# Graph Databases:
# The Case of Neo4j vs TigerGraph

Georgios Michas - cs2190024
cs2190024@di.uoa.gr

HELLENIC REPUBLIC
National and Kapodistrian
University of Athens
EST. 1837

# Graph Databases

- Using this type of databases is ideal when the “problem” needs to be represented as a network with nodes connected via relationships, for example a social network
- Standard databases, like PostgreSQL, can easily find connections that are only one hop away. Graph databases are optimized to handle queries that can follow multiple hops and collect all nodes within a radius. As a result, a question arises, which graph database is the best ?

# Why Neo4j vs TigerGraph ?

- Very popular systems
- Native graph databases freely available that provide a SQL-like declarative query language (Cypher for Neo4j, GSQL for TigerGraph)
- Python API for benchmarks


# LDBC SNB Benchmark

- For the experiments, I used LDBC Social Network Benchmark (LDBC SNB)

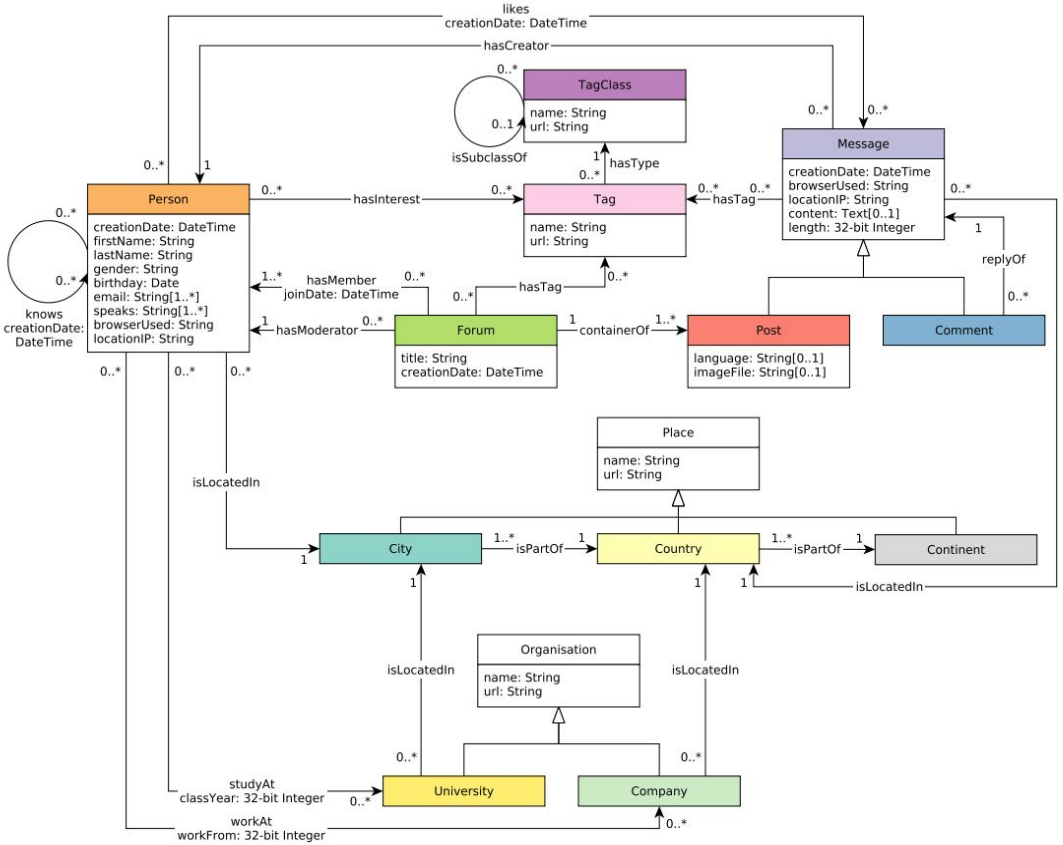
- SNB represents a realistic social network, including people and their activities, over a period of time.

- The LDBC SNB data generator instantiates synthetic datasets of different scale with distributions and correlations similar to those expected in a real social network. This is realized by integrating correlations in attribute values, activity over time, and the graph structure. The attribute values are extracted from the DBpedia dictionary. They are correlated among themselves and also influence the connection patterns in the social graph.

- I used 3 different scale factors for the data generation: {0.1, 0.3, 1}.  The next option was scaling factor 10 but I had issues with my laptop. Moreover, I took the query parameters generated for each scale factor and used the as input.

# Schema

- The main entity is Person. Each person has a series of attributes, such as name, gender, and birthday, and a series of relationships with other entities in the schema. Moreover, the person has a “self referenced” relationship, which is very important in order to generate queries with a lot of hops.



# Queries

- Interactive Short (IS) Workload (7 queries): The queries in the IS workload are relatively simple path traversals that access at most 2-hop vertices from the origin—given as a parameter binding.

- Interactive Complex (IC) Workload (14 queries): The queries in the IC workload go beyond 2-hop paths and compute simple aggregates—rather than returning only tuples. Additionally, two of the queries have to calculate the shortest path between two vertices given as parameters.

- Business Intelligence (BI) Workload (24 queries): The queries in the BI workload access a much larger part of the graph. This is realized by replacing the origin in the interactive queries with more general selections on message creation date or the location of a person. As a result, traversals are not originating from a single source, but rather from multiple points in the graph.

- Micha's Benchmarks (MIB) Workload (10 queries): I wrote 10 queries from scratch. At minimum each query will do 3-hops. Some queries are more complex than others. For example: the mib_7 at the start, finds the friends of a person in max depth 4-hops and then it does extra hops (on other nodes) in order to finish. Most of the queries have filters and sorting. Moreover, some queries contain group bys and aggregations.

# System

- Ubuntu 20.04.4 LTS
- RAM: 16GB DDR4
- CPU: Intel Core I7 9750H
- Neo4j: 3.5/4.3 Community, TigerGraph: 3.4 offline (with syntax V1)

# Implementation

- All the experiments were written in Python 3.8.10
- I configured the systems optimally for single-node execution
- Neo4j uses 6GB of RAM and 32M page size, (java: build 1.8.0 191). For each primary attribute I used an index (Btree) for Neo4j (otherwise it has poor performance). TigerGraph doesn’t provide any option to build an index but it does things internally while loading.
- The queries for TigerGraph are pre-compiled. This is an option that this system provides in order to get significant speed up.
- Each query runs 5 times and I take the average of the last 4 execution times (excluding the first because is a cache cold start). Moreover, I used various configurations for the queries (different parameters).

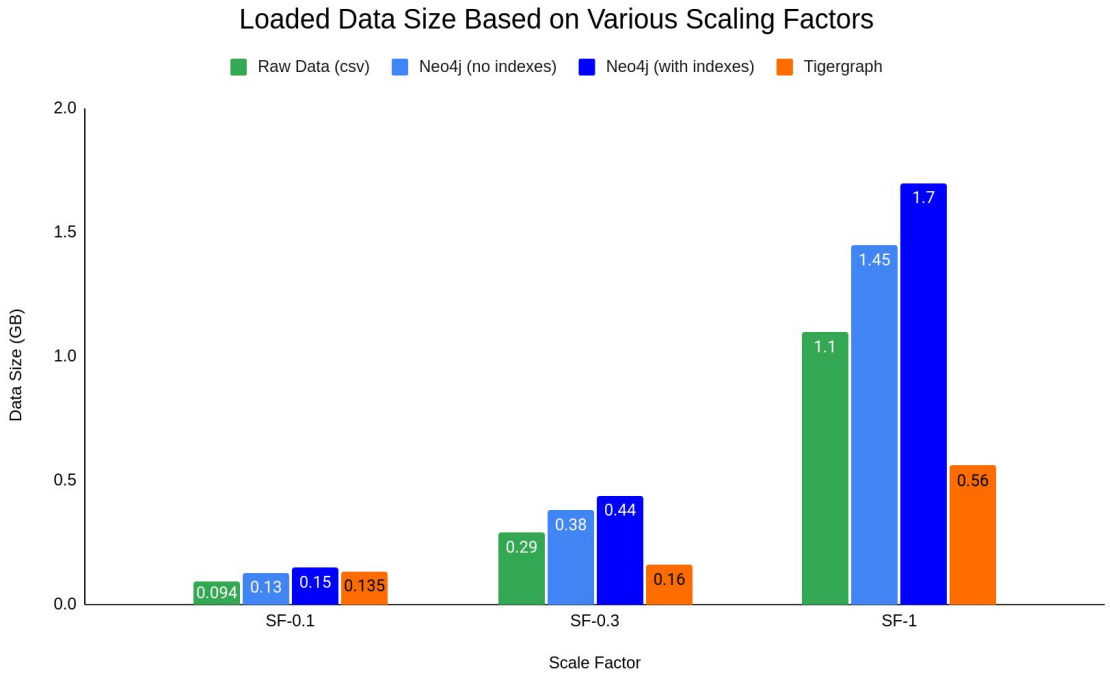
# Datasets Graph Size

- Scale Factor 0.1:
  - Nodes: 327588
  - Edges: 1477965
- Scale Factor 0.3:
  - Nodes: 908224
  - Edges: 4583118
- Scale Factor 1:
  - Nodes: 3181724
  - Edges: 17256038

# Dataset Loaded Size

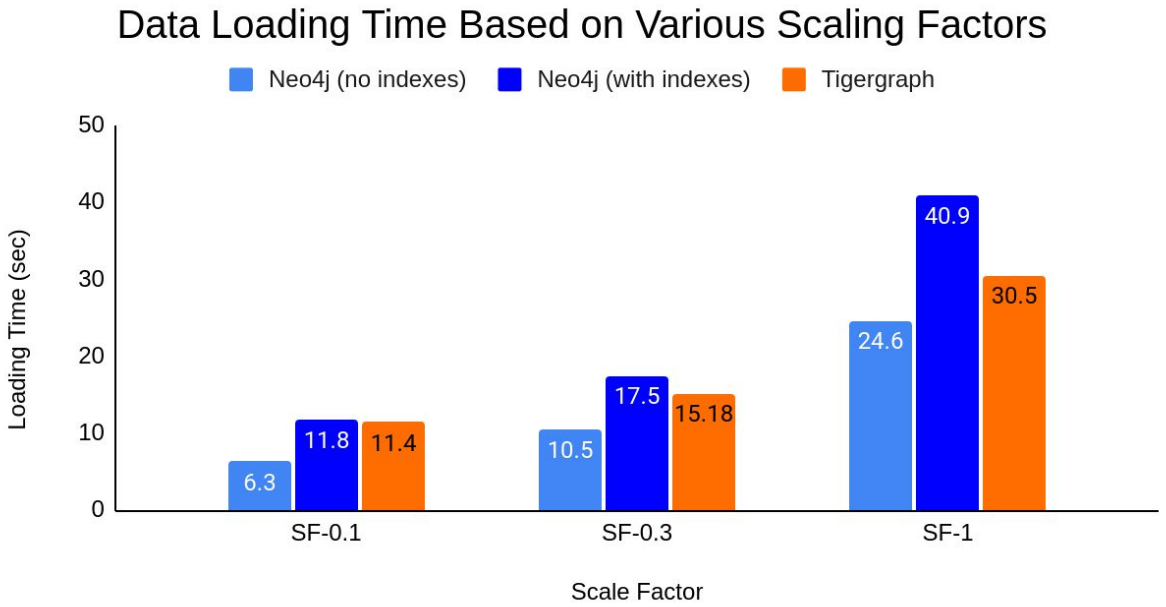
- TigerGraph needs significant less space in order to store the whole dataset. Around 2.5x - 3x less space compared to the others.

- We can clearly see that Neo4j's indexes increase the size of the database significantly (10-15%)

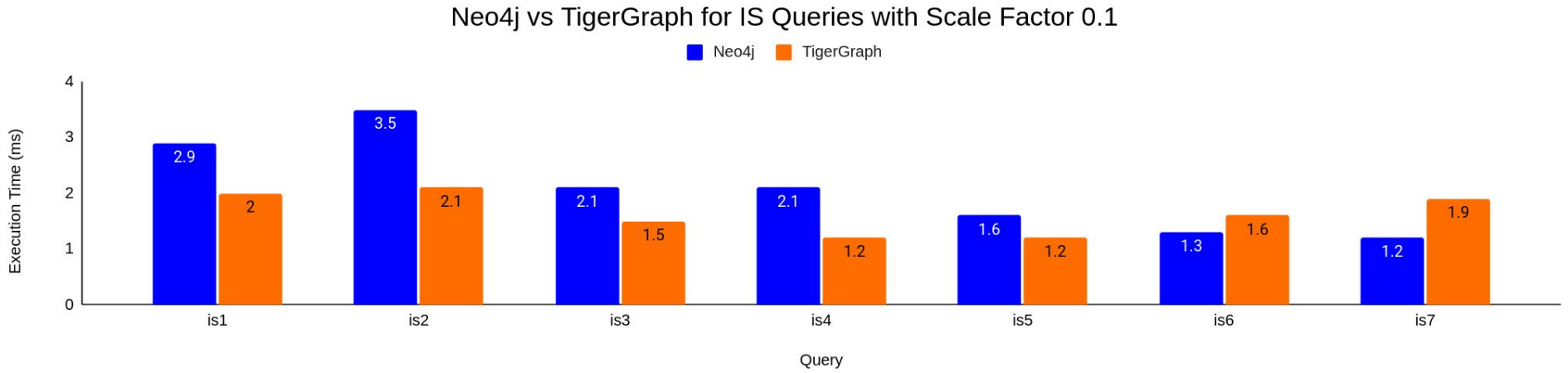


# Dataset Loading Time

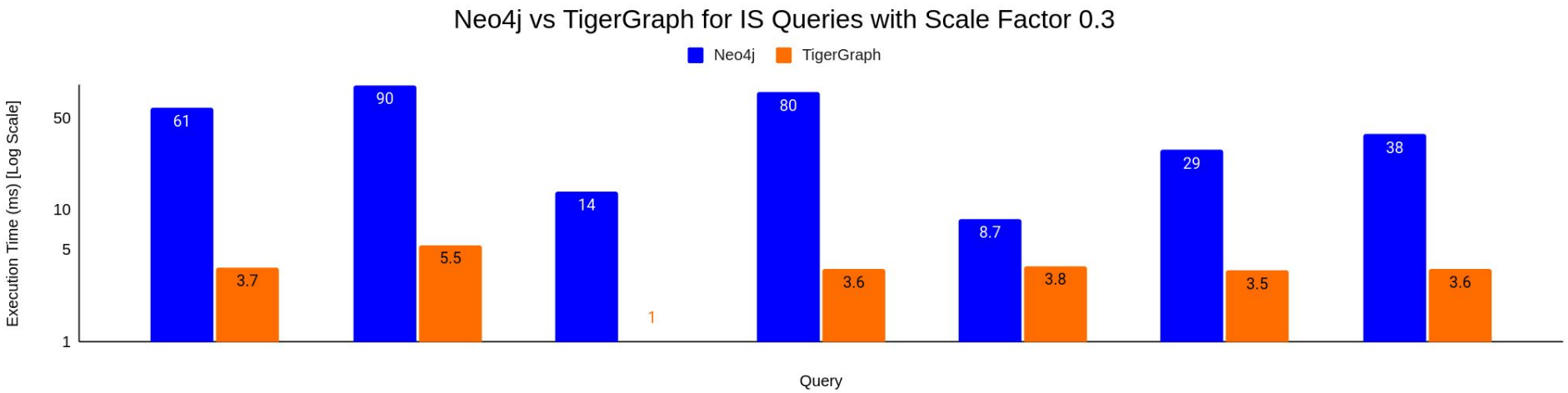
- For all three scale factors Neo4j without indexes has the smallest loading time. On the contrary, when indexes are being used they add a significant overhead on the overall loading time. This is a well-known issue for Neo4j because index-building is not scalable. On the other hand, TigerGraph with indexes is scalable and as a result it is faster than Neo4j.



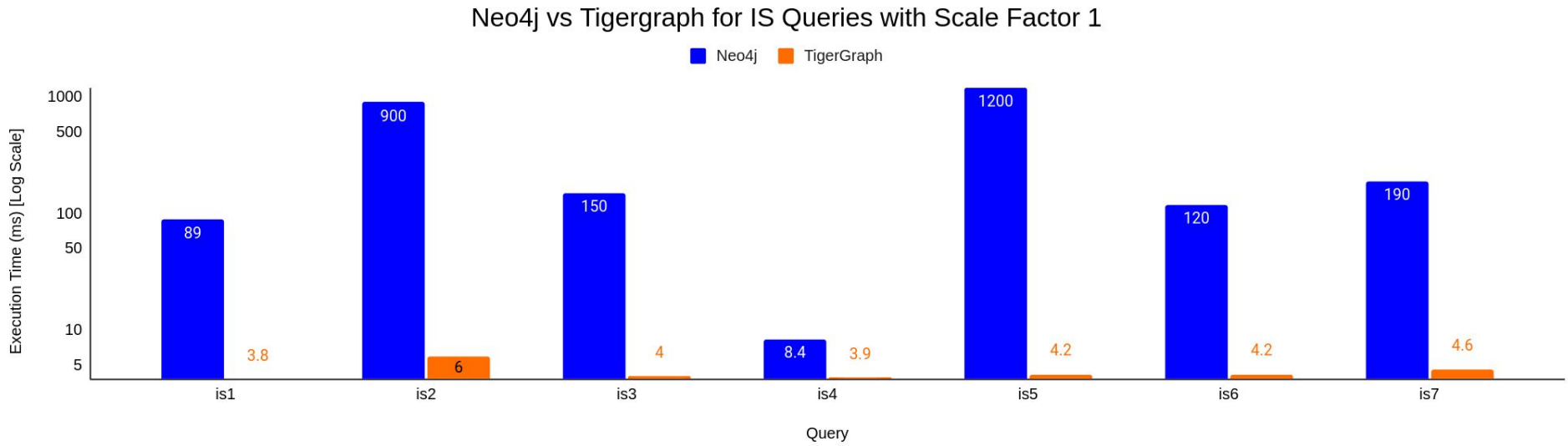
# Execution Time for IS Queries
# SF-0.1



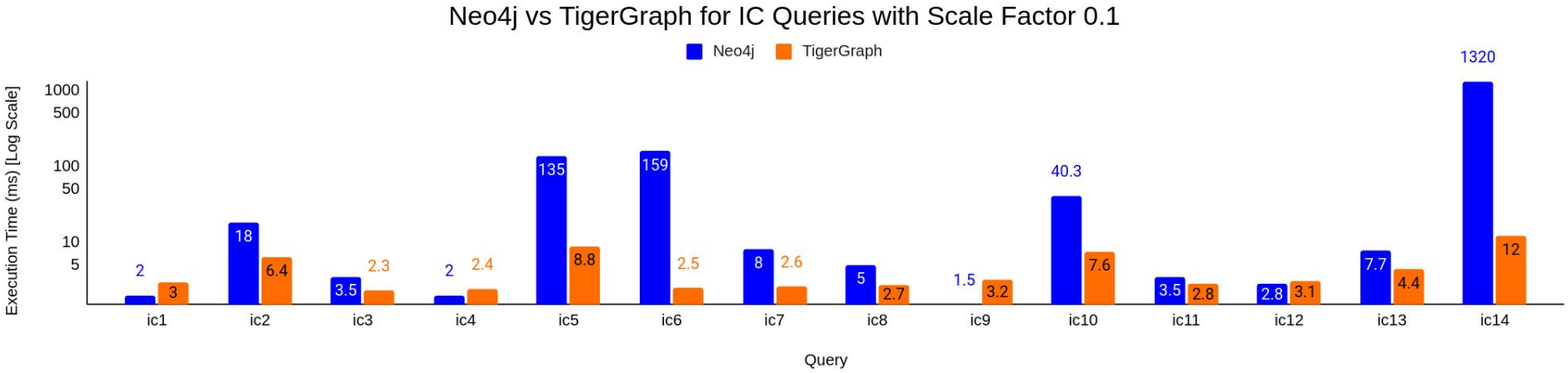
# Execution Time for IS Queries
# SF-0.3



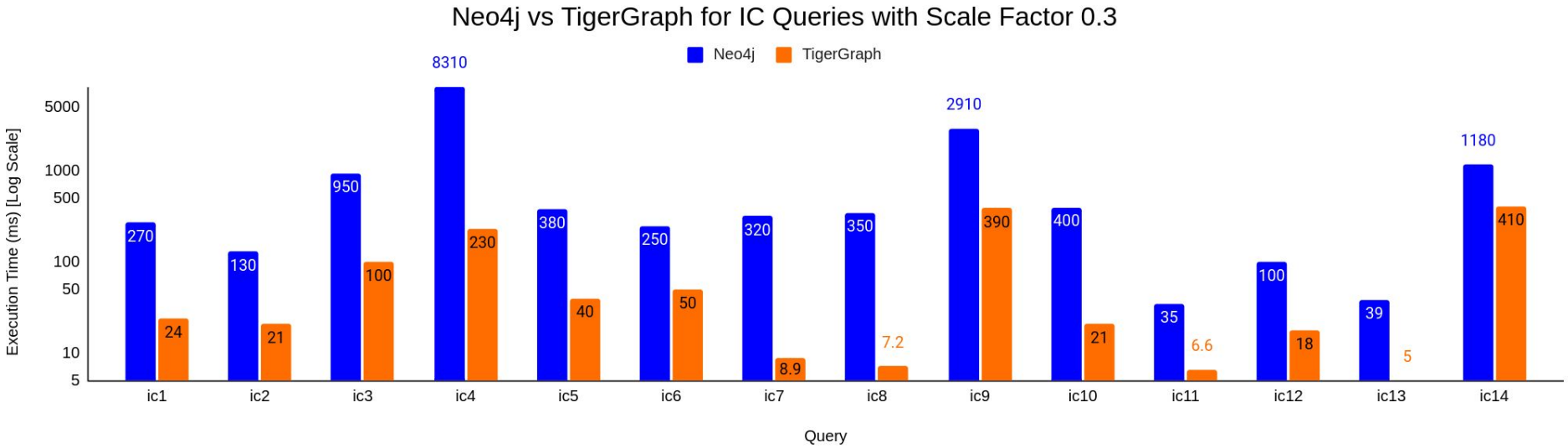
# Execution Time for IS Queries
# SF-1



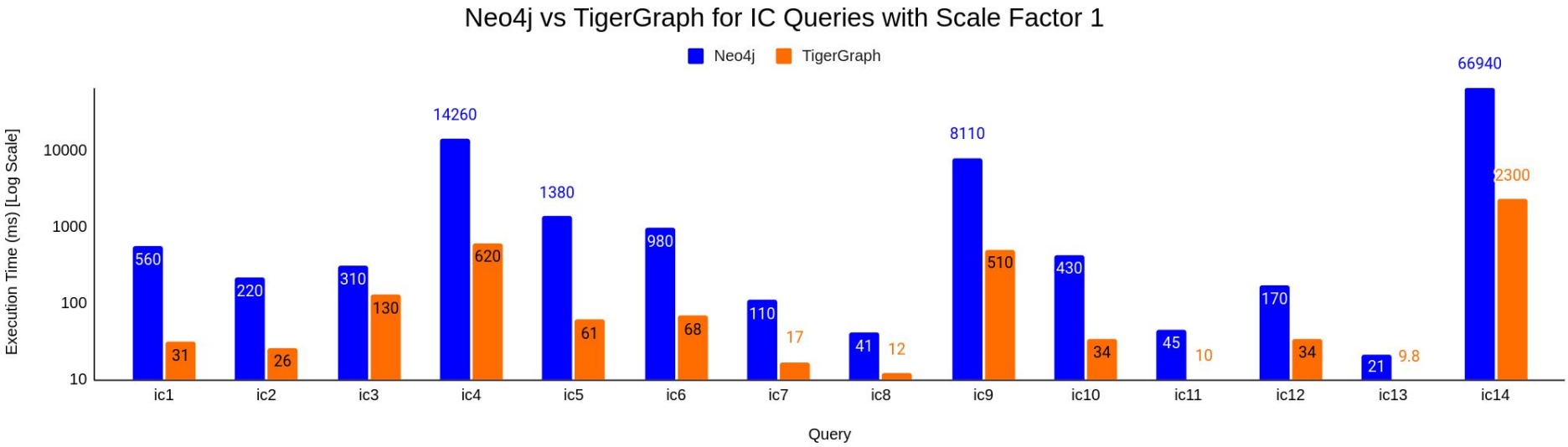
# Execution Time for IC Queries
# SF-0.1



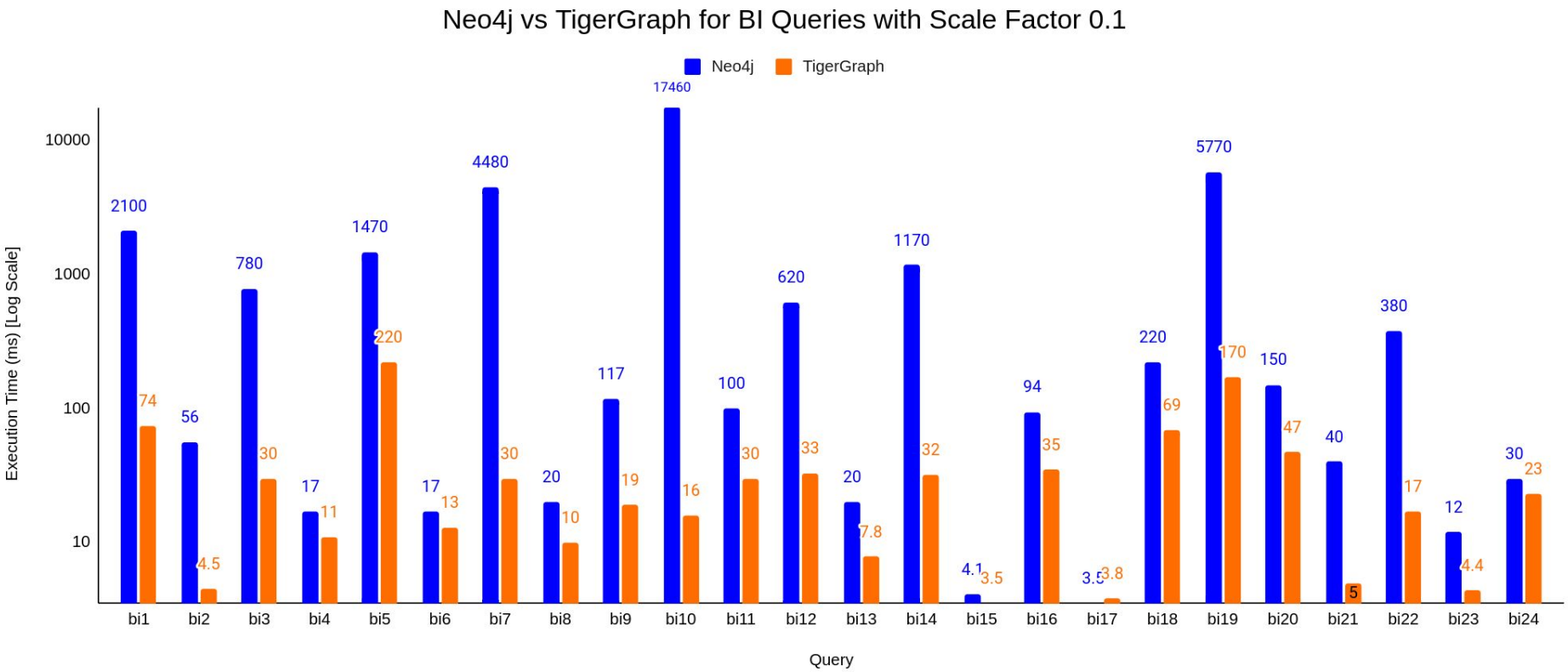
# Execution Time for IC Queries
# SF-0.3



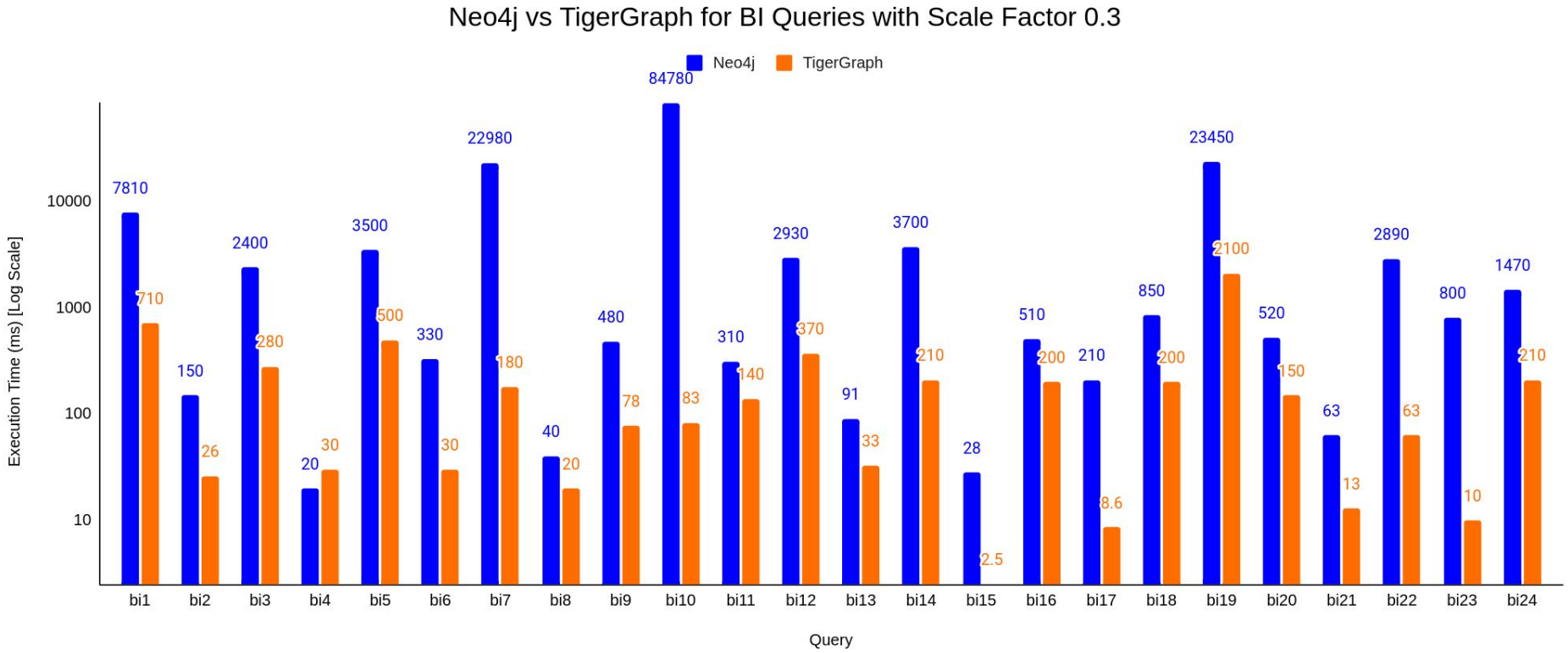
# Execution Time for IC Queries
# SF-1



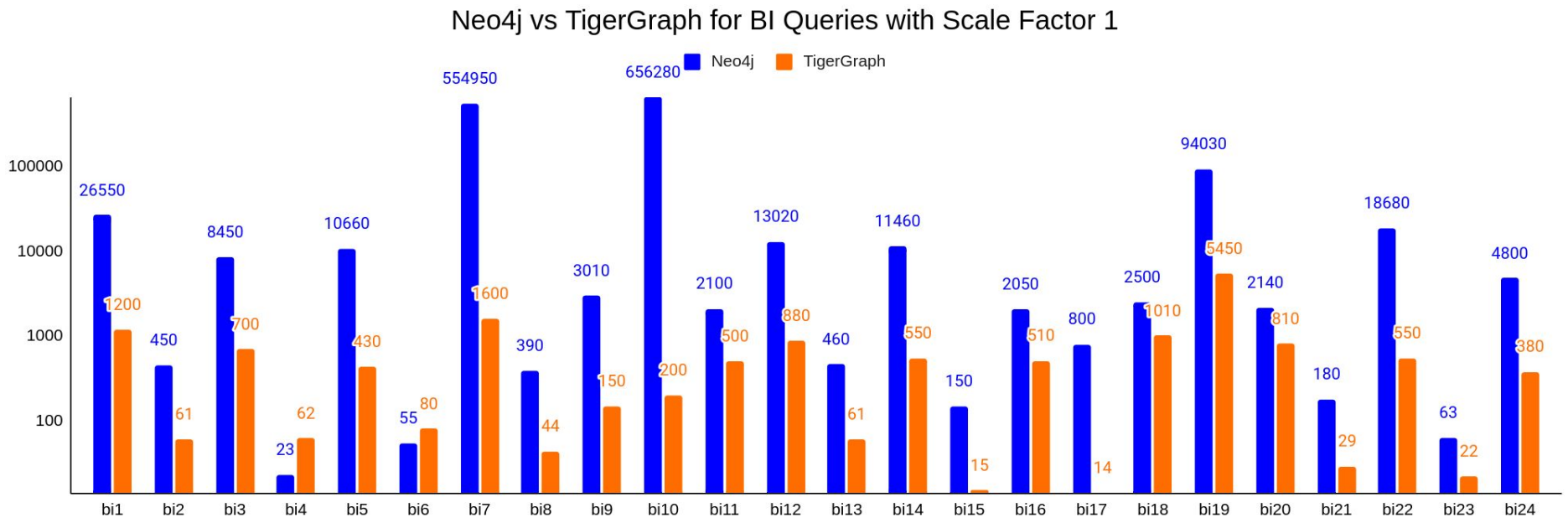
# Execution Time for BI Queries SF-0.1



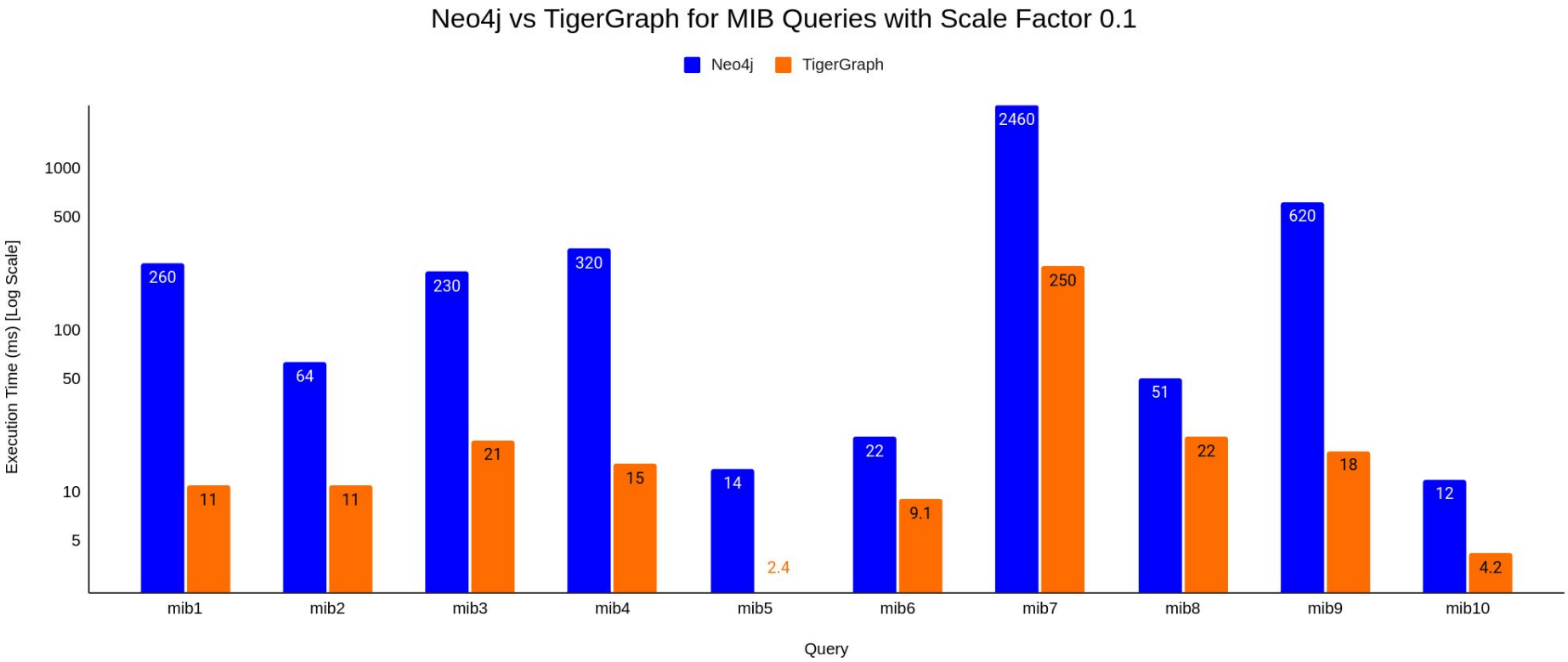
# Execution Time for BI Queries SF-0.3



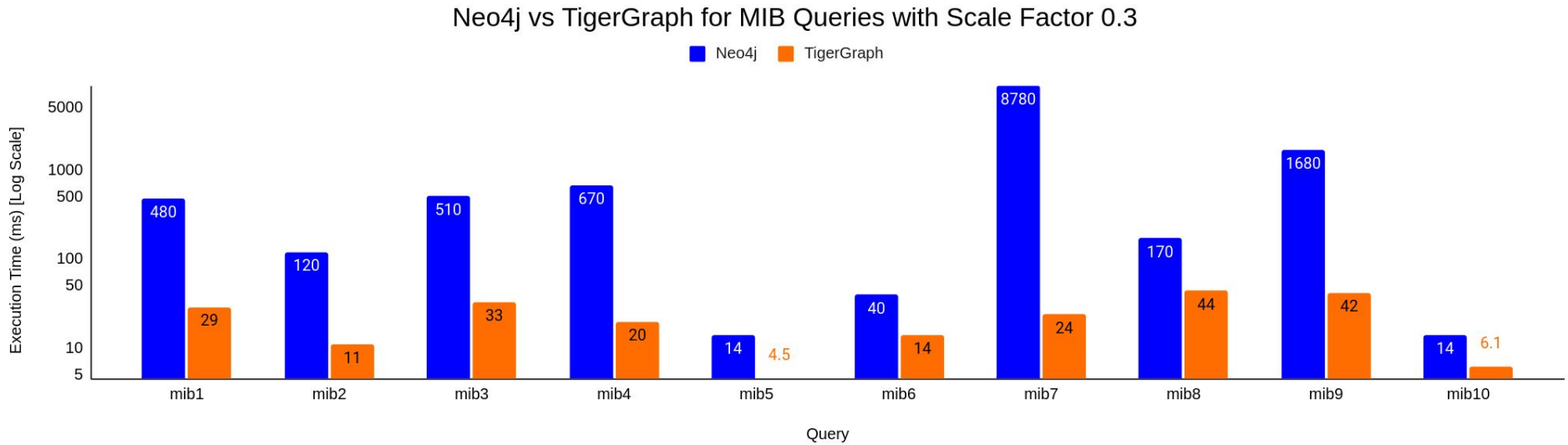
# Execution Time for BI Queries SF-1



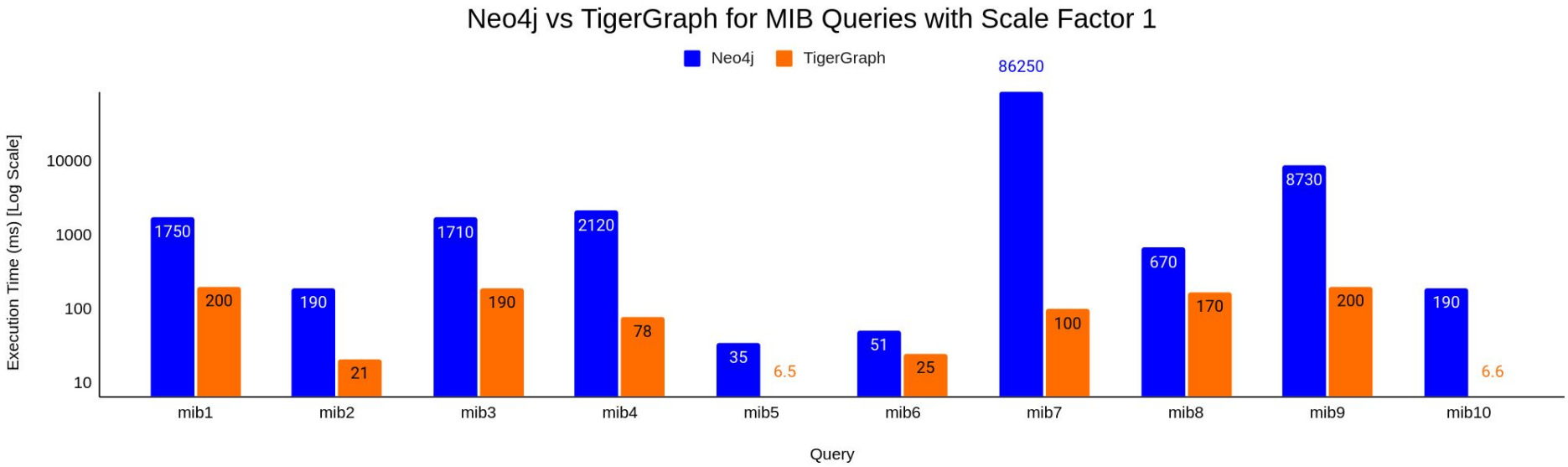
# Execution Time for MIB Queries SF-0.1



# Execution Time for MIB Queries
# SF-0.3



# Execution Time for MIB Queries
# SF-1



# Benchmark Summary

- TigerGraph OBLITERATES Neo4j (with indexes), TigerGraph is orders of magnitude faster than Neo4j

- For complex workloads (more than 2-hops) such as IC, BI or MIB as the scale factor increases, TigerGraph can scale and provides significant low execution times. On the contrary, Neo4j fails to scale and has exponential growth in terms of execution time.

- For smaller scale factors the execution times between Neo4j and TigerGraph have smaller differences but as the Scale Factor increases we can clearly see that Neo4j can’t handle even simpler queries such as in IS (max 2-hops), and the gap between them becomes huge.

- The combination of low cost loading size/time with the ability to scale even when the queries are complex and the data size becomesmassive, makes TigerGraph the only choice. The only pitfall with TigerGraph is that GSQL is too complex compared to Neo4j. In order to write efficient code in GSQL it is not very simple. Cypher is closer to a SQL declarative style language, thus much easier to learn it and write efficient queries.

# Notes

- In my source code, I have the code for the benchmarks in python and the gsql queries for compilation. Moreover, I have bash scripts to load the data into the databases.

- Note that the code(https://github.com/zhuang29/graph_database_benchmark) of the paper (https://arxiv.org/pdf/1907.07405.pdf) is bad :) . As a result, I wrote it from scratch, added missing pieces and borrowed a few pieces.