CSE 512: Distributed Database Systems

Project: Distributed Database System for a Smart Building

Part – 4

Pgpool-II

Pgpool-II is a middleware solution for PostgreSQL that provides features to improve database scalability, load balancing, and high availability. It acts as a proxy between client applications and PostgreSQL database servers, offering various functionalities to optimize database performance and manage distributed transactions effectively.

Key Features:

1. **Connection Pooling:**
   - Pgpool-II manages a pool of database connections, reducing the overhead of establishing and closing connections for each client request. This enhances database performance by reusing existing connections, particularly beneficial in scenarios with a high number of client connections.

2. **Load Balancing:**
   - Load balancing is crucial in distributed database environments to distribute incoming queries across multiple PostgreSQL servers. Pgpool-II intelligently distributes queries among the backend servers, preventing overloading of individual nodes and ensuring optimal resource utilization.

3. **Query Parallelization:**
   - Pgpool-II can parallelize read queries across multiple database nodes, improving read performance, and reducing the overall query execution time. This is especially advantageous in scenarios where read-intensive workloads are common.

4. **Connection Pooling and Transaction Handling:**
   - For distributed transactions, Pgpool-II offers transaction pooling, managing the coordination of transactions across multiple database servers. This ensures data consistency and reliability, even in the presence of multiple database nodes.

5. **High Availability:**
   - Pgpool-II supports high availability configurations through features like automatic failover and online recovery. In the event of a database server failure, Pgpool-II can redirect traffic to healthy servers, minimizing downtime and ensuring continuous service availability.

We require Pgpool to showcase distributed transactions. Pgpool-II's transaction pooling ensures that distributed transactions are coordinated efficiently across multiple database servers. This contributes to maintaining the ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions even in a distributed setup.

1. **Atomicity and Durability**: By managing transactions across multiple database servers, pgpool-II ensures that all parts of a transaction are executed atomically and that the changes persist even after a system failure.
2. **Consistency**: pgpool-II maintains database consistency by distributing queries in a controlled manner and by facilitating accurate replication across servers.
3. **Isolation and Concurrency Control**: pgpool-II enables setting appropriate isolation levels for transactions, ensuring that each transaction operates independently. This is crucial for preventing issues like dirty reads, non-repeatable reads, and phantom reads, which are key for effective concurrency control.

Design

1. Two PostgreSQL server – one master and other slave.

2. Both the servers are managed by pgpool-II.

3. Python code to query the data which connects to pgpool-ll server instead of individual servers.

Steps to setup

Postgresql Replication setup:

You need to configure two separate instances of PostgreSQL to run on different ports and with different data directories.

Configure the Master Server

1. Initialize the Master Data Directory:

    a. sudo mkdir -p /var/lib/postgresql/16/master

    b. sudo chown postgres:postgres /var/lib/postgresql/16/master

    c. sudo -u postgres /usr/lib/postgresql/16/bin/initdb -D /var/lib/postgresql/16/master

2. Edit postgresql.conf for the Master:

    a. Path: /var/lib/postgresql/16/master/postgresql.conf

    b. Set listen_addresses to 'localhost'

c. Set port to 5432.

d. Configure wal_level to replica.

e. Set max_wal_senders to 3 or more.

f. Enable hot_standby to on.

3. Edit pg_hba.conf for the Master:

   a. Path: /var/lib/postgresql/16/master/pg_hba.conf

   b. Add replication permissions:
   host replication all 127.0.0.1/32 md5

Start the Master Server

1. sudo -u postgres /usr/lib/postgresql/16/bin/pg_ctl -D /var/lib/postgresql/16/master start

Configure the Slave Server

1. Initialize the Slave Data Directory:

   a. sudo mkdir -p /var/lib/postgresql/16/slave

   b. sudo chown postgres:postgres /var/lib/postgresql/16/slave

2. Use pg_basebackup to clone the master:

   a. sudo -u postgres pg_/usr/lib/postgresql/16/bin/basebackup -h localhost -p 5432 -D /var/lib/postgresql/16/slave -U postgres -v -P --wal-method=stream

3. Create Recovery Configuration for the Slave:

   i) For PostgreSQL 12+, use standby.signal and postgresql.conf

   ii) Set primary_conninfo to 'host=localhost port=5432 user=replication_user'.

Start the Slave Server:

1. sudo -u postgres /usr/lib/postgresql/16/bin/pg_ctl -D /var/lib/postgresql/16/slave start

Setting up pgpool=II :

1) Edit the configuration for pgpool-II:

   a. PATH: /etc/pgpool2/pgpool.conf

```
listen_addresses = '*'

port = 9999

backend_hostname0 = 'localhost'

backend_port0 = 5432

backend_weight0 = 1

backend_data_directory0 = '/var/lib/postgresql/12/master'

backend_flag0 = 'ALLOW_TO_FAILOVER'

backend_hostname1 = 'localhost'

backend_port1 = 5433

backend_weight1 = 1

backend_data_directory1 = '/var/lib/postgresql/12/slave'

backend_flag1 = 'ALLOW_TO_FAILOVER'

load_balance_mode = on

replication_mode = off

health_check_period = 10

health_check_timeout = 5

health_check_user = 'pgpool_check_user'

health_check_password = 'check_password'

num_init_children = 32

max_pool = 4
```

2) Start pgpool-II server:

```
pgpool -n -f /etc/pgpool2/pgpool.conf
```

Implement ACID Compliant Transactions

This report discusses the implementation of ACID-compliant distributed transactions in a Smart Building System using PostgreSQL. Two scenarios are showcased: one illustrating a successful transaction, and another demonstrating rollback scenarios for Atomicity, Consistency, Isolation, and Durability (ACID) properties.

Successful Transaction Scenario

In the successful transaction scenario, the system successfully handles a user's movement from an occupied room to an available room. Key steps in the transaction include:

1. **Data Retrieval:**
    - Identification of the user occupied room, and available room.

2. **Data Updates:**
    - Updating the room statuses to 'Available' and 'Occupied' accordingly.
    - Recording the user's exit from the occupied room and entry into the available room in the access logs.

3. **Commit:**
    - Committing the transaction to ensure the changes are persistent.

```
Showcasing successful transaction block for Atomicity, Consistency, Isolation, Durability
user with user_id 89 moving from room 3 to room 2
Room status of 3 is Available
Room status of 2 is Occupied
```

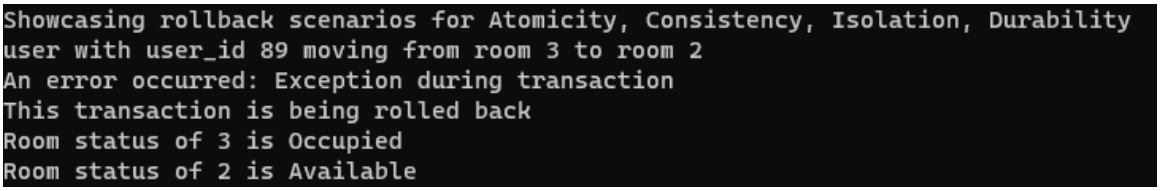
Rollback Scenario

In the rollback scenario, an intentional exception is raised during the transaction to simulate an error. The system handles this by rolling back the changes to maintain data consistency. Key steps include:

1. **Data Retrieval:**
    - Like the successful scenario, identifying the user, occupied room, and available room.

2. **Data Updates:**
    - Updating the room statuses.
    - Recording the user's exit from the occupied room.
    - Intentional exception raising.

3. **Rollback:**
    - Rolling back the transaction to maintain data consistency.



After both scenarios, the system checks and prints the room statuses to verify the impact of the transactions. This demonstrates the system's ability to maintain data integrity even in the face of errors.

The implementation of ACID-compliant distributed transactions in the Smart Building System using PostgreSQL ensures the reliability and consistency of data updates. The success and rollback scenarios showcase the system's ability to handle both normal operations and exceptional cases, emphasizing the importance of ACID properties in maintaining data integrity.

Concurrency Control Mechanism

Locking Mechanism Theory:

Concurrency control is essential in a multi-user database system to ensure that transactions are executed in a manner that preserves the consistency of the database. Locking is a widely used mechanism to control access to shared resources in a multi-user environment.

- **Lock Types:**
  - **Shared Lock (S):** Allows multiple transactions to read a resource simultaneously but prevents any of them from writing.
  - **Exclusive Lock (X):** Grants exclusive access to a resource, preventing any other transactions from reading or writing to it.

- **Lock Granularity:**
  - **Table-level Locks:** Locks an entire table.
  - **Row-level Locks:** Locks specific rows within a table.
  - **Page-level Locks:** Locks a page, which is a contiguous block of data.

- **Deadlock Prevention:**
  - Techniques such as wait-die and wound-wait are employed to prevent and resolve deadlocks.

Implementation:

The provided Python script demonstrates concurrency control mechanisms using the locking approach. The script defines two functions:

1. **increase_capacity_with_lock:**
   - Begins a transaction.
   - Optionally acquires an exclusive lock on the 'rooms' table.
   - Retrieves the current occupancy limit of a room, increases it, simulates a delay, and updates the database.
   - Commits the transaction.

2. **distributed_transaction_with_lock:**
   - Creates two separate processes, each invoking the **increase_capacity_with_lock** function with different parameters.
   - One process increases the room capacity by 5, and the other by 10, concurrently.

Race Conditions Demonstration:

1. **Without Lock (Race Conditions):**
   - The script showcases a scenario where concurrent transactions occur without locking.

- Simulates a race condition by allowing both processes to update the room capacity without exclusive locks.
- Race conditions occur as both processes can update the room capacity simultaneously, leading to inconsistent data.

```
Showcasing Race Conditions without lock during concurrent transactions
Increasing room capacity by 5 from 5 to 10
Increasing room capacity by 10 from 5 to 15
```

2. **With Lock (Resolved Race Conditions):**
    - Demonstrates a scenario where exclusive locks are acquired before updating the room capacity, preventing race conditions.
    - Serialization is achieved, ensuring that one transaction completes before the other starts.
    - The exclusive lock prevents concurrent access, resolving race conditions and ensuring data consistency.

```
Showcasing resolved Race Conditions by serialization with lock during concurrent transactions
EXCLUSIVE Lock acquired by Process 1 for table rooms
Increasing room capacity by 5 from 15 to 20
EXCLUSIVE Lock released by Process 1 for table rooms
EXCLUSIVE Lock acquired by Process 2 for table rooms
Increasing room capacity by 10 from 20 to 30
EXCLUSIVE Lock released by Process 2 for table rooms
```

The implementation highlights the importance of locking mechanisms in preventing race conditions during concurrent transactions. The script demonstrates how serialization through locking ensures consistent and predictable behavior in a multi-user database environment.

Compiled By:

Distributed Nerds

Ahraz Rizvi
Keshava Rajavaram
Poorvik Dharmendra
Sahara Abdi