AN INVESTIGATION INTO THE SUITABILITY OF GRAPH DATABASE TECHNOLOGY IN THE ANALYSIS OF SPATIO-TEMPORAL DATA

S.D. Baker Effendi **A.B. van der Merwe**
Department of Computer Science
Stellenbosch University
Stellenbosch, South Africa
{dbe@, abvdm@cs}.sun.ac.za

W.-T. Balke
Institute for Information Systems
TU Braunschweig
Braunschweig, Germany
balke@ifis.cs.tu-bs.de

February 4, 2020

ABSTRACT

Every day large quantities of spatio-temporal data are captured, whether by Web -based companies for social data mining or by other industries for a variety of applications ranging from disaster relief tiloto marine data analysis. Making sense of all this data dramatically increases the need for intelligent backend systems to provide realtime query response times while scaling well (in terms of storage and performance) with increasing quantities of structured or semi-structured, multi-dimensional data. Currently , relational database solutions with spatial extentions such as PostGIS, seem to come to their limits . However, the use of graph database technology has been rising in popularity and has been found to handle graph-like data much more effectively.

This work is motivated by the need to effectively store multi-dimensional, interconnected data and to investigate whether or not graph database technology is better suited to address this when compared to the traditional relational database approach. Three database technologies will be investigated using this dataset namely: PostgreSQL, JanusGraph, and TigerGraph. The datasets used are the Yelp challenge dataset and an ambulance response simulation dataset from Umeå University, Sweden . The evaluation is based on how each database performs under data analysis scenarios similar to those found on an enterprise level.

Keywords Graph database · spatio-temporal data · NoSQL · Yelp dataset

1 Introduction

The collection of spatio-temporal data is commonplace today: prime examples are user-generated data from mobile devices or social platforms, streams of sensor data from static or moving sensors, satellite or remote sensing data. To make sense of this vast volume of heterogeneous data, a variety of aggregation and fusion steps have to be applied to achieve semantic data enrichment for subsequent use in value-added services. However, enabling these enrichment steps generally relies on stable and highly scalable data handling and storage infrastructures. However, with ever increasing volumes of spatio-temporal data relational database technology (even with special object-relational extensions for spatio-temporal data) is stretched to its limits and currently new paradigms commonly referred to as 'Not Only SQL' (NoSQL) databases are tried out as more scalable replacements, see. e.g., [1].
Given that a significant amount of data logged daily with geotags and timestamps (see e.g., [2]) and considering the distributed and global/local nature of such spatio-temporal data, simple *graph structures as basic representations* come quite natural: traffic information in a road network, the spreading of epidemics over adjacent geographic regions, or time series data for chains of events. In a nutshell, graphs are composed of vertices (nodes) and edges (relations) joining two vertices together. Graph databases form an integral part of a group of NoSQL database technologies offering flexible and scalable solution for many enterprise level problems. In a graph database, a vertex could be a user, business, vehicle, or city and edges could represent adjacency, roads, or location affiliations. Due to the relational nature of graphs, they are already used to model a wide variety of real-world phenomena such as physical, biological,and social information systems [3].
However, although many categories of data are interpreted as a graph structures today , traditional relational databases still lack the architecture and high-level query languages to effectively model and manipulate such structures . Graph databases are designed to represent data as attributes on vertices and edges. They are often schema-less (i.e. there is no fixed data structure) and their attributes are described as key-value pairs. Thus, a major strength of graph databases is that they excel in complex join-style queries where relational databases are notoriously inefficiently for large tables [4]. In contrast, graph databases tend to perform poorly when moving from local subgraphs to full graph aggregate queries, where relational database technology plays out its strength .
This paper investigates the suitability of which database technology is best suited to handle large-scale, spatio-temporal data in *typical data manipulation tasks under real world settings* . In particular, the investigation will cover the query speed, expressiveness of the available query languages, and computational demand on the open source graph database, JanusGraph, the open source object-relational database system, PostgreSQL, and the enterprise level graph analytics platform, TigerGraph. Two graph database technologies from two different implementations of property graph databases have been selected to measure the suitability of each implementation.
We have selected two datasets with different representations of spatio-temporal properties, complexities from additional attributes, and total size.
To allow for a fair and authoritative evaluation we use a real world data set for benchmarking. Yelp is an internationally operating business directory service and review forum. The 'Yelp dataset' is a current subset of Yelp's businesses, reviews, and user data comprising spatio-temporal data for 10 metropolitan areas across two countries with about 200,000 businesses and 6.6 million reviews. It is offered for academic use in the yearly yelp challenge [5], for a variety of data analysis and discovery tasks.
As an additional dataset to further strengthen our results, the ambulance response simulation dataset from Umeå University[1] will be used for our investigation. The medical response dataset holds the simulation results where, given a number of hospital resources such as dispatch centres and ambulances, emergency calls are responded to. The dataset records spatio-temporal properties such as the time intervals within the response life cycle and the origin and destination of the resource during the response. Other technical properties regarding the emergency itself are also recorded. The dataset only contains ambulance responses.
Our contributions can be summarized as follows : (i) modelling and importing a real world and artificial dataset into three state of the art databases, (ii) rigorously benchmarking them using various analyses that fairly represent the type of querying and data manipulation each dataset would typically undergo, and (iii) comparing the suitability of JanusGraph vs. TigerGraph by investigating their graph query language implementations.

2 Problem Setting & Research Questions

This paper tackles the problem of modelling, visualizing, and querying large-scale spatio-temporal data using traditional relational database approaches versus graph database approaches . We investigate query performance, storage and computational cost, as well ease and efficiency of simulation of real-world applications. In particular,...
How to store spatio-temporal data efficiently? Coordinates are comprised of latitude and longitude and time adds a third dimension. The accessing times of secondary storage is an issue when housing large volumes of data. The use of B-Trees, R-Trees, and a geo-graph[2] are three techniques investigated for indexing both, uni- and multi-dimensional data efficiently.
How to query a graph topology effectively? Basic graph pattern matching is popular when extracting data from graph databases, but there are a range of graph query languages implementing more complex graph patterns, too . Section 4 addresses three graph querying languages

[1] From here forth will be referred to as the "medical response dataset".

[2] A grid-based geospatial system where vertices are grids and edges connect other vertices to these grid vertices to indicate their physical location.

namely; Gremlin, Cypher, and GSQL. This section will also cover how traditional relational operators, such as union and difference, are implemented in graph pattern matching . The conciseness of querying graphs versus using traditional SQL is briefly addressed in the conclusion (Section 8)[3].

Can the respective database components be easily integrated into applications? Due to the rise of popularity in web-based applications, the ease of incorporating three typical databases into applications will be investigated with a Flask [6] back-end and Angular [7] driven front-end. This will show how appropriate it is in real-world applications and production settings to use graph database technologies and any challenges encountered during implementation versus using a classical SQL database back-end.

3 Literature Review

The following literature review aims to define the terms related to database technologies. This section begins by defining the following principles; ACID compliance and the CAP theorem. The relational model will be explained to show how classic relational database technology came to be. The shortfalls of relational databases will then be used to introduce what NoSQL technology is, and how this contrasts with the many variations of NoSQL databases including key-value, column family, and document stores. Based on our literature review, we will summarize the contributions of our research to spatio-temporal data mining, analysis, and visualization.

3.1 ACID Compliance

From an article published in *Database Guide* [8]; Atomicity, Consistency, Isolation, Durability (ACID) are the properties that can guarantee that transaction based databases perform reliably. These properties concern themselves with how databases recover after failures such as a transaction, system, or media failure.

Atomicity. A single transaction could be comprised of one or more steps, and if one of them fails then the transaction will only be partially successful. Atomicity guarantees that if one step fails, the whole transaction fails and the state of the database will be rolled back to a state prior to the transaction.

Consistency. Data may need to conform to a variety of rules and constraints, especially with regard to relational databases. If any data does not conform to these rules or the given schema, it will not be consistent with the rest of the database. Allowing this data to become part of the database implies that we cannot guarantee our predefined rules. This is one of the properties that NoSQL databases relax on in order to provide higher availability and partition tolerance. A consistent database ensures that any data being inserted into the database follow all predefined rules or else the transaction will be rejected.

[3]To effectively measure the difficulty of learning and writing complex queries would require experiments involving developers new to these languages and measuring their progress in terms of error rates, which is clearly beyond the scope of this paper.

Isolation. To avoid transaction conflicts, each transaction needs to be performed in isolation. This means that no transaction will affect another, and if necessary, a given transaction will take precedence over another in case of a conflict.

Durability. For a database to be reliable in the event of any of the aforementioned failures, it should be the case that if a transaction has been committed, it needs to be able to guarantee that the transaction has been stored permanently. Durability along with atomicity work together to keep a database reliable when faced with a failure either during or directly after a transaction has been committed.

A database that is ACID-compliant is robust against data being corrupted during a failure and guarantees that only successful transactions are processed.

3.2 CAP Theorem

Gilbert & Lynch [9] explain that, in distributed systems, there is often a trade-off between consistency, availability, and network partition tolerance (CAP). Eric Brewer presented this theorem in the context of geographically separated datacenters supporting web services, implicating that a multi-node database cannot ensure all three properties under CAP. Following this web service context, the three properties are described as follows:

- **Consistency** means that the server will return the correct response to each request.
- **Availability** will guarantee that every request will receive a response.
- **Partition tolerance** refers to the system implementation allowing the database to be split into multiple groups unable to communicate with one another.

Gilbert & Lynch, when referring to these properties, say:

> "CAP states that any protocol implementing an atomic read/write register cannot guarantee both safety and liveness in a system prone to partitions."

The CAP theorem illustrates the trade-off between safety and liveness when considering an unreliable system. For the safety property to hold, every point in each step of a transaction should be atomically consistent. For the liveness property to hold, a desirable outcome should result as long as the execution continues for long enough.

3.3 Relational Databases

The relational model's structure stores data in tuples with relations represented as tables. Each tuple has attributes, primary and candidate keys. Each attribute has a domain which is the data type of the attribute's values. The tuples became rows and attributes the columns for each row under their respective tables. Furthermore, the degree refers to the number of columns in a table whereas cardinality the number of rows in a table. An illustration of this can be seen in figures 1 and 2.

The high level language concepts introduced are the algebraic operators we see in SQL today such as `SELECT`,

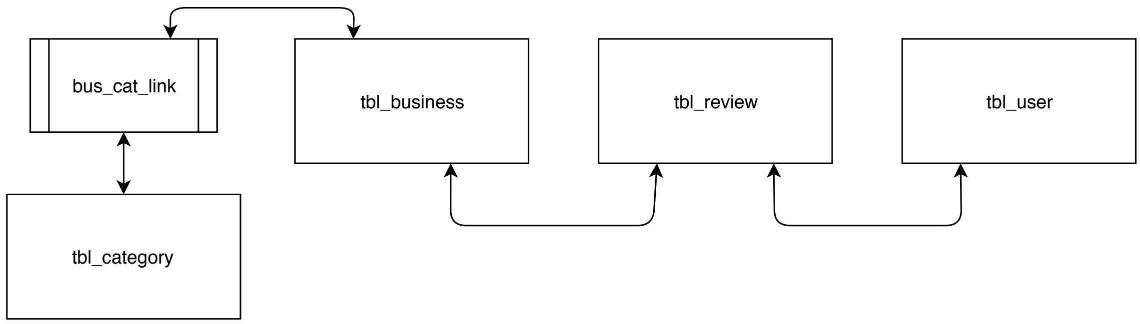

Figure 1: A simplified representation of the Yelp dataset in a relational database context illustrating tables and their relations.

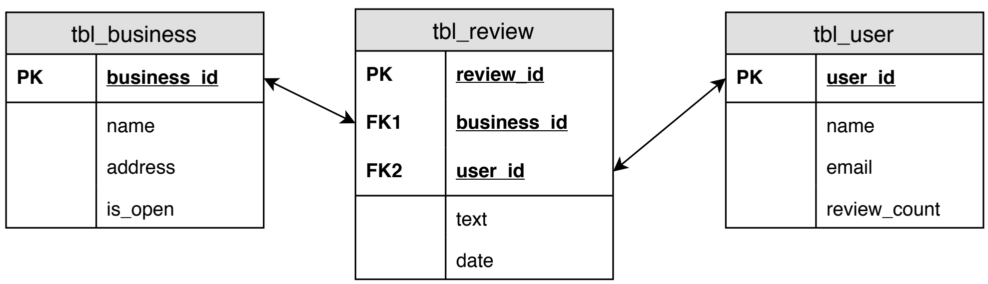

Figure 2: A closeup on the foreign key and primary key structure within the business, user, and review tables.

`UNION`, `JOIN`, etc. These operators allow users to convert relations (tables) into other relations (resulting in tables as output).
Since the 1970's, relational database management systems have been the leading database technology with a variety of capabilities and language dialects. This was until NoSQL obtained traction among large Internet corporations and was implemented in the form of distributed, non-relational databases in the 2000s [4].

3.4 NoSQL Databases

NoSQL databases were created to address the shortfalls in classic relational databases when used for data of large volume, variety, and velocity [10]. Many of these issues were introduced with the increasing popularity of the Internet and systems hosted in the cloud. Relational databases scale poorly over multiple nodes and on large volumes of data, which is why SQL databases could no longer be used as the data solution for major Internet companies such as Facebook, Amazon, and Google.
NoSQL database technologies are non-relational and use non-SQL languages to manipulate and query the data. NoSQL is designed to run on multiple nodes (distributed systems), scale horizontally, and some even implemented technologies such as massively parallel processing (MPP) [11] or in-memory processing (such as H-store [12]). NewSQL is a distributed SQL technology which implements NoSQL features to achieve ACID-compliance (alongside partition tolerance). They will not be referred to as relational database technology when mentioned in this paper [10].
A main challenge for non-relational database systems is the conflict between high availability in distributed systems and remaining transaction based and ACID-complaint (see Subsection 3.1). This results in NoSQL DBMS to typically choose availability and partition tolerance over consistency (see Subsection 3.2) and have become BASE (Basically Available, Soft-state, Eventually consistent)-type systems to compensate [13]. Not all NoSQL databases follow this trend and there are some vendors such as Neo4j and OrientDB that are fully ACID compliant [8].
NoSQL databases fall under a broad scope and, in this paper, they will be classified under four categories; key-value stores, document stores, column family (or wide-column) stores, and graph databases.

Key-Value Store The basis of a key-value store is that the data is organized in the form of key-value pairs. The key is typically a numeric or string value whereas the value can be any data object or collection of data objects (where a list would be represented by multiple entries of the same key). Each document has a unique identifier with a list of attributes forming the key-value paired data. A simple example of this can be seen in Figure 3.
Castellano [14] explains that key-value stores are designed to be lightning fast, simple, and unstructured. They expose three operations namely; `PUT`, `GET`, and `DELETE`, with some implementations adding an additional operation, `SEARCH`, that matches keys or key-value pairs given a specified search expression and key namespace. These

Businesses	
Key	Attributes
0	Name: "McDonald's" Categories: "Fast-food" Categories: "Takeaway" Stars: 2.5 Open: true
1	Name: "KFC" Categories: "Fast-food" Categories: "Restaurant" Stars: 3.0 Open: true

Figure 3: An example of business entries stored in a key-value store.

Yelp Keyspace	
Users Column-Family	Businesses Column-Family
RowID: 0 FirstName: "David" Email: "david@sun.ac.za" ReviewCount: 8	RowID: 0 Name: "McDonald's" Categories: "Fast-food" Categories: "Takeaway" Stars: 2.5 Open: true
RowID: 2 FirstName: "Kyle" Email: "kyle@gmail.com" ReviewCount: 3	RowID: 4 Name: "KFC" Categories: "Fast-food" Categories: "Restaurant" Stars: 3.0 Open: true

Figure 4: An example of user and business column-families.

databases are decentralized in nature so struggles to provide the transactional guarantees of ACID.

Document Store As the name suggests, document store databases store their data in the form of documents. Typically, the document formats are JSON, XML, PDF, etc. Unlike key-value stores, document stores are semi-structured and both keys and values are fully searchable [10]. Document stores are still schema-less and are well suited for data that is dissimilar such as those which do not fit well in a table with set columns [15] or require many "nulls". These databases, like key-value stores, do not perform well if the data is highly relational and requires some kind of normalization.

Column Family Store Manoj [15] explains column families as more structured data stores in that they store data in columns. They are a hybrid row/column store but store their data in distributed architectures instead of tables. Data is stored in a column-family (analogous to a table in SQL), but column-families have no relation to one another unlike in relational models. Column-families are less flexible than the previous key-value and document store databases as one will have to predefine a column family's attributes and are grouped together under keyspaces. A column-family model is illustrated in Figure 4.

3.5 Graph

Graph databases, the focus of this paper, applies a graph structure to store and manipulate data. Data can be stored as key-pair attributes on both the vertices or edges of the database (as can be seen in Figure 5). Graph databases are considered as NoSQL databases but are unique in that they are significantly more structured than the NoSQL models aforementioned, can handle both semi-structured and structured data effectively, and concern themselves with relations. Edges can be unidirectional or bidirectional which may add more information about the kind of relation between two edges [10]. Graph databases typically persists the data using a storage back-end. This can be a NoSQL database, as in JanusGraph's case, or a relational database, as in Facebook Tao's case [16].

Graph databases are strong at finding patterns and revealing information about the relationship between vertices rather than aggregate queries on the data itself.
Within the category of graph databases we find three main subcategories; property (or attributed) graphs, hypergraphs, and resource description framework (RDF) triples [3]. The type of graph databases focused on in this investigation are property graphs.

3.6 Summary

From the discussion above, we face the following questions:

> **What would make a non-relational solution more suitable than a relational solution?**

Moniruzzaman & Hossain [10] elaborate on the shortcomings of relational database technologies in terms of performance when distributed over geographically diverse datacenters due to their strong emphasis on consistency while maintaining ACID-compliance. As emphasized by Chen [3] & Makris et. al. [1], due to the trend of services hosted in the cloud and velocity of incoming data and requests, a horizontally scalable, distributed system would be best suited to address these issues – which would suggest a NoSQL solution.

> **When compared among other NoSQL databases, what makes a graph database solution stand out?**

Spatial data is often highly relational due to the ties between vertices representing physical locations and their association with other entities in the database. This means the data will be closely related and that trend will follow in how the queries are written. Chen [3] explains that the queries in a graph database are designed to reveal hidden trends among the relationships in the data (especially when visualized) rather than within the data itself adding additional value in this implementation.
Time adds an ordinal relation among all entries in the data that hold this property. Graph databases are the most

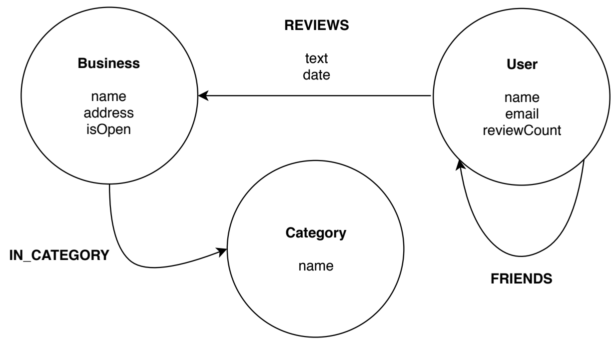

Figure 5: A simplified representation of the Yelp dataset in a property graph database context illustrating vertices, edges, and their attribute keys.

relational among the NoSQL databases and have the ability to be queried to answer complex questions on complex data. These complex questions would be multi join-style queries which are typically expensive, but due to a graph's relational architecture, graph databases should be better suited in comparison to other NoSQL database solutions in handling semi-structured data [4].

Why compare a relational solution to a graph database and not another NoSQL solution?

Relational databases should come close to what is required in our setting due to their ability to manipulate data based on table relations, but their lack of scalability may impact overall benchmark performance negatively when tested on large volumes of the kind of data we consider, especially due to how join-style queries scale with table size [4].
Makris, et. al. [1] compares how MongoDB (with GeoJSON) and PostgreSQL (with PostGIS) perform against one another on spatial data and show that PostgreSQL outperforms its NoSQL rival. Considering our discussion in subsection 3.4, it does not come as a surprise, given the lack of suitability of relational queries in a document store databases. In general, complex queries perform poorly in databases such as MongoDB. One important observation that this paper does point out is how well PostgreSQL performs, despite the volume of data. This suggests that the scalability issue of relational databases was not as prominent in this comparison as much as how well these complex queries were handled.

Why spatio-temporal data?

Spatio-temporal data is constantly being generated by GPS-equipped devices everyday [2][17]. This data has important applications in epidemiologic [18], marine, disaster relief, and social data investigation [19]. Few papers address a graph database solution for spatio-temporal data, but due to its importance and abundance it would be important to seek an appropriate technology to store, model, and visualize these datasets.

In conclusion The findings in this paper would answer which technology, graph or relational, would perform the best not only in benchmark performance but also in suitability in terms of querying, modelling, and visualizing of spatio-temporal data. This answer will be useful if one would like to perform analysis on not only a spatio-temporal dataset, but also investigate relationships in the data with a suitable database, and perform/write efficient queries.

4 Graph Querying Techniques and Languages

The following section explores the three popular graph querying languages – two of which are used in the databases being investigated in this report – and properties and techniques of graph querying. Graph query languages have a notion of traversing the graph and accumulating results from pattern matching. This makes it very different to traditional SQL and is the main factor in contributing to the learning curve – how to conceptualize graph expressions. These languages are still fairly new with some of the latest additions being GraphQL by Facebook (released in 2015) [20] and GSQL by TigerGraph (released in 2017) [21]. As techniques of graph traversal becomes more refined, one can expect a few more query languages to be developed.

4.1 Languages

Gremlin Gremlin is a functional, data-flow query language running in its own virtual machine [22]. Due to how Gremlin is compiled, it can be written in both an imperative and declarative manner and also be embedded in a host-language [23]. Imperatively querying with Gremlin involves explicitly stating the traversal pattern, whereas declarative allows the traverser to decide. The benefits of using an embedded query language is that security risks from string concatenation and sanitizing input is handled automatically.
Gremlin integrates into multiple vendor's products (in our case, JanusGraph) and benefits from Turing-completeness. Each transaction is local to a given thread which means that a transaction is automatically created in that thread without having to explicitly call a create method. Transactions are atomic and support rollbacks.

```
g.V().has("User", "user_id", "qUL3CdRRF1vedNvaq06rIA")
        .outE("REVIEWS").has("stars", gt(3)).order().by("date", desc)
            .as("stars", "text")
        .inV().has("location", geoWithin(Geoshape.circle(35.15,-80.79, 5)))
            .as("business_id")
    .select("stars", "text", "business_id")
    .by("stars").by("text").by("business_id")
```

Figure 6: Returns all the reviews – with a star rating larger than 3 – by the specified user ID within a 5km radius of 35°15N 80°79W ordered by date descending. The ability to query and index spatio-temporal properties is provided by JanusGraph's integration with a search engine such as ElasticSearch.

Cypher In an effort to create an easier graph language to learn, Cypher (see the openCypher project [24]) was created as a very SQL-like query language. Unfortunately, while it is easier to pick up, it is not Turing-complete [25]. It should be noted that whether the fact that native Cypher is not Turing-complete should be considered as a drawback, is a problem dependent consideration. SQL, as per the SQL92 standard, is not Turing-complete which implies that being Turing-complete is not necessary for almost all business needs . For most ordinary querying, not having a Turing-complete query language poses few to no major drawbacks at all. Examples of algorithms, commonly used on an enterprise level, which cannot be implemented in native Cypher, are given by PageRank/Label Propagation style algorithms [25]. The workaround is that Cypher provides calls and algorithms libraries to conduct these. Many important algorithms affected by these limitation are implemented in the Neo4j Graph Algorithms library[4]. An example of using this library is given in Figure 7. A problem with a hard-coded PageRank-style algorithm is given by the limitation of not having full control over the number of iterations and terminating conditions. Ultimately, one should consult the documentation before deciding if a Turing-complete graph query language is required.

```
CALL algo.pageRank("Page", "LINKS", {
    iterations:20,
    dampingFactor:0.85,
    write: true,
    writeProperty:"pagerank"
})
```

Figure 7: An example from the Neo4j documentation [26] on using the PageRank algorithm from the Neo4j Graph Algorithms library.

Use of ASCII-art helps to create more intuitive and easy to visualize queries, e.g. edges are denoted by `-->` or `--[..]->` and vertices by the use of parenthesis as in `(b:Business)`.
Cypher is a declarative language, so there is less control over how the traverser goes about each step. This means that Cypher has less flexibility and could perform worse than Gremlin. Another criticism in terms of performance is that Cypher compiles into Gremlin which is then executed by the TinkerPop engine [27] and this overhead should still be reduced. At the time of writing, Neo4j has been gathering more attention around Cypher, so the performance issues may improve significantly in the near future.
Cypher has the advantage of being able to express complex traversals in a simple and more intuitive manner than Gremlin. Due to the open-source nature of Cypher and, how closely it is linked to Gremlin, it benefits from the same portability advantages by being able to integrate with multiple vendors. TinkerPop 3 supports Cypher[5] so any TinkerPop 3 enabled graph database can be queried with either language and thus benefits from the ability that either query language can be used depending on if a high-level traversal or simple query is required. An example of a useful query on the dataset we consider are given in Figure 8.

```
CALL apoc.periodic.iterate(
    "MATCH (p1:User) RETURN p1",
    "MATCH (p1)-[:REVIEWS]->(r1)
           -->()<--
           (r2)<-[:REVIEWS]-(p2)
        WHERE id(p1) < id(p2)", {
        batchSize:100,
        parallel:true,
        iterateList:true
    }
);
```

Figure 8: Returns all users who have reviewed the same businesses as a given user. The query iterates through all users. Note how the return match verifies that the user `p1` does not match user `p2`.

GSQL GSQL is described as a SQL-like language which is the conceptual descendent from technologies such as Gremlin, Hadoop MapReduce, SQL, Cypher and SPARQL [28]. GSQL, like Gremlin, is a Turing-complete graph query language that accumulates data along a traversal. One limitation that Gremlin has compared to GSQL, is that Gremlin cannot simultaneously group two tables by

[4] `https://neo4j.com/docs/graph-algorithms/3.5/labs-algorithms/`

[5] https://github.com/opencypher/cypher-for-gremlin

separate group-by attributes[6]. GSQL achieves this by providing the ability to define multiple grouping accumulators and can use these accumulators to accumulate data based on varying criteria within the same steps.
Accumulators are variables which accumulate information over a graph traversal and come in two major groups namely, scalar and collection. Scalar accumulators such as `OrAccum` or `SumAccum` store a single value. Collection accumulators such as `ListAccum` or `SetAccum` store a set of values or, as is the case with `ListAccum`, can nest accumulators. An example of a GSQL query using accumulators can be seen in Figure 9.
In GSQL, there is a large emphasis on creating a language that enables massively parallel processing on queries. The vertex and edge blocks in GSQL queries indicate independent computations separated by incoming vertices or edges referred to as guarding conditions. These blocks are pieced together by the output of one block being the input to another. Control flow can be handled by if-then-else or while statements, allowing for subsequent blocks using dynamically calculated input.
As with Gremlin, there is an emphasis on a strong, functional programming style. One has the ability to define named, parameterized queries which is analogous to creating a function with arguments. These parameterized queries can then be called by other queries enabling the re-use of code. As is the case with Gremlin and Cypher, TigerGraph allows for the conversion of Cypher to GSQL for those migrating from their competitor, Neo4j [29].

4.2 Graph Pattern Matching

Graph pattern matching is an example of *declarative* (descriptive) querying. Basic graph patterns follow the structure of the graph to query. A basic graph pattern for a property graph[7] is a graph where variables appear on the edges and vertices. A *match* for a basic graph pattern is then mapped against the graph being queried. The variables in the basic graph pattern subgraph is matched to selected values or constants in the original graph and returned as a result [30]. An example of the pattern produced by Figure 6 can be seen in Figure 10.
Complex graph patterns extend on basic graph patterns by including the traditional relational operators used for sets such as union, difference, optional, and filter. These operators are described next.
Projection. A projection returns a subset of data from the accumulated results of the pattern match. An example of this is to return only the stars from reviews between a user and business and exclude the text and edge IDs.
Join. The join of two basic graph patterns corresponds to the function of a natural join in classic relational query languages such as SQL. Since the output of a basic graph pattern is the result of the variables specified on the graph pattern, the output of a join between two basic graph patterns is the union of their output variables.
Union and difference. The union of two basic graph patterns is satisfied when one pattern or the other satisfies the pattern match. The difference of two basic graph patterns where the set of matches in the one are not in the set of matches in the other.
Optional. Optional works much the same as *join*, but instead of discarding the results from the evaluation which cannot be joined, the results from both matches are kept. This allows data with incomplete or unavailable properties to remain in the output.
Filter. The filter operator restricts the matches over which the traversal is performed. In practice, these filtering criteria vary in complexity with the ability to search over regular expressions (when querying string data), between dates (when querying temporal data), and over a radius (when querying spatial data).

4.3 Navigational Queries

Angles, et. al. [30] describes navigational queries as queries where the length of the traversal is potentially arbitrary such as *path queries*. Path queries are the most basic navigational queries, where one is only interested in the results accumulated when traversing from a source to a destination. Path queries are useful when looking at friend-of-a-friend relations between users in social networks and find applications in route-finding [31].
An example of one such query can be seen in Figure 11 and 12.
Navigational queries that try to match no-repeated-node or edge paths problems are typically NP-complete. Due to this, it is often necessary to add additional limitations on the pattern to be matched or use imperative querying techniques. Another common path traversal includes shortest paths from one vertex to another or path existence queries. It is clear that path traversals may be complex and so it is important to bound queries using constructs such as `repeat...times(x)`, in Gremlin, to limit the search space.

5 Design and Architecture

The design of the software, architecture, and technologies used to create the web-application and databases are described in the following section. The analysis being performed by the web-application, how it performs each analysis, and how the back-end queries each database is discussed.

5.1 Database Technologies

This section covers the underlying technologies used to implement the databases being benchmarked. This is important to note in terms of how portable each technology is, how difficult the configuration is before each database can be used for a given project, or limitations on performance or hardware requirements due to software requirements. While each database technology has varying capabilities, in terms of being supported for a given operating system,

[6]Note that Gremlin is able to group two tables by separate group-by attributes, but it would need to do this while using the `store` step between groupings due to the dataflow architecture of the language.

[7]Which is the type of graph being investigated in this paper.

```
CREATE QUERY getNearbyBusinesses(DOUBLE lat, DOUBLE lon, DOUBLE distKm) FOR GRAPH YelpGraph {
  SetAccum<STRING> @@vSet;
  @@vSet += getNearbyGridId(distKm, lat, lon);
  Grids = to_vertex_set(@@vSet, "Geo_Grid");

  bus =
    SELECT b
    FROM Grids:s-(Business_Geo:e)-Business:b
    WHERE geoDistance(lat, lon, e.LATITUDE, e.LONGITUDE) <= distKm;
  PRINT bus;
}
```

Figure 9: A GSQL query that returns business vertices nearby the specified coordinates which lie within the radius `distKm`.

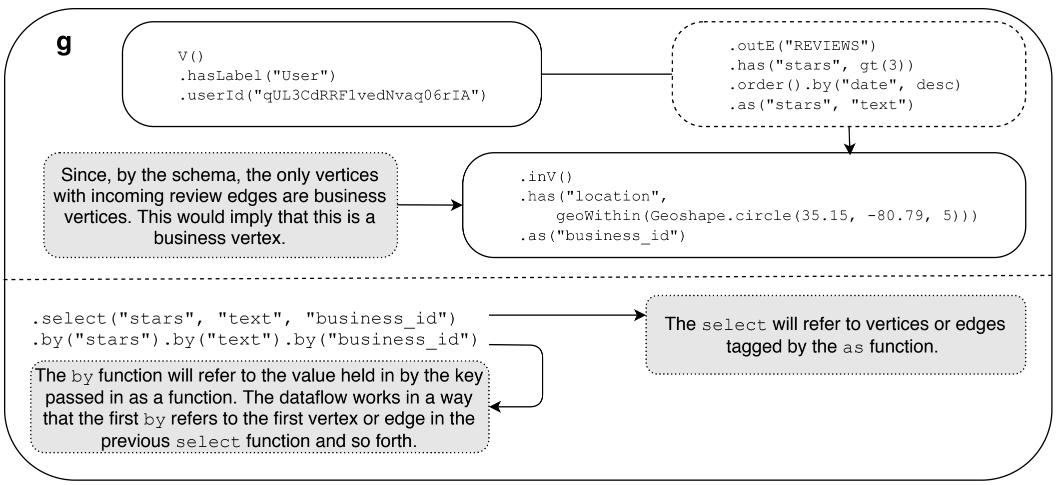

Figure 10: An illustration of the graph pattern produced by Figure 6. The top half shows the graph traversal pattern and the bottom half how the output is projected from the matched vertices and edges.

```
MATCH (u:User)-[:friends*]->f
WHERE f <> u
RETURN f
```

Figure 11: Simple friend-of-a-friend query written in Cypher.

```
g.V().hasLabel("User").as("u")
        .out("FRIENDS")
     .hasLabel("User").except("u")
```

Figure 12: Simple friend-of-a-friend query written in Gremlin.

they each provide support for being deployed in a containerized environment.
It is important to note that JanusGraph and TigerGraph differ in an important way regarding each database's implementation of their graph schema rules. TigerGraph advertise their product as an analytics platform so these kinds of design decisions may fall back on this point. JanusGraph is schemaless, so new keys can be defined at anytime, whereas TigerGraph's schema must be defined beforehand, very much as in the case of relational databases. TigerGraph has vertex-attached accumulators if values attached to vertices during a query is important, but if this information is to be persisted it needs to be a part of the schema.

5.1.1 PostgreSQL

PostgreSQL is written in the C programming language, is an object-oriented relational database, and is queried via SQL commands. PostgreSQL, like many relational databases, is ACID-compliant and robust to transactional failures. It is built to be extensible with a variety of extensions one can install for additional features such as UUID or spatial indexing. PostgreSQL can be deployed on a variety of major operating systems such as Windows, Mac, Linux (Redhat, Debian, and a few others), Solaris, and BSD.
PostgreSQL is a mature product and has a large amount of support from its open-source community. Postgres is flexible in that it supports a variety of data types and allows the definition of own types. Founder of NewsBlur[8], Samuel

[8] `https://newsblur.com/`

Clay, mentions using Postgres for multiple years for storing millions of sites and subscriptions. Canonical[9] founder, Mark Shuttleworth, explains that, while using Postgres during the development of Launchpad, finding it "robust, fast, and professional in every regard" [32].
Many of these features and opinions of using PostgreSQL in production environments on this scale is why PostgreSQL was a reputable relational database to benchmark against. PostgreSQL 11 and PostGIS 2.5 are used in this investigation.

5.1.2 JanusGraph

JanusGraph is a highly scalable graph database that is ready to be clustered between multiple machines. It is a transactional database which supports ACID-compliance and eventual consistency [33]. It is written in Java and is thus platform independent. JanusGraph is a project under The Linux Foundation and is forked from the Titan project as a continuation of the vision in creating an open-source, scalable, highly concurrent graph database. There is support for those wishing to migrate from Titan in order to benefit from the bug fixes and additional features now supported via JanusGraph [34].
JanusGraph is largely based on the Apache tech-stack making use of technologies such as Apache TinkerPop[10], Lucene, Cassandra, Hadoop, and more. This adds to complexity when configuring JanusGraph as, for each technology plugged in, there may be configuration necessary. Gremlin is the native language through which JanusGraph is queried but, as mentioned in Section 4.1, can be extended for Cypher queries. JanusGraph benefits from optional support for advanced search capabilities and having no-single point of failure [35].
JanusGraph can store graph data via three supporting backends; Apache Cassandra, Apache HBase, and Apache Berkeley. The CAP Theorem (Section 3.2) should be taken into account when considering which of the three backends to use – this is illustrated in Figure 13.
Examples of companies who have deployed JanusGraph in production include Netflix, Redhat, Uber, and IBM [37]. The professional support, documentation, and fact that one is able to leverage all these advanced features for free is why JanusGraph is one of the graph databases used in this investigation. The configuration of JanusGraph used in this paper is with Apache Cassandra as the storage backend and ElasticSearch as the search engine for spatial and temporal query support. JanusGraph 0.4 is used in this investigation.

5.1.3 TigerGraph

TigerGraph is an enterprise level graph analytics platform developed in the C++ programming language. TigerGraph was developed with hindsight from projects such as Apache TinkerPop and Neo4j and provides features such as native parallel graph processing and fast offline batch loading [38] [39]. Unlike JanusGraph, TigerGraph was developed from scratch in order to effectively create the next generation of graph database technology. TigerGraph won Strata Data's "Most Disruptive Startup" Award for its return in this decision [40].
Some of the use cases explicitly mentioned by TigerGraph[11] are in geospatial and time series analysis. This lends itself as a promising database technology for this investigation. TigerGraph is queried using their GSQL querying language (see Section 4.1) where queries are optimized via an installation process where a REST endpoint is also generated in the process. Like JanusGraph, TigerGraph can be deployed on multi-machine clusters, but this is limited to the enterprise version of this product. TigerGraph uses Apache Zookeeper for cluster management and Apache Kafka for message queuing.
GraphStudio is a web interface which is packaged along with TigerGraph which provides an interface to write, install and visualize queries, design and export one's graph schema, and monitor database performance. This makes use of an Nginx web server [29].
For all intents and purposes, the developer edition is more than capable to perform the investigation required for this paper. There is an enterprise version that allows for additional features such as multi-machine clustering. TigerGraph 2.5 is used in this investigation.

5.2 Web-application Simulation

The web-application, called Providentia[12], is used to queue analysis in a pipeline on which each benchmark is to be run, server performance measured, and accumulated results be displayed. This is deployed on target hardware and will import a subset of the data determined by a configurable percentage. Then one will be able to use the web-based interface to perform all necessary benchmarking tasks.
The architecture and how each technology communicates is illustrated in Figure 14. The databases are containerized using Docker[13].
JanusGraph has some Java specific features that add limitations when making use of embedded Gremlin in Python. The limitations are, when trying to make use of mixed indexing search predicates such as spatial queries, that one may only do this via Java or a superset language thereof. The workaround for this was to make use of the Gremlin Translator which takes a Gremlin query as a string and interprets it on the server side.
The first motivation towards using a Python back-end is that the text in the reviews can be analysed using NLTK for easily implemented sentiment analysis. The second is that a simple REST API can be easily and quickly designed and deployed using the Flask framework. Angular was subjectively chosen as the front end framework as it allows for fast front end web development. All benchmarking results are stored in a separate database within PostgreSQL.

---

[9] `https://canonical.com/`

[10] Thus makes use of the property graph model.

[11] `https://www.tigergraph.com/solutions/`

[12] The name of the web-application is a nod to JanusGraph and Titan's theme of Roman mythology. Providentia is associated with provision and forethought [41]. This was thought to be fitting due to the nature of our experiments designed to find the best database for storing and modeling our particular data.

[13] `https://www.docker.com/`

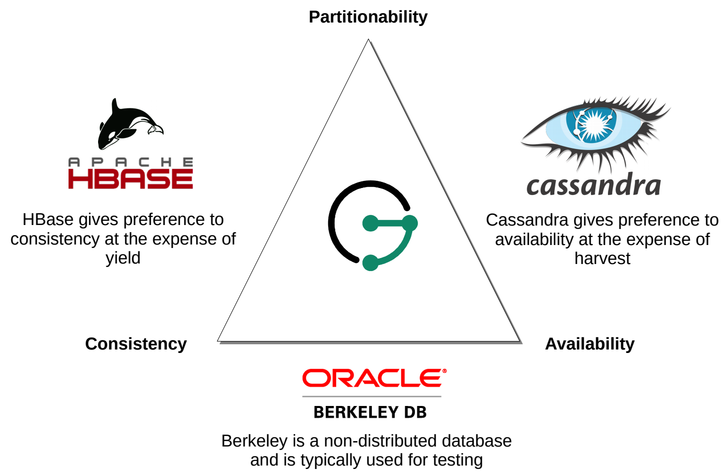

Figure 13: The CAP theorem illustrated using JanusGraph's three supporting storage back-ends. This diagram is largely inspired from Chapter 1 in Titan's documentation and is adjusted for JanusGraph [36].

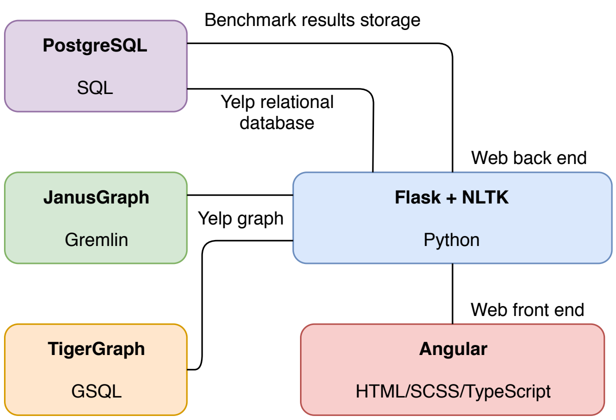

Figure 14: The architecture of Providentia.

The front end of Providentia allows a user to query each database, test the sentiment classifier, add benchmarking jobs and to review the performance and results of each analysis. Each job is run serially to avoid too much interference and competition between each database for resources. At intervals the CPU performance and memory consumption is measured and stored in PostgreSQL. The server performance and results of an analysis can be viewed together to validate that the outputs are the same and how each database utilizes the server's resources.

5.3 Data Analysis

Using each of the databases, a number of data analysis jobs are performed on the data. This section describes what each analysis aims to do and how they align to a typical real world use case. Each of these jobs have some kind of spatiotemporal aspect to test the accessibility of the data to demonstrate how well the given database handles the data. These analysis are run over different percentages of the data loaded in each database and the performance is then measured and discussed in Section 7. This section goes into more detail about the types of queries written and how well each language expresses each query.

5.3.1 Sentiment Analysis

One application for graph analytic platforms is in machine learning. In order to further build context around each of the kernels mentioned in the next section, a simple binary sentiment classifier is used to classify the text of reviews as representing either a positive or negative sentiment. Sentiment analysis is a class of natural language processing where subjective information is extracted from a given text [42].
Although this investigation does not explore the applications of machine learning models on spatiotemporal data, it will explore how it could reveal interesting information alongside patterns among the data. All natural language processing is done using the Natural Language Toolkit (NLTK) [14] for Python.

5.3.1.1 Machine Learning Model

NLTK's Naïve Bayes classifier[15] was used as the model to train and classify sentiment. Naïve Bayes is a probabilistic machine learning model which has proven very effective in text classification. A limitation of this model, on a binary classification problem, includes only being able to perform linear separability. This should not be an issue regarding

[14] `https://www.nltk.org/`
[15] `nltk.NaiveBayesClassifier`

the use of adjectives as feature vectors, since the most informative adjectives seem to be verbose and particular to a specific sentiment e.g. words such as horrible, disgusting, perfect, and wonderful [43].
One problem faced by sentiment analysis is that of negation which may make certain adjectives less information e.g. good vs not good [44]. Nevertheless, the model performs well in terms of separating the two classes.

5.3.2 Kernels

Each of the following kernels represent some kind of user story for data analysis. Each one has some spatio-temporal constraint which will be applied on the respective queries. Each of these kernels will be benchmarked 30 times each such that the mean and standard deviation can be considered when comparing response times.
For each kernel, their respective database queries are available in Appendix A.

5.3.2.1 Medical: Priority 1 Mean Response Times

This analysis looks at the mean of two intervals within the response life cycle namely; travel time to the hospital and the time taken for the ambulance to start driving to the patient. This will filter out all responses which do not take the patient to the hospital and responses without a priority of 1 (highest).
One hospital is isolated during this query. This query is useful in post-simulation analysis when optimizing response times with respect to a specific hospital. This can be extended to be grouped by hospitals if labels are associated with the coordinates.

5.3.2.2 Medical: Second Resource to Transfer Patients

During a response, sometimes multiple resources need to be sent depending on varying evaluation criteria made either by the call receiver or first resource upon arrival at the scene. This analysis considers how many resources, grouped by priority, were sent as the second resource to the same destination hospital from the kernel used above. This analysis can be used to investigate each priority's resource usage and if their priority was perhaps misclassified. It can be further extended to look at third and fourth resources in order to be more thorough with the analysis.

5.3.2.3 Medical: Long Response Count

This analysis is purely temporal and combines the time intervals from when the ambulance leaves up until the point where the patient is delivered to a hospital. The number of responses are returned for all responses which take longer than 15 minutes. This analysis could be useful when considering introducing a new hospital into a given area to improve response times.

5.3.2.4 Yelp: Kate's Restaurant Recommendation

This analysis selects a user near the beginning of the dataset named Kate. This user has a number of reviews for restaurants in the Las Vegas area. A subset of reviews which hold a strictly greater than 3 star rating by Kate are selected, sorted by date descending, and limited by 10 reviews per user in order to take the most relevant ones. These businesses are then selected, filtered by category "Restaurants". The users who have a star rating equal to or greater than Kate's ratings for the same businesses are selected as the recommending users.
Now assume that Kate has relocated to a new area. All businesses which have been rated strictly larger than 3 stars by the recommending users are then selected as restaurants to recommend to Kate in the new area. The text in these reviews are checked for sentiment and the percentage positive sentiment is displayed alongside the average star rating for each recommended restaurant. This sentiment vs. average star rating is used as a metric to analyse in terms of asking the question: How reliable is the star rating versus the actual sentiment found in the text?
The purpose of the first part of this kernel is to test the performance of a 1-hop graph traversal pattern. This hop is demonstrated in category filtering and finding users with mutual sentiment for a given review. This type of situation is faced by many recommendation technologies and this is quite a basic technique for recommendation. The additional challenge is the relocation of Kate and seeing how responsive the database is to the spatial and temporal aspect which is the second part of this kernel. The accumulated list of users is split into a separate query for each user to test the ability of each database technology to perform concurrent reads on subsets of data which is a strength of NoSQL databases.

5.3.2.5 Yelp: Review Trends in Phoenix 2018

This analysis goes deeper into observing the trend of various characteristics of reviews versus their star ratings. This is a common analysis performed on the Yelp dataset [45], but the version in this investigation selects a subset of reviews only within the 2018 year in the Phoenix area. The spatiotemporal boundaries placed on this subset may reveal hidden trends to be considered in future work.
Reviews are extracted first by location (which results in a much smaller subset than extracting by date first) then by date. The reviews are then separated by star rating. For each star rating, the characteristics of "funny", "useful", and "cool" are accumulated and the text is classified as either positive or negative. For each star rating these are normalized and placed next to one another to see the characteristic of a review from each star group. Below is a comparison of the queries written for this kernel.

SQL This kernel is the least complex of the three as it has a single join with a spatio-temporal constraint. Listing 1 returns selected characteristics on reviews where the date year is 2018 and reviewed businesses are within the Phoenix area.

```
SELECT text, review.stars, cool, funny, useful
FROM business
JOIN review ON business.id = review.business_id
    AND ST_DWithin(
        location,
        ST_MakePoint(-112.56, 33.45)::geography,
        50000)
    AND date_part("year", date) = 2018)
```

Listing 1: A SQL query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

Gremlin One caveat of using mixed indexes on dates via the Gremlin Translator is highlighted in the query for this kernel. Usually, since Gremlin is designed to be embedded, one should make use of objects when appropriate. Since JanusGraph is being queried from Python[16], with no support for mixed query specific parameters, date related parameters need to be parsed using static methods from the `Instant` class in Java. This can be seen in Listing 2 when filtering reviews by date. Alternatively, one could also use the `filter` step as is done in Listing 23.
Out of the set of businesses within the Phoenix area and set of all the reviews in 2018, the businesses set would be the smaller of the two. This is important when using a dataflow language since the whole subset will be accumulated before moving to successive functions in the query. Due to this characteristic of Gremlin, businesses are accumulated before the reviews.

```
g.V().has("Business",
    "location", geoWithin(
        Geoshape.circle(33.45,-112.56, 50)))
    .inE("REVIEWS")
    .has("date", between(
      Instant.parse("2018-01-01T00:00:00.00Z"),
      Instant.parse("2018-12-31T23:59:59.99Z")
    )).valueMap()
```

Listing 2: A Gremlin query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

GSQL Since only selected characteristics of a review are desired, a tuple is created at the beginning of Listing 3. The businesses within the Phoenix area are selected first, then reviews where the date part is 2018. The review tuples are accumulated into a `ListAccum`.

```
CREATE QUERY getReviewsFromPhoenix2018()
    FOR GRAPH MyGraph {
    TYPEDEF tuple<STRING text, INT stars,
        INT cool, INT funny,
        INT useful> review;
    DOUBLE lat = 33.45;
    DOUBLE lon = -112.56;
    INT distKm = 50;
    ListAccum<review> @@reviewList;
    Grids = to_vertex_set(
        getNearbyGridId(distKm, lat, lon),
        "Geo_Grid");

    NearbyBusinesses =
      SELECT b
      FROM Grids:s-(Business_Geo:e)-Business:b
      WHERE geoDistance(lat, lon,
          e.LATITUDE, e.LONGITUDE) <= distKm;

    ReviewsForBusinesses =
        SELECT b
        FROM NearbyBusinesses:b
            -(reverse_Reviews:r)-
            User
        WHERE YEAR(r.REVIEW_DATE) == 2018
        ACCUM @@reviewList +=
          review(r.TEXT, r.STARS,
          r.COOL, r.FUNNY, r.USEFUL);

  PRINT @@reviewList;
}
```

Listing 3: A GSQL query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

5.3.2.6 Yelp: Ranking Las Vegas by Friends' Sentiment

The purpose of this analysis is the ability to aggregate relations from depth 1 – 2 of a graph pattern while maintaining spatio-temporal constraints. The user story of this kernel is that a user in the dataset would like to travel to Las Vegas over the Nov – Dec period. Instead of asking from each of the hundreds of direct friends to thousands of mutual friends, the sentiment from their reviews written during the Nov – Dec period (irrespective of year) in the Las Vegas area will be analysed.
Both friends and mutual friends will be aggregated and all reviews written during the Nov – Dec period will be filtered. These reviews will be filtered by the spatial constraint of whether they are connected to businesses within 30km of the Las Vegas center. The remaining reviews will have their text data extracted and returned for analysis. Using the sentiment classifier, a percentage positive sentiment will be generated and this will be the result of the data analysis.

6 Implementation

The following section describes how the systems were setup, data processed, modelled and describes notable indexes used in each technology. The diagrams illustrating each of the databases' schemas can be found under Appendix B.

[16]Where ideally, it would be within a JVM language which has access to the JanusGraph specific classes and functions, e.g. Groovy.

6.1 Benchmark Setup

This subsection describes the hardware used for benchmarking as well as the technical details of the dataset and preprocessing performed.

6.1.1 Hardware Platform

All experiments were run on the following two machines listed in Table 1, that made it possible to consider how the technologies utilize multiple cores and perform with different storage limitations.

6.1.2 Datasets

The medical response dataset contains 7 568 records, which represent one year's worth of simulated ambulance responses. The spatial properties are originally based on a Cartesian coordinate system. Unfortunately, JanusGraph's geo-predicates only support a geographic coordinate system. To support this limitation, a "Flat Earth" projection is used to convert the Cartesian points into latitude and longitude during the data normalization process.

Due to the enormity of the Yelp Challenge dataset [5][17] and hardware limitations, only subsets of the data are used to observe how well each database scales. These subsets were queried incrementally based on the percentage of users and businesses e.g. 10% would mean 10% of `business.json` and 10% of `user.json`. All of the reviews between the remaining users and businesses from `review.json` are applied afterwards. The dataset is stored as non-valid JSON and first had to be preprocessed and converted into valid JSON.

During preprocessing many attributes not used in the analysis or benchmarking were removed to save on import time and storage costs. Only businesses, users, and reviews were used from the dataset. This resulted in a $\pm 11.39\%$ reduction in uncompressed storage size[18], significant reduction in complexity and improvement in consistency among attributes. Attributes such as `average_stars` and `review_count` are removed as they won't make sense when working with subsets of the data. The result of this preprocessing and feature selection can be seen in Table 3.

6.2 Schema Design

The following subsections describe the different indexing techniques used on the spatio-temporal attributes of the data and graphically present the schemas used in each database.

6.2.1 Indexing

When storing a database considered to be large, it will necessitate that the data be stored on secondary storage, since it would most likely not fit in memory. This slows down the data access speeds considerably. When specific records need to be retrieved among large volumes of data, an intelligent method of organizing the data needs to be implemented. We can do this by narrowing our search space to a small subset where our target data lies – these small subsets are identified by our *indexes*.

There are two broad classes [46] of retrievals methods used when gathering data, namely:

- Sequential, e.g. when we retrieve from our reviews all records between June 2018 and November 2018.
- Random, e.g. when we retrieve from our users records containing information about J. Doe.

The way we search for our data using these two classes is guided by our indexes in order to improve the performance of our search. The method of indexing differs depending on the data type being used.

Traditionally, databases only stored primitive data types, but now support various others types such as IP, timestamps, arrays, UUID, and JSON. Spatial data is typically two-dimensional and this cannot be efficiently indexed with an B-tree but, for example, we would use an R-tree. TigerGraph make use of a supposedly more efficient method of querying spatial data that fits graph architecture appropriately, called a geograph, the performance of which will be compared in our experiments. In the following paragraphs the underlying index structures used for our databases will be discussed.

B-trees B-trees can be considered as a generalization of binary search trees [46]. Unlike binary trees, more than two paths may leave a given node depending on the outcome of the query at a node, e.g. at node 0 if $x > 0$ traverse to node 1, $x = 0$ traverse to node 2, $x > 1$ traverse to node 3.

Typical, binary trees may become unbalanced after some number of insert and deletion operations, but B-trees always remain balanced. All leaves in a B-tree have the same depth and any search operation among n records will never visit more than $1 + log_d n$ nodes[19].

B-trees are popular for indexing one-dimensional data and are the default indexing method for many databases, and in our use case, are used for not only primary keys but also temporal indexing. One of the important uses of B-trees is the efficiency gain in sequential and range queries, further optimized by clustering each record in the data by their date fields.

R-Trees The nature of spatial data being two-dimensional reveals a shortcoming in using B-trees as the method of efficiently indexing coordinates. Most successful methods of indexing multi-dimensional data follow a B-tree-like structure [47] and, in a similar fashion, guide the search to a smaller space.

Traditional R-trees implement indexing by guiding the search toward bounded (hyper) rectangles enclosing the multi-dimensional spatial object. This allows us to query over arbitrary regions such as the nearest restaurants within a 5km radius of a given point without doing a full scan.

[17] Totaling in size around 8.69 gigabytes in uncompressed format.

[18] This results in a close to 1 gigabyte reduction, which is a significant improvement.

[19] Where d is the order of the tree.

Table 1: The specifications of the two machines used to benchmark database performance. Setup 1 makes use of AMD Ryzen 5 2600 whereas Setup 2 is an AWS c5.4xlarge instance with Intel Xeon Platinum 8000 series CPUs.

Machine	vCPUs	Base Clock	Memory	Storage Media	OS
Setup 1	12	3.4GHz	32GB	SSD	Debian 10
Setup 2	16	2.5GHz	32GB	SSD (Amazon EBS)	Ubuntu 18.04

Table 2: Data used from the medical response dataset after preprocessing.

Medical Responses	
Attribute	Data Type
ID	integer
x	float
y	float
prio	integer
time_to_ambulance_starts	float
on_scene_duration	float
transfer	bit
time_at_hospital	float
travel_time_patient	float
travel_time_hospital	float
travel_time_station	float
resource	integer
x_dest	float
y_dest	float
resource_ready_time	float

Disadvantages of R-trees include that they are slow to update and that they create a significant redundancy in terms of data storage [48].
ElasticSearch (the search engine used for our JanusGraph configuration) and PostGIS are two examples of technologies that implement R-trees in the database technologies being benchmarked this paper.

Geograph The geograph is a grid-based "indexing"[20] solution used by TigerGraph which naturally fits the graph architecture and saves on data storage costs. The idea is that two-dimensional coordinates are mapped to a given grid ID where a grid is represented as a graph vertex. Any vertex associated at that point is then linked by an edge.
A grid can be of any size but setting this size may be dependent on the distribution of points in the dataset. This allows queries to leverage the massively parallel processing (MPP) techniques implored by TigerGraph which create fast updates in contrast to R-trees. The mapping from coordinates to grid ID works in a way such that one can still do searches over an arbitrary region without scanning the whole graph.
A disadvantage of this approach is an uneven distribution of vertices linked to each grid, but this can be managed by manually configuring grid sizes.

[20]Mentioned in quotations due to TigerGraph not implementing indexes, but rather optimizing data access by the notion of installing queries.

6.2.2 Transcribing from Relational to Graph

One advantage with graph databases is that the schema can be simpler than its relational counterpart. This holds true for relational schemas which have one or more many-to-many relations or joining tables, but not when there are only one-to-many type of relations.
On the simplest level, a table can be transcribed to a vertex and the relations can become the edges with labels discerning between two vertices or two edges. To extend this, if a table joins exactly two other tables by foreign key relations, this joining table can become an edge as is the case with the "reviews" table in the Yelp dataset. The attributes are then stored as key-value pairs on their respective graph component.
This is important to note as the medical response dataset holds no joining tables, but the Yelp dataset does have one.

6.2.3 Relational Design

Figure 25 and 28 show the designs of the medical response and Yelp dataset modelled in a relational database, specifically PostgreSQL. The *location*, *origin*, and *destination* attributes are indexed with R-trees using the PostGIS[21] extension.
Regarding the medical response dataset, one major feature is that responses may result in either the patient being transported or treated on site. Due to this, two one-to-zero-or-one relations are created with the `transfer` and `on_scene` tables and the presence of a relation to either

[21]`https://postgis.net/`

Table 3: Data used from the Yelp dataset after preprocessing.

Business		User		Review	
Attribute	Data Type	Attribute	Data Type	Attribute	Data Type
business_id	string	user_id	string	review_id	string
name	string	name	string	user_id	string
address	string	friends	string array	business_id	string
city	string	yelping_since	timestamp	stars	integer
state	string	useful	integer	date	timestamp
postal code	string	funny	integer	text	string
latitude	float	cool	integer	useful	integer
longitude	float	fans	integer	funny	integer
stars	float			cool	integer
is_open	integer				
categories	string array				

will deduce whether the response resulted in a transport or not. The priority of a response and the resource number of a response are low cardinality attributes which are then separated into tables with a one-to-many relation.
For the Yelp dataset, the decision not to extend *city* and *state* attributes into separate tables was taken, otherwise more joins would be required for an attribute that is never queried. *Location* is the only purely spatial attribute – the main attribute in the benchmarking. It may be faster to simply index state as an attribute due to the low cardinality of city and state paired in a single table. City is indexed and clustered such that records in the same city are physically stored together which, alongside location, should help retrieval speeds from a spatial query perspective. PostgreSQL makes use of B-trees and hash indexes for native data types [49].
The review table is commonly used in queries and holds the most interesting attribute in terms of temporal insight. Since temporal information is one-dimensional and ordinal, the *date* attribute is indexed and clustered.
A business category and a user's friends are many-to-many relationships. The linking tables `bus_2_cat` and `friends` tables handle these relationships.

6.2.4 Graph Design

The JanusGraph designs for the two datasets can be seen in Figure 26 and 29 , whereas that of TigerGraph is given in Figure 27 and 30 . The motivation for the difference in the two designs is mainly due to the databases handling spatial data differently.
JanusGraph indexes attributes on nodes and edges using either composite indexes [50], which index native data types on equality conditions, or mixed indexes which leverages an indexing back-end on more complex data types or for complex search predicates, e.g. fuzzy search on strings [51].
Figure 26 and 29 show which attributes are indexed using composite indexes and which use mixed indexes – making use of the indexing back-end. As in Section 6.2.3, *location*, *origin*, and *destination* are indexed using R-trees with ElasticSearch's geo-search predicates. *Date* is indexed using ElasticSearch for equality conditions using the `java.time.Instant` class – this uses a Bkd-tree indexing implementation [52]. Select temporal attributes in the medical response dataset use composite indexes as they are represented as floating point values and not as dates . TigerGraph puts less of a focus on indexing and more on writing efficient and fast queries. One notable difference between the structure in the JanusGraph implementation and the TigerGraph implementation (see Figure 30) is that there are no indexes and the extension of the spatial attributes as the `_Geo`-suffixed edges and `Geo_Grid` vertices are present in TigerGraph . The code leveraged for this design idea was inspired from the TigerGraph geospatial webinar [48] and C++ code on the TigerGraph "ecosys"[22].

7 Results and Discussion

This section discusses how each database performed on implementing the kernels mentioned in Section 5.3.2 and discusses the effectiveness of each query language when producing a query to extract the required data for each kernel.
The results of the data analysis can be viewed in Appendix C.

7.1 Priority 1 Mean Response Times

JanusGraph can be seen to be far outperformed by the other two database technologies in terms of response times in Figure 15. This is a trend that features in all three queries regarding the medical response dataset. Another trend is that TigerGraph and PostgreSQL perform closely where TigerGraph responds the fastest.
One observation is that it takes around 5 minutes for an ambulance to start from the time a call is made for a priority 1 emergency. This might be considered long for a life threatening situation but the personnel handling the call needs to carefully diagnose the situation before considering it priority 1 - so this is not unusual. There is more room to optimize the transfer times to hospitals by potentially adding another hospital to the area.

[22] `https://github.com/tigergraph/ecosys/tree/master/guru_scripts/geospatial_search`

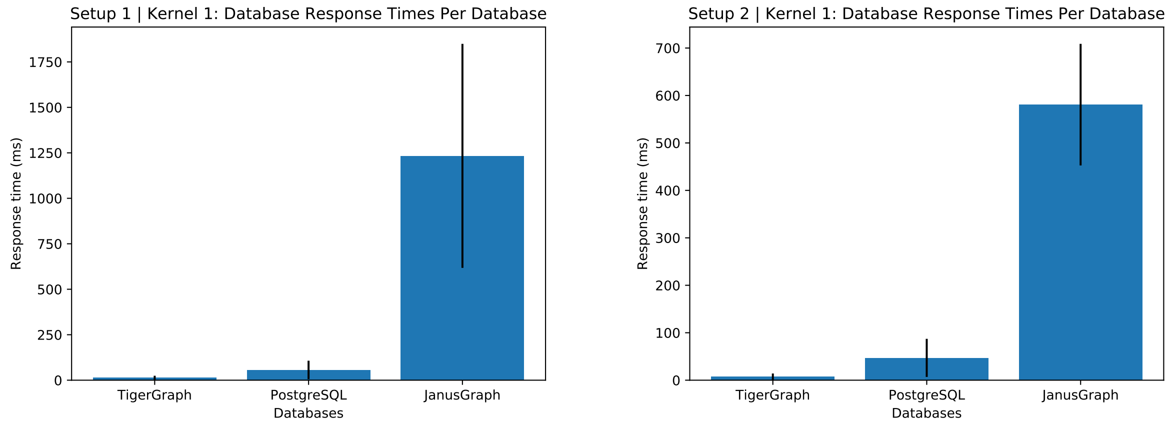

Figure 15: Database response times using the medical response dataset for setup 1 and 2 for the kernel: "Priority 1 Mean Response Times". The error bars display standard deviation.

7.2 Second Resource to Transfer Patients

Each database appears to perform much more consistently (see Figure 16) with lower deviation as apposed to its performance in Figure 15. JanusGraph appears to have some kind of consistent overhead between the three containers (JanusGraph, Cassandra, and ElasticSearch) which appears to effect the response time.
Regarding the result of this query (see Table 5), it appears that more resources are sent to priority 2 emergencies. This simulation does not take into account emergencies that are recategorised on site which is something that happens outside of simulations. This could suggest that priority 2 responses are initially miscategorised and many should be priority 1 or that priority 2 responses simply require more resources in general.

7.3 Long Response Count

This is only a temporal query and one can see that TigerGraph and PostgreSQL perform much more similarly than in the previous two queries. This suggests that added complexity from constraints in a query perform better in TigerGraph than in PostgreSQL.
Although 15 minutes is considered quite fast for the whole response cycle to complete - which is why there are 5284 responses returned. This kind of query could be used to inform the medical infrastructure of a country or state how resource location affects response cycles.

7.4 Kate's Restaurant Recommendation

Figures 18 shows linear growth in the response time of PostgreSQL whereas JanusGraph and TigerGraph remain fairly horizontal over increasing volumes of data. Due to the complexity of this query it comes as no surprise that the graph databases far outperform their relational counterpart. TigerGraph outperforms both PostgreSQL and JanusGraph in terms of query response time and shows a very high consistency as there is almost no standard deviation around the mean.
Table 7 shows which restaurants would be recommended to Kate. One can see that both positive sentiment and star average over reviews for highly regarded restaurants compare well and remain consistent.
The reviews returned by the query were typically well above 3 stars and, since the Naïve Bayes performed well when predicting unseen text data, it comes as no surprise that most of the reviews were tagged as positive. Further analysis could look at how positive sentiment and star rating would compare for inconsistently performing restaurants, using a heuristic such as variance of mean star rating over time to indicate the consistency of restaurant performance.

7.5 Review Trends in Phoenix 2018

Figure 19 shows a phenomena of interest where JanusGraph performs poorly in relation to the other two database technologies with high deviation around the mean. PostgreSQL scales horizontally for this kernel which is most likely be due to the query being the simplest of the three kernels. TigerGraph outperforms both.
The result of a subgraph produced by this query can be seen in Figure 20.
The general trends shown in the review data for Phoenix during 2018 have the following characteristics:

- More critical, lower scoring reviews tend to be longer and most useful.
- Reviews with 3 or 4 stars seem to be the funniest.
- Reviews with 4 or 5 stars tend to be the coolest.

The percentage positive sentiment, when scored relatively, is ordered consistently with the average star rating. This validates the performance of the binary sentiment classifier in that one almost does not need to see the star rating and can rely on text data alone when considering a broad spectrum of reviews.
This analysis could be performed over varied year brackets and different areas to see if performance is consistent or not.

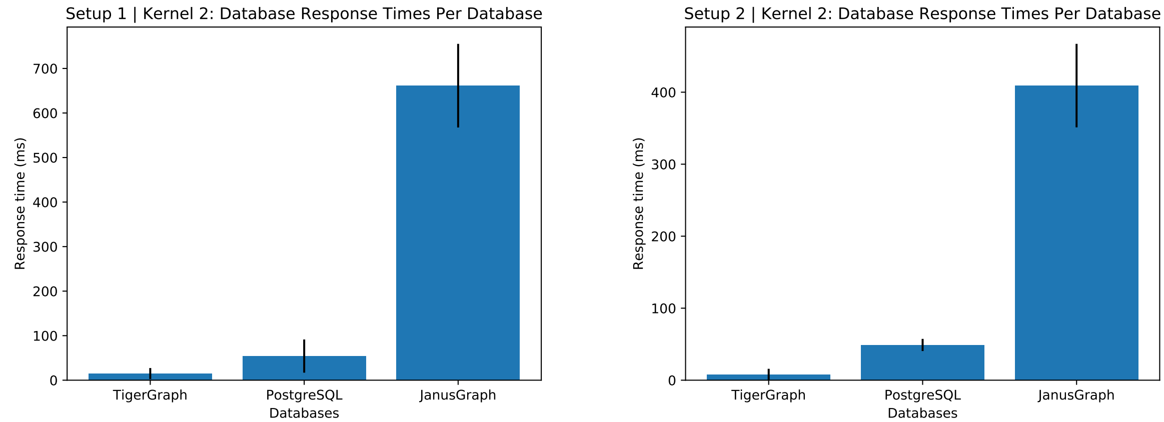

Figure 16: Database response times using the medical response dataset for setup 1 and 2 for the kernel: "Second Resource to Transfer Patients". The error bars display standard deviation.

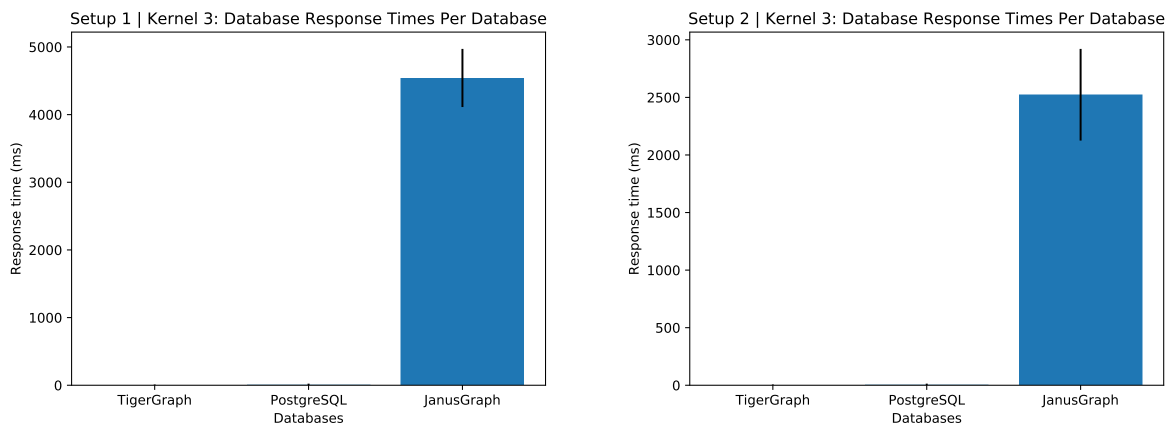

Figure 17: Database response times using the medical response dataset for setup 1 and 2 for the kernel: "Long Response Count". The error bars display standard deviation.

The implication of this could lead to experimenting with more sophisticated machine learning models on the dataset to be more precise in that it could potentially predict the star rating as is done in [53] and [54].

7.6 Ranking Las Vegas by Friends' Sentiment

Figure 21 shows both graph database technologies outperform PostgreSQL as PostgreSQL shows linearly growth as it did in Figures 18. For the experiments on 13% of the dataset, JanusGraph shows a lot of deviation around the mean. This may be due to all the moving parts on which JanusGraph is implemented on and it's multi-level caching implementation behaving poorly at this size of the dataset. The result of this analysis was focused more on the performance rather than the data extracted. The results of the data analysis on this kernel does not show anything more interesting about the review data than what was already discussed in Section 7.5. One notable difference in this kernel however, is that it produces a much more complex query and the databases perform accordingly.

This kernel shows that results may vary and can be correlated depending on the relationships between different data points.

7.7 Memory Consumption

The memory consumption of each database (while idle) is illustrated in Figure 24 and clearly shows the high cost of graph database technology with TigerGraph and JanusGraph using about 2GiB and 10GiB respectively. PostgreSQL only makes use of around 9MiB which makes is extremely lighter than the graph databases.

The technologies used for JanusGraph all run on the JVM and, given the 32GB memory available, allowed the JVM's garbage collector to run infrequently. Most of the memory used by JanusGraph is in the cache for ElasticSearch, Cassandra, and JanusGraph, each storing an independent cache. Most of the fluctuations in memory came from the

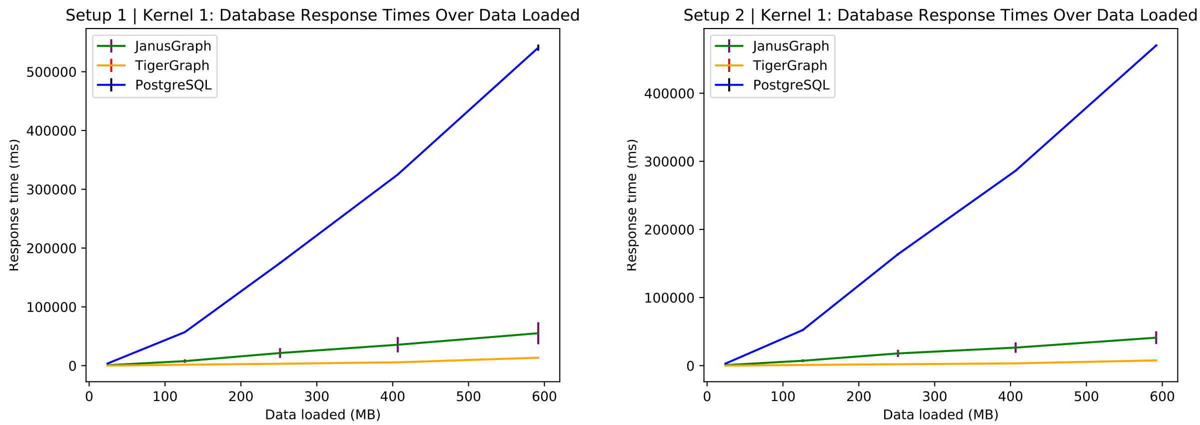

Figure 18: Database response times over varying percentages of the Yelp dataset for setup 1 and 2 for the kernel: "Kate's Restaurant Recommendation". The error bars display standard deviation.

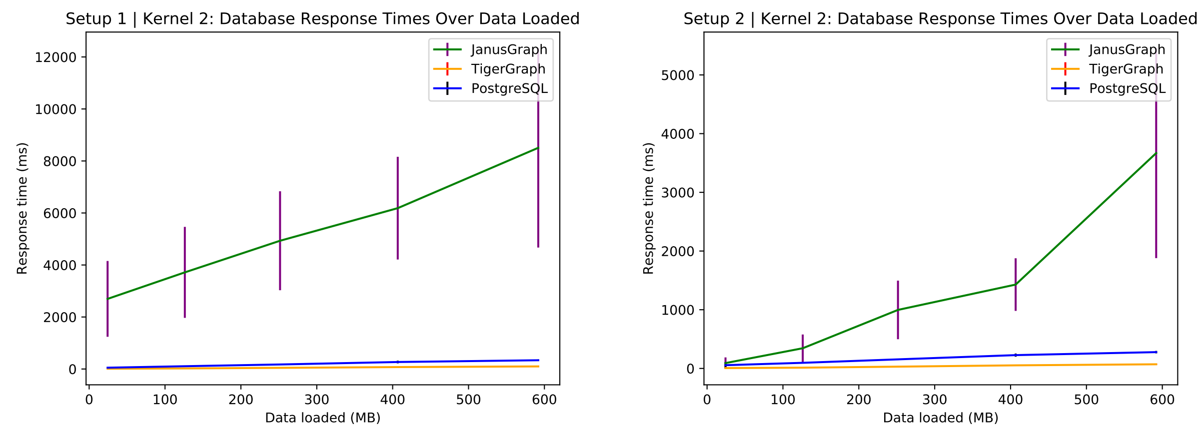

Figure 19: Database response times over varying percentages of the Yelp dataset for setup 1 and 2 for the kernel: "Review Trends in Phoenix 2018". The error bars display standard deviation.

JanusGraph container and this happens from the moment the container is run. This cache can be limited by configurations but these results are from an unlimited cache setup as this is how the benchmarks were run.
TigerGraph and PostgreSQL held constant (as their lower and upper memory bounds were within MiB differences) as the majority of those two databases are written in C or C++ which requires manual memory management. It is important to note that while JanusGraph uses a lot of memory, there are other graph database variants adhering to the Apache TinkerPop standard which claim to be more efficient in terms of memory such as Neo4j and OrientDB. Neo4j and OrientDB memory consumption is also configurable but may still use multiple GiB of memory because of the JVM [55][56].

7.8 Queries

SQL SQL is a mature and well supported querying language which makes it simple to implement a solution. The caveat to this simplicity is that the resulting solution may in complex cases lead to long and convoluted queries – such as the one produced in Listing 13. The SQL queries produced for these kernels have a good balance between readability and expressiveness but, as complexity grew, so did size and queries began to lose the readability aspect. SQL handles temporal data well and, in the PostgreSQL dialect, comes well supported with functions to operate on various temporal data types. This level of support provides ease of programming when implementing a SQL-based solution to a dataset with spatio-temporal properties.

Gremlin Gremlin was found to produce the most concise queries of the three languages. The limitation of Gremlin is that, if one makes use of the mixed indexing search predicates, one may be limited to programming languages with support from these drivers to have the embedded Gremlin functionality. In the context of the technologies implemented in this investigation, a JVM language would

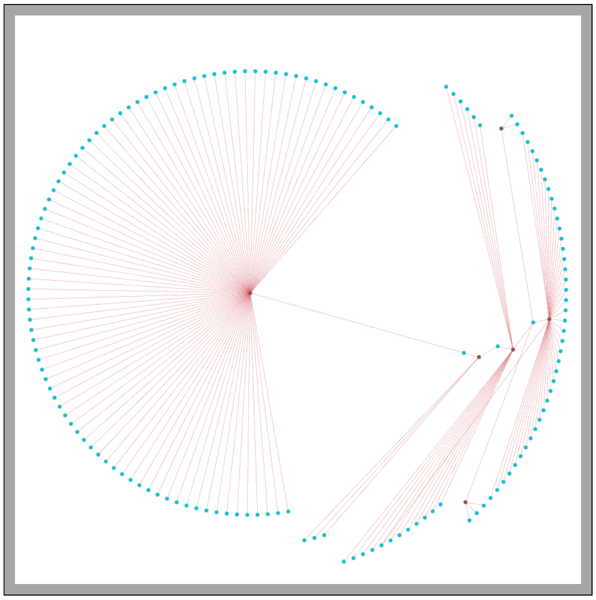

Figure 20: A subset of the graph produced by TigerGraph on the result of the query for all reviews in the Phoenix area in 2018. Maroon edges represent reviews. Blue vertices are the users and brown vertices represent the businesses.

be better suited as the back-end for a JanusGraph data storage solution. The Gremlin queries produced for these kernels were found to be readable in terms of describing the data flow of the traversal within a graph topology context. One may not enjoy Gremlin's referencing steps going back and forth within a query using the `as` and `select` steps but, after some experience with Gremlin, this will no longer be an issue.

The ability of certain steps allowing one to skip across edges to refer to vertices directly is part of why Gremlin is able to produce such concise queries. The performance of these queries heavily relies on the data flow produced by the ordering of such steps. This makes it important to use `filter` steps and be conscious of the ordering of each step.

Gremlin is well supported and has an extensive documentation[23] but is a vastly different querying language when compared to SQL. The implication of this is that there is a small but significant learning curve involved. Effective imperative Gremlin queries will most likely only be written after some experience. Fortunately, Gremlin supports declarative querying which allows a user new to Gremlin to write effective queries with little experience.

GSQL The GSQL queries produced by each kernel resulted in the queries with the most vertical space of the three query languages. This is necessary for segmentation of the query which is used for parallel graph traversals. The result of this segmentation and vertical space has made each query extremely readable and expressive. The conservative use of ASCII art and use of keywords from SQL provides a good balance between query visualization and familiarity. The development of queries using GraphStudio reduces the learning curve significantly as queries are developed in statically typed, compiler driven context – only allowing one to install a query once all errors have been addressed.

GSQL is well suited for spatial queries using the geo-grid approach – which integrates well within a graph topology – and temporal queries with a selection of built-in function for manipulating temporal data types – as with SQL. By segmenting the query, the compiler is able to determine what can automatically be executed in parallel which adds to the fast response times of TigerGraph when compared to the other two databases.

The Rest++ API allows one to write a parameterized query once and access it anywhere without having to worry about driver issues other than being able to communicate with a REST API. This was a particular pleasure in the post-query development of connecting a web application back-end to communicate with TigerGraph.

8 Conclusion and Future Work

In this paper, we analyzed and compared the response times of three database technologies with respect to handling interconnected spatio-temporal data. The technologies compared were two open source database technologies, PostgreSQL and JanusGraph, and one enterprise level technology, TigerGraph. The linear growth in the relational model was clearly illustrated in the results whereas the graph database solutions scaled more horizontally. This alone is an advantage NoSQL databases have over traditional relational models when querying large volumes of data. These three systems were evaluated by employing a set of spatio-temporal queries similar to those that would be found in real world scenarios when analysing data in a dataset such as the Yelp Challenge Dataset.

The results show that graph database technology has been shown to outperform PostgreSQL in all three of the kernels. This result is partially due to the fact that the kernels produce complex queries due to the interconnected nature of the data. This dataset produced a dense graph which graph databases have the ability to perform effective traversals over when compared to multi-join style queries produced by the relational implementation. The spatio-temporal multi-dimensional aspect has shown to be supported well in all of the databases as evident by the response times of the queries with constraints of this nature.

Benchmark Results The results in Section 7 strongly suggest that graph database technology and, specifically TigerGraph, provide the faster response times relative to PostgreSQL when querying with complex queries on larger percentages of the dataset.

The spatio-temporal constraints seem to play a minor role in influencing the mean response times when compared to the influence the size of the dataset and complexity of

[23] `http://tinkerpop.apache.org/docs/current/reference/`

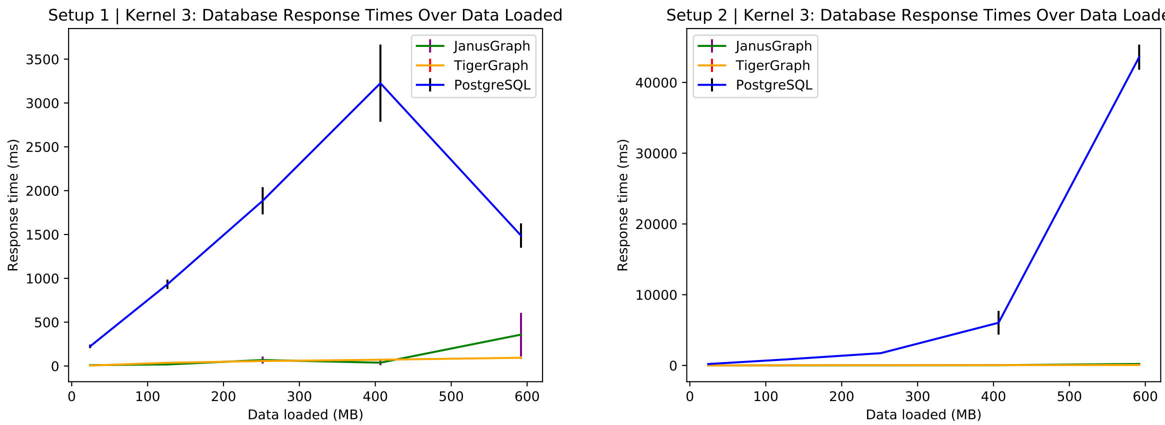

Figure 21: Database response times over varying percentages of the dataset for setup 1 and 2 for the kernel: "Ranking Las Vegas by Friends' Sentiment". The error bars display standard deviation.

queries have . One notable observation is how inconsistent JanusGraph performed and this may be due to JanusGraph's caching implementation. JanusGraph maintains multiple levels of caching both on the transaction level and database level. This excludes the storage back-end's caching – in this case Cassandra. The cache has an expiration time and, since these experiments were run serially but chosen randomly, the JanusGraph specific jobs were run out of order and the cache could have expired at this time.
Another potential reason for the inconsistent gradient between the means in each result may be due to the fact that the dataset adds unpredictable levels of complexity – in terms of how connected the data is – at the end of an import for a given percentage. The horizontal scaling for the graph databases suggests that the impact of this is minimal. Nevertheless, as the queries became more and more complicated, the graph databases maintained a horizontal scale whereas PostgreSQL grew linearly in these cases. When the queries were not complicated, as was the case in Section 7.5, or the dataset was small, as was the case with the medical response dataset, PostgreSQL responded nearly as fast as TigerGraph and outperforming JanusGraph.

Visualization Graph databases have an advantage in data visualization with tools such as Cytoscape[24] or GraphStudio which is built into TigerGraph. This can be important in use cases involving further analysis on data patterns such as those found on social datasets such as the Yelp dataset. Relational databases can store data processed and re-shaped into a graph structure, but this requires extra overhead and configuration as it is not a native topology of this technology.

Migration and Product Maturity Graph database technology is currently under rapid development, with each vendor having their own API and query language. Each graph query language is designed to express graph traversals in a more graph-oriented approach. The learning curve when migrating from SQL may pose an issue, but languages such as Cypher or GSQL make this minimal by applying concepts from SQL in the graph context.
Security and reliability used to be an issue when considering migrating to graph database technology, but both of these products can be configured to use encrypted communication and can be robust to failures. For example, TigerGraph's Rest++ API can be encrypted, integrated with Single Sign-On, and require authorization with LDAP authentication. JanusGraph transactions can be configured to be ACID-compliant when using BerkeleyDB but this is not generally the case with Cassandra or HBase. TigerGraph and Neo4j are ACID-compliant so it is in the position to compete with relational databases with regards to reliable transactions. The implication of this is that graph database technology can provide the same level of robustness and security as relational database technology can, so one should not have to sacrifice on either of these when migrating.

Data Structure Before pre-processing, the Yelp dataset contains additional attributes, many of which are missing or partial and this makes the dataset semi-structured. Since a relational database schema is fixed, this would increase the number of tables in a relational database for each potential attribute that could have a relationship to any other data entry. Graph databases are schemaless and are well-suited to handle such unstructured or semi-structured data.

Query Languages Of the three graph querying languages, GSQL was found to be the easiest to learn and implement and, with an imperative and statically typed language, many developers may find GSQL very familiar. The Rest++ API feature was found to greatly enhance the post-query process due to any other applications only hav-

[24] `https://cytoscape.org/`

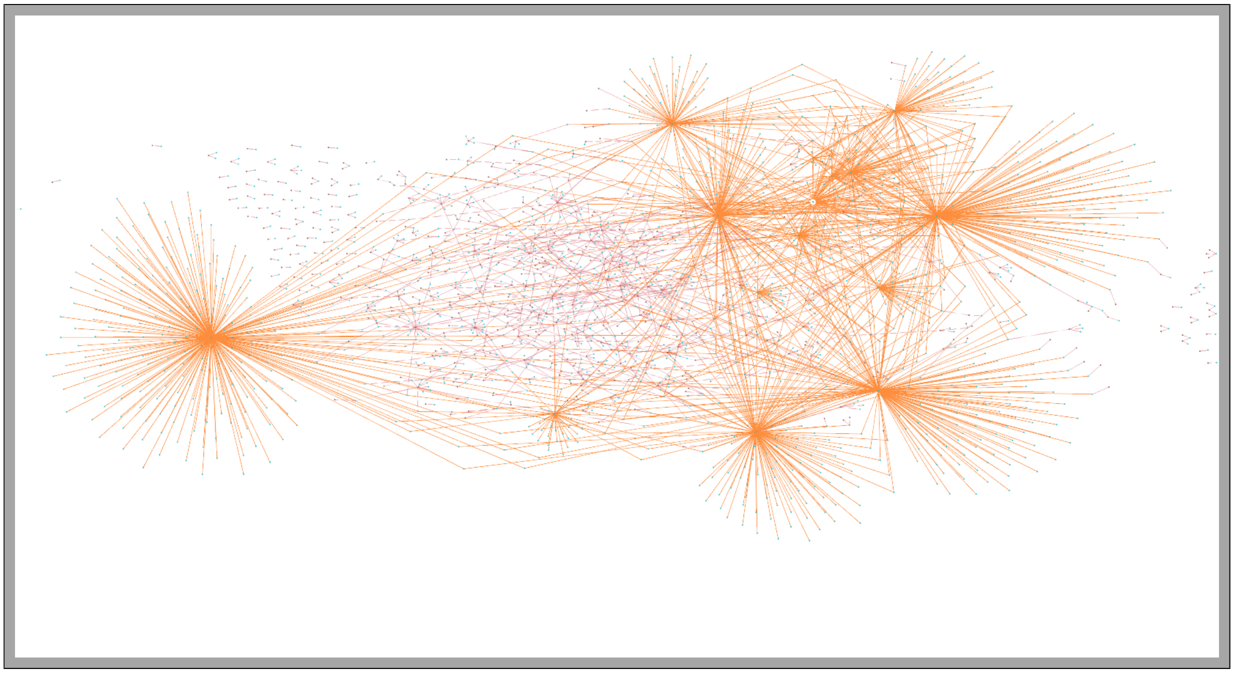

Figure 22: A subset of the graph produced by TigerGraph on the result of the query for the third kernel, *Ranking Las Vegas by Friends' Sentiment*. Orange edges represent friend relations and maroon edges represent reviews. Blue vertices represent users and brown vertices represent businesses. The white center of the cluster on the top right is Julie and once can see the center of the giant cluster on the left is a mutual friend of Julie's. The topology of this graph suggests that Julie has direct and mutual friend circles whose influence extends well beyond her own.

ing to query a parameterized HTTP endpoint. GSQL also adds a lot of flexibility in terms of how the data is formatted and structured in the HTTP response which helps for seamless deserializing of the JSON result.

Future Work This paper investigated the response times and, to some degree, how effective each query language was in producing queries to return the necessary data. Neither the storage efficiency nor performance capabilities for varied limitations on hardware were measured in any sophisticated way. Investigating these issues for large-scale spatio-temporal data would only add to the findings in this paper in terms of how suitable graph database technology is.

Only one dataset was used in this investigation and thus benchmarking querying over other spatio-temporal datasets of varying quantities should also be considered. Doing so would create a more robust conclusion to the suitability of graph database technology for storing and querying spatio-temporal data.

It would be worthwhile to add not only more graph database technologies, but other NoSQL, SQL, and NewSQL solutions. One could, for example, investigate the performance of Cassandra with ElasticSearch and measure whether the graph abstraction provided by JanusGraph is truly beneficial or not. This would add to a more complete view on how well suited each database solution is for spatio-temporal data, or large-scale data in general. There are new relational database technologies which employ auto-sharding and other techniques to help traditional SQL technologies scale horizontally such as MySQL Cluster[25] and the Citus[26] extension for PostgreSQL. Using these technologies could help relational database technologies compete with NoSQL technologies when facing large-scale data in general.

[25] `https://www.mysql.com/products/cluster/`
[26] `https://www.citusdata.com/product`

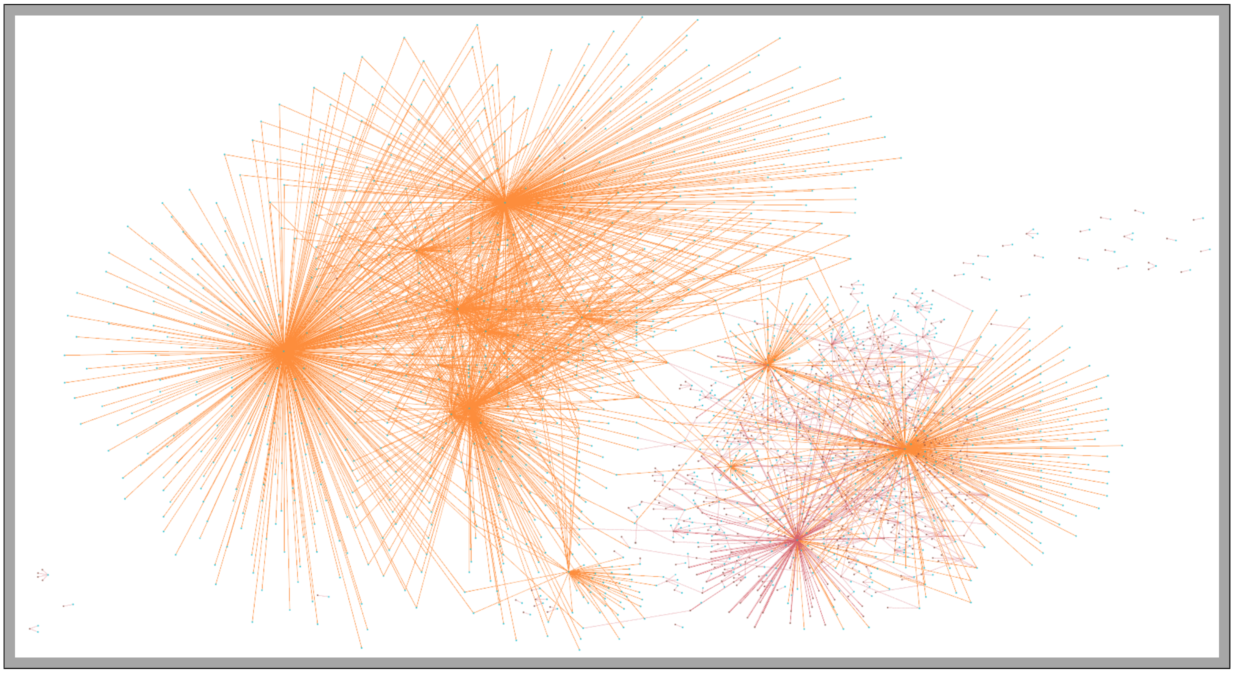

Figure 23: A subset of the graph produced by TigerGraph on the result of the query for the third kernel, *Ranking Las Vegas by Friends' Sentiment* but on the user Kate from the first kernel. The topology of Kate's friend groups suggest a very cliquey circle of friends with influence not far from her own.

References

[1] Antonios Makris, Konstantinos Tserpes, Giannis Spiliopoulos, and Dimosthenis Anagnostopoulos. Performance evaluation of mongodb and postgresql for spatio-temporal data. In *EDBT/ICDT Workshops*, 2019.

[2] Luke Sloan and Jeffrey Morgan. Who tweets with their location? understanding the relationship between demographic characteristics and the use of geoservices and geotagging on twitter. *PloS one*, 10(11):e0142209, 2015.

[3] Yaowen Chen et al. *Comparison of Graph Databases and Relational Databases When Handling Large-Scale Social Data*. PhD thesis, University of Saskatchewan, 2016.

[4] Jaroslav Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

[5] Yelp Inc. Yelp Dataset Challenge. `https://www.yelp.com/dataset/challenge/`, Jan-Dec 2019.

[6] Armin Ronacher and contributors. Flask. `https://palletsprojects.com/p/flask/`, 2015.

[7] Google and contributors. Angular. `https://angular.io/`, 2010.

[8] Ian. What does ACID mean in Database Systems? `https://database.guide/`, June 2016.

[9] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[10] ABM Moniruzzaman and Syed Akhter Hossain. Nosql database: New era of databases for big data analytics-classification, characteristics and comparison. *arXiv preprint arXiv:1307.0191*, 2013.

[11] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. Tigergraph: A native mpp graph database. *arXiv preprint arXiv:1901.08248*, 2019.

[12] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[13] Deka Ganesh Chandra. Base analysis of nosql database. *Future Generation Computer Systems*, 52:13–21, 2015.

[14] Luca Castellano. Distributed, transactional key-value store, May 19 2015. US Patent 9,037,556.

[15] V Manoj. Comparative study of nosql document, column store databases and evaluation of cassandra. *International Journal of Database Management Systems*, 6(4):11, 2014.

[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. {TAO}: Facebook's distributed data store for the social graph. In *Presented as part of*

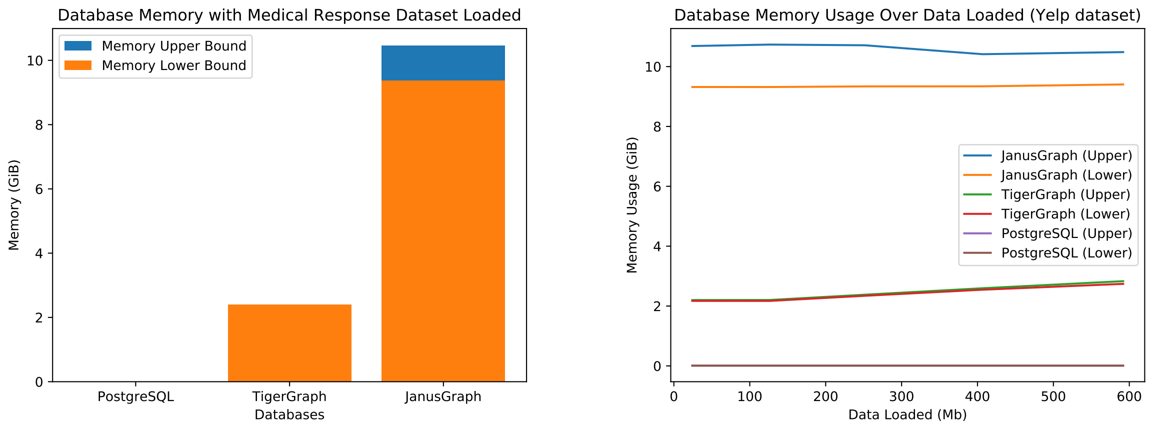

Figure 24: Database memory usage for each dataset. The medical response dataset is 1.5Mb large.

the 2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13), pages 49–60, 2013.

[17] Haoyu Tan, Wuman Luo, and Lionel M Ni. Clost: a hadoop-based storage system for big spatio-temporal data analytics. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 2139–2143. ACM, 2012.

[18] Jaymie R Meliker, Melissa J Slotnick, Gillian A AvRuskin, Andrew Kaufmann, Geoffrey M Jacquez, and Jerome O Nriagu. Improving exposure assessment in environmental epidemiology: Application of spatio-temporal visualization tools. *Journal of Geographical Systems*, 7(1):49–66, 2005.

[19] A. Govardhan K. Venkateswara Rao and K.V. Chalapati Rao. Spatiotemporal data mining: Issues, tasks and applications. *International Journal of Computer Science and Engineering Survey*, 3(1):39, 2012.

[20] The GraphQL Foundation. Graphql. `https://graphql.org/`, 2019.

[21] TigerGraph. Tigergraph docs : Tigergraph v1.0. `https://doc-archive.tigergraph.com/1.0/TigerGraph.html`, 2017.

[22] Apache. The Gremlin Graph Traversal Machine and Language. `https://tinkerpop.apache.org/gremlin.html`, 2019.

[23] Apache. TinkerPop3 Documentation. `http://tinkerpop.apache.org/docs/3.3.0/reference/`, 2019.

[24] Neo4j Inc. openCypher. `https://www.opencypher.org`, 2018.

[25] Yu Xu. Modern Graph Query Language – GSQL. `https://www.kdnuggets.com/`, 2019.

[26] Inc. Neo4j. The pagerank algorithm. `https://neo4j.com/docs/graph-algorithms/current/algorithms/page-rank/`, 2019.

[27] George Anadiotis. Back to the future: Does graph database success hang on query language? `https://www.zdnet.com/`, 2018.

[28] *Querying Graph Databases with the GSQL Query*, 2018.

[29] Martin Heller. TigerGraph review: A graph database designed for deep analytics. `https://www.infoworld.com/`, 2018.

[30] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L Reutter, and Domagoj Vrgoc. Foundations of modern graph query languages. *CoRR, abs/1610.06264*, 2016.

[31] Riko Jacob Christopher L. Barrett and Madhav V. Marathe. Formal-language-constrainted path problems. *SAIM J. Comput.*, 30(3):809–837, 2000.

[32] The PostgreSQL Global Development Group. PostgreSQL About. `https://www.postgresql.org/about/`, 2019.

[33] JanusGraph Authors. JanusGraph. `https://janusgraph.org/`, 2019.

[34] JanusGraph Authors. Migrating from Titan. `https://docs.janusgraph.org/advanced-topics/migrating/`, 2019.

[35] JanusGraph Authors. Introduction: The Benefits of JanusGraph. `https://docs.janusgraph.org/`, 2019.

[36] Titan Authors. Chapter 1. The Benefits of Titan. `http://s3.thinkaurelius.com/docs/titan/1.0.0/benefits.html#_titan_and_the_cap_theorem`, 2015.

[37] Oleksandr Porunov. JanusGraph GitHub. `https://github.com/JanusGraph/janusgraph/blob/master/README.md#powered-by-janusgraph`, 2019.

[38] Florin Rusu & Zhiyi Huang. Benchmarking Graph Analytic Systems: TigerGraph, Neo4j, Neptune,

JanusGraph, and ArangoDB. arXiv:1907.07405, 2019.

[39] Trip Beernink. Private communication. 2019.

[40] Lila Razzaqui. TigerGraph Wins Strata Data's "Most Disruptive Startup" Award. https://www.globenewswire.com/, 2018.

[41] MCMXCV–MMXIX Encyclopedia Mythica. Providentia. https://pantheon.org/articles/p/providentia.html, 1997.

[42] Shashank Gupta. Sentiment Analysis: Concept, Analysis and Applications. https://towardsdatascience.com/, 2018.

[43] Irina Rish et al. An empirical study of the naive bayes classifier. *IJCAI 2001 workshop on empirical methods in artificial intelligence*, 3(22):41–46, 2001.

[44] Eduardo Blanco and Dan Moldovan. Some issues on detecting negation from text. In *Twenty-Fourth International FLAIRS Conference*, 2011.

[45] Zhiwei Zhang. Machine Learning and Visualization with Yelp Dataset. https://medium.com/, 2017.

[46] Douglas Comer. Ubiquitous b-tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[47] DM Gavrila. *R-tree index optimization*. University of Maryland, Center for Automation Research, Computer Vision ..., 1994.

[48] Xinyu Chang. Graph Gurus Episode 8: Location, Location, Location. https://www.youtube.com/watch?v=gPF_SXibDxw&t=887s, 2019.

[49] Jason Harris. PostgreSQL Vs. MySQL: Differences In Performance, Syntax, And Features. https://blog.panoply.io/postgresql-vs.-mysql, 2018.

[50] JanusGraph. Indexing for Better Performance. https://docs.janusgraph.org/v0.2/basics/index-performance/, 2019.

[51] JanusGraph. Index Backend. https://docs.janusgraph.org/v0.2/index-backend/, 2019.

[52] Nick Knize. Numeric and Date Ranges in Elasticsearch: Just Another Brick in the Wall. https://www.elastic.co/, 1 2017.

[53] Ch Sarath Chandra Reddy, K Uday Kumar, J Dheeraj Keshav, Bakshi Rohit Prasad, and Sonali Agarwal. Prediction of star ratings from online reviews. In *TENCON 2017-2017 IEEE Region 10 Conference*, pages 1857–1861. IEEE, 2017.

[54] Dagmar Monett and Hermann Stolte. Predicting star ratings based on annotated reviews of mobile apps. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 421–428. IEEE, 2016.

[55] Neo4j Inc. Understanding memory consumption. https://neo4j.com/developer/kb/understanding-memory-consumption/, 2020.

[56] A Lomakin. How to calculate maximum amount of memory consumed by orientdb. https://orientdb.com/database/memory-consumed-by-orientdb/, 4 2016.

Appendices

A Queries

A.1 Priority 1 Mean Response Times

A.1.1 SQL

```
SELECT  avg(RES.time_to_ambulance_starts)
            as avg_ttas,
        avg(TNS.travel_time_hospital)
          as avg_tth
FROM priority PRI
JOIN response RES ON RES.id = PRI.response_id
JOIN transfer TNS ON RES.id = TNS.response_id
WHERE PRI.id = 1
  AND ST_DWithin(
      RES.destination,
      ST_MakePoint(19.11, 63.67)::geography,
      500);
```

Listing 4: A SQL query that returns the mean of the time it takes for ambulances to start and travel time to the hospital of priority 1 calls for a given hospital.

A.1.2 Groovy

```
g.V().hasLabel("Priority")
    .has("priority_id", "1")
  .in("RESPONSE_PRIORITY").hasLabel("Response")
  .has("destination", geoWithin(
      Geoshape.circle(63.67, 19.11, 0.5))
  ).fold().aggregate("avg_ttas")
    .by(unfold()
        .values("time_to_ambulance_starts")
        .mean())
    .unfold().out("RESPONSE_TRANSFER")
  .fold().aggregate("avg_tth")
    .by(unfold()
        .values("travel_time_hospital")
        .mean())
  .select("avg_ttas", "avg_tth")
```

Listing 5: A SQL query that returns the mean of the time it takes for ambulances to start and travel time to the hospital of priority 1 calls for a given hospital.

A.1.3 GSQL

```
CREATE QUERY postSim1() FOR GRAPH MyGraph {
    SetAccum<STRING> @@vSet;
    AvgAccum @@avgTth;
    AvgAccum @@avgTtas;

    INT priority_id = 1;
    FLOAT lat = 63.67;
    FLOAT lon = 19.11;
    FLOAT distKm = 0.5;
    priorities = { Priority.* };
```

```
    Grids = to_vertex_set(
        getNearbyGridId(distKm, lat, lon),
        "Geo_Grid");

    ResponsesByGeo =
        SELECT r
        FROM Grids:s-(Dest_Geo:e)-Response:r
        WHERE geoDistance(lat, lon,
            e.LATITUDE, e.LONGITUDE) <= distKm;

  ResponsesByPrio =
        SELECT r
        FROM priorities:p-(Response_Priority)-:r
        WHERE p.id == "1";

  ResponsesByGeoAndPrio =
        ResponsesByPrio UNION ResponsesByGeo;

  TargetResponses =
        SELECT h
        FROM ResponsesByGeoAndPrio:h
            -(Response_Transfer)->
            Transfer:t
        ACCUM
        @@avgTtas += h.TIME_TO_AMBULANCE_STARTS,
        @@avgTth += t.TRAVEL_TIME_HOSPITAL;

  PRINT @@avgTtas;
  PRINT @@avgTth;
}
```

Listing 6: A SQL query that returns the mean of the time it takes for ambulances to start and travel time to the hospital of priority 1 calls for a given hospital.

A.2 Second Resource to Transfer Patients

A.2.1 SQL

```
SELECT count(*) as responses_by_prio
FROM resource RSC
JOIN response RES ON RSC.response_id = RES.id
JOIN on_scene SCN ON SCN.response_id = RES.id
JOIN priority PRI ON PRI.response_id = RES.id
WHERE RSC.id = 2
    AND ST_DWithin(
      RES.destination,
      ST_MakePoint(19.11, 63.67)::geography,
      500)
GROUP BY PRI.id;
```

Listing 7: A SQL query that returns the total number of responses, grouped by priority of the call, that were the second resource and ended up transferring patients to a given hospital.

A.2.2 Groovy

```
g.V().hasLabel("Resource")
    .has("resource_id", "2")
        .in("RESPONSE_RESOURCE")
    .hasLabel("Response")
    .has("destination",
        geoWithin(
            Geoshape.circle(63.67, 19.11, 0.5))
        ).as("RES")
    .out("RESPONSE_SCENE").as("SCN")
      .select("RES")
    .groupCount()
        .by(out("RESPONSE_PRIORITY")
            .values("priority_id"))
```

Listing 8: A SQL query that returns the total number of responses, grouped by priority of the call, that were the second resource and ended up transferring patients to a given hospital.

A.2.3 GSQL

```
CREATE QUERY postSim2() FOR GRAPH MyGraph {
    SetAccum<STRING> @@vSet;
    GroupByAccum<
        STRING prio,
        SumAccum<INT> total
    > @@group;

    STRING resource_id = "2";
    FLOAT lat = 19.11;
    FLOAT lon = 63.67;
    FLOAT distKm = 0.5;
    resources = { Resource.* };

    Grids = to_vertex_set(
            getNearbyGridId(distKm, lat, lon),
            "Geo_Grid");

    ResponsesByGeo =
        SELECT r
        FROM Grids:s-(Dest_Geo:e)-Response:r
        WHERE geoDistance(lat, lon,
            e.LATITUDE, e.LONGITUDE) <= distKm;

    ResponsesByRsrc =
      SELECT r
        FROM resources:rsc
            -(Response_Resource)-
            Response:r
        WHERE rsc.id == resource_id;

    ResponsesByGeoAndRsrc =
        ResponsesByRsrc
            INTERSECT
        ResponsesByGeo;

    ResponsesByOnScene =
        SELECT r
        FROM ResponsesByGeoAndRsrc:r
            -(Response_Scene)-
            On_Scene;

    TargetResponses =
        SELECT h
        FROM ResponsesByOnScene:h
            -(Response_Priority)-
            Priority:p
        ACCUM @@group += (p.id -> 1);

    PRINT @@group;
}
```

Listing 9: A SQL query that returns the total number of responses, grouped by priority of the call, that were the second resource and ended up transferring patients to a given hospital.

A.3 Long Response Count

A.3.1 SQL

```
SELECT count(*)
FROM transfer TNS
JOIN response RES ON TNS.response_id = RES.id
WHERE RES.time_to_ambulance_starts
  + RES.on_scene_duration
  + TNS.travel_time_hospital > 15 * 60;
```

Listing 10: A SQL query that returns how many responses took longer than 15 minutes from the time the call was logged until the patient was at the hospital.

A.3.2 Groovy

```
g.V().hasLabel("Transfer").as("tth")
  .in("RESPONSE_TRANSFER").as("ttas", "osd")
  .where(math("ttas + osd + tth")
    .by(values("time_to_ambulance_starts"))
    .by(values("on_scene_duration"))
    .by(values("travel_time_hospital"))
    .is(gt(15 * 60)))
    .count()
```

Listing 11: A SQL query that returns how many responses took longer than 15 minutes from the time the call was logged until the patient was at the hospital.

A.3.3 GSQL

```
CREATE QUERY postSim3() FOR GRAPH MyGraph {
    SumAccum<INT> @@totalResponses;
    responses = { Response.* };

    LongResponses =
        SELECT res
        FROM responses:res
            -(Response_Transfer)-
            Transfer:tns
        WHERE
            res.TIME_TO_AMBULANCE_STARTS +
            res.ON_SCENE_DURATION +
            tns.TRAVEL_TIME_HOSPITAL > 15 * 60
        ACCUM @@totalResponses += 1;
    PRINT @@totalResponses;
}
```

Listing 12: A SQL query that returns how many responses took longer than 15 minutes from the time the call was logged until the patient was at the hospital.

A.4 Kate's Restaurant Recommendation

A.4.1 SQL

```
SELECT DISTINCT OtherReviews.user_id
FROM users
JOIN review KateReviews
    ON users.id = KateReviews.user_id
    AND users.id = "qUL3CdRRF1vedNvaq06rIA"
    AND KateReviews.stars > 3
JOIN business KateBus
    ON KateReviews.business_id = KateBus.id
JOIN review OtherReviews
    ON OtherReviews.user_id != KateReviews.
    user_id
    AND OtherReviews.business_id = KateReviews.
    business_id
JOIN bus_2_cat Bus2Cat
    ON OtherReviews.business_id = Bus2Cat.
    business_id
JOIN category Categories
    ON Bus2Cat.category_id = Categories.id
        AND Categories.name = "Restaurants"
```

Listing 13: A SQL query that returns "recommending users" of Kate (a user in the dataset). These are users who have reviewed restaurants that Kate has also been to with a rating above 3 stars.

```
SELECT review.stars, review.text, review.
    business_id
FROM review
JOIN business
    ON review.business_id = business.id
    AND review.user_id = "..."
    AND ST_DWithin(location,
        ST_MakePoint(-80.79, 35.15)::geography,
        5000)
    AND review.stars > 3
ORDER BY review.date DESC
LIMIT 10
```

Listing 14: A SQL query that returns the star rating, text and business ID for restaurants a user has reviewed above 3 stars.

A.4.2 Gremlin

```
g.V().has("User",
    "user_id", "qUL3CdRRF1vedNvaq06rIA")
    .as("kate")
  .outE("REVIEWS").has("stars", gt(3))
  .inV().in("REVIEWS").where(neq("kate"))
    .as("users")
  .out("REVIEWS").out("IN_CATEGORY")
    .has("name", "Restaurants")
  .select("users").dedup()
  .values("user_id").fold()
```

Listing 15: A Gremlin query that returns "recommending users" of Kate (a user in the dataset). These are users who have reviewed restaurants that Kate has also been to with a rating above 3 stars.

```
g.V().has("User", "user_id", "...")
    .outE("REVIEWS").has("stars", gt(3))
        .order().by("date", desc)
        .as("stars", "text")
    .inV().has("location",
        geoWithin(
            Geoshape.circle(35.15,-80.79, 5)
        )).as("business_id")
    .select("stars").limit(10)
    .select("stars", "text", "business_id")
    .by("stars").by("text").by("business_id")
```

Listing 16: A Gremlin query that returns the star rating, text and business ID for restaurants a user has reviewed above 3 stars.

A.4.3 GSQL

```
CREATE QUERY getSimilarUsersBasedOnRestaurants(
    VERTEX<User> p) FOR GRAPH MyGraph {
    SetAccum<STRING> @@userIds;
```

```
    categories = { Category.* };
    businesses = { Business.* };
    PSet = { p };

  Restaurants =
    SELECT b
    FROM businesses:b-(In_Category)->Category:c
    WHERE c.id == "Restaurants";

  PRatedBusinesses =
    SELECT b
    FROM PSet-(Reviews)->Business:b
    WHERE r.STARS > 3;

  PRatedRestaurants =
    PRatedBusinesses INTERSECT Restaurants;

  PeopleRatedSameBusinesses =
    SELECT tgt
      FROM PRatedRestaurants:m
          -(reverse_Reviews:r)->
          User:tgt
    WHERE tgt != p AND r.STARS > 3
    ACCUM @@userIds += tgt.id;

  PRINT @@userIds;
}
```

Listing 17: A GSQL query that returns "recommending users" of Kate (a user in the dataset). These are users who have reviewed restaurants that Kate has also been to with a rating above 3 stars.

```
CREATE QUERY getRecentGoodReviewsNearUser(
    Vertex<User> p) FOR GRAPH MyGraph {
    TYPEDEF tuple<DATETIME reviewDate,
        STRING businessId, INT stars,
        STRING text> restAndReview;

    DOUBLE lat = 35.15;
    DOUBLE lon = -80.79;
    INT distKm = 5;
    HeapAccum<restAndReview>
        (10, reviewDate DESC) @@busAndReviews;
    ListAccum<restAndReview> @@finalReviews;
    businesses = { Business.* };
    users = { User.* };
    PSet = { p };
    Grids = to_vertex_set(
        getNearbyGridId(distKm, lat, lon),
        "Geo_Grid");

  NearbyBusinesses =
    SELECT b
    FROM Grids:s-(Business_Geo:e)-Business:b
    WHERE geoDistance(lat, lon,
      e.LATITUDE, e.LONGITUDE) <= distKm;

  Restaurants =
    SELECT b
    FROM businesses:b-(In_Category)->Category:c
    WHERE c.id == "Restaurants";

  NearbyRestaurants =
      NearbyBusinesses INTERSECT Restaurants;

  NearbyRestReviewsByP =
    SELECT u
    FROM NearbyRestaurants:b
      -(reverse_Reviews:tgt)->
      User:u
    WHERE tgt.STARS > 3 AND u == p
    ACCUM @@busAndReviews +=
      restAndReview(tgt.REVIEW_DATE, b.id,
                    tgt.STARS, tgt.TEXT);
  FOREACH i IN RANGE[0, 9] DO
    @@finalReviews += @@busAndReviews.pop();
  END;

  PRINT @@finalReviews;
}
```

Listing 18: A GSQL query that returns the star rating, text and business ID for restaurants a user has reviewed above 3 stars.

A.5 Review Trends in Phoenix 2018

A.5.1 SQL

```
SELECT text, review.stars, cool, funny, useful
FROM business
JOIN review ON business.id = review.business_id
    AND ST_DWithin(
        location,
        ST_MakePoint(-112.56, 33.45)::geography,
        50000)
    AND date_part("year", date) = 2018)
```

Listing 19: A SQL query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

A.5.2 Gremlin

```
g.V().has("Business",
    "location", geoWithin(
        Geoshape.circle(33.45,-112.56, 50)))
    .inE("REVIEWS")
    .has("date", between(
      Instant.parse("2018-01-01T00:00:00.00Z"),
      Instant.parse("2018-12-31T23:59:59.99Z")
    )).valueMap()
```

Listing 20: A Gremlin query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

A.5.3 GSQL

```
CREATE QUERY getReviewsFromPhoenix2018()
    FOR GRAPH MyGraph {
    TYPEDEF tuple<STRING text, INT stars,
        INT cool, INT funny,
        INT useful> review;

    DOUBLE lat = 33.45;
    DOUBLE lon = -112.56;
    INT distKm = 50;
    ListAccum<review> @@reviewList;
    Grids = to_vertex_set(
        getNearbyGridId(distKm, lat, lon),
        "Geo_Grid");

    NearbyBusinesses =
```

```
      SELECT b
      FROM Grids:s-(Business_Geo:e)-Business:b
      WHERE geoDistance(lat, lon,
          e.LATITUDE, e.LONGITUDE) <= distKm;

    ReviewsForBusinesses =
        SELECT b
        FROM NearbyBusinesses:b
            -(reverse_Reviews:r)-
            User
        WHERE YEAR(r.REVIEW_DATE) == 2018
        ACCUM @@reviewList +=
          review(r.TEXT, r.STARS,
          r.COOL, r.FUNNY, r.USEFUL);

  PRINT @@reviewList;
}
```

Listing 21: A GSQL query that returns all the review text and ratings for businesses within 50km of the Phoenix area during 2018.

A.6 Ranking Las Vegas by Friends' Sentiment

A.6.1 SQL

```
SELECT DISTINCT R.text, R.stars FROM review R
JOIN business B ON R.business_Id = B.id
INNER JOIN friends F2 ON R.user_id = F2.
    friend_id
INNER JOIN friends F1 ON F2.user_id = F1.
    friend_id
WHERE F1.user_id = "7weuSPSSqYLUFga6IYP4pg"
    AND F2.user_id <> "7weuSPSSqYLUFga6IYP4pg"
    AND (R.user_id = F1.user_id
    OR   R.user_id = F1.friend_id)
    AND ST_DWithin(
        B.location,
        ST_MakePoint(-115.14, 36.16)::geography,
        30000)
    AND (date_part("month", R.date) >= 11
    AND  date_part("month", R.date) <= 12)
```

Listing 22: A SQL query that returns all the review text from reviews written by friends and mutual friends for businesses within 30km of the Las Vegas center.

A.6.2 Gremlin

```
g.V().has("User",
    "user_id", "7weuSPSSqYLUFga6IYP4pg")
    .as("julie")
    .out("FRIENDS").as("f1")
    .out("FRIENDS").as("f2")
    .union(select("f1"), select("f2"))
        .dedup().where(neq("julie"))
    .outE("REVIEWS").filter{
        it.get().value("date")
            .atZone(ZoneId.of("-07:00"))
            .toLocalDate().getMonthValue() >= 11
        &&
        it.get().value("date")
            .atZone(ZoneId.of("-07:00"))
            .toLocalDate().getMonthValue() <= 12
    }.as("text").as("stars")
    .inV().has("location", geoWithin(
        Geoshape.circle(36.16, -115.14, 30)))
    .select("text", "stars")
        .by("text").by("stars")
```

Listing 23: A Gremlin query that returns all the review text from reviews written by friends and mutual friends for businesses within 30km of the Las Vegas center.

A.6.3 GSQL

```
CREATE QUERY getFriendReviewsInArea(
    VERTEX<User> p, DOUBLE lat, DOUBLE lon)
    FOR GRAPH MyGraph {
    TYPEDEF tuple<STRING text,INT stars> review;
    SetAccum<review> @@reviews;
    SetAccum<VERTEX> @@F1F2;

    INT distKm = 30;
    users = { User.* };
    PSet = { p };
    Grids = to_vertex_set(
        getNearbyGridId(distKm, lat, lon),
        "Geo_Grid");

    NearbyBusinesses =
        SELECT b
        FROM Grids:s-(Business_Geo:e)-Business:b
        WHERE geoDistance(lat, lon,
            e.LATITUDE, e.LONGITUDE) <= distKm;

    F1 =
        SELECT f
        FROM PSet-(Friends)-User:f
        ACCUM @@F1F2 += f;

    F2 =
        SELECT f
        FROM F1-(Friends)-User:f
        ACCUM @@F1F2 += f;

    @@F1F2.remove(p);

    FReviewedBusinesses =
        SELECT b
        FROM users:f-(Reviews)-Business:b
        WHERE @@F1F2.contains(f);

    NearbyFBusiness =
        NearbyBusinesses
        INTERSECT
        FReviewedBusinesses;

    GetTheReviews =
        SELECT b
        FROM NearbyFBusiness:b
            -(reverse_Reviews:tgt)-
            User:u
        WHERE MONTH(tgt.REVIEW_DATE) >= 11
            AND MONTH(tgt.REVIEW_DATE) <= 12
            AND @@F1F2.contains(u)
        ACCUM @@reviews +=
            review(tgt.TEXT, tgt.STARS);

    PRINT @@reviews;
```

```
}
```

Listing 24: A GSQL query that returns all the review text from reviews written by friends and mutual friends for businesses within 30km of the Las Vegas center.

B Database Schemas

B.1 Medical Response Dataset

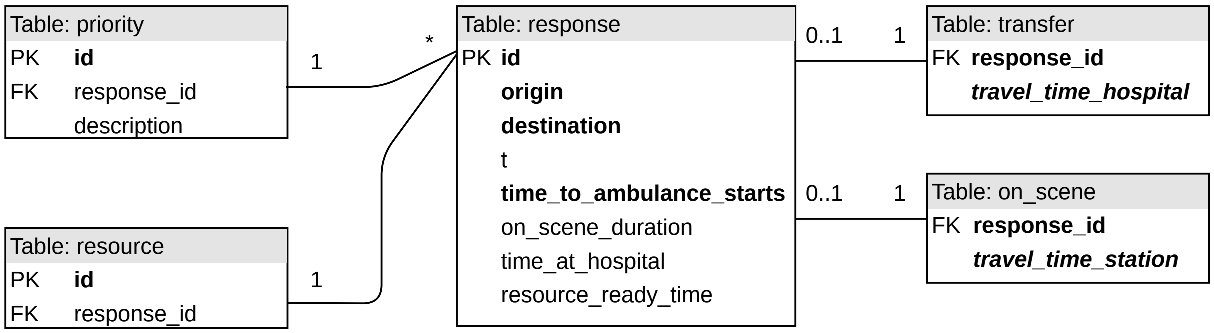

Figure 25: A UML diagram of the relational design of the simulation dataset. Indexed attributes are in bold whereas clustered attributes are in italics.

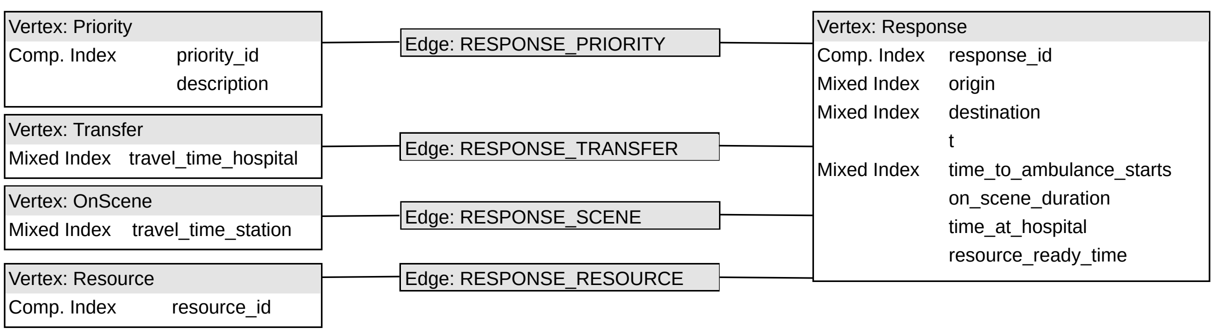

Figure 26: A UML diagram of the graph design of the simulation dataset in JanusGraph.

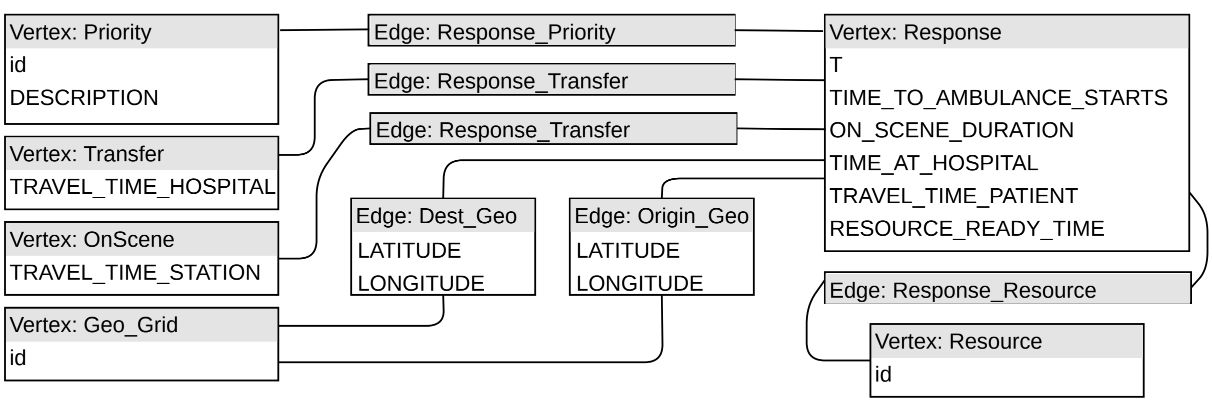

Figure 27: A UML diagram of the graph design of the simulation dataset in TigerGraph.

B.2 Yelp Dataset

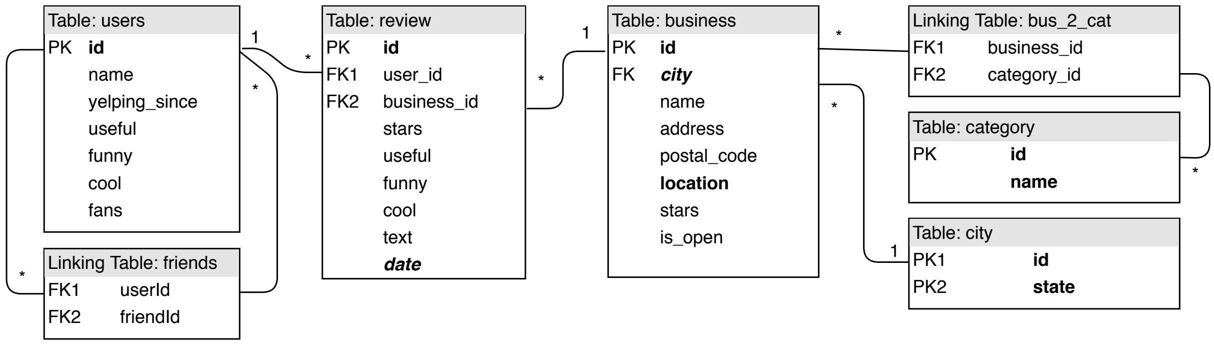

Figure 28: A UML diagram of the relational design of the Yelp dataset. Indexed attributes are in bold whereas clustered attributes are in italics.

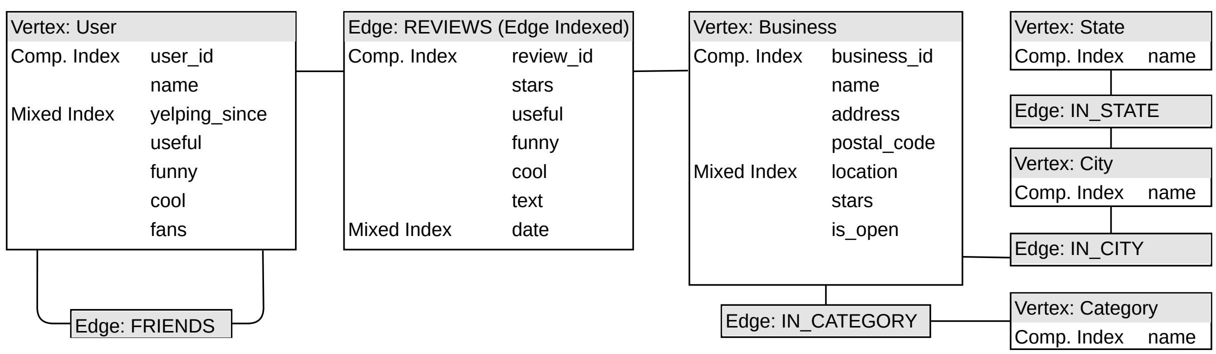

Figure 29: A UML diagram of the graph design of the Yelp dataset in JanusGraph.

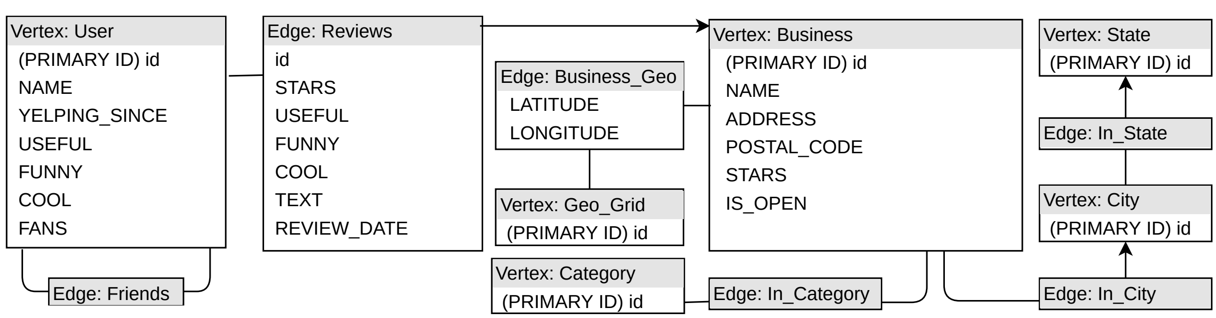

Figure 30: A UML diagram of the graph design of the Yelp dataset in TigerGraph.

C Data Analysis

C.1 Medical Response Simulation Results

Table 4: The result of the query "Priority 1 Mean Response Times". Values are in seconds.

Mean Time Till Ambulance Starts	Mean Time to Hospital
301.6636	880.6665

Table 5: The result of the query "Second Resource to Transfer Patients".

Number of Responses Grouped by Priority		
Priority 1	Priority 2	Priority 3
219	289	93

Table 6: The result of the query "Long Response Count".

Number of Responses Grouped by Priority
5284

C.2 Yelp Dataset Analysis

Table 7: The result of the analysis "Kate's Restaurant Recommendation". Only results with 5 reviews or more are displayed.

Businesses in Phoenix 2018		
Name	Pos Sentiment	Star Average
Paco's Tacos & Tequila	92.9825%	4.5833
Oak Steakhouse Charlotte	100.0%	5.0
The Cheesecake Factory	100.0%	4.4615
Block & Grinder	90.0%	4.3333
Best Wok	100.0%	4.2

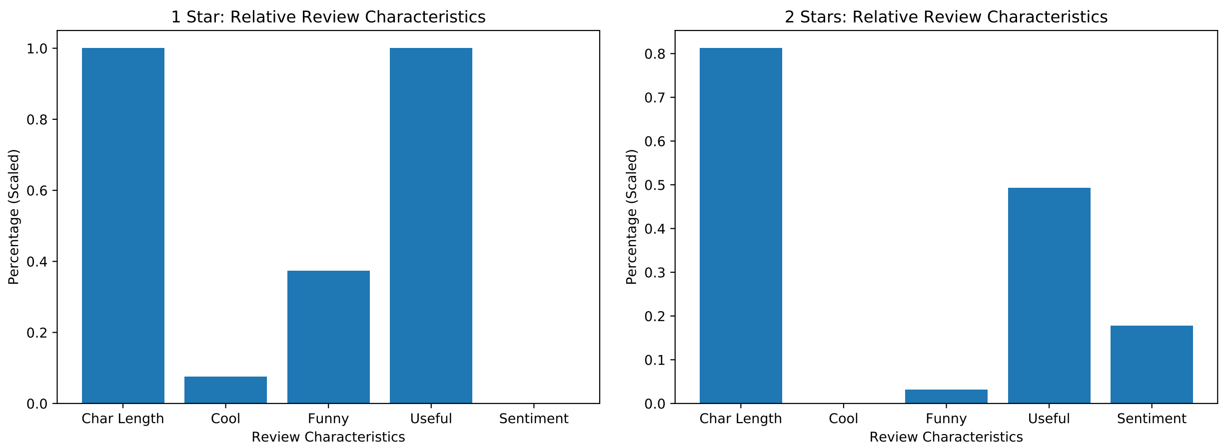

Figure 31: Results from analysis "Review Trends in Phoenix 2018". Displayed are relative characteristics of 1 and 2 star reviews over reviews of restaurants in Phoenix 2018.

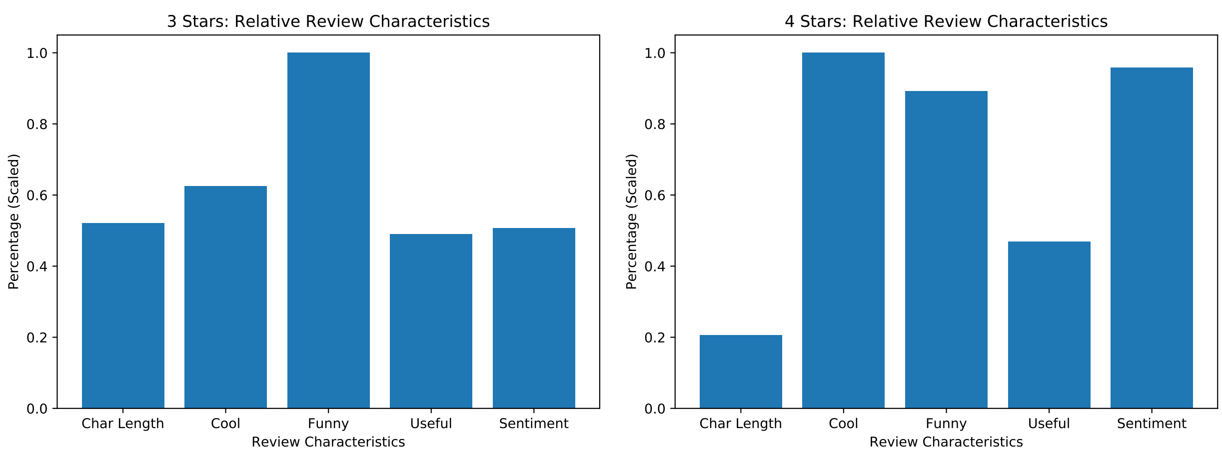

Figure 32: Results from analysis "Review Trends in Phoenix 2018". Displayed are relative characteristics of 3 and 4 star reviews over reviews of restaurants in Phoenix 2018.

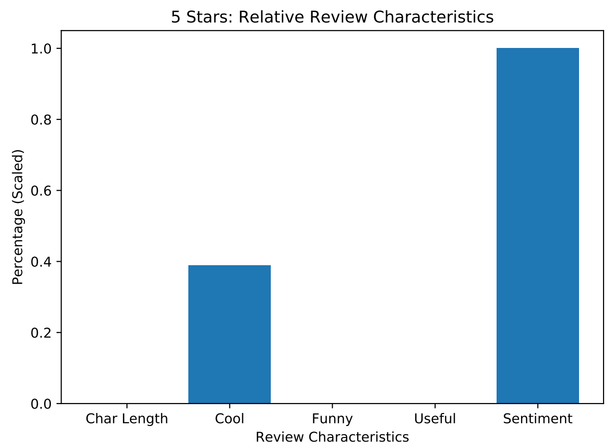

Figure 33: Results from analysis "Review Trends in Phoenix 2018". Displayed are relative characteristics of 5 star reviews over reviews of restaurants in Phoenix 2018.

Table 8: The result of the analysis "Ranking Las Vegas by Friends' Sentiment".

Las Vegas Sentiment vs Star Average	
Positive Sentiment (%)	Star Average
70.7767	3.8917