# Improving Spatial Search using a Graph Database and Uber H3

Jaye Renzo Montejo
University of the Philippines, Diliman
Quezon City, Metro Manila
jlmontejo@upd.edu.ph

Irvin Kean Paulus Paderes
University of the Philippines, Diliman
Quezon City, Metro Manila
itpaderes@upd.edu.ph

Ligayah Leah Figueroa
University of the Philippines, Diliman
Quezon City, Metro Manila
llfigueroa@up.edu.ph

## ABSTRACT

With the rise in popularity of applications about ride-sharing, food delivery, courier service, and any subject that caters in fulfilling the needs of an on-demand economy, access to spatial information in real-time becomes an important aspect. To achieve this, spatial databases are used to store and process location information that are consumed by these applications. Normally, these databases use relational modelling. This research proposes the use of hexagonal discrete global grid systems in graph databases in order to improve upon existing spatial databases. In measuring the query performance, a geofence surrounding Quezon City, Philippines was created and various amenities within the area were extracted then used as the dataset. Bounding box search queries were then performed on this dataset. A working implementation of a graph-powered spatial database with a hexagon discrete global grid system called H3 was presented, and it was shown that this setup can produce lower query execution times at scale than those of relational databases.

## CCS CONCEPTS

• **Information systems → Graph-based database models**; *Query languages for non-relational engines*; **Geographic information systems**.

## KEYWORDS

spatial database systems, discrete global grid systems, graph database, Neo4J, Uber H3

## 1 INTRODUCTION

There has been an increased development in the service industry that points it towards being an on-demand economy. These developments are mostly dependent on transportation, which means applications must have instant and continuous access to location data in order for them to choose optimal routes that consider current traffic and total travel time without having prior knowledge of the area. Because of this, there is an increased demand in spatial database systems (SDS) that use the power of database management systems (DBMS) in order to store large amounts of location data. The challenging issue here is to efficiently perform standard DBMS operations to these location data, as well as capture the geometric processing that can be implemented on them.

Database indexing is a way of optimizing disk access in processing queries. In SDS, this comes in the form of discrete global grid systems (DGGS). DGGS is a series of regions that partitions the Earth's surface, where each region contains points that can be associated to a location data. These regions can be further subdivided to finer resolutions, giving a better representation of regions that can be utilized for various applications.

Over time, DGGS was repurposed to being a data structure for location data as it provides consistent storage and ease of reference to the spatial data [8]. Uber's H3 is one example of this case. H3 is a hierarchical geospatial indexing system that uses hexagonal grids (which can be further subdivided to finer hexagonal grids) and hierarchical subdivisions. [3] Uber has been using H3 as a data structure for various events and processes that rely on its location data.

As it stands, most SDS today were implemented on relational DBMS, therefore they perform spatial operations that involve graph theory under the constraints of being a relational model. With this, a large amount of SDS data should enjoy the advantages of being implemented on graph databases (GDB). GDB came about from NoSQL databases which improved upon the limitations set by having to use relations in order to define the connections of each data. By having an actual graph structure to explicitly lay out the dependencies between nodes of data, it can now perform semantic queries on its nodes and edges and have proper data structure to represent its properties.

In this study, the aim is to provide an alternative system for existing relational and graph databases by leveraging a DGGS to use as an indexing system in a graph database. Its performance is then compared to existing relational and graph databases. The research goes as follows: Section 2 provides information on the different technologies that the research utilizes or is based of, Section 3 gives an overview of the existing state of the art in terms of the use of graph databases in SDS and implementing DGGS as an indexing system, Section 4 presents the methodology on how the performance of DGGS-powered graph databases is compared to normal relational and graph databases, and Sections 5 and 6 discuss the results of the experiment and summarize the conclusions drawn from the analysis of the results.

## 2 BACKGROUND

There has been a rise in applications that require on-demand access to spatial information. These applications rely on some technologies that will be utilized for this research. In order to fully define the scope of the experiments to be done, each of the base technologies are discussed first in the following subsections.

### 2.1 Spatial Database Systems

A spatial database system (SDS) focuses on implementing DBMS to handle spatial or geographic data. It offers additional functionality over traditional databases to support spatial object types through specialized queries in order to analyze, edit, and present the data with ease. [5] SDS reveals deeper insight into location data, such as patterns, relationships, and situations, all of which can help in implementing applications to make smarter decisions.

SDS represents geographic data by its vector approach. It allows the use of geometric objects such as points, lines, and polygons to represent geographic data. This representation is more efficient as it can already represent locations with the use of these features. But in cases where there is a need to handle large amounts of location data, SDS still suffers from the pitfalls of traditional databases.

## 2.2 Discrete Global Grid Systems

A discrete global grid (DGG) implements space partitioning to a polyhedron (in this case, Earth), and uses these region partitions as spatial data models [10]. These grids simplify position calculations by providing an abstraction to the partitions of the Earth through points which can be called a cell. Further improvement to this is the introduction of resolutions which allows further partitioning of these regions in a recursive pattern, turning into a series of discrete global grids with progressively finer resolutions. This improvement turns DGG into discrete global grid systems (DGGS).

One of the popular partitioning topology used for DGGS are hexagons. Hexagonal grids have higher packing density, approximate circular regions, and equal distance from other hexagon neighbors. Kimerling et al. [7] stated that due its equal orientation, hexagon cells can be used for recursive partitioning of spherically rectangular quadrilaterals with desirable rotational invariance present for any n-fold partitioning, an advantage when assembling global datasets of partition at varying spatial resolutions.

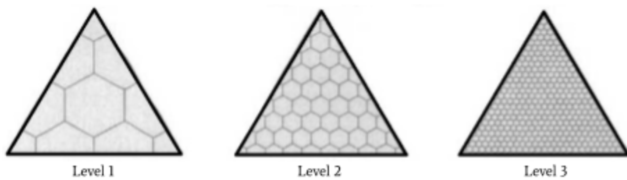


**Figure 1: Hexagon partitioning on a part of an icosahedron**

As discussed in Section 1, Uber's H3 is used as the DGGS in this research as it provides a great variety of language bindings for development [2]. At its current version, H3 delivers all basic functionalities required by the Open Geospatial Consortium in their standard data protocol for a technology to be considered a DGGS [4].

H3 uses map projection and grid partitioning to create a global grid system. It uses hexagon grids to give the system only one distance between the centerpoint and its neighbours as seen in Figure 2. It also has icosahedron partitions that, instead of being unfolded to produce two-dimensional maps, have the hexagon grids laid out to the faces themselves. A partial icosahedron with hexagon grids is seen in Figure 3

The resulting grid is constructed with 122 base cells covering the Earth, having ten cells per face. There are cells contained to more than one face. At each icosahedron vertex, there are pentagons (totaling up to twelve) to fill it up whole. H3 supports sixteen resolutions, and as it gets finer, it has cells having one seventh the area of the coarser resolution.

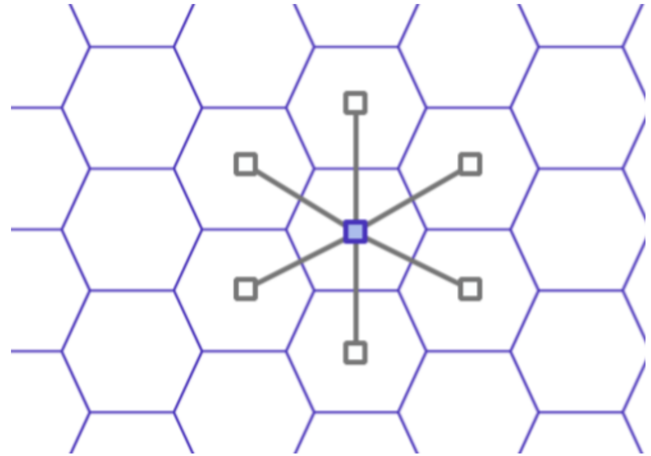

**Figure 2: Distance of a hexagon to its neighbors, Image taken from Uber's whitepaper about H3 [3]**

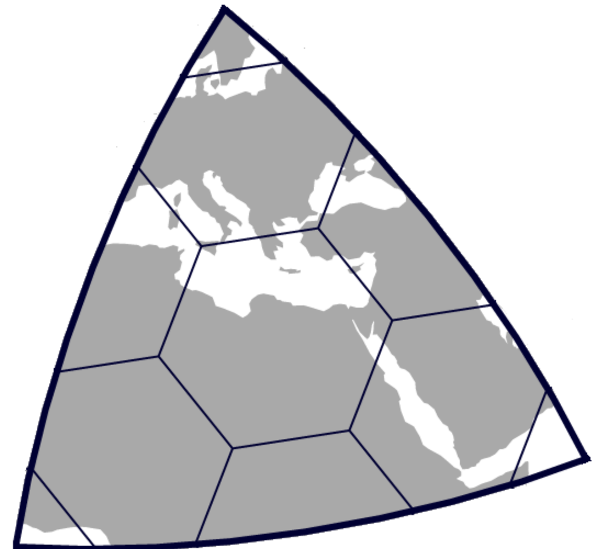

**Figure 3: A partial icosahedron with hexagon grids, Image taken from Uber's whitepaper about H3 [3]**

## 2.3 Graph Databases

Just like in normal DBMS, most established spatial database systems are running on relational DBMS. The introduction of NoSQL paved the way for developers to have an alternative to relations-motivated schema, allowing them to use a database that does not require establishing any form of relation. From NoSQL databases came about graph databases, where it improves upon the representation of graph relationships in RDBMS as implicit connection of nodes [1]. Graph databases directly implement graph structures for running semantic queries and for defining its data structure. The graph integrates data as a collection of nodes and edges, where the edges represent the relationships between the nodes. These

relationships allow data to be linked together directly and, in many cases, be retrieved with one operation. In a graph database, relationships between data are treated as priority. This makes querying relationships fast because they are perpetually stored within the database itself [11].

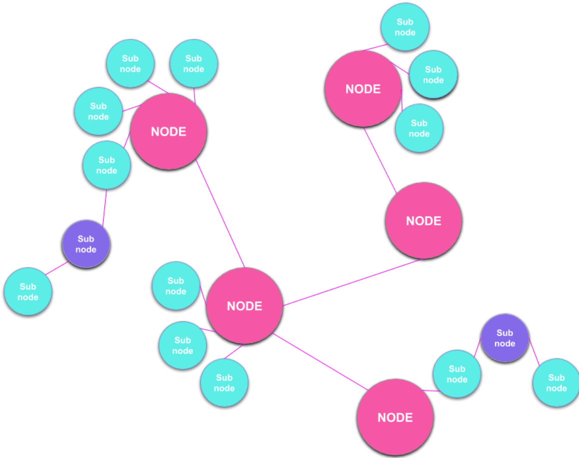


**Figure 4: Graph database with a set of nodes and their connections**

Naturally, with the advent of this technology came its implementation to improve upon spatial databases. Geographic systems can be easily abstracted to a graph system in order to utilize network analysis of each node [6]. One pitfall of relational SDS is that it doesn't scale well to large amounts of data. In theory, with the usage of the underlying network of nodes inside it, a graph database can easily perform operations and represent large amounts of data with ease by showing it as a graph. This database manages network features and elements in a graph in order to naturally interact with network features to solve useful problems. With this type of database and a DGGS as an indexing system, it is possible that this setup can match or even beat existing relational SDS technologies when operating on large datasets.

## 3 RELATED WORK

This section presents publications about graph databases and spatial databases that talk about the current state of art about using graph database and discrete global grid systems to improve geospatial search. These papers helped shape the direction of this research.

Research on the use of graph database in geospatial data to leverage graph network properties were only a recent trend. Kanaka et al. [6] explored in their 2015 paper the power of graph databases in holding spatial data sources by implementing a graph-oriented spatial database with the use of the Neo4j graph database platform and OpenStreetMap geographic datasets to create a geographical information system (GIS). Neo4j was connected to an API service to allow queries to it and the results were served to an HTML application with the Leaflet Mapping library to help present geographic data. This paper served like a technical white paper in which it focused on showing a working implementation of these technologies that one can use to create their own GIS. It also demonstrated the use of graph databases to store complex geographic information like customer reviews of restaurants in an area.

In 2017, Roberto et al. [9] acknowledged the benefit of using graph databases in solving specific problems that were ill fitted to being implemented on relational models and chose to study the behavior of graph-oriented spatial databases. They compared Neo4j-spatial, a spatial extension of Neo4j to PostGIS, which is a PostgreSQL-based spatial database and used their performance on various spatial queries as a metrics of comparison. At the time of writing, the latest version of Neo4j-spatial was able to do proximity queries (closest geometry given a radius of distance), distance queries (geometries within a certain radius), and bounding box queries (get all geometries inside a bounding box) on par with PostGIS but there were some disadvantages like not having a single language binding and having little development support due to it being a new technology.

Recently this year, Bondaruk et al. [2] published a study about the current state of art of discrete global grid systems. The researchers here focused on two technologies, H3 and OpenEAGGR. They compared both technologies based on how well they fared in following the guidelines specified by the OGC DGGS Standard Data Protocol, as well as their performance in operational and practical applications. They both failed to meet the guidelines of OGC but they managed to fulfill at least some to be able to call themselves DGGS technologies. In terms of their operational proficiency, H3 and OpenEAGGR lacked some functionalities but OpenEAGGR had a more complete spatial analysis, data query, and broadcasting implementations. However, the researchers pointed out that even with missing functionalities, new features and corrections in H3 are being implemented regularly and H3 enjoys a far more rapid development compared to OpenEAGGR.

## 4 METHODOLOGY

The proposed methodology consists of setting up a PostgreSQL database with the PostGIS extension installed and a Neo4j graph database with the Uber H3 library added as a plugin. The dataset, which consists of location points of various amenities around Quezon City, Metro Manila, are then imported to both databases. To evaluate the performance of each database, multiple bounding box search queries are performed on the dataset, and the execution times of each search query are recorded.

To perform the experiments, the collection and pre-processing scripts were developed on *Python 3.0* and *Jupyter 4.4.0*. For the database systems, *PostgreSQL 12.1* and *PostGIS 3.0* were used for the relational SDS while *Neo4j 3.5.12* and *Uber H3 3.6* (with *OpenJDK 13* to handle its Java bindings) were used for its graph counterpart. The experiments were run in a machine running *macOS Catalina v10.15.1*. The details of the experiment are further discussed in the following subsections.

### 4.1 Data Collection and Preprocessing

To create the dataset for this research, a geofence is created surrounding Quezon City as seen in Figure 5. All of the spatial amenities within this geofence are promptly extracted from OpenStreetMaps (OSM) and Google Maps. The set of amenities from both OSM and

Google Maps are then merged into one single dataset, eliminating duplicates in the process. After extraction and merging, the final dataset is stored in CSV format.

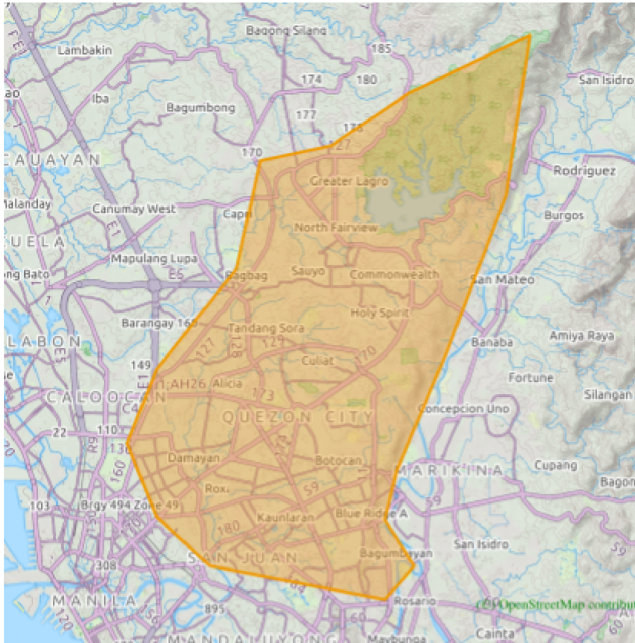


**Figure 5: Geofence used for this experiment surrounding Quezon City**

The resulting dataset contains entries that have no names or some form of identification. In order to fill them up, a Python script is used to search for the coordinates of these unidentified points. Using Google Places API, a search is performed on nearby points with the same type within a 100 m range. After adding these points, the initial unidentified point is then removed from the dataset. After identifying all mystery points inside the dataset, another script is executed to go through the dataset and check if the location matches its type by cross-referencing the location information to Google Places API and renaming the said point if it is a different place. This script corrects amenities placed with the wrong location name. All in all, there are 19,148 entries, each representing a single amenity, with name, type and coordinates as its attributes. A sample snapshot of the dataset is shown in Figure 6.

Naive random over-sampling is then performed on the dataset. Additional synthetic location points are generated to see the impact of having a larger dataset on query performance. After over-sampling, the number of entries in the dataset is increased to 516,996.

## 4.2 Experimental Setup

Neo4j provides a desktop application which allows for a quick setup of a graph database server, and provides an interface for performing queries using the Cypher query language. For this experiment, a local graph database is created, and the dataset is then imported. Each row or entry in the dataset is added as a node in the graph. A single `Amenity` node is composed of properties `name`, `type`, `lat` and

| | lon | lat | amenity | name |
|---|---|---|---|---|
| 15404 | 121.038514 | 14.630712 | restaurant | WOW Organic Restaurant |
| 13419 | 121.040184 | 14.641927 | restaurant | Chi Em Gai Vietnamese Restaurant |
| 7133 | 121.079214 | 14.606948 | restaurant | Heaven's Barbeque |
| 10174 | 121.056905 | 14.648566 | restaurant | Box O' Rice |
| 18001 | 121.050700 | 14.602511 | atm | Bank Of The Philippine Island |
| 410 | 121.026817 | 14.659972 | school | San Francisco National High School |
| 13343 | 121.002564 | 14.630416 | restaurant | Wholesale Dimsum Price |
| 9243 | 121.034219 | 14.628428 | restaurant | McDonald's |
| 13486 | 121.041169 | 14.630673 | restaurant | Burger Machine |
| 14615 | 121.052655 | 14.602667 | restaurant | Can We Pho |

**Figure 6: Snapshot of the Dataset**

`lon`. For visualization purposes, a `Type` node consisting of a `name` property is also added. A `HAS_TYPE` relationship is then created between `Amenity` and `Type`. The code for the setup is presented in Listing 1. A sample graph of parking amenities is shown in Figure 7.

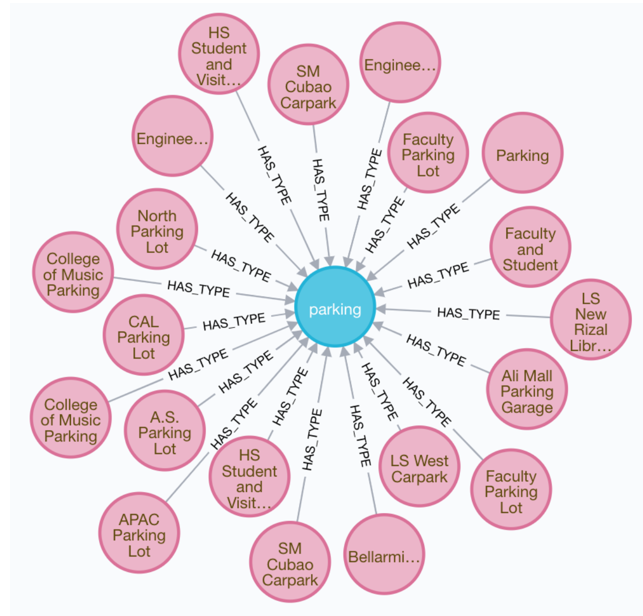


**Figure 7: Parking Amenities in a Graph Database**

```
1 LOAD CSV WITH HEADERS FROM 'file:///dataset.csv' AS line
2 CREATE (:Amenity { name: line.name, lon: line.longitude,
      lat: line.latitude, type: line.amenity });
3
4 LOAD CSV WITH HEADERS FROM 'file:///types.csv' AS line
5 CREATE (:Type { name: line.name });
6
7 MATCH (a:Amenity),(t:Type)
8 WHERE a.type = t.name
9 CREATE (a)-[:HAS_TYPE]->(t);
```

**Listing 1: Neo4j Setup**

Neo4j already has built-in support for handling spatial values. In order to perform a bounding box search query on the dataset, the `lon` and `lat` properties need to be converted to a single `point` property, which is a spatial geometry type provided by Neo4j. Each point is associated with a specific Coordinate Reference System (CRS). For geographic coordinates, the WGS-84 is used as the CRS. The query for this process is presented in Listing 2.

```
MATCH (a:Amenity)
SET a.location = point({latitude: a.lat, longitude: a.lon
    , crs: 'WGS-84'});
```

**Listing 2: Point Conversion**

To integrate H3 with Neo4j, a plugin which maps the set of H3 Java bindings to Neo4j procedures is implemented. In order to perform a bounding box search query using H3, two main procedures are created. The first procedure, `hexAddress()`, makes use of `geoToH3()` from the H3 API. This procedure allows the retrieval of the hex address given the coordinates and a resolution value. For this experiment, a resolution of value 9 is used. The second procedure, `polygonSearch()`, accepts a set of coordinates as a parameter, creates a region using these coordinates, generates a set of hexes within that region, then performs a search on each of the hexes. It uses `polyfill()` from the H3 API, which handles the filling up of hexes inside the region.

Using `hexAddress()`, a new property called `hexAddr` is added for each `Amenity`. The procedure `polygonSearch()` uses this value instead of the coordinates when performing a search query. Additionally, an index is also created in order to speed up the search. The process is presented in Listing 3.

```
MATCH (a:Amenity)
CALL com.h3.hexAddress(a.lat, a.lon, "9") YIELD value
SET a.hexAddr = value;

CREATE INDEX ON :Amenity(hexAddr);
```

**Listing 3: Hex Address Generation**

For the performance tests, a total of 10 search queries are performed. For each query run, 10 bounding boxes are randomly generated. Each bounding box search are executed on all three setups, namely Neo4j Spatial, Neo4j + H3 and PostGIS. The PostGIS setup are not discussed in detail, as it simply requires enabling the PostGIS extension in the PostgreSQL installation. Results of the experiment are presented in 5. Listings 4, 5 and 6 provide sample bounding box search queries.

```
MATCH (a:Amenity)
WHERE point({latitude:14.620510, longitude:121.013297}) <
     a.location < point({latitude:14.674593, longitude
    :121.095295})
RETURN DISTINCT(a.name), a.type;
```

**Listing 4: Neo4j Spatial Search Query**

```
CALL com.h3.polygonSearch([
  {lat:14.620510, lon:121.013297},
    {lat:14.674694, lon:121.014557},
    {lat:14.674593, lon:121.095295},
    {lat:14.620204, lon:121.094549}
],[{}]) YIELD nodes
UNWIND nodes AS a
RETURN DISTINCT a.name AS name, a.type AS type;
```

**Listing 5: H3 Polygon Search Query**

**Table 1: Query Results (19,148 location points)**

| Query # | PostGIS | Neo4j-spatial | Neo4j + H3 |
|---|---|---|---|
| 1 | 45 ms | 257 ms | 97 ms |
| 2 | 26 ms | 155 ms | 90 ms |
| 3 | 26 ms | 150 ms | 94 ms |
| 4 | 19 ms | 130 ms | 88 ms |
| 5 | 20 ms | 110 ms | 83 ms |
| 6 | 21 ms | 141 ms | 82 ms |
| 7 | 20 ms | 110 ms | 76 ms |
| 8 | 19 ms | 102 ms | 71 ms |
| 9 | 18 ms | 97 ms | 72 ms |
| 10 | 21 ms | 120 ms | 74 ms |

```
SELECT name, type, lon, lat FROM amenities WHERE
    amenities.geom && ST_MakeEnvelope(120.850250,
    14.473059, 121.217164, 14.775077, 4326);
```

**Listing 6: PostGIS Search Query**

## 5 RESULTS AND DISCUSSION

The execution times of the bounding box search queries performed on the original dataset are presented in Table 1. The graph of the results is shown in Figure 8. Observing the results above, one can say that integrating H3 with a Neo4j graph database improves the search execution time significantly. Based on the 10 query runs, Neo4j + H3 had an average execution time of 82.7 ms, compared to Neo4j-spatial's 137.2 ms. PostGIS performed best with an average execution time of only 23.5 ms.

The execution times of the bounding box search queries performed on the larger dataset are presented in Table 2, and the graph of the results is shown in Figure 9. In the larger dataset, PostGIS performed the worst, with an average execution time of 7.06 seconds. The performance of Neo4j-spatial and Neo4j + H3 were very close; their average execution times are 2.9 and 2.5 seconds, respectively. On the larger dataset, both graph database setups performed better than PostGIS. This might entail that graph databases are generally better in terms of scalability.

Both Neo4j and PostGIS have some sort of caching enabled by default, so it is expected that the first few query runs have slightly longer execution times. Leaving caching on during the experiment was intentional, so that the obtained results reflect how search queries really perform in production environments, where enabling caching is almost always recommended.

## 6 CONCLUSION

The researchers were able to produce a working implementation of Neo4j that uses H3 for spatial queries. By measuring the query execution times, it was shown that, compared to a relational database, a graph database integrated with H3 can perform better at scale. As part of future work, one can perform some tests on a dataset with multiple tables and relationships. Another aspect of the study that one can explore is to perform some tests where there is an increasing number of polygon points for each search query, and check if it impacts database performance.

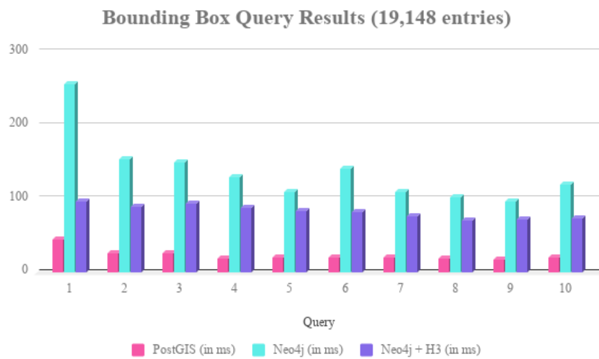


**Figure 8: Query Results (19,148 location points)**

**Table 2: Query Results (516,996 location points)**

| Query # | PostGIS | Neo4j-spatial | Neo4j + H3 |
|---|---|---|---|
| 1 | 7146 ms | 4943 ms | 2602 ms |
| 2 | 7165 ms | 3282 ms | 2594 ms |
| 3 | 7080 ms | 3133 ms | 2571 ms |
| 4 | 7069 ms | 3234 ms | 2568 ms |
| 5 | 7004 ms | 2604 ms | 2513 ms |
| 6 | 7037 ms | 2474 ms | 2545 ms |
| 7 | 7059 ms | 2446 ms | 2425 ms |
| 8 | 7043 ms | 2473 ms | 2366 ms |
| 9 | 6988 ms | 2273 ms | 2388 ms |
| 10 | 6975 ms | 2195 ms | 2354 ms |

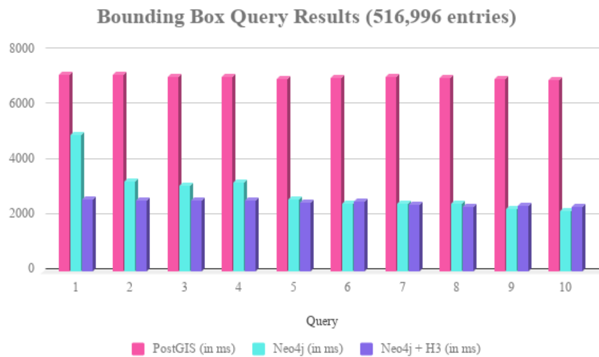


**Figure 9: Query Results (516,996 location points)**

The resulting implementation can be a base point into creating a fully working graph-powered GIS. With H3 enjoying rapid development and a growing community, it would be a worthy technology to consider for various spatial operations. With its ability to pinpoint locations via smaller resolutions, it can be used for various applications that cover transportation or spatial services. It can also be used to provide a heat map laid on top of a map that leverages graph networks in order to provide more meaningful relationships between grids. With a world that moves toward on-demand economy and an increasing need of real-time information to aid in decision-making, a graph-network powered GIS can bring improved performance to various applications.

## REFERENCES


[1] E. Baralis, A. Dalla Valle, P. Garza, C. Rossi, and F. Scullino. 2017. SQL versus NoSQL databases for geospatial applications. In *2017 IEEE International Conference on Big Data (Big Data)*. 3388–3397. https://doi.org/10.1109/BigData.2017.8258324

[2] Ben Bondaruk, Steven Roberts, and Colin Robertson. 2019. Discrete Global Grid Systems: Operational Capability of the Current State of the Art. In *2019 Spatial Knowledge and Information Canada*, Vol. 7.

[3] Isaac Brodsky. 2018. *H3: Uber's Hexagonal Hierarchical Spatial Index.* https://eng.uber.com/h3/

[4] Open Geospatial Consortium. [n.d.]. Discrete Global Grid Systems Abstract Specification. ([n. d.]). https://portal.opengeospatial.org/files/15-104r5

[5] Ralf Hartmut Güting. 1994. An introduction to spatial database systems. *The VLDB Journal—The International Journal on Very Large Data Bases* 3, 4 (1994), 357–399.

[6] Tharik Kanaka, PPG Dinesh Asanka, and Pearson Lanka. 2015. GeoSpatial Intelligence on a Graph. *JMPT* 6, 2 (2015), 35–42.

[7] Jon A Kimerling, Kevin Sahr, Denis White, and Lian Song. 1999. Comparing geometrical properties of global grids. *Cartography and Geographic Information Science* 26, 4 (1999), 271–288.

[8] M. B. J. Purss, R. Gibb, F. Samavati, P. Peterson, and J. Ben. 2016. The OGC® Discrete Global Grid System core standard: A framework for rapid geospatial integration. In *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*. 3610–3613. https://doi.org/10.1109/IGARSS.2016.7729935

[9] E. Roberto, M. T. de Holanda, A. P. F. de Araújo, and M. Victorino. 2017. Geographic data in a graph oriented database: A study with Neo4j and PostgreSQL. In *2017 12th Iberian Conference on Information Systems and Technologies (CISTI)*. 1–6. https://doi.org/10.23919/CISTI.2017.7975999

[10] Kevin Sahr, Denis White, and A. Jon Kimerling. 2003. Geodesic Discrete Global Grid Systems. *Cartography and Geographic Information Science* 30, 2 (2003), 121–134. https://doi.org/10.1559/152304003100011090 arXiv:https://doi.org/10.1559/152304003100011090

[11] Byoung-Ha Yoon, Seon-Kyu Kim, and Seon-Young Kim. 2017. Use of graph database for the integration of heterogeneous biological data. *Genomics & informatics* 15, 1 (2017), 19.