# IMT 563 - Advanced Topics in Relational and Non-Relational Databases

## Performance Experiment: Bulk Ingest


## Group 7
## (Aditya Chatterjee, Akshay Khanna, Sahil Aggarwal, Shreya Sabharwal)


# Abstract


This document outlines a set of experiments in which we compare the performance of different methods of bulk inserts into a PostgreSQL database hosted on AWS RDS, using client written in Node.js.

Our findings indicate that bulk inserts (inserting multiple rows into the database using a single statement) are significantly (a factor of ~100, for large inserts) faster than using individual queries - even when the client was sending inserts asynchronously (inserts were sent serially but insert n+1 was sent before receiving confirmation that insert n was successful).

We found that the creation of successive indexes resulted in a small drop in throughput during bulk inserts, but this was not consistent upon increasing the number of non-clustered indexes. This partially supported our assumption that indexing would affect insert speeds negatively.

# Hypothesis

We compare the time taken to ingest bulk data into a ‘hotel reservation’ database by performing the following experiments with three different indexing strategies:

1. Inserting 10,000 records with one insert statement per record and no indexes.
2. Inserting data in batches of different sizes (1, 5, 10, 20, 50, 100, 1000) without any indexes.
3. Inserting data in batches of different sizes with a clustered index on the primary key.
4. Inserting data in batches of different sizes with a secondary non-clustered index on any of the columns and a clustered index on the primary key.
5. Using COPY command to insert data into a table from a file with different indexing strategies.
6. Inserting 10000 records with additional non-clustered b-tree indexes.
7. Calculating network latency by executing select queries.
8. Inserting data from another table and comparing throughput of different indexing strategies.

We believe that using bulk inserts of higher batch size (populating the database using a single insert statement) will perform the better in terms of insert throughput (records inserted per unit time). Since the connection has to be established once (resulting in lower network latency) and the database server is hit with a single request to process, the bulk inserts should ideally be faster than one insert per record.

We expect that each additional index will result in a reduction in insert throughput because of increased index updation overhead.

# Assumptions and Setup

1. **Purging Between Inserts:** We purge the table before performing each experiment, ensuring the same control setup.
2. **Network Latency:** Our experiments are designed under the assumption that network latency (the sum of the time it takes for our HTTP request packets to reach our database server from the client and the amount of time it takes for response packets to reach our client from the database server) is roughly consistent during inserts, and is too small to significantly affect our findings, even with some jitter (change in latency between successive inserts), particularly when the number of records being inserted is large.

We modeled a hotel reservation system with entities such as hotels, rooms, reservations and guests. Below are the tables used:

| Srl No. | Table | Description | Primary Key | Foreign Key |
|---|---|---|---|---|
| 1 | Reservation | Contains all the reservations made by the guests. | reservationID | hotelid, roomid, guestid |
| 2 | Guest | Contains details of the guests staying in the hotels such as First & Last Name, Address, City, Zip Code | guestID | N/A |
| 3 | Hotel | Contains data about the hotels such as Hotel Name, Address, City, Zip Code. | hotelID | N/A |
| 4 | Room | Contains data for rooms of all hotels such as Room Number, Hotel ID and Room Type. | roomID | N/A |

All our experiments for this assignment are done on the ***guest*** table as it is one of the largest tables in our schema and does not have any foreign-key dependencies on any other table.

We have used PostgreSQL as our database primarily because it is open source and widely used in the industry. It is hosted on the Relational Database Service (RDS) provided by the Amazon Web Services (AWS). AWS has native support for PostgreSQL databases through its RDS service. The AWS instance class we are using is db.t2.micro. Inbound security rules were created so that every member of the team can connect to the DB instance from their local systems. Pgadmin was used as the interface to administer the database.

Our client program was run on a laptop (Intel i5 8250U@1.60GHz, 8CPUs, 8192MB RAM, running Windows 10) over the University of Washington's campus internet service (~54Mbps download, ~114Mbps upload when the tests were conducted). Our client was written using Node.js, which follows an asynchronous, non-blocking IO model.

The client program can be found at the GitHub repository [1].
The program was written as a command line tool that allows us to denote specifics of the insert (separate queries or in bulk, which table, number of records, whether or not to save metrics) as flags in the command line.

For example:
*$ ./bin/run addRecords -g 10000 -b -s*
//add 10000 records to the guest (-g) table using bulk (-b) inserts and save (-s) the performance metrics for this insert
*Attempting to BULK insert 10000 guest records into database*
*Insertion took 1.325*
*$ ./bin/run purge* //purge the database to reset for a new experiment
*purging database...*
*...done*

Our program makes use of the following Node.js packages:

- **node-postgres**: A promise-based PostgreSQL client for Node.js used to query our database [2]
- **faker.js**: A Node.js package with enables the rapid generation of a variety of synthetic data [3]
- **perfy**: A timing and performance tool [4]
- **oclif:** A framework to create command line interfaces in node.js [5]

The performance metrics (calculated by the client program using *perfy* package) of all our experiments have been recorded in a log table, *results_log* in order to track the performance results, run the experiments multiple times and compare the results easily. The table helped us refer back to the results in case of any discrepancies and abnormalities and was exported to create visualizations in tableau.

An additional table was created in order to run additional experiments - 9(a) and 9(d). The table, *guest_copy* is a replica of *guest* table with same set of columns. Different indexing strategies were performed on this table based on the experiments.

# Main Results

## Observations during standard (non-batch) inserts

Our ‘single insert per record’ approach turned out not to be viable for use in the wild - inserting 10000 records took roughly 933 seconds (around fifteen minutes) - we ran this multiple times and results were consistent.

We tried two approaches to this ‘single insert per record’ experiment - initially, we waited for a response from the database server between each successive request (i.e., there was only ever one insert request in transit to/from the database, pending completion). We also tried an ‘asynchronous’ approach, where all 10000 insert requests were sent at the same time, non-sequentially, and our client calculated the amount of time it took for all insert requests to be acknowledged by the database server.

While both approaches resulted in similar throughput values, the second, asynchronous approach, resulted in tangible performance degradations on the database - for example, our pgAdmin management interface was unusable at this time. It is possible that having a large number of concurrent HTTP requests hitting the database server resulted in a degradation of its ability to respond to messages from pgAdmin (which we assume is via HTTP/SSH or another generic application layer protocol).

## Atomic vs. Batch Inserts

As we had hypothesized, increasing the batch size resulted in an increase in throughput for insertions. However, the magnitude of the speedup observed was far beyond our initial expectations. For example, inserting 10,000 records in a batch size of 5 with no indexes took ~182 seconds, as opposed to ~933 seconds for atomic inserts, which is a 5x speed up in terms of insertion time. Further, time decreased as we increased the batch size for bulk inserts. For a batch size of 1000, the time taken was ~3 seconds, which is ~30x faster than atomic inserts. There are two factors that may contribute to this phenomenon, although measuring the effect of each would require additional experimentation:

1. The PostgreSQL database may be highly optimized to parse large queries, but less so to handle a large number of incoming HTTP requests. In such as scenario one can imagine a performance difference between the query engine and the database server resulting in a performance bottleneck.
2. Network latency may play a role in decreased throughput, if round-trip time taken for a request to reach the server and a response to reach the client is significant.

Note: In additional experiment C we find that the median latency is 0.062s (measured using a trivial query, “SELECT 1;”). For 10000 requests (atomic inserts, or batch size 1) this would be a total 620 seconds. Using this information, we can conclude that network latency contributes

significantly to total time and has a large diminishing effect on throughput. This also explains why we see a gradual increase in throughput as we increase batch size.

## Indexing

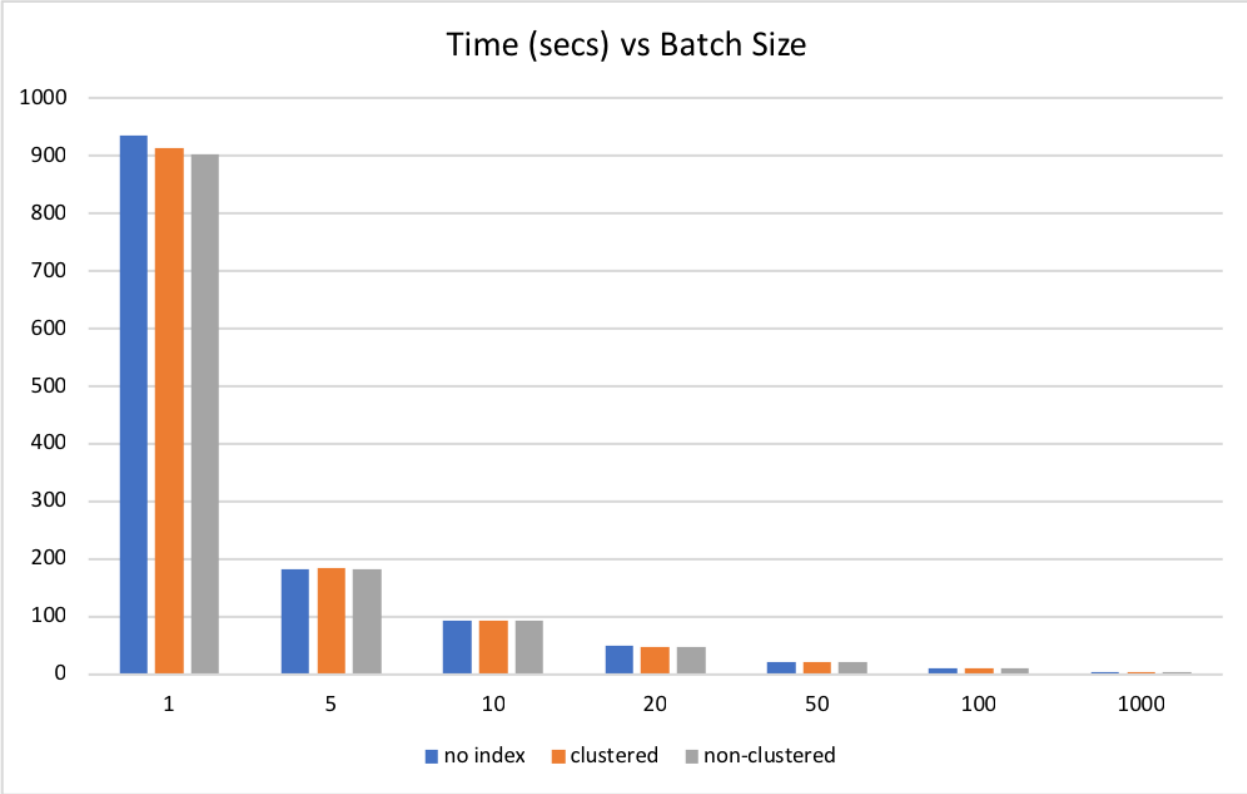
We hypothesized that adding indexes to a table will increase the time taken for a bulk insert and in turn lead to a lower throughput. The results from our experiments, partially support our hypothesis as seen in the figure below. For a higher batch size (1000), we observed that the time taken for inserts reduces when adding an index on the identity column and subsequently clustering it, but slightly increases again on adding a secondary non-clustered index [6].

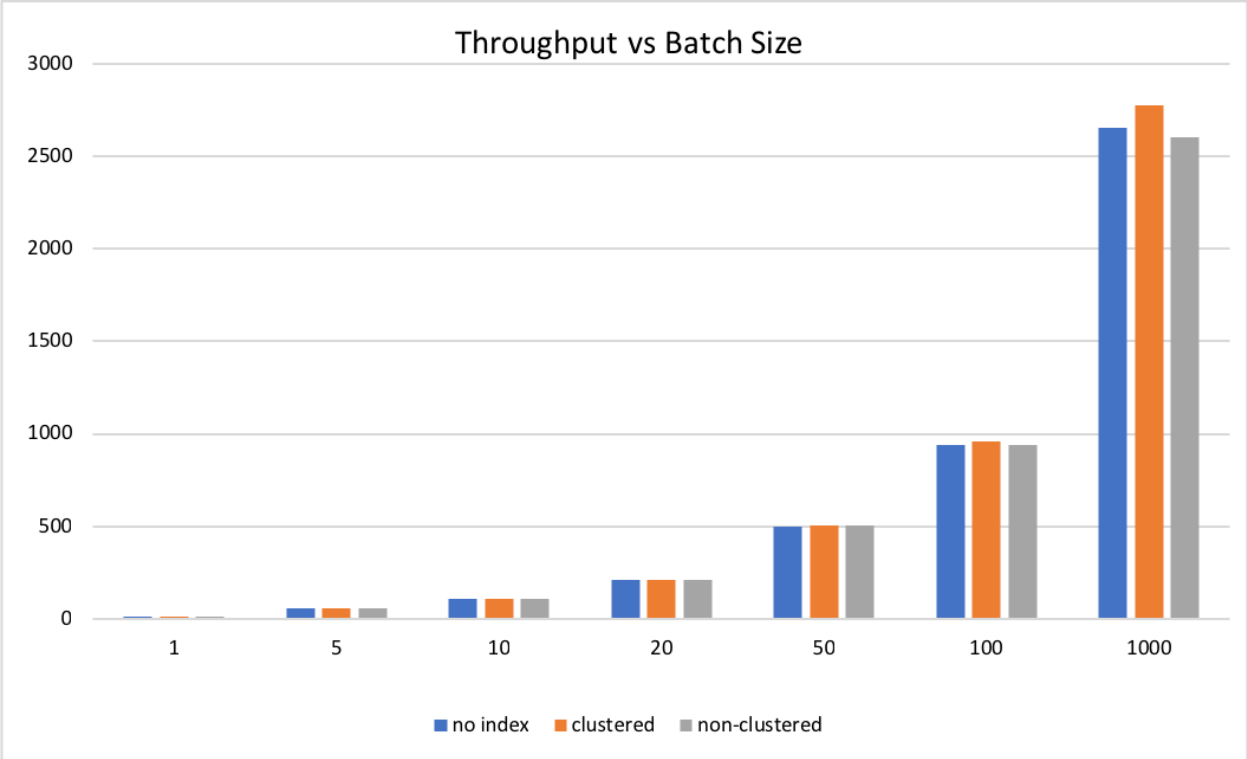
It is important to note that clustering a table on an index in PostgreSQL, is actually a non-clustered index with rearranged index keys for a one-time operation. So, presumably, it is just a non-clustered index and Postgres will still append the record to the end of the page.

For batch size of 1, we observed results that do not support our hypothesis, where we could see a decrease in time on adding an index on the table and a further decline on adding a non-clustered secondary index. These results are different from other batches probably due to network latency caused when the database is hit with a single insert statement for each record every time.

As per PostgreSQL documentation,the fastest method is to add large amounts of data to a table in bulk before creating indexes [7]. This is because every time a record is inserted, postgreSQL appends the record to the last page on the btree and subsequently updates the indexes for those records for easier retrieval in the future. More indexes mean more time to update every index, hence lower throughput [8].

However, we could not reproduce any such results and would like to experiment with records more than 10000 to see significant differences in insertion time.


Time (secs) vs Batch Size
1000
900
800
700
600
500
400
300
200
100
0
1
5
10
20
50
100
1000
no index
clustered
non-clustered


Throughput vs Batch Size
3000
2500
2000
1500
1000
500
0
1
5
10
20
50
100
1000
no index
clustered
non-clustered

# Additional Experiments

## Experiment A (Shreya)

The copy command in Postgres requires the user to be a Superuser in order to run the query from client. Due to this limitation we executed the Copy command from command line and recorded the time elapsed.

**Export data to a CSV file:**

*psql -h rds-postgresql-hotelreservation.cqfnnuiplrsh.us-east-2.rds.amazonaws.com -U masterusername -d hotelreservation -c "\copy guest to 'D:/guest_copy.csv' DELIMITER ',' CSV HEADER;"*

**Bulk Insert using Copy Command:**

*psql -h rds-postgresql-hotelreservation.cqfnnuiplrsh.us-east-2.rds.amazonaws.com -U masterusername -d hotelreservation -c "SELECT extract(epoch from now());" -c "\copy guest_copy from 'D:/guest_copy.csv' DELIMITER ',' CSV HEADER;" -c "SELECT extract(epoch from now());"*

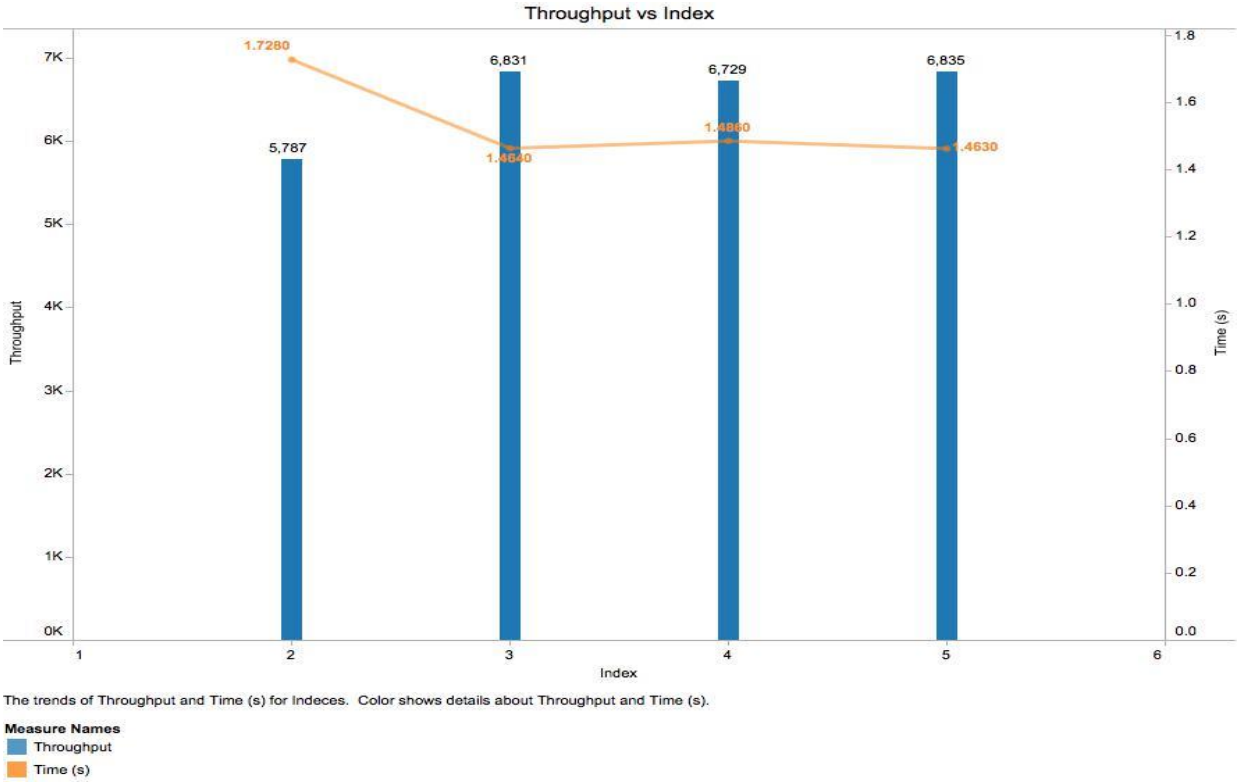
| Index Type | Time Elapsed (copy command) | Throughput (copy command) | Throughput (Batch size: 1000) |
|---|---|---|---|
| No Index | 0.5572 | 17946.87 | 2653.2237 |
| Clustered Index | 0.58374 | 17130.91 | 2776.2354 |
| Non-clustered Index | 0.68397 | 14621.16 | 2776.2354 |

As expected, maximum throughput was observed without indexes. This experiment was performed to copy 10,000 records from guest to guest_copy. As compared to the batch insert results, the bulk loading feature, i.e. Copy command takes significantly less time. For bulk insert, we recorded a throughput of 17946(no index) as compared to 2653.22 for a batch insert of size 1000, which is 6x better than the batch insert. We can infer that copy command isn't affected by network latency much and might be inserting records on the same network link.

## Experiment B (Sahil)

We calculated throughput with additional indexes by re-running the same experiment. We added a non-clustered secondary index each time, along with the existing indexes on the table.

*cluster guest*
*CREATE INDEX second_non_clustered ON guest (lastname);*
*CREATE INDEX third_non_clustered ON guest (address);*
*CREATE INDEX fourth_non_clustered ON guest (city);*




From the figure above, we see that the throughput is the least when there are only two indexes (clustered and non-clustered) on the table. We see that the throughput decreases slightly every time a non-clustered index is added. This is because every time a layer of indexing is added, the record has to be updated in each of the indexes, so that it is easier to retrieve it using that index in the future.

We however, observed a very minimal change in throughput on adding additional non-clustered indexes. This behavior was similar to bulk batch inserts, where we did not see a significant difference in insertion times on increasing the batch size.

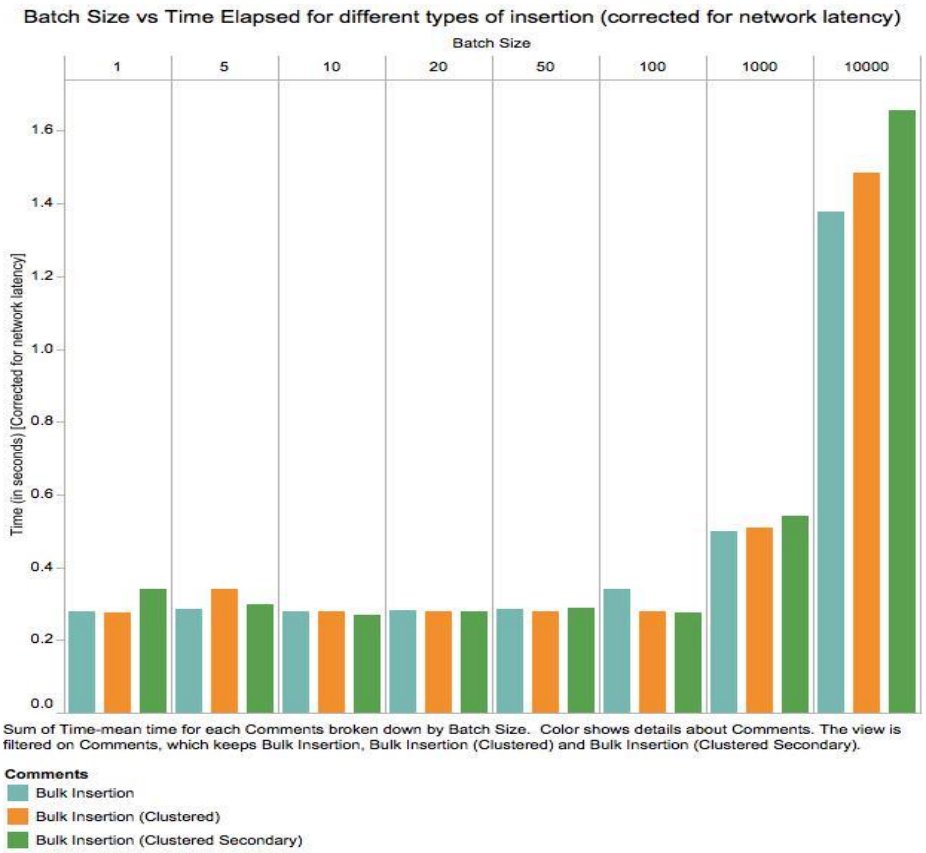
## Experiment C (Aditya)

We created a function that runs the query ‘SELECT 1;’ 100 times and reports statistics.

*$ ./bin/run addRecords -l*
*{ mean: 0.0694, median: 0.062, variance: 0.004775939999999984, sd: 0.06910817607201035 }*

The function, named ‘latency’, is used [9]. Based on these observations, we initially hypothesized that our PostgreSQL database is extremely optimized for query parsing and inserts but does not handle incoming HTTP traffic at a comparable rate, which results in a performance bottleneck.

However, in experiment C, we found that average round-trip network latency (measured using the query “SELECT 1;”) had a median value of 0.062. Scaled for 10000 synchronous, sequential insertions, this would be 620 seconds - which aligns perfectly with our prior observations for inserts.




With this additional knowledge, we were now aware that network latency is the primary factor in slow insert times with sequential inserts. As for the slowdown during asynchronous insert requests, we believe that the large number of incoming HTTP requests degrades our database server’s performance, although we’d need to conduct other experiments to confirm this.

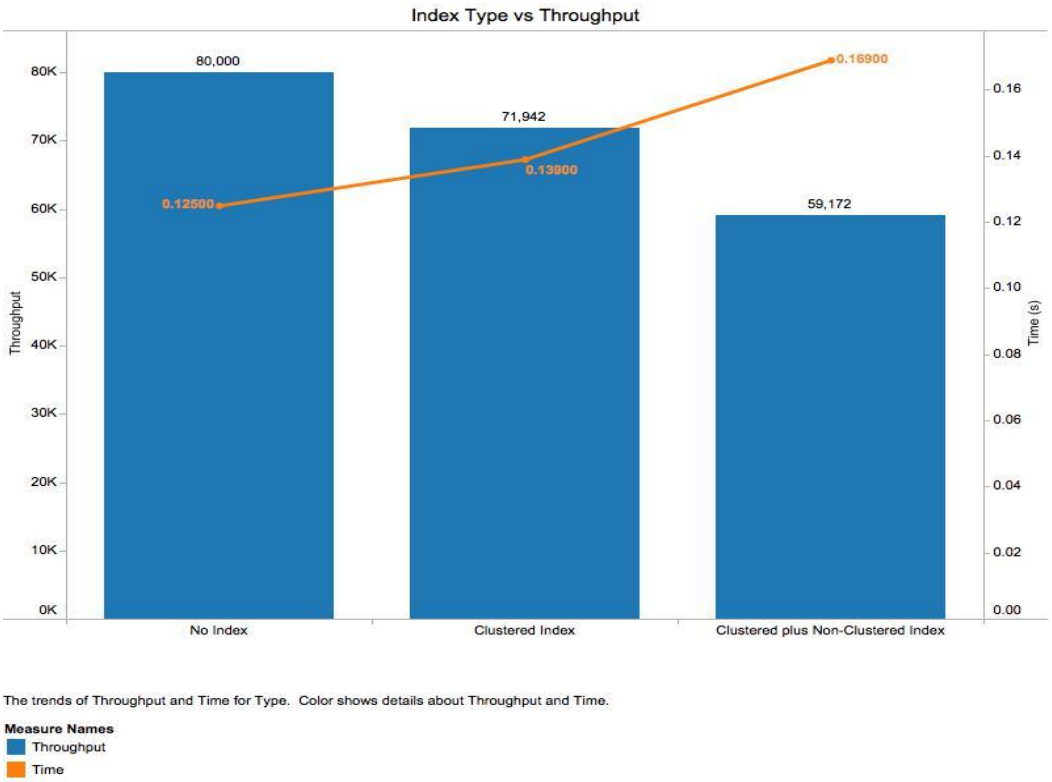
## Experiment D (Akshay)

A table guest_copy was created in our database specifically to carry out this experiment. The following statements were used:

*INSERT INTO guest_copy (guestid, firstname, lastname, address, city, zipcode)*
*SELECT guestid, firstname, lastname, address, city, zipcode FROM guest;*

The experiment was performed for three cases: when there were no indexes, when there were clustered indexes and the case when there were both clustered as well as non-clustered indexes. The results of each of these experiments are tabulated as follows:

| Index Type | Throughput | Time (ms) |
|---|---|---|
| No Index | 71,942.44 | 125 |
| Clustered Index | 59,171.59 | 139 |
| Clustered + Non-Clustered Index | 80,000 | 169 |




From the above figure, we observe that throughput decreases as the number of indexes increase on the table.

# Conclusions and Discussion

From our performance experiments we concur that the bulk insert runs the fastest with the COPY command (bulk load feature of postgreSQL). COPY command runs on the server and does not involve using the client, which makes it way faster than even bulk inserts. We observed that inserting 10000 records using the copy command took 0.55 seconds as compared to 1.24 seconds for a single multirow insert statement for the same insertion. In sum, COPY gave us ~2x faster inserts.

For the first experiment with 10000 atomic inserts, we experimented with following a synchronous and an asynchronous approach. We hypothesized that updating the data without waiting for a response would be faster than receiving confirmation and then updating, as this deals with network latency overhead. Surprisingly, we saw that both these experiments produced similar results even on multiple iterations.

Our additional experiments support our hypothesis that *with increase in the number of indexes, time taken to insert data increases, thereby reducing the throughput*. However, batch inserts do not perform as per our expectations. If we had more time, we would like to experiment with a million-record insertion to reproduce results as listed in the PostgreSQL documentation.

The limitation to our performance experiments is that PostgreSQL does not support clustered indexes. However, it supports clustering the table on an already existing non-clustered index. This is a one-time operation i.e. when the table is subsequently updated, the changes are not clustered. For our experiments involving insertion with different batch sizes, this one-time operation doesn't happen between batches. Irrespective of that, Postgres always appends a record at the end of the table. We run the CLUSTER table command every time we run an experiment, but presumably we should not see a difference in the throughput.

DBA's should consider writing insert statements in a single query or make use of the bulk insertion feature of database for maximum throughput. The fastest method for insertion is to bulk load the table's data using copy and then create any indexes. Creating an index on a pre-populated table is faster than updating it with each row load.

Additionally, indexes on the table should be created based on how they are going to be accessed. The purpose of having an index on the table is to have faster retrieval. The tables that are accessed frequently should have indexes on columns that are used in predicates and join conditions in order to ensure optimal application performance.

# References

[1] https://github.com/adityachatterjee42/AASS_Autumn18/tree/master/poparty
[2] https://www.npmjs.com/package/pg
[3] https://www.npmjs.com/package/faker
[4] https://www.npmjs.com/package/perfy
[5] https://www.npmjs.com/package/oclif
[6] https://stackoverflow.com/questions/12206600/how-to-speed-up-insertion-performance-in-postgresql
[7] https://www.postgresql.org/docs/current/static/populate.html
[8] http://www.postgresonline.com/journal/archives/10-How-does-CLUSTER-ON-improve-index-performance.html
[9] https://github.com/adityachatterjee42/AASS_Autumn18/blob/master/poparty/src/operations/populateDatabase.js