DESIGN A DISTRIBUTED DATABASE FOR AN E-COMMERCE PLATFORM TO HANDLE ORDER PROCESSING

Exp. No. : 5
Date : 08/05/2024

AIM:

To Design a distributed Database for an E-Commerce Platform to Handle order processing.

PROCEDURE:

i) understanding Entities :-

*) Identity the main components: order, customers, and products.

*) Decide how you'll typically search for information, like filtering orders by customer or product.

2) choose Key structure:

*) pick a key, such as customer ID or product ID, for data retrived.

*) If you often search for orders by a specific customer.

3) streamlining Data Management:-

*) Keep related data together to simply retrieved.

*) For example, include product details directly in the order table.

4) Ensuring Data Redundancy and Consistency:

*) Determine how many copies of your data are needed for backup.

QUERY & OUTPUT:

5) Testing and Optimization:

*) use tools like cqlsh to implement and evaluate your data organization plan.

Query & Output :-

```
from cassanda-cluster import cluster
from cassandra.auth import PlainText Auth Provider
import uuid

cloud-config = {
        'secure_connect_bundle' : "D:/secure-connect-your-cluster.zip'

}

auth_provider = PlainText Auth Provider ('username', 'password')
cluster = Cluster (cloud = cloud_config , auth-provider=auth_provide
session = cluster . connect()

Session . execute ("""
            CREATE KEYSPACE IF NOT EXISTS ecommerce.
            WITH REPLCATION={'class' : 'simple strategy', 'replication_fact
                                                                       :3}
""")

Session. set _keyspace ('ecommerce')
Session. execute ("""""
    Create table if not exists orders(
        customer_id UUID,
        order_id UUID,
        product_id UUID,
        product_name TEXT,
        quantity INT,
        order_date Timestamp,
        Primary key (customer_id ,order_id)
    )
""""")

def create_order (customer_id, product_id, product_name, quantity, order_date):
    order_id = uuid . uuid4()
    Session . execute (""  ""
        Insert into order (customer_id, order_id, product_id, product name,
                          quantity, order_date )
        values (%s, %s, %s, %s, %s, %s)
    "" " ", (customer_id, order_id, product_id, product_name, quatity, order_date

def get_order_by_customer (customer_id):
    row = session. execute ("""" Select * from order where customer_id = %s"""",
    (customer_id ))
    for row in rows:
        print(f "Order ID: {row.order_id}, Prodct ID: {row.product_id}, Product Name:
            {row.product_name}, Quatity: {row.quatity}, Order Date: {raw.order_date
```

```
def  get_orders_by_product(product_id):
     row = session.execute(""" Select * From orders Where product_id=%s
     """, (product_id))
     for row in rows:
         print(f"Order ID: {row.order_id}, customer ID: {row.customer_id},
         Quatity: {row. quatity}, Order Date: {row.order_date}")

If __name__ = "__main__":
     create_order(uuid.uuid(), uuid.uuid(), "Product A", 2, '2024-05-15')
     customer_id = uuid.uuid4()
     get_orders_by_customer(customer_id)
     product_id = uuid.uuid4()
     get_orders_by_product(product_id)

cluster.shutdown()
```

Output :-

Order 4200 processed successfully.

RESULTS:

Therefore the distributed database for an e-commerce application handle order processing in Apache cassandra is created and executed successfully.

DEVELOP AN IN-MEMORY CACHING SOLUTION USING REDIS FOR A CONTENT PUBLISHING PLATFORM (BLOG).

Exp. No. : 6
Date : 10/05/2024

AIM:

To develop an in memory caching solution using Redis for a content publishing platform (Blog).

PROCEDURE:

1. Determine commonly accessed, unchanging content like posts or user profiles.

2. Establish suitable expiration periods based on how frequently the data is updated.

3. Establish a connection to the Redis server using a client library.

4. Prioritizing checking the cache for data availability before resorting to fetching from the database.

5. If cached data exists, utilize it; otherwise, retrieve from the database and store in the cache with a defined expiration.

6. Invalidate cache entries upon database updates to ensure data consistency.

7. Keep track of cache hit rates and adjust expiration times accordingly for optimal performance.

8: Implement eviction policies such as least recently used (LRU) to mange cache size efficiently.

9. Consider cache invalidation mechanisms for user actions that impact cached data.

10. Optionally, explore Redis Pub / Sub for facitating real-time cache updates across multiple servers.

QUERY & OUTPUT:

```
import json
import redis

class BlogContentCache:

    def __init__(self, redis_host = 'Local host', redis_port=6379,
                         redis_db=0):
        self.redis_client = redis.StrictRedis(host=redis_host,
                                      port=redis_port,
                                      db=redis_db)

    def get_post(self, post_id):
        ~~ch~~ cached_post = self.redis_client.get(f'post:{post_id}')
        if cached_post:
            print("Retrieving post from cache.")
            return json.loads(cached_post.decode('utf-8'))
        else:
            post = {'post_id': post_id, 'title': f'Title of Post
                   {post_id}', 'content': f'Content of {post_id}'}
                                                                 }
        Self.redis_client.set(f'post:{post_id}', json.dumps(post)
        print("Fetching post from the database and caching.")
        return post
```

```
if __name__ == "__ ~~kau~~ main__":
        blog_cache = BlogContentCache()
        post_id_1 = 1
        post_id_2 = 2
        post_1 = blog_cache.get_post(post_id_1)
        printf(f"Post :{post_1}")
        cached_post_1 = blog_cache.get_post(post_id_1)
        printf(f"Cached Post : {cached_post_1}")
        post_2 = blog_cache.get_post(post_id_2)
        print(f"Post : {post_2}")
        cached_post_2 = blog_cache.get_post(post_id_2)
        print(f"Cached post : {cached_post_2}")
```

Output :-

```
Fetching post from the database and caching
Post : { 'post_id': 1, 'title': 'Harry Potter', 'content': 'movie' }

Retrieving post from ~~the~~ cache.
Cached post : { 'post_id': 1, 'title': 'Harry potter', 'content': 'movie' }

Fetching post from the database and caching.
Post : { 'post_id': 2, 'title': 'Harry Book', 'content': "cricketer" }

cached post : { "post_id': 2, 'title': 'Harry Book', 'content': 'cricketer' }.
```

DEVELOP A SECURE RDBMS SOLUTION FOR A FINANCIAL TRANSACTIONS SYSTEM.

Exp. No. : 7
Date : 15/05/2024

AIM:

To develop a secure RDBMS solution for a financial transactions system.

PROCEDURE:

1. Set up who can do what in the system by creating different roles, giving them specific abilities and assigning those roles to users.

2. Make sure passwords are strong and safe by following strict rules and storing them in a protected way that includes adding extra security & random information.

3. Add extra security by requiring users to confirm their identities in more than one way when logging in.

4. Find and protect information that's particularly sensitive, like personal details or financial records.

~~QUERY & OUTPUT~~:

5. Keep data safe when it's stored by scrambling it

6. Use a secure way to send by scrambling it, ~~either~~, user's device and the server, making sure nobody can snoop it.

7. Think about adding another layer of protection for data while it's traveling from one place to another, in case someone tries to intercept it.

8. Keep a record of who's trying to access the system, what they make, to help spot any problems and hold people accountable.

program:-

```
from flask import Flask, request , jsonify
from flask_sqlalchemy import SQLAlchemy
from flask _bcrypt import Bcrypt
from sqlalchemy import Column, Interger, string , create_engine
from sqlalchemy . ext . declarative import declarative _base
from sqlalchemy . orm import sessionmaker
from flask_cors import CORS

app = Flask (__name__)
CORS (app)

app. config ['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///banking.db'
app . config ['SQLALCHEMY_TRACK_MODIFICATION'] = False
app. config ['SECRET_KEY'] = '123987'
app. config ['BCRYPT_LOG_ROUNDS'] = 12

db = SQLAlchemy(app)
bcrypt = Bcrypt (app)
Base = declarative_base()
```

```
class User (Base):

    __table name__ = 'users'
     id = Column (Integer, primary_key = True)
    username = Column (string (50), unique = True , nullable = False)
    password = Column ( string(60), nullable = False)
    account_number = Column (string (20) , unique = true, nullable = False,
                    info {'encrypt': 'rest'})

engine = create_engine('sqlite:///banking.db', echo = True)
Base . metadata . create_all (bind = engine)
@ app . route ('/login', methods=['POST'])
def login():
    data = request.get_json()
    username = data . get ('username')
    password = data . get ('password')
    user = User . query . filter_by (username= username) . first()
    if user and bcrypt . check_password_hash (user . password, password):
        return jsonify ({'message' : 'Login Successful'})
    else:
        return jsonify ({'message' : 'Invalid credentials'}), 401

@ app . route ('/register', method: ['POST'])
def register():
    data = request. get_json()
```

```
username = data.get('username')
password = data.get('password')
hashed_password = bcrypt.generate_password_hash(password).decode('utf-8')
new_user = User(username = username, password = hashed_password, ac no: 12345678)
db.session.add(new_user)
db.session.commit()
return jsonify({'message': 'User registered successfully'})

if __name__ == '__main__':
    app.run(debug = True)
```

Output:-

The flask application is run at the port url http://127.0.0.1:5000/.
It will respone based on the resquest.

RESULTS:

Therefore the secure RDBMS Solution for a banking finacial transactions system is created and executed succesfully.