xmin = 2269 xmax = 2269.

The row is **created and deleted in the same transaction**. As a result, this tuple **is never visible** to any other transaction — not even the one that reads committed data after commit.

After running all of them we get this result:





To conclude all these steps, In PostgreSQL, every row has internal metadata: xmin indicates the transaction that inserted the row, xmax indicates the transaction that deleted it, and ctid shows the physical location of the tuple in the heap. When a row is inserted, PostgreSQL appends it to the heap with a new ctid, sets xmin to the inserting transaction ID, and xmax to 0, meaning the row is visible and alive.

When a row is updated, PostgreSQL performs a logical delete followed by an insert: the old version is marked with xmax as the updating transaction, and a new version is inserted with a new ctid and xmin equal to the same transaction ID.

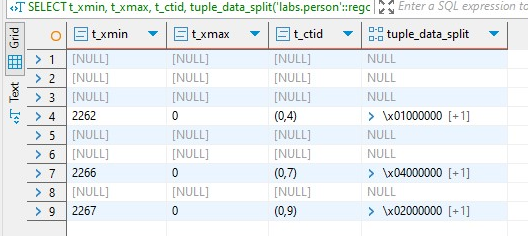
When a row is deleted, PostgreSQL does not remove it physically but marks it with xmax equal to the transaction ID, making it invisible to new transactions.

If a row is inserted and then deleted in the same transaction, it is marked with xmin and xmax equal to that transaction ID, and the row is never visible to any transaction, not even to the one that created it after commit.

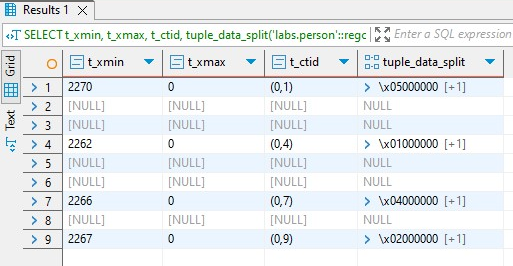
Overall, PostgreSQL retains multiple versions of rows to support concurrent transactions and isolation, leaving old versions in the heap until cleaned up by VACUUM.

# 3.2 TASK 5 INVESTIGATE VACUUM

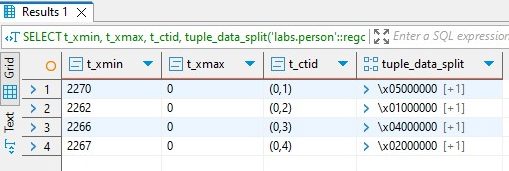
## 1. Run: vacuum labs.person; Check results.

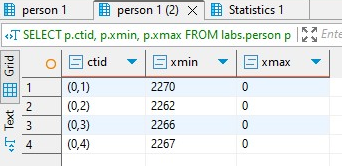


PostgreSQL **marks dead tuples as reusable**, but **does not physically remove them** or **shrink the table file**. It only **updates visibility maps and frees up dead space for reuse** in future inserts. Because of this, dead rows still appear in heap\_page\_items(...) inspection. PostgreSQL can now **reuse** space occupied by dead tuples in future writes.

2. Run: INSERT INTO person VALUES(5, ‘Sarah’); Check results.

PostgreSQL checks the free space from the **visibility map**. If there is a slot where a dead tuple (like id = 3 or 999) was stored, it will **reuse that same physical location** (ctid) for the new row. Because of it new row gets the ctid of a previously deleted one.

3. Run: vacuum full labs.person; Check results.



**VACUUM FULL** performs a full table **rewrite**, copies all live rows to **a new physical file**, deletes the old file with dead/obsolete rows. CTIDs of all rows are changed (because their physical positions change) and disk space is actually **reclaimed** (file size shrinks)After all, all dead tuples are gone. Inspection with heap\_page\_items shows **only four live rows**.

In conclusion running the basic VACUUM command marks dead tuples as reusable and updates visibility maps but does not physically remove or shrink the table, dead tuples still remain in the heap and can be seen in diagnostic queries. When a new row is inserted after a basic VACUUM, PostgreSQL may reuse the space left by a deleted tuple, resulting in the new row inheriting the ctid of the previously deleted one.

Running VACUUM FULL rewrites the entire table by copying live rows to a new physical file, discarding dead tuples entirely, and reclaiming disk space; as a result, all ctid values change and dead rows are no longer visible in the heap, so regular VACUUM helps manage space and maintain performance by marking dead rows for reuse, while VACUUM FULL physically compacts the table and completely removes all dead data.