Introduction:

We have implemented efficient data layout & retrieval strategy using PostgreSQL. PostgreSQL, or postgres, is a popular database management system that can organize and manage the data associated with websites or applications. In order to do have an efficient data layout a couple of steps have been implemented. They are:

1. Replication
2. Partitioning

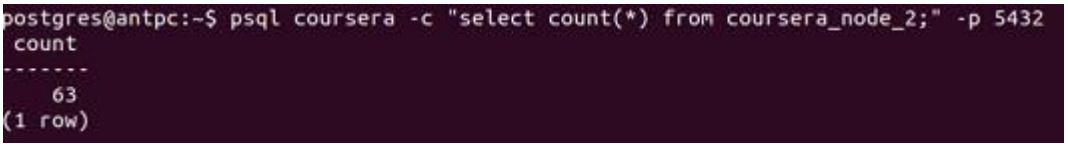
Approach:

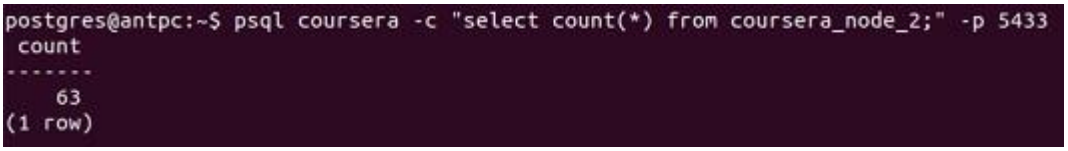
Replication:

Replication is a means of copying database information to a second system in order to create high availability and redundancy. The master/slave database replication is a process of copying (syncing) data from a database on one server (the master) to a database on another server (the slaves). The main benefit of this process is to distribute databases to multiple machines, so when the master server has a problem, there is a backup machine with the same data available for handling requests without interruption. It can be used for backup purposes and to provide a high availability database server.

In order to perform replication, the steps mentioned in the Appendix A were implemented.

The master server has been configured to run on the port 5432 whereas the slave server is configured to run on the port 5433. The screenshots below show the queries run on both ports to show that replication has been achieved with the desired effect.





Partitioning:

As table size increases with data load, more data scanning, swapping pages to memory, and other table operation costs also increase. Partitioning may be a good solution, as It can help divide a large table into smaller tables and thus reduce table scans and memory swap problems, which ultimately increases performance.

Partitioning helps to scale PostgreSQL by splitting large logical tables into smaller physical tables that can be stored on different storage media based on uses. Users can take better advantage of scaling by using declarative partitioning along with foreign tables using postgres.

Benefits of partitioning

- PostgreSQL declarative partitioning is highly flexible and provides good control to users. Users can create any level of partitioning based on need and can modify, use constraints, triggers, and indexes on each partition separately as well as on all partitions together.
- Query performance can be increased significantly compared to selecting from a single large table.
- Partition-wise-join and partition-wise-aggregate features increase complex query computation performance as well.
- Bulk loads and data deletion can be much faster, as based on user requirements these operations can be performed on individual partitions.
- Each partition can contain data based on its frequency of use and so can be stored on media that may be cheaper or slower for low-use data.

Most benefits of partitioning can be enjoyed when a single table is not able to provide them. So, we can say that if a lot of data is going to be written on a single table at some point, users need partitioning. Apart from data, there may be other factors users should consider, like update frequency of the data, use of data over a time period, how small a range data can be divided, etc. With good planning and taking all factors into consideration, table partitioning can give a great performance boost and scale your PostgreSQL to larger datasets.

A range partition is created to hold values within a range provided on the partition key. Both minimum and maximum values of the range need to be specified, where minimum value is inclusive and maximum value is exclusive. Range partition does not allow NULL values. 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. Range partitioning maps data to partitions based on ranges of values of the partitioning key that you establish for each partition. This is what we have implemented in each node under slave. We have partitioned based on the course rating.

Partitioning can improve scalability, reduce contention, and optimize performance. It can also provide a mechanism for dividing data by usage pattern.

The first query is run on the master table with no distributed system & partitioning. The cost to run the query is 644.83ms. The second query is run on the master table with distributed system & partitioning. The cost to run the query is 228.07ms.

On comparison, the cost to run the same query on table without partitioning was almost 3 times when compared to the cost of the query run on table with partitioning. This gives the us an idea of how efficient the distributed architecture is.

On further analysis of the explanation of query on partitioned system, we can see that since rating was asked between 4.1 & 4.3 the search happened only in good_rating_nodes as compared to search on all the table. This helps in achieving the required efficiency.

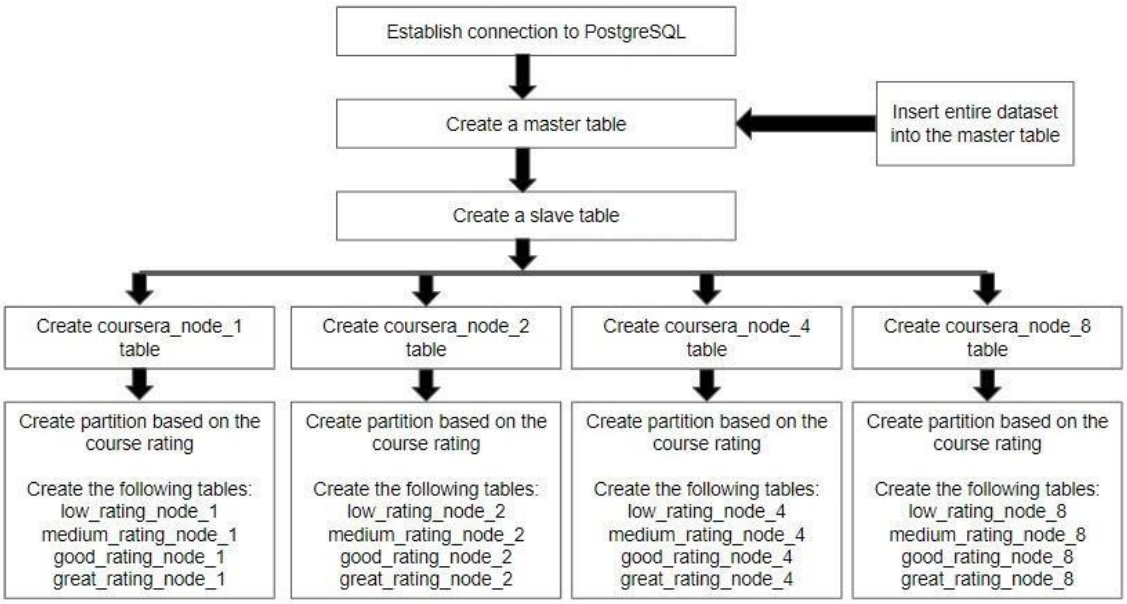
```
command = """EXPLAIN select course_name, university, course_rating from master_data where course_rating >= 4.1 AND cou
explanation = run_queries(command)
for row in explanation:
    print(row[0])
```

```
Seq Scan on master_data  (cost=0.00..644.83 rows=371 width=77)
  Filter: ((course_rating >= '4.1'::double precision) AND (course_rating <= '4.3'::double precision))
```

```
command = """EXPLAIN select course_name, university, course_rating from coursera_master where course_rating >= 4.1 AND
explanation = run_queries(command)
for row in explanation:
    print(row[0])
```

```
Append  (cost=0.00..228.07 rows=360 width=77)
  ->  Seq Scan on good_rating_node_1 coursera_master_1  (cost=0.00..184.12 rows=357 width=77)
        Filter: ((course_rating >= '4.1'::double precision) AND (course_rating <= '4.3'::double precision))
  ->  Seq Scan on good_rating_node_2 coursera_master_2  (cost=0.00..14.05 rows=1 width=72)
        Filter: ((course_rating >= '4.1'::double precision) AND (course_rating <= '4.3'::double precision))
  ->  Seq Scan on good_rating_node_4 coursera_master_3  (cost=0.00..14.05 rows=1 width=72)
        Filter: ((course_rating >= '4.1'::double precision) AND (course_rating <= '4.3'::double precision))
  ->  Seq Scan on good_rating_node_8 coursera_master_4  (cost=0.00..14.05 rows=1 width=72)
        Filter: ((course_rating >= '4.1'::double precision) AND (course_rating <= '4.3'::double precision))
```

Data Flow:



The above table gives us a good idea of how the partitioning is implemented in the system to ensure that the data is distributed based on 2 factors:

1. Duplication factor
2. Coursera rating

The master table would be partitioned by the column "size" which reflects the number of times a row is present in the actual data table. There are 4 tables into which the master table is partitioned based on the duplication factor.

- coursera_node_1: will have data which has come only once in the data
- coursera_node_2: will have data which is present twice in the data
- coursera_node_4: will have the data occuring 3 or 4 times in the data
- coursera_node_8: will have the data occuring 4 to 8 times in the data

Each of the node table is then partitioned into 4 based on the rating received.

- low_rating_node_1: will have data where courses have ratings between 0 & 2 in the rows which have occured only 1 in the master data
- medium_rating_node_1: will have data where courses have ratings between 2 & 4 in the rows which have occured only 1 in the master data
- good_rating_node_1: will have data where courses have ratings between 4 & 4.51 in the rows which have occured only 1 in the master data.
- great_rating_node_1: will have data where courses have ratings between 4.51 & 5.1 in the rows which have occured only 1 in the master data.

Note: values taken by partitioning dont take into consideration the upper limit. Hence if limit is mentioned as 5, 5 wont be considered in the partitioning. Hence, upper limit is mentioned as .1 more than the intended limit.

Appendix

Appendix A: Replication Steps

1. Install initial PostgreSQL 10 cluster and verify it exists
   sudo pg_lsclusters
2. create a second postgres cluster
   sudo pg_createcluster 10 replica1
   sudo pg_ctlcluster 10 replica1 status
   sudo systemctl status postgresql@10-main
3. create archive directories for both clusters
   sudo -H -u postgres mkdir /var/lib/postgresql/pg_log_archive/main
   sudo -H -u postgres mkdir /var/lib/postgresql/pg_log_archive/replica1
4. ## Configure Main Cluster (Primary / Master) # edit configuration file
   sudo nano /etc/postgresql/10/main/postgresql.conf
   wal_level = replica
   wal_log_hints = on
   archive_mode = on # (change requires restart)
   archive_command = 'test ! -f /var/lib/postgresql/pg_log_archive/main/%f && cp %p /var/lib/postgresql/pg_log_archive/main/%f'
   max_wal_senders = 10
   wal_keep_segments = 64
   hot_standby = on

5. edit host based access file
   sudo nano /etc/postgresql/10/main/pg_hba.conf
   local replication rep_user trust
6. create replication user
   sudo -H -u postgres psql -c "CREATE USER rep_user WITH replication;"
7. restart the main cluster
   sudo systemctl restart postgresql@10-main
8. Configure Replica1 Cluster # edit configuration file
   sudo nano /etc/postgresql/10/replica1/postgresql.conf
   wal_level = replica
   wal_log_hints = on
   archive_mode = on # (change requires restart)
   archive_command = 'test ! -f /var/lib/postgresql/pg_log_archive/replica1/%f && cp %p /var/lib/postgresql/pg_log_archive/replica1/%f'
   max_wal_senders = 10
   wal_keep_segments = 64
   hot_standby = on
9. edit host based access file
   sudo nano /etc/postgresql/10/replica1/pg_hba.conf
   local replication rep_user trust
10. Setup Replica1 Cluster Replication #remove replica1 existing database files
    sudo su - postgres
    rm -rf /var/lib/postgresql/10/replica1
11. sync replica1 with main cluster
    pg_basebackup -D /var/lib/postgresql/10/replica1 -U rep_user -w -P -R # -X stream
12. configure recovery.conf
    nano /var/lib/postgresql/10/replica1/recovery.conf

restore_command = 'cp /var/lib/postgresql/pg_log_archive/replica1/%f %p'
recovery_target_timeline = 'latest'
standby_mode = 'on'
primary_conninfo = 'user=rep_user passfile=''/var/lib/postgresql/.pgpass'' host''/var/run/postgresql''
port=5432 sslmode=prefer sslcompression=1 krbsrvname=postgres target_session_attrs=any'
archive_cleanup_command = 'pg_archivecleanup /var/lib/postgresql/pg_log_archive/replica1 %r'

13. start replica cluster and verify in sync
    sudo pg_ctlcluster 10 replica1 start
    tail -n 100 /var/log/postgresql/postgresql-10-replica1.log
14. Verify Replica1 Cluster In Sync # create database with some data
    sudo su - postgres
    psql -c "create database test;" -p 5432
    psql test -c "
    create table posts (
    id integer,
    title character varying(100),
    content text,
    published_at timestamp without time zone,
    type character varying(100)
    );

    insert into posts (id, title, content, published_at, type) values
    (100, 'Intro to SQL', 'Epic SQL Content', '2018-01-01', 'SQL'),
    (101, 'Intro to PostgreSQL', 'PostgreSQL is awesome!', now(), 'PostgreSQL');
    "
15. verify data has been replicated on replica1
    *psql test -c "select * from posts;" -p 5433*
16. stop main cluster (simulate failure condition)
    sudo systemctl status postgresql@10-main
17. promote replica1
    sudo pg_ctlcluster 10 replica1 promote
18. verify replica1 is now a master / primary cluster
    tail -n 100 /var/log/postgresql/postgresql-10-replica1.log
    psql test -c "insert into posts (id, title, content, type) values
    (102, 'Intro to SQL Where Clause', 'Easy as pie!', 'SQL'),
    (103, 'Intro to SQL Order Clause', 'What comes first?', 'SQL');" -p 5433
    *psql test -c "select * from posts;" -p 5433*