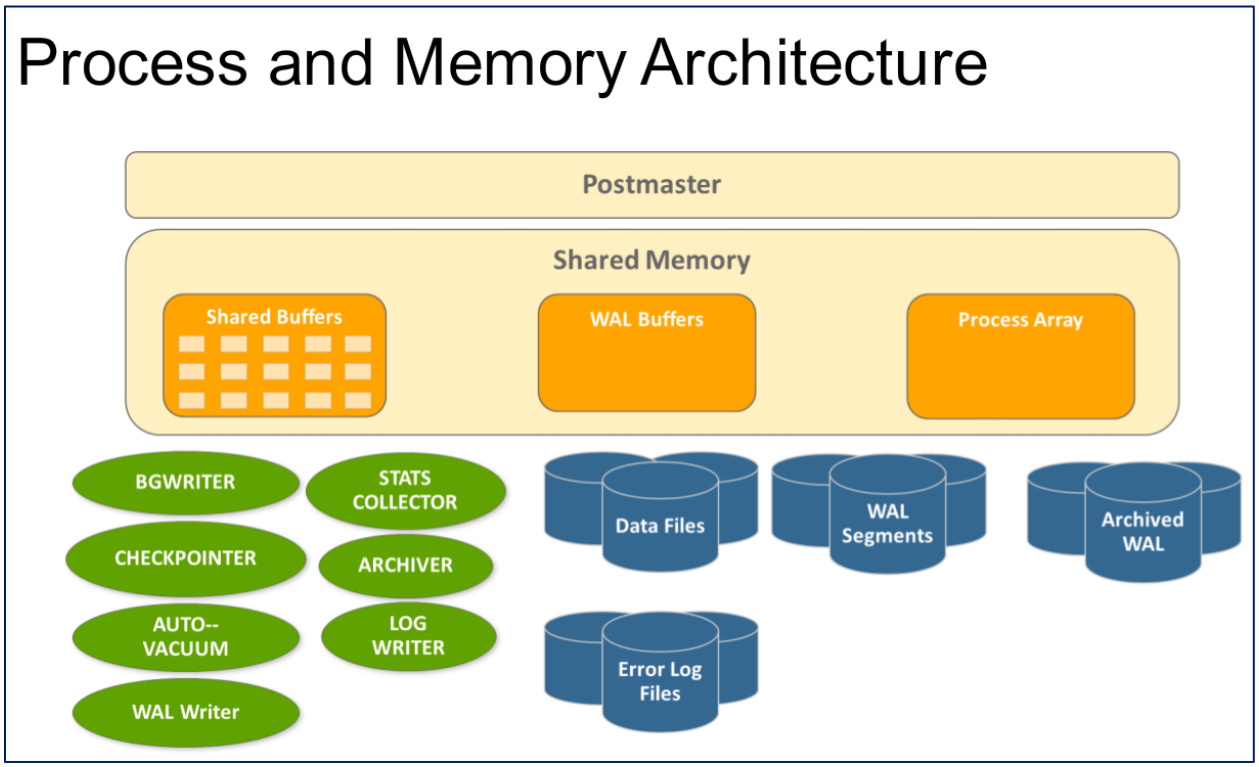
Data Fragmentation in PostgreSQL

PostgreSQL is **a relational database management system with a client-server architecture**. At the server side the PostgreSQL's processes and shared memory work together and build an instance, which handles the access to the data. Client programs connect to the instance and request read and write operations.

Architecture:



What is Fragmentation?

Fragmentation is a database server feature that allows you to control where data is stored at the table level. Fragmentation enables you to define groups of rows or index keys within a table according to some algorithm or scheme.

Fragmentation is often called ***bloating in PostgreSQL***. It relates to its implementation of **MVCC** (Multi-version Concurrency Control) where rows are not updated in place or directly deleted, but are copied with a different ID. Those rows are then made visible or invisible

depending on the transaction looking at the data. **Basically, any update on the table is a Delete – Insert** where existing row is first deleted and a new row is inserted.

MVCC (Multi-version Concurrency Control)

Multi-version concurrency control (MVCC), is a concurrency control method commonly used by database management systems to provide concurrent access to the database and in programming languages to implement transactional memory. Without concurrency control, if someone is reading from a database at the same time as someone else is writing to it, it is possible that the reader will see a half-written or inconsistent piece of data.

Unlike most other database systems which use locks for concurrency control, Postgres maintains data consistency by using a multi-version model. This means that while querying a database each transaction sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing transaction isolation for each database session.

Fragmentation Example (Bloating)

Let us take a simple example to understand what bloating is, how it occurs and how to get rid of it.
Step1: Create a simple table with one column. I have created a table called *datafreg* which has id column and it is also a primary key column as shown in fig below.
You can see the specifics of the table with \d <table_name> command as shown below.

```
postgres=# create table Datafreg(id numeric primary key);
CREATE TABLE
postgres=# \dt Datafreg
          List of relations
 Schema |   Name   | Type  |  Owner
--------+----------+-------+----------
 public | datafreg | table | postgres
(1 row)

postgres=# \d Datafreg
              Table "public.datafreg"
 Column |  Type   | Collation | Nullable | Default
--------+---------+-----------+----------+---------
 id     | numeric |           | not null |
Indexes:
    "datafreg_pkey" PRIMARY KEY, btree (id)
```

Step2: Generate some sample data in the table. I have used generate_series function and inserted 4000 rows. You can confirm the data inserted by using the count (*) on the table as shown below.

```
postgres=# insert into Datafreg values(generate_series(1,4000));
INSERT 0 4000
postgres=# select count(*) from Datafreg;
 count
-------
  4000
(1 row)
```

Step3: Query ***pg_stat_all_tables*** to check the details on live and dead **tuples**. Any rows inserted in the table will be shows as live tuples as its live data under a table and any rows that you delete will be shown under dead tuples. As shown in the below fig, as we have inserted 4000 records, 400 live tuples are available for the table.

```
postgres=# select relname,n_live_tup,n_dead_tup,last_vacuum from pg_stat_all_tables where relname='datafreg';
 relname  | n_live_tup | n_dead_tup | last_vacuum
----------+------------+------------+-------------
 datafreg |       4000 |          0 |
(1 row)
```

Step4: Also, check the size of the table along with the count. Here, I have used ***pg_size_pretty*** function to check the size of the table. There are lot of ways to check the size of the table, you can use any of them. As shown, the size of the table is **416 KB**

```
postgres=# SELECT pg_size_pretty( pg_total_relation_size('Datafreg')) "Table_Size",count(*) from Datafreg;
 Table_Size | count
------------+-------
 416 kB     |  4000
(1 row)
```

Step5: Delete some rows of the table, Here I have deleted 2000 rows using the between clause. After deleting the rows, check the tuples again, since you have 2000 rows in the table, it shows under live tuple and you deleted 2000 rows, those are shown under dead tuple.

```
postgres=# delete from datafreg where id between 1 AND 2000;
DELETE 2000
postgres=#  select relname,n_live_tup,n_dead_tup,last_vacuum from pg_stat_all_tables where relname='datafreg';
 relname  | n_live_tup | n_dead_tup | last_vacuum
----------+------------+------------+-------------
 datafreg |       2000 |       2000 |
(1 row)
```

Step6: After deleting the rows, check the size and count of the table again. As shown below, you will see the count has reduced to 2000 but the size of the table remains the same. It was 416 KB (Check Step 4) initially and even after deleting 2000 rows, the size of the table remains same.

This is called bloating. Even after deleting the data, space used by the deleted data is not released from the table. It will lie in the table and database will think that data is using the space but in real it's just an impression and there does not exist any data since we have deleted it already.

```
postgres=# SELECT pg_size_pretty( pg_total_relation_size('Datafreg')) "Table_Size",count(*) from Datafreg;
 Table_Size | count
------------+-------
 416 kB     |  2000
(1 row)
```

Step7: In order to free up the unutilized space and defragment the table, you will need to periodically perform **VACUUM** on the tables. So any tables which has high volume of DML's happening on it should be vacuumed periodically to make sure performance degradation does not happen and the unused space is released to the file system.

As shown below, after vacuuming the data, the size of the table has come down from **416 KB to 144 KB** keeping the number of records same.

```
postgres=# vacuum full datafreg;
VACUUM
postgres=# SELECT pg_size_pretty( pg_total_relation_size('Datafreg')) "Table_Size",count(*) from Datafreg;
 Table_Size | count
------------+-------
 144 kB     |  2000
(1 row)
```

Also, the dead tuples (2000 – Step 5) which were consuming the space even after deletion have been removed after vacuuming the table. See the fig below.

```
postgres=# select relname,n_live_tup,n_dead_tup from pg_stat_all_tables where relname='datafreg';
 relname  | n_live_tup | n_dead_tup
----------+------------+------------
 datafreg |       2000 |          0
(1 row)
```

Vacuum can be scheduled and can be performed on adhoc basis. It is very imp to create a vacuum strategy for your postgres database. It will play a considerable part in maintaining the performance of the database and also managing the space capacity of the file system.