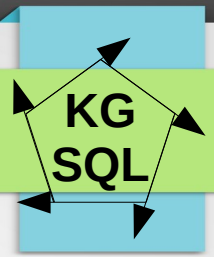
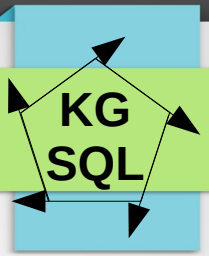


Introducing KGSQL
A Knowledge Graph System Query Language

Ken Baclawski

Outline

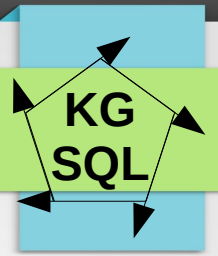


- Motivation for Knowledge Graphs (KGs)
- Definition of Knowledge Graphs
- Issues with the Resource Description Framework (RDF)
- The Knowledge Graph System Query Language (KGSQL)
- Implementation
- Conclusion

About Knowledge Graphs

- Lightweight semantic networks
- Based on the mathematical graph structure
- Scale to massively large data sets
- Critical to intelligent virtual assistants (Siri, Alexa, etc.)
- Popular research area
- Rapidly growing industry with many new applications

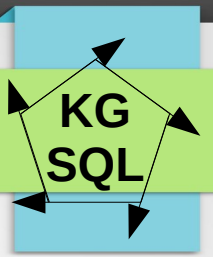
What is a Knowledge Graph?



- Many different definitions of a KG
- Most presume that a KG is implemented using RDF
- The Ontology Summit 2020 examined the problem of a definition and published the community consensus in its Communiqué:

> A KG is a representation of a set of statements in the form of a node- and edge-labeled directed multigraph allowing multiple, heterogeneous edges for the same nodes. A collection of definitional statements specifying the meaning of the knowledge graph's labels is called its schema.

KG Definition



- Definition: A *node- and edge-labeled directed 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.

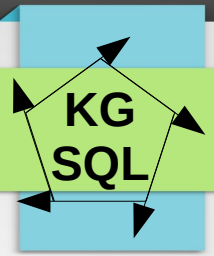
  2. $s: E \longrightarrow V$ and $t: E \longrightarrow V$ are functions that specify the source and target nodes of the edges.

  3. Σ_V is a set of node labels, and Σ_E is a set of edge labels.

  4. $\ell_V: V \longrightarrow \Sigma_V$ and $\ell_E: E \longrightarrow \Sigma_E$ are functions that specify the labels of the nodes and edges.

- Note that V and E need not be disjoint.

What is RDF?



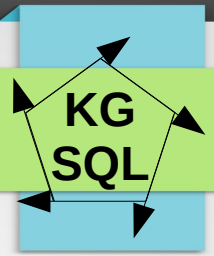
- RDF describes *resources*
  - Properties of resources
  - Relationships between resources
  - Resources are identified by a namespace and a name
- An RDF database consists of *statements*
  - A statement has a subject, verb (predicate), and object
  - Subjects are resources (global or local)
  - Verbs are properties or relationships
  - Objects can be resources or typed literals
  - RDF statements are sometimes called “triples”

rdf:type is the resource named “type” in the rdf namespace

:Fred is the resource named “Fred” in the namespace with an empty name

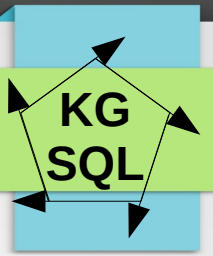
A local resource is also called a “blank node”

RDF Issues



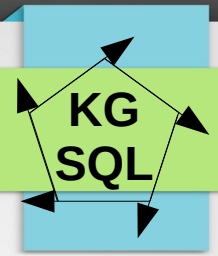
- A KG is most commonly implemented with RDF
- But there are mismatches between the needs of knowledge representation and the facilities of RDF that are now discussed
  - Higher order relations
  - Literals
  - Context and Provenance
  - Other Annotations

Higher Order Relations



- Both RDF and KGs are based on binary relations
- Relations with higher arity (e.g., ternary relations, general records) must be synthesized.
- The classical Suppliers and Parts database illustrates this issue
  - Slide 9 has the relational database schema
  - Representation in RDF is shown on Slide 10
  - Slide 11 discusses the issue

Supplier and Parts Schema and Data

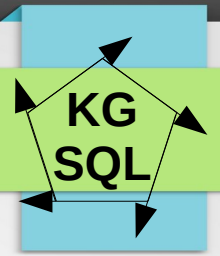



```
CREATE TABLE Supplier (
  id int primary key,
  name varchar(100),
  address varchar(250)
)

(14, "Qrst", "Denver")
```

```
CREATE TABLE Part (
  id int primary key,
  name varchar(250),
  color int,
  weight real
)

(34, "Tube", 5, 1.6)
```

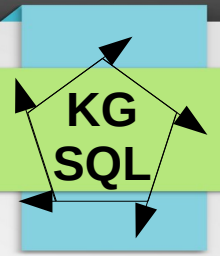
```
CREATE TABLE Shipment (
  supplier int not null
    references Supplier(id),
  part int not null
    references Part(id),
  quantity int,
  primary key(supplier, part)
)

(14, 34, 500)
```

Supplier and Parts in RDF



- Each Supplier record is represented by a resource
  - The primary key is mapped to a unique namespace and name (e.g., `:supplier14`)
  - A statement specifies each non-primary attribute value (e.g., `<:supplier14,:name,"Qrst">`)
- Part records are represented in a similar manner
- The Shipment table is a many-to-many binary relationship between Supplier and Part
  - Each shipment record for a supplier and a part can be represented with an RDF statement (e.g., `<:supplier14,:shipment,:part34>`)
  - **The quantity attribute cannot be added to the RDF statement, so a different strategy is necessary**

Supplier and Parts in RDF



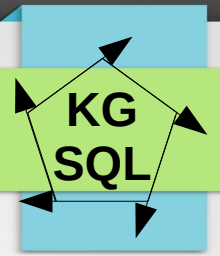
- To represent a shipment with a quantity in RDF there must be a resource to serve as the subject of the RDF statement that specifies the quantity.
- This is done by ***reifying*** the statement `<:supplier14,:shipment,:part34>` which can be done like this

```
<:shipment8581,rdf:type,rdf:Statement>
<:shipment8581,rdf:subject,:supplier14>
<:shipment8581,rdf:predicate,:shipment>
<:shipment8581,rdf:object,:part34>
```

- One can then specify the quantity like this

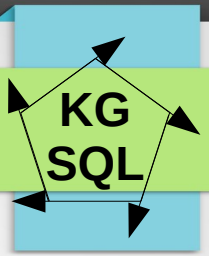
```
<:shipment8581,:quantity,500>
```

Problems with Reification



- Inefficient: One statement becomes four
- Awkward: Queries become much more complicated
- Inconsistent: No connection between the unreified statement and the reified statement
  - If only some shipment statements are reified, queries must take this into account, which increases complexity even more
  - Constraints on shipment statements are not automatically imposed on the reifications
- Inflexible: Adding an attribute can be very complicated and error-prone
  - Compare with adding a new column to a relational database table

RDF Literals

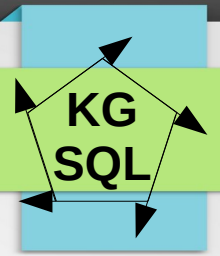


- Because RDF literals are not resources, one cannot specify properties, such as measurement units
- To give properties to a literal, it could be ***reified*** like this

```
<:eg887,:type,rdfs:CompoundLiteral>
<:eg887,rdf:value,”887”>
<:eg887,:unit,uom:gram>
```

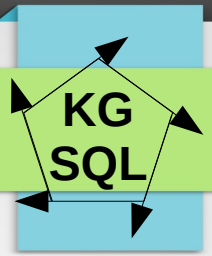
- While this solves the problem, it has many of the same problems as with higher order relations

Context and Provenance



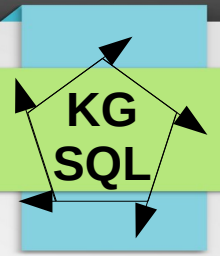
- Generally, individual RDF statements are not meaningful in themselves
  - Semantics and pragmatics are associated with collections of statements
  - The provenance of the the whole collection is important
  - The context within which the collection occurs is also important
- In order to represent this in RDF, it is necessary to be able to specify that an RDF statement belongs to a collection
  - Such a collection is called a *named graph*
  - The name of a graph is a resource that can be used like any other resource
- Because of this, in practice, RDF statements in an RDF data store will have four parts: subject, verb (predicate), object, and name of the graph
- The name of the graph is the ***reification*** of the named graph

Uncertainty, Risk, Trust, ...



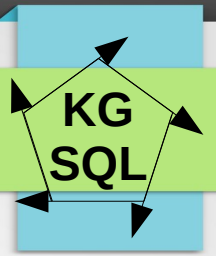
- There are many more annotations that are necessary for application domains
- In general, these are represented using reifications.
- Reifications are used in RDF and OWL for still other constructs, such as linked lists
- Given how often one needs reifications, it would help to be more systematic.

Knowledge Graph System Query Language



- Not just another query language for RDF
- Designed for the needs of general knowledge representation
- Supports general knowledge graphs in which V and E can intersect or even in which E is a subset of V, i.e., *every* edge has been reified
  - Actually, there are some exceptions that will be explained later
- Nearly all KGSQL statements are resources and so have identifiers which can be global or local
- Unlike RDF, a statement and its reification are the same

KGSQL Bracket Pairs



- The notation `[B C]` in KGSQL means that B has type C.
- This is more general in KGSQL than in RDF
  - For a resource B and class C, it is the same as the RDF statement `<B rdf:type C>`
  - A statement is regarded as an instance of the property, so if S is a statement with property Prop and identifier ID, then `[ID Prop]`
  - If L is a literal that has datatype D, then `[L D]`. This is the equivalent to the RDF syntax `L^^D`.

KGSQL Commands

- KGSQL supports SELECT, CONSTRUCT, INSERT and DELETE
- All of the commands use the same pattern syntax
- Here is an example:

```
DELETE ?e WHERE {
  [?a :Person] [?e :name] "Fred" .
  ?e :probability ?p . filter(?p <= 0.9)
}
```

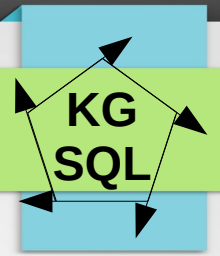
- This deletes all statements that specify that a person is named Fred with probability at most 90%.

KGSQL Syntax

SELECT list of variables WHERE { pattern }	Retrieve the resource identifiers or literals specified by some of the variables in the pattern
CONSTRUCT pattern WHERE { pattern }	Same as SELECT but use variables to construct statements that are to be returned
INSERT pattern WHERE { pattern }	Same as CONSTRUCT but add the constructed statements to the KG database
DELETE list of variables WHERE { pattern }	Delete the statements in the KG database that are identified by the variables in the list

All of the above are preceded by definitions of the prefixes to be used in the patterns.

Resolving RDF Issues



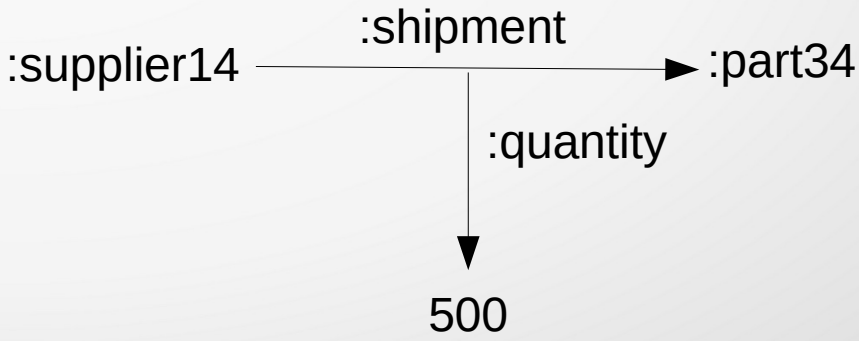
- The various issues discussed above are resolved on the following slides
  - Higher order relations
  - Literals
  - Context and Provenance
  - Other Annotations
- In addition, some other features are discussed
  - Linked Lists
  - Multiplicities

Higher Order Relations

- Although KGSQL is based on binary relations, KGSQL statements can have attributes and relations
- For the Suppliers and Parts Database, the shipment example that required 5 statements in RDF now requires only 2:

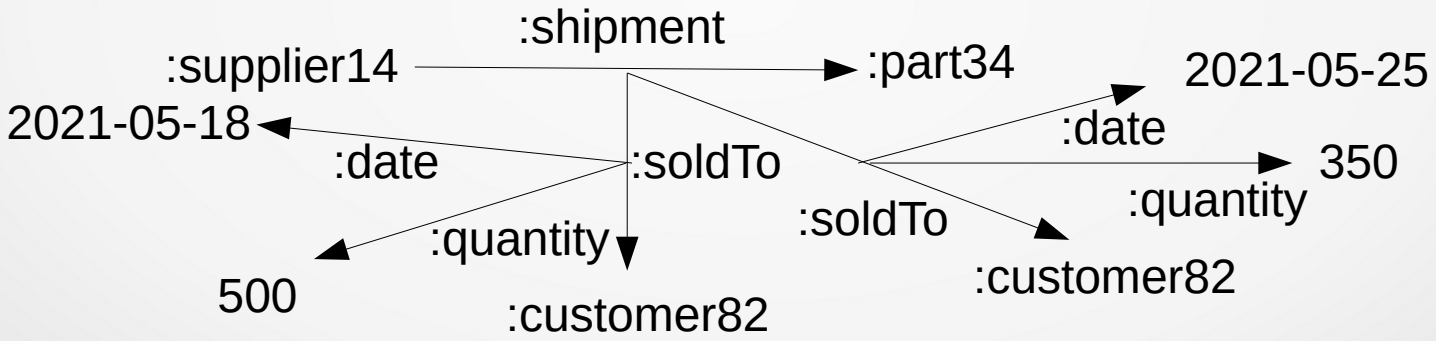
```
:supplier14 [?e :shipment] :part34 .
?e :quantity [“500” xsd:decimal] .
```

- More importantly, queries involving shipments are much simpler.
- One can visualize the statements above like this



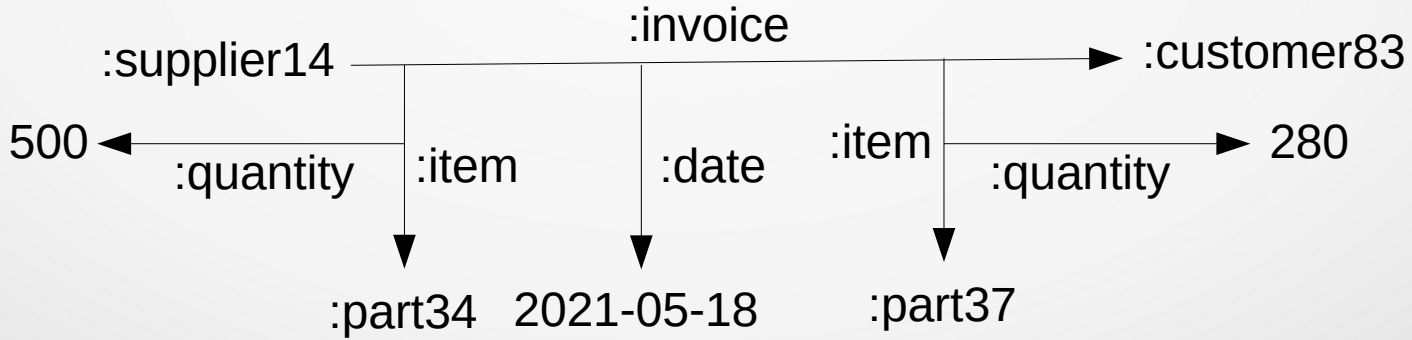
Higher Order Relations

- The Suppliers and Parts Database is unrealistic since there is only one quantity for each supplier-part pair, and the customer is not specified.
- These can easily be added, for example, like this

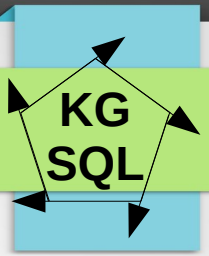


Higher Order Relations

- The representation on the previous slide is also not very realistic
  - 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 like this:



Advantages

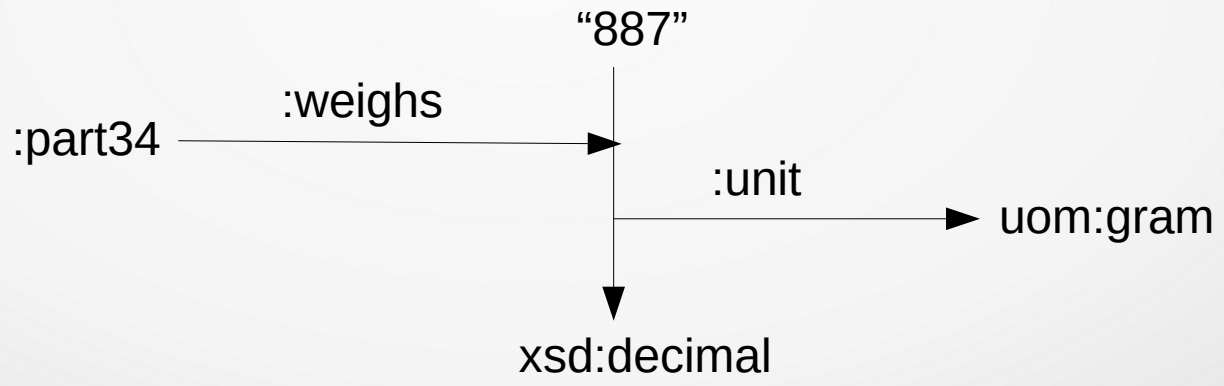


- Efficient: No extra statements
- Convenient: Queries are much simpler
- Consistent: The unreified statement and the reified statement are the same
  - There is never partial reification so there is no added complexity
  - Constraints on shipment statements are imposed on the reifications
- Flexible: Adding more attributes is very easy
- Relational and hierarchical models can be specified in a graph framework

Literals

- Literals can be given properties by using the datatype statement for the literal like this

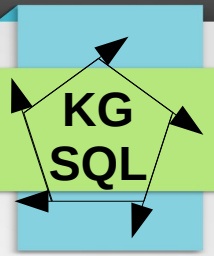
```
:part34 :weighs [“887” ?d xsd:decimal] .
?d :unit uom:gram .
```



Context, Provenance and other Annotations

- There are two techniques for KGSQL named graphs
  1. All the objects of a fixed subject and verb
     - The objects are the statements of the named graph
     - The fixed subject is the name of the named graph
     - This technique was rejected for RDF because it requires that the statements in the named graph be reified
  2. All the statements whose identifiers have the same namespace
     - The namespace is the name of the named graph
- Each technique has advantages and disadvantages

Linked Lists



- An RDF linked list is specified using the built-in properties rdf:first and rdf:rest, and the built-in resource rdf:nil.
- For example, the linked list (:Ken :Greer :Qing) is specified like this

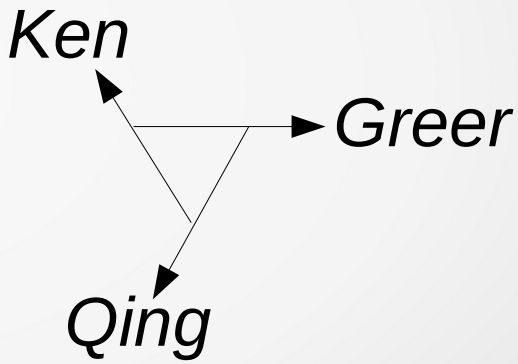
```
<a rdf:first :Ken>
<a rdf:rest b>
<b rdf:first :Greer>
<b rdf:rest c>
<c rdf:first :Qing>
<c rdf:rest rdf:nil>
```

where a, b and c are local resources.

KGSQL Linked Lists

- The KGSQL linked list for (:Ken :Greer :Qing) is specified like this

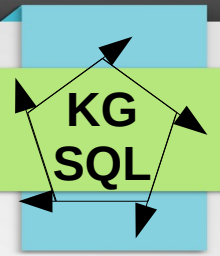
```
c [a prop] :Ken .
a [b prop] :Greer .
b [c prop] :Qing .
```



- Where a, b and c are local resources as in the RDF linked list, and prop is a property
- The KGSQL notation for this linked list is

```
[(:Ken :Greer :Qing) prop]
```

KGSQL Linked Lists



- A KGSQL linked list has some advantages over RDF
  - Requires only one builtin resource
    - kgq:nil is necessary for empty lists
  - Requires half as many statements
  - Can use any property for linking, provided the property has compatible domain and range.
- Retrieval of an element of a linked list can be done as shown on the next slide
  - Note that KGSQL linked lists are circular, so if one requests an element beyond the “end” of the list, the list repeats.

Multiplicities

- Multiplicities can be specified on a pattern
  - The general form is {min..max}
  - {m} is the same as {m..m}
  - {min..*} means that there is no max
- There are two ways to iterate
  - From object to object

    ```
    :a :prop ?x {6}
    ```

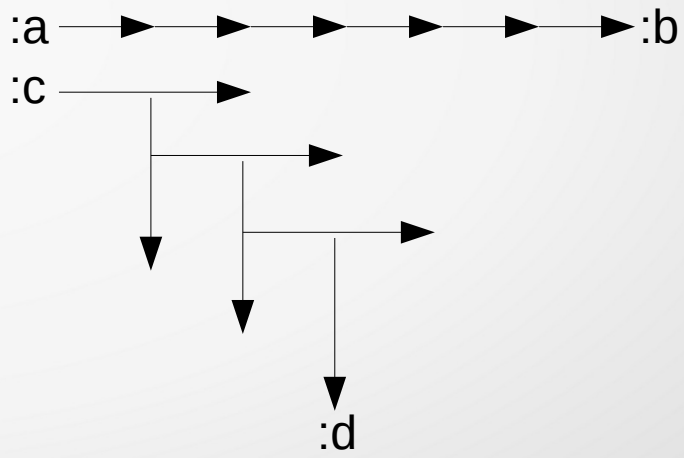
    The result is :b
  - From statement to statement

    ```
    :c :prop {6} ?x
    ```

    The result is :d

Statement to statement iteration is almost the same as array selection in programming languages:

x = c.prop[6]



Negative Multiplicities

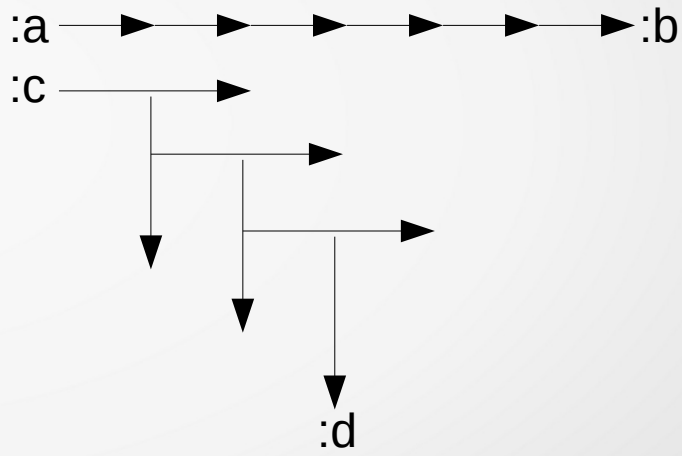
- Multiplicities can be negative
  - {*..max} means that there is no min
- There are two ways to iterate
  - From object to object

    ```
    :x :prop ?b {-6}
    ```

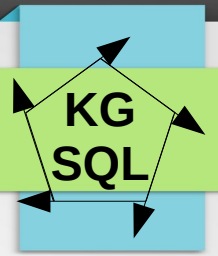
    The result is :a
  - From statement to statement

    ```
    :x :prop {-6} ?d
    ```

    The result is :c



KGSQL Patterns

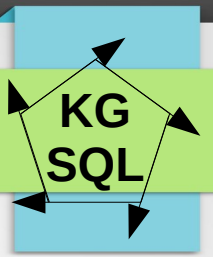


- The WHERE clause of a query is specified using patterns.
  - KGSQL patterns are similar to SPARQL, but slots can have bracket pairs and multiplicities.
- Here is an example:

```
[?a :Person] [?e :name] "Fred" .
?e :probability ?p . filter(?p > 0.9)
```

- This pattern matches persons named Fred with probability greater than 90%.
  - In the bracket pair `[?e :name]` the variable ?e is the identifier of a statement with verb :name
  - The bracket pair `[?a :Person]` restricts the variable ?a to be the identifier of a resource whose type is :Person

Type Unions

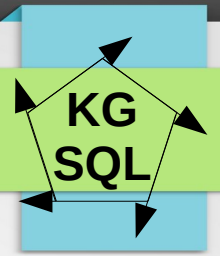


- One can specify more than one class or property in a KGSQL bracket pair
- Here is an example:

```
[?a :Person | :Company] [?e :name] “Fred” .
?e :probability ?p . filter(?p > 0.9)
```

- This pattern matches persons or companies named Fred with probability greater than 90%.
  - The bracket pair `[?a :Person | :Company]` restricts the variable ?a to be the identifier of a resource whose type is :Person or :Company

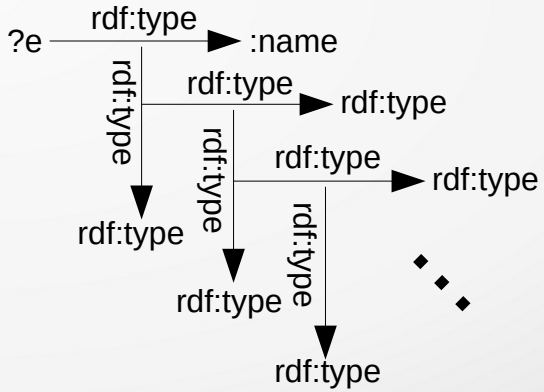
Implementation



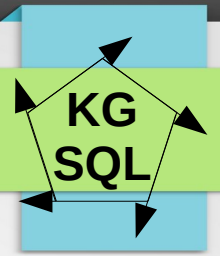
- Obviously implementing KGSQL with RDF statement reification would be very inefficient
- However, there is an implementation that is both simple and efficient:
  - Store each statement as a quad: subject, verb (predicate), object, statement ID
  - RDF “triple stores” are, in fact, based on quads
    - The fourth component is the name of the named graph
    - The fourth component could be used for the statement identifier
    - The namespace of the statement identifier could be used as the name of the named graph
  - The statement ID is the primary key
    - A tree index for the primary key would group statements by the namespace of the primary key and therefore group statements by namespace

Implementation Technicalities

- A statement in KGSQL is an instance of the property of the statement.
- For example if `:Fred [?e :name] "Fred"` then ?e has type :name.
- However, one cannot assert `?e rdf:type :name` as doing so would result in an infinite series of statements:
- So in KGSQL reification is not quite universal

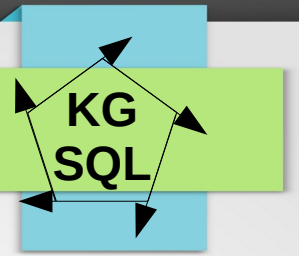


Implementation Technicalities



- The following RDF builtin resources are not needed by KGSQL:
  - rdf:Statement, rdf:subject, rdf:predicate, rdf:object
  - rdf:CompoundLiteral, rdf:value
  - rdf:List, rdf:first, rdf:rest
- In addition,
  - Multiplicities allow one to retrieve elements of linked lists almost as succinctly as the array notation in programming languages
  - KGSQL lists might be implemented as arrays to improve performance

KG Definition Revisited



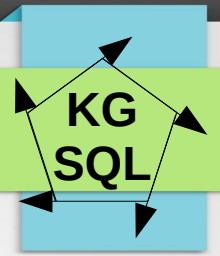
How is KGQ an implementation of the definition of a knowledge graph?

V is the set of resources
E is the set of statements.
E is almost a subset of V.
The functions s and t are the subject and object of a sentence.
Node and edge labels (Σ_V and Σ_E) are bracket pairs.

- Definition: A *node- and edge-labeled directed 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. $s: E \longrightarrow V$ and $t: E \longrightarrow V$ are functions that 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. $\ell_V: V \longrightarrow \Sigma_V$ and $\ell_E: E \longrightarrow \Sigma_E$ are functions that specify the labels of the nodes and edges.
- Note that V and E need not be disjoint.

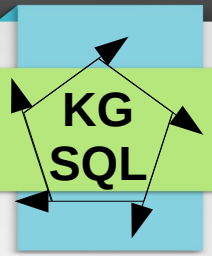
$\ell_V: V \longrightarrow \Sigma_V$ maps a resource to the bracket pair specifying the resource identifier and its type(s).
$\ell_E: E \longrightarrow \Sigma_E$ maps a statement to the bracket pair specifying the statement identifier and its property or properties.

Other Graph Query Languages



- There are many other graph query languages
- None presumes that every property, relationship and type specification is reified
- None takes full advantage of the near universal reification of KGSQL
- The closest graph notions and query languages are the following:
  - RDF* is an improvement over RDF but is still based on RDF reification
  - A property graph is a special kind of knowledge graph
    - Relationships (edges) are reified
    - Edges can have properties but do not participate in relationships
    - Properties are not reified

Conclusion



- KGSQL has some useful features as a language for knowledge representation using graphs
  - Supports non-graph structures such as arrays, trees, and JSON
- KGSQL resolves some of the issues with using RDF for representing KGs
- There is an efficient implementation using existing RDF stores
- There are opportunities for other implementation improvements