QUERY LANGUAGES FOR GRAPH DB

FOUNDATIONS OF MODERN QUERY LANGUAGES FOR GRAPH DATABASES

Research Paper by Center for Semantic Web Research:
RENZO ANGLES, Universidad de Talca
MARCELO ARENAS, Pontificia Universidad Católica de Chile
PABLO BARCELÓ and AIDAN HOGAN, DCC, Universidad de Chile
JUAN REUTTER and DOMAGOJ VRGOČ, Pontificia Universidad Católica de Chile

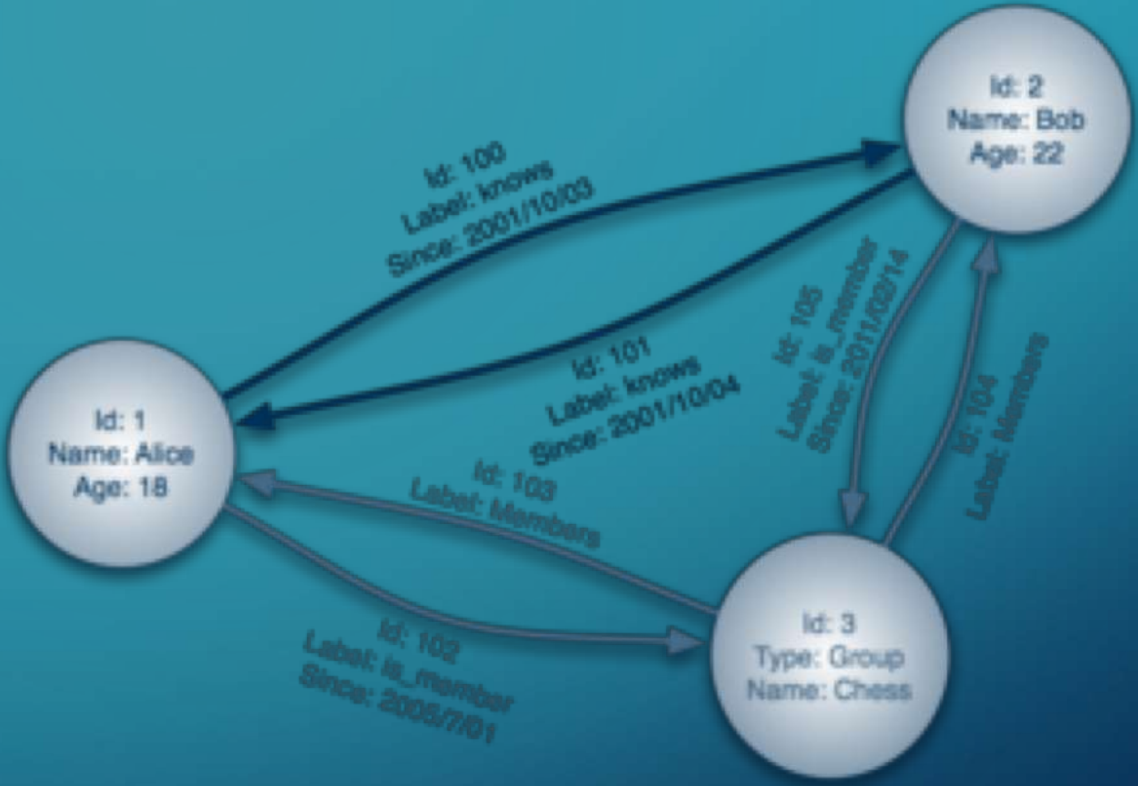
WHAT IS A GRAPH DATABASE?

- Graph Structures
- Has relationships
- Single operation.

- A Graph Database uses Graph Structures for semantic queries.
- Comprises Nodes, Edges and Properties.
- Directly relates data items.
- Relationships allow data to be linked together directly.

WHAT IS A GRAPH DATABASE?

- Nodes
- Edges
- Properties



RELATIONAL DATABASES VS GRAPH DATABASES

- The relational model gathers data together using information in the data.
- This can be a time consuming process in large tables.
- Relational databases do not inherently contain the idea of fixed relationships between records.
- Depending on the complexity of the query, the number of joins, and the indexing of the various keys, the system may have to search through multiple tables and indexes, gather lots of information, and then sort it all to match it together.

RELATIONAL DATABASES VS GRAPH DATABASES

- Graph databases directly store the relationships between records.
- The true value of the graph approach becomes evident when one performs searches that are more than one level deep.
- Time Complexity: O(log n)

RELATIONAL DATABASES VS GRAPH DATABASES

- The relative advantage of graph retrieval grows with the complexity of a query.
- Properties add another layer of abstraction to this structure that also improves many common queries.
- Relational databases are very well suited to flat data layouts, where relationships between data is one or two levels deep.
- Graph databases are aimed at datasets that contain many more links.

GRAPHQL

- GraphQL is an open source data query and manipulation language, and a runtime for fulfilling queries with existing data.
- It provides a more efficient, powerful and flexible alternative to REST and ad-hoc web service architectures.
- It allows clients to define the structure of the data required, and exactly the same structure of the data is returned from the server, therefore preventing excessively large amounts of data from being returned.
- GraphQL supports reading, writing (mutating) and subscribing to changes to data (realtime updates).

AQL (ARANGODB QUERY LANGUAGE)

- AQL is the SQL-like query language used in the ArangoDB database management system.
- It supports CRUD operations for both documents (nodes) and edges.
- But it is not a data definition language (DDL).
- AQL does support geospatial queries.

AQL (ARANGODB QUERY LANGUAGE)

AQL is **JSON-oriented** as illustrated by the following query, which also illustrates the intuitive "dot" notation for accessing the values of keys:

```
FOR x IN [{"a": {"A":1}}, {"a": {"A": 2}}]
   FILTER x.a.A < 2
   RETURN x.a
```

CYPHER QUERY LANGUAGE

- Cypher is a declarative graph query language that allows for expressive and efficient querying and updating of a property graph.
- Cypher is a relatively simple but still very powerful language.
- Very complicated database queries can easily be expressed through Cypher.
- This allows users to focus on their domain instead of getting lost in database access.

CYPHER QUERY LANGUAGE

- Cypher is based on the Property Graph Model, which in addition to the standard graph elements of **Nodes** and **Edges** (called as **Relationships**) adds **Labels** and **Properties** as concepts.
- Nodes may have **ZERO** or **MORE** labels, while each relationship has exactly **ONE** relationship type.
- Nodes and relationships also have **ZERO** or **MORE** properties, where a property is a **key-value** binding of a string key and some value from the Cypher type system.
- Each node and relationship can have **ZERO** or **MORE** properties.

CYPHER QUERY LANGUAGE

Cypher contains a variety of clauses. Among the most common are: MATCH and WHERE. These functions are slightly different than in SQL.

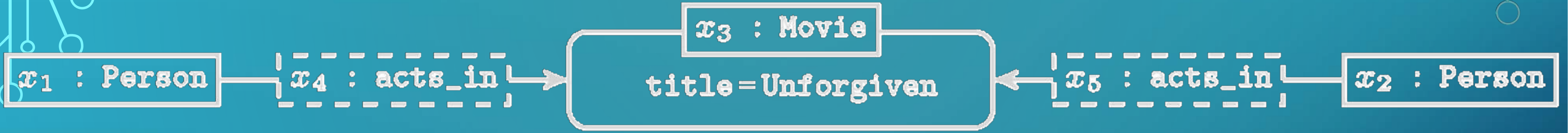
For example, the below query will return all movies where an actor named 'Nicole Kidman' has acted, and that were produced before a certain year (sent by parameter):

```
MATCH (nicole:Actor {name: 'Nicole Kidman'})-
[:ACTED_IN]->(movie:Movie)

WHERE movie.year < $yearParameter

RETURN movie
```

THE FOLLOWING CYPHER QUERY REPRESENTS THE FOLLOWING GRAPH STRUCTURE:



MATCH
(x1:Person) -[:acts_in]-> (:Movie {title:"Unforgiven"})
<-[:acts_in]- (x2:Person)}
RETURN x1,x2

if we wanted to construct a pattern that retrieves all pairs of actors who act in the same movie, including pairs that repeat the same actor, we would use the following Cypher statement:
MATCH (x1:Person) -[:acts_in]-> (x3:Movie {title:"Unforgiven"})
MATCH (x2:Person) -[:acts_in]-> (x3)
RETURN x1,x2

USING UNION, MINUS AND OPTIONAL OPERATORS

- We start with an example of a union to find movies that Clint Eastwood has acted or directed in.

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie {title:"Unforgiven"})
MATCH (x2:Person) -[:acts_in]-> (x3) RETURN x1,x2
```

- We could use difference to ask for people who acted in the movie Unforgiven but who did not (also) direct it:

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie {title:"Unforgiven"})
WHERE NOT (x1) -[:directs]-> (x3) RETURN x1.name
```

- We now use optional and filter to find movies in which people have acted and other ways they participated in the movie:

```
MATCH (x1:Person) -[:acts_in]-> (x3:Movie) OPTIONAL MATCH (x1) -[x4]-> (x3) WHERE type(x4)
<> "acts_in" RETURN x1.name AS name, x3.title AS movie, type(x4) as part
```

THIS RETURNS…

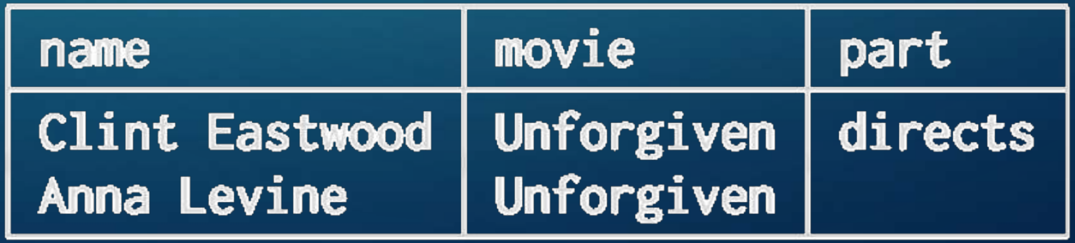
- Both patterns to the left and right of the UNION will be evaluated independently and their results unioned. The “ALL” keyword indicates that duplicates should be returned. Omitting the “ALL” keyword, the title would appear once.

:Unforgiven (Returned twice)

- The “NOT” keyword indicates the difference operator: any match for the initial pattern that is compatible with a match for the pattern indicated after “NOT” will be removed.

:Anna_Levine

- Cypher allows the use of the operator AS in the RETURN clause to indicate that the results of the query should be displayed under some specific names for columns. For instance, the use of “x3.title AS movie” indicates that the values of the property title of the nodes stored in the variable x3 will be displayed in a column with name movie.



name	movie	part
Clint Eastwood	Unforgiven	directs
Anna Levine	Unforgiven	

GREMLIN

- Gremlin is a graph traversal language and virtual machine developed by Apache TinkerPop of the Apache Software Foundation.
- Gremlin works for both OLTP-based graph databases as well as OLAP-based graph processors.
- Analogy: **Apache TinkerPop** and **Gremlin** are to **graph databases** what the **JDBC** and **SQL** are to **relational databases.**
- The **Gremlin traversal machine** is to graph computing as what the **Java virtual machine** is to general purpose computing.
- Each node and edge can have **ZERO** or **MORE** attributes.

GREMLIN

For each vertex in the graph, emit its label, then group and count each distinct label.

```
gremlin> g.V().label().groupCount()
==>[occupation:21, movie:3883, category:18,
user:6040]
```

GREMLIN

What year was the oldest movie made?

```
gremlin>
g.V().hasLabel('movie').values('year').min()
==>1919
```

What is Die Hard's average rating?

```
gremlin> g.V().has('movie','name','Die
Hard').inE('rated').values('stars').mean()
==>4.121848739495798
```

GREMLIN

For each category, emit a map of its name and the number of movies it represents.

```
gremlin>
g.V().hasLabel('category').as('a','b').select('a','b′).
by('name').by(inE('category').count())
```

```
==>[a:Animation, b:105]           ==>[a:Horror, b:343]
==>[a:Comedy, b:1200]             ==>[a:Sci-Fi, b:276]
==>[a:Adventure, b:283]           ==>[a:Documentary, b:127]
==>[a:Romance, b:471]             ==>[a:Mystery, b:106]
==>[a:Drama, b:1603]              ==>[a:Film-Noir, b:44]
==>[a:Thriller, b:492]            ==>[a:Western, b:68]
```

GREMLIN

For each movie with at least 11 ratings, emit a map of its name and average rating. Sort the maps in decreasing order by their average rating. Emit the first 10 maps (i.e. top 10).

```
gremlin>
g.V().hasLabel('movie').as('a','b').where(inE('rated').count().
is(gt(10))).select('a','b').by('name').by(inE('rated’).
values('stars').mean()). order().by(select('b'),decr). limit(10)
```

```
==>[a:Sanjuro, b:4.608695652173913]
==>[a:Seven Samurai (The Magnificent Seven),
b:4.560509554140127]
==>[a:Shawshank Redemption, The, b:4.554557700942973]
==>[a:Godfather, The, b:4.524966261808367]
==>[a:Close Shave, A, b:4.52054794520548]
==>[a:Usual Suspects, The, b:4.517106001121705]
```

GREMLIN

What 80's action movies do 30-something programmers like? Group count the movies by their name and sort the group count map in decreasing order by value. Clip the map to the top 10 and emit the map entries.

```
gremlin>
g.V().match(__.as('a').hasLabel('movie’),__.as('a').out('category’).
has('name','Action'),__.as('a').has('year',between(1980,1990)),__.as('a’).
inE('rated').as('b’),__.as('b').has('stars',5),__.as('b').outV().as('c'),__.
as('c').out('occupation’).has('name','programmer’),__.as('c’).
has('age',between(30,40))).select('a’).groupCount().by('name').order(local).
by(valueDecr).limit(local,10)
```

```
==>Raiders of the Lost Ark=26
==>Star Wars Episode V - The Empire Strikes Back=26
==>Terminator, The=23
==>Star Wars Episode VI - Return of the Jedi=22
==>Princess Bride, The=19==>Boat, The (Das Boot)=11
==>Indiana Jones and the Last Crusade=11
==>Star Trek The Wrath of Khan=10
```

GREMLIN

Which movies are most central in the implicit 5-stars graph?

```
gremlin> g.V().repeat(outE('rated’).
has('stars',5).inV().groupCount('m').by('name’).
inE('rated').has('stars', 5).outV()).times(4).cap('m')
```

```
==>Star Wars Episode IV - A New Hope  35405394353105332
==>American Beauty  31943228282020585
==>Raiders of the Lost Ark  31224779793238499
==>Godfather, The 30258518523013057
==>Shawshank Redemption, The  28297717387901031
==>Schindler's List 27539336654199309
==>Silence of the Lambs, The  26736276376806173
```

TO RETRIEVE ALL MOVIES WHERE CLINT EASTWOOD IS AN ACTOR

```
G.V().hasLabel('Person').has('name','Clint Eastwood') .out('acts_in').hasLabel('Movie’)
```

- The call G.V() will return the set of all nodes in the graph (V stands for “vertex”)
- Apply two selections on the set of nodes.
- The sequence of calls G.V().hasLabel('Person').has('name','Clint Eastwood') retrieves precisely those nodes with label Person and name Clint Eastwood.
- The command out('acts_in') retrieves all nodes that can be reached from these latter nodes with an edge labelled acts_in.
- hasLabel('Movie’) filters nodes not labelled with Movie.

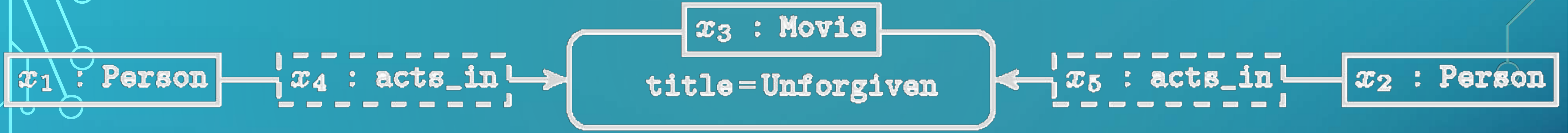
GRAPH TRAVERSALS IN GREMLIN

- The following Gremlin traversal allows us to obtain all co-actors of Clint Eastwood:

```
G.V().hasLabel('Person').has('name','Clint Eastwood')
.out('acts_in').hasLabel('Movies')
.in('acts_in').hasLabel('Person')
```

- This query navigates through the movies of Clint Eastwood as before, but then continues.
- The command in('acts_in') looks for nodes that are connected by an edge labelled acts_in in the opposite direction as the traversal.
- Then hasLabel(‘Person’) again filters out any nodes that are not of label Person.

AN EXPLICIT FILTER TO ENSURE THAT X1 DOES NOT MAP TO THE SAME CONSTANT AS X2 IN ANY MATCH, AND ALSO ADDS A PROJECTION



```
G.V().match(
            __.as('x1').hasLabel('Person').out('acts_in').hasLabel('Movies').as('x3'),
            __.as('x3').has('title','Unforgiven').in('acts_in').hasLabel('Person').as('x2'), .where('x1', neq('x2'))
).select('x1','x2')
```

- Each inner traversal can thus be seen as a tree-shaped bgp.
- These inner traversals are then joined to create a more complex bgp that may contain cycles.
- The two inner traversals are accompanied by a where command that calls a notequals (neq) filter to ensure that x1 and x2 are not bound to the same result.
- The select command (outside match) then performs a projection to select the output of the query: only the co-stars, not the movie.

SPARQL - SPARQL PROTOCOL AND RDF QUERY LANGUAGE

- RDF Query Language.
- Allows for a query to consist of triple patterns, conjunctions, disjunctions, and optional patterns.
- There exist tools that allow one to connect and semi-automatically construct a SPARQL query for a SPARQL endpoint.
- There exist tools that translate SPARQL queries to other query languages.
- SPARQL allows users to write queries against what can loosely be called "key-value" data.

SPARQL - SPARQL PROTOCOL AND RDF QUERY LANGUAGE

- In SQL relational database terms, RDF data can also be considered a table with **three** columns – the subject column, the predicate column, and the object column.
- SPARQL provides a full set of analytic query operations such as **JOIN, SORT, AGGREGATE** for data whose schema is intrinsically part of the data rather than requiring a separate schema definition.

SPARQL - SPARQL PROTOCOL AND RDF QUERY LANGUAGE

In the case of queries that read data from the database, the SPARQL language specifies four different query variations for different purposes.

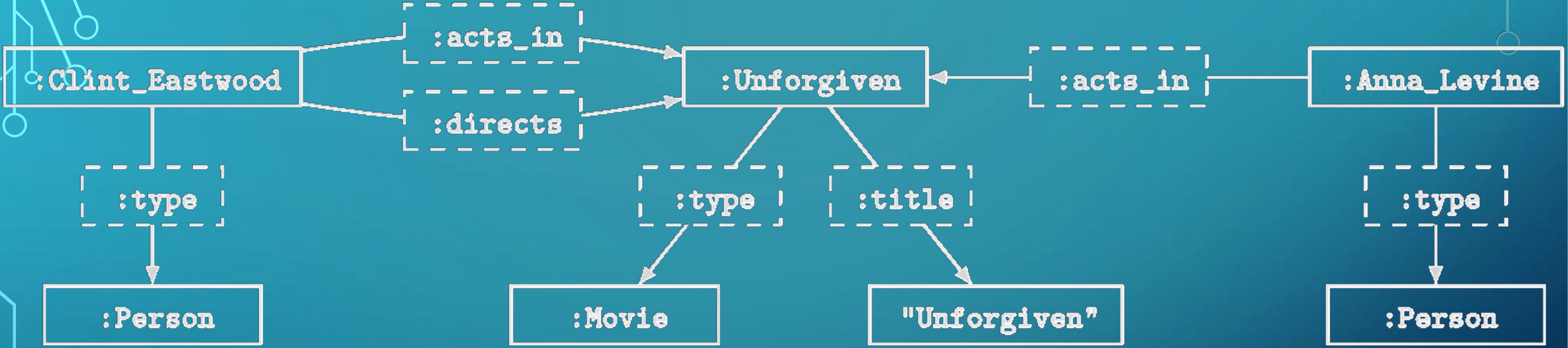
SELECT query: Used to extract raw values from a SPARQL endpoint, the results are returned in a table format.

CONSTRUCT query: Used to extract information from the SPARQL endpoint and transform the results into valid RDF.

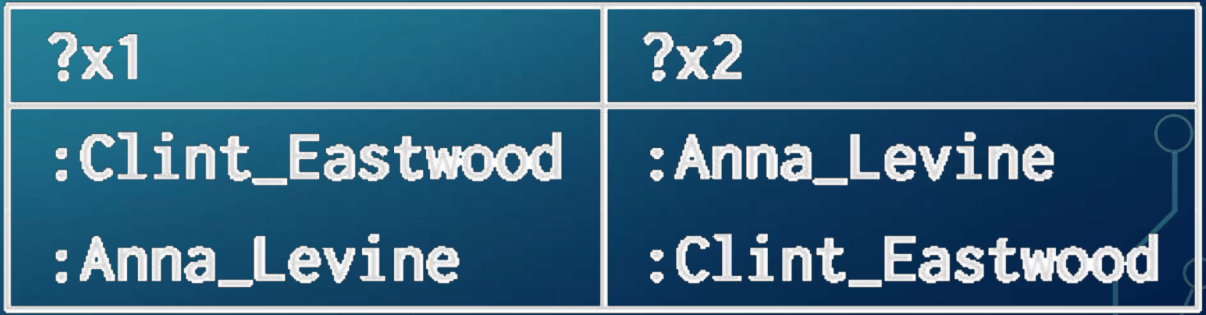
ASK query: Used to provide a simple True/False result for a query on a SPARQL endpoint.

DESCRIBE query: Used to extract an RDF graph from the SPARQL endpoint, the content of which is left to the endpoint to decide based on what the maintainer deems as useful information.

THE FOLLOWING SPARQL QUERY REPRESENTS A COMPLEX GRAPH PATTERN THAT COMBINES THE BASIC GRAPH PATTERN OF FIGURE WITH A PROJECTION THAT ASKS TO ONLY RETURN THE CO-STARS AND NOT THE MOVIE IDENTIFIER:



```
SELECT ?x1 ?x2
WHERE {
      ?x1 :acts_in ?x3 . ?x1 :type :Person .
      ?x2 :acts_in ?x3 . ?x2 :type :Person .
      ?x3 :title "Unforgiven" .
      ?x3 :type :Movie .
      FILTER(?x1 != ?x2)
}
```



?x1	?x2
:Clint_Eastwood	:Anna_Levine
:Anna_Levine	:Clint_Eastwood

USING UNION, MINUS **AND** OPTIONAL **OPERATORS**

- We start with an example of a union to find movies that Clint Eastwood has acted or directed in.

```
SELECT ?x WHERE {{ :Clint_Eastwood :acts_in ?x . } UNION { :Clint_Eastwood :directs ?x . }}
```

- We could use difference to ask for people who acted in the movie Unforgiven but who did not (also) direct it:

```
SELECT ?x WHERE {{ ?x :acts_in :Unforgiven . } MINUS { ?x :directs :Unforgiven . }}
```

- We could ask for movies that actors have appeared in, and any other participation they had with the movie besides acting in it:

```
SELECT ?x1 ?x2 ?x3 WHERE {{ ?x1 :acts_in ?x2 .} OPTIONAL { ?x1 ?x3 ?x2 . FILTER(?x3 != :acts_in) }}
```

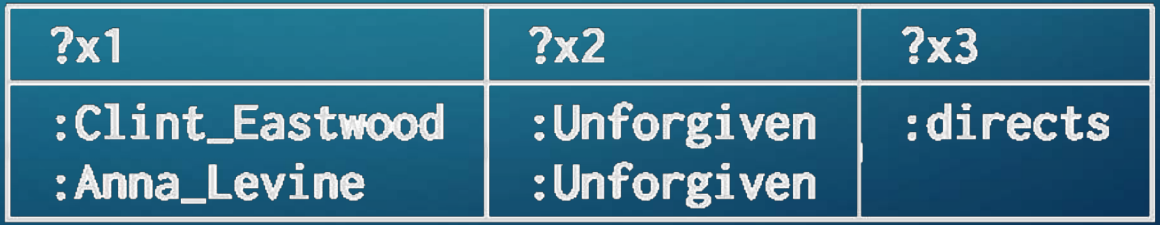
THIS RETURNS…

- Both patterns to the left and right of the UNION will be evaluated independently and their results unioned.

:Unforgiven

- Any match for the left side of the MINUS that is compatible with a match from the right side will be removed.

:Anna_Levine

- 

?x1	?x2	?x3
:Clint_Eastwood	:Unforgiven	:directs
:Anna_Levine	:Unforgiven	

A result is still returned for :Anna_Levine, even though she had no other participation in the movie; instead, the relevant column is left blank for that result.

AMAZON NEPTUNE

- Amazon Neptune supports popular graph models property graph and W3C's RDF, and their respective query languages Apache TinkerPop Gremlin and SPARQL.
- Allowing to easily build queries that efficiently navigate highly connected datasets.
- Neptune powers graph use cases such as recommendation engines, fraud detection, knowledge graphs, drug discovery, and network security.
- Neptune is a purpose-built, high-performance graph database engine.
- Compute scaling operations typically complete in a few minutes.
- Amazon Neptune will automatically grow the size of your database volume as your database storage needs grow.

AMAZON NEPTUNE

- With Neptune, you can quickly build fast Gremlin traversals over property graphs.
- Existing Gremlin applications can easily use Neptune by changing the Gremlin service configuration to point to a Neptune instance.
- With Neptune, you can easily use the SPARQL endpoint for both existing and new graph applications.
- Using Neptune’s VPC configuration, you can configure firewall settings and control network access to your database instances.
- Tag Neptune resources, and control the actions that your IAM users and groups can take on groups of resources that have the same tag (and tag value).

AMAZON NEPTUNE

- Amazon Neptune allows you to encrypt your databases using keys you create and control through AWS Key Management Service (KMS).
- Amazon Neptune allows you to log database events with minimal impact on database performance.
- Refer to the webpage below for the complete documentation of Neptune.

https://docs.aws.amazon.com/neptune/latest/userguide/intro.html

DGRAPH

- Dgraph is a horizontally scalable and distributed graph database, providing ACID transactions, consistent replication and linearizable reads.
- It's built from ground up to perform for a rich set of queries.
- Being a native graph database, it tightly controls how the data is arranged on disk to optimize for query performance and throughput, reducing disk seeks and network calls in a cluster.
- Dgraph's goal is to provide Google production level scale and throughput, with low enough latency to be serving real time user queries, over terabytes of structured data.
- Dgraph supports GraphQL-like query syntax, and responds in JSON and Protocol Buffers over GRPC and HTTP.

DGRAPH

Features	Dgraph	Neo4j	Janus Graph
Architecture	Sharded and Distributed	Single server (+ replicas in enterprise)	Layer on top of other distributed DBs
Replication	Consistent	None in community edition (only available in enterprise)	Via underlying DB
Data movement for shard rebalancing	Automatic	Not applicable (all data lies on each server)	Via underlying DB
Language	GraphQL inspired	Cypher, Gremlin	Gremlin
Protocols	Grpc / HTTP + JSON / RDF	Bolt + Cypher	Websocket / HTTP

DGRAPH

Features	Dgraph	Neo4j	Janus Graph
Transactions	Distributed ACID transactions	Single server ACID transactions	Not typically ACID
Full Text Search	Native support	Native support	Via External Indexing System
Regular Expressions	Native support	Native support	Via External Indexing System
Geo Search	Native support	External support only	Via External Indexing System
License	Apache 2.0 + Commons Clause	GPL v3	Apache 2.0

FEATURE CATEGORIZATION

- Identified the two main core features that are common in all modern graph query languages: pattern matching and navigation.
- To categorise pattern matching features, we identified the class of **Basic Graph Patterns** (**BGPs**), which should arguably form the core of any graph query language, and are indeed present in all of the practical systems we reviewed.
- These can be further extended with operators such as projection, union, or optional, among others, giving rise to **Complex Graph Patterns** (**CGPs**).
- In terms of navigational queries, following both the research literature and the practical solutions currently available, we identify paths as the core of all navigational queries over graphs, and adopt the well studied notion of **Regular Paths Queries** (**RPQs**) as the basis for navigating graphs.
- These can then be incorporated into **BGPs** giving rise to **Navigational Graph Patterns** (**NGPs**), which themselves can be further extended with operators such as union, optional, and so on, to create the notion of **Complex Navigational Graph Patterns** (**CNGPs**).

FEATURE CATEGORIZATION

- For matching basic graph patterns we classified the main proposals for the semantics into two categories:
  - **Homomorphism-based**: matching the pattern onto a graph with no restrictions.
  - **Isomorphism-based**: one of the following restrictions is imposed on a match:
    - no-repeated-anything: no part of a graph is mapped to two different variables,
    - no-repeated-node: no node in the graph is mapped to two different variables,
    - no-repeated-edge: no edges in the graph is mapped to two different variables.
- On the other hand, for path queries one can consider:
  - arbitrary paths;
  - shortest paths only;
  - paths not repeating a node (aka simple paths); and
  - paths not repeating an edge.

FEATURE CATEGORIZATION

- To exemplify the categorization, there is a review some of the key design choices made for SPARQL, Cypher and Gremlin: three of the currently most popular query languages used in graph database engines.
- Of course, all three languages extend upon these core features presented; however, this core offers a good starting point to further formalize, study and understand these languages.
- Table 1 contains a summary of these choices for pattern matching, and Table 2 likewise for navigational queries.

SUMMARY TABLE
GENERAL

Language	Type	Representation
SPARQL	Declarative Query Language	Triple patterns – subject, object and predicate
Cypher	Declarative Query Language	Patterns – Expressed syntactically following a pictorial intuition to encode nodes and edges with arrows between them.
Gremlin	Functional Query Language	Based on Navigational Queries rather than patterns – similar to a programming language.

SUMMARY TABLE
CHOICES FOR PATTERN MATCHING

Language	Supported Patterns	Semantics
SPARQL	all complex graph patterns	homomorphism-based.
Cypher	all complex graph patterns	no-repeated-edges.
Gremlin	complex graph patterns without explicit optional	homomorphism-based.

SUMMARY TABLE
CHOICES FOR NAVIGATIONAL QUERIES

Language	Path Expressions	Semantics	Choice of Output
SPARQL	more than RPQs	arbitrary paths, sets.	boolean / nodes
Cypher	fragment of RPQs	no-repeated-edge, bags	boolean / nodes / paths / graphs
Gremlin	more than RPQs	arbitrary paths, bags.	nodes / paths

CONCLUSION

This categorization also opens interesting possibilities for further work in terms of surveying and classifying other important aspects of graph databases that are infeasible to cover in this survey with the necessary depth.

Recent years have seen the re-emergence of graph databases as an important alternative to their more widely established relational cousin, bringing with them a variety of new challenges, new demands, and new questions. In this survey, we have provided an overview of the fundamental query features that underlie such databases and have provided a categorization that generalizes much of these recent developments and offers a bridge to known theoretical results while raising some new questions.

Of course, there are many open challenges facing graph databases in terms of standardizing query languages, implementing and optimizing engines for query evaluation, studying the theoretical properties of related problems, as well as evolving graph databases to meet emerging demands for graph analytics. We hope that this survey may serve as a useful guide for those involved in such efforts.

REFERENCES

- Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (September 2017), 40 pages.
- Silberschatz, Avi (28 January 2010). Database System Concepts, Sixth Edition (PDF). McGraw-Hill. p. D-29. ISBN 0-07-352332-1.
- "SQL and AQL (ArangoDB Query Language) Comparison". Arangodb.com. Retrieved 17 December 2017.
- "Cypher: An Evolving Query Language for Property Graphs" (PDF). Proceedings of the 2018 International Conference on Management of Data. ACM. Retrieved June 27, 2018.
- Rodriguez, Marko A. (2015). "The Gremlin graph traversal machine and language (invited talk)". *The Gremlin Graph Traversal Machine and Language*. p. 1. arXiv:1508.03843 Freely accessible [cs.DB]. doi:10.1145/2815072.2815073. ISBN 9781450339025.
- Hebeler, John; Fisher, Matthew; Blace, Ryan; Perez-Lopez, Andrew (2009). *Semantic Web Programming*. Indianapolis, Indiana: John Wiley & Sons. p. 406. ISBN 978-0-470-41801-7.