A Knowledge Graph Data Model and Query Language

Kenneth Baclawski

Northeastern University

Abstract

An increasing amount of data is now available on public and private sources. Furthermore, the types, formats and number of sources of data are also increasing. The data sources have many different levels and types of structuring. Techniques for extracting, processing and analyzing such data have been developed in the last few years for managing this bewildering variety based on a structure called a knowledge graph. In this article, a new knowledge graph data model featuring formal reification is introduced and specified mathematically. This data model has many advantages compared with existing data models that have been used for representing graph structures. One important advantage is that all graph edges are reified, which can reduce the cost and complexity of storage and retrieval for knowledge graphs that have rich semantics, such as provenance, units of measure, and uncertainty specifications. In spite of the added capabilities of this knowledge graph data model, one can efficiently store knowledge graphs in existing triple stores, and existing tools can be used with only minor modifications. This article also introduces a new data language, called KGSQL, which is specifically designed for the new knowledge graph data model. Both the syntax and the denotational semantics of KGSQL are specified formally.

Introduction

THE NOTION OF A KNOWLEDGE GRAPH (KG) has emerged in the last few years to be an important semantic technology and research area. As structured representations of semantic knowledge that are stored in a graph, KGs are lightweight versions of semantic networks that scale to massive datasets such as the entire World Wide Web. Industry has devoted a great deal of effort to the development of knowledge graphs, and they are now critical to the functions of intelligent virtual assistants such as Siri and Alexa. Some of the research communities where KGs are relevant are Ontologies, Big Data, Linked Data, Open Knowledge Network, Artificial Intelligence, Deep Learning, and many others.

A KG is defined to be a labeled multigraph that has the following mathematical definition (Baclawski et al., 2020):

A node- and edge-labeled multigraph is an 8-tuple $(V, E, s, t, \Sigma_V, \Sigma_E, \ell_V, \ell_E)$ such that

1. V is a set of nodes, and E is a set of edges.
2. The functions $s: E \to V$ and $t: E \to V$ specify the source and target nodes of the edges.
3. Σ_V is a set of node labels, and Σ_E is a set of edge labels.
4. The functions $\ell_V: V \to \Sigma_V$ and $\ell_E: E \to \Sigma_E$ specify the labels of the nodes and edges.

In the definition above V and E need not be disjoint. This led to my proposal that it would be advantageous if E were a subset of V (Baclawski, 2021). In other words, every edge of the graph should itself be reified as a node of the graph. This article elaborates my proposal, explaining the rationale for introducing a new data model and a new data language; and formally specifying them.

We begin in Section 2 with the terminology for knowledge graphs that will be used. The many advantages of reifying the edges of a graph as nodes are presented in Section 3. There are other data models that have been adapted for KGs as well as many data languages. There are too many to be adequately covered in this article, but only a few are directly relevant to the aims of the work developed in this article. These data models and languages are reviewed briefly in Section 4. Our new data model is called the *knowledge graph data model* (or more briefly, the *KG model*). It formally reifies all edges, so that it has the advantages described in Section 3. We also introduce a data language that is designed for the new KG model. Our new data language is called the *knowledge graph system query language* (KGSQL). Both our new data model and our new data language are presented in Section 5. The technical details of the formal specifications of the KG model and KGSQL are given in three appendices:

1. The denotational semantics of KGSQL is specified in Appendix A. Denotational (or compositional) semantics is useful for specifying how to program an implementation of a language such as KGSQL.
2. The KG model is specified in the Distributed Ontology, Modeling, and Specification Language^{TM} in Appendix B. The specification of the KG model uses the notion of an institution that was introduced by Goguen and Burstall as a means of systematizing and relating different logical systems (Goguen and Burstall, 1983). Institutions

depend on the mathematical notions of category and functor so a brief introduction to categories and functors is also presented.

3. The formal grammar of KGSQL is in Appendix C. A parser and compiler for KGSQL queries was implemented in Java to show that the KGSQL language is consistent. The parser and compiler are available on request from the author.

The article ends with a Conclusion and Acknowledgments.

Knowledge Graph Terminology

The most common terminology for KGs is taken from the Resource Description Framework (RDF). This framework was developed to represent metadata on the Web; however, it is now being used for representing information of any kind. DF is sometimes regarded as being a specific XML representation for data; but RDF is, in fact, a data model for graph data. There are many representation languages for RDF data, all of which use the same RDF data model, most of which do not use XML. For example JSON-LD uses JSON to represent KGs.

As its name suggests, RDF is used to represent information about resources. RDF can be used to specify properties of resources and relationships between resources. A *resource* is specified using either an IRI or a blank node. An IRI is a universally unique identifier of a specific resource. A blank node specifies that a resource exists without explicitly naming it. A property of a resource is specified with a literal which can be typed or can be tagged as being in a particular natural language. In RDF, properties and relationships are specified with *statements*. Each statement specifies a subject, a predicate, and an object. Because every statement consists of three components, statements are also called *triples*. The subject and predicate are resources, and the object can be either a resource or a literal. We will write RDF statements by using angle brackets. Note that RDF uses the same term for properties and relationships; both are called "properties."

The Web Ontology Language (OWL) is a family of languages for representing ontologies. OWL is built on RDF and adds many new features and distinctions to the notions in RDF. For example OWL distinguishes relationships from properties. The former are called ObjectProperties and the latter are called DatatypeProperties.

An important relationship between resources is the one that specifies the class of a resource. For example to specify that George is a person, one might use the statement <:George rdf:type :Person>. The colons are used to specify the prefixes of resources. IRIs can be very long, so it is useful to have a mechanism for declaring abbreviations. Note that a prefix can be the empty string. Because the type relationship is so common, we will use a simpler notation for it; namely, [:George :Person].

It is often useful to focus on a subset of a graph, especially when the graph is very large. The mathematical notion for this is called a subgraph. More precisely, a *subgraph* of a graph *G* is another graph *H* formed from a subset of the vertices and edges of *G*. The vertices of *H* must include all endpoints of the edges of *H*, but may also include additional vertices of *G*.

An *edge-induced subgraph* of a graph *G* is a subgraph that does not have any additional vertices, *i.e.*, it only has vertices that are endpoints of at least one edge. The RDF standard has a notion of a *named graph* that implements an edge-induced subgraph. The name of a named graph is a blank node or IRI that can be used in RDF statements. Within an RDF database, there can be any number of named graphs. Each RDF statement in an RDF database must specify the named graph to which the RDF statement belongs. Consequently, an RDF statement is stored using a 4-tuple or quad. In spite of this, an RDF database is usually called a "triple store."

Advantages of Reifying Statements

While RDF can be used to represent KGs, it is not a perfect match. This section presents a number of examples for which a data model where all statements are reified would have significant advantages.

Higher Order Relations

One of the controversial issues of RDF is that it only has native support for binary relationships. This issue has been addressed in more recent frameworks, and these are discussed in Section 4. In RDF relationships with higher arity (*e.g.*, records) must be synthesized using properties and binary relationships. For example suppose that one wishes to implement the classic Suppliers and Parts relational database using RDF ("Suppliers and Parts", 2021). This database has three tables: Supplier, Part, and Shipment. Each record in the Supplier and Part tables is implemented by assigning a unique IRI to each supplier and part, and by asserting an RDF statement for each non-primary key attribute. The only table that is problematic is the Shipment table. This table has three columns: one each for the supplier and part, and one for the quantity of the shipment. An RDF property cannot be used to represent this table because of the third column. For example suppose that the supplier IRI is :supplier8, the part IRI is :part25, the quantity shipped is 300, and the RDF property relating them is :shipment. Then the RDF statement <:supplier8 :shipment :part25> asserts that the supplier ships the part, but does not specify the quantity shipped. To specify the quantity shipped as well as any other attributes of the shipment, one must reify the RDF statement in some way. The standard technique for RDF reification is to assert the following statements:

<b rdf:type rdfs:Statement>
<b rdfs:subject :supplier8>
<b rdfs:predicate :shipment>
<b rdfs:object :part25>

where b is either a blank node or an IRI that is unique for the reified RDF statement. The quantity may now be specified using the RDF statement <b :quantity "300"^^xsd:int>.

While this solves the problem of higher arity relations, those who have used languages that natively support higher relationship arities find the RDF limitation to binary relationships to be awkward for a number of reasons. Aside from the inefficiency of expanding a single statement into four statements, queries are now more complicated, especially if one must deal with many relationships that have been reified.

Another complication of reifying an RDF statement is that there is no connection between a statement and a reification of the statement, as the two are independent. If some statements of a relationship have been reified while others have not been reified, then queries, updates and inference rules will be much more complicated. This significantly undermines one of the supposed advantages of the RDF data model; namely, the claimed ease with which one can introduce new properties and relations. By comparison, it is relatively easy to add new columns to a relational database table without any need to update any existing queries or updates.

Yet another problem with the reification of an RDF statement is that the domain and range of a property do not apply to the reification of statements whose rdf:predicate is that property. One can specify domain and range axioms for a reification with OWL, but not with RDF, and the OWL axioms are relatively complicated.

Literals

Another example of an issue with RDF is that literals are not resources, so one cannot specify properties of literals. For example ["Hello, world!" rdfs:Literal] is necessarily true, but one cannot assert this fact using an RDF statement. It is also not possible to specify the datatype of a literal using a statement, so a separate syntax is employed to specify the datatype of a literal. For example "15"^^xsd:int specifies the number fifteen. Similarly, the (human) language of a text string is specified with a separate syntax. For example "hello"@en specifies that the language of the string "hello" is English.

One way to specify properties of literals, other than the datatype or language, is to reify the literal. A reification of a literal is called a compound literal, and its type is rdfs:CompoundLiteral. The properties of a compound literal can include its datatype and language as well as other properties such as directionality (*i.e.*, left-to-right or right-to-left), unit of measurement, *etc*. The value of such a reified literal is specified by the rdf:value property. While this solves the problem of specifying properties of literals with RDF, there are now two different ways to specify a property value: directly with the literal or indirectly with the reification of the literal. The disadvantages of this situation are then similar to the ones already mentioned in Section 3.1.

Incidentally, RDF has a non-standard extension that allows literals to be used in any slot of a statement, including the subject and predicate slots. However, this does not resolve the problem of adding properties to literals. The same string might be in many languages, might have various measurement units dependent on context, and so on.

Incremental Development

It is a common practice to develop a complex information system by using an iterative process to provide incremental improvements (Darrin and Devereux, 2017). KGs can be very complex, and there are significant practical reasons for developing them incrementally, starting from a KG represented in RDF with a minimal schema or no schema at all, and later gradually improving the schema (Berg-Cross, 2021). However, one problem with such an approach is that improving the KG with an improved schema will often require a wholesale restructuring of the KG to a new version that is not backwardly compatible with earlier versions. Sections 3.1 and 3.2 give examples in which improving the schema requires changing the structure of the KG. For example, the initial KG might have statements with properties such as distances, speeds, and weights, expressed as raw numbers. Adding appropriate units to these statements requires designing a schema that allows for specifying the unit of measure for such statements, as well as other properties such as uncertainty and provenance. Reifying all statements allows one to incrementally add such properties to statements without any restructuring. While the KG model cannot entirely eliminate the need for KG restructuring during an incremental development process, it can make it much easier in many important cases.

Another advantage of the KG model is that it reduces the number of possible designs for incrementally adding properties to statements and collections of statements. While one can easily design an RDF schema that will allow one to add units of measure to a statement, there are several ways to do this, as noted in Section 3.2. This can hamper interoperability as well as incremental development efforts that wish to incorporate existing KGs (Berg-Cross, 2021). Since the KG model reifies edges, there are fewer designs for adding properties to statements, and they are more easily accommodated during incremental development and interoperation. For example, two KGs might use different properties for specifying a unit of measure, but the basic structure is the same. Again, the KG model does not entirely solve this problem, but it could make it easier to solve.

Graph Languages with Edge Quoting and Reification

While there are many graph query languages, only a few have mechanisms for simplifying edge reification. In this section we review these languages.

The Property Graph Query Language (PGQL) is a graph query language built on top of SQL. The purpose PGQL is to extend SQL to have graph pattern matching capabilities (PGQL, 2021). A *property graph* is a graph in the usual sense of nodes and edges, such that nodes and edges can have properties. A PGQL schema is similar to an SQL schema except that tables are either vertex tables or edge tables. Each edge table has a source table and a destination table. The difference between the PGQL and KG models is that in the KG model all statements, including properties, are reified. Another difference is that the type of a KG resource is specified with a statement, while in PGQL the type of a resource is the table it belongs to. So a KG resource can have many types, and a resource can change its type. Changing the table of an entity is not meaningful in SQL and PGQL.

The RDF-star is an unofficial draft data model that is intended to simplify specifying RDF reification (RDF-star, 2021). The corresponding data language is SPARQL-star. The feature that RDF-star adds to RDF is the ability to *quote* a triple. Quoted triples are specified in double angle brackets. Quoting a triple is not the same as the reification of the triple. If the same quoted triple appears multiple times, then all of them are necessarily the same. By contrast, the same triple can be reified more than once in the KG model, and each reification will have its own triple identifier. To associate an identifier with a quoted triple, one must specify the identifier with another triple. One can then use this identifier in other triples. While this does reify the triple, there is now a distinction between the quoted triple and its reification. Moreover, the property that is used to reify the quoted triple is not unique. Indeed, within the same RDF-star store one could reify the same triple using more than one property.

One can nest RDF-star triples to any depth. All of the triples in such an expression are quoted except for the outermost triple. One cannot have infinite nesting of RDF-star triples, so that it appears to be impossible to specify a circular structure. However, if one reifies the quoted RDF-star triples, then one can define circular structures.

The Knowledge Graph Data Model and System Query Language

A common feature of the issues with RDF discussed in Section 3 is the use of reification to resolve the issue. This suggests that a language where reification is provided in a seamless manner would have many advantages. While some languages have attempted to deal with this issue to some degree, as discussed in Section 4, it might be worthwhile to consider a more dramatic approach; namely, reify *every* statement and introduce a data language that leverages the reifications. We call this language KGSQL, and specify both the KG model and KGSQL in this section along with some applications.

The Knowledge Graph Data Model

We define the KG model by specifying a concrete realization of the notion of a knowledge graph as defined in Section 1. This realization uses ordinary set theory. The category theoretic specification is given in Appendix B.

We start by specifying the different kinds (or, more precisely, sorts) of the nodes that can be in a knowledge graph as follows:

- *GId* is the set of all possible global resource identifiers;
- *LId* is the set of all possible local resource identifiers;
- *Lit* is the set of literal strings;
- *Num* is the set of double-precision numbers;
- *Bool* is the set of Boolean values;
- $\perp$ denotes the undefined result; and
- *V* is the set of all possible variables.

It is assumed that the sets *GId, LId, Lit, Num, Bool,* and *V* are disjoint and do not contain $\perp$, and that *V* is infinite. The union $GId \cup LId$ is the set of resource identifiers and will be written *Id*. The union $Id \cup Lit \cup Num \cup Bool$ will be written *Res*.

A *knowledge graph* is a relation $G \subseteq Res \times Id \times Res \times Id$ such that if $(s, p, o, e), (s', p', o', e) \in G$ then $s{=}s'$, $p{=}p'$ and $o{=}o'$. In other words the fourth component is a unique column of the relation. The elements of a KG are called *statements*, and the components are called the *subject*, *predicate*, *object*, and *statement identifier*, respectively.

The set of all elements of *Res* that occur as one or more of the components of an edge of a knowledge graph *G* will be written *G.node*. The type of a node that is not a statement identifier is specified by a statement whose property is rdf:type. The type of a statement is its property (*i.e.*, its edge label), and it is not necessary to have an explicit edge in *G* that specifies this fact. A node can have more than one type. Table 1 shows the correspondence between the terms of a multigraph and the terms of a KG. In this table, *FinSet(S)* is the collection of all finite subsets of a set *S*.

Multigraph	Knowledge Graph
V	*G.node*
E	*G*
s	The subject of a statement
t	The object of a statement
Σ_V	*FinSet(G.node)*
Σ_E	*FinSet(G.node)*
ℓ_V	The types or properties of a node
ℓ_E	The properties of a statement

Table 1: Relationship between multigraph and knowledge graph terminology

The realization of the notion of a KG not only defines the KG model it also gives an example of an implementation; namely, one can implement a KG with a single relational table with four columns. The fourth column is the statement identifier. Another implementation is to store a KG in a triple store. This is possible because, in practice, triple stores actually store quads. The fourth component is the name of the named graph. The fourth component could store both the name of the named graph and the statement identifier by concatenating the name of the named graph with a unique identifier for the statement within the named graph, separating the concatenated strings with a special character. In other words the statement identifier includes the name of the named graph to which it belongs. Of course this strategy requires that the statement identifiers have a particular form.

The Query and Update Languages

We now discuss the syntax of KGSQL. The syntax was designed to be as similar as possible to SPARQL; indeed, most SPARQL commands should also be KGSQL commands. The most important additional feature is the ability to explicitly reference the identifier of any statement. KGSQL supports SELECT, ASK, CONSTRUCT, INSERT, and DELETE commands. The SELECT command finds all collections of edges in the KG that satisfy the WHERE clause and returns the selected variables. The ASK command is the same as the SELECT command except that it only returns whether there were any matching edges. The ASK command can be used as a subquery in the where clause of another command. The CONSTRUCT command is the same as the SELECT command except that the selected variables are used to specify a set of edges in a graph which are then returned. The INSERT command is the same as the CONSTRUCT command except that the constructed edges are stored in the KG database. The INSERT command can modify edges in the KG database so that the INSERT command is also an update command. The DELETE command is the same as the SELECT command except that the variables that are returned are the statement identifiers of the edges that are removed from the KG database.

The WHERE clause of any command consists of patterns and filters. A pattern specifies a subject, predicate and object, which can be variables or constants. A variable is indicated with an initial question mark. CONSTRUCT and INSERT commands use patterns to specify the edges to be constructed or inserted.

As in SPARQL, some abbreviations are supported. If several patterns have a common subject, then one can specify the subject followed by a sequence of predicate-object pairs separated by semicolons. If several patterns have the same subject and predicate, then they can be followed by a sequence of objects separated by commas. Unlike SPARQL, the subject and object can be bracketed expressions which specify an instance and its class, either or both of which can be a variable. The class can be a pipe-delimited sequence of classes, denoting a union of the classes. A union does not allow variables. A bracketed expression can specify a constant or variable in between the instance and the class (or class union), which represents the statement identifier of the type statement. The predicate can also be a bracketed expression, but it cannot specify a third slot in the middle. The bracketed expression of a predicate specifies the statement identifier and the property. In other words it is as if properties are classes whose instances are statements.

Patterns are a textual notation for specifying KGs. Another way to specify a KG is with a drawing of the nodes and edges. Since the edges are directed from the subject to the object, one uses arrows for edges. The property of an edge is shown next to the edge. Since edges are also nodes, it is possible for an edge to start at another (or the same) edge and also to end at an edge. For example the Suppliers and Parts example in Section 3.1 could be specified in the KG model as follows:

:supplier14 [?e :shipment] :part34 .
?e :quantity ["500" xsd:decimal] .

and it could be drawn using the diagram in Figure 1.

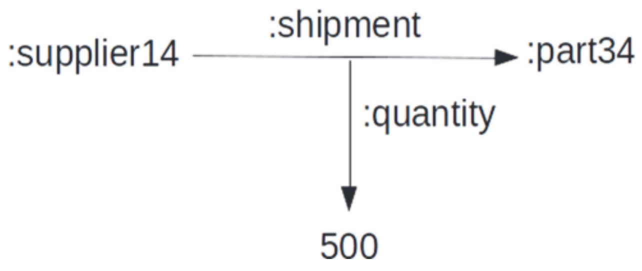

Figure 1: The Suppliers and Parts Example

The Suppliers and Parts database is unrealistic. In practice one shipment may include several parts but would always be to one customer on one date, organized as an invoice. So it would be better to make the customer the main attribute of a shipment as in Figure 2.

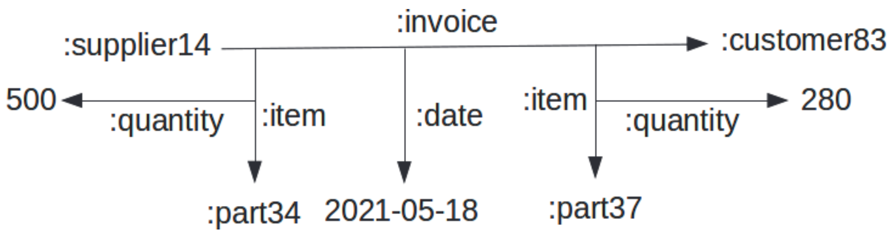

Figure 2: Example of an invoice

All the edges in Figure 2 are reified so one can easily add additional information about each one as needed. For example one could add context information such as provenance and uncertainty (Baclawski *et al*., 2018). As another example, one could add rationales which could be used for the purposes of explanation (Baclawski *et al*., 2019; Baclawski, 2020).

The reason for not allowing a middle slot in a bracketed expression for the predicate is subtle. In the KG model, each statement is an instance of its predicate. However, if the relationship between each statement and its predicate were always reified, then this would entail an infinite sequence of reifications. For example suppose that we have the statement :supplier8 :name "QWPNW". Let :e1 be its statement identifier. The predicate of the statement identified by :e1 is :name. So :e1 has the type :name. If this fact is reified, then the statement :e1 rdf:type :name would have a statement identifier. Suppose that :e2 is the statement identifier of :e1 rdf:type :name. Then the predicate of :e2 is rdf:type. If this fact is reified, then :e2 rdf:type rdf:type would also have a statement identifier, and so on, as drawn in Figure 3. The result is an infinite sequence of distinct statement identifiers. Presumably, one could find a way to manage such infinite sequences, since they all have the same simple form, but it is simpler to agree to omit them.

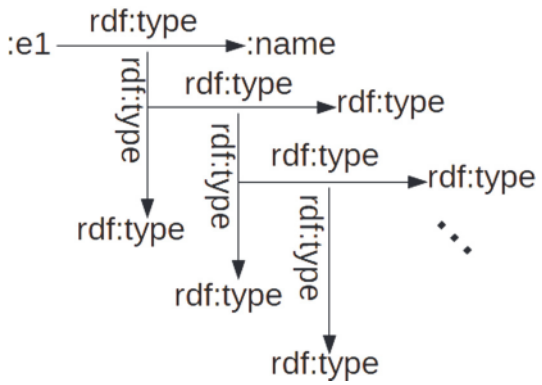

Figure 3: An infinite sequence of reifications

A pattern can specify a multiplicity for following paths in a KG. A multiplicity is one or two integers or asterisks, representing the minimum and maximum number of edges to follow. An asterisk represents an unlimited number of edges. If there is one integer or asterisk, then the minimum and maximum are the same. The multiplicity can be in one of two places in a pattern. If the multiplicity is after the second slot, then the multiplicity specifies that one should follow statement identifiers; and if the multiplicity is after the third slot, then the multiplicity specifies following objects.For example in the drawing in Figure 4, the pattern

:a prop ?x {6}

would set ?x to :b, and the pattern

:c prop {6} ?x

would set ?x to :d. This notation is analogous to the notation for array selection in programming languages where it would look something like this: x = c.prop[6].

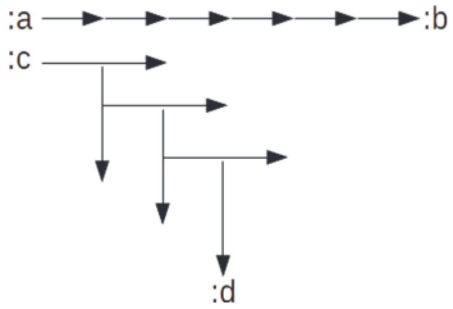

Figure 4: Path following in a knowledge graph

A negative multiplicity goes in reverse (*i.e.*, via the inverse relationship). For example, the pattern

?x prop :b {-6}

would set ?x to :a, and the pattern

?x prop {-6} :d

would set ? x to :c.

If the maximum or minimum is an asterisk (*i.e.*, unbounded), then the result is a transitive closure.

The formal syntax of the initial version of KGSQL is in Appendix C. Only the grammar is shown in Appendix C. The lexical rules were omitted for simplicity. The full grammar is available at kgsql.org/KGSQL.g4. The syntax notation is that of Antlr (Parr, 2014). Not all features of SPARQL were included in the initial version KGSQL, since the initial version is only a proof of concept. Other features of SPARQL will be added in later versions.

Sequences

One of the advantages of reifying all statements is that one can construct sequences (also called linked lists) much more easily and efficiently. Moreover, one can specify sequences with any property as the sequence property. In RDF, a sequence is specified using the built-in properties rdf:first and rdf:rest, and the built-in resource rdf:nil. For example the sequence (:Kim :Greer :Qing) is specified with the following statements:

```
<a rdf:first :Kim>
<a rdf:rest b>
<b rdf:first :Greer>
<b rdf:rest c>
<c rdf:first :Qing>
<c rdf:rest rdf:nil>
```

In KGSQL, a sequence is specified like this:

```
c [a :prop] :Kim .
a [b :prop] :Greer .
b [c :prop] :Qing .
```

where :prop could be any property. Note that this sequence is circular and that no notion of nil is necessary for nonempty sequences. Of course one does require a way to specify an empty sequence, such as the built-in resource kgsql:empty. In KGSQL, the notation for this sequence is [(:Kim :Greer :Qing) :prop]. The drawing for this sequence is shown in Figure 4.

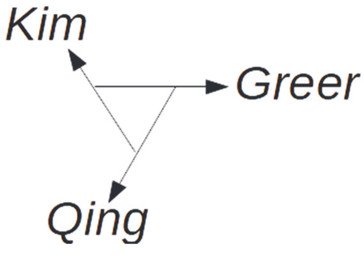

Figure 4: Example of a three-element sequence

Comparison with SQL

In this section, we discuss how KGSQL differs from SQL, or more precisely comparing the KG model with the relational model. Many comparisons of triple stores and the relational databases have been published, such as (Comparison, 2013). These comparisons generally focus on pragmatic issues of the use of different products rather than the conceptual differences between the models. While a pragmatic focus is valuable, it might also be worthwhile to consider the conceptual differences, and we do so in the following.

One major conceptual difference between KGs and the relational model is the way that entities are conceptualized. In a KG, the entities are the nodes in the graph. In RDF entities are called resources and are specified with IRIs. By contrast, the entities in the relational model are records (also called rows or tuples). A record is specified with its primary key. To see why this distinction is more than just a question of data representation, consider how queries for the two models are mapped to the variables of predicate logic. For the relational model, the variables represent records. In other words, when one quantifies in the relational model (using either existential or universal quantification), the quantification is over the records of a specific relation (or table). This is fundamental to any relational query language. To borrow a term from programming languages, relational variables are strongly typed. Indeed, in SQL, by default, the name of a variable is the name of a table; non-default variable names are necessary only if a query is quantifying over the same table more than once.

By contrast the variables of a KG query vary, in principle, over all possible nodes of a graph. To the extent that a variable is typed at all, the type constraint is just one more statement which is no more fundamental than any other constraint. In particular a type constraint need not be imposed at all, and if it is, then it is imposed explicitly. The relational model has no such flexibility.

Some relational database systems maintain a rowid (also called a "rid") that uniquely identifies each record in a table. This appears to be analogous to a statement identifier. However, while a rowid can be accessed in a query, it is set and maintained by the database system. More importantly, the database can change a rowid when a record is updated and can reuse the rowid of a deleted record.

Another way to look at the distinction between the KG and relational models is to treat a KG as being a single relation whose records are the nodes and whose primary key is the node identifier. This single relation has infinitely many multi-valued columns, one for every possible property or relationship. This relation is not quite a relation in the classical sense because it has multi-valued columns, but many relational databases support such columns. In any case, the example is only a conceptual one as it is not at all practical, and it completely omits the reification of statements that is fundamental to KGSQL.

A more practical and complete implementation of a KG with a relational database is to use a four-column table to represent the statements, with an additional column to specify either a named graph identifier (for the RDF data model) or a statement identifier (for the KG model). Let T(G) be the table for a knowledge graph G. This implementation is more promising than the previous one, and some products use this technique. However, it would be misleading to view a KG as being T(G) or even being analogous to T(G) as is commonly mentioned, such as in the Wikipedia article (SPARQL, 2021). The difficulty with this analogy is that from a relational point of view the entities of T(G) are the *records* of T(G), not the entries in any of the columns of one of the records. This confusion is especially significant for the RDF data model because RDF statements, which correspond to the records of T(G), are never entities of the RDF data model. The distinction is not merely conceptual as it affects how one programs queries for the RDF model compared with relational queries. In the relational model one is iterating over records in tables while in the RDF model one is iterating over resources. It is somewhat less significant for the KG model because the records of T(G) are always KG nodes and hence are entities, but there will also be many other entities, so programming a KGSQL query will also differ from programming an SQL query.

In summary the KG and relational models for data differ not only pragmatically but also conceptually. The two models have different notions of what an entity is in the model. As a result, the two models tend to have different approaches to design, development, and employment. While the RDF data model and the KG model are similar, the differences between them are also likely to result in different approaches to design, development and employment.

Conclusion and Future Work

We have developed a new data model specifically designed for knowledge graphs. In this regard the development of the KG model is analogous to the development of the relational model by Codd and many others, which was designed for relational (tabular) data (Codd, 1970). While the KG model is designed for KGs, it is effective for representing data of other kinds, such as hierarchical and tabular data. We have also developed KGSQL, a data language designed for the KG model. KGSQL helps to resolve some of the disadvantages of existing data languages for KGs. Both the KG model and the KGSQL language are formally specified, with the details given in the appendices.

The KG model and KGSQL are concerned only with data, not the schema. This has the advantage that the data and schema are separate concerns. In principle a KG could have more than one schema; for example, different schemas might be used by different communities for their own purposes. That said, an interesting future direction for the work on the KG model would be to develop a schema language for specifying semantics. Ideally, the schema language would also use the KG model, so that the schema could be written in KGSQL.

One of the most powerful insights of the KG model is that properties (including relationships) are classes whose instances are statements. In OWL, properties and classes are disjoint. In the KG model not only are properties and classes not disjoint, but the properties are a subset of the classes. This idea makes it possible for properties to be domains and ranges of other properties, thereby allowing more opportunities for specifying the semantics of data expressed as KGs. An interesting future project would be to develop this idea both practically and formally.

Acknowledgments

The author would like to thank Gary Berg-Cross, Ravi Sharma, Janet Singer, Ram D. Sriram, and Andrea Westerinen for many helpful comments that improved this article.

References

K. Baclawski (2020). Decision Rationales as Models for Explanations. In *J. Wash. Acad. Sci.*106(4):107-124.

K. Baclawski (2021). Introduction to KGSQL: A Knowledge Graph System Query Language, July 7 2021. Retrieved 1 October 2021 from https://bit.ly/3xocWEj.

K. Baclawski, M. Bennett, G. Berg-Cross, C. Casanave, D. Fritzsche, J. Ring, T. Schneider, R. Sharma, J. Singer, J. Sowa, R.D. Sriram, A. Westerinen and D. Whitten (2018). Ontology Summit 2018 Communiqué: Contexts in Context. In *Journal of Applied Ontology*13(3):181-200. IOS Press, The Netherlands. (July, 2018)

K. Baclawski, M. Bennett, G. Berg-Cross, D. Fritzsche, R. Sharma, J. Singer, J. Sowa, R.D. Sriram, M. Underwood and D. Whitten (2019). Ontology Summit 2019 Communiqué: Explanation. In *Applied Ontology*. IOS Press, The Netherlands. DOI: 10.3233/AO-200226.

K. Baclawski, M. Bennett, G. Berg-Cross, D. Fritzsche, R. Sharma, J. Singer, J. Sowa, R.D. Sriram, M. Underwood, and D. Whitten (2020). Ontology Summit 2020 Communiqué: Knowledge Graphs. *Applied Ontology*, 16(2):229-247, April 2020.

Comparison of triple stores vs relational databases (2013). Retrieved 2 October 2021 from https://bit.ly/3ooqIoV.

G. Berg-Cross (2021). Introduction to Harmonizing Definitions and the EnvO Ontology. Retrieved 1 June 2021 from https://bit.ly/3pnkwdG.

E. Codd (1970). A Relational Model of Data for Large Shared Data Banks. In *Communications of the ACM* 13 (6) 377-387.

M. Darrin and W. Devereux (2017). The agile manifesto, design thinking and systems engineering. In *Annual IEEE International Systems Conference (SysCon)*, pages 1-5. IEEE.

J. Goguen and R. Burstall (1983). Introducing institutions. In *Proc. Carnegie Mellon Workshop on Logic of Programs*, volume 164, pages 221-256.

D. Jurafsky and H.J. Martin (2000). Speech and language processing: an introduction to natural language processing, computational linguistics, and speech recognition. Upper Saddle River, N.J.: Prentice Hall. ISBN 978-0-13-095069-7.

T. Parr (2014). Antlr website. Retrieved 2 October 2014 from https://www.antlr.org.

The Property Graph Query Language (PGQL) website (2021). Retrieved 2 October 2021 from https://pgql-lang.org/.

RDF-star and SPARQL-star (2021). Draft Community Group Report 01 October 2021. Retrieved 2 October 2021 from https://bit.ly/3F6ORX3.

Suppliers and Parts (2021). Wikipedia article. Retrieved 2 October 2021 from https://bit.ly/3B4wv6p.

Wikipedia article on SPARQL (2021). Retrieved 2 October 2021 from https://bit.ly/3B4eXHC.

Appendix A Denotational Semantics

A KGSQL query is a collection of patterns (or more precisely, pattern instances) and a collection of filters that constrain the results of a query. A pattern is a statement where each of the components may be either a constant or a variable. A constant is a resource identifier, string, number or Boolean value. A result of a query is a function from a finite set of variables to a constant. The result set of a query is a multiset of results. We now specify using denotational (compositional) semantics how to determine the result set of a query.

We begin with some mathematical notation in A.1. In A.2, the evaluation of filter expressions is specified recursively, and in A.3, the semantics of a query is defined.

A.1 Mathematical Background

In this section we will use the notation for KGs introduced in Section 5.1. In particular, we use the set *Res* of all possible nodes that can be in a KG.

A *result* (of a query) is a function $f: W \to Res$, where $W \subset V$is a finite set of variables. The domain of a result f is written $dom(f)$. The domain of a result can be empty, and there is a unique result with an empty domain (*i.e.*, the empty function). The *projection* of a result $f: W \to Res$ to a subset $U \subseteq W$ is the restriction of f to U and is written $\pi_U f$. Two results $f: W \to Res$ and $g: Y \to Res$ are said to be *compatible* if they coincide on the intersection of their domains, *i.e.*, $\forall v \in W \cap Y, f(v) = g(v)$. The *join* of compatible results f and g is the unique function $h = f \bowtie g: W \cup Y \to Res$ such that $\pi_W h = f$ and $\pi_Y h = g$.

A *multiset* is a set that can have repeated elements. A *result set S* is a multiset of results all of which share the same domain which is written *dom(S)*. The projection of a result set with domain W to a subset $U \subseteq W$ is the multiset $\pi_U(S) = \{\pi_U f: U \to Res | f \in S\}$ such that if several results become the same after projection to *U*, then their multiplicities are summed. The *join* of two result sets S and T is the multiset $S \bowtie T = \{f \bowtie g | f \in S \text{and} g \in T \text{are compatible}\}$. The join of results sets accounts for results that occur more than once. This means that if $f \in S$ has multiplicity m and if $g \in T$ has multiplicity *n,* then the multiplicity of $f \bowtie g \text{in} S \bowtie T$ is the product *mn*.

A *pattern* (or more precisely, a *pattern* instance) is a quadruple *P* = *(s,p.o,e)* such that the four components are in $V \cup Res$, *i.e*., $s, p, o, e \in Res \cup V$. The set of variables that occur in a pattern *P* will be written *var(P).* We extend a result set $f: W \to Res$ to all of $V \cup Res$ by setting *f(x)* to *x* for every $x \notin dom(f)$. Similarly, we define *f(P)* for a pattern $P = (s, p, o, e)$ to be $(f(s), f(p), f(o), f(e))$. In other words, for each component of a pattern, if the component is in *dom(f)* then apply *f* to the component; otherwise, leave the component alone.

A pattern is the primary KGSQL query mechanism. Given a knowledge graph *G* and pattern *P,* the semantics of *P* with respect to *G* is the result set

$$G(P) = \{f: var(P) \to Res | f(P) \in G\}$$

The syntax for a pattern $P = (s, p, o, e)$ in a KGSQL query is

s [e p] o

We will use the following notation for the semantics of a primary pattern in a *KGSQL* query:

$$[\![\text{s [e p] o}]\!]_G = G(s, p, o, e).$$

A.2 Query Filters

A *filter* is a Boolean expression that limits the result set of a query to contain only the results that satisfy the filter expression. The evaluation of an expression *expr* with respect to a result *f* and knowledge graph *G* is written $eval(expr, f, G)$. Filter expressions in a KGSQL query use a typical syntax, except that the logical operators (*i.e*., AND, OR and NOT) can be specified using either infix operators or functions. The logical infix operators are left-associative short-circuit operators, whereas the logical functions are not short-circuit operators. The evaluation of expressions is defined recursively as follows.

1. $eval(expr_1 || expr_2. f, G) =$
$$\begin{cases} \text{true}, & \text{if} eval(expr_1, f, G) = \text{true}, \\ \text{false}, & \text{if} eval(expr_1, f, G) = \text{false and } eval(expr_2, f, G) = \text{false}, \\ \bot, & \text{otherwise.} \end{cases}$$

2. $eval(expr_1 \&\& expr_2. f, G) =$
$$\begin{cases} \text{false}, & \text{if} eval(expr_1, f, G) = \text{false}, \\ \text{true}, & \text{if} eval(expr_1, f, G) = \text{true and } eval(expr_2, f, G) = \text{true}, \\ \bot, & \text{otherwise.} \end{cases}$$

3. For a binary operator op ∈ {=,!=},
$eval(expr_1 \text{op} expr_2. f, G)$
$$= \begin{cases} eval(expr_1, f, G) \text{op} eval(expr_2, f, G) & \text{if both evaluations are defined,} \\ \bot, & \text{otherwise.} \end{cases}$$

4. For a binary operator op ∈ {<,>,<=,>=}, $eval(expr_1 \text{op} expr_2. f, G) =$
$$\begin{cases} eval(expr_1, f, G) \text{op} eval(expr_2, f, G) & \text{if both evaluations are in} Lit, \\ eval(expr_1, f, G) \text{op} eval(expr_2, f, G) & \text{if both evaluations are in} Num, \\ \bot, & \text{otherwise.} \end{cases}$$

5. For a binary operator op ∈ {+,-,*},
$eval(expr_1 \text{op} expr_2. f, G)$
$$= \begin{cases} eval(expr_1, f, G) \text{op} eval(expr_2, f, G) & \text{if both evaluations are in} Num, \\ \bot, & \text{otherwise.} \end{cases}$$

6. $eval(expr_1 / expr_2 . f, G) =$

7. $eval(+expr. f, G) = eval(expr, f, G)$.

8. $eval(-expr. f, G) = \begin{cases} -eval(expr, f, G) & \text{if the evaluation is in} Num, \\ \bot, & \text{otherwise.} \end{cases}$

9. $eval(!expr.f,G) = \begin{cases} \text{false}, & \text{if} eval(expr,f,G) = \text{true}, \\ \text{true}, & \text{if} eval(expr,f,G) = \text{false}, \\ \bot, & \text{otherwise.} \end{cases}$

10. For a function *g* that can take *n* arguments,
$eval(g(expr_1, expr_2, ..., expr_n), f, G) =$

11. For a variable $v \in V$, $eval(v.f,G) = \begin{cases} f(v) & \text{if } v \in dom(f), \\ \bot, & \text{otherwise.} \end{cases}$

12. For a constant $c \in Res$, $eval(c,f,G) = c$.

13. For a WHERE clause *W,* $eval([\![\text{ask}W]\!]_G, f, G) =$
$\begin{cases} \text{true}, & \text{if } [\![W]\!]_G \neq \emptyset, \\ \text{false}, & \text{if } [\![W]\!]_G = \emptyset, \\ \bot, & \text{otherwise.} \end{cases}$

A.3 Query Semantics

The semantics of a KGSQL query is specified in this section. A query has both query clauses and filter clauses. A filter clause is an expression in the variables occurring in the query clauses. The filter expressions are evaluated for each result of the query clauses, and only the results for which the filter expressions all evaluate to true are returned. During this process, if a filter expression is undefined, then the entire query is undefined.

The denotational semantics of a KGSQL query with respect to a knowledge graph *G* is specified recursively below. The symbols s, p, o, e, r, and c are elements of $Res \cup V$, and the symbols v and w are variables in *V* that do not occur among the variables in the query.

1. $[\![$s [e p] o$]\!]_G = G(s,p,o,e)$

2. $[\![$[r e c]$]\!]_G = [\![$r [e rdf:type] c$]\!]_G$

3. For a finite subset $\{c_1, c_2, ..., c_n\} \subseteq Id$, $[\![$r e $c_1|c_2|...|c_n]]\!]_G = \cup_{i=1}^{n} [\![$r e $c_i]\!]_G$

4. For an integer m, $[\![\text{s [e p] o \{m\}}]\!]_G =$
   a) $\emptyset$, if $m = 0$,
   b) $[\![\text{s [e p] o}]\!]_G$, if $m = 1, -1$,
   c) $[\![\text{s [v}_1\text{ p] w}_1]\!]_G \bowtie_{\bowtie} [\![\text{w}_1\text{ [v}_2\text{ p] w}_2]\!]_G \bowtie \cdots \bowtie [\![\text{w}_{m-1}\text{ [e p] o}]\!]_G$, if $m > 1$,
   d) $[\![\text{w}_1\text{ [v}_1\text{ p] o}]\!]_G \bowtie [\![\text{w}_2\text{ [v}_2\text{ p] w}_1]\!]_G \bowtie \cdots \bowtie [\![\text{s [e p] w}_{-m-1}]\!]_G$, if $m < -1$,

5. For integers $m \leq n$, $[\![\text{s [e p] o \{m..n\}}]\!]_G = \cup_{i=m}{}^{n}[\![\text{s [e p] o \{i\}}]\!]_G$

6. For an integer m, $[\![\text{s [e p] o \{m..*\}}]\!]_G = \cup_{i\geq m}[\![\text{s [e p] o \{i\}}]\!]_G$

7. For an integer n, $[\![\text{s [e p] o \{*..n\}}]\!]_G = \cup_{i\leq m}[\![\text{s [e p] o \{i\}}]\!]_G$

8. $[\![\text{s [e p] o \{*..*\}}]\!]_G = \cup_{i}[\![\text{s [e p] o \{i\}}]\!]_G$

9. For an integer m, $[\![\text{s [e p] \{m\} o}]\!]_G =$
   e) $\emptyset$, if $m = 0$,
   f) $[\![\text{s [e p] o}]\!]_G$, if $m = 1, -1$,
   g) $[\![\text{s [v}_1\text{ p] w}_1]\!]_G \bowtie_{\bowtie} [\![\text{v}_1\text{ [v}_2\text{ p] w}_2]\!]_G \bowtie \cdots \bowtie [\![\text{v}_{m-1}\text{ [e p] o}]\!]_G$, if $m > 1$,
   h) $[\![\text{w}_1\text{ [v}_1\text{ p] o}]\!]_G \bowtie [\![\text{w}_2\text{ [v}_2\text{ p] v}_1]\!]_G \bowtie \cdots \bowtie [\![\text{s [e p] v}_{-m-1}]\!]_G$, if $m < -1$,

10. For integers $m \leq n$, $[\![\text{s [e p] \{m..n\} o}]\!]_G = \cup_{i=m}{}^{n}[\![\text{s [e p] \{i\} o}]\!]_G$

11. For an integer m, $[\![\text{s [e p] \{m..*\} o}]\!]_G = \cup_{i\geq m}[\![\text{s [e p] \{i\} o}]\!]_G$

12. For an integer n, $[\![\text{s [e p] \{*..n\} o}]\!]_G = \cup_{i\leq m}[\![\text{s [e p] \{i\} o}]\!]_G$

13. $[\![\text{s [e p] \{*..*\} o}]\!]_G = \cup_{i}[\![\text{s [e p] \{i\} o}]\!]_G$

14. $[\![\text{s p o}]\!]_G = \pi_{var\{s,p,o\}}[\![\text{s [v p] o}]\!]_G$

15. If the subject or object or both are bracketed expressions, then the result set is obtained by join. For example, $[\![\text{[s e c] p o}]\!]_G = [\![\text{[s e c]}]\!]_G \bowtie [\![\text{s p o}]\!]_G$

16. For a set of clauses $Q = \{Q_1, Q_2, ..., Q_n\}$ as in cases (1) to (15),
    $[\![Q]\!]_G = [\![Q_1]\!]_G \bowtie [\![Q_2]\!]_G \bowtie \cdots \bowtie [\![Q_n]\!]_G$

17. For a set of clauses $Q = \{Q_1, Q_2, ..., Q_n\}$ and an expression *expr* in the variables *var(Q)*,

$[\![Q \,.\, \text{filter}(\textit{expr})]\!]G =$

a) $\{f \in [\![Q]\!]_G \mid \textit{eval}(\textit{expr}, f, G) = \text{true}\}$, if every evaluation is defined, and
b) $\perp$, if $\textit{eval}(\textit{expr}, f, G) = \perp$ for any $f \in [\![Q]\!]_G$

18. For a WHERE clause W consisting of a set of query clauses Q = {Q_1, Q_2, ..., Q_n}
and a sequence of filter expressions $\textit{expr}_1, \textit{expr}_2, \ldots, \textit{expr}_n$, in the variables $\textit{var}(Q)$,
$[\![W]\!]_G = [\![Q \,.\, \text{filter}(\textit{expr}_1 \,\&\&\, \textit{expr}_2 \,\&\&\, \ldots \,\&\&\, \textit{expr}_n)]\!]_G$

9. For a WHERE clause W and a nonempty set of variables $v_1, v_2, \ldots, v_n \in V$,
$[\![\text{select } v_1, v_2, \ldots, v_n \text{ where } W]\!]_G =$

a) $\pi_{\{v1, v2, \ldots, vn\}}[\![W]\!]_G$, if $[\![W]\!]_G$ is defined
b) $\perp$, if $[\![W]\!]_G = \perp$

Appendix B Category Theory and Institutions

A category is a labeled directed graph with an associative composition operation. In this section, the notions of category and functor are defined using computer science notation rather than the traditional notation from mathematics.

Category theory abstracts the more concrete notions of function and function composition, which we now discuss because nearly the same notation is used for category theory. For a set *S*, the identity function on *S* is denoted $\mathbf{1}_S$. For a function $f: S \to T$, the *domain* of f is *S* and the *codomain* of f is *T*. For sets *S, T, U*, and functions $f: S \to T$ and $g: T \to U$, the composition of f and g is a function $h: S \to U$ such that for each element *x* in *S*, $h(x) = g(f(x))$. The composition is written either $g \circ f$ or as *f;g*. The latter is generally preferred by computer scientists because it is analogous to the notation in most programming languages for the composition of successive statements in a program. If *F* and *G* are sets of functions, the set of pairs of functions of *F* and *G* that are composable will be written $F \bowtie G$. In other words, $F \bowtie G = \{(f, g) \mid f \in F, g \in G$, and the codomain of f is the same as the domain of $g\}$.

The notation for category theory differs from the notation for sets and functions in a few ways. Instead of functions, one has morphisms. Instead of the domain and codomain of a function, one speaks of the source and target of a morphism. The notation for the composition of morphisms is the same as the notation for functions, except that the formula for the composition need not hold.

A *category* **C** consists of the following components:

1. A collection **C.ob** of *objects*.
2. A collection **C.hom** of *morphisms*.
3. A function **C.src**: **C.hom** ➞ **C.ob** that specifies the *source* of each morphism.
4. A function **C.tar**: **C.hom** ➞ **C.ob** that specifies the *target* of each morphism.
5. A function **C.id**: **C.ob** ➞ **C.hom** that specifies the *identity morphism* of each object.
6. A function **C.comp**: **C.hom** ⋈ **C.hom** ➞ **C.hom** that specifies morphism composition such that composition is associative, and the composition of an identity morphism with another morphism is equal to the other morphism.

It is a common practice to specify category theory axioms and results using commutative diagrams. A diagram of objects and morphisms is said to be commutative when for every pair of objects in the diagram, the composition of every path of morphisms from the first object to the other yields the same morphism. These diagrams are a way of visualizing axioms and results using graphs. Both the nodes and the edges of a commutative diagram are labeled, so a commutative diagram is a knowledge graph. For example Figure 5 specifies that the source and target of an identity morphism are the object of the identity morphism. In Figure 5 there are three paths from the upper left corner to the lower right corner. The diagram is commutative when all three paths yield the same morphism. In other words, when **C.id**;**C.src** = $\mathbf{1}_{\mathbf{C.ob}}$ = **C.id**;**C.tar**.

$$\begin{tikzcd} C.ob \arrow[r, "C.id"] \arrow[d, "C.id"'] \arrow[dr, "1_{C.ob}"] & C.hom \arrow[d, "C.src"] \\ C.hom \arrow[r, "C.tar"'] & C.ob \end{tikzcd}$$

Figure 5: Example of a commutative diagram

The collection of all sets and functions between them forms a category **Set**. If the source and target of every morphism of a category **C** are interchanged, *i.e.*, the direction of every morphism is reversed, then the result is again a category which is written $\mathbf{C}^{\mathbf{op}}$.

While a category is a labeled directed graph, the converse is not necessarily true, since directed graphs do not generally have a composition operation. However, finite paths in a graph can be composed, and the collection of nodes and paths of a KG is a category called the *path category* of the KG.

The collection of all knowledge graphs forms a category **KG** whose morphisms are functions that preserve edges of the graph. More precisely, a *morphism* $f: G \to H$ of knowledge graphs G and H is a function $f: G.node \to H.node$ such that

1. For every $x \in G.node$ such that $x \notin Id$, we have that $f(x)=x$,
2. For every $x \in G.node$ such that $x \in Id$, we have that $f(x) \in Id$, and
3. For every $(s, p, o, e) \in G$, we have that $\big(f(s), f(p), f(o), f(e)\big) \in H$.

A *functor* **F** from a category **C** to a category **D**, written **F**: **C** ➞ **D**, is a pair of functions

1. **F.ob**: **C.ob ➞ D.ob**
2. **F.hom**: **C.hom ➞ D.hom**

such that

a) **C.src;F.ob = F.hom**;**D.src**,
b) **C.tar;F.ob = F.hom**;**D.tar**,
c) **C.id;F.hom = F.ob**;**D.id**, and
d) **C.comp**;**F.hom** = (**F.hom** ⋈ **F.hom**);**D.comp**.

The axioms for a functor can be diagrammed as in Figure 6.

$$\begin{tikzcd}
\text{C.hom} \arrow[r, "\text{F.hom}"] \arrow[d, "\text{C.src}"'] & \text{D.hom} \arrow[d, "\text{D.src}"] \\
\text{C.ob} \arrow[r, "\text{F.ob}"'] & \text{D.ob}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{C.hom} \arrow[r, "\text{F.hom}"] \arrow[d, "\text{C.tar}"'] & \text{D.hom} \arrow[d, "\text{D.tar}"] \\
\text{C.ob} \arrow[r, "\text{F.ob}"'] & \text{D.ob}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{C.ob} \arrow[r, "\text{C.id}"] \arrow[d, "\text{F.ob}"'] & \text{C.hom} \arrow[d, "\text{F.hom}"] \\
\text{D.ob} \arrow[r, "\text{D.id}"'] & \text{D.hom}
\end{tikzcd}
\qquad
\begin{tikzcd}
\text{C.hom} \bowtie \text{C.hom} \arrow[r, "\text{C.comp}"] \arrow[d, "\text{F.hom} \bowtie \text{F.hom}"] & \text{C.hom} \arrow[d, "\text{F.hom}"] \\
\text{D.hom} \bowtie \text{D.hom} \arrow[r, "\text{D.comp}"'] & \text{D.hom}
\end{tikzcd}$$

Figure 6: The Functor Axioms

The collection of categories and functors forms a category **Cat**. We are now ready to define an institution (Goguen and R. Burstall, 1983).

An institution $\mathcal{I}$ consists of the following:

1. A category **Sign** whose objects are called *signatures.*
2. A functor **Sen**: **Sign** ➞ **Set**. For a signature Σ, the elements of **Sen**(Σ).**ob** are called the Σ-sentences.
3. A functor **Mod**: **Sign** ➞ **Cat**$^{\mathbf{op}}$. For a signature Σ, the elements of **Mod**(Σ).**ob** are called the Σ-*models*, and the elements of **Mod**(Σ).**hom** are called the Σ-*morphisms*.
4. A relation $\vDash_\Sigma$ ⊆ **Mod**(Σ).**ob** × **Sen**(Σ).**ob**, for each Σ ∈ **Sign.ob**, called Σ-*satisfaction*.

The satisfaction relation must satisfy the following axiom:

∀φ:Σ→Ξ ∈ **Sign.hom**, ∀s ∈ **Sen**(Σ).**ob**, ∀ξ ∈ **Mod**(Ξ).**ob**, ξ $\vDash_\Xi$ **Sen**(φ)(s) iff **Mod**(φ)(ξ) $\vDash_\Sigma$ s

The KGSQL institution is written 𝒦𝒢𝒮𝒬□, and has the following components:

1. The category **Sign** of 𝒦𝒢𝒮𝒬□ has a single object and morphism. Because **Sign** is a singleton, we omit reference to the unique signature of 𝒦𝒢𝒮𝒬□.
2. A sentence of 𝒦𝒢𝒮𝒬□ is an ask query, and the functor **Sen**: **Sign** ➞ **Set** maps the unique signature to the set of all ask queries.

3. A model of $\mathcal{KGSQ}$□ is a knowledge graph. The functor **Mod**: **Sign** $\rightarrow$ **Cat**$^{\mathbf{op}}$ maps the unique signature to the category **KG**.
4. The satisfaction relation $\vDash$ is the relation $\{(G,Q) | G$ is a KG, Q is an ask query, and $[\![Q]\!]_G = \text{true}\}$

Now **Sign** in $\mathcal{KGSQ}$□ has only one morphism φ, so it must therefore be an identity morphism. Consequently, in the satisfaction axiom for $\mathcal{KGSQ}$□, **Sen**(φ) is the identity function of the set of all ask queries, and **Mod**(φ) is the identity functor from **KG** to itself. Therefore, the satisfaction axiom for $\mathcal{KGSQ}$□ simplifies to the following:

$$\forall s \in \mathbf{Sen}.\mathbf{ob}, \forall \xi \in \mathbf{Mod}.\mathbf{ob}, \xi \vDash s \text{ iff } \xi \vDash s$$

which is trivially true. So $\mathcal{KGSQ}$□ is an institution.

Appendix C KGSQL Syntax

```
grammar KGSQL;

root : command;

command : prologue ( selectQuery | askQuery
  | constructQuery | insertRequest | deleteRequest );

prologue : prefixDecl*;

prefixDecl : Prefix NamedGraph Identifier;

selectQuery : Select Variable+ whereClause;

askQuery : Ask whereClause;

constructQuery : Construct patternBlock whereClause;

insertRequest : Insert patternBlock whereClause;

deleteRequest : Delete Variable+ whereClause;

whereClause : Where?
```

```
  '{' patternBlock? ( filter '.'? patternBlock? )* '}';

patternBlock : patternsSameSubject ( '.' patternBlock? )?;

filter : Filter constraint;

patternsSameSubject : ( noun | linkedList ) predicateList?;

predicateList : verb objectList ( ';' ( verb objectList )? )*;

objectList : object ( ',' object )*;

object : noun | linkedList;

noun : resourceOrVariable multiplicity?
  | '[' resourceOrVariable resourceOrVariable? typeUnion ']' multiplicity?;

verb : typeUnion multiplicity?
  | '[' resourceOrVariable typeUnion ']' multiplicity?;

typeUnion : Variable | prefixedName ( '|' prefixedName )*;

resourceOrVariable : prefixedName | typedLiteral | numericLiteral
  | True | False | Variable;

linkedList : '(' ( resourceOrVariable | linkedList )* ')'
  | '[' '(' ( resourceOrVariable | linkedList )* ')' prefixedName ']';

prefixedName : NamedGraph LocalName;

constraint : '(' expression ')' | LocalName '(' expressionList? ')';

expressionList : expression ( ',' expression )*;

expression : conditionalAndExpression ( '||' conditionalAndExpression )*;

conditionalAndExpression : relationalExpression ( '&&' relationalExpres-
sion )*;

relationalExpression : additiveExpression ( '=' additiveExpression
  | '!=' additiveExpression | '<' additiveExpression | '>' additiveExpression
  | '<=' additiveExpression | '>=' additiveExpression )?;

additiveExpression : multiplicativeExpression ('+' multiplicativeExpres-
sion
  | '-' multiplicativeExpression | NatPositive | NatNegative | RealPositive
  | RealNegative )*;

multiplicativeExpression : unaryExpression
  ( '*' unaryExpression | '/' unaryExpression )*;

unaryExpression : primaryExpression | '+' primaryExpression
  | '-' primaryExpression | '!' primaryExpression;

primaryExpression : '(' expression ')' | LocalName '(' expressionList? ')'
  | resourceOrVariable | askQuery;

typedLiteral : Literal Lang? | '[' Literal Lang? prefixedName ']'
  | Literal Lang? '^^' prefixedName;

numericLiteral : Nat | NatPositive | NatNegative | UnsignedReal
  | RealPositive | RealNegative;

multiplicity : '{' integer '}' | '{' integer '..' integer '}';

integer : Nat | NatPositive | NatNegative | '*';

// Lexical Scanner Tokens

// In general, KGSQL is case-sensitive,
// but the following reserved words are case-insensitive:

Prefix : [Pp][Rr][Ee][Ff][Ii][Xx] WS;
Select : [Ss][Ee][Ll][Ee][Cc][Tt] WS;
Ask : [Aa][Ss][Kk] WS;
Construct : [Cc][Oo][Nn][Ss][Tt][Rr][Uu][Cc][Tt] WS;
Insert : [Ii][Nn][Ss][Ee][Rr][Tt] WS;
Delete : [Dd][Ee][Ll][Ee][Tt][Ee] WS;
Where : [Ww][Hh][Ee][Rr][Ee];
Filter : [Ff][Ii][Ll][Tt][Ee][Rr];
```

```
True : [Tt][Rr][Uu][Ee];
False : [Ff][Aa][Ll][Ss][Ee];

// The lexical rules were omitted.
// See kgsql.org/KGSQL.g4 for the full Antlr grammar.
```