Building Scalable Distributed Systems

# Assignment 2

Git Repository: <https://github.com/akksshah/building-scalable-distributed-systems>

# Server Design Description

The server receives the request from the client in the form of Json. It converts the Json string to the PurchaseOrder object that, the client requests the server to persist in the MySQL database. The conversion of json string to PurchaseOrder object is done using Jackson library. The server then tries to save the object by calling on the data access object (DAO)’s method: savePurchaseOrder(order), in our case it is the instance of the class PurchaseTransaction.

The DAO, requests connection from the ConnectionUtility class. The ConnectionUtility class maintains a pool of database connections. It provides the DAO with Connection objects as and when it requests for the same. For the single server tests, the connection pool size for the Database was set to 60. For the load balanced system, each server was granted 15 connections to the database access pool. The DAO on getting a connection would initiate a transaction, it would then try to insert the purchaseOrder into the database. On successful insertion, it would then commit to the database. If an exception would occur, it would roll back the transaction. In such case, where the insert failed, the server would send the client a SC\_NOT\_OK status code. The client can then retry the request again.

Database design.

Initially I went with a design which had two tables, the first stored the storeId, customerId and date (with an auto generated primary key called id). The second table stored the itemId, numberOfItems purchased of that Item and the foreign key, the “id” from the first table. Having a normalized form was good for querying, however, it compromised on write speeds.

I then switched to a single table where each row stored storeId, customerId, date, itemId, numberOfitems purchased for that time. Moving to this database schema gave a very high throughput compared to the first schema. This schema although is not efficient, it yields high write throughputs.

The first database schema started to use burst credits for the same to keep up with the requests, on the other hand, the second schema scaled pretty fast and easily and did not use any burst credits for the same.

Connection Pools:

Hibernate connection pools were way slow compared to Hikari Connection Pools. Hikari Connection Pool provided almost 10 time the throughput compared to the Hibernate connection pools.

# Single Server Test:

32 Threads

Text

Description automatically generated

64 Threads

Text

Description automatically generated

128 Threads

Text

Description automatically generated

256 Threads

Text

Description automatically generated

Performance graph

# Load Balanced Server Test

32 Threads

Text

Description automatically generated

64 Threads

Text

Description automatically generated

128 Threads

Text

Description automatically generated

256 Threads

Text

Description automatically generated

512 threads

Text

Description automatically generated

1024 threads

Text

Description automatically generated

Performance graphChart, line chart, scatter chart

Description automatically generated

Note: The throughput has been scaled with 1 point referring to 10 points on the actual scale.