FernUniversität Hagen

– Faculty of Mathematics and Computer Science –

Different 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

1 INTRODUCTION

1.1 RESEARCH TOPIC

Graph databases have developed into a powerful tool for modeling and querying com-plex 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 [Yua22]:

- Relationships in social networks
- Recommendation applications
- Biological networks

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 ISO standard is pointed out.

1.2 MOTIVATION

The aim of this paper is to provide an overview of the different languages 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. 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 [Har24]. In this paper, important proprietary languages are presented and differentiated from each other. In the further course, a 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
- relationships

even across various different graphs. In addition, an expressive language offers users the corresponding flexibility when querying graph structures.

2.2 EASE OF USE

For users, it is important that the corresponding query language is intuitive to understand, follows familiar and learned patterns and is as close as possible to the de facto industry standard. On the other hand, a less intuitive language means an increased learning curve, which makes the technology more difficult to use.

2.3 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. A recommendation system does not help anyone if it does not deliver an answer in a reasonable time. This criterion evaluates in particular the ability of the corresponding language to enable techniques that improve the performance of queries on graph databases. Particular attention is paid to the following two factors:

- Implementation of index-based structures
- Parallel processing

2.4 INTEROPERABILITY

How well a query language intergrates with other tools, techniques and API's is an important criteria for the decision for and against the query language and often for the entire database system.

2.5 CLOSENESS TO THE ISO STANDARD

n particular, attention is paid to the extent to which the respective language is oriented towards the ISO standard for query languages for graph databases. A language that is strongly oriented towards the ISO standard or whose model and structure has a high proportion of the ISO standard automatically ensures, that the user has to acquire a low level of proprietary knowledge, which is beneficial to the user-friendliness of the language. In addition, a strong orientation towards the ISO standard ensures interoperability with other technologies and systems, as it can be assumed that these are also oriented towards at least the basic concepts of the ISO standard.

In the next chapter the ISO-standard for Graph databases will be introduced to enable the reader to compare the corresponding language with the ISO-standard when reading chapter XXX.

3

THE ISO-STANDARD FOR QUERY LANGUAGES FOR GRAPH DATABASES

3.1 OVERVIEW OF THE GQL-STANDARD

In short, GQL is a new standard for a property graph database language developed by international standards committee. GQL is an acronym for Graph Query Language. It is the first time in more than 35 years, that the ISO released a new standard for database query language.

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. Various use cases for graph databases have already been mentioned in chapter xxx.

3.2 SCOPE OF THE GQL-STANDARD

As already mentioned 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
- modifying

property graph data.
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.
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.

3.3 EXAMPLES OF COMMON GQL QUERY OPERATIONS

This chapter provides a brief introductionto GQL capabilities including

- Queries and Graph Pattern Matching
- Add, modify and delete operations
- Transactions

3.3.1 *Queries and Graph Pattern Matching*

GQL quries are based on rich Graph Pattern Matching (GPM) language. The below example findes all nodes with a one-hop relationship to a node with a productId of *6594301c4aa9b3232889e7c3*

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
- Textoutput

Here is suggestion how the above listing 3.1 can be displayed: This below table shows how the output can be displayed as text
TODO table erstellen

3.3.2 *Quantified path patterns*

GQL GPM also provides capabilities for quantified graph patterns. The following example finds all paths where one node is related to another noder up to ten hops:

Listing 3.2: Example for quantified graph pattern

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

More complex quantified path patterns are possible.

3.3.3 *Complex Graph pattern*

The following example matches a pattern with nodes of type *Person* and *Company*. The relationships between the nodes are *worksFor* and *partnerOf*.

Listing 3.3: 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
```

The mathing pattern:

Listing 3.4: Matching pattern of complex quantified graph query

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

matches a pattern where a *Person* works for a *Company* and that *Company* patners with another *Company*.

The filter of the query:

Listing 3.5: Filter of complex quantified graph query

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

filters all persons with an age > 25 and companys with a revenue > 100000.

The part

Listing 3.6: Aggregations and collections of complex quantified graph query

```
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:

Listing 3.7: Further matching of complex quantified graph query

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

Listing 3.7 matches additional patterns where a *Person knows* another *Person* who are younger than 30.

The Return statement:

Listing 3.8: Return statement of complex quantified graph query

```
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
```

specifies the output of the querie **??** and returns the name of the *Person*, *Company* and *Partnter* along with the count of *worksFor* relationships, the list of *partnerOf* relationships, and the names of the *youngColleagues*.
The output will orderd by the names of *person.name* and *company.name* of the corresponding nodes *Person* and *Company*. Next we will cover typical Insert, Update and Delete operations.

3.3.4 *Insert, Update and Delete operations*

The following example shows a typical insert statement:

Listing 3.9: Typical insert statement

```
/*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. 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 3.10: 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'})
```

The statement 3.10 inserts two nodes *Customer* and *City* and the relationship *located*.
Insert operations as shown in Listing 3.11 can also be the result of a *MATCH* pattern:

Listing 3.11: Insert statement for an edge

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

In the Listing 3.11 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 a *MATCH* caluse 3.11 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 edgs.
An Update in GQL is done by identifiifying the nodes or edges to be updated. After identifiifying the instances the user can set or remove properties:

Listing 3.12: Update statements in GQL

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

The following example removes properties of a node *Customer*:

Listing 3.13: Delete propteries in GQL

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

The following example deletes a node and the related nodes:

Listing 3.14: Remove nodes and in GQL

```
/*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.

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 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. The following example illustrates the 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)

}
```

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 operations does not follow the restrictions of the graph type the transaction will be rolled back and the database system will throw an error.

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**. The user of the database has 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**.

3.7 SUMMARY OF THE INTRODUCTION OF THE GQL STANDARD

In this chapter, the standard was briefly introduced to give the reader the opportunity to better compare the individual languages presented in the following chapter against the standard and thus to better understand the comparison criterion TODO kriterium referernzieren.

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 this chapter, specific database systems, their concepts and and syntax will be introduced and compared against the criteria definded in chater TODO reference to chapter.

4.1 OPENCYPHER

openCypher is an open Source Framework which is the basis for Cyper the Query language developed by Neo4J. Basically, openCypher is the query language which is most near to the GQL standard or in other words GQL is strong oriented on the fundamentals and concepts of openCypher. The core syntax of GQL and openCypher is largely identical. Next, the five 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 TODO reference to chapter two. The expressiveness of all querie languages will be evaluated against the following criteria:

- Graph Pattern Matching
- Data Manipulation
- Aggregation and Functions
- Expressive Filtering

openCypher suppports intuitive pattern matching with a concise syntax that enables the user to describe complex graph structures easily which the following example shows:

Listing 4.1: Graph Pattern Matching in openCypher

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

As shown in Listing 4.1 the user can **search** all nodes *a* related to node *b* by relationship *r*.
The following example shows how the user can **manipulate** data in openCypher:

Listing 4.2: Graph Pattern Matching in openCypher

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

Also, Aggregation and Functions are supported in openCypher:

Listing 4.3: Aggregation and Functions in openCypher

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

Next is an example for expressive Filtering in openCypher:

Listing 4.4: 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
```

4.1.2 *Ease of use of openCypher*

Due to the proximity to the GQL standard, it can be stated that if a user has learned and understood the basic concepts of query languages for graph databases, they can also learn the openCypher language, simply for the reason that the language is highly oriented towards industry standards. With the GQL standard, the aim was to develop a query language for graph databases that is as intuitive as possible in order to, among other things, make the technology available to a wide range of users. open the technology to a wide range of user. The example in **??** shows how clear and reduced in complexity the language is.

4.1.3 *Performance and performance optimization of openCypher*

The complexity of a query, including the number of nodes, the relationships and the depth of the pattern to match has uge impact of the performance. Also the size of the data which has 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 of large datasets. The following example Listing 4.5 shows the creation of an index in openCypher:

Listing 4.5: Index creation in openCypher

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

The Example shown in Listing 4.5 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.
In openCypher the user can analyze how the query works by using the keyword *EXPLAIN* and *PROFILE* to optimize the query for example by limiting the scope:

Listing 4.6: Using EXPLAIN or PROFILE in openCypher

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

for example by limiting the scope:

Listing 4.7: 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:

Listing 4.8: Batch operations in openCypher using UNWIND

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

4.1.4 *Interoperability of openCypher*

Many vendors for Graph Databases using openCypher for example:

- Neo4J
- Memgraph
- SAP HANA
- RedisGraph
- AWS Amazon Web Services

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 following example shows the integration in an example Python application and is based on the original documentation of Neo4J:

Listing 4.9: 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")
```

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. The example in Listing 4.9 has shown how openCypher can be used via Python.
However, it is critical to note that the further development of openCypher is being driven in particular by providers such as Neo4J and AWS, which in turn have a strong commercial interest in further establishing openCypher as a standard. In this context, it would appear desirable for competition to emerge in the graph database sector and for the technology to remain truly open and for the market power of individual technology providers, which already tend to be very large, to be further consolidated.

4.1.5 *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. 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

4.2 GREMLIN

Gremlin is query and traversal language for graph databases, developed as part of Apache TinkerPop. 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. Gremlin provides a high level abstraction layer. This means that not only TinkerPop-enabled graph systems execute Gremlin traversals, but also, every Gremlin traversal can be evaluated as either a **real-time database query** or as a **batch analytics query**. **TinkerPop** standard.

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
- Reducing

The following example points out the concept and it based on the benchmark query definded in Listing 3.3:

Listing 4.10: 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'))
```

4.2.2 *Expressiveness of Gremlin*

To evaluate the Expressiveness of Gremlin the examples from Chapter TODO will be applied to Gremlin. The example for Graph Pattern Matching defined in TODO will be implemented in Gremlin as the following:

Listing 4.11: 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. In Gremlin the termin vertice is used instead of node but it means almost the same TODO Fußnote.

```
.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 4.12: 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 in TODO reference

Listing 4.13: 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'))
```

The part

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

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

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

traverse out from *c*. 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'),
        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 defined in TODO is for flatten the grouped results and organize the output. The next query is based on TODO and shows how Expressive Filtering works in Gremlin:

Listing 4.14: 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'))
```

The query does the same as in TODO. 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 (Gremlin Language Variants). Important supported languages are:

- Python
- Java
- JavaScript

This allows the developer to use Gremlin within the current or preferred development environment. As already mentioned in TODO Gremlin operates independetly of the corresponding database system. It can be used with every database the TinkerPop stack, but huge vendors like also provide plug in's to Gremlin including Neo4J, AWS and Azure. In addition, there are seamless integrations to big data systems such as:

- Apache Hadoop and Spark
- Apache Giraph

The following example is based on TODO and implements the functionality in native Gremlin:

Listing 4.15: 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()
```

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 TODO and then use Gremlin to run the query. The following rules should be considerd to optimize the performance of Gremlin query:

- Filtering has 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

Below is one example for the Äpply filter earlyßtrategy:

Listing 4.16: 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
    ')
```

As already shown in Example TODO the user can limit the results of a Gremlin query:

Listing 4.17: Set Limits in Gremlin

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

The user can also use the concepts of *PROFILE* and *EXPLAIN* already introduced in TODO 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 openCypher*

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. The syntax of Gremlin is procedural like a programming language and the GQL standard is inspired by SQL.
But, users who are familiar with the concepts of querying non relational databases like MongoDB will recognize the concepts of Gremlin.
In Addition, important Big Data technologies like Apache Hadoop TODO 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. TODO.

4.3 SPARQL

SPARQL is a graph-based query language for querying content from the Resource Description Framework (RDF) description system, which is used in databases to formulate logical statements about any object. The name is a recursive acronym for SPARQL Protocol And RDF Query Language.

4.3.1 *RDF Ressource Description Framework*

The Resource Description Framework (RDF) is a standard model for the exchange and representation of information on the web. It was developed by the W3C (World Wide Web Consortium) and is a central component of the **Semantic Web**. RDF is a simple data structure consisiting of a triple of

- Subjects
- Predicates
- Objects

Any Ressource has an identifier the Uniform Ressource Identifier (URI). 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. Below the Benchmark query TODO is performed by using SPARQL:

Listing 4.18: 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 the almost the same as the *WITCH* Clause in the Benchmark query TODO. *ORDER BY* and *LIMIT* sort and limit the result.

The results of SPARQL queries can be result sets or RDF diagrams.
As in the last Exampled of openCypher TODO and Gremlin TODO the criterias for Expressiveness defined in TODO will be applied to SPARQL.

4.3.2 *Expressiveness of SPARQL*

SPARQL support GPM through triple pattern TODO, which are conceptually similar to to the pattern matching in property graph query languages but the syntax is different. The query of Listing 4.1 TODO can be written in SPARQL as the following:

Listing 4.19: Graph Pattern Matching in SPARQL

```
SELECT ?a ?b
```

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

The query TODO matches any triple where *?a* is connected to *?b* by any predicate *?r*. As already mentioned RDF data is represented in triple TODO. Instead of setting properties on nodes, the user add or modify triples. The following example creates two *Customer* Objects:

Listing 4.20: 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" .
```

On the created Objects the Update query based on TODO will be performed:

Listing 4.21: 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 .
}
```

The query shown in TODO which provides an example for aggregation and the usage of functions can be written in SPARQL as follows:

Listing 4.22: 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
```

The concepts regarding the query TODO are aready explained in the context of TODO- Based on listing TODO the following example shows the Implementation of expressive Filtering in SPARQL:

Listing 4.23: 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)
}
```

4.3.3 *Interoperability of SPARQL*

As Gremlin TODO for SPARQL are libraries for important programming languages available, like

- Python
- Java
- JavaScript

which ensures strong interoperability between SparQL and other systems and APIs. Also Integrations for Big Data and ETL (Extract, Transform, Load) tools like

- Apache NiFi
- Talend
- Apache Hadoop
- Apache Spark

The integration with Python shown in TODO and TODO can be also implemented in SPARQL by using Python and related libraries:

Listing 4.24: 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")
```

4.3.4 *Performance and performance optimization in SPARQL*

As in other Languages so general rules should be applied on SPARQL Queries for example the already mentioned filter early strategy TODO which 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 mentioned in TODO. As in Gremlin the tool set for performance optimization depends on the underlying database system TODO.

4.3.5 *Closeness to the GQL standard of SPARQL*

SPARQL excels in the context of **RDF** and the **Semantic Web**, offering robust querying capabilities for

- Linked Data and
- Ontologies

The capabilities provided in the GQL standard are designed for property graphs and are intuitive for pattern matching and graph data algorithms for example implemented in the context of

- Social Networks
- Recommendation Systems

Both, GQL and SPARQL are declarative languages, allowing the user to define which data is needed without specifing how the query has to be executed. Also both languages are oriented on the SQL standard and proving concepts people know when they are familiar with SQL. In conclusion, the concepts are similar but the underlying datamodel is likely but not the same. Graph data is be

LITERATUR

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

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