GraQL: A Query Language for High-Performance Attributed Graph Databases

Daniel Chavarría-Miranda, Vito Giovanni Castellana, Alessandro Morari, David Haglin and John Feo
Physical and Computational Sciences Directorate
Pacific Northwest National Laboratory (PNNL)
Richland, WA
{*daniel.chavarria, vitogiovanni.castellana, alessandro.morari, david.haglin, john.feo*}*@pnnl.gov*

***Abstract*—Graph databases are becoming a critical tool for the analysis of graph-structured data in the context of multiple scientific and technical domains, including cybersecurity and computational biology. In particular, the storage, analysis and querying of *attributed graphs* is a very important capability. Attributed graphs contain properties attached to the vertices and edges of the graph structure. Queries over attributed graphs do not only include structural pattern matching, but also conditions over the values of the attributes. In this work, we present GraQL, a query language designed for high-performance attributed graph databases hosted on a high memory capacity cluster. GraQL is designed to be the front-end language for the attributed graph data model for the GEMS database system.**

I. Introduction

Graph databases have gained increasing interest in the last few years due to the emergence of data sources that are sparse and not owned by the user. Both make it difficult to define a fixed, dense schema for the data and to analyze the data using traditional relational databases [1], [2], [3]. Cybersecurity and computational biology are two application areas in which graph models of data are increasingly important. In cybersecurity, interaction graphs representing communication occurring over time between different hosts or devices on a network can be modeled and represented accurately in a graph database [4]. Examples from biology include the modeling of biological pathways which represent the flow of molecular "signals" inside a cell for purposes of metabolism, gene expression or other cellular functions.

Flexible and efficient representations of graph data have become increasingly important for critical application areas in science, security, commerce and industry. While the relationship between entities is best represented as a graph, storing the set of fixed attributes associated with many entities as vertex-edge pair is wasteful. Consequently, the representation of *attributed graphs*, in which vertices and edges have collections of arbitrary attributes attached to them—represents a fruitful approach to store graph data models and to enable queries on them.

A multiplicity of graph database approaches have emerged as an answer to these challenges, with the repurposing of the semantic web's Resource Description Framework (RDF) and its associated SPARQL query language as one major ecosystem [5], [6]. Other approaches have built custom and specialized graph database systems [1], [2], [3] with varying degrees of success, including ones that run on clusters [7]. However, none have captured the dual nature of data and provided a transparent manner to specify queries that are a combination of table and graph operations. Typically, the answer to a graph query is an enumeration of subgraphs that *match* a particular pattern specified by the query. These patterns are specified in terms of their structure as well as conditions that the attached properties must satisfy. In the worst case, providing an answer to a graph query is equivalent to the NP-complete *subgraph homomorphism* problem.

The effective mapping of a graph database system to a cluster and the fast, efficient execution of complex graph queries remains a very challenging and open problem, due to the very nature of graph data and dynamic query environment. These challenges include the difficulty of partitioning graphs across nodes on a cluster, irregular and unstructured data accesses and parallelism, the possibility of obtaining large intermediate results, as well as the dynamic, just-in-time nature of the queries. By specifying which relationships are best represented as a graph and which are best stored as dense tables, the user can assist the system in making better decisions on how to layout data in memory and how/where to execute operations.

Our previous research focused on developing an effective mapping of RDF/SPARQL databases to high-performance clusters [8], [9], [10]. While successful, we encountered many difficulties because our system only supported graph representations. We found that we lacked efficient ways to store fixed sets of attributes and to express simple table operations over the data. Thus, to achieve greater user productivity, performance, and scalability, we have started to investigate an attributed graph data model and language extensions to capture the graph-table duality of real-word data and queries. Our focus remains an in-memory graph database in which the data resides across the aggregated memory of the nodes in the cluster. The principal intent is to minimize per query processing time and maximize throughput. For a cluster of large enough size or enough memory capacity per node, the overall capacity can be in the range of tens of Terabytes, enough for non-trivial graph data sets. For our target application space, it is worth while

to trade off the data capacity and persistence of storage, for the increased high performance and throughput available via DRAM.

We have based our attributed graph database design on a few key principles:

- All data is stored in tabular form (equivalent to SQL tables)
- Graph elements (vertices & edges) are represented as *views* over those tables
- All database elements are *strongly typed*

The rationale behind these design principles are to enable efficient yet rich and flexible representation of data for different database uses, while allowing for a clear mapping of the data and execution to a distributed memory cluster. In this paper, we present a new data model that provides both table and graph views of data sets and query language constructs to define attributed graph pattern searches. We provide both as extensions to SQL to minimize user discomfort and workplace disruption.

II. Database and Language Design

We describe in more detail the key principles behind our graph data model design and expand on the rationale for their choices using the Berlin queries, a well-known SPARQL/RDF benchmark [11]. We focus on a subset of queries that correspond to a business intelligence use case. This subset represents an e-commerce database of products, producers, offers, vendors, and reviews, as well as types and features associated with products, and persons writing reviews. Figure 1 illustrates a logical data model for the Berlin benchmarks. The ovals represent the data entities, while the arrows represent the relationships between them. In addition, each entity has an associated set of fixed, dense attributes.

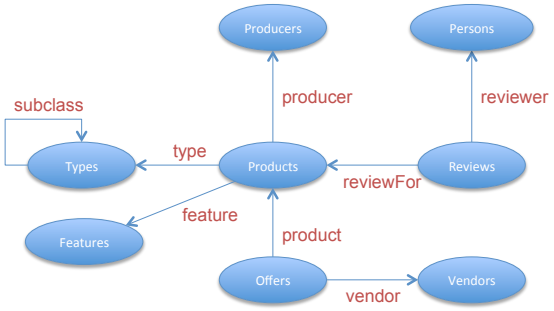

Figure 1. Berlin logical data model

A. Data Model & Data Definition Language (DDL)

All data in our database is intended to be stored in tabular form. We follow the same syntax and semantics established by SQL database systems, enabling us to leverage existing knowledge and experience with efficient algorithms for processing and querying table data. We also have the potential to use existing software platforms that manage tabular data without having to reinvent the wheel, thus enabling us to focus on the areas where we want to innovate: the vertex/edge view of the data and its graph query potential.

Appendix A gives the table declarations for the Berlin benchmarks. The tables' columns, which we refer to as attributes in our data model, are strongly typed. There is a table per entity, and two additional tables for the relations *ProductTypes* and *ProductFeatures*. In the case where the number of attributes is variable, as in the case of the number of types and features associated with a product, it is best to declare a separate table to store that data.

In order to support a graph data model on top of basic tabular data, we have created two new syntactic constructs to enable the declaration of vertices and edges and to specify their connection to the underlying tables that represent the data. Figures 2 and 3 define the vertices and edges of the graph view depicted in Figure 1.

```
create vertex TypeVtx(id)
from table Types

create vertex FeatureVtx(id)
from table Features

create vertex ProducerVtx(id)
from table Producers

create vertex ProductVtx(id)
from table Products

create vertex VendorVtx(id)
from table Vendors

create vertex OfferVtx(id)
from table Offers

create vertex PersonVtx(id)
from table Person

create vertex ReviewVtx(id)
from table Reviews
```

Figure 2. Berlin queries vertex declarations

Consider the declaration of *ProductVtx*, where each vertex instance corresponds directly to a row entry in the *Products* table. The vertex type declaration includes which columns from the table uniquely identify each vertex instance (single column in this case). We call these simple declarations in which a vertex instance maps directly to a row entry in a table, *one-to-one mappings*. We say that the primary key of the table is the same as the unique identifier of the vertex instance. For these one-to-one mappings it is easy to see how, given a vertex instance identifier (key), we can access additional attributes (columns) present in the source table.

An edge declaration specifies the vertex types it connects, and the order of the types indicates the direction of the edge. For example, a *producer* edge connects product vertices to

```
create edge subclass with
  vertices (TypeVtx as A, TypeVtx as B)
where A.subclassOf = B.id

create edge producer with
  vertices (ProductVtx, ProducerVtx)
where ProductVtx.producer = ProducerVtx.id

create edge type with
  vertices (ProductVtx, TypeVtx)
from table ProductTypes
where ProductTypes.product = ProductVtx.id
  and ProductTypes.type = TypeVtx.id

create edge feature with
  vertices (ProductVtx, FeatureVtx)
from table ProductFeatures
where ProductFeatures.product = ProductVtx.id
  and ProductFeatures.feature = FeatureVtx.id

create edge product with
  vertices (OfferVtx, ProductVtx)
where OfferVtx.product = ProductVtx.id

create edge vendor with
  vertices (OfferVtx, VendorVtx)
where OfferVtx.vendor = VendorVtx.id

create edge reviewFor with
  vertices (ReviewVtx, ProductVtx)
where ReviewVtx.reviewFor = ProductVtx.id

create edge reviewer with
  vertices (ReviewVtx, PersonVtx)
where ReviewVtx.reviewer = PersonVtx.id
```

Figure 3. Berlin queries edge declarations

producer vertices. If a **from table** clause appears, an edge is created for each table entry satisfying the **where** clause; otherwise, the tables of the vertex types are joined and an edge created for each entry in the result satisfying the **where** clause.

GraQL also supports more complex situations in which multiple vertex and edge types can be created from a single table, as well as from relational operations over several tables. In Figure 4 we declare two new vertex types, *ProducerCountry* and *VendorCountry*, and a new edge type, *export*. The semantics of our data model and language creates a vertex instance for every unique country code in the *Producers* and *Vendors* tables and an edge for every product produced in one country and offered by a vendor in another country. These more complex declarations where multiple rows in a table correspond to a single vertex or edge instance are called *many-to-one mappings*. Note that in the case of many-to-one mappings, the primary key of the table does not serve as a unique identifier (key) of the vertex or edge. Figure 5 illustrates the many-to-one example visually. In this case, the four-way join between the tables results in two edges created between the US and Canada (CA), and between Italy (IT) and China (CN).

```
create vertex ProducerCountry(country)
from table Producers

create vertex VendorCountry(country)
from table Vendors

create edge export with vertices
  (ProducerCountry, VendorCountry)
from table Producers, Products, Vendors, Offers
where (Producers.id = Products.producer) and
  (Vendors.id = Offers.vendor) and
  (Products.id = Offers.product) and
  (Producer.country != Vendor.country)
```

Figure 4. Richer vertex and edge declarations

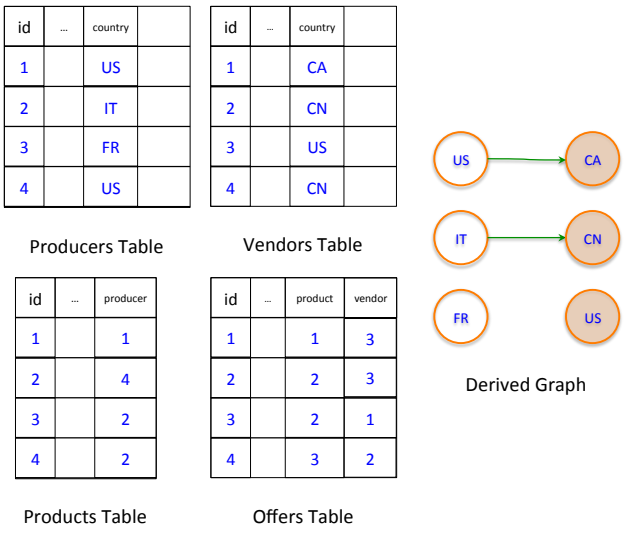

Figure 5. Tables and derived graph using a many-to-one mapping

Using relational algebra, we can define the creation of vertices from a source table more formally:

$$V(a_i, \ldots, a_k) = \Pi_{a_i,\ldots,a_k} \sigma_{\varphi}(T_{a_1,\ldots,a_n}) \quad (1)$$

Equation 1 describes how a vertex type can be created from a source table. First, a selection operation σ is applied to the rows in the table that satisfy condition φ, then a subset of the attributes of those rows are selected to form the key of the vertex type. The table T contains n attribute columns $(a_1, \ldots, a_n)$, while the resulting vertex type contains a subset of those n columns as its key. Only one *vertex instance* will be created for each unique combination of the specified vertex key. In this manner, a vertex type is a *view* over the underlying table data T.

The creation of edges from source (S) and target (T) vertices and associated table (A) can be formally defined as follows:

$$E(a_1, \ldots, a_n) = (S \bowtie (\sigma_{\varphi} A_{a_1,\ldots,a_n})) \bowtie T \quad (2)$$

Equation 2 describes the semantics of edge creation from two vertex types (source and target) and an associated table (relation). First a selection operation σ with condition φ is applied against the associated table. A natural join operation of these selected tuples is then performed with the source vertex instances and finally the result is joined with the target vertex instances.

1) Overall Graph, Vertex and Edge Types: The overall data can be treated as a single graph $G = (V, E)$, where the following properties hold:

$$V = \cup_\rho V_\rho$$
$$E = \cup_\tau E_\tau$$
$$\forall\ V_i, V_j : V_i \cap V_j = \emptyset$$
$$\forall\ E_i, E_j : E_i \cap E_j = \emptyset$$

That is, the set of vertex types forms a *partition* over the vertices of the graph, while the set of edge types forms a partition over the set of edges. We will take advantage of this property in the definition and interpretation of the query language.

We use the notation $E_i(V_a, V_b)$ to indicate that particular edge type E_i connects vertices of type V_a and V_b as its source and target respectively. Since there might be more than one edge type connecting two particular vertex types, we use the notation $\cup_j E_j(V_a, V_b)$ to represent all edge types that connect the same vertex types.

The underlying graph is more precisely described as a *multigraph* since more than one edge can exist between the same pair of vertices.

2) Data Ingest: Populating the database, including tabular data, vertex and edge instances occurs upon execution of an *ingest* command. The data definition commands set up the scaffolding and structure of the database, while the data ingestion process fully populates them. The ingest command is quite simple:

```
ingest table Products products.csv
```

In this case, the data for table "Products", as declared in the Appendix, is ingested from the file "products.csv". The file is expected to be accessible to the cluster nodes executing the database system, and formatted using the CSV (comma separated values) standard. It will be parsed according to the data types of the attributes in the corresponding table. Data ingest triggers not only the population of rows in the table, but also the generation of associated vertex and edge *instances* derived from the table.

Ingest commands are intended to be fully atomic with respect to other data definition and query commands.

B. Query Language

The main objective of our query language is to obtain subgraph selections from an overall graph that satisfy certain structural or attribute (column) properties. The syntax and semantics of our query language enable the expression of combinations of *structural paths* in the graph that the resulting subgraph must match. In addition to these subgraph queries, our language supports a subset of standard SQL relational/tabular operations to enable manipulation, preprocessing and postprocessing of graph data in table form. These relational operations follow naturally from the underlying tabular data store for graph data, as well as the interpretation of graph query results as tables described in the following sections.

```
select y.id
from graph
  ProductVtx(id = %Product1%)
    -productFeature->
  FeatureVtx()
    <-productFeature-
  def y: ProductVtx(id != %Product1%)
into table T1

select top 10
  id, count(*) as groupCount
from table T1
  group by id
  order by groupCount desc
```

Figure 6. Berlin Query 2 - *Select the top 10 products most similar to Product 1 rated by the count of features they have in common.*

Figure 6 shows the GraQL code for the second Berlin query. The result of the first select statement is a table of product ids, with each id repeated for each feature the product has in common with the specific product.

Path queries must start with a vertex type and end with a vertex type to enable correct formation of the resultant subgraphs. A vertex type must be followed by an edge type and an edge type must be followed by a vertex type. The series of character $-edge \rightarrow$ indicates a path from the left vertex to the right vertex along an outedge, and $\leftarrow edge-$ indicates a path from the right vertex to the left vertex along an inedge. Each element between $-$ and either $\rightarrow$ or $\leftarrow$ separators is a *graph query step*.

Our query syntax enables the specification of conditional expressions on keys (or other attributes) of each vertex. Vertex instances that satisfy those conditions are then considered for the next step in the query path. An empty parentheses "( )" indicate that no filter is applied. On each step of the query path, attributes can be compared against constants, other attributes belonging to the same step, and/or attributes from previous steps (if labeled, see Section II-B2).

1) Basic Path Queries: More formally, we define a path query q with $2n-1$ steps as follows:

$$q = V_1(\varphi_{V_1}) - E_1(\varphi_{E_1}) \rightarrow \cdots - E_{n-1}(\varphi_{E_{n-1}}) \rightarrow V_n(\varphi_{V_n}) \quad (3)$$

A path query is correctly formed only if it starts with a vertex type and ends with a vertex type. Vertex steps can only be followed and preceded by edge steps. Edge steps can only be followed and preceded by vertex steps. There is an initial vertex step that is not preceded by an edge step (V_1) and a final vertex step that is not followed by an edge step (V_n).

A query step for a vertex type V with attributes $a_1, \ldots, a_k$ with conditions φ is a *selection* on the underlying table data that represents the vertex type:

$$V_\varphi = \sigma_\varphi(V_{a_1,\ldots,a_k}) \tag{4}$$

As V represents a view over the underlying table data, we simplify the notation by treating V as if it was the table. Equation 4 describes the result of applying a vertex step in a query to the vertex type V. It is essentially a *relational selection* over V, in which the subset of vertices that matches condition φ is selected.

Similarly, a query step for an edge type E with attributes $a_1, \ldots, a_k$ represents a selection over the underlying relational data representing the edge type (see Equation 2): $E_\varphi = \sigma_\varphi(E_{a_1,\ldots,a_k})$.

The result of a path query is the subgraph that matches all of the query steps. That is for a path query q the matching subgraph corresponds to the set of vertices $V(q)$ that satisfy Equation 5 (the formulation for the satisfying set of edges $E(q)$ is analogous).

$$\begin{aligned} V(q) = \{&\upsilon_1 \in V_1(\varphi_{V_1}) \wedge \\ &\upsilon_2 \in V_2(\varphi_{V_2}) \wedge (\upsilon_1, \upsilon_2) \in E_1(\varphi_{E_1}) \wedge \\ &\upsilon_3 \in V_3(\varphi_{V_3}) \wedge (\upsilon_1, \upsilon_2) \in E_1(\varphi_{E_1}) \wedge \\ &(\upsilon_2, \upsilon_3) \in E_2(\varphi_{E_2}) \wedge \\ &\cdots \\ &\upsilon_n \in V_n(\varphi_{V_n}) \wedge (\upsilon_1, \upsilon_2) \in E_1(\varphi_{E_1}) \wedge \cdots \wedge \\ &(\upsilon_{n-1}, \upsilon_n) \in E_{n-1}(\varphi_{E_{n-1}})\} \end{aligned} \tag{5}$$

In other words, the path query is satisfied if each step in the query is satisfied by itself and its successor steps. The set of vertices selected at a particular step will be culled by subsequent steps of all vertices that have no path to vertices selected at that step.

2) Step Labels: Our query language enables the *labeling* of a query step. Labels enable references to vertices or edges that were matched as part of a previous step. In Figure 6, the set of products that share a feature with *%Product1%* is labeled with the syntax `def y`, allowing the `select` clause at the start of the query to refer to just those products.

The label mechanism is a simple way of referring to sets of vertices (or edges) from a particular step, that have matched the path *up to the point where the label appears*. In other words, the label *aliases* a set of vertices or edges that matches a step in the query path.

More formally we define *two* types of labels: the *set label* (`def X:`) and the *element-wise label* (`foreach x:`) used in the implementation of the first Berlin query shown in Figure 7.

```
select TypeVtx.id
from graph
  PersonVtx(country = %Country2%)
    <-reviewer-
  ReviewVtx
    -reviewFor->
  foreach y: ProductVtx
    -producer->
  ProducerVtx(country = %Country1%)
 and
   (y -type-> TypeVtx)
into table T1

select top 10
  id, count(*) as groupCount
from table T1
  group by id
  order by groupCount desc
```

Figure 7. Berlin Query 1 - *Select the top 10 most discussed products categories of products from Country 1 based on reviews from reviewers from Country 2.*

Set Labels:

$$\begin{aligned} q = V_1(\varphi_{V_1}) - E_1(\varphi_{E_1}) \rightarrow \cdots \rightarrow \text{def } X : V_i(\varphi_{V_i}) - \ldots \\ X - \cdots - E_{n-1}(\varphi_{E_{n-1}}) \rightarrow V_n(\varphi_{V_n}) \end{aligned} \tag{6}$$

$$\begin{aligned} q = V_1(\varphi_{V_1}) - E_1(\varphi_{E_1}) \rightarrow \cdots \rightarrow V_i(\varphi_{V_i}) - \ldots \\ X \subseteq V_i(\varphi_{V_i}) - \cdots - E_{n-1}(\varphi_{E_{n-1}}) \rightarrow V_n(\varphi_{V_n}) \end{aligned} \tag{7}$$

Equation 6 presents a path query with a set label X for vertex step V_i, which is then referenced in vertex step $i + j$. The path query will match all steps using the normal semantics defined in Equation 5, including for step $i+j$. This makes the path query described in Equation 6 semantically equivalent to the query in Equation 7.

In Equation 7 for step $i + j$, we have specified the same vertex type V_i and same condition φ_{V_i} as for step i. The set X of matching vertices can be a subset of the V_i vertices since by definition, it has been culled by the intervening j steps.

Element-wise Labels:

$$\begin{aligned} q = V_1(\varphi_{V_1}) - E_1(\varphi_{E_1}) \rightarrow \cdots \rightarrow \text{foreach } x : V_i(\varphi_{V_i}) - \ldots \\ x - \cdots - E_{n-1}(\varphi_{E_{n-1}}) \rightarrow V_n(\varphi_{V_n}) \end{aligned} \tag{8}$$

Equation 8 describes a path query that contains an *element-wise* label x for vertex step V_i. The label is then referenced in step $i+j$. The semantics of this query are more restrictive than the set label query described in Equation 6.

In fact, the subgraph patterns matched by Equation 6 are a superset of those matched by Equation 8.

We define $q(i)$ as the partial path query match up to step i of query q. Equation 5 provides the definition of a full path match up to step n of the query. Restricting its scope to step i provides the definition of a partial query match. $V(q(i))$ corresponds to the set of vertices matched up to step i, while $E(q(i))$ corresponds to the set of edges matched up to the same step.

An element-wise label matches each individual vertex (or edge) in the set matched on step i to the elements on step $i+j$. More precisely (and without loss of generality), if vertices ϑ and ψ are members of the set of vertices matched up to step i:

$$\begin{aligned}\vartheta, \psi \in V(q(i)) = \{&v_i \in V_i(\varphi_{V_i}) \wedge (v_1, v_2) \in E_1(\varphi_{E_1}) \wedge \\ &\cdots \wedge (v_{i-1}, v_i) \in E_{i-1}(\varphi_{E_{i-1}}) \wedge \\ &v_1 \in V_1(\sigma_{V_1}) \wedge v_2 \in V_2(\sigma_{V_2}) \wedge \ldots \\ &v_{i-1} \in V_{i-1}(\sigma_{V_{i-1}})\} \quad (9)\end{aligned}$$

Then ϑ matches step $i+j$ if $\vartheta \in V(q(i+j))$ and $\vartheta \in V(q(i))$. That is the exact vertex instance ϑ appears in both step i and step $i+j$ in the path.

A set label would match a path with ϑ in step i and ψ in step $i+j$, if ψ satisfies the conditions for step $i+j$ and ϑ satisfies the conditions for step i.

Less formally, a set label *can* match a cycle, while an element-wise label *will only* match a cycle.

3) Multi-path Queries: Linear path queries with disjunctions in step conditions and type matching as described in Section II-B4 cannot express patterns that comprise multiple paths in a single graph (i.e. contain branches). Figure 8 is a graphical representation of the first Berlin query. Notice there is no linear path through all vertices, as the multiple edges into and out of the *product vertex* create a branch point. Multi-path queries are a straightforward extension that enables the expression of composite graph patterns.

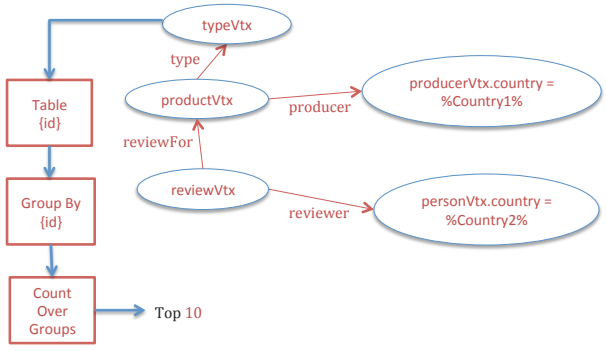

Figure 8. Berlin Query 1 in graphical form

We provide two ways of composing simple path queries into a multi-path query: *and* composition and *or* composition. Or composition is straightforward to describe, given two simple path queries q_1 and q_2 their composition q_1 or q_2 matches the union of the subgraphs described by the result of q_1 and q_2. That is:

$$\begin{aligned}V(q_1 \text{ or } q_2) &= V(q_1) \cup V(q_2) \\ E(q_1 \text{ or } q_2) &= E(q_1) \cup E(q_2)\end{aligned}$$

The *and* composition of two queries is only well defined if the two simple path queries share a label (set or element-wise). Let q_1 define a label ℓ for step i, if q_2 references label ℓ at step j then by Equations 6 and 8 the path match discovered on q_1 up to step i is incorporated in step j on q_2 together with q_2's own path match up to step $j-1$.

$q(i)$ is defined to be the partial path query match up to step i of query q. Using this definition, q_2's step j must satisfy the path matched by $\ell \wedge q_2(j-1)$.

```
graph ProductVtx(id = %Product1%) <-[ ]- [ ]
```

Figure 9. *Query - return subgraph of all reviews and offers of Product 1*

4) Type Matching and Path Regular Expressions: A more powerful query mechanism appears when we want to follow a path in which there is freedom in which kinds of edges or vertices to choose for each step. Figure 9 illustrates an example query in which we return the subgraph of all offers and reviews of *Product1*. The syntax "[ ]" indicates a meta-variable that can match any edge or vertex type. In this case, edge types *product* and *reviewFor* and vertex types *OfferVtx* and *ReviewVtx*. We refer to steps where multiple edge or vertex types may match as *variant steps*.

More precisely, the syntax "[ ]" indicates that for a particular step any vertex type or edge type that matches should be followed. We consider step i with a variant type for edges:

$$V_{i-1} - E_{i-1} \rightarrow V_i - [\,] \rightarrow V_{i+1} \quad (10)$$

In this case, step i of the path query can be satisfied by any edge type that has source vertex type V_i and target vertex type V_{i+1}. If $E(i-1)$ is the set of edges in the path query that match up to step $i-1$, then the set of edges matching for step i corresponds to the union of edges of matching types (using the notation specified in Section II-A1):

$$E(i) = \bigcup_i E_i(V_i, V_{i+1}) \quad (11)$$

The definition of variant query step i when we have variation over the vertex types is analogous.

Conditional expressions for variant query steps are not allowed in the query due to the fact that there is no guarantee that attributes are common across multiple matching vertex or edge types.

Interaction of Labels with Type Matching Steps: It is possible to label a type matching step (outside of a path regular expression). The reason to do so is to be able to write queries that are *type independent*, that is the query specifies a structural pattern in the graph that can be potentially matched by several vertex and edge types.

$$\text{def } X : [\,] - [\,] \rightarrow X \tag{12}$$

Equation 12 illustrates a purely structural query enabled by the combination of type matching and labels. In this case, it specifies a path of length one that starts with any type of vertex, traverses a single edge and must end with the same type of vertex. The same semantics apply to foreach labels.

Since all elements in GraQL are strongly typed, the use of labels for type matching steps must also conform to this restriction. For this reason, the use of a label from a type matching further down the query path carries the type of each matching element (vertex or edge). That is a label X that corresponds to a vertex of type V_1 will only match a vertex of the same type downstream, it cannot match a vertex of type V_2, thus the type of the label becomes bound at matching time. A type matched label expands into a *set* of labels (X, X', X'', ...), an *independent* one for each matching type.

Query Path Regular Expressions: A very general query capability that our design provides is the specification of paths using regular expressions over variant steps. Figure 10 presents an example of the use of regular expressions in specifying query paths. In this case, the path starts at a concrete vertex type (VertexA) with specific conditions (conditionsA), traverses *one or more* (+ regular expression operator) edge/vertex pairs and must end up in a vertex of concrete type VertexB satisfying specific conditions (conditionsB). Our design supports the traditional regular expression operators to specify zero or more times ("*") as well as specific repetition counts ("{10}").

```
graph VertexA(conditionsA) -
  { [] -> [] - [] }+ ->
  VertexB(conditionsB)
```

Figure 10. Regular expression query over variable path lengths

C. Query Results

The previous sections have detailed how queries in GraQL are composed and what the different language components mean with respect to the graph being queried. We now specify how the results of a query can be represented, in particular given the duality of our data model between a relational view and a graph view.

Given a simple path query q with n steps, we define an extension to the standard SQL `select` operation which enables the specification of a matching subgraph as the query, as well as the output of these results as a *named* subgraph or a table. The extensions are illustrated in Figure 11 for the named subgraph case.

```
select *
from graph V0 - E0 -> V1 - E1 -> ... -> Vn
into subgraph resultsG

select V0, Vn
from graph V0 - E0 -> V1 - E1 -> ... -> Vn
into subgraph resultsBE
```

Figure 11. Capturing results of a graph query into a subgraph

In Figure 11 the "`select *`" syntax indicates the selection of *all* matching vertices and edges of the subgraph into a named entity called "`resultsG`", while the second query selects only the vertices matching the first and last step of the query into a named subgraph called "`resultsBE`". The output steps (V0, Vn) must be unambiguous to be used in the "select" statement, if they are not then labels can be used to disambiguate them.

Figure 12 illustrates how the results set of a query (`resQ1`) can be used to seed the second query as its first vertex step by using the "." notation. In this case, the vertices matched by the last step of the first query are used to *restrict* the source vertices for the first step of the second query.

```
select Vn
from graph V0 - E -> Vn
into subgraph resQ1

select *
from graph resQ1.Vn(condQ1) -> ...
into subgraph resQ2
```

Figure 12. Final vertex set of a simple query used in a subsequent query statement

More precisely, the full subgraph result of a query of n steps is the set of vertices and edges that satisfy the query up to the final step, defined as $q(n)$ according to the notation in Section II-B3. A selection of certain vertices or edges of the subgraph corresponds to extracting those from the full matching subgraph and representing them as a (possibly disconnected) subgraph.

1) Results as Tables: Given the duality between tabular and graph views of the data, it is natural to consider how to represent the results of a graph query as tables. This enables further post-processing of those results using powerful non-graph, relational operations. We propose a couple of mechanisms to integrate graph results into standard SQL relational expressions.

Figure 13 illustrates the proposed syntax for representing the full matching subgraph of a simple path query as a new table (`resultsT`). In this case, each row has *all* the attributes of all entities involved in the query path. The table

```
select *
from graph Vertex1(conditionsV1) -
  ... -> VertexN(conditionsVN)
into table resultsT
```

Figure 13. Representing the whole matching subgraph as a table

will have one row for each matching path in the subgraph (with possible repetitions of subpaths).

```
select V0.a, EA.c, V1.d
from graph V0(conds0) - EA(condsA) -> ...
  V1(conds1)
into table resultsPT
```

Figure 14. Representing resulting subgraph elements as a table

Figure 14 illustrates the case where the user does not want the full subgraph as a table, but only a part of it. In this case, the columns of the table will correspond to `V0.a`, `EA.c` and `V1.d`, with one row per matching subpath in the results subgraph (repetitions are possible).

With respect to multi-path queries, their interpretation as a table follows the same rules as for a single path query. Each row will consist of columns matching the selected (possibly all) attributes of the subgraph entities, with the possibility of rows having `NULL` entries for attributes that are not part of certain paths in the multi-path query.

Table I summarizes the core SQL operations on tables supported by GraQL. For more details on SQL operations and their semantics see [12].

SQL operation	Description
`select`	Selection and projection operations
`order by`	Sorting operation
`group by`	Group together rows with the same key
`distinct`	Select distinct rows of a table
`count`	Count row instances in a table
`avg`	Average numeric values across row instances
`min`	Return "smallest" value of a column across row instances
`max`	Return "largest" value of a column across row instances
`sum`	Return summation of numeric values across row instances
`top` *n*	Return *top n* rows of a table
`as` *x*	Used to alias entity names in relational operations

Table I
TABLE OPERATIONS SUPPORTED BY THE RELATIONAL PART OF GRAQL

III. IMPLEMENTATION AND PERFORMANCE CONSIDERATIONS

As mentioned in Section I our target system is a cluster of high-performance servers ("compute nodes") with ample DRAM memory connected via a high speed network such as InfiniBand.

Our database system is named GEMS (Graph Engine for Multi-threaded Systems) and it is a second-generation system intended to target in-memory attributed graph data. The first-generation GEMS system targeted graphs represented as RDF triples and used the SPARQL query language.

Data sources for the in-memory GEMS database are assumed to reside on a high performance parallel filesystem accessible to all compute nodes on the cluster, for purposes of data ingest and eventual output to files.

The GEMS database system is composed of the following components:

1) Clients: clients can range from a simple command-line interface to web-based front-ends.
2) Server: the server centralizes access to the database system in order to provide access control, distinct user accounts, as well as a central metadata repository (catalog) of all existing database objects (tables, vertices, edges). The catalog contains updated information on the *sizes* of those objects (e.g. how many rows in table? how many vertex instances of certain type?).
3) Backend cluster: the backend cluster supports the high-performance, massively parallel execution of graph and tabular queries over the database, which is primarily resident on the aggregated memory of the compute nodes.

A *GraQL script* is a text file with a series of data definition, data ingest and query commands written in GraQL with the intent of obtaining insight via queries into the underlying GEMS graph database. Data definition and ingest commands can be assumed to execute atomically with respect to subsequent query commands.

Let $\Omega = q_1, q_2, \ldots, q_n$ represent the GraQL script in question. Each q_i represents an individual GraQL query *command* as defined in Section II-B. An individual q_i could be a simple path query, a multi-path query or a relational table operation. The *output* of an individual q_i can be used as input to subsequent individual queries as described in Section II-C.

A GraQL script is parsed and compiled into a high-level binary intermediate representation (IR) that is a convenient mechanism for moving the query script from the front-end portion of the GEMS system to the backend for execution.

A. Static Query Analysis

Given the availability of a metadata catalog on the GEMS front-end server, GraQL scripts can be statically checked for correctness as well as limited levels of *feasibility* of the query (e.g. will the query result be empty?).

Correctness checks include a number of different type checking issues: is the query comparing an attribute with a constant (or other attribute) of the wrong type? (e.g. comparing a date to a floating-point number); is the query using an entity of *correct* type for certain operations? (e.g. a table name should be used when a table is required,

rather than a vertex type name); is a path query correctly formulated?

These are a number of possible query checks that can be computed in a fully static manner without having access to the real data in the database. The only requirement is access to the metadata describing the database's entities: tables, vertices and edges.

B. Dynamic Query Analysis, Planning & Optimization

Once the query has been statically analyzed and deemed correct in the type checking sense, the question then becomes how to execute it in the shortest period of time possible by effectively utilizing the backend cluster's capabilities.

Once the binary high-level representation of the query has been transmitted to the cluster environment, further analysis can be performed with respect to *dynamic* properties of the data (which may or may not be available on the front-end catalog). Examples of these properties could be number of instances of vertex and edge types, as well as statistical properties of the degree distribution of a vertex type with respect to an edge type (e.g. how many outgoing edges of type E_1 are there for instances of vertex type V?).

A fundamental data structure that we use in the GEMS cluster backend is the *edge index*. The edge index lets us perform a step of a graph path query of the form $S(\sigma_s) - E(\sigma_e) \rightarrow T(\sigma_t)$, where S corresponds to the source vertex type, E to the edge type and T to the target vertex type (the σ expressions represents conditions on each type).

For performance considerations, we not only create an edge index in the lexical direction declared by the user $S - E \rightarrow T$, but also in the reverse direction $T - E \rightarrow S$ (when memory space on the cluster is available). The existence of both forward and reverse indices enables significant flexibility on how to execute a path query: the execution is not restricted to the forward-looking lexical representation of the path query in GraQL.

With the underlying knowledge of the existence of bidirectional edge indices, we can then formulate path query planning as a series of decisions on which *order* to traverse the edge indices indicated by the query.

1) Multi-statement GraQL scheduling & planning: Given a multistatement GraQL script $\Omega = q_1, q_2, \ldots, q_n$, and the explicit representation of outputs and inputs for each query via the use of the "`into subgraph` and "`into table`" expressions, we can build a multi-statement dependence representation.

This representation enables the query planner to determine whether two separate query statements q_i and q_j can be executed in parallel (if there are enough processing and memory resources on the cluster), or need to be executed in sequence. Pipelined execution of dependent query statements can also be considered to reduce the amount of space needed to materialize intermediate results.

IV. Conclusions & Future Work

We have presented the design of the GraQL query language and its associated GEMS attributed graph data model. We have discussed the key features of GraQL/GEMS including the key notion of a graph view built on top of tabular data sources, strongly typed attributes and entities, different variants of graph pattern matching, as well as flexible manipulation of query results as subgraphs and tables. We are currently working on the implementation and realization of this design in the context of the GEMS database system. We expect that our design will provide a high performance, yet flexible interface to query massive graph databases stored on a distributed cluster.

Acknowledgment

This work was performed as part of the High-Performance Data Analytics (HPDA) program at the Pacific Northwest National Laboratory (PNNL).

References

[1] R. Angles and C. Gutierrez, "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1:1–1:39, Feb. 2008.

[2] V. Kolomičenko, M. Svoboda, and I. H. Mlýnková, "Experimental Comparison of Graph Databases," in *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, ser. IIWAS '13. New York, NY, USA: ACM, 2013, pp. 115:115–115:124.

[3] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader, "A Performance Evaluation of Open Source Graph Databases," in *Proceedings of the First Workshop on Parallel Programming for Analytics Applications*, ser. PPAA '14. New York, NY, USA: ACM, 2014, pp. 11–18.

[4] C. Joslyn, S. Choudhury, D. Haglin, B. Howe, B. Nickless, and B. Olsen, "Massive Scale Cyber Traffic Analysis: A Driver for Graph Database Research," in *First International Workshop on Graph Data Management Experiences and Systems*, ser. GRADES '13. New York, NY, USA: ACM, 2013, pp. 3:1–3:6.

[5] M. Arenas and J. Pérez, "Querying Semantic Web Data with SPARQL," in *Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '11. New York, NY, USA: ACM, 2011, pp. 305–316.

[6] S. Sakr, S. Elnikety, and Y. He, "G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, ser. CIKM '12. New York, NY, USA: ACM, 2012, pp. 335–344.

[7] A. Khan and S. Elnikety, "Systems for big-graphs," *Proc. VLDB Endow.*, vol. 7, no. 13, pp. 1709–1710, Aug. 2014.

[8] J. Weaver, V. G. Castellana, A. Morari, A. Tumeo, S. Purohit, A. Chappell, D. Haglin, O. Villa, S. Choudhury, K. Schuchardt, and J. Feo, "Toward a data scalable solution for facilitating discovery of science resources," *Parallel Computing*, vol. 40, no. 10, pp. 682 – 696, 2014.

[9] V. Castellana, A. Morari, J. Weaver, A. Tumeo, D. Haglin, O. Villa, and J. Feo, "In-memory graph databases for web-scale data," *Computer*, vol. 48, no. 3, pp. 24–35, Mar 2015.

[10] A. Morari, V. Castellana, O. Villa, A. Tumeo, J. Weaver, D. Haglin, S. Choudhury, and J. Feo, "Scaling semantic graph databases in size and performance," *Micro, IEEE*, vol. 34, no. 4, pp. 16–26, July 2014.

[11] C. Bizer and A. Schultz, "The Berlin SPARQL Benchmark," *International Journal On Semantic Web and Information Systems*, 2009.

[12] C. J. Date and H. Darwen, *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

APPENDIX

A. Table Declarations for Berlin queries

```
create table Types(
   id          varchar(*),
   type        varchar(*), // ProductType
   label       varchar(*),
   comment     varchar(*),
   subclassOf  varchar(*), // Types.id [1..N]
   publisher   varchar(*),
   datet       date
)

create table Features(
   id          varchar(*),
   type        varchar(*), // ProductFeatures
   label       varchar(*),
   comment     varchar(*),
   publisher   varchar(*),
   datef       date
)

create table Producers(
   id          varchar(*),
   type        varchar(*), // Producer
   label       varchar(*),
   comment     varchar(*),
   homepage    varchar(*),
   country     varchar(*),
   publisher   varchar(*),
   datep       date
)

create table Products(
   id                  varchar(*),
   type                varchar(*), // Product
   label               varchar(*),
   comment             varchar(*),
   producer            varchar(*), // Producers.id
   propertyNumeric_1   integer,
   propertyNumeric_2   integer,
   propertyNumeric_3   integer,
   propertyNumeric_4   integer,
   propertyNumeric_5   integer,
   propertyText_1      varchar(*),
   propertyText_2      varchar(*),
   propertyText_3      varchar(*),
   propertyText_4      varchar(*),
   propertyText_5      varchar(*),
   publisher           varchar(*),
   datep               date
)

create table ProductTypes(
   // A product has 1 to N types
   product varchar(*), // Products.id
   type    varchar(*)  // Types.id
)

create table ProductFeatures(
   // A product has 9 to 22 features
   product varchar(*), // Products.id
   feature varchar(*)  // Features.id
)

create table Vendors(
   id          varchar(*),
   type        varchar(*), // Vendor
   label       varchar(*),
   comment     varchar(*),
   homepage    varchar(*),
   country     varchar(*),
   publisher   varchar(*),
   datev       date
)

create table Offers(
   id              varchar(*),
   type            varchar(*), // Offer
   product         varchar(*), // Products.id
   vendor          varchar(*), // Vendors.id
   price           float,
   validFrom       date,
   validTo         date,
   deliveryDays    integer,
   offerWebPage    varchar(*),
   publisher       varchar(*),
   dateo           date
)

create table Persons(
   id          varchar(*),
   type        varchar(*), // Person
   name        varchar(*),
   mailbox     varchar(*),
   country     varchar(*),
   publisher   varchar(*),
   datep       date
)

create table Reviews(
   id              varchar(*),
   type            varchar(*), // Review
   reviewFor       varchar(*), // Products.id
   reviewer        varchar(*), // Persons.id
   reviewDate      date,
   title           varchar(*),
   text            varchar(*),
   ratings_1       integer,
   ratings_2       integer,
   ratings_3       integer,
   ratings_4       integer,
   publisher       varchar(*),
   dater           date
)
```