Comparison of Graph Query Languages: A Survey

Archit Joshi
aj6082@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Athina Stewart
as1896@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Nicholas Gardner
nag6650@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

1 INTRODUCTION

In recent years, the advent of interconnected data structures has revolutionized the way we approach and analyze complex relationships in datasets. Although the notion of 'graph data' has a long history of development since the 1980s [7], this paradigm shift has been chiefly driven by the emergence of graph databases, powerful tools designed specifically to navigate and query highly connected information.

The rise of graph databases can be in part attributed to the exponential growth of social media which required understanding the relationships among users, their interests and tailoring content based on these attributes. These platforms required a new approach to data management which could seamlessly capture and traverse intricate webs of connections and complicated enough to prevent realization through conventional RDBMS paradigms. Beyond social media, graph databases have found applications in diverse domains such as recommendation engines, fraud detection, supply chain optimization, and knowledge graphs. They mirror the way we naturally think about and visualize connections in our world. This innate representation of complex inter-dependencies, coupled with their computational prowess in traversing vast networks, empowers us to unlock insights that would otherwise remain undetected in conventional tabular data structures.

This survey aims to delve into one of the critical facets of graph databases: query languages. These specialized languages play a pivotal role in harnessing the full potential of graph databases by bridging the gap between Data Defintion Language (DDL), Data Manipulation Language (DML) and computational graph algorithms. These languages serve as the bridge between raw data and actionable insights, enabling analysts and developers to pose complex questions about relationships, patterns, and structures within their datasets.

In the following sections, we will explore the basics of graph databases and what makes them stand out, the significance of query languages in the context of graph databases, their diverse use cases, and conduct a comparative analysis to elucidate their strengths and weaknesses. By the end of this survey, readers will gain a comprehensive understanding of the various query languages available, their applicability in different scenarios, and the pivotal role they play in harnessing the true potential of graph databases.

2 BACKGROUND

Graph query languages play a pivotal role in the realm of graph databases, facilitating efficient retrieval and manipulation of graph data. Query languages that will be explored in this survey include: Gremlin [13], Cypher [9], and SPARQL [1]. In the world of relational databases, SQL has been the dominant query language for good reason: SQL is efficient at working with structured data. Relational databases organize data into tables with rows and columns, and SQL provides a natural and expressive way to query and manipulate this structured data. It was specifically designed to have its syntax and capabilities closely align with the structure of a relational database. Relational databases adhere to ACID (Atomicity, Consistency, Isolation, Durability) properties and SQL provides mechanisms for defining and managing transactions to ensure data integrity.

However, SQL can often be ill-suited for graph database applications due to the nature of how graph data is represented. Graph databases work with graph data models and data is represented by nodes (entities), edges (relationships) and properties (key-value data associated with nodes and edges). Therefore, the operations needed in querying most often involve traversing paths, finding connected nodes, and identifying patterns, such as paths, cycles, and trees. Relational databases are optimized, on the other hand, for joins, aggregations, and set-based operations.

Each of the graph query languages discussed in this paper were developed to address specific needs within their respective subfields. Cypher, the query language for Neo4j, was designed to simplify querying property graphs (a type of graph database where nodes and relationships can have properties (key-value pairs)) by introducing a pattern-matching syntax [9]. SPARQL, used in RDF databases, focuses on querying linked data and semantic web information. RDF (Resource Description Framework) is a data model for representing linked data and semantic web information. Gremlin, part of the Apache TinkerPop framework, adopts a functional and expressive approach for traversing graph structures.

Distinct syntactical and semantic choices were made for each query language in response to the specific needs of their target databases and use cases. For instance, Cypher's syntax resembles ASCII art patterns to make graph querying intuitive and readable. SPARQL, on the other hand, aligns closely with RDF's triple structure to support semantic querying (data is structured as triples, consisting of a subject, predicate, and object)[5]. Gremlin adopts a flexible and functional style to cater to a wide range of graph databases. These design choices reflect the need for both simplicity and expressiveness, as well as adaptability to diverse data models and query requirements.

3 GRAPH QUERY LANGUAGES

In the following sections we explore the most commonly used and supported graph query languages - Cypher which is a declarative query language for property graphs, Gremlin which is a vendor agnostic graph traversal language and SPARQL which supports retrieval and manipulation of data stored in RDF (Resource Description Framework) format.

3.1 Cypher

Cypher property graph language, originally developed for the Neo4j graph database, is an evolving language which is now widely used by several commercial database products. Cypher has evolved considerably over the past few years and is now used in products such as SAP HANA Graph, Redis Graph, Agens Graph and Memgraph. The developer's goal is to enable the use of Cypher as the industry standard for graph databases and a framework for cross-implementer evolution for new language features [9].

3.1.1 *Graph Model.*

Cypher is a declarative query language for property graphs [6]. A property graph is the most widely accepted and used graph model in the industry and academia. The model is comprised of nodes, which act as entities, and relationships, which represent the connections between nodes as seen in Figure 1. There can be any number of key:value pairs for both nodes and edges, each of which representing an attribute of the respective node or edge.

You
ARE_LEARNING
Cypher

Figure 1: A bare bones visualisation of property graphs.

3.1.2 *Properties.*

The Cypher query language can be summarized by the following properties: linear queries, pattern matching, data modification, pragmatic and Neo4j implementation. As Cypher is declarative, **its queries are linear**. The query takes in a graph as an input and then outputs a table which can be thought of as bindings of parameters for which a certain pattern was observed. Using the **WITH** keyword, a user can chain the clauses which allows the table from the first part of the clause to be used for processing in the second clause of the query. It also supports nested subqueries using the **UNION** keyword.

The core concept in Cypher queries is **pattern matching**. Cypher uses an "ASCII art" visual form and **MATCH** clause to perform pattern matching and evaluate a graph. One example representation of the ASCII art would be (a) - [r] -> (b) where the "()" represent that the entity is a node (a and b) in this case and "[]" would represent a relationship with label "r". So putting the entire expression together describes a pattern where there is a node labeled "a" connected to a node labeled "b" by a relationship labeled "r". The relationship is directed from "a" to "b". In a Cypher query, this pattern could be used to match instances in a graph where this specific structure exists. For example, you might use a query like:

```
MATCH (a) - [r] -> (b)
RETURN a, r, b
```

Cypher is built for performing updates and modifying graphs. The clauses used by Cypher to perform updates include **CREATE**, **DELETE** and **SET** for creating, removing and updating properties. Cypher might look eerily similar to SQL which is intentional for the purpose of a smooth transition between the two. To simplify and secure queries, Cypher has in-built support for query parameters which prevents query injections and has powerful features such as list slicing and comprehension.

3.1.3 *Examples.*

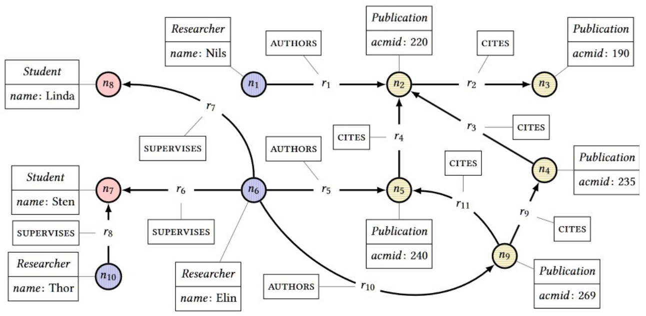


Figure 2: Example data graph showing supervision and citation data for researchers, students and publications. [9]

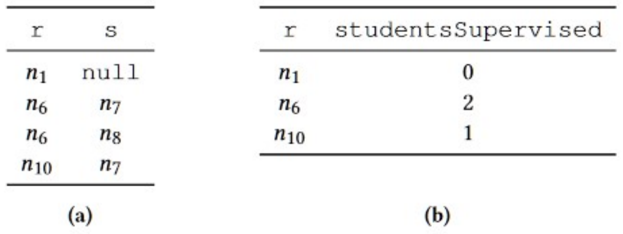
Figure 2 from the referenced paper represents a data graph (G) consisting of researchers, students and publications. The variables denoted by "n" represent the nodes whereas the variables denoted by "r" represent the relations.

The query below will return the name of the researcher in G, the number of students they supervise, and the number of times their publication has been cited by other publications. The query has been referenced from the research paper [9].

1 **MATCH** (r:Researcher)
2 **OPTIONAL MATCH** (r)-[:SUPERVISES]->(s:Student)
3 **WITH** r, **count**(s) **AS** studentsSupervised
4 **MATCH** (r)-[:AUTHORS]->(P_1:Publication)
5 **OPTIONAL MATCH** (P_1)<-[:CITES*]-(P_2:Publication)
6 **RETURN** r.name, studentsSupervised,
7 **count**(**DISTINCT** P_2) **AS** citedCount

Line 1 will produce a matching for all nodes marked as researchers – n_1, n_6 and n_{10}. The **OPTIONAL MATCH** clause in line 2 acts like an outer join in SQL – if there is no corresponding match for a pattern it will set the binding to NONE resulting in Figure 3(a), the clause returns variables "r" and "s". The **WITH** clause in line 3 will perform the projection and aggregation with the current variables in scope and count the number of students each researcher supervises, resulting in Figure 3(b), As you can see variable "s" is not included in Figure 3(b) as it was not returned by the **WITH** clause in line 3.

Line 4 proceeds to match the variable "r" with a new variable "P_1" to denote the publications the researchers have authored using the **MATCH** clause resulting in Figure 4a. Similar to before, we also have the **OPTIONAL MATCH** in line 5 to match the publications authors by "r" in our graph to any other publication, "P_2", which cite it directly or indirectly resulting in Figure 4b. Notice we have a null



r	s
n_1	null
n_6	n_7
n_6	n_8
n_{10}	n_7

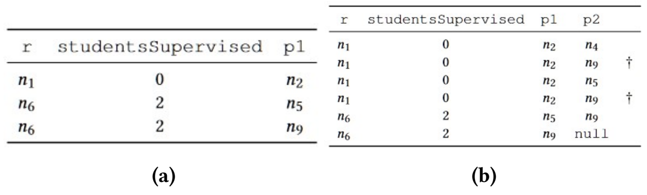
(a)

r	studentsSupervised
n_1	0
n_6	2
n_{10}	1

(b)

Figure 3: Variable bindings till line 3 in the code example. [9]

entry as this operation functions as an outer join like previously discussed.



r	studentsSupervised	p1
n_1	0	n_2
n_6	2	n_5
n_6	2	n_9

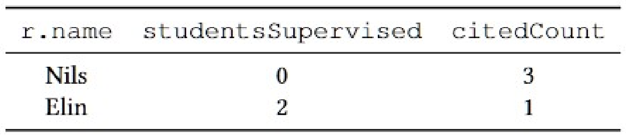
(a)

r	studentsSupervised	p1	p2	
n_1	0	n_2	n_4	
n_1	0	n_2	n_9	†
n_1	0	n_2	n_5	
n_1	0	n_2	n_9	†
n_6	2	n_5	n_9	
n_6	2	n_9	null	

(b)

Figure 4: Variable bindings for lines 4 and 5. [9]

Line 6 is the return statement which will return the variables "r", "studentsSupervised", and the count of citations "citedCount" which is an aggregation performed by line 7. Our final result can be seen in Figure 5.



r.name	studentsSupervised	citedCount
Nils	0	3
Elin	2	1

Figure 5: Final bindings and output result from lines 6 and 7.[9]

As mentioned earlier, Cypher supports list comprehension and slicing. This improves its readability and flexibility with data querying and manipulation. The entire list of clauses with examples can be found on the official documentation for Neo4j [3]. For this literature review we will focus on the **SKIP** and **LIMIT** clauses only.

To demonstrate the use of **SKIP** clause we will use a generic graph from the Neo4j documentation which shows nodes A, B, C, D and E and a "Knows" relationship between them (Figure 6). To return a subset of the "Knows" relation starting at the 3rd result, we can execute the query below:

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 2
```

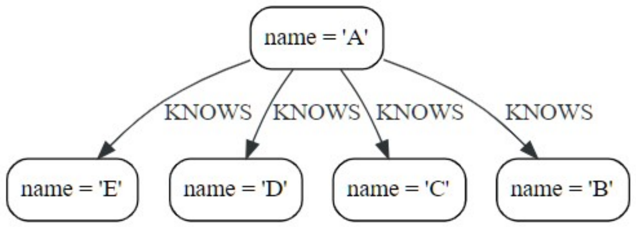


Figure 6: Generic graph to demonstrate SKIP and LIMIT clauses. [3]

The query above will have an intermediate result returning all of the nodes as a single column "n.name" in the order – A, E, D, C, B. The **ORDER BY** clause will order the result in alphabetical order changing the result to – A, B, C, D, E. The **SKIP** clause will skip the first 2 rows and return the table *n.name* with only C, D and E which will be the final result.

If we want to return only the middle two nodes we can employ the **LIMIT** as shown next. The below query which is similar to the one above, will skip the first node and, because of the limit clause, return only the next 2 nodes in the *n.name* table after the skip is performed – B and C.

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 1
LIMIT 2
```

3.2 Gremlin

Gremlin is a graph traversal machine and language, produced by the Apache TinkerPop project, that can be executed over a variety of supporting graph database computing systems including *Online Transaction Processing* (OLTP) graph databases and *Online Analytical Processing* (OLAP) graph processors. Its graph traversal machine functionality can be broken into three components: a **graph**, a **traversal**, and a set of **traversers**.

3.2.1 *Graph*.

Gremlin operates on a multi-relational, attributed, digraph, G, that is represented by a set of vertices V, a set of directed edges E, and a collection of element/property pairs, λ that yield values from the universal set, U. Each node can be related to other nodes with a variety of relations, and can contain any number of key-value pairs representing attributes.

3.2.2 *Traversal*.

The traversal, ψ, represents the series of steps to be executed, or the pattern to match, for each traverser (gremlin). Essentially, in terms of more traditional database languages, this can be thought of as the query/query plan. Gremlin traversals are made up of components that can be classified into 5 major types:

(1) **map**: Traversers at objects of type A are mapped to traversers at objects of type B without changing the size of the set of

traversers. This applies to operations like getting the age for all nodes at the current step of traversal.

(2) **flatMap**: Again, traversers are mapped but this time the size of the traversal set may change. This represents operations like getting all nodes with a particular relation to the current traversal node.
(3) **filter**: Traversers are retained or removed depending on if they meet certain criteria. For instance, if the current node has an age greater than 30.
(4) **sideEffect**: Some change is made to a data structure (typically part of the graph). This describes cases such as when a property is added to a node.
(5) **branch**: Occurs when the traversal takes a different path depending on conditional statements. For instance, a choose or an if/else.

3.2.3 *Traversers*.

Traversers, $t \in T$, are the workhorses of the Gremlin implementation. They each represent a read/write head located somewhere in the graph at some step in the traversal. The name of the language as "Gremlin" is a playful reference to these traversers as individual gremlins popping up throughout the graph. Essentially, traversers are a bundle of local variables (step counter, metadata, etc) with a location in the graph and a location (chronological) in the traversal. Additionally, the number of traversers in the set may change throughout execution. They are furcating automata, meaning that if more than one option meets the traverser's criteria, the traverser will clone itself and place identical traversers at each option.

3.2.4 *Traversal Language*.

The Gremlin traversal language is a functional language that is implemented in the user's native programming language, and is designed for maximimizing simplicity when defining a traversal. Other database querying languages use SQL or their own unique language to define queries. When used as part of an application, this requires messily combining multiple languages through string formatting that must be changed if a database vendor migration occurs. This is not the case with Gremlin. Gremlin is vendor-agnostic, capable of both imperative and declarative styles, and integrates cleanly with existing application code. The only requirement for Gremlin's inclusion in application code is that the native language supports function composition and function types.

3.2.5 *Traversal Examples*.

- **Simple Traversal**

  g.V().has("name","marko").
  out("knows").values("age").max()

  This traversal represents a very simple question: "what is the age of the oldest person that marko knows?" The first function, *g.V()*, places traversers at every vertex in the graph, *g*. Next, we filter these traversers by whether or not the vertex has property "name" with value "marko" (*.has("name", "marko")*). This yields node 1. Next, we get all outgoing relations of type "knows" with *.out("knows")*. This places traversers at nodes 2 and 4. Next, we call *.values("age")*, which yields {32, 27} from our current nodes. Finally, we use *.max()* to return 32.

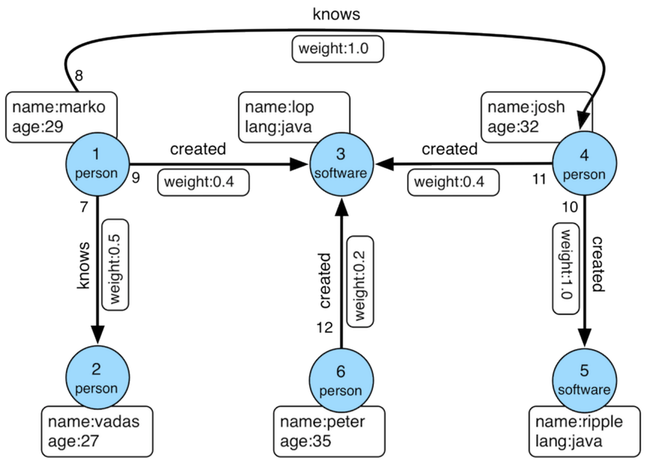


Figure 7: Simple multi-relational, attributed, digraph for use as an example. [2]

- **Branching Traversal**

  g.V().choose(label()).
  option("person", out("created").count()).
  option("software", in("created").count()).
  option(none, label())

  In this traversal, different choices are made depending on what type of node the traverser encounters. We again begin by placing traversers at all nodes. Next, we *choose* based on the node label. In the case of our example graph, nodes are either labelled with "person" or "software". If the node is a person, we return the number of software items that this person created. If the node is a software, we return the number of contributors who created this software. This will return a list of counts for each node (in order of nodes): {1, 0, 3, 2, 1, 1}.
- **Recursive Path Traversal**

  g.V(x).repeat(out().simplePath()).
  until(is(y)).path().limit(1)

  The function *repeat()* is used for recursion with either *.times(n)* to indicate we want to repeat n times or *.until(condition)* to stop once we have reached the condition specified in the until block. In this case, the traversal will return the shortest, non-looping path from x to y.
- **Projecting Path Traversal**

  g.V().as("a").out("knows").as("b")
  select("a","b").
  by(in("knows").count()).
  by(out("knows").count())

  This traversal demonstrates functionality for returning to a previous part of the traversal when necessary. Here we set traversers at every vertex in *g* and save the node's location as "a". Then, we get outgoing "knows" relations and save them as "b". Next, we create two new traversers, using *select()*, that will traverse the "a" and "b" node for each current traverser.

We specify the traversal for each using *by()*, returning the inbound "knows" relations for our "a" nodes and the outbound "knows" relations for our "b" nodes.

- **Graph Centrality Traversal**

  ```
  g.V().repeat(groupCount("m").out()).
      times(30).cap("m")
  ```

  It is often necessary to produce an aggregate measurement for the graph as a whole. This query measures *eigenvector centrality*, which is defined as the probability that any vertex will be visited by a random walk at some point in time. *groupCount("m")* increments a map containing vertexes as keys and the number of times that a vertex has been encountered in this exploration as values. 30 repetitions is not used for any particular reason other than that we will likely achieve convergence by then. *cap("m")* changes the output from the set of traversers to a single traverser that is a function of the data structure, *m*.
- **Mutating Traversal**

  ```
  g.V().as("a").out("created").
  addOutE("createdBy","a")

  g.V().outE("created").drop()
  ```

  Gremlin traversals can also mutate the graph that they are traversing. The first traversal here adds an inverse "createdBy" relation for every "created" relation. The second traversal drops all "created" relations from the graph.
- **Declarative Traversal**

  ```
  g.V().match(
  as("a").out("created").as("b"),
  as("b").in("created").count().is(gt(3)),
  as("b").in("created").as("c"),
  as("a").out("father").as("c")).
  dedup("a").
  select("a").by("name")
  ```

  Gremlin supports graph pattern matching similarly to SPARQL. In contrast to SPARQL, Gremlin's declarative implementation can be contained in a single step of the traversal with imperative steps around it. Additionally, pattern matching is a traversal in Gremlin, allowing it to benefit from all of the compatibilities and optimizations of the Gremlin traversal machine.
  match() contains a set of patterns that can be chained with logical operators *not()*, *or()*, or *and()*. It is essentially a recursive branch step where each branch is taken exactly once. The above pattern matching traversal will return the name of all of the people that created a piece of software in collaboration with 4 or more people where one of those collaborators was this person's father.
- **Domain-Specific Traversal**

  ```
  g.people().named("marko").
  who().know(well).people().
  who().created("software").
  are().named()
  ```

  The Gremlin traversal language syntax more or less mirrors the functionality of the Gremlin traversal machine. However, the syntax for a custom, domain-specific version of Gremlin's functionality can be quickly created and compiled. The above traversal makes substitutions such as "named("marko")" for "has("name", "marko")".

3.2.6 *Traversal Strategies*.

Traversals are optimized natively by Gremlin, with several broad categories describing the types of rewrites that are conducted:

(1) **Decoration**: Steps are only syntactic placeholders.
(2) **Optimization**: There is a more efficient form for a particular step.
(3) **Vendor System**: There is a more efficient form *given* the underlying graph database.
(4) **Finalization**: There are adjustments that can be made to the final form of the traversal.
(5) **Verification**: Certain constraints need to be validated.

3.2.7 *Distributed Traversals*.

Gremlin also supports running on a distributed database. Each node is represented as a logical processor, receives traversers as messages, executes the step contained in the traverser's traversal, and sends a message to the next node. The halted locations of all of the traversers in the graph are the output of the distributed computation. Additionally, traversers can migrate between network clusters, and computation is conducted in a breadth-first, parallel manner. This could lead to an exponential amount of traversers, but Gremlin implements a *bulking* methodology that groups traversers at nodes by equivalence classes and collapses them.

3.3 SPARQL

SPARQL is a standard query language for querying and manipulating *Resource Description Framework* (RDF) data. RDF is a data model for representing information on the World Wide Web. When the web was released in 1998, a problem immediately arose as to how to query it. RDF is a directed labeled graph data format and so it fits that SPARQL is a graph-matching query language [11].

A SPARQL query is in the form $H \leftarrow B$ and consists of two main parts:

(1) The body, B, describes what is to be found in the data. The body may consist of complex RDF graph pattern expressions.
(2) The head, H, describes how the answer of the question should be presented.

To evaluate the query, Q, against dataset, D, two steps are carried out. First, SPARQL matches the description in the "body" of the query. This step involves looking for specific patterns, using variables, and applying conditions. From this step, a set of bindings are obtained from the "body". Next, the bindings are processed according to the instructions in the "head" of the query. This involves applying classical relational operators, such as projection or distinct, to produce the answer.

3.3.1 *SPAQRL Syntax*.

Within the SPARQL language exist the SELECT, CONSTRUCT, and ASK Clauses. This specifies the kind of information to be retrieved or constructed.

1. **SELECT** : Retrieves specific variables from the data.
2. **CONSTRUCT** : Creates a new RDF graph from the data.
3. **ASK** : Determines whether a pattern exists in the data (returns true or false).

SPARQL uses the following operators to combine patterns in the data:

1. **OPTIONAL**
2. **UNION**
3. **FILTER**
4. **Concatenation via '.'**

If P, P_1 and P_2 are graph patterns and R is a SPARQL built-in condition:

1. (P_1 AND P_2) combines patterns using an AND operator.
2. (P_1 OPT P_2) creates optional patterns, which might or might not be in the results.

(3. P_1 UNION P_2) combines patterns using a UNION operator.

4. (P FILTER R) adds conditions to your patterns using a FILTER operator.

Additionally, '{ }' can be used to group patterns and the '.' symbol has precedence over the OPTIONAL(OPT), UNION and FILTER. SPARQL also contains built-in Conditions involving variables, constants, logical operations (like NOT, AND, OR), comparison symbols (like less than, greater than), and equality symbols (like equal to). Additionally, there are unary predicates (special functions) like "bound", "isBlank", and "isIRI" that can be part of these conditions.

There are also conditions based on Boolean combinations of terms created using the '=' symbol and the "bound" predicate:

1. bound($?X$): This checks if the variable $?X$ has a value (i.e., it is bound).
2. $?X = c$: This checks if the variable $?X$ has a value equal to a constant 'c'.
3. $?X = ?Y$: This checks if two variables $?X$ and $?Y$ have the same value.
4. $\neg R_1$: This is the NOT operator applied to another built-in condition, R_1.
5. $R_1 \vee R_2$: This is the OR operator applied to two built-in conditions, R_1 and R_2.
6. $R_1 \wedge R_2$: This is the AND operator applied to two built-in conditions, R_1 and R_2.

Figures 8, 9, and 10 provide an example of querying using SPARQL.

3.3.2 *SPAQRL Semantics.*

Some aspects of the SPARQL semantics include:

1. Mapping μ: In SPARQL, a "mapping" μ is a way to assign values, U, to variables, V, in graph patterns. For example, if there is a triple pattern such as $?X$, hasAge, $?Y$, a mapping μ can assign values as $?X \longrightarrow$ "Alice", $?Y \longrightarrow 30$.
2. Domain (dom(μ)): The "domain" of a mapping μ, denoted as dom(μ), is the set of variables for which the mapping provides assignments. For example, if $\mu = ?X \longrightarrow$ "Alice", $?Y \longrightarrow 30$, then the domain is $\{?X, ?Y\}$.
3. Compatibility: Two mappings μ_1 and μ_2 are "compatible" if they can work together without conflicts. In other words, they can be combined into a new, larger mapping without any contradictions. For example, if μ_1 assigns $?X \longrightarrow$ "Alice" and μ_2 assigns $?Y \longrightarrow 30$, they are compatible because they involve different variables.
4. Join ($\bowtie$): $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2$ and μ_1, μ_2 are compatible mappings$\}$. The join of two sets of mappings, but only if they are compatible.
5. Union (U): $\Omega_1 \cup \Omega_2 = \{\mu \mid \mu \in \Omega_1$ or $\mu \in \Omega_2\}$. The union of two sets of mappings. This operation combines two sets of mappings without considering compatibility.
6. Difference (\\): $\Omega_1 \setminus \Omega_2 = \{\mu \in \Omega_1 \mid$ for all $\mu \in \Omega_2, \mu$ and μ are not compatible$\}$. This is the set of mappings that belong to set 1 but cannot be extended with any mapping from set 2.

3.3.3 *SPAQRL Complexity.*

According to Perez, Arenas, and Guitierrez (2009), in the exploration of SPARQL query evaluation complexity, SPARQL query patterns are PSPACE-complete. In computational complexity theory, a decision problem is PSPACE-complete if it can be solved using an amount of memory that is polynomial in the input length (polynomial space) and if every other problem that can be solved in polynomial space can be transformed to it in polynomial time. Essentially, for these queries, they require resources that grow exponentially with the size of the input.

From a practical perspective, PSPACE-completeness implies that as the size of the input (e.g., the dataset or the complexity of the query) increases, the computational resources required to evaluate SPARQL patterns can become prohibitive. This has implications for the efficiency of SPARQL query processing in real-world scenarios.

This result indicates that the general problem of evaluating SPARQL patterns is theoretically hard. It suggests that finding an optimal solution for general SPARQL patterns may be challenging and could potentially involve exponential time and space complexity [12].

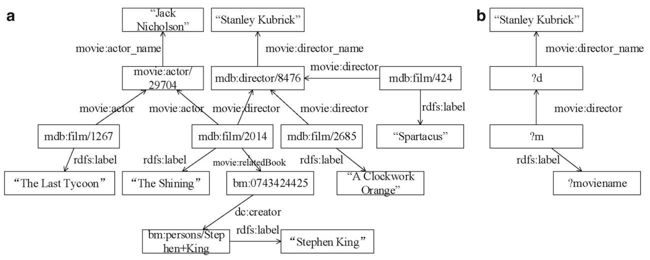
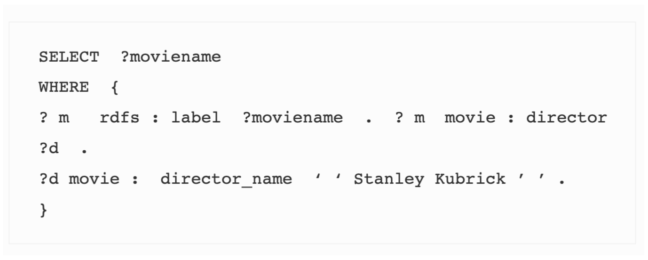
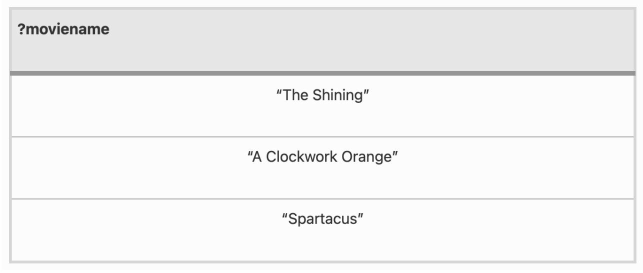


Figure 8: RDF graph and SPARQL query graph. [16]



```
SELECT  ?moviename
WHERE  {
? m   rdfs : label  ?moviename  .  ? m  movie : director
?d  .
?d movie :  director_name  ' ' Stanley Kubrick ' ' .
}
```

Figure 9: Example SPARQL Query. [16]



?moviename
"The Shining"
"A Clockwork Orange"
"Spartacus"

Figure 10: Example query result. [16]

4 DISCUSSION

After a thorough literature review we have come to understand that there cannot be a fair and direct comparison between Cypher, Gremlin and SPARQL. Each one of these languages is tailored for a specific use-case and thus do not have basis for a direct comparison. The research to compare these languages directly is a work in progress. But there do exist comparisons between a subset of the languages and existing relational concepts to highlight the prowess of graph databases and their adoption. In the following sections we discuss how different factors of Cypher, Gremlin, and SPARQL can be compared relative to their use-cases and semantics.

4.1 Graph Databases vs Relational databases

Besides motivating the use of particular graph database querying languages, it is important to motivate the use of graph databases themselves. If they do not ever demonstrate better performance than relational databases, then understanding their nuances would seem quite irrelevant. To provide a quantitative comparison, Do et al. [8] pit Neo4j (Cypher) against MySQL.

4.1.1 ***Dataset****.*

The dataset chosen for this case study is from CareerVillage.org, which is a non-profit that connects underprivileged youth with knowledgeable professionals. Essentially, the site operates much like StackOverflow or Quora where questions are asked that are then answered by experts. This dataset contains tens of thousands of students, professionals, questions, and answers that are densely connected. Refer to Figure 11 for a breakdown of each category by number of entries.

The first (and most obvious) difference between the relational and the graph implementation is how the dataset is represented. Figure 12 shows what the dataset looks like when stored in a relational format, and Figure 13 shows the same but for the graph implementation. Objects like *tag_questions* and *matches* are stored as secondary tables in MySQL, but are stored as edges in Neo4j. In order to perform relational operations on these and other categories, joins must be performed to aggregate all necessary data. For Neo4j, these relationships are stored right alongside the node data and are more computationally effecient to access. Additionally, it is intuitively easier to understand the composition of the dataset by looking at the graph representation than the relational.

4.1.2 ***Results****.*

Table/Object	Row in MySQL	Object in Neo4j
professionals	28,152	28,152 nodes
students	30,971	30,971 nodes
questions	23,931	23,931 nodes
answers	51,123	51,123 nodes
comments	14,966	14,966 nodes
tags	16,269	16,269 nodes
tag_questions	76,553	76,553 edges
tag_users	136,663	135,907 edges
groups	49	49 nodes
group_memberships	1,038	1,038 edges
emails	1,850,101	1,850,101 nodes
matches	4,316,275	1,116,275 edges
schools_memberships/ schools	5,638	2,706 nodes

Figure 11: Dataset description from Do et al. [8]

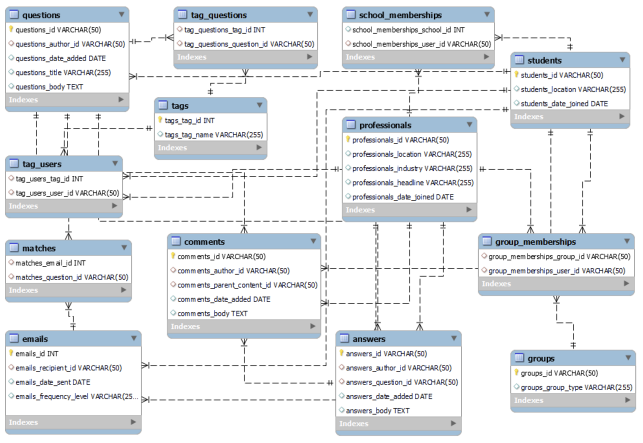

questions
questions_id VARCHAR(50)
questions_author_id VARCHAR(50)
questions_date_added DATE
questions_title VARCHAR(255)
questions_body TEXT
Indexes
tag_questions
tag_questions_tag_id INT
tag_questions_question_id VARCHAR(50)
Indexes
school_memberships
school_memberships_school_id INT
school_memberships_user_id VARCHAR(50)
Indexes
students
students_id VARCHAR(50)
students_location VARCHAR(255)
students_date_joined DATE
Indexes
tags
tags_tag_id INT
tags_tag_name VARCHAR(255)
Indexes
professionals
professionals_id VARCHAR(50)
professionals_location VARCHAR(255)
professionals_industry VARCHAR(255)
professionals_headline VARCHAR(255)
professionals_date_joined DATE
Indexes
tag_users
tag_users_tag_id INT
tag_users_user_id VARCHAR(50)
Indexes
matches
matches_email_id INT
matches_question_id VARCHAR(50)
Indexes
comments
comments_id VARCHAR(50)
comments_author_id VARCHAR(50)
comments_parent_content_id VARCHAR(50)
comments_date_added DATE
comments_body TEXT
Indexes
group_memberships
group_memberships_group_id VARCHAR(50)
group_memberships_user_id VARCHAR(50)
Indexes
emails
emails_id INT
emails_recipient_id VARCHAR(50)
emails_date_sent DATE
emails_frequency_level VARCHAR(25...
Indexes
answers
answers_id VARCHAR(50)
answers_author_id VARCHAR(50)
answers_question_id VARCHAR(50)
answers_date_added DATE
answers_body TEXT
Indexes
groups
groups_id VARCHAR(50)
groups_group_type VARCHAR(255)
Indexes

Figure 12: Dataset shown as relational tables. [8]

Figure 14 shows the results from evaluations performed in Do et al. [8]. For this dataset and query selection, the graph database implementation performs better in all cases. In fact, the performance is magnitudes better in some cases – maxing out at **146x** for query 4 (recursive). In other cases, performance is quite similar. For more simple queries like query 1 or query 10, no significant difference is seen between performance of both implementations. In general, simple retrieval queries are not improved by using Neo4j, but once joins get involved for the relational implementation, the difference becomes obvious. To better understand some of the best use cases for a graph database, this paper investigates a few queries that saw significant improvement gain when converting to a graph representation.

4.1.3 ***Query Investigation****.*

Query 2 (Figure 15) from Do et al. [8] falls in the category of selection/search. Essentially, the goal of this query is to find students that are in a specific group and have a specific tag (in this case students that are in a youth program group and are in college).

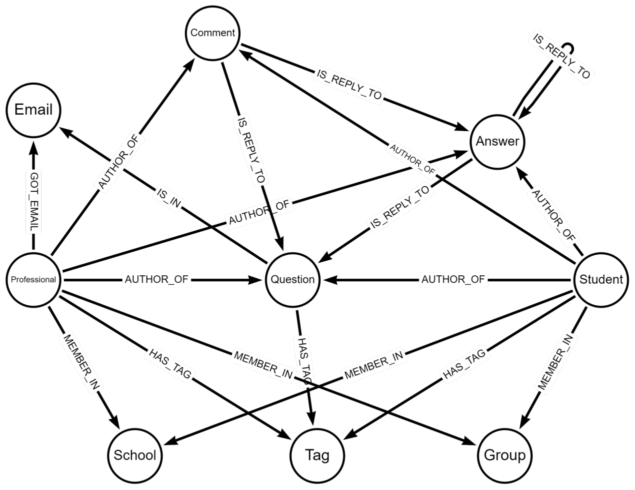


Figure 13: Dataset shown as a graph. [8]

Type of queries	Queries	Neo4j	MySQL
Selection/search	Q1	7 ms	8 ms
	Q2	9 ms	270 ms
	Q3	13 ms	39 ms
Related or recursive data	Q4	2 ms	292 ms
	Q5	3 ms	312 ms
	Q6	3 ms	314 ms
Aggregation	Q7	31 ms	118 ms
	Q8	15 ms	22 ms
	Q9	86 ms	429 ms
Pattern matching	Q10	4 ms	4 ms
	Q11	5 ms	49 ms
	Q12	1 ms	8 ms

Figure 14: Results from query execution on relational vs. graph implementation. [8]

To perform this query in SQL, four different joins are required to collect the necessary student, group, and tag information. In Cypher (the language used by Neo4j), the MATCH keyword is used with two conditions. Both implementations are fairly readable in this case, but the performance gain of the Cypher query is undeniable. Referring to Figure 14, the graph implementation took 9 ms to run this query, and the relational implementation took 270 ms. This is a **30x** improvement gained by using a graph database.

Query 4 (Figure 16) from Do et al. [8] sees the greatest performance gain between implementations (2 ms vs. 292 ms for **146x** improvement). This query is recursive and performs multiple joins to find all answer replies that relate to each question. Given how this dataset is structured, this requires delving into each answer, finding replies, tacking them on, and repeating the process until all replies have been found. In contrast, the Cypher implementation simply uses a one-line query matching answers that are a reply to each question. In this case, readability is massively improved by using Neo4j in addition to the performance gain.

SQL

```
SELECT * FROM students s
 JOIN group_memberships gm ON
   students_id = gm.group_memberships_user_id
 JOIN groups g ON
   g.groups_id = gm.group_memberships_group_id
 JOIN tag_users tu ON
   tu.tag_users_user_id = s.students_id
 JOIN tags t ON t.tags_tag_id = tu.tag_users_tag_id
WHERE t.tags_tag_name = 'college'
 AND g.groups_group_type = 'youth program';
```

Cypher

```
MATCH (t:tags)<-[:HAS_TAG]-(s:students)-
   [:MEMBER_IN]->(b)
WHERE t.tags_tag_name='college'
 AND b.groups_group_type='youth program'
RETURN s,t,b
```

Figure 15: Query 2 from Do et al. [8]. Looks for students in a specific group and with a specific tag.

SQL

```
WITH RECURSIVE answer_replies AS(
 SELECT answers_id, answers_author_id,
   answers_question_id, answers_date_added,
   answers_body
 FROM answers WHERE answers_question_id
 IS not null UNION all
 SELECT a.answers_id, a.answers_author_id,
   a.answers_question_id, a.answers_date_added,
   a.answers_body
 FROM answers a
 INNER JOIN answer_replies ar ON ar.answers_id =
   a.answers_question_id )
 SELECT * FROM answer_replies ar
 LEFT JOIN questions q ON ar.answers_question_id =
   q.questions_id;
```

Cypher

```
MATCH (q:questions)<-[:IS_REPLY_TO*1..]-(a:answer)
RETURN q,a
```

Figure 16: Query 4 from Do et al. [8]. Finds all answers (replies) that correspond to every question.

Query 9 (Figure 17) from Do et al. [8] deals with aggregation. In this case, a counting problem needs to be solved to determine which tag belongs to the most professionals. Here, only two joins are performed, and readability is quite similar between the graph and relational implementations. As for performance, both implementations are relatively slow at this task. The graph implementation takes 86 ms, and the relational implementation takes 429 ms, for a total performance gain of **4.98x**. While this is considerably less impressive than the performance gains seen in queries 2 and 4, an almost 5x improvement is still very significant for high frequency operations.

SQL

```
SELECT tags.tags_tag_id, tags_tag_name,
 COUNT(p.professionals_id)
 AS number_of_professionals FROM tags
 JOIN tag_users tu
   ON tags.tags_tag_id = tu.tag_users_tag_id
 JOIN professionals p ON p.professionals_id =
   tu.tag_users_user_id
 GROUP BY tags.tags_tag_id, tags_tag_name
 ORDER BY COUNT(p.professionals_id)
 DESC LIMIT 1;
```

Cypher

```
MATCH (p:professionals)-[:HAS_TAG]->(t:tags)
RETURN t.tags_tag_name AS TagName,
 COUNT(p) ORDER BY COUNT(p) DESC
 LIMIT 1
```

Figure 17: Query 9 from Do et al. [8]. Returns the tag that belongs to the most professionals.

Finally, Query 11 (Figure 18) from Do et al. [8] is an example of a pattern matching query. Here, it is desired to determine which students and professionals share the same group. The SQL implementation uses 4 different joins and is essentially incomprehensible, while the Cypher implementation is a simple one-liner. Here the graph implementation takes 5 ms and the relational takes 49 ms for a total improvement of **9.8x**. Again, it is demonstrated that for complex, relation-based queries, Cypher is both more efficient and much easier to develop and understand.

4.1.4 *Discussion*.

It is important to understand that, despite the outcome of this particular study, graph databases are not superior in every instance. Some datasets and query types do not benefit from the relationship-focused approach of graph databases and would be better off with a relational database. That being said, it is evident from Do et all [8] that there are certainly instances where graph databases are superior both in terms of performance and readability/development ease. For each of the queries shown in 4.1.3, the graph implementation performed magnitudes better. Besides the work of academic studies like this one, graph databases are also very commonly used in the real-world, as discussed in the case study section of the Neo4j website [4]. Widespread commercial adoption typically is a very strong indicator of the viability of different technological advances.

4.2 Cypher versus Gremlin

Florian Holzschuher and René Peinl conducted a study on the performance of graph query languages, comparing Cypher and Gremlin in the Neo4j database [10]. They cite performance, initial learning effort, code readability, and maintainability as metrics when it comes to choosing and comparing a query language.

SQL

```
SELECT g.groups_id, professionals_id, students_id
 FROM groups g
 JOIN group_memberships gm ON g.groups_id =
   gm.group_memberships_group_id
 JOIN (SELECT group_memberships_group_id
   AS group_id, professionals_id
 FROM professionals p
 JOIN group_memberships gm1
   ON gm1.group_memberships_user_id =
   p.professionals_id) pg ON pg.group_id =
   gm.group_memberships_group_id
 JOIN (SELECT group_memberships_group_id
   AS group_id, students_id
 FROM students s JOIN group_memberships gm2 ON
   s.students_id = gm2.group_memberships_user_id) sg
   ON sg.group_id= gm.group_memberships_group_id;
```

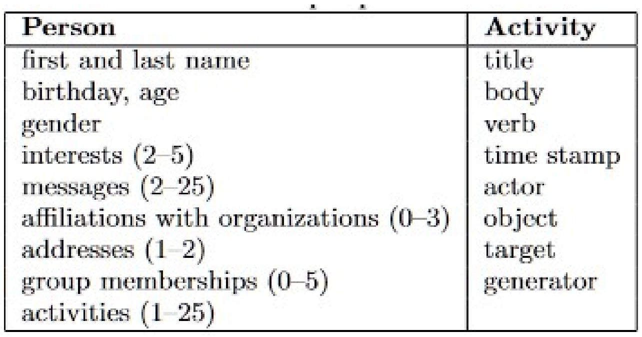
Cypher

```
MATCH (p:professionals)-[]->(g:groups)<-[]-(s:students)
RETURN p, g, s
```

Figure 18: Query 11 from Do et al. [8]. Finds students and professionals with the same group.

4.2.1 *The experiment*.

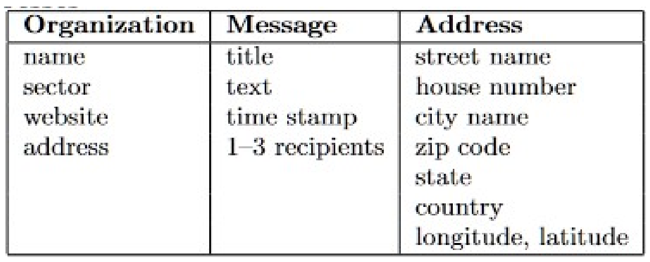
The authors use the Java-based graph database, Neo4j, for their comparison study. They emulate a social Web portal based off of OpenSocial (now deprecated). To generate the data required for their setup, the authors wrote a custom generator program that generates XML data with random people, organizations, messages and activity based data based on anonymized friendship graphs. The data associated with each field can be seen in Figure 19 and Figure 20. The sample dataset they generated contains 2011 people, 2000 addresses, 200 groups and 100 organizations the attributes for which are in the figures mentioned above. When parsed into Neo4j the set generated about 83,500 nodes and around 304,000 relationships.



Person	Activity
first and last name	title
birthday, age	body
gender	verb
interests (2–5)	time stamp
messages (2–25)	actor
affiliations with organizations (0–3)	object
addresses (1–2)	target
group memberships (0–5)	generator
activities (1–25)	

Figure 19: Data for people and activities[10]

The paper performs comparisons between Cypher, Gremlin, Cypher and Gremlin using the REST API, Neo4j native object access, and the MySQL JPA (Java Persistence API). For the scope



Organization	Message	Address
name	title	street name
sector	text	house number
website	time stamp	city name
address	1–3 recipients	zip code
		state
		country
		longitude, latitude

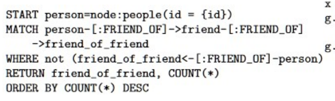
Figure 20: Data for organizations, messages and addresses[10]

of our discussion we will only focus on the Gremlin and Cypher comparison.

The Neo4j database has native support for Cypher as its main query language but it also allows for Gremlin execution. While this may introduce a bias in comparison towards Gremlin, there are still a few distinct aspects which can set the two languages apart in terms of semantics, purpose and performance.

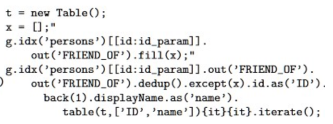
4.2.2 *Language Semantics.*

Cypher queries are basically constant strings which can act as compiled queries because they can be cached by the database. Cypher also has similarities with SQL that make it easy to comprehend. Gremlin, on the other hand, is a low-level traversal language that has a compact syntax and is more complex to read. A query to find friend suggestions by traversing the graph is shown below for Cypher (Figure 21a) and Gremlin (Figure 21b):



```
START person=node:people(id = {id})
MATCH person-[:FRIEND_OF]->friend-[:FRIEND_OF]
    ->friend_of_friend
WHERE not (friend_of_friend<-[:FRIEND_OF]-person)
RETURN friend_of_friend, COUNT(*)
ORDER BY COUNT(*) DESC
```

(a) Cypher



```
t = new Table();
x = [];"
g.idx('persons')[[id:id_param]].
    out('FRIEND_OF').fill(x);"
g.idx('persons')[[id:id_param]].out('FRIEND_OF').
    out('FRIEND_OF').dedup().except(x).id.as('ID').
      back(1).displayName.as('name').
         table(t,['ID','name']){it}{it}.iterate();
```

(b) Gremlin

Figure 21: Queries to retrieve friend suggestions [10]

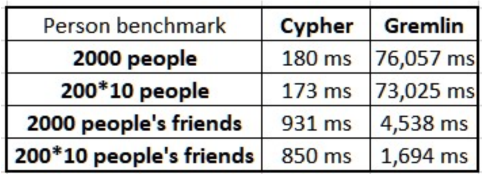
While both the queries perform the same operation, the Cypher query is much easier to interpret for a user just beginning their exploration towards graph databases and querying languages. But the Gremlin language, on the other hand, provides you with fine grain control over the query which can be something a more experienced programmer looks for.

4.2.3 *Performance.*

The authors perform an analysis by comparing the throughput for the queries mentioned above. The query response times for Cypher and Gremlin queries reported by the authors can be seen in Figure 22. The response times are averaged over 10 runs and we used that data to generate a time series graph in Figure 23.

As seen in Figure 23, Gremlin queries take longer initially as we are travelling the entire graph for the first time. Neo4j inherently supports caching, so we see a little performance improvement when we start search for friends. The throughput is consistently high for Cypher due to Neo4j's native support.

Now we try to answer why does Cypher outperform Gremlin in this experiment. For the most part, Cypher is declarative and



Person benchmark	Cypher	Gremlin
2000 people	180 ms	76,057 ms
200*10 people	173 ms	73,025 ms
2000 people's friends	931 ms	4,538 ms
200*10 people's friends	850 ms	1,694 ms

Figure 22: Query response times for friend queries in Section 4.2.2[10]

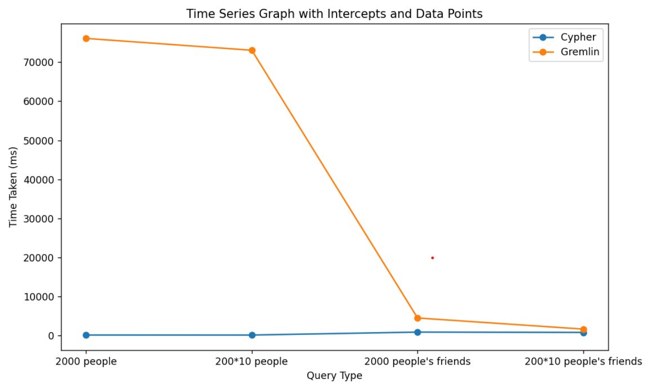


Figure 23: Cypher vs Gremlin against friend queries in Section 4.2.2

Neo4j is built to optimise Cypher queries on its own. Gremlin, on the other hand, is imperative, low-level and vendor-agnostic. Thus, query optimisations have to be handled by the programmer for the most part, which would be appropriate for an experienced and well-versed Gremlin programmer. Cypher is designed with a focus on patterns and relationships which makes it intuitive and expressive for certain graph patterns such as finding friends of friends. Gremlin, being more general purpose, might require more explicit instructions from the user and could lead to potentially less optimized and computationally-expensive queries in certain scenarios. In potentially read-heavy and traversal-only scenarios, Gremlin would be the ideal choice for an experienced programmer as it gives you fine-grained control over the query. For the scenario presented in this research, and situations where data manipulation is required alongside pattern recognition, Cypher is the ideal choice due to its simplicity, mass adoption standards and native support in Neo4j.

4.3 SPARQL

It is difficult to compare SPARQL directly with Cypher and Gremlin because SPARQL was created specifically for RDF traversals. When evaluating these three graph query languages, it is crucial to take into account the unique use cases and demands of your application. Cypher stands out as an ideal option for developers who prioritize readability and simplicity, especially when querying Neo4j databases. On the other hand, Gremlin provides greater versatility and robustness, making it well-suited for intricate graph traversals and manipulations. It also appeals to developers with prior experience in SQL and other imperative languages. As for SPARQL, it emerges as the preferred choice when dealing with RDF data and engaging in tasks related to linked data and semantic web applications.

4.3.1 *S2CTrans Pipeline.*

Currently, there is ongoing research to build bridges between SPARQL and other graph query languages. For example, Zhao et al. [15] create the S2CTrans framework, that can achieve SPARQL-to-Cypher translation while preserving the original semantics of the queries.

The S2CTrans framework, which offers a mapping method for pattern matching and solution modifiers, enabling the translation from SPARQL to Cypher. The mapping function translates SPARQL graph patterns into Cypher graph pattern elements. Essentially, the S2CTrans pipeline involves parsing the SPARQL query, transforming it into Cypher statements through a series of mapping and construction steps, and rendering it into a format compatible with Neo4j for querying property graph data. The mapping function consists of two key components: *Graph Pattern Matching Mapping* (PMM) and *Solution Modifier Mapping* (SMM).

This open-source framework follows a five-step execution pipeline:

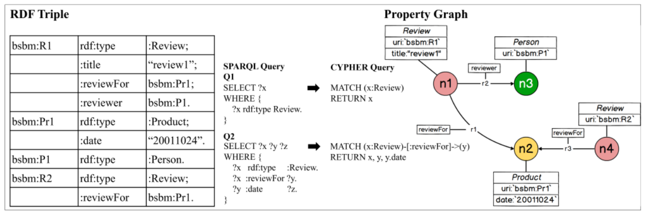
(1) **Parsing:** The input SPARQL query is parsed using the Jena ARQ module, generating an *Abstract Syntax Tree* (AST) representation.
(2) **Graph Pattern Matching:** OpWalker and PMM algorithms are used to access and map the graph pattern matching and solution modifier parts from the AST bottom-up.
(3) **Mapping:** PMM maps SPARQL graph patterns to Cypher graph patterns, and SMM maps SPARQL solution modifiers to Cypher clause keywords.
(4) **Cypher-DSL Construction:** Cypher-DSL generates the final conjunctive traversal and constructs Cypher AST based on pattern element types and operator priorities. The Cypher DSL (Domain-Specific Language) is a language extension or abstraction specifically designed for working with the Cypher query language. Essentially, it is a query language used for expressing patterns and queries on graph-structured data.
(5) **Rendering:** The Cypher AST is rendered into a complete Cypher statement using the Renderer, which can be directly queried in Neo4j.

PMM translates SPARQL graph patterns into Cypher graph pattern elements, while SMM constructs a mapping table to implement the mapping of SPARQL solution modifiers to Cypher clause keywords. Through these algorithms, S2CTrans achieves semantic equivalence between SPARQL and Cypher, enabling seamless translation and execution of queries on both types of databases.

Their experimental results show that with some benchmark datasets, S2CTrans framework converts most SELECT queries into type-safe Cypher statements. They also outline the following comparisons between the two languages in Table 1.

4.3.2 *The Gremlinator.*

Thakkar et al. [14] create the Gremlinator, which bridges the gap between SPARQL and Gremlin. The goal of Gremlinator is to enable the execution of SPARQL queries on property graph databases by translating them into Gremlin traversals. This would reduce the need to learn another query language. The Gremlinator maps SPARQL graph patterns to equivalent Gremlin graph patterns by generating an abstract representation of the query.



RDF Triple

bsbm:R1	rdf:type	:Review;
	:title	“review1”;
	:reviewFor	bsbm:Pr1;
	:reviewer	bsbm:P1.
bsbm:Pr1	rdf:type	:Product;
	:date	“20011024”.
bsbm:P1	rdf:type	:Person.
bsbm:R2	rdf:type	:Review;
	:reviewFor	bsbm:Pr1.

SPARQL Query / **CYPHER Query**

Q1
```
SELECT ?x              ➡   MATCH (x:Review)
WHERE {                    RETURN x
  ?x rdf:type Review.
}
```

Q2
```
SELECT ?x ?y ?z        ➡   MATCH (x:Review)-[:reviewFor]->(y)
WHERE {                    RETURN x, y, y.date
  ?x  rdf:type     :Review.
  ?x  :reviewFor ?y.
  ?y  :date        ?z.
}
```

Figure 24: Examples of RDF triples and corresponding property graph, SPARQL queries and corresponding Cypher queries. [15]

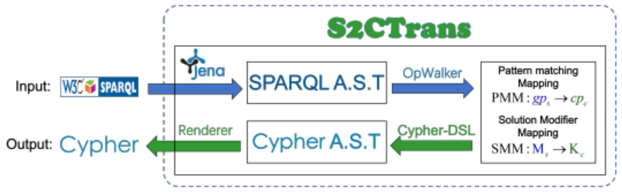


Figure 25: The S2CTrans Pipeline Architecture. [15]

The architectural framework of Gremlinator contains four key steps in its execution pipeline:

(1) The initial processing involves parsing the input SPARQL query using the Jena ARQ module. This step serves to validate the query and generate its AST representation.
(2) Gremlinator, utilizing the AST obtained from the parsed SPARQL query, traverses each *Basic Graph Pattern* (BGP). During this process, BGPs are mapped to corresponding Gremlin *Single-Step Traversals* (SSTs). In Gremlin, an SST is an atomic traversal step.
(3) Depending on the operator precedence derived from the AST of the parsed SPARQL query, Gremlinator maps each SPARQL keyword to its respective instruction steps from the Gremlin instruction library. Subsequently, a final conjunctive traversal (Ψ) is generated by appending the SSTs and instruction steps. This concept aligns with the SPARQL query language, where a set of BGPs constitutes a single complex graph pattern (CGP).
(4) The final conjunctive traversal (Ψ) is employed to generate bytecode, which is adaptable for use across various language and platform variants within the Apache TinkerPop Gremlin family [14].

5 CONCLUSION

Through an extensive literature review, we compare graph databases and graph query languages. We discuss the advent of graph databases and their industrial rise and adoption. We review the basics for Cypher, Gremlin and SPARQL – three widely-used graph query languages. Through our discussion, readers can get an idea about

Aspect	Cypher	SPARQL
Graph Query Languages	Designed for graph databases	Designed for RDF databases
Pattern Matching	Supports pattern matching	Supports pattern matching
Declarative Syntax	Uses declarative syntax	Uses declarative syntax
Querying Relationships	Provides mechanisms for querying and traversing relationships	Allows querying and traversing relationships

Table 1: Comparison of Cypher and SPARQL

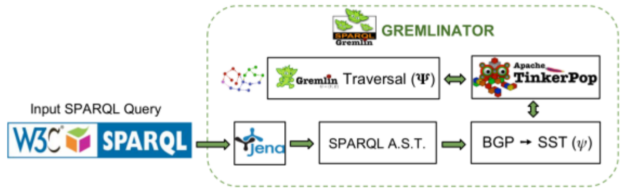


Figure 26: Architectural Overview of Gremlinator. [14]

the basic syntax and semantics of these languages through the examples presented.

We review the research conducted by multiple authors towards performing a comparison between different graph query languages. During the course of conducting this survey, we came to the conclusion that there cannot be a direct comparison between all three of the query languages featured in this paper, as they each serve a different use-case. Cypher, most prominently used in Neo4j, is used to pattern match and express relationships between the data. Its most common applications include social media analysis, recommendation systems and fraud detection by uncovering suspicious patterns. Gremlin is a vendor-agnostic low-level graph traversal language for Apache Tinkerpop frameworks and the databases that support the framework. Some common use-cases include pathfinding and routing, querying knowledge graphs and search engines as these are more read-heavy cases that require graph traversals. SPARQL is used with Apache Spark and is associated with RDF data. It can be used for data exploration, retrieval and relationship analysis.

On first glance, Cypher and SPARQL look similar to SQL when it comes to their semantics and clauses. Thus, these can be good choices for an amateur to start with graph query languages. Cypher is optimized for pattern recognition and Neo4j is the most popular and widely recognized commercial graph database making it a great candidate for most recommendation and analysis tasks. Gremlin, on the other hand, being a low-level language can be difficult to comprehend at first but allows more freedom and control when writing optimised queries. When it comes to complex traversals and scenarios requiring custom algorithms, Gremlin's flexibility makes it a good choice for a professional developer. Thus, one needs to take into account all these factors when it comes to choosing the appropriate language for their use-case as each one comes with its own merits and learning curve.

REFERENCES

[1] [n. d.]. https://www.w3.org/TR/rdf-sparql-query/

[2] [n. d.]. https://tinkerpop.apache.org/docs/current/tutorials/getting-started/

[3] [n. d.]. Clauses - Cypher Manual — neo4j.com. https://neo4j.com/docs/cypher-manual/current/clauses. [Accessed 11-12-2023].

[4] 2023. https://neo4j.com/case-studies/

[5] Waqas Ali, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo. 2021. A Survey of RDF Stores SPARQL Engines for Querying Knowledge Graphs. arXiv:2102.13027 [cs.DB]

[6] Renzo Angles. 2018. The Property Graph Database Model. In *Alberto Mendelzon Workshop on Foundations of Data Management*. https://api.semanticscholar.org/CorpusID:43977243

[7] Renzo Angles and Claudio Gutierrez. 2018. An Introduction to Graph Data Management. In *Data-Centric Systems and Applications*. Springer International Publishing. https://doi.org/10.1007/978-3-319-96193-4_1

[8] Thi-Thu-Trang Do, Thai-Bao Mai-Hoang, Van-Quyet Nguyen, and Quyet-Thang Huynh. 2022. Query-Based Performance Comparison of Graphnbsp;Database and Relational Database. In *Proceedings of the 11th International Symposium on Information and Communication Technology* (Hanoi, Vietnam) *(SoICT '22)*. Association for Computing Machinery, New York, NY, USA, 375–381. https://doi.org/10.1145/3568562.3568648

[9] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. https://doi.org/10.1145/3183713.3190657

[10] Florian Holzschuher and René Peinl. 2013. Performance of Graph Query Languages: Comparison of Cypher, Gremlin and Native Access in Neo4j. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 195–204. https://doi.org/10.1145/2457317.2457351

[11] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3, Article 16, 45 pages. https://doi.org/10.1145/1567274.1567278

[12] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. 2009. Semantics and Complexity of SPARQL. *ACM Trans. Database Syst.* 34, 3, Article 16 (sep 2009), 45 pages. https://doi.org/10.1145/1567274.1567278

[13] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. ACM. https://doi.org/10.1145/2815072.2815073

[14] Harsh Thakkar, Dharmen Punjani, Jens Lehmann, and Sören Auer. 2018. Killing Two Birds with One Stone – Querying Property Graphs using SPARQL via GREMLINATOR. arXiv:1801.09556 [cs.DB]

[15] Zihao Zhao, Xiaodong Ge, and Zhihong Shen. 2023. S2CTrans: Building a bridge from SPARQL to Cypher. arXiv:2304.00531 [cs.DB]

[16] Lei Zou. 2018. *SPARQL*. Springer New York, New York, NY, 3554–3558. https://doi.org/10.1007/978-1-4614-8265-9_80803