FernUniversität Hagen

– Faculty of Mathematics and Computer Science –

Query languages for Graph databases

Seminar „Discovering Big Data“

Chair of Databases and Information Systems

presented by

Daniel Langhann

Registration number: 3788687

Mentoring : Prof. Dr. Uta Störl

DECLARATION

I blab that I have written the seminar paper independently and without the unauthorized use of third parties.

I have only used the sources and aids indicated and have marked the passages taken from them verbatim or analogously as such.

The declaration of independent work also applies to any drawings, sketches or graphic representations contained therein.

The report has not yet been submitted in the same or a similar form to the same or any other audit authority, nor has it been published.

By submitting the electronic version of the final version of the thesis, I acknowledge that it can be checked for plagiarism using a plagiarism detection service and will be stored exclusively for examination purposes.

Hagen, June 30, 2024

Daniel Langhann

ABSTRACT

Graph databases have become increasingly popular in recent years in the field of big data processing and in the area of evaluations for large amounts of data.
At the same time, the amount of available data is continuously increasing, and with it the need for technologies that are capable of efficiently store and query this ever-increasing amount of data and make it searchable.
Graph databases are suitable for precisely this purpose. Various database technologies and query languages have been developed in the past for graph data and graph databases. At the same time, an ISO standard for querying graph databases based on the Structured Query Language (SQL) standard was developed, which was published in the process of this work under ISO/IEC 39075.
The motivation of this work is to analyze selected query languages and their underlying technologies and to make them comparable on the basis of defined criteria.
The paper begins with a general introduction to the topic of graph data and graph databases in which, among other things, important terms are clarified. The main section in chapter 4 then presents three of the most popular query languages and compares them with the previously introduced comparison criteria. The paper concludes in chapter 5 with a summary of the results of the analyses and gives a general outlook on how query languages for and the use of graph databases could develop in the future.

CONTENTS

LISTINGS

LIST OF ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
ETL	Extract, Transform, Load
GPM	Graph Pattern Matching
GQL	Graph Query Language
ISO	International Organization for Standardization
QGP	Quantified Graph Pattern
RDF	Resource Description Framework
SQL	Structured Query Language
URI	Uniform Ressource Identifier
W3C	World Wide Web Consortium

1 INTRODUCTION

1.1 RESEARCH TOPIC

Graph databases,have developed into a powerful tool for modeling and querying complex relations in various data types. They are particularly suitable for dealing with linked data and complex network structures that essentially consist of nodes and their edges (relations). Examples of use are e.g, Relationships in business networks, Recommendation applications and Biological networks [Yua22]. In this paper, the topic of graph databases and selected query languages designed for graph databases will be presented and contextualized. In particular, the path from different query languages to a uniform Graph Query Language (GQL) standard by the International Organization for Standardization (ISO) is pointed out [Har24]. Further the term GQL standard is used.

1.2 MOTIVATION

The aim of this paper is to provide an overview of selected important query languages for graph databases and to highlight the respective advantages and disadvantages, their capabilities, and limitations, as well as their specific use cases.

1.3 STATE OF RESEARCH

Graph databases are a comparatively young technology in the field of database systems [Har24]. Over time, various query languages have been developed with specific focuses and corresponding strengths and weaknesses. A common ISO standard should help to keep the technology and its application more generic [DSW24]. In this paper, important proprietary languages are presented and differentiated from each other. In the further course, an overview of the different languages is created.

1.4 APPROACH

Basically, relevant documentation on the various query languages is analyzed and the key points are highlighted. To support this, selected languages are tried out in a test project and the corresponding learnings are integrated into the work.

2 CRITERIA FOR COMPARISON

First, comparative criteria are introduced against which the different languages are compared. The evaluation is done purely qualitatively as this is a complex topic that is difficult to evaluate on the basis of a uniform scale. Nevertheless, the aim is to evaluate the languages as objectively and reproducibly as possible on the basis of the defined criteria.

2.1 EXPRESSIVENESS

Expressiveness in the context of query languages for graph databases means the ability of a language to allow a wide range of queries on graphs and their underlying databases. An expressive language can be used with complex patterns and relationships even across various different graphs. In addition, an expressive language offers users the corresponding flexibility when querying graph structures [Bar+12].

2.2 PERFORMANCE AND PERFORMANCE OPTIMIZATION

Performance is a decisive factor when dealing with the analysis of large amounts of data. Especially when it comes to evaluations that need to be carried out in real time or very quickly [Yua22]. A recommendation system does not help anyone if it does not deliver an answer in a reasonable time [MBM20]. This criterion evaluates in particular the ability of the corresponding language to enable techniques that improve the performance of queries on graph databases [Yua22].

2.3 INTEROPERABILITY

How well a query language intergrates with other tools, techniques and other API's (Application Programming Interface (API)) is an important criteria for the decision for and against the query language and often for the entire database system [TRRMT21].

2.4 CLOSENESS TO THE GQL STANDARD

Attention is also paid to how closely each language aligns with the GQL standard. A language that closely follows this standard, or whose model and structure incorporate a significant portion of it, ensures that users need to acquire minimal proprietary knowledge, thereby enhancing user-friendliness. Moreover, strong alignment with the GQL standard promotes interoperabil-

ity with other technologies and systems, as these are likely based on at least the fundamental concepts of GQL [Har24]. The next chapter introduces the GQL standard, allowing the reader to compare various languages with it when reading chapter 4.

2.5 BENCHMARK QUERIE

The benachmark querie to compare the capabilities of the different languages can be found in listing 1.

3

THE GQL STANDARD

3.1 OVERVIEW OF THE GQL STANDARD

In short, GQL is a new standard for a property graph database language developed by the ISO committee. It is the first time in more than 35 years, that the ISO released a new standard for database query language [Har24]. In opposite to SQL, where data is organized in tables, graph databases structure data in graphs . This enables new ways to analyze and recognize patterns in very large amounts of data without having specific knowledge about the data itself [Pin+24]. Various use cases for graph databases have already been mentioned in chapter 1.1, one other interesting example can be found in [Saa+23].

3.2 SCOPE OF THE GQL STANDARD

Graph databases store and retrieve nodes (vertexes) and edges between nodes (relationships). GQL is a delclarative language influenced both by existing property graph database products and by the SQL standard. The GQL standard is a complete database language that supports, creating, reading updating, deleting and modifying property graph data [Har24].
Graph data can be organized in two different ways, Schema free or constrained by an database administrator. Schema free graph data has no restrictions for adding and changing graph data. Graph data that is subject to a predefined schema must fulfill this at all times, otherwise an error is triggered [Wih+20].
The schema of graph data is specified via so-called graph types, which on the one hand specify the structure of the nodes as well as that of the edges, i.e. the relationships between the nodes [Har24].

3.3 EXAMPLES OF COMMON GQL QUERY OPERATIONS

This chapter provides a brief introduction to GQL capabilities including, Queries and Graph Pattern Matching (GPM), Add, modify and delete operations.

3.3.1 *Queries and Graph Pattern Matching*

GQL queries are based on rich GPM language. The below example findes all nodes with a one-hop relationship to a node with a with a specific *productId* [Har24]:

Listing 3.1: Example for one hop graph pattern query

```
{
   MATCH (a {productId: "6594301c4aa9b3232889e7c3"}) - [b] -> (c)
       RETURN a, b, c

}
```

The GQL standard does not define how the returned data has to be displayed to the user. Two possible ways can be, Graph Data Visualization tools and Textoutput.

3.3.2 *Quantified graph patterns*

The GPM of GQL also provides capabilities for Quantified Graph Pattern (QGP) [FWX16]. The following example finds all paths where one node is related to another node up to ten hops:

Listing 3.2: Example for quantified graph pattern

```
MATCH((a)-[r]->(b)){1,10}
    RETURN a, r, b
```

More complex QGP are possible.

3.3.3 *Complex Graph pattern*

The listing 1 matches pattern with nodes of type *Person* and *Company*. The relationships between the nodes are *worksFor* and *partnerOf*. The matching pattern:

```
MATCH (person:Person)-[r1:worksFor]->(company:Company)-[r2:partnerOf
    ]->(partner:Company)
```

matches a pattern where a *Person* works for a *Company* and that *Company* has a partnership with another *Company*. The filter of the query:

```
    WHERE person.age > 25 AND company.revenue > 100000
```

filters all persons with an age > 25 and companys with a revenue > 100000. The part or WITH-Clause

```
    WITH person, company, partner, COUNT(r1) AS work_relationships,
        COLLECT(r2) AS partneRelationships
```

aggregates the number of *worksFor* relationships and collects the *partnerOf* relationships for further use in the query in the second match pattern:

```
    MATCH (person)-[r3:knows]->(colleague:Person)
    WHERE colleague.age < 30
```

3.3.4 *Insert, Update and Delete operations*

The following examplein listing 3.3 shows a typical insert statement in GQL:

Listing 3.3: Typical insert statement in GQL

```
/*Insert one node */
    INSERT (:Customer {name: 'XYZ Ltd.', customerStatus: 'Active'})
```

In this example *Customer* is a Label and *name* and *customerStatus* are properties.
Labels are identifiers that are either present or not present. Properties are Key-Value pairs.
Both nodes and relationships can have labels and properties [Pin+24]. Basically, nodes are enclosed in parenthesis:

```
(:Person {firstName: 'Daniel', lastName: 'Langhann'})
```

and relationships (edges) are enclosed in square brackets:

```
[:r3:knows]
```

GQL supports insert operations for complex graph pattern like listing 2. The statement in listing 2 inserts two nodes *Customer* and *City* and the relationship *located*.
Insert operations as shown in Listing 3.4 can also be the result of a *MATCH* pattern:

Listing 3.4: Pattern based insert statement

```
/*Creates an edge isStudentOf. */
MATCH (a {name: 'Langhann'}), (b {name: 'Stoerl'})
INSERT (a)-[:isStudentOf]->(b)
```

In the Listing 3.4 the variables *a* and *b* are aliases.
They are defined in the *MATCH* clause and are only part of the Memory until the Query determines.
The result of the *MATCH* caluse in listing 3.4 is a **cartesian product** of the two nodes. This is why each node expression returns only one node. The *INSERT* statement will only insert one edge.
An Update in GQL is done by identififying the nodes or edges to be updated. After identififying the instances the user can set or remove properties (listing 3.5):

Listing 3.5: Update statements in GQL

```
/*Update*/
MATCH (d:Customer) where d.id = '6594301c4aa9b3232889e7c3'
SET d.status='inactive'
```

The following listing 3.6 removes the properties of a node *Customer*:

Listing 3.6: Delete propteries in GQL

```
/*Remove propertiers*/
```

```
MATCH (d:Customer) where d.id = '6594301c4aa9b3232889e7c3'
REMOVE d.invoiceAccepted
```

In listing 3.7 a node and the related nodes where deleted:

Listing 3.7: Remove nodes in GQL cascading

```
/*Delete Customer and related Nodes*/
MATCH  (a {id: '6594301c4aa9b3232889e7c3'})-[b]->(c)
DETACH DELETE a,c
```

3.4 TRANSACTIONS

In GQL serializable transactions and their additional implmentation-defined transaction modes are supported. Transactions are starting with either an **explicit** or **implicit** *START TRANSACTION* statement and terminated with either a *COMMIT* or *ROLLBACK*. The user can also implement automatic transactions starts and commits [Har24].

3.5 SCHEMA FREE VS GRAPH TYPES

A schema free graph accepts any form of graph data which makes it relatively fast to use. But on the other hand it gives the user control over the data and therefore a certain amount of data proliferation is accepted and has to be managed
A graph type is a kind of template that specifies the structure of the graph, which must be adhered to at all times. The structure of the nodes as well as the edges or both can be specified [CR24]. The listing 3 illustrates the concept of a graph type. If the user is creating a graph by using a graph type, the contents of the nodes and edges are constrained by the graph type, which makes sure the data model will be respected in all transactions. This is also relevant for update operations. If an update operation does not follow the restrictions of the graph type, the transactin will be rolled back and the database system will throw an error [CR24].

3.6 COMPARE GRAPH STRUCTURED DATA AND TABLES

In a typical SQL database, data is organized and stored in tables and rows where each table has a fixed schema. Relationships between tables are basically defined as so called Foreign Keys [CG85]. The user of the database has to know and understand these relationships to be able to write queries over more than one table.
In a property graph database, the level of abstraction is high, and allows the user whole sets of tables to treat as one unit, which can be understand as a data product [Pin+24].

4

DIFFERENT QUERY LANGUAGES FOR GRAPH DATABASES

After clarifiying important terms and concepts of graph databases and a short introduction of the GQL-standard in chapter 3, now, specific database systems, their concepts and syntax will be introduced and compared against the criteria definded in chapter 2.

4.1 OPENCYPHER

openCypher [Noab] is an open Source Framework which is the basis for Cyper the Query language developed by Neo4J [Noaa]. The benchmark querie in openCypher can be implemented as in the following listing 4.1:

Listing 4.1: Benchmark Query in openCypher

```
MATCH (person:Person)-[r1:worksFor]->(company:Company)-[r2:
    partnerOf]->(partner:Company)
WHERE person.age > 25 AND company.revenue > 1000000
WITH person, company, partner, COUNT(r1) AS workRelationships,
    COLLECT(r2) AS partnerRelationships
MATCH (person)-[r3:KNOWS]->(colleague:Person)
WHERE colleague.age < 30
RETURN person.name AS personName, company.name AS companyName,
    partner.name AS partnerName,
        workRelationships, partnerRelationships, COLLECT(
            colleague.name) AS youngColleagues
ORDER BY person.name, company.name
LIMIT 100
```

Obviously, the syntax is almost identical to the syntax of the benchmark query in listing 1. The only difference is that openCypher is case sensitive. Next, the four evaluation criteria will be applied to openCypher.

4.1.1 *Expressiveness of openCypher*

The first criteria is expressiveness or the ability of openCypher to allow a wide range of queries on graph data as described in chapter 2.1. openCypher suppports intuitive pattern matching with a concise syntax that enables the user to describe complex graph structures easily which the following listing 4.2 shows:

Listing 4.2: Graph Pattern Matching in openCypher

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

The user can search all nodes *a* related to node *b* by relationship *r* like in listing 3.1.
The listing 4.3 shows how the user can manipulate data in openCypher:

Listing 4.3: Graph Pattern Matching in openCypher

```
MATCH (c {customerStatus: 'Inactice'})
SET c.invoiceAccepted = 'False'
RETURN a
```

Also, Aggregation and Functions are supported in openCypher as shown in the apendix in listing 4.

4.1.2 *Performance and performance optimization in openCypher*

The complexity of a query, including the number of nodes, the relationships and the depth of the pattern to match has huge impact of the query performance. Also, the size of the data which has to be processed has an impact of the query performance. Indexing is one way to optimize the performance of a query and is an important concept to enable fast queries on large datasets [DRSG15]. The following example in listing 4.4 shows the creation of an index in openCypher [Noab]:

Listing 4.4: Index creation in openCypher

```
CREATE INDEX ON :Customer(id)
```

The example in listing 4.4 creates an Index on the property *id* of the node *Customer*. The user can also run

```
CALL db.indexes()
```

to review the current indexes and optimize them if needed [Noab].
In openCypher the user can analyze how the query works by using the keywords *EXPLAIN* (listing 4.5) and *PROFILE* to optimize the query for example by limiting the scope (listing 4.6):

Listing 4.5: Using EXPLAIN or PROFILE in openCypher

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

Listing 4.6: Set a Limit in openCypher

```
MATCH  (a: Company {name: "ABC GmbH"})-[r:PARTNER_OF]->(b:
    Company {name: XYZ AG"})
RETURN a,b LIMIT 10
```

The user can also use the *UNWIND* clause to organize data operations in batches [Noab]:

Listing 4.7: Batch operations in openCypher using UNWIND

```
UNWIND  [{name: 'ABC Inc.', name: 'XYZ Ltd.'}] AS customer
CREATE (c:Customer {name: customer.name})
```

4.1.3 *Interoperability of openCypher*

Many vendors for Graph Databases using openCypher for example, Neo4J [Noaa], Memgraph [Lop+23] and SAP HANA [Bac+22]. Neo4J for example provides an API to openCypher (or Cypher), so the user can interact with the Graph Databases within the application source code, which it makes easy to integrate openCypher. The example in listing 6 shows the integration in an example Python application [Noaa]. The fact that large database providers use openCypher as a basis ensures a high density of interfaces to other databases and technologies, as these providers have an interest in distributing or selling the technology. In addition, there is a large community of people who use the technology and provide support.

4.1.4 *Closeness to the GQL standard of openCypher*

The previous chapters have already pointed out how closely openCypher and the GQL standard are linked. The syntax is almost identical as shown in listing 1 (benchmark) compared to listing 4.1 (benchmark in openCypher). Differences in the syntax and capabilities exist, but are not decisively relevant and it can be assumed that openCypher and the GQL standard will be further harmonized. It therefore seems appropriate at this point to highlight the differences between openCypher and the GQL standard rather than the similarities:

- GQL uses the keyword INSERT where openCypher uses CREATE
- FOR statement in GQL is equivalent to UNWIND in openCypher
- MERGE, FOREACH and LOAD CSV are available in openCypher but not in GQL
- openCypher is case sensitive and GQL not

4.2 GREMLIN

Gremlin is query and traversal language for graph databases, developed as part of Apache TinkerPop [SF24c]. It supports a standardised way to interact with graph data by enabling users to traverse, query and manipulate, nodes, edges and properties of graph data. Gremlin is not tied to a specific graph database and provides a high level abstraction layer. This means that not only TinkerPop-enabled graph systems can execute Gremlin traversals, but also, every Gremlin traversal can be evaluated as either a real-time database query or as a batch analytics query.

4.2.1 *Gremlin - a traversal language*

Traversals are sequences of steps that navigate through a graph. Theses steps can include operations like, Filtering, Mapping and Reducing. The following

example points out the concept based on the benchmark query definded in listing 1:

Listing 4.8: Benchmark Query in Gremlin

```
g.V().hasLabel('Person')
.has('age', gt(25))
.as('person')
.outE('worksFor').as('r1')
.inV().hasLabel('Company')
.has('revenue', gt(1000000))
.as('company')
.outE('partnerOf').as('r2')
.inV().hasLabel('Company')
.as('partner')
.select('person', 'company', 'partner', 'r1', 'r2')
.group()
        .by(select('person', 'company', 'partner'))
        .by(
        project('workRelationships', 'partnerRelationships', '
            youngColleagues')
                .by(__.select('r1').count())
                .by(__.select('r2').fold())
                .by(
                __.select('person')
                        .out('KNOWS').hasLabel('Person').has('
                            age', lt(30))
                        .values('name').fold()
                )
        )
.unfold()
.order()
        .by(select(keys).by('person').by('name'), asc)
        .by(select(keys).by('company').by('name'), asc)
.limit(100)
.project('personName', 'companyName', 'partnerName', '
    workRelationships', 'partnerRelationships', 'youngColleagues
    ')
        .by(select(keys).by('person').by('name'))
        .by(select(keys).by('company').by('name'))
        .by(select(keys).by('partner').by('name'))
        .by(select(values).by('workRelationships'))
        .by(select(values).by('partnerRelationships'))
        .by(select(values).by('youngColleagues'))
```

It can be noted that the Gremlin concept differs significantly from the GQL standard and requires almost twice as many characters (listing 4.8) as the benchmark query in listing 1 to retrieve the same level of information. The concept of GQL follows the principles of SQL while Gremlin follows the principles of imperative programming.

4.2.2 *Expressiveness of Gremlin*

A simple GPM as described in Chapter 3.3.1 can be implemented as in listing 4.9:

Listing 4.9: Graph Pattern Matching in Gremlin

```
g.V().match(
        __.as('a').out().as('b')
).select('a', 'b')
```

g.V() starts with all vertices in the graph.[1]

```
.match(
        __.as('a').out().as('b')
```

finds all pattern where there is an outgoing edge from vertex *a* to vertex *b*. *select('a', 'b')* selects and returns the matched vertices. Next, the data manipulation in Gremlin will demonstrated based on TODO reference

Listing 4.10: Updating Vertex Properties in Gremlin

```
g.V().has('customerStatus', 'Inactive').property('
    invoiceAccepted', 'False').valueMap()
```

First, it starts to find all vertices in the graph and filters all with a match with

```
.has('customerStatus', 'Inactive')
```

After matching the property, *invoiceAccepted* will be set to *False*. The *.valueMap* returns the properties of the updated vertices. Next, the criteria Aggregations and Functions will be implemented in Gremlin based on listing 7.
The part

```
g.V().hasLabel('Company').as('c')
```

selects all vertices with the label *Company* and set an alias *c*. The part

```
.out('PARTNER_OF').hasLabel('Company').as('p')
```

The traverse is using the outgoing edge *PARTNER_OF* to other vertices with the Label *Company* and set an alias *p*.
With the expression

```
.group()
    .by('c')
```

the results will grouped by the original company vertices. The calculation part:

```
.by(
      fold().coalesce(
        unfold().values('revenue').mean().as('
            averagePartnerRevenue'),
```

1 In Gremlin the term vertice is used instead of node but it means almost the same.

```
        constant(0)
      ).as('averagePartnerRevenue')
      .count().as('partnersCount')
    )
```

calculates for each group the average revenue and the count of the partners. The rest of the query in listing 7 is to flatten the grouped results and organize the output.
The query provided in listing 8 shows how Expressive Filtering works in Gremlin. The query does the same as in lsiting 5.
The part

```
.project('cName', 'pName', 'pRevenue')
   .by(inV().values('name'))
   .by(values('name'))
   .by(values('revenue'))
```

projects the desired properties into a result set.

4.2.3 *Interoperability of Gremlin*

Gremlin is designed to be **language agnostic**, meaning it is compatible and can be used with other programming languages. Important supported languages are Python, Java and JavaScript [SF24c]. This allows the developer to use Gremlin within the preferred development environment. As already mentioned in chapter 4.2 Gremlin operates independetly of the corresponding database system. It can be used with every database compatible with the TinkerPop stack, but huge vendors like also provide plug in's to Gremlin including Neo4J [Noaa], Amazon Web Services (AWS) [Inc] and Microsoft Azure [SWP24]. In addition, there are seamless integrations to big data systems such as, Apache Hadoop [SF24a] and Apache Spark [SF24b]. The example in listing 9 implements the execution of a Gremlin querie with a python client and can be compared against listing 6 which does the same in openCypher.

4.2.4 *Performance and performance optimization in Gremlin*

Due to the fact, that Gremlin provides an indenpendent high level API the performance optimization steps directly related to the underlying database has to be done on the database itself and is not directly part of Gremlin. For example it is possible to create an Index via openCypher as shown in listing 4.4 and then use Gremlin to actually run the query. The following rules should be considerd to optimize the performance of Gremlin query [Xu+22]:

- Filtering has to be applied as early as possible to reduce the query scope.
- If possible using limit().

- Utilize local traversals (e.g. select(), local()) to operate on specific parts of the graph.

In listing 11 the "Apply filter early" strategy is implemented. The user can also limit the scope of a querie (listing 10) and enable the concepts of *PROFILE* and *EXPLAIN* by just setting *.profile* or *.explain* at the end of the query to get more detailed information about the execution plan with and without actually running the query.

4.2.5 *Closeness to the GQL standard of Gremlin*

Due to the fact that the query paradigm of Gremlin is **imperative** and the query paradigm of the GQL standard is **declarative** it can be concluded that Gremlin is comparatively far away from the GQL standard. Because the imperative paradigm forces the user to define **how** to retrieve the data and not as the GQL standard defines **what** data has be retrieved or processed [SF24c]. The syntax of Gremlin is procedural like a programming language and the GQL standard is inspired by SQL [Har24].
But, users who are familiar with the concepts of querying non relational databases like MongoDB will recognize the concepts of Gremlin [Cab+23].
In Addition, important Big Data technologies like Apache Hadoop [SF24a] are seamless interacting with Gremlin which should ensure broad acceptance in the field of Big Data, regardless of the integration of Gremlin into important programming languages such as Python (listing 9).

4.3 SPARQL

SPARQL is a graph-based query language for querying content from a Resource Description Framework (RDF) system, which is used in databases to formulate logical statements about any object and is provided by the World Wide Web Consortium (W3C) [W3C24].

4.3.1 *RDF Ressource Description Framework*

The RDF is a standard model for the exchange and representation of information on the web. It was developed by the W3C and is a central component of the **Semantic Web**. RDF is a simple data structure consisiting of a triple of Subjects, Predicates and Objects. Any Ressource has an identifier the Uniform Ressource Identifier (URI) [BLHL23].
SPARQL can be used to express queries across different data sources, regardless of whether the data is stored natively as RDF or displayed as RDF via middleware. SPARQL includes functions for querying required and optional graph patterns and their AND (conjunction) and OR (disjunction) operations. SPARQL also supports testing expandable values and restricting queries [W3C24].

Below, in listing 4.11 the Benchmark query (listing 1) is performed by using SPARQL:

Listing 4.11: Benchmark query in SPARQL

```
PREFIX ex: <http://examplehost.org/>
SELECT ?personName ?companyName ?partnerName (COUNT(?r1) AS ?
    workRelationships) (GROUP_CONCAT(?r2; separator=",") AS ?
    partnerRelationships) (GROUP_CONCAT(?colleagueName;
    separator=",") AS ?youngColleagues)
WHERE {
?person a ex:Person ;
                ex:worksFor ?company ;
                ex:name ?personName ;
                ex:age ?personAge .
FILTER (?personAge > 25)

?company a ex:Company ;
                ex:partnerOf ?partner ;
                ex:name ?companyName ;
                ex:revenue ?companyRevenue .
FILTER (?companyRevenue > 1000000)

?partner a ex:Company ;
                ex:name ?partnerName .

?person ex:KNOWS ?colleague .
?colleague a ex:Person ;
                        ex:age ?colleagueAge ;
                        ex:name ?colleagueName .
FILTER (?colleagueAge < 30)
}
GROUP BY ?personName ?companyName ?partnerName
ORDER BY ?personName ?companyName
LIMIT 100
```

The *PREFIX* declaration defines the (fictive) base URI. The *SELECT* clause specifies the variables to be returned and the *WHERE* clause contains the tripple patterns and filter that match the definded relationships and constraints. The *GROUP BY* clause is doing almost the same as the *WITCH* Clause in the Benchmark query. *ORDER BY* and *LIMIT* sort and limit the result. The results of SPARQL queries can be a result of sets or RDF diagrams [W3C24].

4.3.2 *Expressiveness of SPARQL*

SPARQL support GPM (cf. chapter 3.3.1) through triple pattern, which are conceptually similar to to the pattern matching in property graph query languages, but the syntax is different (cf. listing 3.1 and 17). The query in listing 17 matches any triple where *?a* is connected to *?b* by any predicate *?r*. Instead of setting properties on nodes, the user add or modify triples [W3C24].

The example in listing 12 creates two *Customer* Objects. On the created Objects an update querie can be performed as shown in listing 13.
Examples for Filtering and Aggregation can be found in listing 14 and 15.

4.3.3 *Interoperability of SPARQL*

For SPARQL are libraries for important programming languages available, like for Python, Java and JavaScript (cf. listing 16), which ensures a strong interoperability between SparQL and other systems and APIs. Also Integrations for Big Data- and Extract, Transform, Load (ETL) tools like Apache Hadoop [SF24a] and Apache Spark [SF24b] are available for SPARQL [NCA16].

4.3.4 *Performance and performance optimization in SPARQL*

As in other Query languages, general rules should be applied on SPARQL Queries for example the already mentioned filter early strategy (cf. chapter 4.2.4) which is crucial when the query is processed on a large dataset. It is also possible to reduce and paginate the scope of a query by using *LIMIT* and *OFFSET* clauses. In addition it is also helpful, when the related RDF Store provides indexing. As in Gremlin, the tool set for performance optimization depends on the underlying database system (cf. chapter 4.2.4).

4.3.5 *Closeness to the GQL standard of SPARQL*

SPARQL, ideal for RDF and the Semantic Web, offers robust querying capabilities for Linked Data and Ontologies. GQL, on the other hand, is designed for property graphs, intuitive for pattern matching and graph data algorithms, such as those used in Social Networks and Recommendation Systems. Both languages are declarative, allowing users to specify the needed data without specifying the related execution methods. Both languages are SQL-oriented, providing concepts familiar to SQL users. While concepts are similar, the underlying data models differ: GQL maps graph data more precisely, while SPARQL represents graph data as triples, specifically adapted to its use cases [Har24] [W3C24].

5 CONCLUSION

The work now concludes with a summary of the results in chapter 5.1 and an outlook in chapter 5.2 to what extent query languages for graph databases and graph databases in general could develop.

5.1 EXECUTIVE SUMMARY

In chapter 4, openCypher was introduced as the first language (cf. chapter 4.1). Cypher, as derivat of openCypher is devveloped and distributed by Neo4J as part of a database management system for graph databases [Noaa]. Other major providers such as AWS also offer graph databases that can be queried or operated using the syntax of openCypher [AWS24]. Essential parts of openCypher have made it into the GQL standard. The syntax is, with a few exceptions is identical between the openCypher and the GQL standard (cf. chapter 4.1.4). Since it is to be expected that, as in the case of SQL, the query languages for graph databases will continue to converge with the standard, it seems worthwhile to rely on technologies that come very close to the GQL standard or map large parts of it.
On the other hand, concepts based on imperative programming and no sql concepts are widely used for various reasons for big data processing and ETL, which argues for the use of technologies that implement these concepts (cf. chapter 4.2).
SparQL is designed for the specific purpose of queries related to the Semantic Web and thus fulfills a different use case or is not designed for the data model of property graph data but, as has been shown, can be applied to such data (cf. chapter 4.3).

5.1.1 *Expressiveness of the presented languages*

All the languages presented reflect the benchmark for expressiveness described in chapter 2.1 and can be described as equivalent or fulfilled in relation to this benchmark.

5.1.2 *Options for optimizing the performance of the presented languages*

openCypher offers direct possibilities to optimize the performance of the underlying database as the language is to be understood as part of a database system (cf. chapter 4.1.2). Gremlin and SPARQL only offer options for evaluating the performance of queries and improve these by e.g. switching to filter-first strategies (cf. chapter 4.2.4 and 4.3.4).

5.1.3 *Interoperability of the presented languages*

All languages also offer interoperability with important systems, and programming languages such as Python and Java (cf. chapter 4.1.3, 4.2.3 and 4.3.3). It is noticeable that Gremlin, as part of the Apache Foundation, is specifically connected to important systems in the context of big data and ETL, which are also provided by the Apache Foundation such as Apache Spark and Apache Hadoop (cf. 4.2.3).

5.1.4 *Closeness to the GQL standard of the presented languages*

In general, only openCypher (cf. chapter 4.1) and Gremlin (cf. chapter 4.2) are concurrent here, since SPARQL does not map the data model of propertiers graph data. openCypher and the GQL standard are very similar (cf. chapter 4.1.4). So if there is a need to stick to the GQL standard as close as possible when choosing the query language for a graph database, the choice should currently fall directly on openCypher or a derivative of it [Noab] [Har24].

5.2 FINAL REVIEW AND OUTLOOK

In general, all the presented languages are designed to search and process large volumes of graph data or graph-like structures. openCypher seems to be the language to use in the context of pure graph data, that is exactly tailored to this use case. The close alignment with the GQL standard ensures that the concepts and syntax of openCypher will continue to be relevant in the future. The seamless integration of Gremlin into the previously mentioned big data technologies also makes this language appear sufficiently generic, even if it is only slightly compatible with the industry standard GQL. SparQL follows a specific use case and a dedicated data structure. The relevance of SPARQL is strongly related to the relevance of the Semantic Web. In the context of pure graph databases, it can be assumed that the concepts of openCypher will continue to gain relevance and that the providers of commercial products in this sector will continue to drive standardization in the direction already taken.
In general the future of graph databases seems to be promising due to their ability to handle complex and interconnected data relationships efficiently. As businesses collect more data, graph databases will be crucial for managing data complexity [BF17]. They also enable real-time analytics [Shi+22], and are being integrated with other technologies such as machine learning and artificial Intelligence [Pat+24]. Graph databases are designed to scale horizontaly [Sun24] and are optimized for performance, making them suitable for handling large amounts of data and high traffic volumes. The increased use of large language models is likely to further increase the demand for graph databases [Kum+24].

.1 GQL QUERIES

Listing 1: Example for a complex quantified graph pattern (Benchmark)

```
MATCH (person:Person)-[r1:worksFor]->(company:Company)-[r2:partnerOf
    ]->(partner:Company)
    WHERE person.age > 25 AND company.revenue > 1000000
    WITH person, company, partner, COUNT(r1) AS workRelationships,
        COLLECT(r2) AS partnerRelationships
    MATCH (person)-[r3:KNOWS]->(colleague:Person)
    WHERE colleague.age < 30
    RETURN person.name AS personName, company.name AS companyName,
        partner.name AS partnerName,
            workRelationships, partnerRelationships, COLLECT(
                colleague.name) AS youngColleagues
    ORDER BY person.name, company.name
    LIMIT 100
```

Listing 2: Complex insert statement

```
/*Insert two nodes and one edge */
    INSERT (:Customer {name: 'XYZ Ltd.',
                                    customerStatus: 'Active',
                                    customerSince: date
                                        ("2024-05-19")})
    - [:located {since: date('2024-01-01')}] ->
    (:City {name: 'Bremen',
                    state: 'Bremen',
                    country: 'Germany'})
```

Listing 3: Concept of a graph type

```
{
   CREATE GRAPH TYPE /folder
   (client: Client => {cid::STRING, cname::STRING}),
   (agent:Agent => {no::STRING, office::STRING}),
   (client)-[:SUPERVISED_BY]->(agent)

}
```

.2 QUERIES OF OPENCYPHER

Listing 4: Aggregation and Functions in openCypher

```
    MATCH (c:Company)-[:PARTNER_OF]->(p:Company)
    RETURN c.name, COUNT(p) AS partnersCount, AVG(p.revenue) AS
        averagePartnerRevenue
```

Listing 5: Expressive Filtering in openCypher with Company Nodes

```
MATCH (c:Company)-[:PARTNER_OF]->(p:Company)
WHERE c.industry = 'Tech' AND p.revenue > 1000000
RETURN c.name, p.name, p.revenue
```

Listing 6: Interact with Cypher via Python

```
from neo4j import GraphDatabase, RoutingControl
URI = "neo4j://<host>:7687"
AUTH = ("neo4j", "password")

def add_customer(driver, customerName, locationName):
        driver.execute_query(
                "MERGE (a:Customer {name: $customerName}) "
                "MERGE (location:City {name: $locationName}) "
                "MERGE (a)-[:located]->(location)",
                customerName=customerName, locationName=
                    locationName, database_="neo4j",
        )


def print_customer(driver, customerStatus):
        records, _, _ = driver.execute_query(
                "MATCH (a:Customer)-[:LOCATED]->(location) WHERE
                     a.customerStatus = $customerStatus"
                "RETURN location.name ORDER BY location.name",
                name=name, database_="neo4j", routing_=
                    RoutingControl.READ,
        )
        for record in records:
                print(record["friend.name"])


with GraphDatabase.driver(URI, auth=AUTH) as driver:
        add_customer(driver, "XYZ GmbH", "Bremen")
        print_customer(driver, "active")
```

.3 QUERIES OF GREMLIN

Listing 7: Aggregation and Functions in Gremlin with Company Nodes

```
g.V().hasLabel('Company').as('c')
  .out('PARTNER_OF').hasLabel('Company').as('p')
  .group()
    .by('c')
    .by(
      fold().coalesce(
        unfold().values('revenue').mean().as('
            averagePartnerRevenue'),
        constant(0)
      ).as('averagePartnerRevenue')
      .count().as('partnersCount')
```

```
    )
  .select(keys, values)
  .unfold()
  .project('c.name', 'partnersCount', 'averagePartnerRevenue')
    .by(select(keys).values('name'))
    .by(select(values).select('partnersCount'))
    .by(select(values).select('averagePartnerRevenue'))
```

Listing 8: Expressive Filtering in Gremlin with Company Nodes

```
g.V().hasLabel('Company').has('industry', 'Tech')
 .out('PARTNER_OF').hasLabel('Company').has('revenue', gt
     (1000000))
 .project('cName', 'pName', 'pRevenue')
   .by(inV().values('name'))
   .by(values('name'))
   .by(values('revenue'))
```

Listing 9: Interact with Gremlin via Python

```
from gremlin_python.structure.graph import Graph
from gremlin_python.process.graph_traversal import __
from gremlin_python.driver.driver_remote_connection import
    DriverRemoteConnection

URI = "ws://<host>:8182/gremlin"
AUTH = ("username", "password")

def add_customer(g, customerName, locationName):
        g.V().hasLabel('Customer').has('name', customerName).
            fold().coalesce(
                __.unfold(),
                __.addV('Customer').property('name',
                    customerName)
        ).as_('customer').V().hasLabel('City').has('name',
            locationName).fold().coalesce(
                __.unfold(),
                __.addV('City').property('name', locationName)
        ).as_('location').addE('located').from_('customer').to('
            location').iterate()

def print_customer(g, customerStatus):
        customers = g.V().hasLabel('Customer').has('status',
            customerStatus).out('located').values('name').toList
            ()
        for customer in customers:
                print(customer)

graph = Graph()
connection = DriverRemoteConnection(URI, 'g')
g = graph.traversal().withRemote(connection)

try:
```

```
        add_customer(g, "XYZ GmbH", "Bremen")
        print_customer(g, "active")
finally:
        connection.close()
```

Listing 10: Set Limits in Gremlin

```
g.V().hasLabel('Company').has('name', 'ABC GmbH')
        .outE('PARTNER_OF')
        .inV().hasLabel('Company').has('name', 'XYZ AG')
        .limit(10)
```

Listing 11: Apply filter early strategy in Gremlin

```
// No alligned with filter early strategy
g.V().out('partner_of').has('revenue', gt(100000)).values('name
    ')

// Alligned  filter early strategy
g.V().has('revenue', gt(100000)).out('partner_of').values('name
    ')
```

.4 QUERIES OF SPARQL

Listing 12: Create Customer Objects in SPARQL

```
@prefix ex: <http://examplehost.org/> .

ex:Customer1 a ex:Customer ;
        ex:customerStatus "Inactive" ;
        ex:invoiceAccepted "True" .

ex:Customer2 a ex:Customer ;
        ex:customerStatus "Active" ;
        ex:invoiceAccepted "True" .
```

Listing 13: Update Customer Objects in SPARQL

```
PREFIX ex: <http://examplehost.org/>

DELETE {
        ?c ex:invoiceAccepted ?oldValue .
}
INSERT {
        ?c ex:invoiceAccepted "False" .
}
WHERE {
        ?c a ex:Customer ;
        ex:customerStatus "Inactive" ;
        ex:invoiceAccepted ?oldValue .
}
```

Listing 14: Aggregation and Functions in SPARQL

```
PREFIX ex: <http://examplehost.org/>

SELECT ?companyName (COUNT(?partner) AS ?partnersCount) (AVG(?
    partnerRevenue) AS ?averagePartnerRevenue)
WHERE {
?company a ex:Company ;
                ex:name ?companyName ;
                ex:partnerOf ?partner .

?partner a ex:Company ;
                        ex:revenue ?partnerRevenue .
}
GROUP BY ?companyName
```

Listing 15: Expressive Filtering in SPARQL

```
PREFIX ex: <http://example.org/>

SELECT ?companyName ?partnerName ?partnerRevenue
WHERE {
?company a ex:Company ;
                ex:name ?companyName ;
                ex:industry "Tech" ;
                ex:partnerOf ?partner .

?partner a ex:Company ;
                        ex:name ?partnerName ;
                        ex:revenue ?partnerRevenue .
FILTER (?partnerRevenue > 1000000)
}
```

Listing 16: Interact with SPARQL and Python

```
from SPARQLWrapper import SPARQLWrapper, JSON

ENDPOINT = "http://your-sparql-endpoint/sparql"
USERNAME = "your-username"
PASSWORD = "your-password"

def add_customer(sparql, customer_name, location_name):
        query = f"""
        PREFIX ex: <http://example.org/>

        INSERT DATA {{
        ex:{customer_name} a ex:Customer ;
                                                ex:name "{
                                                    customer_
                                                    name}" ;
                                                ex:located ex:{
                                                    location_
                                                    name} .
```

```
        ex:{location_name} a ex:City ;
                                                ex:name "{
                                                    location_
                                                    name}" .
        }}
        """
        sparql.setQuery(query)
        sparql.method = 'POST'
        sparql.query()

def print_customer(sparql, customer_status):
        query = f"""
        PREFIX ex: <http://example.org/>

        SELECT ?locationName
        WHERE {{
        ?customer a ex:Customer ;
                                ex:customerStatus "{customer_
                                    status}" ;
                                ex:located ?location .
        ?location ex:name ?locationName .
        }}
        ORDER BY ?locationName
        """
        sparql.setQuery(query)
        sparql.setReturnFormat(JSON)
        results = sparql.query().convert()

        for result in results["results"]["bindings"]:
                print(result["locationName"]["value"])

sparql = SPARQLWrapper(ENDPOINT)
sparql.setHTTPAuth("BASIC")
sparql.setCredentials(USERNAME, PASSWORD)

add_customer(sparql, "XYZ_GmbH", "Bremen")
print_customer(sparql, "active")
```

Listing 17: Graph Pattern Matching in SPARQL

```
SELECT ?a ?b
WHERE {
        ?a ?r ?b .
}
```

BIBLIOGRAPHY

[AWS24] AWS Amazon Web Services. *Accessing the Neptune Graph with openCypher*. June 2024. URL: https://docs.aws.amazon.com/neptune/latest/userguide/access-graph-opencypher.html (visited on 06/24/2024).

[Bac+22] Thomas Bach et al. "Testing Very Large Database Management Systems: The Case of SAP HANA." In: *Datenbank-Spektrum* 22.3 (Nov. 2022), pp. 195–215. ISSN: 1610-1995. DOI: 10.1007/s13222-022-00426-x. URL: https://doi.org/10.1007/s13222-022-00426-x.

[BF17] Pablo Barceló and Gaëlle Fontaine. "On the data complexity of consistent query answering over graph databases." In: *Journal of Computer and System Sciences* 88 (2017), pp. 164–194. ISSN: 0022-0000. DOI: https://doi.org/10.1016/j.jcss.2017.03.015. URL: https://www.sciencedirect.com/science/article/pii/S0022000017300521.

[Bar+12] Pablo Barceló, Leonid Libkin, Anthony Lin, and Peter Wood. "A Expressive Languages for Path Queries over Graph-Structured Data." In: *ACM Transactions on Database Systems* 37 (Dec. 2012), p. 31. DOI: 10.1145/2389241.2389250.

[BLHL23] Tim Berners-Lee, James Hendler, and Ora Lassila. "The Semantic Web: A New Form of Web Content that is Meaningful to Computers will Unleash a Revolution of New Possibilities." In: *Linking the World's Information: Essays on Tim Berners-Lee's Invention of the World Wide Web*. 1st ed. New York, NY, USA: Association for Computing Machinery, 2023, 91–103. ISBN: 9798400707940. URL: https://doi.org/10.1145/3591366.3591376.

[Cab+23] João Vitor Lopes Cabral, Viviana Elizabeth Romero Noguera, Ricardo Rodrigues Ciferri, and Daniel Lucrédio. "Enabling schema-independent data retrieval queries in MongoDB." In: *Information Systems* 114 (2023), p. 102165. ISSN: 0306-4379. DOI: https://doi.org/10.1016/j.is.2023.102165. URL: https://www.sciencedirect.com/science/article/pii/S0306437923000017.

[CG85] S. Ceri and G. Gottlob. "Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries." In: *IEEE Transactions on Software Engineering* SE-11.4 (1985), pp. 324–345. DOI: 10.1109/TSE.1985.232223.

[CR24] Hassan Nazeer Chaudhry and Matteo Rossi. "Optimising queries for pattern detection over large scale temporally evolving graphs." In: *IEEE Access* (2024), pp. 1–1. DOI: 10.1109/ACCESS.2024.3417352.

[DRSG15] Emir Septian Sori Dongoran, W. Kemas Rahmat Saleh, and Alfian Akbar Gozali. "Analysis and implementation of graph indexing for graph database using GraphGrep algorithm." In: *2015 3rd International Conference on Information and Communication Technology (ICoICT)*. 2015, pp. 59–64. DOI: 10.1109/ICoICT.2015.7231397.

[DSW24] Matt Duckham, Qian Sun, and Michael F. Worboys. *GIS A Computing Perspective*. 3rd ed. London: CRC Press, 2024. ISBN: 0.1201/9780429168093-2.

[FWX16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. "Adding Counting Quantifiers to Graph Patterns." en. In: *Proceedings of the 2016 International Conference on Management of Data*. San Francisco California USA: ACM, June 2016, pp. 1215–1230. ISBN: 978-1-4503-3531-7. DOI: 10.1145/2882903.2882937. URL: https://dl.acm.org/doi/10.1145/2882903.2882937 (visited on 05/27/2024).

[Har24] Keith W. Hare. *ISO/IEC 39075:2024 Information Technology – Database languages*. Apr. 2024. URL: https://jtc1info.org/slug/gql-database-language/ (visited on 04/28/2024).

[Inc] Gremlin Inc. *Chaos Engineering on AWS*. URL: https://www.gremlin.com/aws (visited on 06/24/2024).

[Kum+24] Saurav Kumar, Deepika Deepika, Karin Slater, and Vikas Kumar. "AOPWIKI-EXPLORER: An interactive graph-based query engine leveraging large language models." In: *Computational Toxicology* 30 (2024), p. 100308. ISSN: 2468-1113. DOI: https://doi.org/10.1016/j.comtox.2024.100308. URL: https://www.sciencedirect.com/science/article/pii/S2468111324000100.

[Lop+23] André Lopes, Diogo Rodrigues, João Saraiva, Maryam Abbasi, Pedro Martins, and Cristina Wanzeller. "Scalability and Performance Evaluation of Graph Database Systems: A Comparative Study of Neo4j, JanusGraph, Memgraph, NebulaGraph, and TigerGraph." In: *2023 Second International Conference On Smart Technologies For Smart Nation (SmartTechCon)*. 2023, pp. 537–542. DOI: 10.1109/SmartTechCon57526.2023.10391694.

[MBM20] Safikureshi Mondal, Anwesha Basu, and Nandini Mukherjee. "Building a trust-based doctor recommendation system on top of multilayer graph database." In: *Journal of Biomedical Informatics* 110 (2020), p. 103549. ISSN: 1532-0464. DOI: https://doi.org/10.1016/j.jbi.2020.103549. URL: https://www.sciencedirect.com/science/article/pii/S1532046420301775.

[NCA16] Hubert Naacke, Olivier Curé, and Bernd Amann. "SPARQL query processing with Apache Spark." In: *CoRR* abs/1604.08903 (2016). arXiv: 1604.08903. URL: http://arxiv.org/abs/1604.08903.

[Noaa] *Neo4J*. 223. URL: https://neo4j.com/ (visited on 06/23/2024).

[Pat+24] Ayushi Patil, Shreya Mahajan, Jinal Menpara, Shivali Wagle, Preksha Pareek, and Ketan Kotecha. "Enhancing fraud detection in banking by integration of graph databases with machine learning." In: *MethodsX* 12 (2024), p. 102683. ISSN: 2215-0161. DOI: https://doi.org/10.1016/j.mex.2024.102683. URL: https://www.sciencedirect.com/science/article/pii/S2215016124001377.

[Pin+24] Elvira Pino, Fernando Orejas, Nikos Mylonakis, and Edelmira Pasarella. "A logical approach to graph databases." In: *Journal of Logical and Algebraic Methods in Programming* 141 (2024), p. 100997. ISSN: 2352-2208. DOI: https://doi.org/10.1016/j.jlamp.2024.100997. URL: https://www.sciencedirect.com/science/article/pii/S2352220824000518.

[Saa+23] Mohamed Saad, Yingzhong Zhang, Jinghai Tian, and Jia Jia. "A graph database for life cycle inventory using Neo4j." In: *Journal of Cleaner Production* 393 (2023), p. 136344. ISSN: 0959-6526. DOI: https://doi.org/10.1016/j.jclepro.2023.136344. URL: https://www.sciencedirect.com/science/article/pii/S0959652623005024.

[SWP24] Manish Sharma, William W Wang, and Simon Porter. *What is Azure Cosmos DB for Apache Gremlin?* June 2024. URL: https://learn.microsoft.com/en-us/azure/cosmos-db/gremlin/introduction (visited on 06/24/2024).

[Shi+22] Eunji Shin, Sangwoo Yoo, Yongtaek Ju, and Dongil Shin. "Knowledge graph embedding and reasoning for real-time analytics support of chemical diagnosis from exposure symptoms." In: *Process Safety and Environmental Protection* 157 (2022), pp. 92–105. ISSN: 0957-5820. DOI: https://doi.org/10.1016/j.psep.2021.11.002. URL: https://www.sciencedirect.com/science/article/pii/S095758202100598X.

[SF24a] Apache Software Foundation. *Apache Hadoop*. June 2024. URL: https://hadoop.apache.org/ (visited on 06/24/2024).

[SF24b] Apache Software Foundation. *Apache Spark*. June 2024. URL: https://spark.apache.org/ (visited on 06/24/2024).

[SF24c] Apache Software Foundation. *Apache TinkerPop: Gremlin*. June 2024. URL: https://tinkerpop.apache.org/gremlin.html.

[Sun24] Ricky Sun. "Chapter 5 - Scalable graphs." In: *The Essential Criteria of Graph Databases*. Ed. by Ricky Sun. Elsevier, 2024, pp. 223–269. ISBN: 978-0-443-14162-1. DOI: https://doi.org/10.1016/B978-0-443-14162-1.00006-4. URL: https://www.sciencedirect.com/science/article/pii/B9780443141621000064.

[TRRMT21] Santiago Timón-Reina, Mariano Rincón, and Rafael Martínez-Tomás. "An overview of graph databases and their applications in the biomedical domain." en. In: *Database* 2021 (May 2021), baab026. ISSN: 1758-0463. DOI: 10.1093/database/baab026. URL: https://academic.oup.com/database/article/doi/10.1093/database/baab026/6277712 (visited on 05/27/2024).

[W3C24] World Wide Web Consortium (W3C) W3C. *SPARQL*. June 2024. URL: https://www.w3.org/TR/sparql11-query/ (visited on 06/24/2024).

[Wih+20] Kemas Wiharja, Jeff Z. Pan, Martin J. Kollingbaum, and Yu Deng. "Schema aware iterative Knowledge Graph completion." In: *Journal of Web Semantics* 65 (2020), p. 100616. ISSN: 1570-8268. DOI: https://doi.org/10.1016/j.websem.2020.100616. URL: https://www.sciencedirect.com/science/article/pii/S1570826820300494.

[Xu+22] Chengshuo Xu, Abbas Mazloumi, Xiaolin Jiang, and Rajiv Gupta. "SimGQ+: Simultaneously evaluating iterative point-to-all and point-to-point graph queries." In: *Journal of Parallel and Distributed Computing* 164 (2022), pp. 12–27. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2022.01.007. URL: https://www.sciencedirect.com/science/article/pii/S0743731522000120.

[Yua22] Tian Yuanyuan. *The World of Graph Databases from An Industry Perspective*. Nov. 2022. URL: https://arxiv.org/pdf/2211.13170 (visited on 04/28/2024).

[Noab] *openCypher*. June 2024. URL: https://opencypher.org/ (visited on 06/23/2024).