# Experiment No.2

**Title:** Implementation of Distributed Database.

**Batch: B-4** **Roll No.: 16010422234** **Name: Chandana Ramesh Galgali**

**Experiment No.: 2**

**Aim:** To Implement Distributed Database.

---

**Resources needed:** PostgreSQL 9.3

---

**Theory**

A distributed database system allows applications to access and manipulate data from local and remote databases. It partitions the data and stores it at different physical locations. Partitioning refers to splitting what is logically one large table into smaller physical pieces.

Partitioning can provide several benefits:

- Query performance can be improved dramatically for certain kinds of queries.
- Update performance can be improved too, since each piece of the table has indexes smaller than an index on the entire data set would be. When an index no longer fits easily in memory, both read and write operations on the index take progressively more disk accesses.
- Bulk deletes may be accomplished by simply removing one of the partitions, if that requirement is planned into the partitioning design. DROP TABLE is far faster than a bulk DELETE, to say nothing of the ensuing VACUUM overhead.
- Seldom-used data can be migrated to cheaper and slower storage media.
- Partitioning enhances the performance, manageability, and availability of a wide variety of applications and helps reduce the total cost of ownership for storing large amounts of data. Partitioning allows tables, indexes, and index-organized tables to be subdivided into smaller pieces, enabling these database objects to be managed and accessed at a finer level of granularity.

The benefits will normally be worthwhile only when a table would otherwise be very large. The exact point at which a table will benefit from partitioning depends on the application, although a rule of thumb is that the size of the table should exceed the physical memory of

The following forms of partitioning can be implemented in PostgreSQL:

**Range Partitioning**

The table is partitioned into "ranges" defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example one might partition by date ranges, or by ranges of identifiers for particular business objects.

**List Partitioning**

The table is partitioned by explicitly listing which key values appear in each partition.

After creating the partition, **database link (DBLINK)** is used to create a connection of the host database server with the client database.

**Database Links:**

The central concept in distributed database systems is a **database link**. A database link is a connection between two physical database servers that allows a client to access them as one logical database. Database link is a pointer that defines a one-way communication path from one Database server to another database server. The link pointer is actually defined as an entry in a data dictionary table. To access the link, you must be connected to the local database that contains the data dictionary entry.

A database link connection is one-way in the sense that a client connected to local database A can use a link stored in database A to access information in remote database B, but users connected to database B cannot use the same link to access data in database A. If local users on database B want to access data on database A, then they must define a link that is stored in the data dictionary of database B.

A database link connection allows local users to access data on a remote database. For this connection to occur, each database in the distributed system must have a unique global database name in the network domain. The global database name uniquely identifies a database server in a distributed system.

dblink executes a query (usually a SELECT, but it can be any SQL statement that returns rows) in a remote database.

When two text arguments are given, the first one is first looked up as a persistent connection's name; if found, the command is executed on that connection. If not found, the first argument is treated as a connection info string as for dblink_connect, and the indicated connection is made just for the duration of this command.



**Arguments**

conname
: Name of the connection to use; omit this parameter to use the unnamed connection.

connstr
: A connection info string, as previously described for dblink_connect.

sql
: The SQL query that you wish to execute in the remote database, for example select * from <table_name>.

fail_on_error
: If true (the default when omitted) then an error thrown on the remote side of the connection causes an error to also be thrown locally. If false, the remote error is locally reported as a NOTICE, and the function returns no rows.

**Return Value**

The function returns the row(s) produced by the query. Since dblink can be used with any query, it is declared to return records, rather than specifying any particular set of columns. This means that you must specify the expected set of columns in the calling query — otherwise PostgreSQL would not know what to expect. Here is an example:

```
SELECT *
  FROM dblink('dbname=mydb', 'select proname, prosrc from pg_proc')
    AS t1(proname name, prosrc text)
  WHERE proname LIKE 'bytea%';
```

**Procedure:**

**Implementing distributed database:**

1. **Create the parent table.**

```
CREATE TABLE sales(org int, name varchar(10));
```

2. **Create the child (partitioned) tables**

```
CREATE TABLE sales_part1
(CHECK (org < 6))
INHERITS (sales);

CREATE TABLE sales_part2
(CHECK (org >=6 and org <=10))
INHERITS (sales);
```

3. **Create the rules**

```
CREATE OR REPLACE RULE insert_sales_p1
AS ON INSERT TO sales
WHERE (org <6)
DO INSTEAD
INSERT INTO sales_part1 VALUES(NEW.org, NEW.name);

CREATE OR REPLACE RULE insert_sales_p2
AS ON INSERT TO sales
WHERE (org >=6 and org <=10 )
DO INSTEAD
INSERT INTO sales_part2 VALUES(New.org,New.name);
```

4. **Add sample data to the new table.**

```
INSERT INTO sales VALUES(1,'Craig');
INSERT INTO sales VALUES(2,'Mike');
INSERT INTO sales VALUES(3,'Michelle');
INSERT INTO sales VALUES(4,'Joe');
INSERT INTO sales VALUES(5,'Scott');
INSERT INTO sales VALUES(6,'Roger');
INSERT INTO sales VALUES(7,'Fred');
INSERT INTO sales VALUES(8,'Sam');
INSERT INTO sales VALUES(9,'Sonny');
INSERT INTO sales VALUES(10,'Chris');
```

5. **Confirm that the data was added to the parent table and the partition tables**

```
SELECT * FROM sales;
SELECT * FROM sales_part1;
SELECT * FROM sales_part2;
```

6. **Create a dblink_connect to create a connection string to use.**

   Access the file : pg_hba.conf file under C:\Program Files\PostgreSQL\9.3\data and make the following entry , stating that the host machine is accessible to other machines.

   ```
   host    all         all         all         trust
   Create Extension dblink;
   SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.103  dbname=postgres
   user=postgres password=postgres')
   ```

   172.17.17.103 *is the host address that has the database and the partitions .*

7. **Use dblink command on the remote machine to access the partitions present in the host machine.**
   Access the file : pg_hba.conf file under C:\Program Files\PostgreSQL\9.3\data and make the following entry , stating that the remote machine needs to access the host machine:
   host postgres postgres 172.17.17.103/32 md5

   *And the client can execute the following command, in the SQL Query window:*
   sample:

   Create Extension dblink;
   SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.103 dbname=postgres user=postgres password=postgres')

   select * from dblink('myconn','select * from sales_part2')AS T1(Column1 int, column2 varchar(10)) order by column2 desc;

   Inserting data into the table remotely

   select dblink_exec('myconn','insert into sales values(12,"John")')
   select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;

   Delete data from table remotely

   select dblink_exec('myconn','delete from sales where org=3')
   select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;

---

**Results: (Program printout with output)**

postgres1 on postgres@PostgreSQL 9.6

```
1  CREATE TABLE sales(org int, name varchar(10));
2  CREATE TABLE sales_part1
3  (CHECK (org < 6))
4  INHERITS (sales);
5  CREATE TABLE sales_part2
6  (CHECK (org >=6 and org <=10))
7  INHERITS (sales);
```

Data Output | Explain | Messages | History

```
CREATE TABLE

Query returned successfully in 590 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
1  CREATE TABLE sales(org int, name varchar(10));
2  CREATE TABLE sales_part1
3  (CHECK (org < 6))
4  INHERITS (sales);
5  CREATE TABLE sales_part2
6  (CHECK (org >=6 and org <=10))
7  INHERITS (sales);
```

Data Output | Explain | Messages | History

```
CREATE TABLE

Query returned successfully in 430 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
1  CREATE TABLE sales(org int, name varchar(10));
2  CREATE TABLE sales_part1
3  (CHECK (org < 6))
4  INHERITS (sales);
5  CREATE TABLE sales_part2
6  (CHECK (org >=6 and org <=10))
7  INHERITS (sales);
```

Data Output | Explain | Messages | History

```
CREATE TABLE

Query returned successfully in 342 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
8  CREATE OR REPLACE RULE insert_sales_p1
9  AS ON INSERT TO sales
10 WHERE (org <6)
11 DO INSTEAD
12 INSERT INTO sales_part1 VALUES(NEW.org, NEW.name);
```

Data Output | Explain | Messages | History

```
CREATE RULE

Query returned successfully in 342 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
13  CREATE OR REPLACE RULE insert_sales_p2
14  AS ON INSERT TO sales
15  WHERE (org >=6 and org <=10 )
16  DO INSTEAD
17  INSERT INTO sales_part2 VALUES(New.org,New.name);
```

Data Output | Explain | Messages | History

```
CREATE RULE

Query returned successfully in 309 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
18  INSERT INTO sales VALUES(1,'Craig');
19  INSERT INTO sales VALUES(2,'Mike');
20  INSERT INTO sales VALUES(3,'Michelle');
21  INSERT INTO sales VALUES(4,'Joe');
22  INSERT INTO sales VALUES(5,'Scott');
23  INSERT INTO sales VALUES(6,'Roger');
24  INSERT INTO sales VALUES(7,'Fred');
25  INSERT INTO sales VALUES(8,'Sam');
26  INSERT INTO sales VALUES(9,'Sonny');
27  INSERT INTO sales VALUES(10,'Chris');
```

Data Output | Explain | Messages | History

```
INSERT 0 0

Query returned successfully in 589 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
18  INSERT INTO sales VALUES(1,'Craig');
19  INSERT INTO sales VALUES(2,'Mike');
20  INSERT INTO sales VALUES(3,'Michelle');
21  INSERT INTO sales VALUES(4,'Joe');
22  INSERT INTO sales VALUES(5,'Scott');
23  INSERT INTO sales VALUES(6,'Roger');
24  INSERT INTO sales VALUES(7,'Fred');
25  INSERT INTO sales VALUES(8,'Sam');
26  INSERT INTO sales VALUES(9,'Sonny');
27  INSERT INTO sales VALUES(10,'Chris');
```

Data Output | Explain | Messages | History

```
INSERT 0 0

Query returned successfully in 390 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
26  INSERT INTO sales VALUES(9,'Sonny');
27  INSERT INTO sales VALUES(10,'Chris');
28  SELECT * FROM sales;
29  SELECT * FROM sales_part1;
30  SELECT * FROM sales_part2;
```

Data Output | Explain | Messages | History

| org integer | name character varying (10) |
|---|---|
| 1 | Craig |
| 2 | Mike |
| 3 | Michelle |
| 4 | Joe |
| 5 | Scott |
| 6 | Roger |
| 7 | Fred |
| 8 | Sam |
| 9 | Sonny |
| 10 | Chris |

postgres1 on postgres@PostgreSQL 9.6

```
26  INSERT INTO sales VALUES(9,'Sonny');
27  INSERT INTO sales VALUES(10,'Chris');
28  SELECT * FROM sales;
29  SELECT * FROM sales_part1;
30  SELECT * FROM sales_part2;
```

Data Output | Explain | Messages | History

| org integer | name character varying (10) |
|---|---|
| 1 | Craig |
| 2 | Mike |
| 3 | Michelle |
| 4 | Joe |
| 5 | Scott |

postgres1 on postgres@PostgreSQL 9.6

```
26  INSERT INTO sales VALUES(9,'Sonny');
27  INSERT INTO sales VALUES(10,'Chris');
28  SELECT * FROM sales;
29  SELECT * FROM sales_part1;
30  SELECT * FROM sales_part2;
```

Data Output | Explain | Messages | History

| org integer | name character varying (10) |
|---|---|
| 6 | Roger |
| 7 | Fred |
| 8 | Sam |
| 9 | Sonny |
| 10 | Chris |

postgres1 on postgres@PostgreSQL 9.6

```
28  SELECT * FROM sales;
29  SELECT * FROM sales_part1;
30  SELECT * FROM sales_part2;
31  Create Extension dblink;
32  SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.137  dbname=postgres1 user=postgres password=postgres')
```

Data Output | Explain | Messages | History

```
CREATE EXTENSION

Query returned successfully in 461 msec.
```

postgres1 on postgres@PostgreSQL 9.6

```
28  SELECT * FROM sales;
29  SELECT * FROM sales_part1;
30  SELECT * FROM sales_part2;
31  Create Extension dblink;
32  SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.137  dbname=postgres1 user=postgres password=postgres')
```

Data Output | Explain | Messages | History

| dblink_connect<br>text |
|---|
| OK |

postgres1 on postgres@PostgreSQL 9.6

```
1  Create Extension dblink;
2  SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.103  dbname=postgres user=postgres password=postgres')
3  select * from dblink('myconn','select * from sales_part2')AS T1(Column1 int, column2 varchar(10)) order by column2 desc;
4
5
```

Data Output | Explain | Messages | History

```
CREATE EXTENSION

Query returned successfully in 1 secs.
```

postgres1 on postgres@PostgreSQL 9.6

```
1  Create Extension dblink;
2  SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.137  dbname=postgres1 user=postgres password=postgres')
3  select * from dblink('myconn','select * from sales_part2')AS T1(Column1 int, column2 varchar(10)) order by column2 desc;
4
5
```

Data Output | Explain | Messages | History

| dblink_connect<br>text |
|---|
| OK |

postgres1 on postgres@PostgreSQL 9.6

```
1  Create Extension dblink;
2  SELECT dblink_connect('myconn' ,'hostaddr=172.17.17.137  dbname=postgres1 user=postgres password=postgres')
3  select * from dblink('myconn','select * from sales_part2')AS T1(Column1 int, column2 varchar(10))
4  order by column2 desc;
5
```

Data Output | Explain | Messages | History

| column1<br>integer | column2<br>character varying (10) |
|---|---|
| 9 | Sonny |
| 8 | Sam |
| 6 | Roger |
| 7 | Fred |
| 10 | Chris |

postgres1 on postgres@PostgreSQL 9.6

```
8
9  select dblink_exec('myconn','insert into sales values(12,''John'')')
10 select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;
11
```

Data Output | Explain | Messages | History

| dblink_exec<br>text |
|---|
| INSERT 0 1 |

postgres1 on postgres@PostgreSQL 9.6

```
7
8
9  select dblink_exec('myconn','insert into sales values(12,''John'')')
10 select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;
11
```

Data Output | Explain | Messages | History

| column1 integer | column2 character varying (10) |
|---|---|
| 1 | Craig |
| 2 | Mike |
| 3 | Michelle |
| 4 | Joe |
| 5 | Scott |
| 6 | Roger |
| 7 | Fred |
| 8 | Sam |
| 9 | Sonny |
| 10 | Chris |
| 12 | John |

postgres1 on postgres@PostgreSQL 9.6

```
1
2  select dblink_exec('myconn','delete from sales where org=3');
3  select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;
4
5
```

Data Output | Explain | Messages | History

| dblink_exec text |
|---|
| DELETE 1 |

postgres1 on postgres@PostgreSQL 9.6

```
1
2  select dblink_exec('myconn','delete from sales where org=3');
3  select * from dblink('myconn','select * from sales')AS T1(Column1 int, column2 varchar(10)) order by column1 asc;
4
5
```

Data Output | Explain | Messages | History

| column1 integer | column2 character varying (10) |
|---|---|
| 1 | Craig |
| 2 | Mike |
| 4 | Joe |
| 5 | Scott |
| 6 | Roger |
| 7 | Fred |
| 8 | Sam |
| 9 | Sonny |
| 10 | Chris |
| 12 | John |

---

**Questions:**

1. **What are the different types of distributed database systems?**

**Ans:** Distributed database systems are designed to store and manage data across multiple interconnected nodes or locations. There are several types of distributed database systems, each with its own characteristics and advantages. Here are some common types:

1. Homogeneous Distributed Database:
   - In this type, all the database nodes use the same database management system (DBMS) software.
   - Data and processing are distributed uniformly across the nodes.
   - Offers a consistent environment across the distributed system.

2. Heterogeneous Distributed Database:
   - In contrast to homogeneous systems, heterogeneous distributed databases involve different types of DBMS software on different nodes.
   - This type of system is more complex due to the need for interoperability between different DBMSs.

3. Federated Database System:
- A federated database system is a collection of independent databases that cooperate to provide users with a single, integrated view of the data.
- Each database in the federation remains autonomous and can be a different DBMS.
- Provides a higher level of data independence.

4. Multi-database System:
- Similar to federated databases, multi-database systems involve multiple independent databases.
- However, in multi-database systems, there is typically a common schema or global schema that integrates the data from different databases.

5. Client-Server Database System:
- In a client-server distributed database system, clients interact with a centralized server to access and manipulate data.
- The server manages the data and provides services to multiple clients.
- Common in many enterprise applications.

6. Parallel Database System:
- In a parallel database system, a single database is distributed across multiple nodes, and each node processes a subset of the data concurrently.
- This type is designed to improve performance through parallel processing.

7. Replicated Database System:
- In a replicated database system, the same data is stored on multiple nodes or servers.
- Enhances fault tolerance and availability.
- Synchronization mechanisms are required to keep replicas consistent.

8. Distributed File System:
- While not strictly a database system, distributed file systems like Hadoop Distributed File System (HDFS) or Google File System (GFS) distribute and manage large volumes of data across multiple nodes.

Each type of distributed database system has its own set of advantages and challenges, and the choice depends on factors such as the application requirements, scalability needs, fault tolerance, and performance considerations.

**2. Give steps to insert and delete records in the remote table.**

**Ans:** Inserting and deleting records in a remote table in a distributed database system involve communication between the local and remote nodes. The specific steps may vary depending on the database management system (DBMS) and the type of distributed architecture you're working with. Here are generalized steps for both inserting and deleting records:

Inserting Records in a Remote Table:

1. Connect to the Local Database:
- Establish a connection to the local database using a database client or application.

2. Prepare the Insert Statement:
- Write an SQL INSERT statement to insert records into the local table.
- Specify the values or data you want to insert.

3. Execute the Insert Statement:
- Execute the INSERT statement using the database client or application connected to the local database.
- The local DBMS processes the insert operation for the local table.

4. Replication or Distribution:
- If the distributed database system uses replication or distribution mechanisms, the local DBMS communicates with the remote node(s) to replicate or distribute the new records.

5. Remote Insert Operation:
- The remote DBMS executes the corresponding insert operation on the remote table, ensuring data consistency across nodes.

Deleting Records from a Remote Table:

1. Connect to the Local Database:
- Establish a connection to the local database using a database client or application.

2. Prepare the Delete Statement:
- Write an SQL DELETE statement to delete records from the local table.
- Specify the condition for selecting the records to delete.

3. Execute the Delete Statement:
- Execute the DELETE statement using the database client or application connected to the local database.
- The local DBMS processes the delete operation for the local table.

4. Replication or Distribution:
- If the distributed database system uses replication or distribution mechanisms, the local DBMS communicates with the remote node(s) to replicate or distribute the deletion information.

5. Remote Delete Operation:
- The remote DBMS executes the corresponding delete operation on the remote table, ensuring data consistency across nodes.

Considerations:

Transaction Management:
- Ensure that these operations are part of a transaction to maintain data consistency and integrity.

Error Handling:
- Implement proper error handling mechanisms to address any issues that may arise during the insertion or deletion process.

Network Communication:
- Consider network latency and reliability when dealing with remote operations.

It's important to note that the specific steps and mechanisms for inserting and deleting records in a remote table may vary based on the DBMS and the distributed database architecture in use.

---

**Outcomes: Design advanced database systems using Parallel, Distributed and In - memory databases and its implementation.**

---

**Conclusion: (Conclusion to be based on the outcomes achieved):**
The experiment to implement a distributed database yielded positive outcomes in terms of scalability, fault tolerance, and overall performance. However, it also emphasized the

importance of addressing complexity, network considerations, and effective management strategies to harness the full potential of distributed database systems.

---

**Grade: AA / AB / BB / BC / CC / CD /DD**

**Signature of faculty in-charge with date**

---

**References:**

**Books/ Journals/ Websites:**

1. Elmasri and Navathe, “Fundamentals of Database Systems”, Pearson Education
2. https://www.postgresql.org/docs/