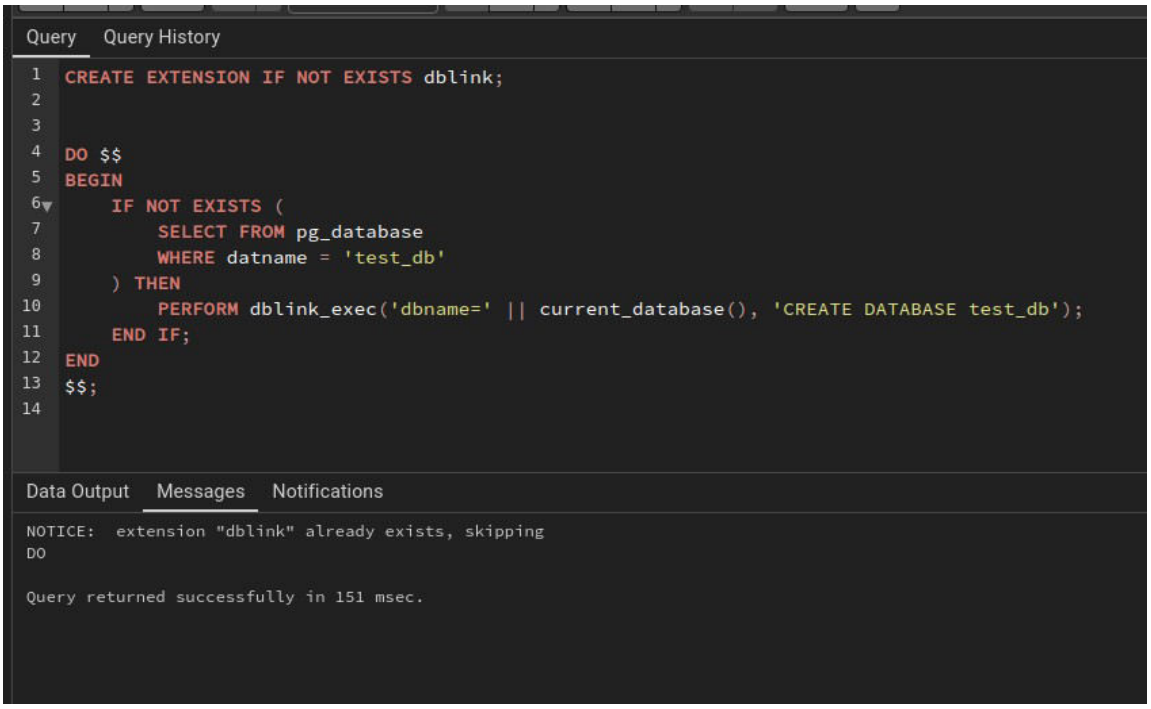
PG_DWH TASK 1

1.1 TASK 1 CREATE NEW DATABASE

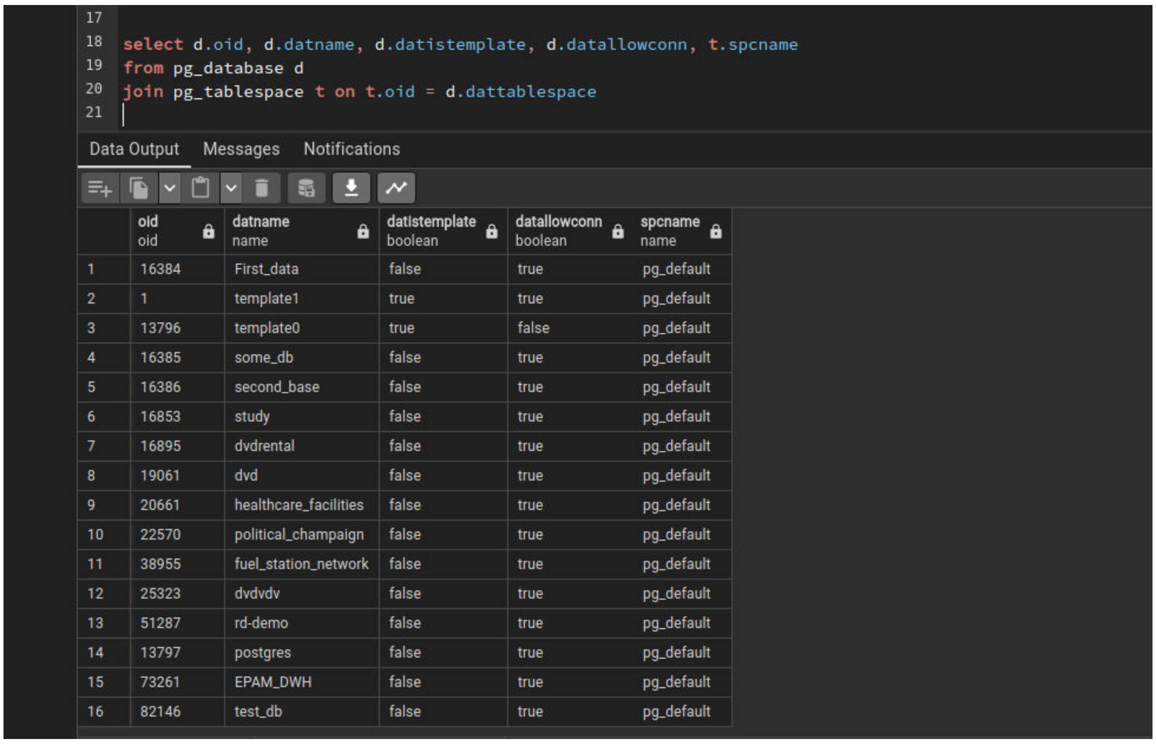


```
CREATE EXTENSION IF NOT EXISTS dblink;


DO $$
BEGIN
    IF NOT EXISTS (
        SELECT FROM pg_database
        WHERE datname = 'test_db'
    ) THEN
        PERFORM dblink_exec('dbname=' || current_database(), 'CREATE DATABASE test_db');
    END IF;
END
$$;
```

```
NOTICE:  extension "dblink" already exists, skipping
DO

Query returned successfully in 151 msec.
```

Database created.

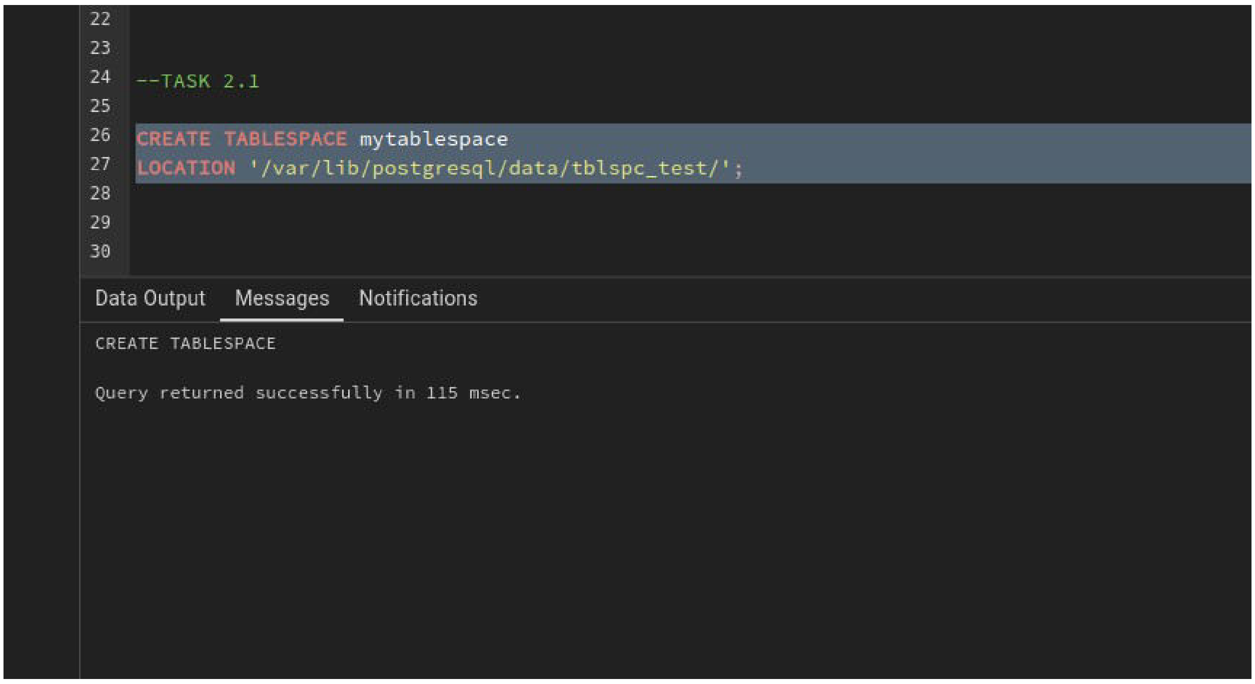
In the statement provided we are getting information about all databases in the system,oid is the unique identifier of each database ,datname column shows name of database,datistemplate column show's if that particular database is a template, not actual database, datallowconn column shows if connection allowed to this database,spcname shows where data will be stored(location in the comupter).



```
select d.oid, d.datname, d.datistemplate, d.datallowconn, t.spcname
from pg_database d
join pg_tablespace t on t.oid = d.dattablespace
```

	oid oid	datname name	datistemplate boolean	datallowconn boolean	spcname name
1	16384	First_data	false	true	pg_default
2	1	template1	true	true	pg_default
3	13796	template0	true	false	pg_default
4	16385	some_db	false	true	pg_default
5	16386	second_base	false	true	pg_default
6	16853	study	false	true	pg_default
7	16895	dvdrental	false	true	pg_default
8	19061	dvd	false	true	pg_default
9	20661	healthcare_facilities	false	true	pg_default
10	22570	political_champaign	false	true	pg_default
11	38955	fuel_station_network	false	true	pg_default
12	25323	dvdvdv	false	true	pg_default
13	51287	rd-demo	false	true	pg_default
14	13797	postgres	false	true	pg_default
15	73261	EPAM_DWH	false	true	pg_default
16	82146	test_db	false	true	pg_default

1.2 TASK 2 CREATE NEW TABLESPACE

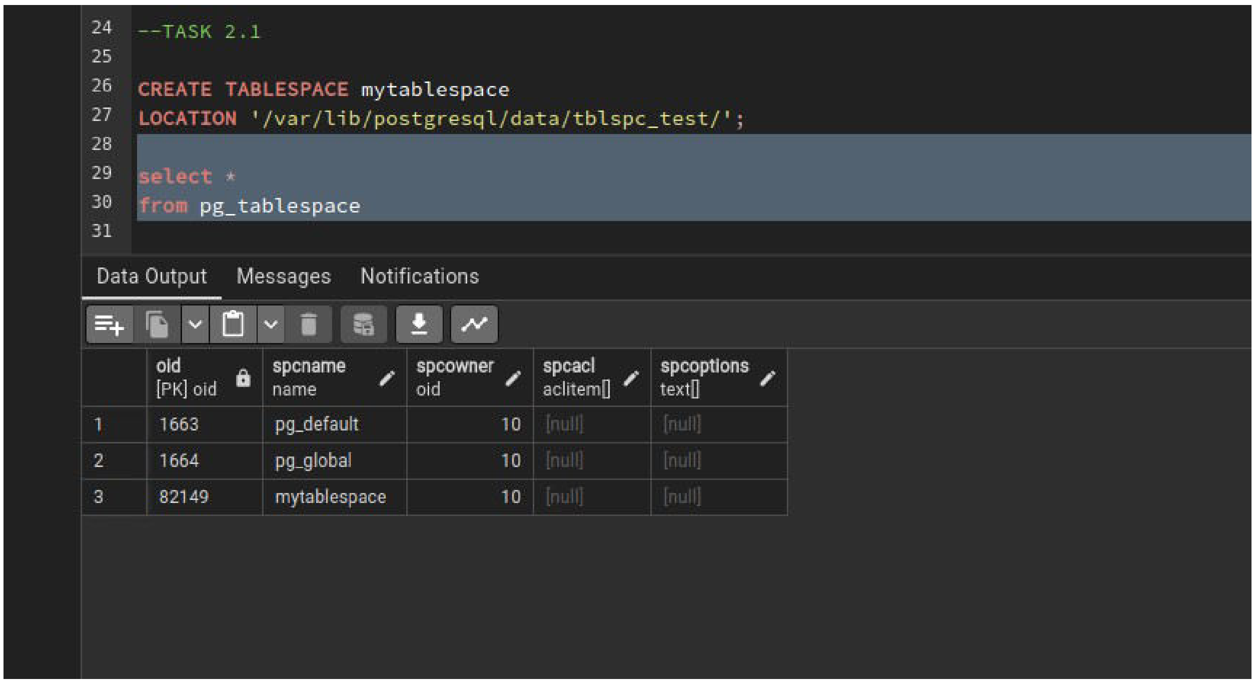


```
--TASK 2.1

CREATE TABLESPACE mytablespace
LOCATION '/var/lib/postgresql/data/tblspc_test/';
```

Data Output | Messages | Notifications

```
CREATE TABLESPACE

Query returned successfully in 115 msec.
```

TABLESPACE CREATED.

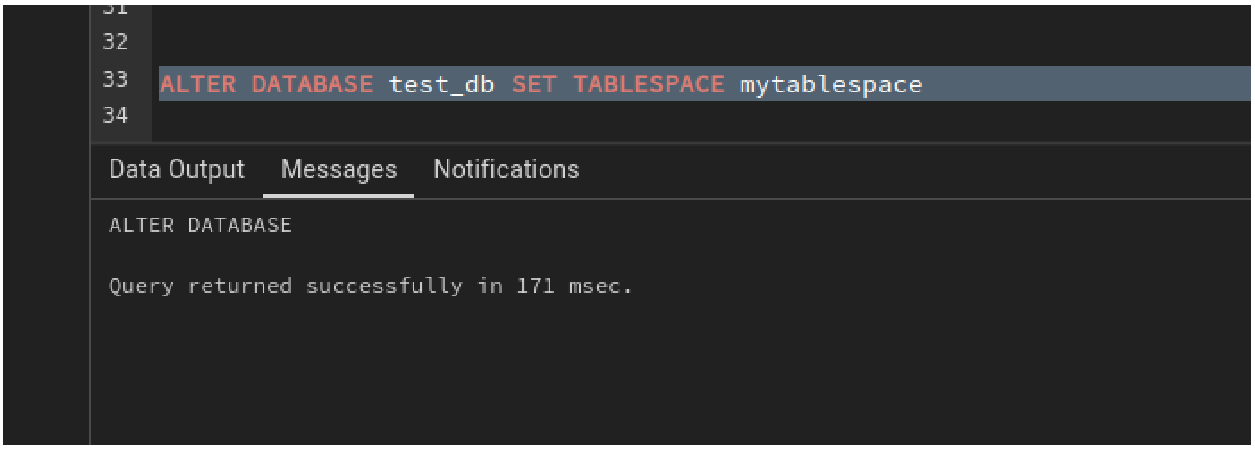


```
--TASK 2.1

CREATE TABLESPACE mytablespace
LOCATION '/var/lib/postgresql/data/tblspc_test/';

select *
from pg_tablespace
```

Data Output | Messages | Notifications

	oid [PK] oid	spcname name	spcowner oid	spcacl aclitem[]	spcoptions text[]
1	1663	pg_default	10	[null]	[null]
2	1664	pg_global	10	[null]	[null]
3	82149	mytablespace	10	[null]	[null]

Here you can see it's created successfully.

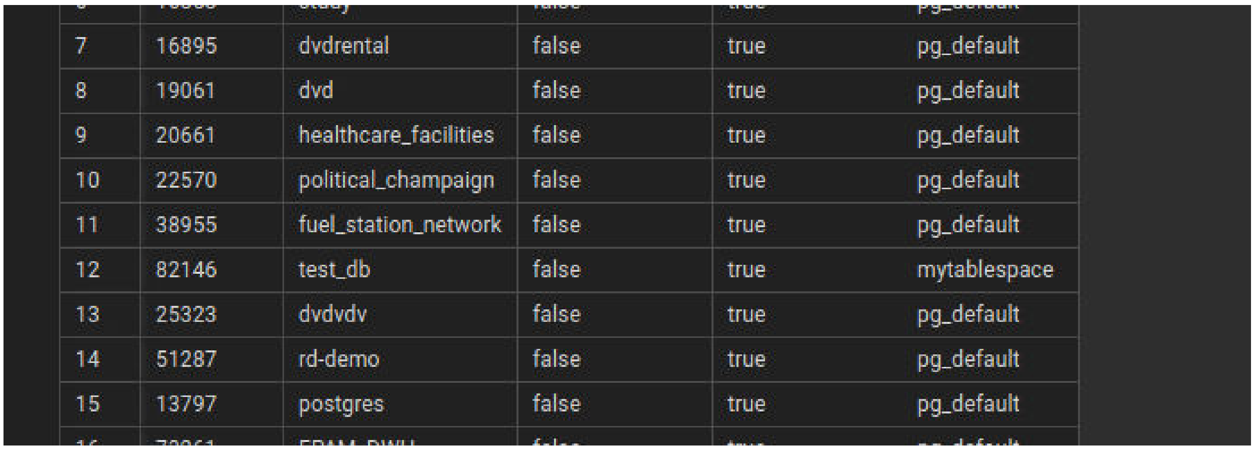


```
ALTER DATABASE test_db SET TABLESPACE mytablespace
```

Data Output | Messages | Notifications

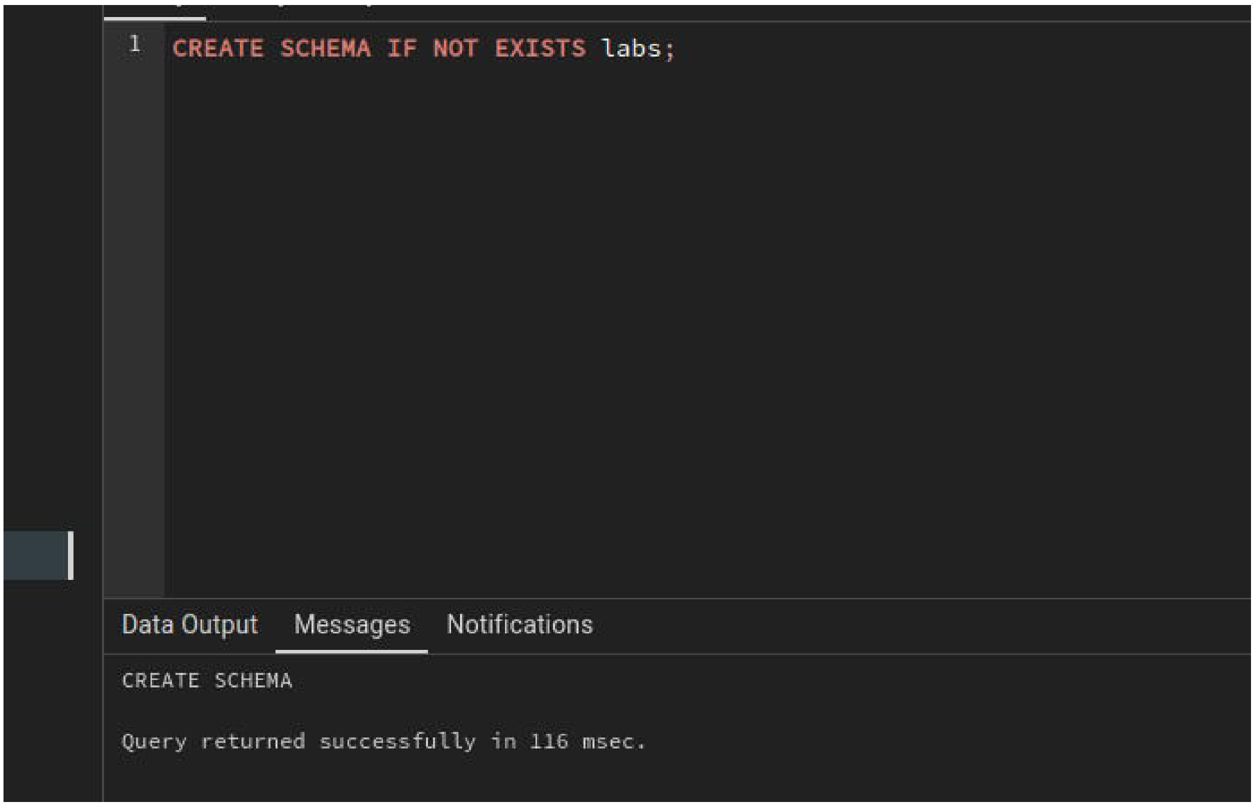
```
ALTER DATABASE

Query returned successfully in 171 msec.
```

Tablespace changed for our test_db database, after this query all data will be located inspecified location.



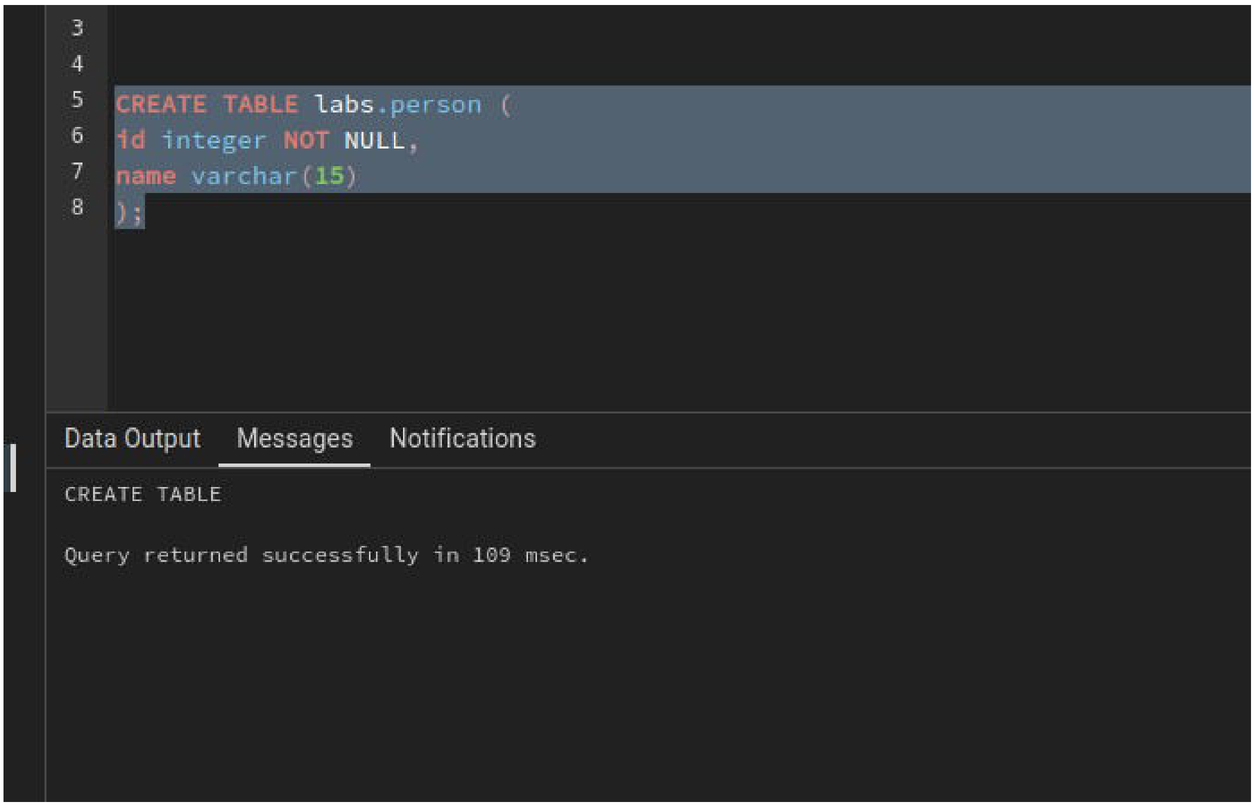
7	16895	dvdrental	false	true	pg_default
8	19061	dvd	false	true	pg_default
9	20661	healthcare_facilities	false	true	pg_default
10	22570	political_champaign	false	true	pg_default
11	38955	fuel_station_network	false	true	pg_default
12	82146	test_db	false	true	mytablespace
13	25323	dvdvdv	false	true	pg_default
14	51287	rd-demo	false	true	pg_default
15	13797	postgres	false	true	pg_default

1.3 TASK 3 CREATE NEW SCHEMA


```
CREATE SCHEMA IF NOT EXISTS labs;
```

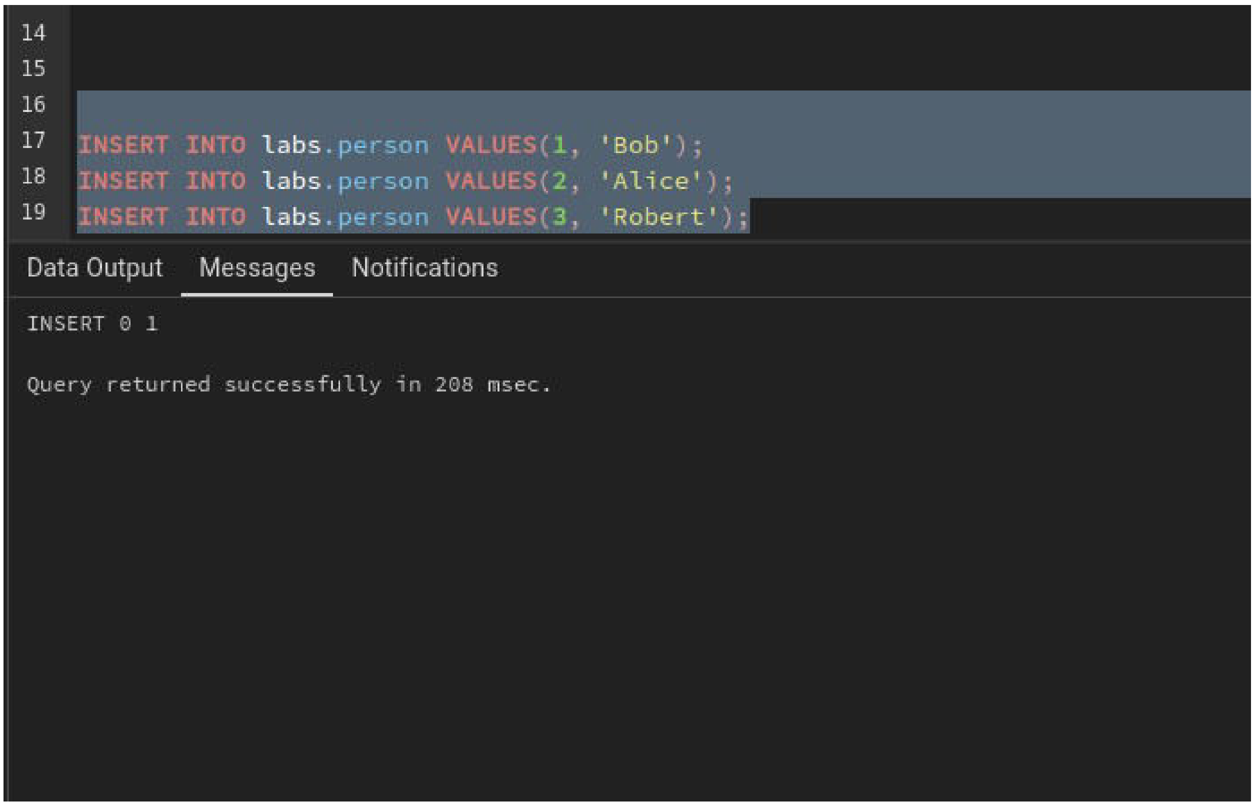
```
CREATE SCHEMA

Query returned successfully in 116 msec.
```

Schema created.


```
CREATE TABLE labs.person (
id integer NOT NULL,
name varchar(15)
);
```

```
CREATE TABLE

Query returned successfully in 109 msec.
```

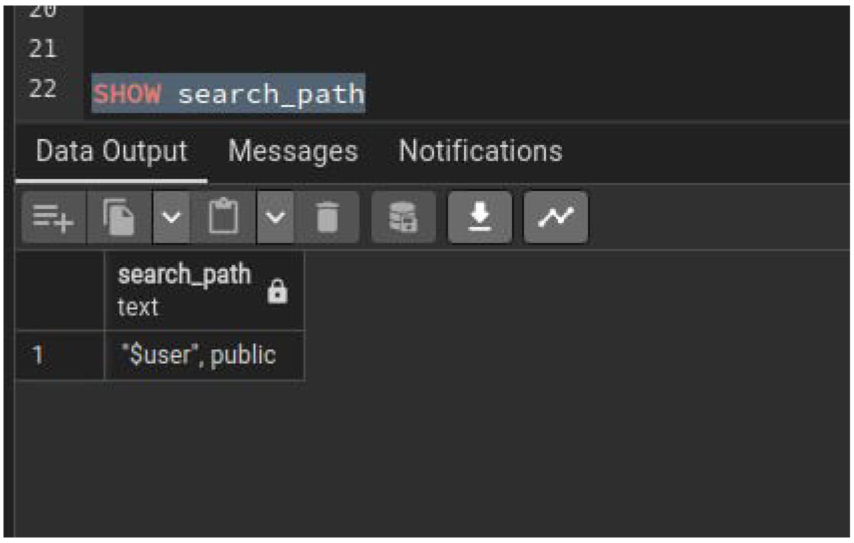
Table created.



```
INSERT INTO labs.person VALUES(1, 'Bob');
INSERT INTO labs.person VALUES(2, 'Alice');
INSERT INTO labs.person VALUES(3, 'Robert');
```

```
INSERT 0 1

Query returned successfully in 208 msec.
```

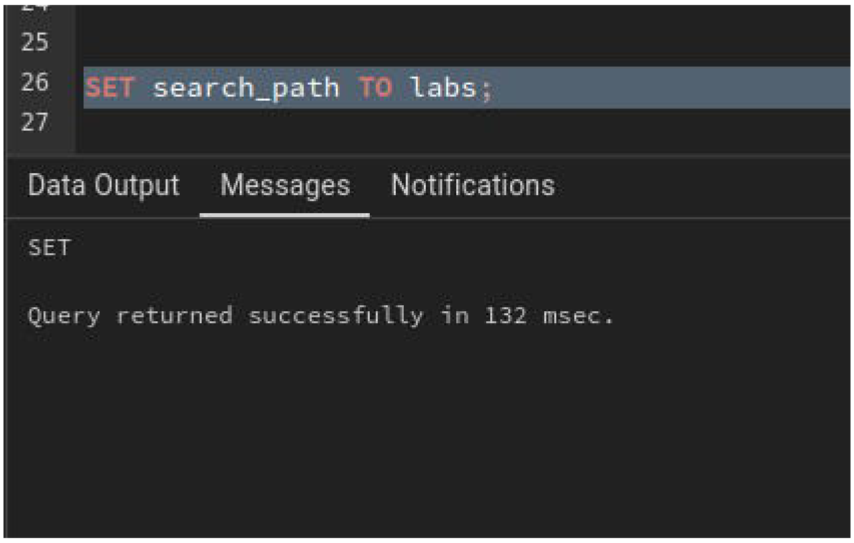
Insert query corrected and executed.



```
SHOW search_path
```

	search_path text
1	"$user", public

This query shows the default search schema name , in which postgres will search for tables if we didn’t specify schema name in the query.

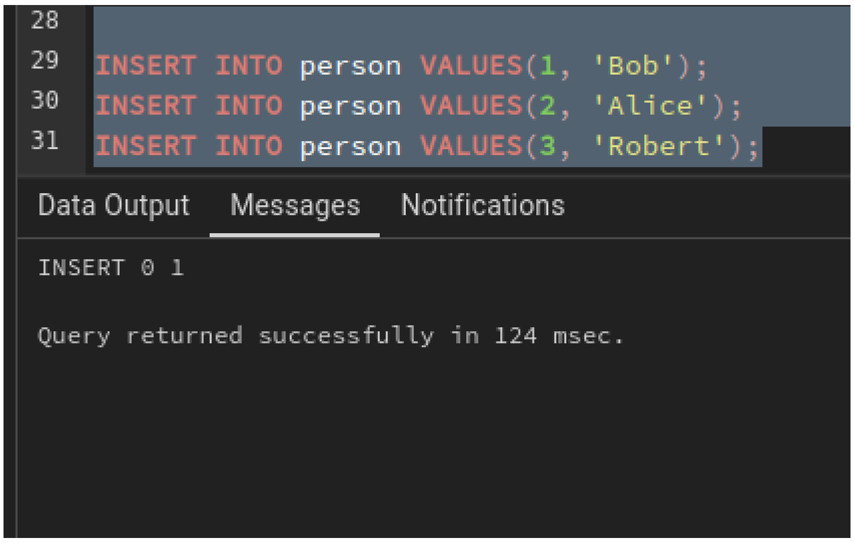


```
SET search_path TO labs;
```

Data Output Messages Notifications

```
SET

Query returned successfully in 132 msec.
```

Here we set the schema name to tell postgres that we are working in labs schema.



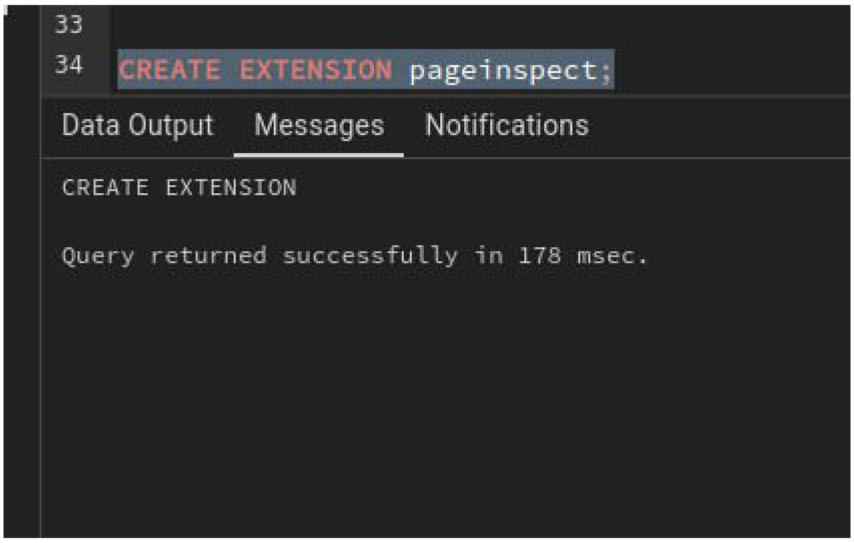
```
INSERT INTO person VALUES(1, 'Bob');
INSERT INTO person VALUES(2, 'Alice');
INSERT INTO person VALUES(3, 'Robert');
```

Data Output Messages Notifications

```
INSERT 0 1

Query returned successfully in 124 msec.
```

Now we see that it works without specifying schema name in INSERT query.

1.4 TASK 4 INVESTIGATE MVCC*

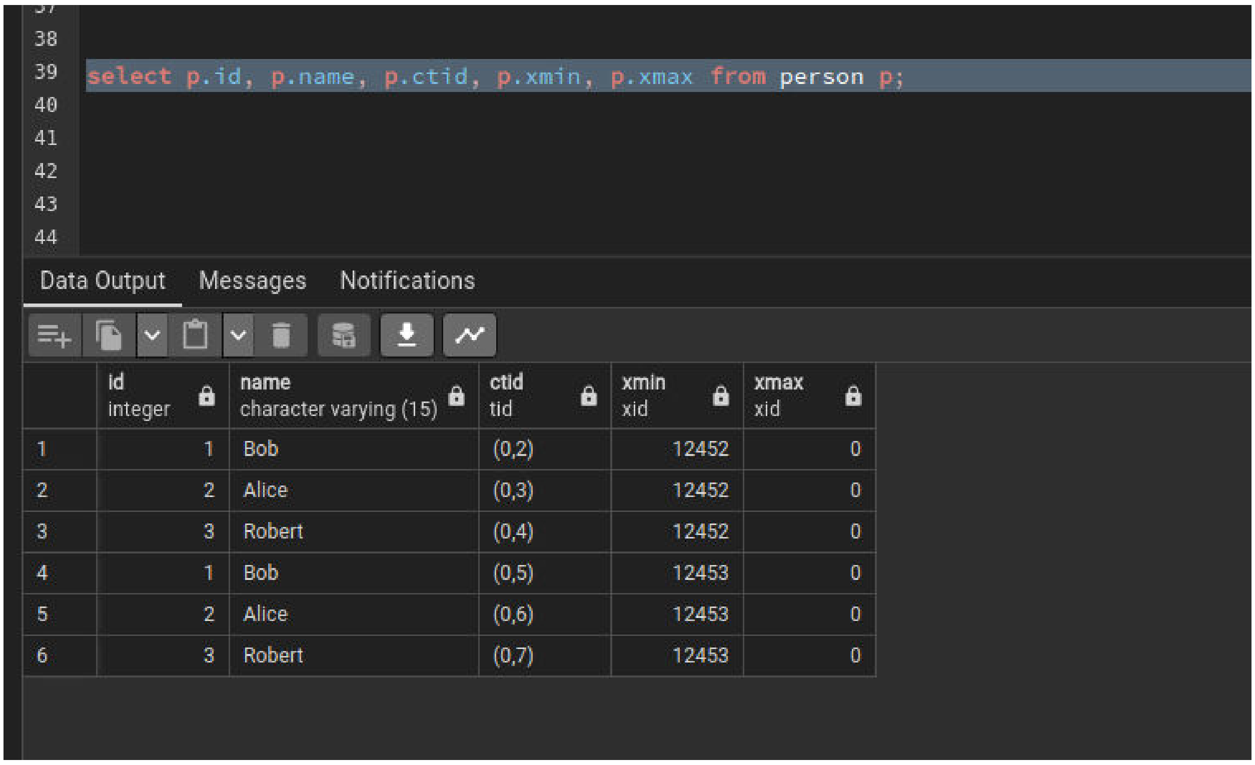


```
CREATE EXTENSION pageinspect;
```

Data Output Messages Notifications

```
CREATE EXTENSION

Query returned successfully in 178 msec.
```

Extension created.



```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;
```

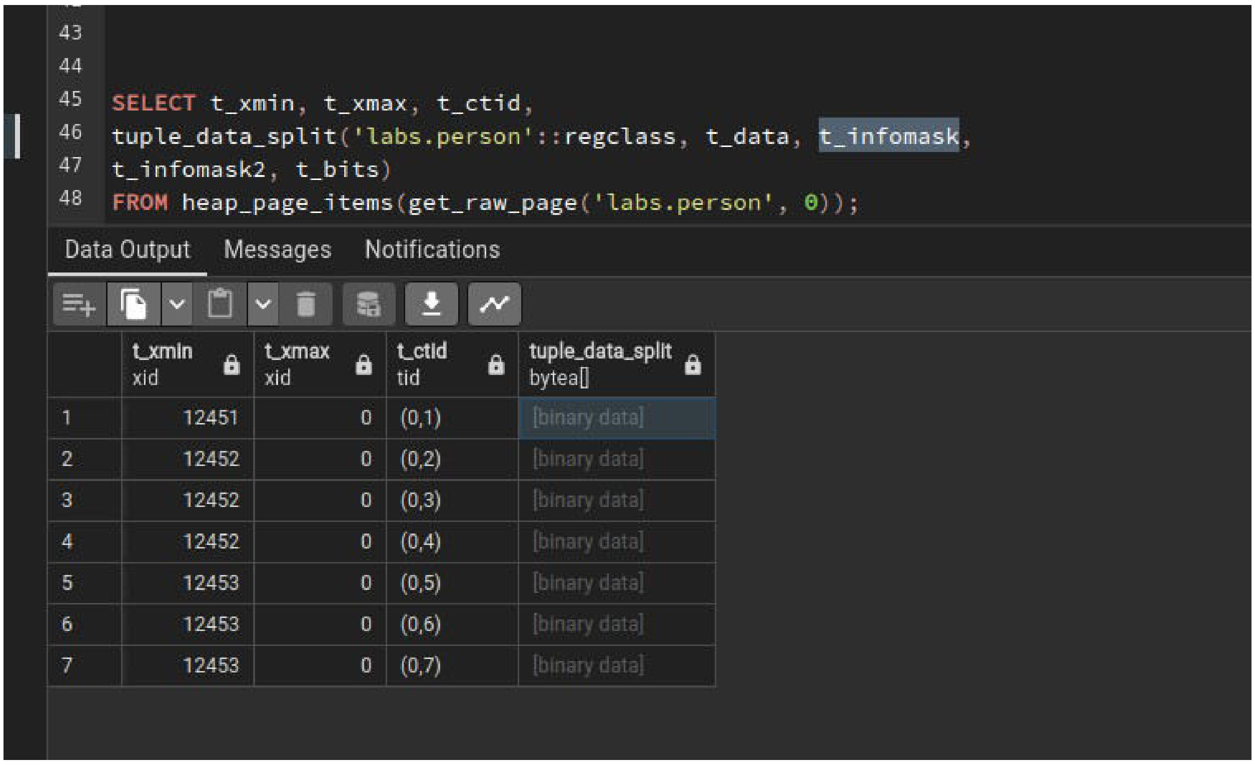
Data Output Messages Notifications

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	2	Alice	(0,3)	12452	0
3	3	Robert	(0,4)	12452	0
4	1	Bob	(0,5)	12453	0
5	2	Alice	(0,6)	12453	0
6	3	Robert	(0,7)	12453	0

This query shows following information

id show's unique identifier of the row, name is the name we set when creating this table,

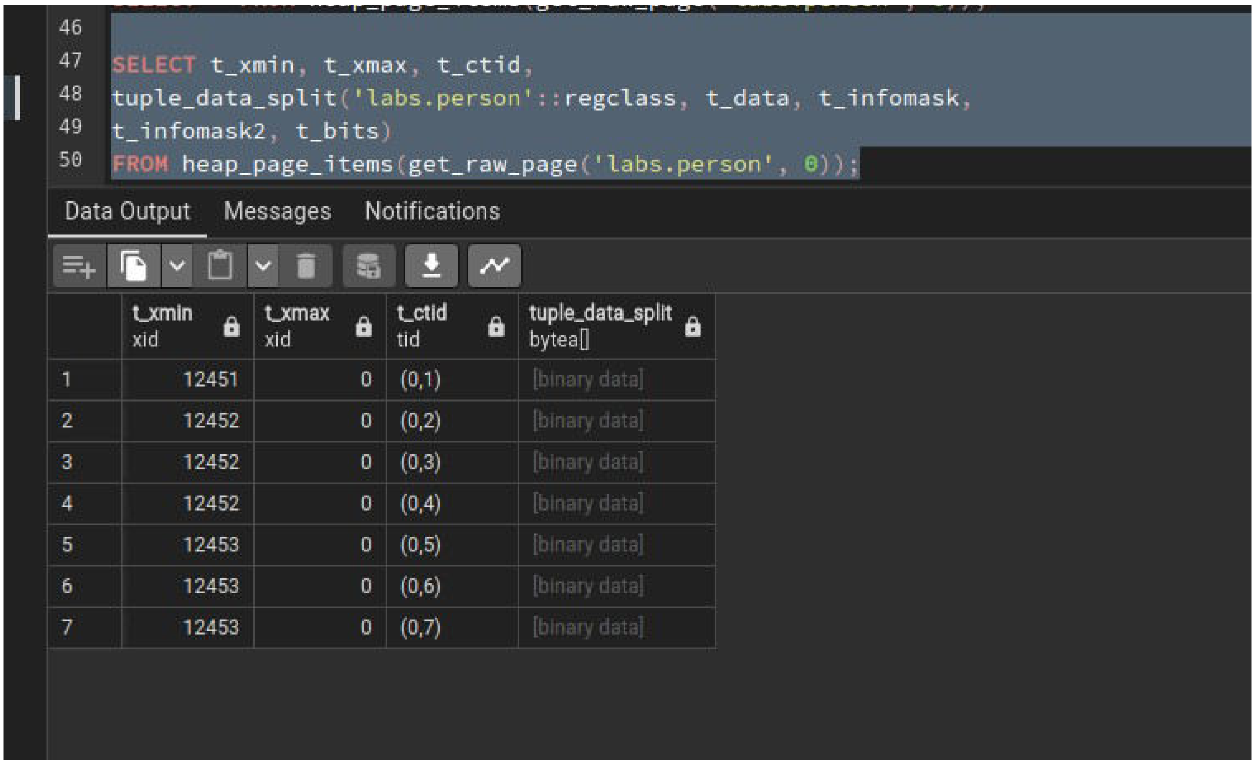
ctid is a unique identifier for the row within its table. It consists of the block number and the position within the block.PostgreSQL uses a block size of 8 kilobytes (8192 bytes) by default it menas in when block 0 get 8192 bytes it will continue in next block.
Xmin shows transaction ID of the INSERT transaction.xmax shows deleting transaction ID , it show's 0 as the rows are not deleted right now.



```
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));
```

	t_xmin xid	t_xmax xid	t_ctid tid	tuple_data_split bytea[]
1	12451	0	(0,1)	[binary data]
2	12452	0	(0,2)	[binary data]
3	12452	0	(0,3)	[binary data]
4	12452	0	(0,4)	[binary data]
5	12453	0	(0,5)	[binary data]
6	12453	0	(0,6)	[binary data]
7	12453	0	(0,7)	[binary data]

get_raw_page function retrieves the raw data for page 0 of the `labs.person` table. heap_page_items function takes the raw page data and returns a set of rows, each representing a tuple (row) in the specified page.

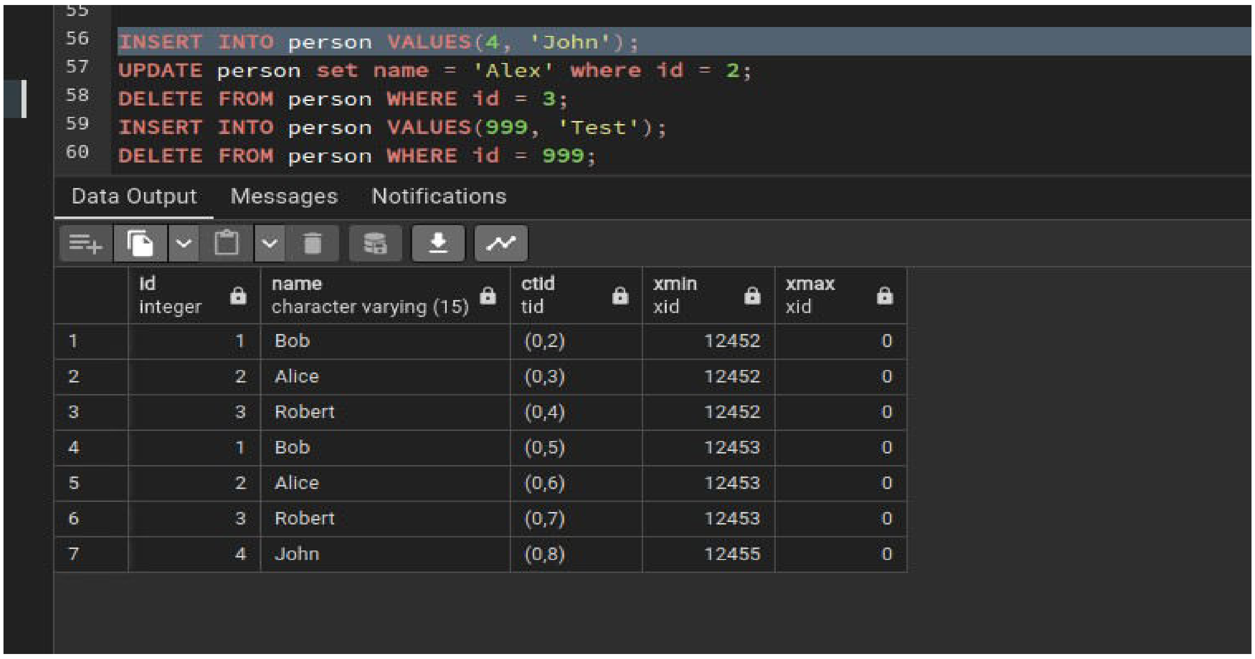


```
SELECT t_xmin, t_xmax, t_ctid,
tuple_data_split('labs.person'::regclass, t_data, t_infomask,
t_infomask2, t_bits)
FROM heap_page_items(get_raw_page('labs.person', 0));
```

Data Output Messages Notifications

	t_xmin xid	t_xmax xid	t_ctid tid	tuple_data_split bytea[]
1	12451	0	(0,1)	[binary data]
2	12452	0	(0,2)	[binary data]
3	12452	0	(0,3)	[binary data]
4	12452	0	(0,4)	[binary data]
5	12453	0	(0,5)	[binary data]
6	12453	0	(0,6)	[binary data]
7	12453	0	(0,7)	[binary data]

Idk why it's shows binary data but in documentation says it shows all line pointers on a heap page. For those line pointers that are in use, tuple headers as well as tuple raw data are also shown.

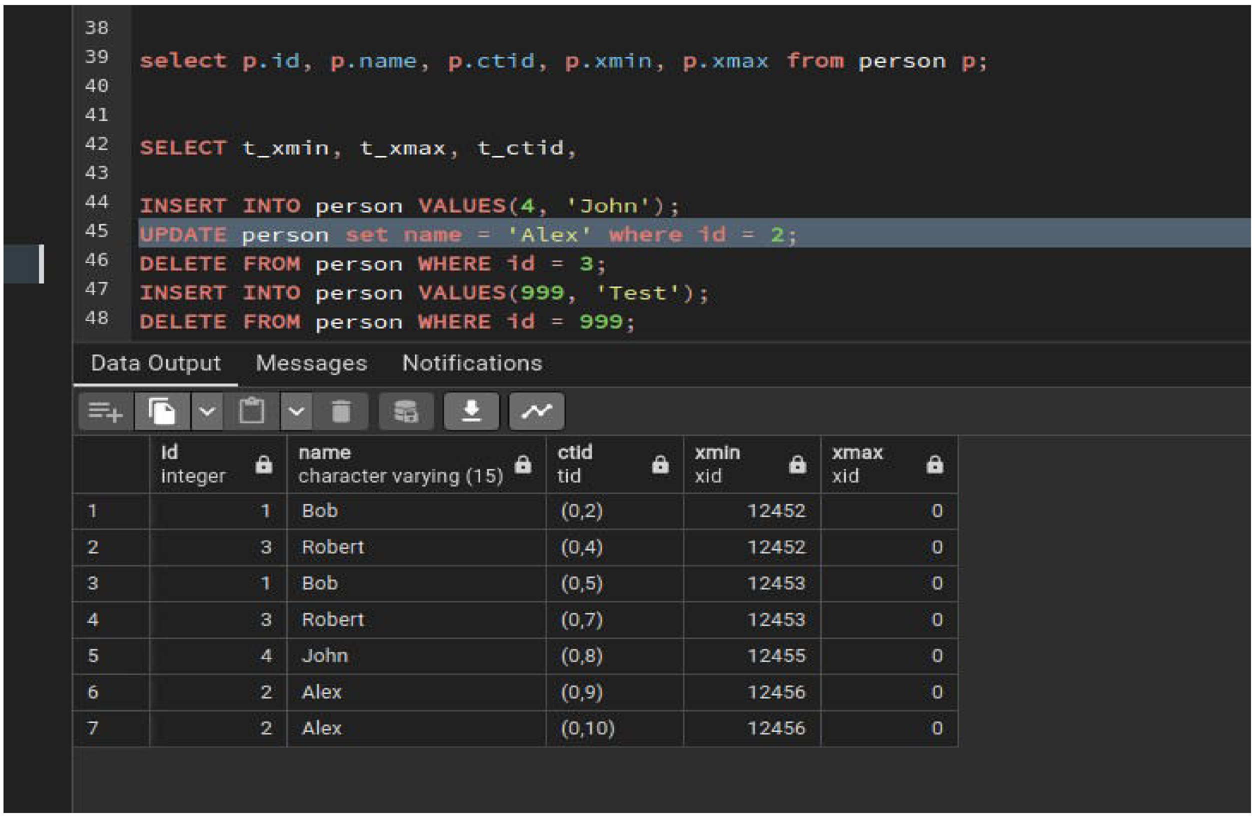


```
INSERT INTO person VALUES(4, 'John');
UPDATE person set name = 'Alex' where id = 2;
DELETE FROM person WHERE id = 3;
INSERT INTO person VALUES(999, 'Test');
DELETE FROM person WHERE id = 999;
```

Data Output Messages Notifications

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	2	Alice	(0,3)	12452	0
3	3	Robert	(0,4)	12452	0
4	1	Bob	(0,5)	12453	0
5	2	Alice	(0,6)	12453	0
6	3	Robert	(0,7)	12453	0
7	4	John	(0,8)	12455	0

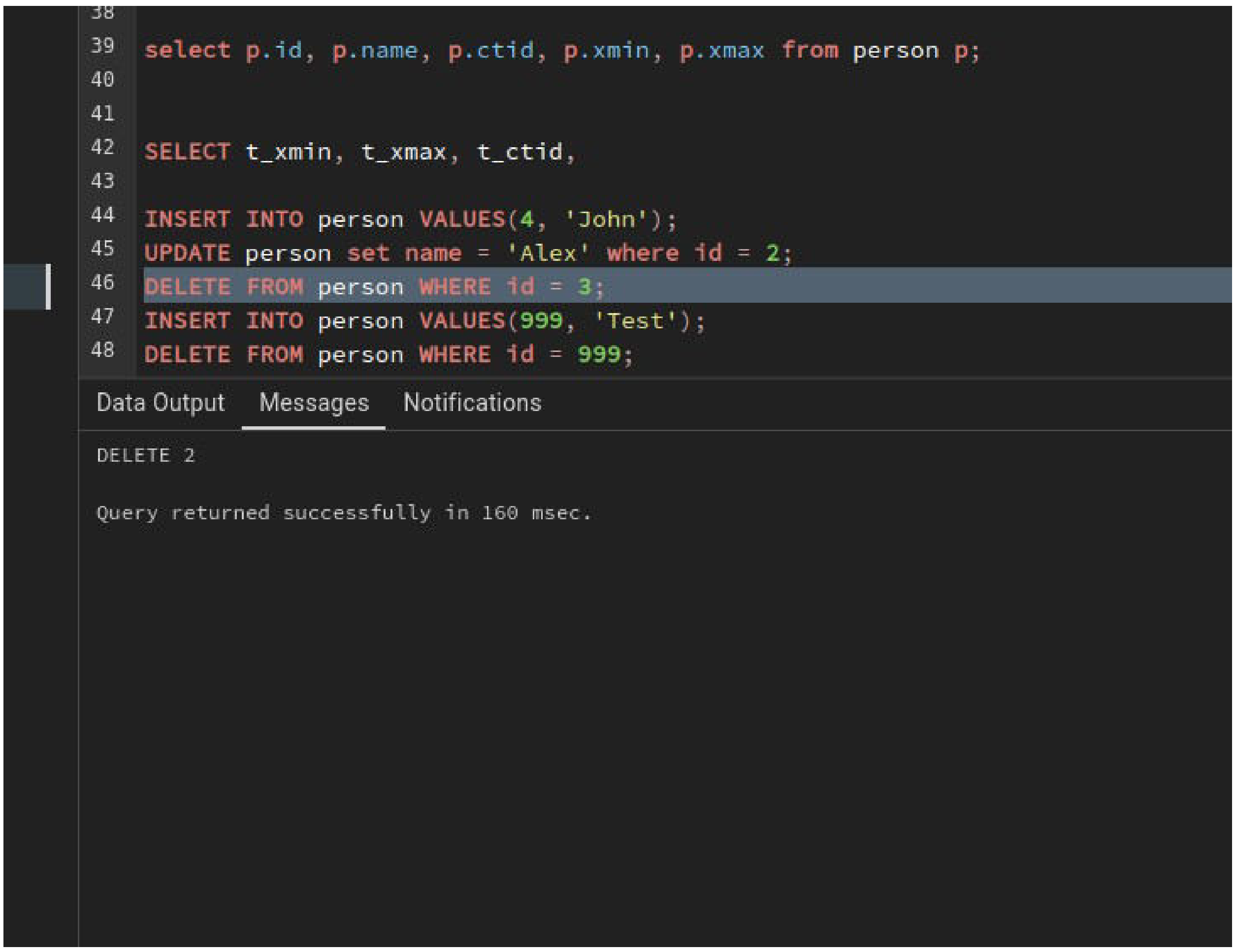
Inserted new row , and the xmin id is 12455



```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;


SELECT t_xmin, t_xmax, t_ctid,

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;
```

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	3	Robert	(0,4)	12452	0
3	1	Bob	(0,5)	12453	0
4	3	Robert	(0,7)	12453	0
5	4	John	(0,8)	12455	0
6	2	Alex	(0,9)	12456	0
7	2	Alex	(0,10)	12456	0

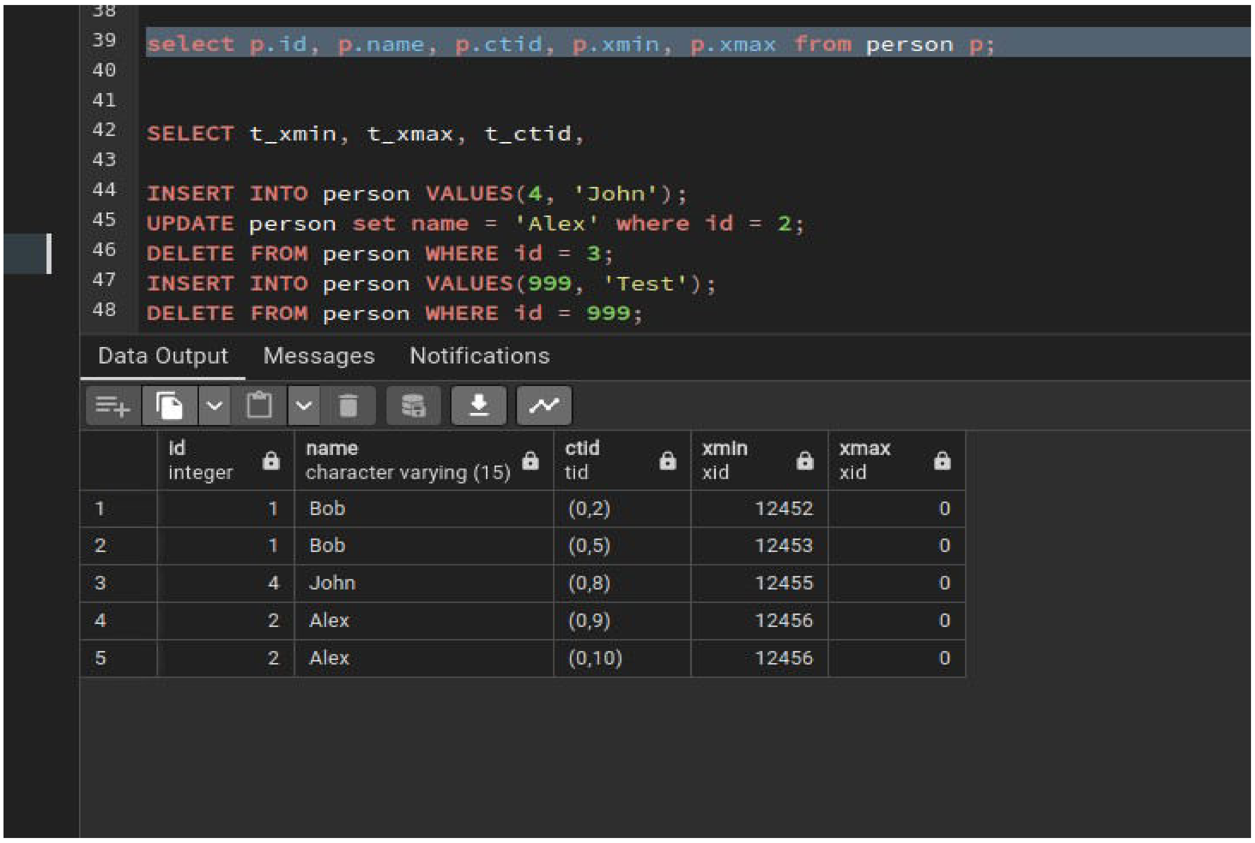
When we update existing row its got new ctid and and the xmin ID was updated to last transaction ID.



```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;


SELECT t_xmin, t_xmax, t_ctid,

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;
```

```
DELETE 2

Query returned successfully in 160 msec.
```

Deleted id 3.

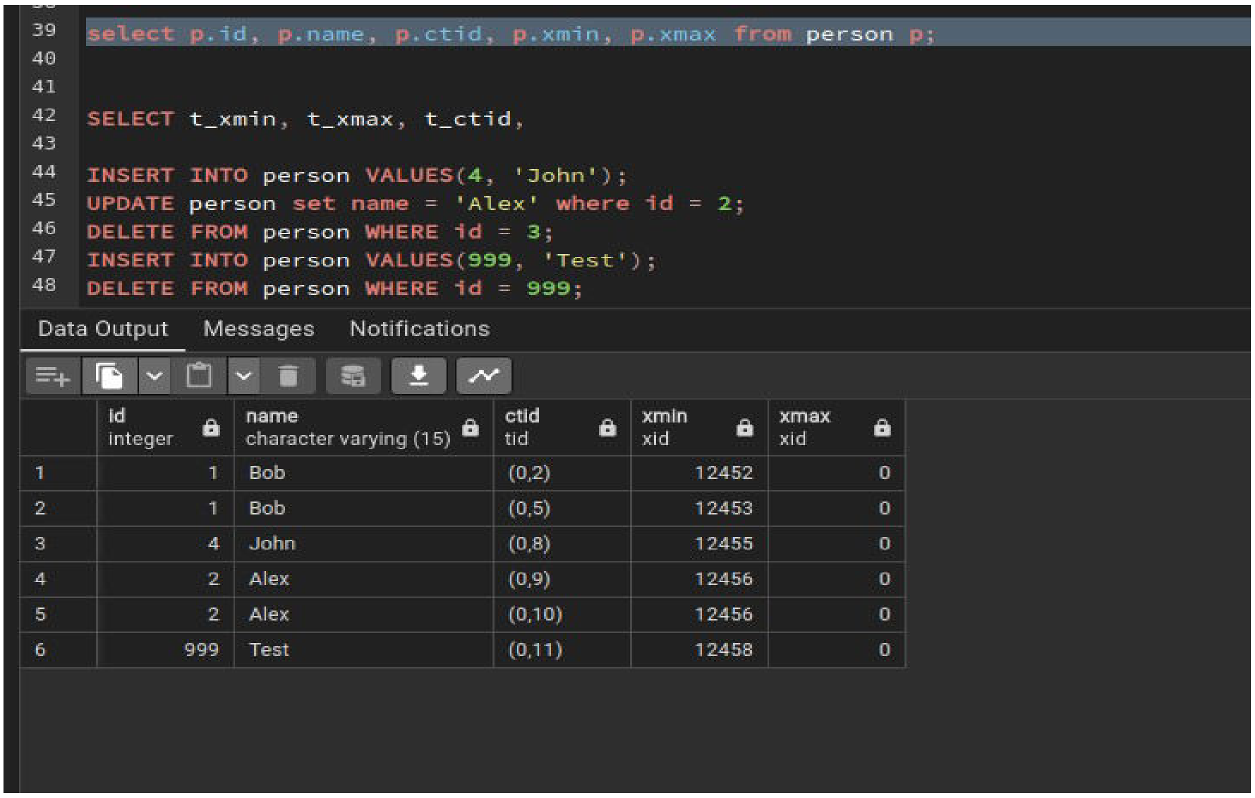


```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;


SELECT t_xmin, t_xmax, t_ctid,

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;
```

Data Output Messages Notifications

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	1	Bob	(0,5)	12453	0
3	4	John	(0,8)	12455	0
4	2	Alex	(0,9)	12456	0
5	2	Alex	(0,10)	12456	0

We can see that there is no row with id 3.

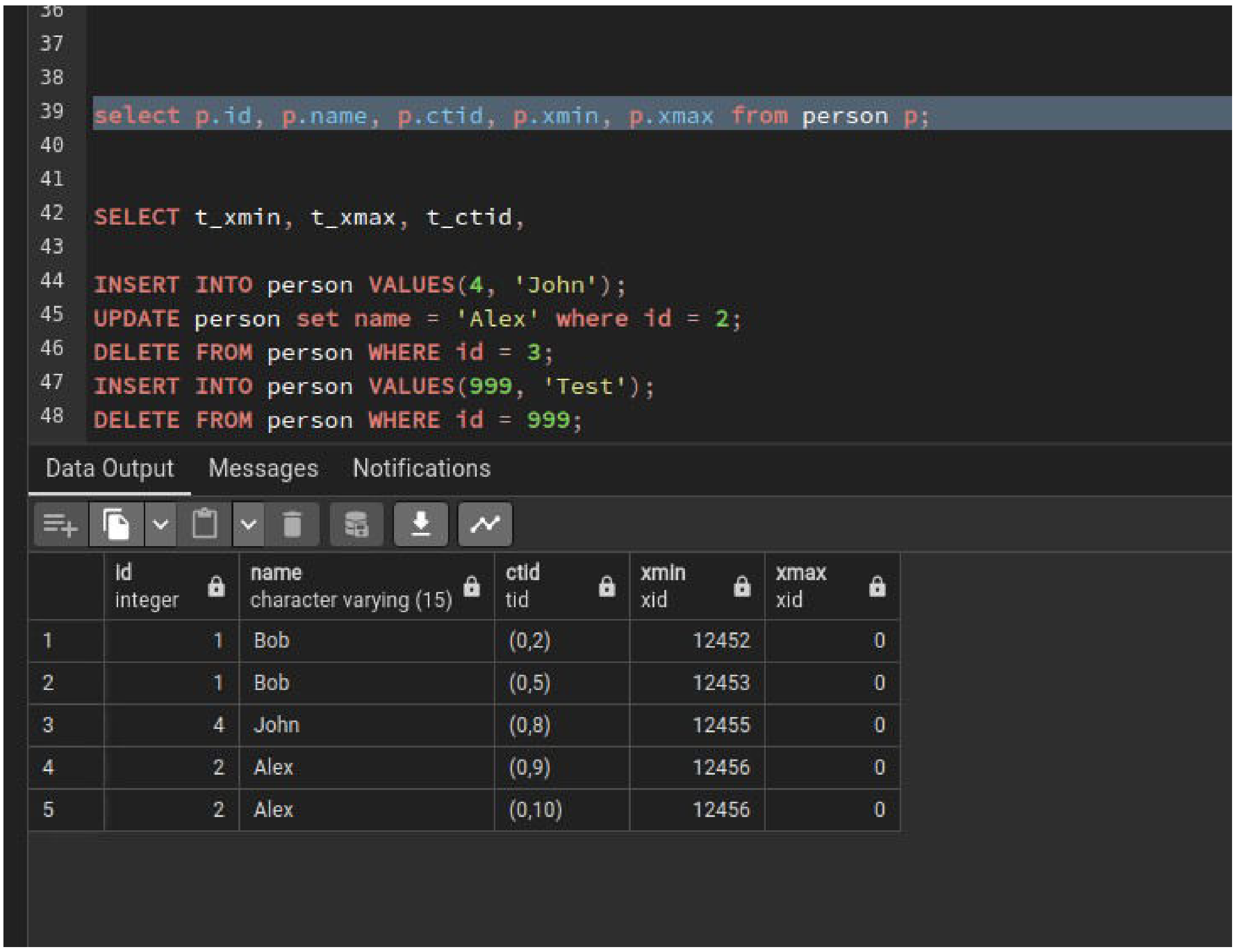


```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;


SELECT t_xmin, t_xmax, t_ctid,

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;
```

Data Output Messages Notifications

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	1	Bob	(0,5)	12453	0
3	4	John	(0,8)	12455	0
4	2	Alex	(0,9)	12456	0
5	2	Alex	(0,10)	12456	0
6	999	Test	(0,11)	12458	0

We can see its always get next values of ctid and xmin .



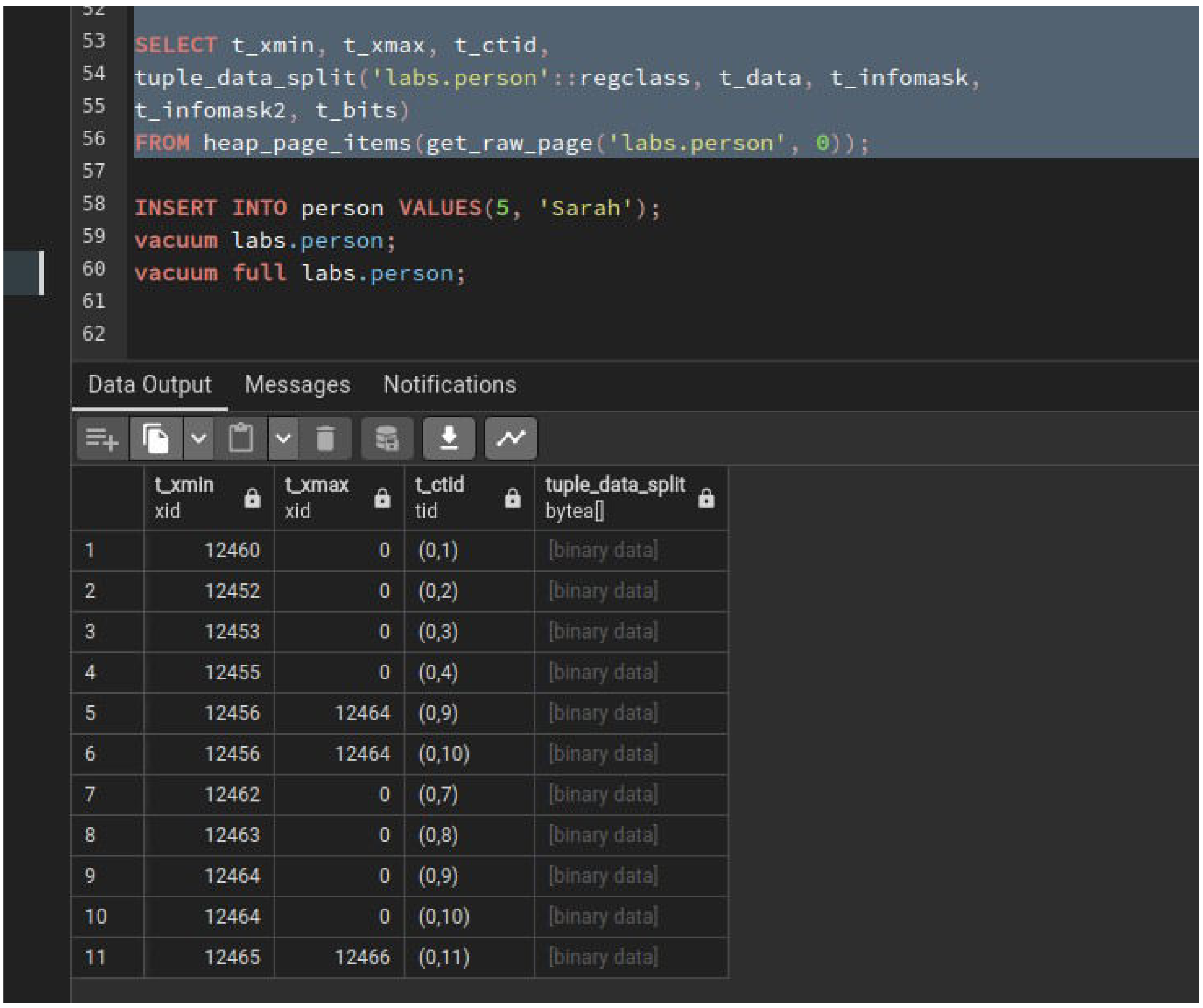
```
select p.id, p.name, p.ctid, p.xmin, p.xmax from person p;


SELECT t_xmin, t_xmax, t_ctid,

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;
```

Data Output Messages Notifications

	id integer	name character varying (15)	ctid tid	xmin xid	xmax xid
1	1	Bob	(0,2)	12452	0
2	1	Bob	(0,5)	12453	0
3	4	John	(0,8)	12455	0
4	2	Alex	(0,9)	12456	0
5	2	Alex	(0,10)	12456	0

Deleted last row.

1.5 TASK 5 INVESTIGATE VACUUM

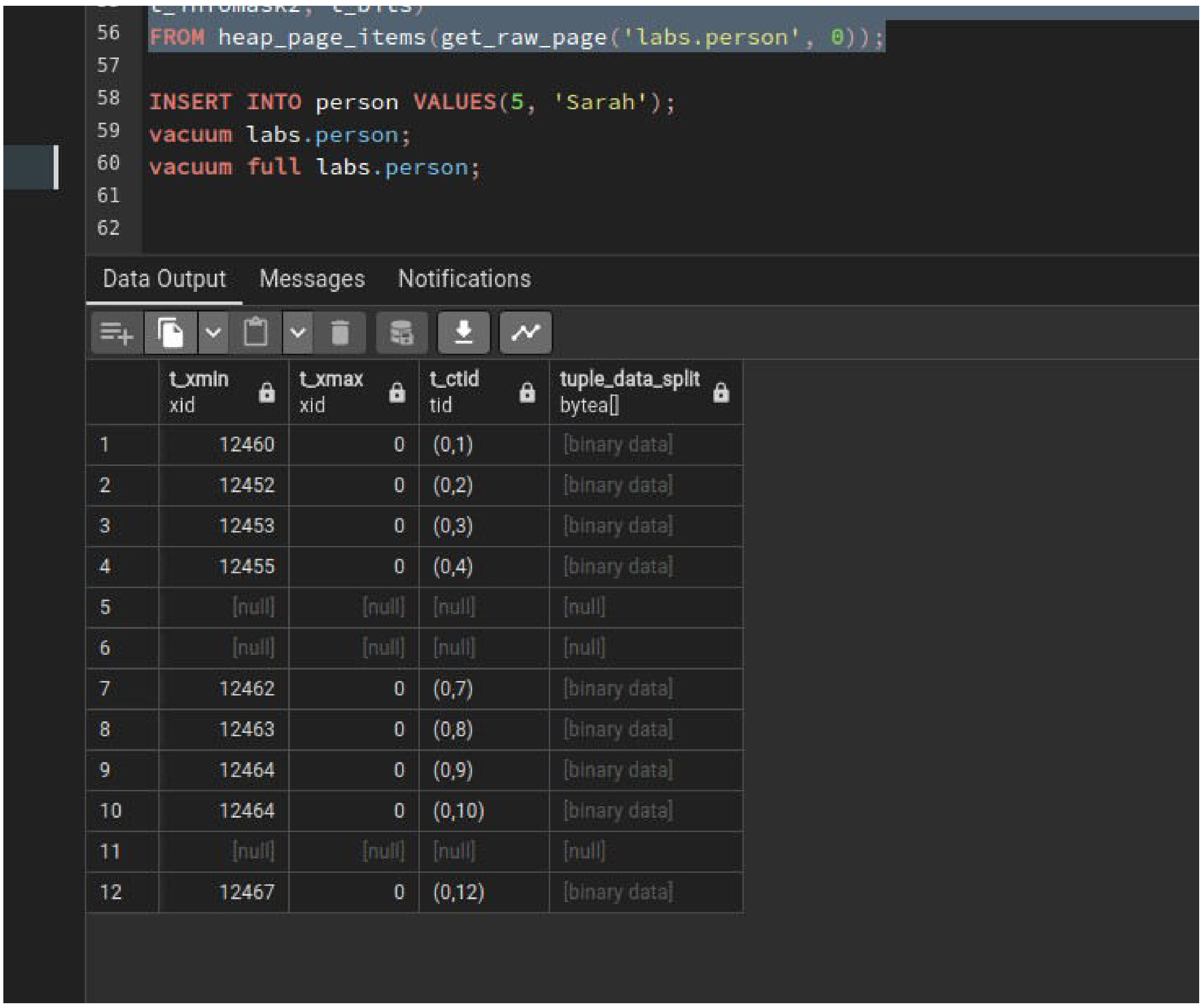


```
SELECT t_xmin, t_xmax, t_ctid,
tuple_data_split('labs.person'::regclass, t_data, t_infomask,
t_infomask2, t_bits)
FROM heap_page_items(get_raw_page('labs.person', 0));

INSERT INTO person VALUES(5, 'Sarah');
vacuum labs.person;
vacuum full labs.person;
```

Data Output Messages Notifications

	t_xmin xid	t_xmax xid	t_ctid tid	tuple_data_split bytea[]
1	12460	0	(0,1)	[binary data]
2	12452	0	(0,2)	[binary data]
3	12453	0	(0,3)	[binary data]
4	12455	0	(0,4)	[binary data]
5	12456	12464	(0,9)	[binary data]
6	12456	12464	(0,10)	[binary data]
7	12462	0	(0,7)	[binary data]
8	12463	0	(0,8)	[binary data]
9	12464	0	(0,9)	[binary data]
10	12464	0	(0,10)	[binary data]
11	12465	12466	(0,11)	[binary data]

When we delete some rows the xmax id shows that the row is deleted by that transaction ID but it’s remains heap_page_items so , to clear all deleted rows we need to exequte VACUUM



```sql
FROM heap_page_items(get_raw_page('labs.person', 0));

INSERT INTO person VALUES(5, 'Sarah');
vacuum labs.person;
vacuum full labs.person;
```

	t_xmin xid	t_xmax xid	t_ctid tid	tuple_data_split bytea[]
1	12460	0	(0,1)	[binary data]
2	12452	0	(0,2)	[binary data]
3	12453	0	(0,3)	[binary data]
4	12455	0	(0,4)	[binary data]
5	[null]	[null]	[null]	[null]
6	[null]	[null]	[null]	[null]
7	12462	0	(0,7)	[binary data]
8	12463	0	(0,8)	[binary data]
9	12464	0	(0,9)	[binary data]
10	12464	0	(0,10)	[binary data]
11	[null]	[null]	[null]	[null]
12	12467	0	(0,12)	[binary data]

After executing vacuum we see that deleted row gone.
VACUUM FULL rewrites the entire contents of the table into a new disk file with no extra space, allowing unused space to be returned to the operating system.