

Property Graph Schema

Title	Property Graph Schema	
Authors	Individual Experts Contribution Neo4j Query Languages Standards and Research Team ¹	
Status	SQL/PGQ WD draft change proposal	
Date	Original	6 December 2018
	Revision r1	16 January 2018

Copyright © 2018, Neo4j Inc. Please see the last page of this document for Apache 2.0 licence grant.

Contents

[1 References](#)

[2 Introduction](#)

[2.1 Purpose and status](#)

[2.2 Structure of this document](#)

[2.3 Relationship to other proposals](#)

[3 Rationale \(Informative\)](#)

[3.1 Property graph](#)

[3.2 Property graph type](#)

[3.2.1 Element types](#)

[3.2.2 Vertex types](#)

[3.2.3 Edge types](#)

[3.2.4 Sharing graph types between graphs](#)

[3.3 Property graph views over tables](#)

[3.4 Vertex type and edge type views](#)

[4 Rules](#)

[4.1 Property graph type](#)

[4.1.1 Property graph type](#)

[4.1.2 Element type](#)

[4.1.3 Super element type](#)

[4.1.4 Vertex type](#)

[4.1.5 Edge type](#)

[4.1.6 Property type](#)

[4.1.7 Element key](#)

[4.2 Schema statements](#)

[4.3 Property graph definition and manipulation](#)

[4.3.1 <property graph definition>](#)

¹ Hannes Voigt, Petra Selmer, Stefan Plantikow, Tobias Lindaaker, Alastair Green, Peter Furniss.

[Function](#)

[Format](#)

[Syntax rules](#)

[General rules](#)

[4.3.2 <element type declaration>](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.3 <vertex type declaration>](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.4 <edge type declaration>](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.5 <property type declaration>](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.6 <element key declaration>](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.7 Names and identifiers](#)

[Function](#)

[Format](#)

[Syntax Rules](#)

[Access Rules](#)

[General Rules](#)

[Conformance Rules](#)

[4.3.8 Changes to 5.1, additional SQL terminal symbol](#)

[Add to Format:](#)

[5 Glossary of symbols \(Informative\)](#)

[6 Possible Problems](#)

[7 Language Opportunities](#)

[8 Examples \(Informative\)](#)

[8.1 Element type](#)

[8.1.1 Examples](#)

[8.2 Vertex type](#)

[8.2.1 Examples](#)

[8.2.2 Negative examples:](#)

[8.3 Edge type](#)

[8.3.1 Examples](#)

[8.4 Complete example showing element, vertex and edge type declarations](#)

[8.5 Mapping examples](#)

[8.5.1 Relational schema](#)

[8.5.2 Two vertex types, one edge type: using a link table](#)

[8.5.3 Two vertex types, one edge type: edge from one of the vertex tables](#)

[8.5.4 Vertex loaded from more than one input table](#)

[8.5.5 Vertex types with composition](#)

[8.5.6 Edges and vertices using a single table \(self-referencing\)](#)

[8.5.7 Edges from multiple input tables](#)

[8.5.8 Example showing how one graph can be created reusing types from another graph](#)

[8.5.9 Using keys](#)

[8.5.9.1 Element type with a key](#)

[8.5.9.2 Element type using inheritance with a key](#)

[8.5.9.3 Vertex type with a key](#)

[8.5.9.4 Edge type with a key](#)

[9 Checklist](#)

[10 Revision details](#)

[10.1 Changes in sql-pg-2018-0056r1 \(WG3:BNE-022\)](#)

[11 An ITI, ISO and openCypher/GQL contribution from Neo4j Inc.](#)

1 References

[Framework:2020]	ISO/IEC JTC1/SC32 WG3:ERF-002 Jim Melton (ed), “ISO International Standard (IS) Database Language SQL- Part 1: SQL/Framework ”, ISO/IEC IWD 9075-2:2020(E)
[Foundation:2020]	ISO/IEC JTC1/SC32 WG3:ERF-003 Jim Melton (ed), “ISO International Standard (IS) Database Language SQL- Part 2: SQL/Foundation ”, ISO/IEC IWD 9075-2:2020(E)
[Schemata:2020]	ISO/IEC JTC1/SC32 WG3:ERF-008 Jörn Bartels, Jim Melton, (eds), “ISO International Standard (IS) Database Language SQL- Part 11: SQL/Schemata ”, ISO/IEC IWD 9075-11:2016:2020(E)
[SQL/PGQ IWD]	ISO/IEC JTC1/SC32 WG3:BNE-012 ANSI INCITS DM32.2-2019-00xxx Jim Melton (ed), “ Working Draft (WD) Database Language SQL - Part 16: SQL/PGQ ”
[PPG data model]	ISO/IEC JTC1/SC32 WG3:YTZ-033 ANSI INCITS DM32.2-2018-00091 sql-pg-2018-0002 Jan Michels, “ The Pure Property Graph Data Model ”
[PG DM for SQL]	ISO/IEC JTC1/SC32 WG3:YTZ-034 ANSI INCITS DM32.2-2018-00092 ANSI INCITS sql-pg-2018-0003r2 Neo4j SQL working group, “ Property Graph Data Model for SQL ”, April 2018
[Graph patterns]	ISO/IEC SC32/WG3:ERF-035 ANSI INCITS DM32.2-2018-00153r1 ANSI INCITS sql-pg-2018-0029r1 Fred Zemke, “ Fixed graph patterns ”, September 2018
[PPG data model 2]	ISO/IEC JTC1/SC32 WG3:ERF-042 ANSI INCITS DM32.2-2018-001nn ANSI INCITS sql-pg-2018-0035 Jan Michels,

	“ The pure property graph data model ”, September 2018
[SQL PG Schema]	ISO/IEC JTC1/SC32 WG3:ERF-043 <i>ANSI INCITS sql-pg-2018-0036r2</i> Peter Furniss, “ SQL/PG graph schema and join syntax mapping examples ”, September 2018
[PG DM Concepts]	ANSI INCITS sql-pg-2018-0037r1 <i>ISO/IEC JTC1/SC32 WG3:ERF-044</i> Alastair Green, “ Property Graph Data Model Concepts and Terms ”, October 2018
[PG Types DDL]	ANSI INCITS sql-pg-2018-39r1 Mingxi Wu, “ Property Graph Type System and Data Definition Language ”, November 2018
[Cardelli1984]	Luca Cardelli, “ A semantics of multiple inheritance ”, Proceedings of Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984; pages 51–67, Springer.
[Conceptual Graphs]	J. F. Sowa, “ Conceptual Structures ”, Chapter 3, “Conceptual Graphs”, 1984, Addison-Wesley.

2 Introduction

2.1 Purpose and status

This document is the first draft of a Change Proposal against [\[SQL/PGQ IWD\]](#), which also implies changes to [\[Framework:2020\]](#), [\[Foundation:2020\]](#) and [\[Schemata:2020\]](#).

To allow consideration by the SQL PG Ad Hoc, and by DM32.2 before the end of 2018, and to facilitate presentation to WG3 at its Brisbane meeting in January 2019, we have presented this document in a partial state, and expect to issue further revisions.

It presents a normative definition of the descriptor of a new kind of complex schema object: a property graph (and its included objects, including those that define a property graph type), and the format and rules that establish and govern the DDL used to create these objects.

Please note that this revision does not yet describe an `ALTER GRAPH` statement that could be used to progressively construct or evolve a property graph definition. This will be addressed in future revisions.

This draft does not yet make precise the relationship of the proposed specifications of features to existing parts of the standard or to the Working Draft of Part 16.

This draft presents an informative Rationale, Examples and Glossary of Symbols used in rules.

2.2 Structure of this document

Section 3 is informative. It motivates, describes and exemplifies the ability to define and manipulate a new kind of complex schema object: a *property graph*, using one or more <SQL schema statement> (DDL).

The change proposal in section 4 of this paper is normative.

Section 5 is an informative glossary of symbols used in the normative rules.

Sections 6 and 7 list possible problems and language opportunities.

Section 8 contains example DDL with commentary.

Section 9 is a checklist of specification tasks to be accomplished.

Section 10 contains revision control notes.

Section 11 is a copyright notice, including Apache Software License 2.0 grant.

2.3 Relationship to other proposals

This proposal allows schema objects to be created which result in the establishment of pure property graphs (PPGs) in an SQL-implementation.

Pure property graphs are described in [\[PPG data model\]](#). There are differences in terminology between this proposal and PPG, but the model presented here is a superset of the PPG model².

The definition of a graph as the expression of a graph type, whose contained vertex and edge types are the target of mappings from tables, is in line with [\[SQL PG Schema\]](#) and [\[PG DM Concepts\]](#).

The approach to graph typing in this proposal reflects these goals and experiences:

- a) ability to express the whole information content of an Entity-Relationship model in a concise, pattern-based piece of “ASCII art”
- b) the evolution of internal thinking in Neo4j, based in part on implementation experience in the Cypher for Apache Spark project³
- c) combining the use of labels in Cypher and PGQL with a type inheritance scheme, which was reinforced by the desire to create convergence with some of the goals of the design presented in [\[PG Types DDL\]](#)
- d) elision of edge types and edge triplets, an idea crystallized by reading [\[PG Types DDL\]](#)
- e) leaving unaltered the use of label expressions described in [\[Graph patterns\]](#)
- f) support for vertex type and edge type keys (as proposed in [\[PG DM for SQL\]](#) and [\[PG DM Concepts\]](#), as well as cardinalities for vertexes, edges, and the heads and tails of relationships
- g) graph typing in SQL (where graphs can only exist through views over tables, and types are a facet of graphs) to be a subset of graph typing in GQL (where we anticipate being able to define graph types independently of graphs; the ability to create graphs that are not populated by mappings from tables; and where we wish to support partial schemas).

² In a future revision we will propose the additive changes necessary to bring the PPG into line with this proposal.

³ Spark Cypher engineers Mats Rydberg, Martin Junghanns and Tobias Johansson have been particularly active in contributing to Neo4j’s design thinking in this area.

A future paper for the SQL Ad-Hoc GQL stream will discuss our early thoughts on the last point (graph schema in SQL as a subset of graph schema in GQL), in more detail.

3 Rationale (Informative)

In the following sections a *non-transient table* is a base table or a derived table, as defined in [\[Foundation:2020\]](#). The unqualified term *graph* should be read to mean property graph.

3.1 Property graph

SQL/PGQ describes how to define and read property graphs. An SQL property graph is a directed multi-graph. An edge has a tail vertex and a head vertex⁴; there may be multiple edges between any two vertices. In addition data properties may be associated with each element (vertex or edge). A *pure property graph* as described in [\[PPG data model\]](#) is such a graph.

The change proposal defines a new kind and name class of SQL-schema object, a *property graph*.

A property graph may include a *property graph type*, containing *vertex types* and *edge types*.

Vertex types and edge type definitions refer to *element types*, each of which defines a set of *labels* and a set of *property types*. A property type defines the name, datatype and optionality of a *property* that can be part of an *element* (*vertex* or *edge*) occurring in a property graph's data.

A property graph also may also include *vertex type views*, and *edge type views*, each made up of *mappings*.

A mapping defines the source of data for its view, and how that data populates vertices or edges of the type targeted by that view. The sources of mappings are non-transient tables, which either are, or map in turn to, base tables.

A property graph may be *empty*, in which case it has no user-defined components. An empty property graph contains only a *base element type* (which has no labels or properties), and therefore contains no usable metadata nor any data.

A property graph may contain only element types (including the base element type). Such a graph is like an abstract type, that requires further specification to describe a concrete type that can be instantiated. It contains no data.

A property graph may contain only a property graph type, made up of at least one vertex type and zero to many edge types, each defined with reference to element types (including

⁴ We use the terms “head” and “tail” to avoid using “source” or “end” (which is an SQL keyword), and to mirror the highly mathematical terms “vertex” and “edge”. “Source” is very useful as a term to describe the tables from which data is drawn into the graph.

the base element type). Such a graph contains concrete metadata, but no data. It is (roughly) analogous to a view over empty base tables.

Such a graph may be used as a template for creating graphs of a given graph type, by use of CREATE GRAPH ... LIKE ... In that sense, such a viewless (dataless) graph can be used purely to define a property graph type.

A property graph may include a view for any vertex type or edge type that it includes. A property graph that includes at least one vertex type view contains metadata and data. In effect it contains a *graph view over tables*. (It also contains data relating to the identity and relationships of graph elements which is not derived from SQL-data.)

Of course, it is always possible that the base tables underlying the vertex type and edge type views of such a graph will be devoid of data, but that is an accident of their state at a point in time, and not an inherent characteristic of the property graph schema object. A property graph which contains views is a graph variable in the same sense that a SQL table is a table variable.

3.2 Property graph type

A property graph includes a set of schema objects, which (taken together) describe such a graph: this collection, defines a *property graph type*. A property graph type constrains the content of its containing graph: no data can exist in a graph that does not conform to the property graph type of that graph.

A *property graph type* is a specification concept: there is no single schema object contained by the graph schema object which contains the hierarchy of components which define a graph type.

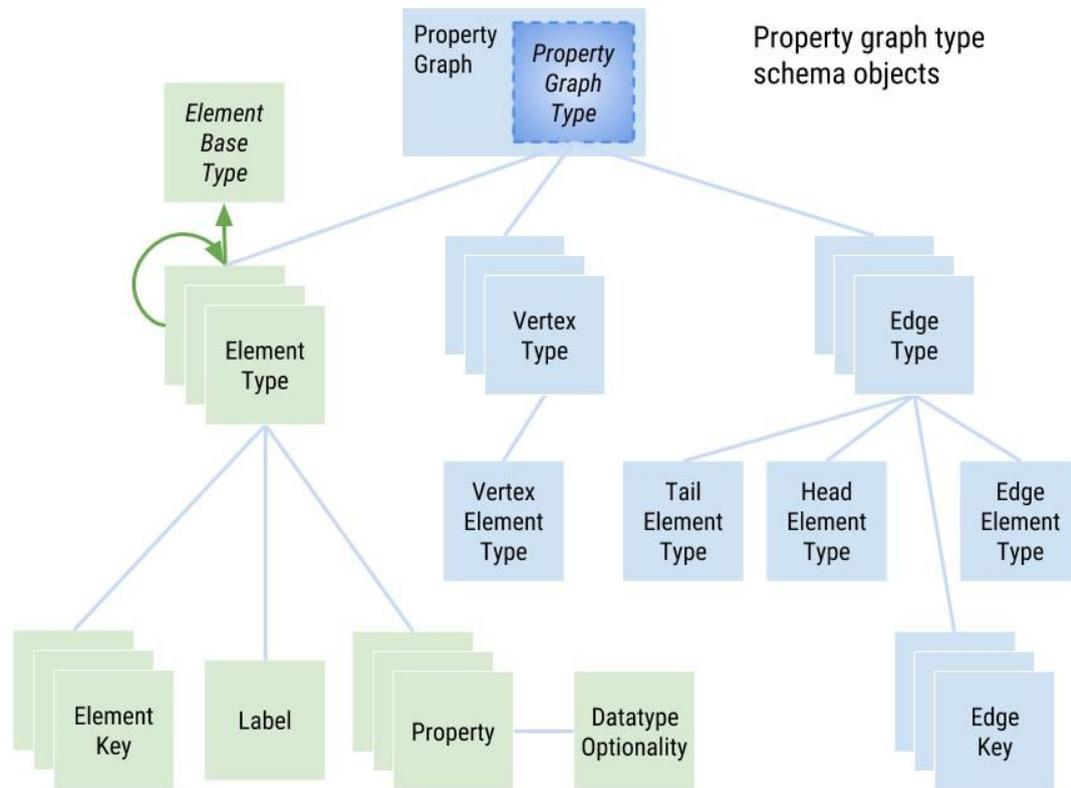
To be used in the instantiation of a property graph containing data, a property graph type must include at least one *vertex type* and zero to many *edge types*. These types have names. They may have zero to many *keys* (sets of mandatory property types that define uniqueness for all instances of such a type), and can include a *cardinality*.

A vertex type includes an *element type*.

An edge type includes three element types. Edge types can also include a *tail cardinality* and a *head cardinality*⁵.

An element type may include other element types (and in so doing *extends* those element types, which become its supertypes). An element type includes a set of *property types* (which may be empty), and have a name which is also called a *label*. Property types have a name, and include a data type, and an *optionality* flag. Element types may also include keys.

⁵ Cardinalities are not defined in the normative part of this document. This will be addressed in a future revision.



An element type generally includes a set of labels, comprising its label and the labels of all its supertypes. It generally includes a set of property types that is the union of its own property type set, and the set of property types of each of its supertypes.

The kind of data that can be associated with a vertex is determined by its vertex type; the kind that can be applied to an edge is determined by its edge type. These types therefore also govern the set of labels and the set of properties that can be evaluated in graph pattern predicates (label expressions), as defined in [\[Graph patterns\]](#).

The following schematic DDL example illustrates the structure and semantics of a property graph type:

```
CREATE GRAPH g (
  -- the vertex types and edge types, together with the element types
  -- they refer to, constitute a property graph type

  -- element types
  NodeA (a1 char(10), a2 integer), -- label NodeA, 2 properties
  NodeB <: NodeA (b1 integer), -- labels NodeB and NodeA, 3 properties
  NodeC EXTENDS NodeA (c1 varchar(255)), -- labels NodeC and NodeA,
  -- 3 properties
```

```

NodeD (d1 char(10)),
NodeE <: (NodeC, NodeD), -- NodeE multiply inherits, creates a label
LINKS_TO, -- label LINKS_TO, no properties

-- vertex types
(NodeA), -- NodeA vertexes can exist, matching label NodeA.
      -- Parentheses are a graph-pattern marker for VERTEX
(NodeB), -- NodeB vertexes can exist, matching labels NodeA and NodeB
(NodeC), -- NodeC vertexes can exist, matching labels NodeA and NodeC
(NodeE), -- NodeE vertexes can exist, matching labels
      --
      -- NodeA, Node C, NodeD and NodeE
-- vertexes with only the label NodeD cannot exist

-- edge types
(NodeA)-[LINKS_TO]->(NodeB), -- Brackets are a marker for EDGE
(NodeA)-[LINKS_TO]->(NodeC) -- not the same edge type as the
      -- trio of element types differs.

-- for both edge types instance tail nodes can be of type
-- NodeA, NodeB, NodeC or NodeE, as NodeA is a subtype of all four
-- for the first edge type head nodes can be of type NodeB
-- for the second edge type head nodes can be of type NodeC or NodeE
)

```

3.2.1 Element types

Element types associate labels and properties in a way that allows label-based pattern matching to be combined with a structured system of complex data types. These data types allow subtype and intersection type relationships to be expressed.

This facility is a subset of the interface inheritance provided by existing programming languages, as it only allows “mixing in” of pure data structures.

We start with property types. Apache Spark has a class called *StructField* that captures/exposes a name, a datatype and optionality: a property type describes these three attributes. A list or sequence of property types describes a data tuple. SQL calls this kind of structure a row type; Spark has a specialized sequence type called a *StructType* that plays the same role.

An element type has a name, it includes a list of property types, and it also has a set of element types that it extends (a set of supertypes). That set may be empty.

If Java had data structures (objects with no behaviour, only state), and you could apply the interface inheritance model provided by the keyword “extends” to such data structures, then you would have a similar capacity to that provided by element types.

```

CREATE GRAPH ConfigurationServiceTemplate
(
  -- element types
  Proxy (address VARCHAR(255) NOT NULL,
         port VARCHAR(10)),
  Configuration extends Proxy (val VARCHAR(255) NOT NULL))
)

```

In the Java fragment that follows, imagine that an interface method returning a type was akin to a C# struct field (C# structs can't be extended, of course). Java's approach to naming and type extension, with multiple interface inheritance, is analogous to the way in which element types are defined.

```

interface Proxy
{
  String address();
  Option<String> port();
}

interface Configuration extends Proxy /* could be a commalist */
{
  string val():
}

```

The Go language has structs which can be extended.

```

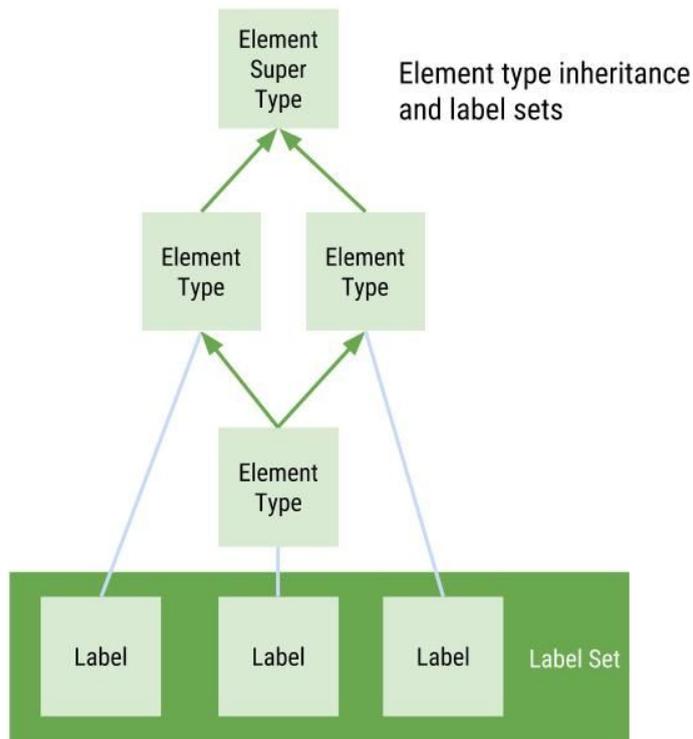
type Configuration struct {
  Val string
  Proxy // this struct extends Proxy, it could extend more structs
}

type Proxy struct {
  Address string
  Port    String // this is either a string value or nil
}

```

This allows the element type approach to be modelled (but Go also allows behaviour [methods] to be added to structs). Typescript interfaces can also be used to model sets of properties, with Java-like multiple inheritance, in a similar fashion.

The name of an element type is called its *label*. It follows that an element type X, which extends element types Y and Z, has the name X, but can also be said to have three labels, X, Y and Z. Those three labels are all available for use in label expressions in graph patterns.



This use of labels for type identification mirrors the scheme in [Conceptual Graphs].

3.2.2 Vertex types

Given an element type named V , it is possible to define a vertex type, (V) , which has the name " V ". A vertex type composes its element type. A vertex type cannot be extended, it is sealed. No vertex can exist in a graph which is not of a specified vertex type.

3.2.3 Edge types

Given three element types, T , E and H , it is possible to define an edge type, $(T)\text{-}[E]\text{-}(H)$, which has the name " TEH ". An edge type composes tail, edge and head element types. It cannot be extended, it is sealed. No edge can exist in a graph whose tail and head vertices are not subtypes of the specified head and tail element types, and whose edge is not of the edge element type. The label set of an edge type is the label set of its edge element type alone.

3.2.4 Sharing graph types between graphs

A property graph type is a facet of a property graph. A property graph type can be copied from one graph to another with the LIKE clause:

```
CREATE GRAPH ConfigurationService
(
  -- clauses omitted
)
```

```
CREATE GRAPH SecondaryConfigurationService LIKE ConfigurationService
(
  -- clauses omitted
)
```

The user is free to define a property graph that contains the objects required to define its type, but does not contain any vertex type or edge type views. Such a graph can be treated as a graph type definition.

```
CREATE GRAPH ConfigurationServiceTemplate
(
  -- element, vertex and edge types only: no views
)
CREATE GRAPH PrimaryConfigurationService LIKE ConfigurationServiceTemplate
(
  -- no element type definitions required

  (CONFIGURATION) FROM configLDN, FROM configNY
  -- mappings defined for each view
  -- remaining clauses omitted
)
```

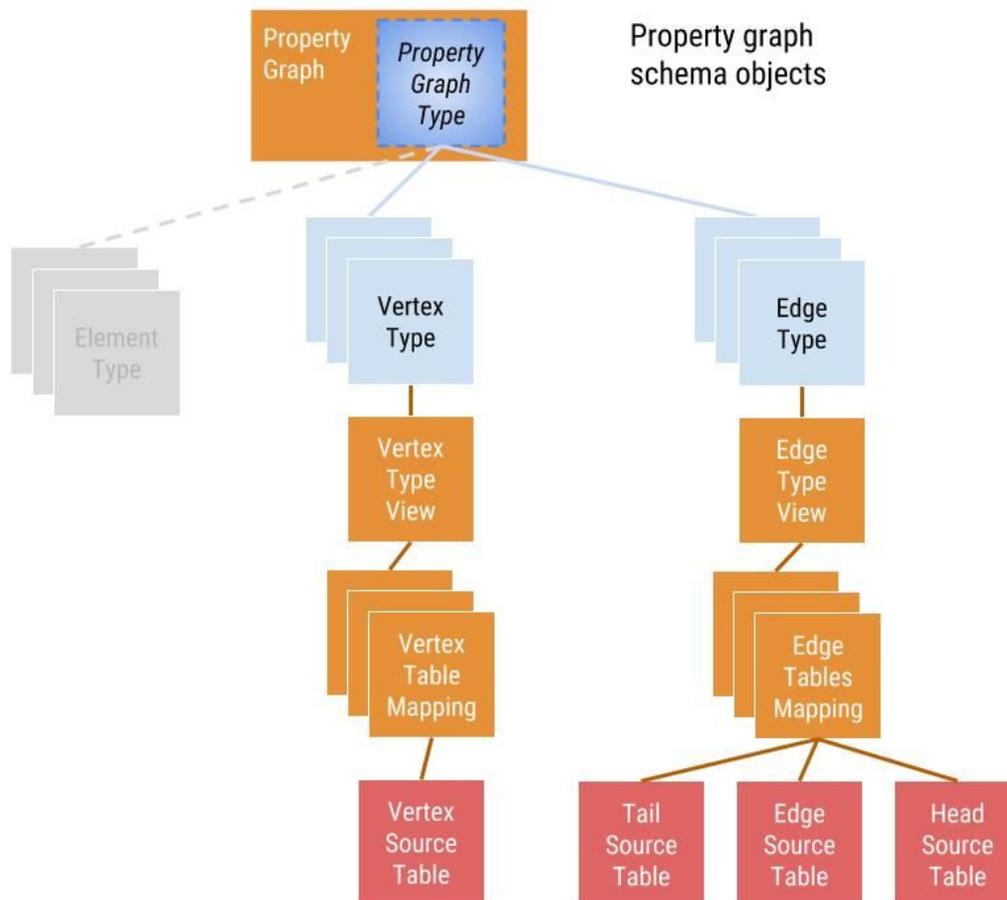
3.3 Property graph views over tables

Each property graph is an instance of a property graph type, but may include additional components, *vertex type views* and *edge type views* (these are collectively referred to as *element views* in this informative section).

These views map data in non-transient tables into a property graph, selecting rows which will engender elements, and columns from those rows which will be viewed as properties. There may be multiple mappings (from multiple distinct sets of tables) for each view.

Element views resemble derived tables, but differ in several ways:

1. a graph element may exist but have no data
2. the data attributes of a graph element are called properties, rather than columns
3. each element has a set of labels (names) associated with it, as well as a named type
4. elements may have relationships to other elements that are maintained by the SQL-implementation and are not visible as value correspondences (so they have no equivalent to foreign keys)
5. elements can have multiple unique (candidate) keys, but none is distinguished as primary



For many purposes it is helpful to think of a property graph as a new kind of complex view over multiple tables. The values of properties are stored in cells in persistent base tables, and are therefore SQL-data.

However, some of the data accessible in a graph, called *SQL-property graph-specific data*, is stored by the SQL-implementation in a way undefined by the standard. Property graph-specific data comprises label associations (one or more labels is associated with each graph element) and graph structures, including representations of the identity of graph elements and of linkages between elements. (An SQL-implementation is of course free to store property graph-specific data in hidden tables not created by the user, and may choose to make that data available to a user through conventional SQL queries that do not use SQL/PGQ extensions.)

The SQL-data available through graph element views over underlying source tables and SQL-property graph-specific data, taken together, are called *SQL-property graph data*.

As SQL/PGQ only offers read access to property graph data, data can only be inserted into a graph by means of SQL INSERT, UPDATE or MERGE statements on tables that form the

underlay for graph element views, and can only be removed from a graph by DELETE and MERGE statements.

3.4 Vertex type and edge type views

The relationship between data held or viewed as tables, and the graph structure and property values available when matching a graph pattern is as follows.

Each vertex in a graph, including any property values, is manifested by a row in a non-transient table, and is therefore stored in cells in one or more base tables. The properties of a vertex are a subset of the columns in the row that manifests the vertex: that subset may be empty.

In addition, each vertex may have one to many labels, one of which is the name of the label set. This label set derives from an element type: a vertex type composes one element type, which may extend other element types: each extended element type has a name or label.

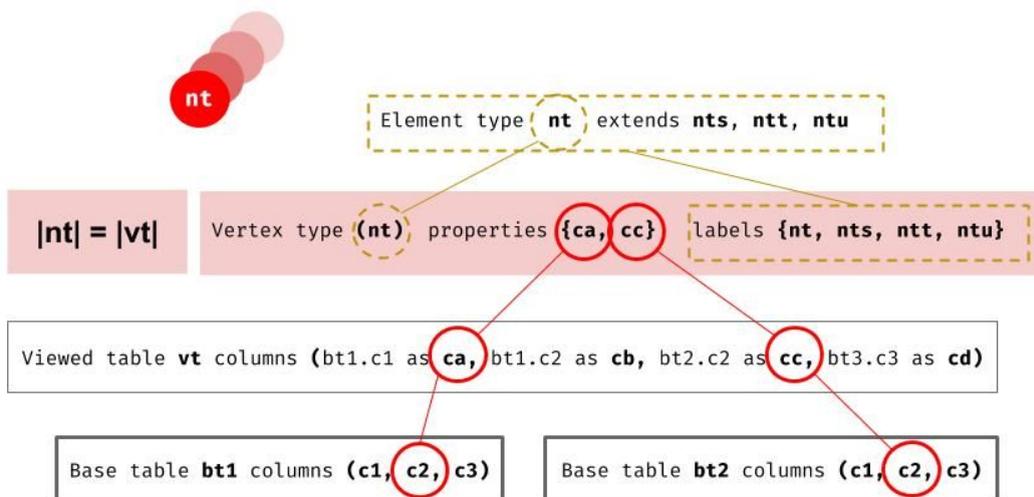
The name of a vertex type is the same as the name of its composed element type. The name classes for element type, vertex type and edge type are disjoint.

Example 1

Number of vertices = cardinality of source table.

Properties are mapped from a subset of columns.

Label set derives from element type composed by vertex type.



The syntax of <schema statement>s for defining a graph view over tables is not shown at this point. The definition of a viewed table in the above example, which draws on data from the two base tables illustrated, occurs in the normal fashion of SQL:2016.

The association of a vertex with an element type is stored as part of the property graph-specific data of a graph.

One graph may have an element whose existence and any property values derive from the same row in a table as an element in another graph. Such a connection is deemed to be unknown. Put another way: given a total injective function (one-to-one mapping) that yields a unique graph-qualified identity for any element in any graph in any schema in a given catalog, there is a partial function that evaluates to true or false over any two such input identities if and only if the identified elements belong to the same graph, and are the same element, but otherwise yields no result.

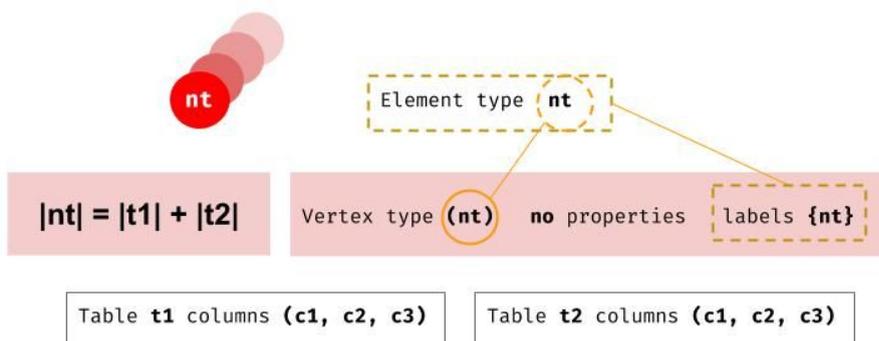
A second example shows how multiple tables can contribute to the generation of vertices of a particular vertex type, and that a vertex can exist without any properties, and with only one label.

Example 2

Number of vertices = cardinality of two source tables.

There are no properties.

There is one member of the label set, deriving from a “marker” element type.



Each edge in a graph which has property values is derived from the combination of a row in a non-transient table for each of the vertices that end the edge, and a row in a non-transient table which is used to generate edges. This has been referred to as a “three-table join”.

These three rows may be distinct; or the second may be the same as the first; and the third may be the same as the first (so at the limit, there may a single row playing three roles). The three rows may be drawn from one, two or three distinct tables. (Of course, viewed tables can be used to split a row, rendering a consistent, simple pattern of three distinct rows from three logical tables.)

It is possible, for example, to split a single row in a single table into a subset of columns that engenders one vertex, a subset of columns that engenders a second vertex, a third subset that engenders an edge connecting the two vertices, and a fourth subset that engenders a second edge between them.

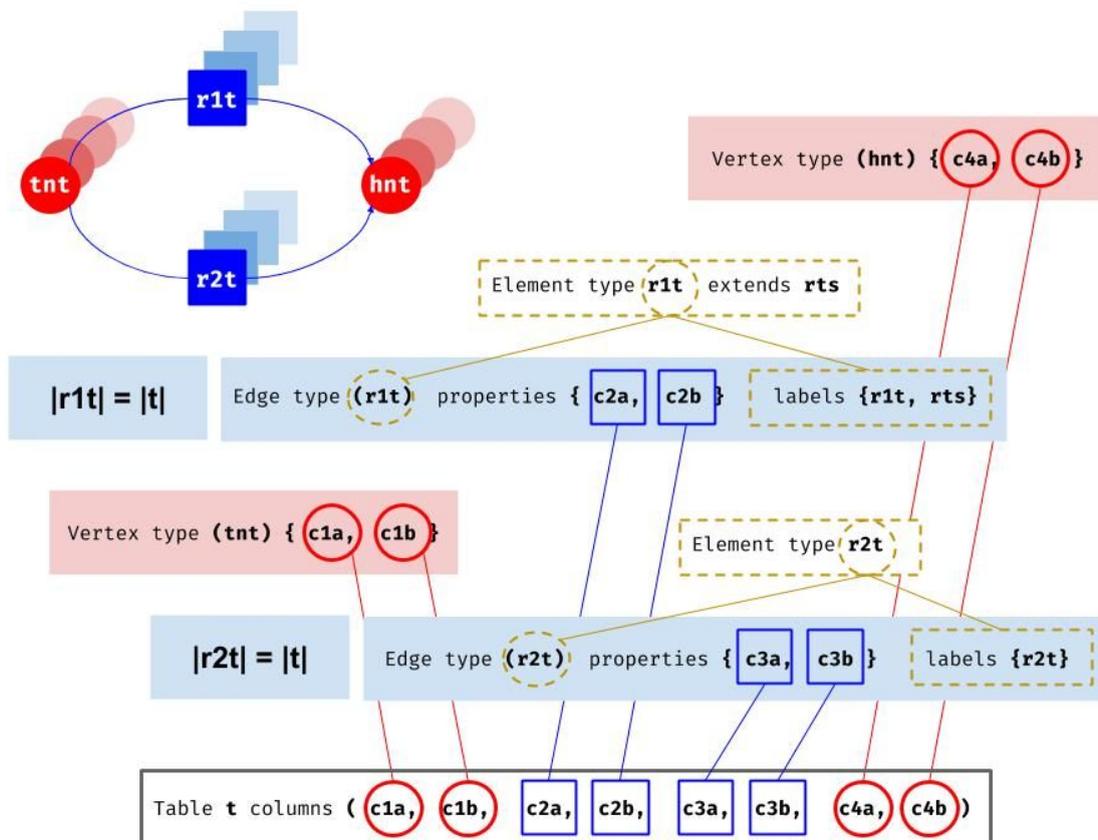
Example 3

Number of tail vertices = number of head vertices = cardinality of the source table.

Number of edges = twice the source cardinality.

Each edge type draws properties from different columns in the source row.

Each vertex and each edge is of a different type.



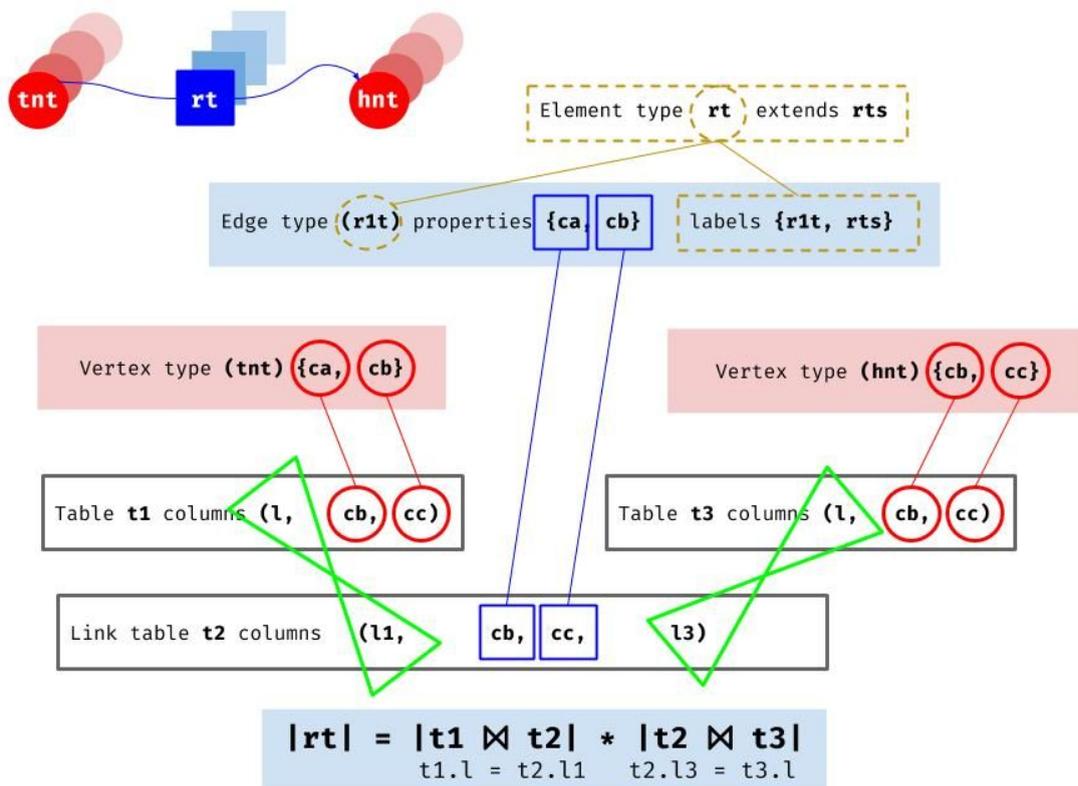
At the other extreme, a row in one table can engender one vertex, a row in a second table can engender the second vertex and a row in a third (link) table can engender one or more edges between them.

A single row in a link table can cause the generation of multiple edges. In one common case this will not happen, by design. A link table is often created with two foreign keys: one points at a primary key in one table, the other at a primary key in a second table. This arrangement generates a 1:1:1 relationship between a row in the first table, a row in the link table and a row in the second table. However, if we view a foreign key as a special case of a join key, then it is clear that we can (in principle) see a M:1:M relationship, where a single row in a link table can join to many rows in each of the other two tables. A 1:1:M relationship is more common.

Example 4

Number of vertices of a given type = cardinality of the respective source table(s).

Number of edges = product of the join cardinality for head vertices and that for tail vertices, assuming that the edge source table is a set with respect to its mapped columns and to the join key columns.



An example of a M:1:M relationship arises when we wish to answer questions like: “Which employees have a degree from the same university and have worked in the same department?”

The edges we wish to create to allow analysis of this kind are between degree or other diploma, and work assignment. The employment data is organized as a table for employee data, a table for assignments showing employee id, department, start and end date of assignment, and educational records in a third table, with a row per diploma, including a column which identifies the institution. For every combination of diploma and assignment, we wish to create an edge, and each edge will have a property which is the employee id.

Note that we are creating a graph view over existing data, so it is reasonable to “freeze” a particular query in the structure of the graph, just as it is reasonable to create views which express a restricted or special interest in base data. Perhaps we are interested in detecting correlations between departments, gender or other bias factors, degree subjects and

universities, and pay and rank. To what degree are people who have worked in finance and went to Cambridge University disproportionately highly paid or high up in the organization? Did they join the company at a similar time? Are they also disproportionately male? What proportion of people who went to Coventry University have ever worked in finance? There are many such questions that would be facilitated by the degree-assignment edge type.

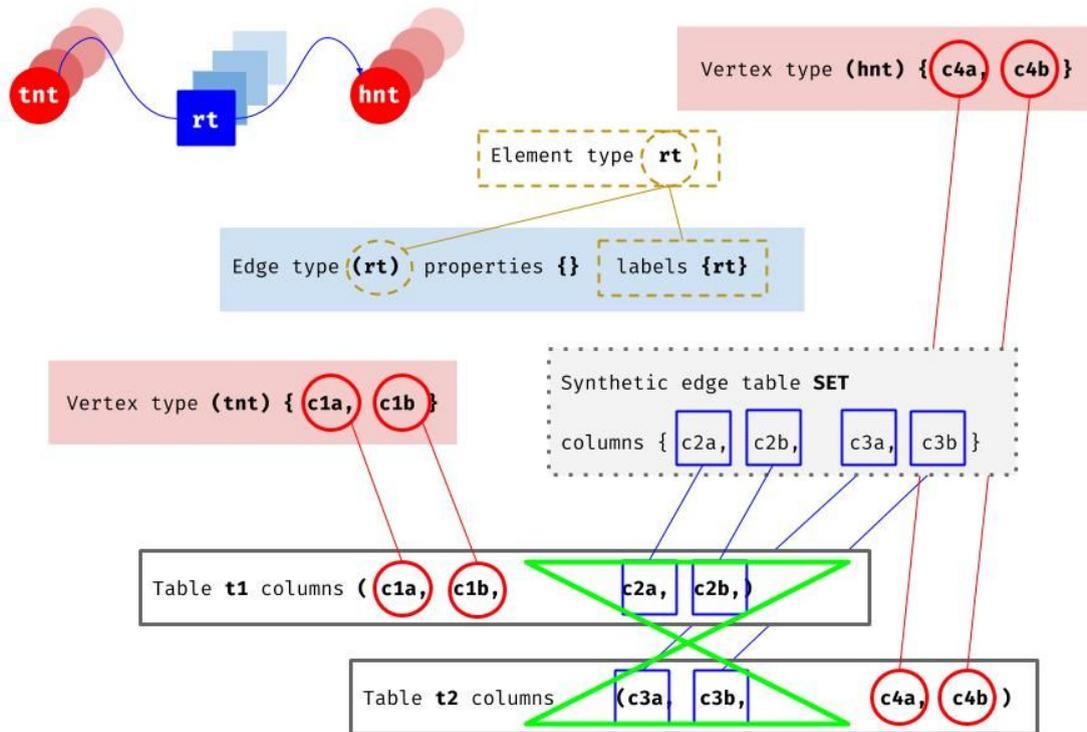
It is important to note that a single row may play a role in many different combinations of three rows (vertex, edge-generating, vertex). As we have already seen, it is possible for many edges to be generated from a single edge-generating row in a single combination: each combination may therefore create a set of edges.

For each edge that has no property values, there need only be two rows, one for each end of the edge. A cell in the first row links to a cell in the second row, the linkage is represented by an edge in the graph. A PK-FK relationship exhibits this pattern. More generally, an equijoin over the two columns will produce one to many rows for one row, and this can occur in both directions, producing a many row to many row set of resulting linkages or edges.

This can be modelled as three logical rows: one for the tail vertices, one for the head vertices, and a “synthesized link row” for the edges. The synthesized (logical) link row contains columns that map to columns in the tail row, and columns that map to the head row. The link row is a row in a synthesized (logical) edge table, that is distinct with respect to the union of the tail-mapping and head-mapping columns.

Example 5

Two-table join, with synthesized (logical) edge table. E.g., FK-PK edges pattern.



The following schematic DDL example illustrates the structure and semantics of a property graph:

```

CREATE GRAPH g1 OF g (
-- element types defined in graph type
  NodeA (a1 char(10), a2 integer), -- label NodeA, 2 properties
  NodeB <: NodeA (b1 integer), -- labels NodeB and NodeA, 3 properties
  NodeC EXTENDS NodeA (c1 varchar(255)), -- labels NodeC and NodeA,
  -- 3 properties

  NodeD (d1 char(10)),
  NodeE <: (NodeC, NodeD), -- NodeE multiply inherits, creates a label
  LINKS_TO, -- label LINKS_TO, no properties

-- vertex types
  (NodeA) FROM ("Node Table AA", "Node Table AB"),
  -- one view, two mappings
  (NodeB) FROM ("Base Table B"),
  (NodeC) FROM ("View C"),
  (NodeE), -- no mapping, no vertexes, might be ALTERed into existence

-- edge types
  (NodeA)-[LINKS_TO]->(NodeB)
  -- one view for the edge type, with two mappings
  -- mapping for data from "Node Table AA"
  FROM "Edge Table AB" relationship
  (NodeA) FROM "Node Table AA" tail_node

```

```
        JOIN relationship ON relationship.eab1 = tail_node.aa1
(NodeB) FROM "Base Table B" AS head_node
        JOIN relationship ON relationship.eab2 = head_node.b1
-- mapping for data from "Node Table AB"
FROM "Edge Table AB" relationship
(NodeA) FROM "Node Table AB") tail_node
        JOIN relationship ON relationship.eab1 = tail_node.ab1
(NodeB) FROM "Base Table B") head_node
        JOIN relationship ON relationship.eab2 = head_node.b1
-- the other views are omitted
-- ...
)
```

4 Rules

4.1 Property graph type

A property graph type is a schema object that is a component of a property graph.

4.1.1 Property graph type

A property graph type

- 1) Includes
 - a) A name, said to be the *graph type name*
 - b) A set of zero or more element types
 - c) A set of zero or more vertex types
 - d) A set of zero or more edge types
 - e) One super element type.

4.1.2 Element type

An element type ET

- 1) Includes
 - a) A name, said to be the *element type label*
 - b) A set of zero or more property types
 - c) A set of zero or more element keys
- 2) References
 - a) A set of zero or more element types, said to be *extended element types*
- 3) Exposes
 - a) A name, said to be the *element type name*
 - i) Note: Its purpose is to unambiguously identify element types, i.e. specify which elements types are allowed to coexist within a property graph type.
 - b) A set of one or more labels, which is said to be the *exposed label set*
 - i) Note: The exposed label set's purpose is to specify which labels the elements of an element type shall have.
 - c) A set of zero or more property types, which is said to be the *exposed property type set*
 - i) Note: The exposed property type set contains the property types which elements conforming to the element type have.
- 4) Let PGT be the property graph type that directly includes ET .
- 5) Let L be the element type label directly included in ET .
- 6) The element type name shall be L .
- 7) The element type name shall be unambiguous among all element types directly included in PGT .
- 8) Let EET_1, \dots, EET_n be the extended element types directly referenced by ET .
- 9) Let $els(ET)$ be the exposed label set of ET
 - a) Let $EL(ET)$ be a singleton set of ET 's element type label.
 - b) $els(ET)$ shall be the set union of $EL(ET)$ and all $els(EET_i)$, with $1 \text{ (one)} \leq i \leq n$.
- 10) Let $eps(ET)$ be the exposed property type set of ET

- a) Let $EP(ET)$ be the set of property types directly included in ET .
- b) $eps(ET)$ shall contain all property types contained in $EP(ET)$ and in all $eps(EET_i)$, such that:
 - i) Let PT_i be a property type contained in a EET_i , with $1 \text{ (one)} \leq i \leq n$.
 - ii) Let PT_j be a property type contained in a EET_j , with $1 \text{ (one)} \leq j \leq n$ and $i \neq j$.
 - iii) For any two ET_i and ET_j which have the same name
 - (1) ET_i and ET_j shall have the same data type.
 - (2) $eps(ET)$ shall contain only either ET_i or ET_j .
 - (3) If ET_i and ET_j have the same nullability characteristic, whether $eps(ET)$ contains ET_i or ET_j shall be implementation specific.
 - (4) If ET_i and ET_j have different nullability characteristics, $eps(ET)$ shall contain
 - (a) ET_i if ET_i is known not nullable
 - (b) ET_j if ET_j is known not nullable

4.1.3 Super element type

The distinguished super element type SET

- 1) Is an element, that
 - a) Includes
 - i) No element type label.
 - ii) Zero property types.
 - iii) Zero element keys.
 - b) References
 - i) Zero extended element types.
 - c) Exposes
 - i) No element type name.
 - ii) An empty exposed label set.
 - iii) An empty exposed property type set.

4.1.4 Vertex type

A vertex type VT

- 1) References
 - a) One element type, said to be the *referenced element type*
- 2) Exposes
 - a) A name, said to be the *vertex type name*
- 3) Let PGT be the property graph type that directly includes VT .
- 4) If RET is the element type referenced by VT , then RET must be an element type directly included in PGT .
- 5) Let L be the element type label directly included in RET .
- 6) The vertex type name shall be L .
- 7) The vertex type name shall be unambiguous among all vertex types directly included in PGT .

4.1.5 Edge type

A edge type *ET*

- 1) References
 - a) One element type, said to be the *tail element type*
 - b) One element type, said to be the *edge element type*
 - c) One element type, said to be the *head element type*
- 2) Exposes
 - a) A name, said to be the *edge type name*
- 3) Let *PGT* be the property graph type that directly includes *ET*.
- 4) If *TET* is a tail element type referenced by *ET*, then *TET* must be an element type directly included in *PGT*.
- 5) If *EET* is all the edge element types referenced by *ET*, then *EET* must be an element type directly included in *PGT*.
- 6) If *HET* is a head element type referenced by *ET*, then *HET* must be an element type directly included in *PGT*.
- 7) Let *TL* be the element type label directly included in *TET*.
- 8) Let *EL* be the element type label directly included in *EET*.
- 9) Let *HL* be the element type label directly included in *HET*.
- 10) The edge type name shall be the triple (*TL,EL,HL*).
- 11) The edge type name shall be unambiguous among all edge types directly included in *PGT*.

4.1.6 Property type

A property

- 1) Includes
 - a) A name, said to be the *property name*
 - b) A data type
 - c) A nullability characteristic, that indicates whether the value from that property can be the null value. A nullability characteristic is either *known not nullable* or *possibly nullable*.
 - i) Note: Nullability characteristic as specified in ISO/IEC 9075-2:2016(E) 4.13 “Columns, fields, and attributes”.

4.1.7 Element key

A element key *EK*

- 1) Includes
 - a) A name, said to be the *element key name*
- 2) References
 - a) A set of one or more property types
- 3) Let *T* be the element type or edge type that directly includes *EK*.
- 4) For each property type *P* referenced by *EK*:
 - a) *P* shall be known not nullable
 - b) Cases:

- i) If T is an element type, P shall be an exposed property of T .
- ii) If T is an edge type, P shall be an exposed property of T , the tail element type referenced by T , or the head element type referenced by T .

NOTE xxx -- A vertex type does not need an element key distinct from the element key of its referenced element type, whereas the element key of an edge type can include properties of the tail or head vertices.

4.2 Schema statements

The following SQL-schema statement is defined:

- <property graph definition>

4.3 Property graph definition and manipulation

4.3.1 <property graph definition>

Function

Define a property graph as an instance of its included defined graph type and define the vertex type and edge type views that map data in non-transient tables to populate that instance with graph elements and property values, such that it can be operated upon as a pure property graph.

[Drafting note: We need a formal definition of a non-transient table.]

Format

```
<property graph definition> ::= CREATE PROPERTY GRAPH <property graph name>
    [ LIKE <referenced graph name> ]
    [ <left paren> <graph definition component list> <right paren> ]
```

```
<property graph name> ::= <schema qualified name>
```

```
<referenced graph name> ::= <property graph name>
```

```
<graph definition component list> ::= <graph definition component>
    [ <comma> <graph definition component> ]...
```

```
<graph definition component> ::=
    <element type declaration> |
    <vertex type definition> |
    <edge type definition>
```

```
<vertex type definition> ::= <vertex type declaration> [<vertex type view>]
```

```
<vertex type view> ::= <vertex type mapping> [<comma> <vertex type mapping>]...
```

```
<vertex type mapping> ::= FROM <table name>
    [<left paren> <column to property> [<comma> <column to property>]...
    <right paren>]
    [<candidate key assertion>]
```

```
<candidate key assertion> ::= UNIQUE <left paren> <column name>
    [ <comma> <column name>]... <right paren>
```

```
<edge type definition> ::= <edge type declaration> [<edge type view>]
```

<edge type view> ::= <edge type mapping> [<comma> <edge type mapping>]...

<edge type mapping> ::= FROM <table name> [[AS] <correlation name>]
[<left paren> <column to property> [<comma> <column to property>]...
<right paren>]
<tail mapping>
<head mapping>

<tail mapping> ::= <endpoint mapping>

<head mapping> ::= <endpoint mapping>

<endpoint mapping> ::= [<left paren> <referenced element type label>
<right paren>]
FROM <table name> [[AS] <correlation name>]
JOIN ON <join phrase> [AND <join phrase>]...

<join phrase> ::= <left correlation> <period> <left column>
{<equals operator> | <not equals operator>}
<right correlation> <period> <right column>

<left correlation> ::= <correlation name>

<right correlation> ::= <correlation name>

<left column> ::= <identifier>

<right column> ::= <identifier>

<column to property> ::= <column name> AS <property name>

Syntax rules

- 1) Let *PG* be the property graph defined by the <property graph definition> *PGD*. Let *PGN* be the <property graph name> simply contained in *PGD*.
- 2) *PGN* shall not identify an existing property graph descriptor
- 3) Case:
 - a) If a <referenced graph name> is specified in *PGD*
 - i) The <referenced graph name> shall identify an existing property graph descriptor *RPG*.
 - ii) Let *PGT* be the property graph type that is the *defined graph type of RPG*
 - iii) *PGT* is said to be *pre-defined*
 - b) Otherwise
 - i) let *PGT* be a new property graph type defined by *PGD*
 - ii) *PGT* is said to be *defined in-line*

- 4) Let NET be the number of <edge type declaration>s contained in PGD. Let NVTD be the number of <vertex type definition>s contained in PGD. Let NEGTD be the number of <edge type definition>s contained in PGD.
- 5) If PGT is *pre-defined*, NET shall be zero.
- 6) If <graph definition component list> GDCL is specified, then
 - a) Let $ETD_{i,1}, \dots, ETD_{i,n_i}$ be the <element type declaration>s *ETD* simply contained in GDCL, with *L* being the <element type label> simply contained in an *ETD* and *stratum(L)* being equal to *i* and $n_i, 0 \leq n_i \leq NET$, being the number of <element type declaration>s with <element type label> with *stratum(i)*
 - i) Note: *stratum(L)* is specified according to the rules for <element type declaration>. The word *stratum* refers to the concept of stratification, i.e. layering of dependent declarations.
 - b) Let *M* be the maximum of *stratum(EL)* for all <element type label>s *EL* simply contained in GDCL
 - c) Let $VTD_i, 0 \text{ (zero)} \leq i \leq NVTD$, be the *i*-th <vertex type definition> in GDCL.
 - d) Let $EGTD_i, 0 \text{ (zero)} \leq i \leq NEGTD$, be the *i*-th <edge type definition> in GDCL.
 - e) GDCL is effectively replaced by a <graph definition component list> *OGDCL*, for *i*, $1 \text{ (one)} \leq i \leq M$, of the form:

$$\begin{array}{l}
 ETD_{1,1} \dots ETD_{1,n_1} \\
 \dots \\
 ETD_{M,1} \dots ETD_{M,n_M} \\
 VTD_1 \dots VTD_{NVTD} \\
 EGD_1 \dots EGD_{NEGTD}
 \end{array}$$

Note xxx This rule stratifies the <graph definition component>s and by doing so ensures that for the processing of an <element type declaration> *ETD*, all <element type declaration> referenced by *ETD* have already been processed and similarly for <vertex type definition>s and <edge type definition>s.

Note xxx This rule specifies a partial order of <graph definition component>. If two <graph definition component>s are not discriminated by this partial order, their processing order is insignificant for the semantics of the <graph definition component list>.

- 7) Let $VTDC_i$ be the <vertex type declaration> in VTD_i . Let VTL_i be the <label> in the <referenced element type label> in $VTDC_i$.
 - a) Each VTL_i shall be different ($VTL_i \neq VTL_j$ for $i \neq j$)
 - b) If PGT is *pre-defined*, VTL_i shall be the name of a vertex type descriptor $DVTD_i$ in PGT
- 8) Let NVM_i be the number of <vertex type mapping>s in the <vertex type view> in VTD_i .
- 9) If NVM_i is greater than zero, let $VM_{i,j}, 1 \text{ (one)} \leq j \leq NVM_i$ be the *j*-th <vertex type mapping> in VTD_i . Then
 - a) The <table name> $VMTN_{i,j}$ in $VM_{i,j}$ shall be different from the <table name>s in all other <vertex type mapping>s in PGD. ($VMTN_{i,j} \neq VMTN_{p,q}$ unless $i = p, j = q$)

NOTE xxx -- the table names are required to be different to allow the references in the <endpoint mapping>s in <edge type mapping>s to unambiguously refer to the set of vertices derived from a <vertex type mapping>. If more than one <vertex type mapping> refers to the same underlying table (probably taking different columns), views should be defined to allow different <table name>s.

Drafting note : if in-line views are supported, the syntax will need to define an unambiguous alias for the view.

- b) Let $VMTBL_{ij}$ be the table identified by $VMTN_{ij}$.
 - c) Let $NVCP_{ij}$ be the number of <column to property> in VM_{ij} . If $NVCP_{ij} > 0$ then
 - i) Let $VMCP_{ij,k}$, $1 \text{ (one)} \leq k \leq NVCP_{ij}$ be the k -th <column to property> in VM_{ij}
 - ii) The <column name> $VMCPC_{ij,k}$ in $VMCP_{ij,k}$ shall be the name of a column in $VMTBL_{ij}$
 - iii) The <property name> $VMCPP_{ij,k}$ in $VMCP_{ij,k}$ shall be different from the <property name>s in all other <column to property>s in VM_{ij}
 - d) If VM_{ij} contains a <candidate key assertion>, $VMCK_{ij}$
 - i) Let $NVMCK_{ij}$ be the number of <column name>s in $VMCK_{ij}$. Let $VMCKC_{ij,l}$, $1 \text{ (one)} \leq l \leq NVMCK_{ij}$ be the l -th <column name> in $VMCK_{ij}$
 - ii) $VMCKC_{ij,l}$ shall be the name of a column in $VMTBL_{ij}$
- 10) Let $ETDC_i$ be the <edge type declaration> in $EGTD_i$. Then
- a) $ETDC_i$ shall be different from all other <edge type declaration> in $OGDCL(EV_i \leftrightarrow EV_j)$ unless $i = j$)
 - b) Let $EVRTET_i$ be the <referenced tail element type> in $ETDC_i$
 - c) Let $EVRHET_i$ be the <referenced head element type> in $ETDC_i$
 - d) If PGT is *pre-defined*, $ETDC_i$ shall not contain a <element key declaration>
- 11) Let NEM_i be the number of <edge type mapping>s in the <edge type view> in $EGTD_i$.
- 12) If NEM_i is greater than zero, let EM_{ij} , $1 \text{ (one)} \leq j \leq NEM_i$, be the j -th <edge type mapping> in EV_i .
- a) Let $EMTBL_{ij}$ be the table identified by the <table name> contained in EM_{ij}
 - b) Let $NEMCP_{ij}$ be the number of <column to property> in EM_{ij} . If $NEMCP_{ij}$ is greater than 0 then:
 - i) Let $EMCP_{ij,k}$, $1 \text{ (one)} \leq k \leq NEMCP_{ij}$, be the k -th <column to property> in EM_{ij} .
 - ii) The <column name> $EMCPC_{ij,k}$ in $EMCP_{ij,k}$ shall be the name of a column in $EMTBL_{ij}$
 - iii) The <property name> $EMCPP_{ij,k}$ in $EMCP_{ij,k}$ shall be different from the <property name>s in all other <column to property>s in $EMCP_{ij,k}$
 - c) Case:

- i) If EM_{ij} directly contains a <correlation name> let $EMCN_{ij}$ be that <correlation name>
 - ii) Otherwise, let $EMCN_{ij}$ be the <qualified identifier> in the <table name> contained in EM_{ij}
- d) Let $EMTEM_{ij}$ be the <endpoint mapping> in the <tail mapping> in EM_{ij} . Let $EMTVTBL_{ij}$ be the table identified by the <table name> contained in $EMTEM_{ij}$.
- i) $EMTVTBL_{ij}$ shall be the same as exactly one table $VMTBL_{p,q}$ (i.e. a table specified in a <vertex type mapping> in PGD) for 1 (one) $\leq p \leq NVTD$ and 1 (one) $\leq q \leq NVM_p$. Then Case
 - (1) If a <referenced element type label> is present in $EMTEM_{ij}$, let $EMTVT_{ij}$ be the <label> in that <referenced element type label>. It shall be the same as the <label> in the <referenced element type label> in <vertex type declaration> $VTRD_p$ for the same value of p .
 - (2) If no <referenced element type label> is present in $EMTEM_{ij}$, let $EMTVT_{ij}$ be the <label> in the <referenced element type label> in <vertex type declaration> $VTDC_p$ for the same value of p .
 - ii) Case
 - (1) If $EMTEM_{ij}$ directly contains a <correlation name>, let $EMTCN_{ij}$ be that <correlation name>
 - (2) Otherwise, let $EMTCN_{ij}$ be the <qualified name> in the <table name> contained in $EMTEM_{ij}$.
 - iii) $EMTCN_{ij}$ shall be different from $EMCN_{ij}$
 - iv) Let $NEMTJP_{ij}$ be the number of <join phrase>s in $EMTEM_{ij}$. Let $EMTJP_{ij,k}$, 1 (one) $\leq k \leq NEMTJP_{ij}$, be the k -th <join phrase> in $EMTEM_{ij}$
 - v) Case
 - (1) If $EMTJP_{ij,k}$ contains a <equals operator>, let $EMTJPO_{ij,k}$ be True
 - (2) If $EMTJP_{ij,k}$ contains a <not equals operator>, let $EMTJPO_{ij,k}$ be False
 - vi) The <left correlation> in $EMTJP_{ij,k}$ shall be equivalent to either $EMCN_{ij}$ or $EMTCN_{ij}$.
 - vii) Case:
 - (1) If the <left correlation> in $EMTJP_{ij,k}$ is equivalent to $EMCN_{ij}$
 - (a) The <right correlation> in $EMTJP_{ij,k}$ shall be equivalent to $EMTCN_{ij}$
 - (b) The <left column> $EMTJPEC_{ij,k}$ in $EMTJP_{ij,k}$ shall be a column name in $EMTBL_{ij}$
 - (c) The <right column> $EMTJPVC_{ij,k}$ in $EMTJP_{ij,k}$ shall be a column name in $EMTVTBL_{ij}$
 - (2) If the <left correlation> in $EMTJP_{ij,k}$ is equivalent to $EMTCN_{ij}$
 - (a) The <right correlation> in $EMTJP_{ij,k}$ shall be equivalent to $EMCN_{ij}$

- (b) The <right column> $EMTJPEC_{i,j,k}$ in $EMTJP_{i,j,k}$ shall be a column name in $EMTBL_{i,j}$
- (c) The <left column> $EMTJPVC_{i,j,k}$ in $EMTJP_{i,j,k}$ shall be a column name in $EMTVTBL_{i,j}$
- e) Let $EMHEM_{i,j}$ be the <endpoint mapping> in the <head mapping> in $EM_{i,j}$. Let $EMHVTBL_{i,j}$ be the table identified by the <table name> contained in $EMHEM_{i,j}$.
- i) $EMHVTBL_{i,j}$ shall be the same as exactly one table $VMTBL_{p,q}$ (i.e. a table specified in a <vertex type mapping> in PGD) for $1 \text{ (one)} \leq p \leq NVTD$ and $1 \text{ (one)} \leq q \leq NVM_p$.
 - ii) Case
 - (1) If a <referenced element type label> is present in $EMHEM_{i,j}$, let $EMHVT_{i,j}$ be the <label> in that <referenced element type label>. It shall be the same as the <label> in the <referenced element type label> in <vertex type declaration> $VTRD_p$ for the same value of p .
 - (2) Otherwise (no <referenced element type label> is present in $EMHEM_{i,j}$), let $EMHVT_{i,j}$ be the <label> in the <referenced element type label> in <vertex type declaration> $VTDC_p$ for the same value of p .
 - iii) Case
 - (1) If $EMHEM_{i,j}$ directly contains a <correlation name>, let $EMHCN_{i,j}$ be that <correlation name>
 - (2) Otherwise, let $EMHCN_{i,j}$ be the <qualified name> in the <table name> contained in $EMTEM_{i,j}$.
 - iv) $EMHCN_{i,j}$ shall be different from $EMCN_{i,j}$ and $EMTCN_{i,j}$
 - v) Let $NEMHJP_{i,j}$ be the number of <join phrase>s in $EMHEM_{i,j}$. Let $EMHJP_{i,j,k}$, $1 \text{ (one)} \leq k \leq NEMHJP_{i,j}$, be the k -th <join phrase> in $EMHEM_{i,j}$
 - vi) Case
 - (1) If $EMHJP_{i,j,k}$ contains a <equals operator>, let $EMHJPO_{i,j,k}$ be True
 - (2) If $EMHJP_{i,j,k}$ contains a <not equals operator>, let $EMHJPO_{i,j,k}$ be False
 - vii) The <left correlation> in $EMHJP_{i,j,k}$ shall be equivalent to either $EMCN_{i,j}$ or $EMHCN_{i,j}$.
 - viii) Case:
 - (1) If the <left correlation> in $EMHJP_{i,j,k}$ is equivalent to $EMCN_{i,j}$
 - (a) The <right correlation> in $EMHJP_{i,j,k}$ shall be equivalent to $EMHCN_{i,j}$
 - (b) The <left column> $EMHJPEC_{i,j,k}$ in $EMTJP_{i,j,k}$ shall be a column name in $EMTBL_{i,j}$
 - (c) The <right column> $EMHJPVC_{i,j,k}$ in $EMHJP_{i,j,k}$ shall be a column name in $EMTVTBL_{i,j}$

- (2) If the <left correlation> in $EMHJP_{i,j,k}$ is equivalent to $EMHCN_{i,j}$
- (a) The <right correlation> in $EMHJP_{i,j,k}$ shall be equivalent to $EMCN_{i,j}$
 - (b) The <right column> $EMHJPEC_{i,j,k}$ in $EMHJP_{i,j,k}$ shall be a column name in $EMTBL_{i,j}$
 - (c) The <left column> $EMHJPVC_{i,j,k}$ in $EMHJP_{i,j,k}$ shall be a column name in $EMHVTBL_{i,j}$

General rules

- 1) If *PGT* is *defined in-line*, then a property graph type descriptor *PGTDS* is created that describes the property graph type *PGT*. *PGTDS* includes:
 - i) The descriptors of every element type of *PGT*, according to the Syntax Rules and General Rules of Subclause “<element type declaration>” applied to the <element type declaration>s contained in *OGDCL*, in the order in which they are specified.
 - ii) The descriptors of every vertex type of *PGT*, according to the Syntax Rules and General Rules of Subclause “<vertex type declaration>” applied to the <vertex type declaration>s contained in *OGDCL*, in the order in which they are specified.
 - iii) The descriptors of every edge type of *PGT*, according to the Syntax Rules and General Rules of Subclause “<edge type declaration>” applied to the <edge type declaration>s contained in *OGDCL*, in the order in which they are specified
 - iv) The descriptor of a super element type.

Note: The content of descriptor of the super element type is given from the definition of the super element type.
- 2) A pure property graph *PPG* is created. The *defined graph type* of *PPG* is *PGT*
- 3) *PPG* contains
 - a) The property graph name *PGN*
 - b) For each vertex type descriptor *DVTD* in *PGT*, a possibly empty set of vertices with the labels and properties specified by *DVTD* by the procedures of General Rule 4) applied to the <vertex type view> whose <vertex type declaration> identifies *DVTD*
 - c) For each edge type descriptor *DETD* in *PGT*, a possibly empty set of edges with the labels, properties specified by *DETD*, created by the procedures of General Rule 5) applied to the <edge type definition> whose <edge type declaration> identifies *DETD*
- 4) If *NVTD* is greater than zero, then for each <vertex type definition> VTD_i , $1 \text{ (one)} \leq i \leq NVTD$ for which NVM_i is greater than zero, then
 - a) Let $VTDS_i$ be the vertex type descriptor in *PGT* identified by VTL_i in $VTDC_i$
 - b) For each <vertex type mapping> $VM_{i,j}$, $1 \text{ (one)} \leq j \leq NVM_i$
 - i) If there are any <column to property> in $VM_{i,j}$, $VMCPP_{i,j,k}$, $1 \text{ (one)} \leq k \leq NVCP_{i,j}$ shall be the name of a property descriptor in $VTDS_i$,
 - ii) For each row R of $VMTBL_{i,j}$:

- (1) A vertex V is created and added to PPG as follows:
- V is assigned an implementation-dependent identifier that is unique within PPG
 - For each label included in $VTDS_p$, V is assigned a label with the same identifier
 - For each property included in $VTDS_p$, V has a property with the same name and declared type. The value of property P in V is obtained by:

Case:

 - If the name of P matches any $VMCPP_{i,j,k}$ for $1 \text{ (one)} \leq k \leq NVCP_{i,j}$, the evaluation of $VMCPC_{i,j,k}$ in R
 - If $VMTBL_{i,j}$ contains a column C whose name matches the name of P , ignoring case differences, the evaluation of C in R
 - otherwise the null value
- (2) The association between R and V is retained during the processing of General Rule 6 below. An implementation can assume the list of values for columns named in $VMCKC_{i,j,l}$ for $1 \text{ (one)} \leq l \leq NVMCK_{i,j}$, in R are unique across all rows of $VMTBL_{i,j}$.
- 5) If $NEGTD$ is greater than zero, then for each <edge type definition> $EGTD_i$, $1 \text{ (one)} \leq i \leq NEGTD$ for which NEM_i is greater than zero, then
- Let $EGTDS_i$ be the edge type descriptor in PGT identified by $ETDC_i$
 - For each <edge type mapping> $EM_{i,j}$, $1 \text{ (one)} \leq j \leq NEM_i$
 - $EMTVT_{i,j}$ shall identify a vertex type descriptor in PGD which is an extension of $EVRTET_i$
 - $EMHVT_{i,j}$ shall identify a vertex type descriptor in PGD which is an extension of $EVRHET_i$
 - If there are any <column to property> in $EM_{i,j}$, $EMCPP_{i,j,k}$, $1 \text{ (one)} \leq k \leq NEMCP_{i,j}$, shall be the name of a property descriptor in the edge type descriptor identified by $EGTDS_i$
 - For each row R of $EMTBL_{i,j}$
 - An edge template $ETEM$ is created as follows:
 - For each label included in $EGTDS_p$, $ETEM$ is assigned a label with the same identifier
 - For each property included in $EGTDS_p$, $ETEM$ has a property with the same name and declared type. The value of property P in $ETEM$ is obtained by: Case
 - If the name of P matches any $EMCPP_{i,j,k}$ the evaluation of $EMCPC_{i,j,k}$ in R
 - If $EMTBL_{i,j}$ contains a column C whose name matches the name of P , ignoring case differences, the evaluation of C in R
 - otherwise the null value

- (2) Let TVV be the set of vertices derived from the rows of $EMTVTBL_{ij}$ where for $1 \text{ (one)} \leq k \leq NEMTJP_{ij}$:
- (a) If $EMTJPO_{ij,k}$ is True, the value of $EMTJPEC_{ij,k}$ is equivalent to the value of $EMTJPVC_{ij,k}$
 - (b) If $EMTJPO_{ij,k}$ is False, the value of $EMTJPEC_{ij,k}$ is not equivalent to the value of $EMTJPVC_{ij,k}$
- (3) Let HVV be the set of vertices derived from the rows of $EMHVTBL_{ij}$ where for $1 \text{ (one)} \leq k \leq NEMHJP_{ij}$:
- (a) If $EMHJPO_{ij,k}$ is True, the value of $EMHJPEC_{ij,k}$ is equivalent to the value of $EMHJPVC_{ij,k}$
 - (b) If $EMHJPO_{ij,k}$ is False, the value of $EMHJPEC_{ij,k}$ is not equivalent to the value of $EMHJPVC_{ij,k}$
- (4) For each vertex TV in TVV
- (a) For each vertex HV in HVV , an edge E is created and included in PPG as follows
 - (i) The labels and properties are assigned from the template $E\text{TEM}$
 - (ii) TV is the tail vertex of E
 - (iii) HV is the head vertex of E

4.3.2 <element type declaration>

Function

Defines an element type.

Format

```

<element type declaration> ::=
  <element type label>
  [ <element type content> [ <element key declarations> ] ]

<element type content> ::=
  <element type extension> |
  { [ <element type extension> ] { <property type declarations> | <like clause> }
  }

<element type extension> ::=
  <extension demarcator> <extended element type label list>

<extension demarcator> ::=
  EXTENDS | <subtype symbol>

<subtype symbol> ::=
  <less than operator> <colon>

<extended element type label list> ::=
  <extended element type label>
  [ <comma> <extended element type label> ]...

<property type declarations> ::=
  <left property type declaration list delimiter>
  [ <property type declaration list> ]
  <right property type declaration list delimiter>

<left property type declaration list delimiter> ::=
  <left paren>

<right property type declaration list delimiter> ::=
  <right paren>

<property type declaration list> ::=
  <property type declaration> [ <comma> <property type declaration> ]...

```

Syntax Rules

- 1) Let ET be the element type defined by the <element type declaration> ETD .
- 2) Let PGT be the property graph type defined by the <property graph definition> that contains ETD .
- 3) Let L be the <element type label> simply contained in ETD .
- 4) Let $EETLL$ be the <extended element type label list> simply contained in ETD
 - a) Let $EETL_1, \dots, EETL_n$ be the <referenced element type label>s that are simply contained in the $EETLL$.
 - i) $refs(L)$ shall be the set union of all $refs(EETL_i)$ and the set of all $EETL_i$.
 - ii) No label contained in $refs(L)$ shall be equal to L .

- (1) Note: This prevents an element type from including itself.
- iii) Let M be the maximum of the specified $stratum(EL_i)$ of all EL_i contained in $refs(L)$.
 - iv) $stratum(L)$ shall be the M incremented by 1 (one).
- 5) If no <extended element type label list> is simply contained in ETD
 - a) $refs(L)$ shall be the empty set.
 - b) $stratum(L)$ shall be 1 (one).
 - 6) Each <extended element type label> in $EETLL$ shall identify an element type in PGT and no element type shall be identified more than once.
 - 7) All <property name>s simply contained in the <property type declarations> shall be unambiguous.
 - 8) If <like clause> LC is specified, then:
 - a) <like options> shall not be specified.
 - i) Note: This is to reuse the <like clause> ISO/IEC 9075-2:2016(E) 11.3 <table definition> when dealing with the <like options>.
 - b) Let LT be the table identified by the <table name> contained in LC .
 - c) Let D be the degree of LT . For i , $1 \text{ (one)} \leq i \leq D$:
 - i) Let LCD_i be the column descriptor of the i -th column of LT .
 - ii) Let LCN_i be the column name included in LCD_i .
 - iii) Let LDT_i be the data type included in LCD_i .
 - iv) If the nullability characteristic included in LCD_i is known not nullable, then let LNC_i be NOT NULL; otherwise, let LNC_i be the zero-length character string.
 - d) Let PD_i be a <property type declaration> of the form

$$LCN_i \ LDT_i \ LNC_i$$
 - e) LC is effectively replaced by a <property type declarations> of the form:

$$(PD_1, \dots, PD_D)$$

Access Rules

No.

General Rules

- 1) The degree of ET is initially set to 0 (zero); the General Rules of Subclause “<property type declaration>” specify the degree of L during the definition of the properties of L
- 2) An element type descriptor $ETDS$ is created that describes the element type ET . $ETDS$ includes:
 - a) The label L .
 - b) The property descriptors of every property of ET , according to the Syntax Rules and General Rules of Subclause “<property type declaration>”, applied to the <property type declaration>s simply contained in ETD .
 - c) References to
 - i) If an <extended element type label list> is simply contained in ETD , the element type descriptors referenced by the <extended element type label list>.

- ii) Otherwise, the super element type descriptor included in *PGT*.
- d) The element key descriptors of every element key of *ET*, according to the Syntax Rules and General Rules of Subclause “<element key declaration>”, applied to the <element key declaration>s simply contained in *ETD*.

Conformance Rules

No.

4.3.3 <vertex type declaration>

Function

Defines a vertex type.

Format

```
<vertex type declaration> ::=
  <left vertex type delimiter>
  [ <referenced element type label> ]
  <right vertex type delimiter>
<left vertex type delimiter> ::=
  <left paren>
<right vertex type delimiter> ::=
  <right paren>
```

Syntax Rules

- 1) Let *VT* be the vertex type defined by the <vertex type declaration> *VTD*.
- 2) Let *PGT* be the property graph type defined by the <property graph definition> that contains *VTD*.
- 3) If a <referenced element type label> *RETL* is simply contained in *VTD*, *RETL* shall identify an element type of *PGT*.

Access Rules

No.

General Rules

- 1) A vertex type descriptor *VTDS* is created that describes the vertex type *VT*. *VTDS* includes a reference to
 - a) If a <referenced element type label> is simply contained in *VTD*, the element type descriptors referenced by the <referenced element type label>.
 - b) Otherwise, the super element type descriptor included in *PGT*.

Conformance Rules

No.

4.3.4 <edge type declaration>

Function

Defines a edge type.

Format

```

<edge type declaration> ::=
  <referenced tail element type>
  <minus sign> <referenced edge element type> <right arrow>
  <referenced head element type>
  [ <element key declarations> ]
<referenced tail element type> ::=
  <left vertex type delimiter>
  [ <referenced element type label> ]
  <right vertex type delimiter>
<referenced edge element type> ::=
  <left bracket or trigraph> [ <referenced element type label> ]
  <right bracket or trigraph>
<referenced head element type> ::=
  <left vertex type delimiter>
  [ <referenced element type label> ]
  <right vertex type delimiter>

```

Syntax Rules

- 1) Let *EGT* be the edge type defined by the <edge type declaration> *EGTD*.
- 2) Let *PGT* be the property graph type defined by the <property graph definition> that contains *ETGD*.
- 3) If the <referenced edge element type> *REET* simply contained in *EGTD* simply contains a <referenced element type label>, that <referenced element type label> shall identify an element type of *PGT*.
- 4) If the <referenced tail element type> *RTET* simply contained in *EGTD* simply contains a <referenced element type label>, that <referenced element type label> shall identify an element type of *PGT*.
- 5) If the <referenced head element type> *RHET* simply contained in *EGTD* simply contains a <referenced element type label>, that <referenced element type label> shall identify an element type of *PGT*.

Access Rules

No.

General Rules

- 1) A edge type descriptor *EGTDS* is created that describes the edge type *EGT*. *EGTDS* includes:

- a) The edge element type reference to
 - i) If a <referenced edge element type> is simply contained in *EGTD*, the element type descriptors referenced by the <referenced edge element type>.
 - ii) Otherwise, the super element type descriptor included in *PGT*.
- b) The tail element type reference to
 - i) If a <referenced tail element type> *RTET* is simply contained in *EGTD*, the element type descriptors referenced by the <referenced element type label> simply contained in *RTET*.
 - ii) Otherwise, the super element type descriptor included in *PGT*.
- c) The head element type reference to
 - i) If a <referenced head element type> *RHET* is simply contained in *EGTD*, the element type descriptors referenced by the <referenced element type label> simply contained in *RHET*.
 - ii) Otherwise, the super element type descriptor included in *PGT*.
- d) The element key descriptors of every element key of *EGT*, according to the Syntax Rules and General Rules of Subclause “<element key declaration>”, applied to the <element type declaration>s simply contained in *EGTD*.

Conformance Rules

No.

4.3.5 <property type declaration>

Function

Defines a property.

Format

```

<property type declaration> ::=
  <property name> <property data type> [ <property constraint> ]
<property name> ::=
  <identifier>
<property data type> ::=
  <data type>
<property constraint> ::=
  NOT NULL

```

Syntax Rules

- 1) Let *P* be the <property name> of the <property type declaration>.
- 2) The declared type of the property is the data type specified by <data type>.
 - a) Note: <data type> as in ISO/IEC 9075-2:2016(E) 6.1 <data type>
- 3) If <property constraint> is contained in the <property type declaration>, the nullability characteristic shall be known not nullable otherwise the nullability characteristic shall be possibly nullable.

Access Rules

No.

General Rules

- 1) A <property type declaration> defines a property in an element type.
- 2) A data type descriptor is created that describes the declared type of the property being defined.
- 3) The degree of the element type *ET* being defined in the containing <element type declaration> is increased by 1 (one).
- 4) A property descriptor is created that describes the property being defined. The property descriptor includes:
 - a) *P*, the name of the property.
 - b) The data type descriptor of the declared type of the property.
 - c) The nullability characteristic of the property.

Conformance Rules

No.

4.3.6 <element key declaration>**Function**

Defines an element key.

Format

```

<element key declarations> ::=
  <element key declaration> [ [ <comma> ] <element key declaration> ]...
<element key declaration> ::=
  KEY
  <constraint name>
  <left paren> <key properties list> <right paren>
<key properties list> ::=
  <key property> [ <comma> <key property> ]...
<key property> ::=
  <key property on element>
  | <key property on tail>
  | <key property on head>
<key property on element> ::=
  <property name>
<key property on tail> ::=
  TAIL <property name>
<key property on head> ::=
  HEAD <property name>

```

Syntax Rules

- 1) Let *EKD* be the <element key declaration>.
- 2) Let *TD* be the <element type declaration>, <vertex type declaration>, or <edge type declaration> containing *EKD*.
- 3) Let *T* be the element type or edge type identified by the containing *TD*.
- 4) If *T* is an element type, <key properties list> shall only contain <key property on element>s.
- 5) Each <key property> in the <key properties list> shall identify a property *P*, and the same *P* shall not be identified more than once in the same <key properties list>.
 - a) Cases:
 - i) If <key property> is a <key property on element>, *P* shall be an exposed property type of *T*
 - ii) If <key property> is a <key property on tail>, *P* shall be an exposed property type of the tail element type referenced by *T*
 - iii) If <key property> is a <key property on head>, *P* shall be an exposed property type of the head element type referenced by *T*
 - b) *P* shall be known not nullable

Access Rules

No.

General Rules

- 1) A element key descriptor is created that describes the element key being defined. The element key descriptor includes:
 - a) <constraint name> contained in EKD.
 - i) Note: <constraint name> as in ISO/IEC 9075-2:2016(E) 5.4 Names and identifiers.
 - b) The set of properties reference in the <key properties list>.

Conformance Rules

No.

4.3.7 Names and identifiers

Function

Specify names.

[Drafting note: Need to modify ISO/IEC 9075-2:2016(E) 5.4 Names and identifiers]

Format

<element type label> ::=

<label>

<extended element type label> ::=

<label>

```
<referenced element type label> ::=  
  [ <colon> ] <label>  
<label> ::=  
  <identifier>
```

Syntax Rules

No.

Access Rules

No.

General Rules

No.

Conformance Rules

No.

4.3.8 Changes to 5.1, additional SQL terminal symbol

Add to Format:

```
<right arrow> ::=  
  ->
```

5 Glossary of symbols (Informative)

Symbol	Description
<i>C</i>	Column: a column in row <i>R</i>
<i>D</i>	The degree of <i>LT</i>
<i>DETD</i>	Defined Edge Type Descriptor: an edge type descriptor in the graph type <i>PGT</i>
<i>DVTD</i>	Defined Vertex Type Descriptor: a vertex type descriptor in the graph type <i>PGT</i>
<i>DVTD_i</i>	Defined Vertex Type Descriptor: the vertex type descriptor in the graph type <i>PGT</i> identified by the <label> in the <referenced element type label> in <vertex type declaration> <i>VTD_i</i>
<i>E</i>	Edge: an edge to be added to the property graph <i>PPG</i>
<i>EET</i>	Edge Element Types: referenced by <i>ET</i>
<i>EETLL</i>	Extended Element Type Label List: the <extended element type label list> simply contained in <i>ETD</i>
<i>EETL_i</i>	Referenced Element Type Label: the <referenced element type label> simply contained in <i>EETLL</i>
<i>EGT</i>	Edge Type: defined by <i>EGTD</i>
<i>EGTD</i>	Edge Type Declaration: the <edge type declaration>
<i>EGTDS</i>	Edge Type Descriptor: the edge type descriptor that is created which describes <i>EGT</i>
<i>EK</i>	Element Key
<i>EKD</i>	Element Key Declaration: the <element key declaration> contained in <i>TD</i>
<i>EL</i>	Element Type Label: the element type label directly included in <i>EET</i> Also: Element Type Labels: the <element type label>s simply contained in <graph type element declaration list>
<i>EL_i</i>	Element Type Label: the <i>i</i> -th element type label contained in refs(<i>L</i>)
<i>EM_{ij}</i>	Edge Type Mapping: the <i>j</i> -th <edge type mapping> in <i>EV_i</i>
<i>EMCN_{ij}</i>	Edge Mapping Correlation Name: the <correlation name> directly contained in <i>EM_{ij}</i> ; otherwise the <qualified identifier> in the <table name> contained in <i>EM_{ij}</i>
<i>EMCP_{ij,k}</i>	Edge Mapping Column-to-Property: the <i>k</i> -th <column to property> in <i>EM_{ij}</i>
<i>EMCPC_{ij,k}</i>	Edge Mapping Column-to-Property Column: the <column name> in <i>EMCP_{ij,k}</i> ; the name of a column in <i>EMTBL_{ij}</i>
<i>EMCPP_{ij,k}</i>	Edge Mapping Column-to-Property Property: the <property name> in <i>EMCP_{ij,k}</i> ; shall be the name of a property descriptor in <i>ETD_i</i>
<i>EMHCN_{ij}</i>	Edge Mapping Head Correlation Name: the <correlation name> directly contained by <i>EMHEM_{ij}</i> ; otherwise, the the <qualified name> in the <table name> contained in <i>EMTEM_{ij}</i>
<i>EMHEM_{ij}</i>	Edge Mapping Head Endpoint Mapping: the <endpoint mapping> in the <head mapping> in <i>EM_{ij}</i>

$EMHJP_{i,j,k}$	Edge Mapping Head Join Phrase: the k -th <join phrase> in $EMHEM_{i,j}$
$EMHJPEC_{i,j,k}$	Edge Mapping Head Join Phrase Edge Column: the <left column> (resp. <right column>) in $EMTJP_{i,j,k}$ shall be a column name in $EMTBL_{i,j}$
$EMHJPO_{i,j,k}$	Edge Mapping Head Join Phrase Operator: True if $EMHJP_{i,j,k}$ contains a <equals operator>; False if $EMHJP_{i,j,k}$ contains a <not equals operator>
$EMHJPVC_{i,j,k}$	Edge Mapping Head Join Phrase Vertex Column: the <right column> (resp. <left column>) in $EMHJP_{i,j,k}$ shall be a column name in $EMTVTBL_{i,j}$
$EMHVT_{i,j}$	Edge Mapping Head Vertex Type: the label of the vertex type descriptor for the head mapping in edge mapping $EM_{i,j}$
$EMHVTBL_{i,j}$	Edge Mapping Head Vertex Table: the table identified by the <table name> contained in $EMHEM_{i,j}$
$EMTBL_{i,j}$	Edge Mapping Table: the table identified by the <table name> contained in $EM_{i,j}$
$EMTCN_{i,j}$	Edge Mapping Tail Correlation Name: the <correlation name> directly contained in $EMTEM_{i,j}$; otherwise the <qualified name> in the <table name> contained in $EMTEM_{i,j}$
$EMTEM_{i,j}$	Edge Mapping Tail Endpoint Mapping: the <endpoint mapping> in the <tail mapping> in $EM_{i,j}$
$EMTJP_{i,j,k}$	Edge Mapping Tail Join Phrase: the k -th <join phrase> in $EMTEMi,j$; <left correlation> (resp. <right correlation>) in this shall be equivalent to either $EMCNi,j$ or $EMTCNi,j$
$EMTJPEC_{i,j,k}$	Edge Mapping Tail Join Phrase Edge Column: the <left column> (resp. <right column>) in $EMTJP_{i,j,k}$ shall be a column name in $EMTBL_{i,j}$
$EMTJPO_{i,j,k}$	Edge Mapping Tail Join Phrase Operator: True if $EMTJP_{i,j,k}$ contains a <equals operator>; False if $EMTJP_{i,j,k}$ contains a <not equals operator>
$EMTJPVC_{i,j,k}$	Edge Mapping Tail Join Phrase Vertex Column: the <right column> (resp. <left column>) in $EMTJP_{i,j,k}$ shall be a column name in $EMTVTBL_{i,j}$
$EMTVT_{i,j}$	Edge Mapping Tail Vertex Type: the label of the vertex type descriptor for the tail mapping in edge mapping $EM_{i,j}$
$EMTVTBL_{i,j}$	Edge Mapping Tail Vertex Table: the table identified by the <table name> contained in $EMTEM_{i,j}$
ET	Element Type: directly included by PGT
ETD_i	Element Type Declaration: the i -th <element type declaration> in $OGDCL$
$ETD_{p,q}$	Element Type Declaration: the q -th <element type declaration> with $stratum(p)$ in $GDCL$, in the context of replacing $GDCL$ with $OGDCL$
$ETDC_i$	Edge Type Declaration: the <edge type declaration> in $EGTD_i$
$ETDS$	Element Type Descriptor: describes ET
EEM	Edge Template: an edge template derived from the edge input table that becomes zero or more edges when the endpoint vertices are determined
EV_i	Edge Type Definition: the i -th <edge type definition> in PGD
$EVRHET_i$	Edge View Referenced Head Element Type: head element type of the edge type for edge view i

$EVRTET_i$	Edge View Referenced Tail Element Type: tail element type of the edge type for edge view i
$GDCL$	Graph Definition Component List: the <graph definition component list>
HET	Head Element Type: referenced by ET
HV	Head Vertex: a vertex in HVV , head vertex of the edge
HVV	Head Vertices: the set of vertices derived from the rows of the head input table for an edge $EMHVTBL_{i,j}$
L	Element Type Label: which is directly included in ET and the value of the element type name Also: Element Type Label: which is directly included in RET , and the value of the vertex type name Also: Element Type Label: the <element type label> which is simply contained in ETD
LC	Like Clause: <like clause>
LCD_i	Column Descriptor: the i -th column of LT
LCN_i	Column Name: included in LCD_i
LDT_i	Data Type: included in LCD_i
LNC_i	Character String: this is "NOT NULL" if the nullability characteristic included in LCD_i is known not nullable, otherwise, it is the zero-length character string
LT	Table: identified by the <table name> contained in C
M	Maximal of $stratum(EL)$ for all <element type label>s EL simply contained in <graph definition componen list> Also: Maximal of $stratum(EL_i)$ contained in $refs(L)$
$NEGTD$	The number of <edge type declaration>s contained in PGD
NEM_i	Number of Edge Mapping: number of <edge type mapping>s in <edge type definition> i
$NEMCP_{i,j}$	Number of Edge Mapping Column to Property: number of <column to property>s in <edge type mapping> $EM_{i,j}$
$NEMHJP_{i,j}$	Number of Edge Mapping Head Join Phrase: number of <join phrase>s in <head mapping> of <edge type mapping> $EM_{i,j}$
$NEMTJPI_{i,j}$	Number of Edge Mapping Tail Join Phrase: number of <join phrase>s in <tail mapping> of <edge type mapping> $EM_{i,j}$
NET	The number of <edge type declaration>s contained in PGD
$NVCP_{i,j}$	Number of Vertex Mapping Column to Property: number of <column to property>s in <vertex type mapping> $VM_{i,j}$
NVM_i	Number of Vertex Mapping: number of <vertex type mapping>s in <vertex type definition> VTD_i
$NVMCK_{i,j}$	Number of Vertex Mapping Candidate Key: number of columns named in <candidate key assertion> in <vertex type mapping> $VM_{i,j}$
$NVTD$	Number of Vertex Views

<i>OGDCL</i>	Ordered Graph Definition Component List: the <graph definition component list> reordered to ensure components come after any components they reference
<i>P</i>	Property: a property in <i>E_{TEM}</i> Also: Property Type: referenced by <i>E_K</i> Also: Property Name: the <property name> of the <property type declaration> Also: Property: a property identified by each <key property> in the <key properties list>
<i>PD_i</i>	Property Type Declaration: a <property type declaration>
<i>PG</i>	Property Graph: property graph defined by the <property graph definition>
<i>PGD</i>	Property Graph Definition: the <property graph definition>
<i>PGN</i>	Property Graph Name: the property graph name contained in <i>PPG</i>
<i>PGT</i>	Property Graph Type: defined by <property graph definition>, described by <i>PGTDS</i> , and directly includes <i>ET</i> and <i>VT</i>
<i>PGTDS</i>	Property Type Descriptor: describes <i>PGT</i>
<i>PPG</i>	Pure Property Graph
<i>R</i>	Row : a row in an input table for vertices (<i>VMTBL_i</i>) or edges (<i>EMTBL_i</i>)
<i>REET</i>	Referenced Edge Element Type: a <referenced edge element type> that is simply contained in <i>EGTD</i>
<i>RET</i>	Element Type: referenced by vertex type <i>VT</i>
<i>RETL</i>	Referenced Element Type Label: a <referenced element type label> that is simply contained in <i>VTD</i> and identifies an element type of <i>PGT</i>
<i>RHET</i>	Referenced Head Element Type: a <referenced head element type> that is simply contained in <i>EGTD</i> , and which simply contains the <referenced element type label> identifying an element type of <i>PGT</i> .
<i>RPG</i>	Referenced Property Graph: an existing property graph descriptor
<i>RTET</i>	Referenced Tail Element Type: a <referenced tail element type> that is simply contained in <i>EGTD</i> , and which simply contains the <referenced element type label> identifying an element type of <i>PGT</i> .
<i>SET</i>	The distinguished super element type
<i>T</i>	Element Type, Vertex Type or Edge Type: directly includes <i>E_K</i> or identified by <i>TD</i>
<i>TD</i>	Element Type Declaration: the <element type declaration>, <vertex type declaration>, or <edge type declaration> containing <element key declaration> <i>E_{KD}</i> .
<i>TET</i>	Tail Element Type: referenced by <i>ET</i>
<i>TV</i>	Tail Vertex: a vertex in <i>TV</i>
<i>TVV</i>	Tail Vertices: the set of vertices derived from the rows of <i>EMTVTBL_{i,j}</i>
<i>V</i>	Vertex: a vertex in <i>PPG</i>

VM_{ij}	Vertex Mapping: the j -th <vertex type mapping> contained in VTD_i
$VMCK_{ij}$	Vertex Mapping Candidate Key: the <candidate key assertion> in vertex mapping VM_{ij}
$VMCKC_{i,j,l}$	Vertex Mapping Candidate Key Column: the l -th column name listed in $VMCK_{ij}$ (k is already defined in the sub-rule defining this)
$VMCP_{i,j,k}$	Vertex Mapping Column to Property: the k -th <column to property> in VM_{ij}
$VMCPC_{i,j,k}$	Vertex Mapping Column to Property Column: the <column name> (in $VMCP_{i,j,k}$) shall be the name of a column in $VMTBL_i$
$VMCPP_{i,j,k}$	Vertex Mapping Column to Property Property: the <property name> in $VMCP_{i,j,k}$
$VMTBL_{ij}$	Vertex Mapping Table: the table identified by $VMTN_{ij}$
$VMTN_{ij}$	Vertex Mapping Table Name: the <table name> in VM_{ij}
VT	Vertex Type: directly included by PGT , and defined by <vertex type declaration> VTD and described by vertex type descriptor $VTDS$
VTD	Vertex Type Declaration: defines (or, if PGT is <i>pre-defined</i> , identifies) the vertex type VT
$VTDS_i$	Vertex Type Descriptor: the name of a vertex type descriptor in PGT identified by the i -th <vertex type definition> in $OGDCL$
$VTDS$	Vertex Type Descriptor: describes the vertex type VT
VTL_i	Label: the <label> in the <referenced element type label> in $VTDC_i$
$VTDC_i$	Vertex Type Reference or Descriptor: the <vertex type declaration> in VTD_i

6 Possible Problems

These problems may be addressed in revisions of this CP before Brisbane, or in separate proposals, as appropriate.

- 1) It is considered essential that property graph types and the types they contained can be altered. There is the ambition to add this functionality in form of ALTER statements to the specification with a separate change proposal.
- 2) There is the ambition to allow (1) the definition element types, vertex types, element types, property types independently from their containing types and (2) the citation of independently defined types and types defined in a different scope (e.g. in a different property graph type) by means like OF and LIKE.
- 3) We wish to augment vertex types with the possibility of a vertex cardinality constraint and edge types with the possibility of a edge cardinality constraint, a tail cardinality constraint, and a head cardinality constraint.
- 4) For an element key, <key property on tail> and <key property on head> we currently use the keyword TAIL and HEAD as demarcation. We aim to find a keyword-less syntax, just as we have been able to eliminate any use of EDGE, VERTEX, NODE, RELATIONSHIP and so on.
- 5) We intend to examine alternative syntax for simple two-table joins in a mapping, which currently involves an unnatural-seeming self-join on a table viewed through the prism of two correlation names. Perhaps like this:

```
(NodeA)-[CONNECTS_TO]->(Node_B)
FROM A join B ON A.a1 = B.b2
```

- 6) Changes required for [\[Framework:2020\]](#)
 - a) 4.2. SQL-data is no longer the only data in a database. Add SQL-property-graph data, and SQL-property-graph-specific data.
 - b) Two new name classes of schema objects – PGT and PG
 - c) PGT and PG component naming
 - i) list of usages of uniquely named components a.b.c.d etc
 - d) 4.6.* clauses need to be added to for graphs
- 7) Changes required to [\[Foundation:2020\]](#)
 - a) Section 11. 1 <schema definition>, Section 13.4 <SQL procedure statement> new <sql schema statement>s to be added, as per part 4 of this proposal.
- 8) Changes required for [\[Schemata:2020\]](#)
 - a) Add new views to DS and IS
 - b) Describe DG as a tabular view on a graph

7 Language Opportunities

- 1) For element types, any two exposed property types with the same name are required to have same data type. There is the possibility to relax this requirement in the future, based on the basis of union compatibility.
- 2) There is currently no restriction on the tail element type and the head element type. However, it possible to require a covariance [Cardelli1984] between the edge element type on the one hand and the tail element type and the head element type on the other hand. Covariance means that the subtyping relations of simple types are preserved for complex types composed from the simple types. In our case that means that the subtyping relation of the edge element types referenced edge types have to line up with the subtyping relations of element types referenced the tail and head element types referenced the edge types.
- 3) The column-to-property constructs in vertex and edge mappings could be extended to allow expressions instead of just column names
- 4) In-line views could be accommodated as input “tables” for vertex and edge. It would be necessary to have syntax to name an inline view used as input for vertices so that it could be referred to explicitly in edge mappings.
- 5) Graph-scoped view objects which represent the data of a vertex type, or of an edge type as normal SQL tables could automatically generated in the schema.

```
CREATE GRAPH mycatalog.myschema.foo (A (a char(10)), (A))
```

would generate these objects:

```
graph object: mycatalog.myschema.foo  
graph type object: mycatalog.myschema.foo  
graph-induced view: mycatalog.myschema.foo.A
```

8 Examples (Informative)

8.1 Element type

An element type is a schema object that defines the structure of element, i.e. which labels and properties they expose. An element type includes a label and zero or more property types, and references one or more other element type(s), which it extends. The label of an element type has to be unambiguous among all element types. Element types cannot have instances in a property graph; hence, they are said to be abstract.

An element type can be extended by zero or more other element types. Type extension follows the concept of mixin type composition. An element type extending other element types exposes a union of all characteristics (labels, properties, rights to have edges of certain types) of the element types it extends. An element type must not extend itself, either directly or indirectly.

8.1.1 Examples

- The element type declaration


```
Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
```

 declares an element type with label `Person` and two included property types, `firstName` of data type `VARCHAR(100)` (and not nullable) and `birthday` of data type `DATE`. Note that this document does not specify anything regarding the set of available and allowed data types.
- The element type declaration


```
Intellectual
```

 declares an element type with label `Intellectual` and with no included property types.
- Given the element types


```
Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
```

 and nothing else, the element type declaration


```
Student EXTENDS Person ()
```

 declares an element type with label `Student`, which extends the element type `Person` and consequently exposes the two property types `firstName` and `birthday`. The element type does not include any additional property types. It exposes the labels `Student` and `Person`.
- Given the element types


```
Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
```

```
Intellectual (cigarettesPerMonth INT)
```

 and nothing else, the element type declaration


```
Student EXTENDS Person, Intellectual ()
```

 declares an element type with label `Student`, which extends the element types `Person` and `Intellectual` and consequently exposes the three property types `firstName`,

birthday, and cigarettesPerMonth. The element type does not include any additional property types. It exposes the labels Student, Person, and Intellectual.

- Given the element types

Person (firstName VARCHAR(100) NOT NULL, birthday DATE)

Intellectual (cigarettesPerMonth INT)

and nothing else, the element type declaration

Student EXTENDS Person, Intellectual (terms INT)

declares an element type with label Student, which extends the element types Person and Intellectual, and includes the property type terms, and consequently exposes the property types firstName, birthday, cigarettesPerMonth, and terms. It exposes the labels Student, Person, and Intellectual.

- Given the element types

Person (firstName VARCHAR(100) NOT NULL, birthday DATE)

Unemployed EXTENDS Person ()

Intellectual (cigarettesPerMonth INT)

and nothing else, the element type declaration

Student EXTENDS Person, Unemployed, Intellectual (terms INT)

declares an element type with label Student, which extends the element types Person, Unemployed, and Intellectual, and includes the property type terms, and consequently exposes the property types firstName, birthday, cigarettesPerMonth, and terms. It exposes the labels Student, Person, Unemployed, and Intellectual.

8.2 Vertex type

A vertex type references one element type. Every vertex in a property graph conforms to a certain vertex type, i.e. vertex types can have instances in a property graph; hence, they are said to be concrete.

The label of the vertex type is the label of the referenced element type. The label of a vertex type has to be unambiguous among all vertex types. The exposed labels and property types are precisely the exposed labels and property types of the referenced element type. A vertex conforming to a vertex type has all the exposed labels of the vertex types and properties of the all exposed property types of the vertex types.

Note that, a vertex conforming to a vertex type (X) also conforms to element type X and to all element types that X directly or indirectly extends.

8.2.1 Examples

- Given the element type

Person (firstName VARCHAR(100) NOT NULL, birthday DATE)

the vertex type declaration

(Person)

declares a vertex type with label Person, which references element type Person and consequently exposes the two property types firstName and birthday.

- Given the element types
 - Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
 - Intellectual (cigarettesPerMonth INT)
 - Student EXTENDS Person, Intellectual (terms INT)
 the vertex type declaration


```
(Student)
```

 declares a vertex type with label Student, which references element types Student and consequently exposes the four property types firstName, birthday, cigarettesPerMonth, and terms.
- Without any element type declaration or existing element types the following vertex type declaration


```
( )
```

 declares a vertex type without a label, which references the super element type and consequently exposes no property types.

8.2.2 Negative examples:

- Without any element type declaration or existing element types the following vertex type declaration


```
(Person)
```

 is illegal. Generally, every vertex type shall reference one element type.
- Given the element types
 - Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
 - Intellectual (cigarettesPerMonth INT)
 the following vertex type declaration


```
(Person, Intellectual)
```

 is illegal. Syntactically, the token “Person” has to be followed by a token “)”. Generally, every vertex type shall reference no more than one element type.

8.3 Edge type

An edge type references three element types, the *edge element type*, the *tail element type*, and the *head element type*. Every edge in a property graph conforms to a certain edge type, i.e. edge types can have instances in a property graph; hence, they are said to be concrete. The edge element type of the edge type defines the structure of the “content” of conforming edges, analogous to the referenced element type of vertex types. The label of the edge type is the label of the edge element type. The exposed labels and property types are precisely the exposed labels and property types of the edge element type. An edge conforming to an edge type has all the exposed labels of the edge type and properties of the all exposed property types of the edge type.

The tail element type and the head element type of the edge type define the allowed adjacency of conforming edges. In particular, the tail vertex of an edge shall conform to the tail element type of the edge’s edge type, while the head vertex of an edge shall conform to the head element type of the edge’s edge type.

The name of an edge type is a triple of the label of the tail element type, the label of the edge element type, and the label of the head element type. The name of an edge type has to be unambiguous among all edge types.

Note that the requirement for edge types to be unambiguous allows for multiple edge types with the same label between different pairs of tail element type and head element type as well as in different directions (forward and backward).

Note that an edge type in and of itself does not require the existence of vertex types. An edge type only references element types. However, the creation of an edge conforming to an edge type requires at least one vertex type, so that there can be vertices that the edge connects.

Note that there is currently no restriction on the tail element type and the head element type. However, it possible to require a covariance [Cardelli1984] between the edge element type on the one hand and the tail element type and the head element type on the other hand. Covariance⁶ means that the subtyping relations of simple types are preserved for complex types composed from the simple types. In our case that means that the subtyping relation of the edge element types referenced edge types have to line up with the subtyping relations of element types referenced the tail and head element types referenced the edge types.

8.3.1 Examples

- Given the element types

```
Comment
Message
REPLY_OF
```

the edge type declaration

```
(Comment)-[REPLY_OF]->(Message)
```

declares an edge type with label REPLY_OF, which has edge element type REPLY_OF, tail element type Comment, and head element type Message. Consequently, it exposes the label REPLY_OF and no property types. Edges of this edge type can connect vertices conforming to element type Comment to vertices conforming to element type Message.

- Given the element types

```
Comment
Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
WRITES (durationOfWriting INT)
POSTS EXTENDS WRITES (time TIMESTAMP)
```

the edge type declaration

```
(Person)-[POSTS]->(Comment)
```

declares an edge type with label POSTS, has edge element type POSTS, tail element type Person, and head element type Comment. Consequently, it exposes the labels POSTS and WRITE and the property types durationOfWriting and time. Edges of this

⁶ <https://apiumhub.com/tech-blog-barcelona/scala-generics-covariance-contravariance/>

edge type can connect vertices conforming to element type `Person` to vertices conforming to element type `Comment`.

- Given the element types, vertex types, and edge type

```
Student
Teacher
PhDStudent EXTENDS Student, Teacher
TEACHES
(PhDStudent)
(Teacher)-[TEACHES]->(Student)
```

conforming edges of edge type `(Teacher)-[TEACHES]->(Student)` can connect vertices of vertex type `PhDStudent`, since `PhDStudent` vertices also conform to element type `Student` and `Teacher`.

- Given the element types

```
Person (firstName VARCHAR(100) NOT NULL, birthday DATE)
LIKES
```

the edge type declaration

```
(Person)-[LIKES]->( )
```

declares an edge type with label `LIKES`, has edge element type `LIKES`, tail element type `Person`, and the super element type as head element type. Consequently, it exposes the labels `POSTS` and no the property types. Edges of this edge type can connect vertices conforming to element type `Person` to any other vertex, since every vertex conforms to the super element type.

- Given the element types, vertex types, and edge type

```
Weighted (weight DOUBLE)
(Weighted)
(Weighted)-[Weighted]->(Weighted)
```

conforming edges of edge type `(Weighted)-[Weighted]->(Weighted)` have the label `Weighted` and a property `weight` and connect vertices, which also have the label `Weighted` and a property `weight`.

- Given the element types, vertex types, and edge type

```
Weighted (weight DECIMAL)
(Weighted)
(Weighted)-[ ]->(Weighted)
```

declares an edge type without a label, which has the super element type as edge element type, tail element type `Weighted`, and head element type `Weighted`. Consequently, it exposes no property types.

- Without any element type declaration or existing element types the following edge type declaration

```
( )-[ ]->( )
```

declares an edge type without a label, which has the super element type as edge element type, as tail element type, and as head element type. Consequently, it exposes no property types.

8.4 Complete example showing element, vertex and edge type declarations

The following shows all the element, vertex and edge type declarations for a property graph:

```
CREATE PROPERTY GRAPH UniversityMessageBoardTemplate (
  -- Messages
  (Person)-[WRITES]->(Comment),
  (Person)-[POSTS]->(Comment),
  (Comment)-[REPLY_OF]->(Message),
  (Person)-[PUBLISHED]->(Picture),
  (Person)-[PUBLISHED]->(Message),
  (Person)-[SENT]->(Message),
  (Commenting)-[REFERS_TO]->(Posting),

  (Comment),
  (Message),
  (Posting),
  (Commenting),
  (Picture),
  (Post),

  Message (text VARCHAR(200)),

  Picture
    (caption  VARCHAR(200),
     imageFile VARCHAR(200)),

  Posting
    (caption  VARCHAR(200),
     imageFile VARCHAR(200)),

  Commenting (text VARCHAR(200)),

  Post EXTENDS Message (imageFile VARCHAR(200)),
  Comment EXTENDS (Message),

  REPLY_OF,
  REFERS_TO,
  WRITES (durationOfWriting INT),
  PUBLISHED (creationDate  DATE),
  SENT (creationDate DATE),

  POSTS EXTENDS WRITES (time TIMESTAMP),
```

```

-- Persons
Person
  (firstName VARCHAR(100) NOT NULL,
   lastName  VARCHAR(100),
   birthday  DATE),

Intellectual (cigarettesPerMonth INT),

Teacher,

Student EXTENDS Person, Intellectual (terms INT),
PhDStudent EXTENDS (Student, Teacher),

TEACHES,
SAME_VILLAGE (village VARCHAR(100)),

(Person),
(Teacher),
(Student),
(PhDStudent),

(Teacher)-[TEACHES]->(Student),
(Person)-[SAME_VILLAGE]->(Person)
)

```

The order of the individual type declarations contained by the definition is insignificant, which means that an element type can be declared after it is referenced; this is exemplified in the `Message` type declarations. The syntax rules of the definition rewrite the declaration so that the type declarations can be processed top-down. The rewriting rules only demand a partial order. The order beyond the partial order imposed by the rule is implementation dependent.

For example, the definition above could be rewritten to:

```

CREATE PROPERTY GRAPH UniversityMessageBoardTemplate (
  -- element types in stratum 1 ('stratum' is defined in Section 3.3.1.3)
  Person
    (firstName VARCHAR(100) NOT NULL,
     lastName  VARCHAR(100),
     birthday  DATE),

  Teacher,

  Intellectual (cigarettesPerMonth INT),

```

```

Message (text VARCHAR(200)),

Picture
  (caption   VARCHAR(200),
   imageFile VARCHAR(200)),

Posting
  (caption   VARCHAR(200),
   imageFile VARCHAR(200)),

Commenting (text VARCHAR(200)),

REPLY_OF,
TEACHES,
REFERS_TO,
WRITES (durationOfWriting INT),
PUBLISHED (creationDate DATE),
SENT (creationDate DATE),
SAME_VILLAGE (village VARCHAR(100)),

-- element types in stratum 2
Student EXTENDS Person, Intellectual (terms INT),
Post EXTENDS Message (imageFile VARCHAR(200)),
Comment EXTENDS (Message),

POSTS EXTENDS WRITES (time TIMESTAMP),

-- element types in stratum 3
PhDStudent EXTENDS (Student, Teacher),

-- vertex and edge types
(Person),
(Picture),
(Teacher),
(Student),
(PhDStudent),
(Comment),
(Message),
(Posting),
(Commenting),
(Post),

(Teacher)-[TEACHES]->(Student),
(Person)-[SAME_VILLAGE]->(Person),
(Person)-[WRITES]->(Comment),

```

```

(Person)-[POSTS]->(Comment),
(Comment)-[REPLY_OF]->(Message),
(Person)-[PUBLISHED]->(Picture),
(Person)-[PUBLISHED]->(Message),
(Person)-[SENT]->(Message),
(Commenting)-[REFERS_TO]->(Posting)
)

```

The rewriting facility gives the user the freedom to order the type declarations in whichever order the user finds helpful.

8.5 Mapping examples

In this section, we illustrate the population of a PPG using data from existing tables with a series of examples, each of which exemplifies a particular scenario.

We note that - and this applies to all the examples shown in this section - mappings could indeed simply be references to views. However, we show in all cases the version with the base tables with a predicate applied in order to illustrate the convenience and flexibility conferred by going down this route.

8.5.1 Relational schema

The examples that follow all use the same relational tables as the input to the graphs.

The scenario is that people post pictures with a caption, and then people comment on the picture postings.

```
SET SCHEMA MESSAGES_TABLES;
```

```
CREATE TABLE PEOPLE (
    "ID"                BIGINT PRIMARY KEY,
    "firstName"         VARCHAR(100) NOT NULL,
    "lastName"          VARCHAR(100) NOT NULL,
    "village "          VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE POSTS (
    "PostID"           BIGINT PRIMARY KEY,
    "caption"           VARCHAR(200),
    "imageFile"        VARCHAR(100)
);
```

```
CREATE TABLE COMMENTS (
    "CommentID"        BIGINT PRIMARY KEY,
```

```

        "post"          BIGINT NOT NULL REFERENCES POSTS("PostID"),
        "text"         VARCHAR(2000)
    );

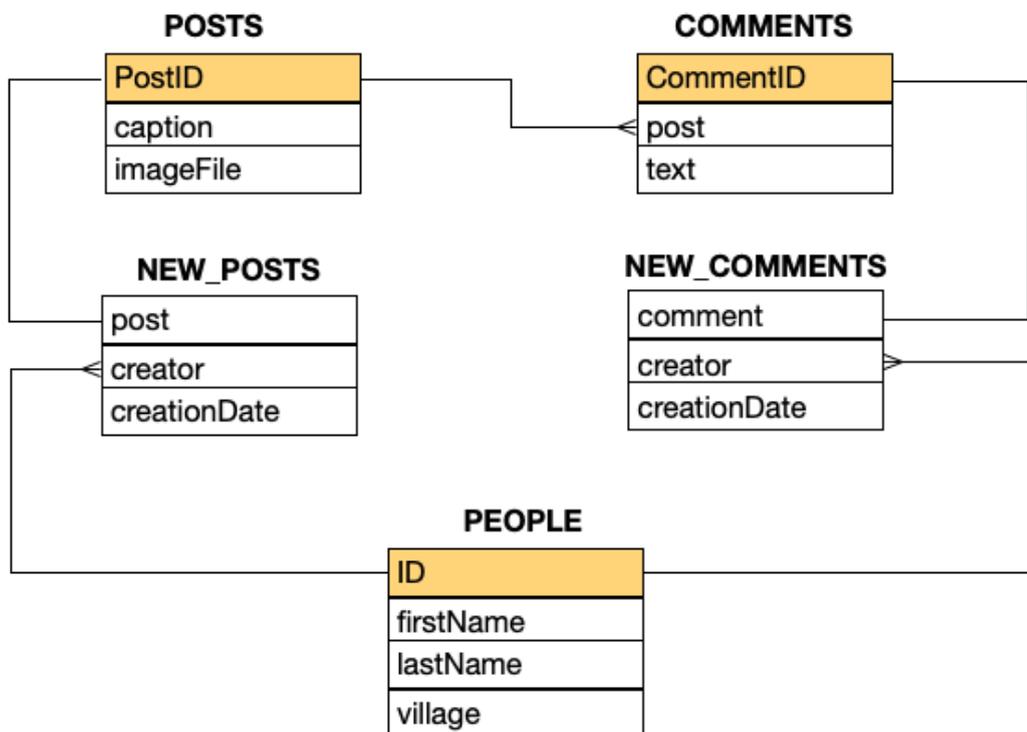
-- link tables

CREATE TABLE NEW_POSTS (
    "post"          BIGINT NOT NULL UNIQUE REFERENCES POSTS("PostID"),
    "creator"      BIGINT NOT NULL REFERENCES PEOPLE("ID"),
    "creationDate"  TIMESTAMP
);

CREATE TABLE NEW_COMMENTS (
    "comment"      BIGINT NOT NULL UNIQUE REFERENCES COMMENTS("CommentID"),
    "creator"      BIGINT NOT NULL REFERENCES PEOPLE("ID"),
    "creationDate"  TIMESTAMP
);

```

The schema is shown in the following diagram, where the primary keys are indicated by an orange background:



8.5.2 Two vertex types, one edge type: using a link table

Persons publish pictures, using the link table NEW_POSTS to define the PUBLISHED edge.

Next, we define the vertex type and edge type views - each of which is composed of, respectively, at least one vertex type or edge type mapping - to obtain the graph over the tables view:

```
CREATE PROPERTY GRAPH published_messages_2018
(
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName   VARCHAR(100)),

  Picture
    (caption    VARCHAR(200),
     imageFile  VARCHAR(200)),

  PUBLISHED (creationDate  DATE),

  (Picture)      -- a 'vertex type view', which has one mapping:
    FROM POSTS, -- a 'vertex type mapping'

  (Person)
    FROM PEOPLE,

  (Person) - [PUBLISHED] -> (Picture) -- an 'edge type view', with one mapping:
    -- What follows is an example of an 'edge type mapping'
    FROM NEW_POSTS pub -- an 'edge mapping'
    -- the next three lines constitute a 'tail mapping'
    (Person) -- this is optional
      FROM PEOPLE ppl -- the choice of correlation name is up to the user
      JOIN ON ppl.ID = pub."creator"
    -- the next three lines constitute a 'head mapping'
    (Picture) -- this is optional
      FROM POSTS pst
      JOIN ON pst."PostID" = pub."post"
)
)
```

We note that (i) the vertex type references in the edge mapping are optional (we highlight this fact in comments), as these can be deduced from the edge type reference; and (ii) the choice of correlation names is completely up to the user.

For example, instead of using `ppl` (as in the mapping example above), we could use `start_vertices` to make it very clear - to future readers - that the vertices induced by the records from `PEOPLE` act as *starting* vertices (i.e. tail vertices) for the purposes of the `PUBLISHED` edges. We show the example snippet below, with changes in **bold**.

```
(Person) - [PUBLISHED] -> (Picture)
...
```

```
FROM PEOPLE start_vertices
  JOIN ON start_vertices.ID = pub."creator"
...

```

8.5.3 Two vertex types, one edge type: edge from one of the vertex tables

The graph structure is the same as [Example 8.5.2](#), but the edge is determined by a column in one of the vertex input tables. In this case, this is truly a foreign key to the primary key of the other vertex input table, but the mappings would be exactly the same if the input tables were views with no declared fk:pk relationship.

```
CREATE PROPERTY GRAPH comments_and_postings2018
(
  Posting
    (caption  VARCHAR(200),
     imageFile VARCHAR(200)),

  Commenting (text  VARCHAR(200)),

  REFERS_TO,
  (Posting)
    FROM POSTS,
  (Commenting)
    FROM COMMENTS,

  (Commenting) - [REFERS_TO] -> (Posting)
    FROM COMMENTS ref
      (Commenting)
        FROM COMMENTS cmt -- this is the same table
          JOIN ON cmt."CommentID" = ref."CommentID"
      (Posting)
        FROM POSTS pst -- using the foreign key (sensu latu)
          JOIN ON pst."PostID" = ref."Post"
)

```

8.5.4 Vertex loaded from more than one input table

Posts and comments are treated as variations of Message, so the Message vertices are loaded from both tables, and the related edges from both link tables.

The column caption in the POSTS table is mapped to the text property in the vertex.

```
CREATE PROPERTY GRAPH person_message_2018
(
  Person

```

```

        (firstName    VARCHAR(100),
         lastName    VARCHAR(100)),

Message    (text          VARCHAR(200)),

PUBLISHED (creationDate DATE),

(Message)
  FROM POSTS
    ("caption" AS text),
  FROM COMMENTS,
(Person)
  FROM PEOPLE,

(Person) - [PUBLISHED] -> (Message)
  FROM NEW_POSTS pub
    (Person)
      FROM PEOPLE ppl
        JOIN ON ppl.ID = pub."creator"
    (Message)
      FROM POSTS pst
        JOIN ON pst."PostID" = pub."post",
  FROM NEW_COMMENTS pub
    (Person)
      FROM PEOPLE ppl
        JOIN ON ppl.ID = pub."creator"
    (Message)
      FROM COMMENTS cmt
        JOIN ON cmt."CommentID" = pub."comment"
)

```

8.5.5 Vertex types with composition

This example uses a similar form to that of [Example 8.5.4](#), but this time illustrates element type composition, with respect to the vertex types Post and Comment. Both of these are composed from the Message element type.

```

CREATE PROPERTY GRAPH person_message_composition_2018
(
  Person
    (firstName    VARCHAR(100),
     lastName    VARCHAR(100)),

  Message    (text          VARCHAR(200)),

  Post EXTENDS Message (imageFile VARCHAR(200)),

```

```

Comment EXTENDS (Message),

SENT      (creationDate DATE),

(Person)
  FROM PEOPLE,
(Post)
  FROM POSTS
    ("caption" AS text),
(Comment)
  FROM COMMENTS,

(Person) - [SENT] -> (Message)
  FROM NEW_POSTS pub
    (Person)
      FROM PEOPLE ppl
        JOIN ON ppl.ID = pub."creator"
    (Post)
      FROM POSTS pst
        JOIN ON pst."PostID" = pub."post",
  FROM NEW_COMMENTS pubcmt
    (Person)
      FROM PEOPLE ppl
        JOIN ON ppl.ID = pubcmt."creator"
    (Comment)
      FROM COMMENTS cmt
        JOIN ON cmt."CommentID" = pubcmt."comment"
)

```

The graph induced by this mapping would potentially (depending on the data present) connect Person vertices to Post vertices and Comment vertices with a SENT edge. This is because both Post and Comment conform to the Message element type.

8.5.6 Edges and vertices using a single table (self-referencing)

In this example, all the vertices and edges are drawn from the same table, PEOPLE. There is only one type of vertex - Person - and the SAME_VILLAGE edge simply connects which two Persons reside in the same village.

```

CREATE PROPERTY GRAPH person_village_2018
(
  Person
    (firstName  VARCHAR(100),
     lastName   VARCHAR(100)),

  SAME_VILLAGE (village  VARCHAR(100)),

```

```

(Person)
  FROM PEOPLE UNIQUE (firstName, lastName),

(Person) - [SAME_VILLAGE] -> (Person)
  FROM PEOPLE edge
    (Person)
      FROM PEOPLE ppl
        JOIN ON ppl.ID = edge.ID
    (Person)
      FROM PEOPLE neighbour
        JOIN ON edge."village" = neighbour."village"
        AND edge.ID <> neighbour.ID -- we don't want any
self-loops
)

```

This example also illustrates a vertex mapping using a UNIQUE assertion, stating that the combination of the values in the `firstName` and `lastName` columns must be unique within the `PEOPLE` table (in essence, a candidate key).

8.5.7 Edges from multiple input tables

This example illustrates the case of where the same edge, `SENT`, is populated by three different tables. This scenario will in all likelihood prove to be quite common in the business domain, where tables with similar entities - such as people - are stored in different databases across different divisions of the same corporation.

```

CREATE PROPERTY GRAPH person_posts_messages_2018
(
  -- snipped for brevity
  ...

(Person) - [SENT] -> (Message)
  FROM IB.POSTS relationship
    (Person)
      FROM PEOPLE start_vertex
        JOIN ON start_vertex.ID = relationship."creator",
    (PostMessage)
      FROM IB.POSTS end_vertex
        JOIN ON end_vertex."PostID" = relationship."post"
  FROM WEALTH.POSTS relationship
    (Person)
      FROM WH.PERSON wh_start_vertex
        JOIN ON wh_start_vertex.ID = relationship."creator",
    (PostMessage)
      FROM WH.POSTS end_vertex

```

```

        JOIN ON end_vertex."PostID" = relationship."post"
FROM RETAIL.POSTS relationship
  (Person)
  FROM RETAIL.PEOPLE dl_start_vertex
    JOIN ON dl_start_vertex.ID = relationship."creator"
  (PostMessage)
  FROM DL.POSTS end_vertex
    JOIN ON end_vertex."PostID" = relationship."post"
)

```

8.5.8 Example showing how one graph can be created reusing types from another graph

It is self-evident to see that it is possible to create a property graph that contains no vertex type views and no edge type views; i.e. the graph contains only element, vertex and edge type declarations. In effect, such a graph will contain no data, and can thus be seen as a 'prototype' graph, or a template for actual, instantiated graphs where it is desirable to reuse the type declarations, but to define separate mappings for each concrete graph.

This example is analogous to [Example 8.5.2](#) but shows how a graph may reuse the types from an existing (so-called 'template') graph.

We reproduce here the relevant snippet of `UniversityMessageBoardTemplate` (which was given in full in [Section 8.4](#)):

```

CREATE PROPERTY GRAPH UniversityMessageBoardTemplate
(
  ...
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName   VARCHAR(100)),

  Picture
    (caption    VARCHAR(200),
     imageFile  VARCHAR(200)),

  PUBLISHED (creationDate  DATE),

  (Person),
  (Picture),

  (Person) - [PUBLISHED] -> (Picture)
  ...
)

```

Next, we define the vertex type and edge type views in the context of creating another graph, `published_messages_2018`, which reuses the same type information as `UniversityMessageBoardTemplate`:

```
CREATE PROPERTY GRAPH published_messages_2018 LIKE UniversityMessageBoardTemplate
(
  (Picture)
    FROM POSTS,
  (Person)
    FROM PEOPLE,

  (Person) - [PUBLISHED] -> (Picture)
    FROM NEW_POSTS pub
      (Person) -- a reminder that this is optional
        FROM PEOPLE ppl
          JOIN ON ppl.ID = pub."creator"
      (Picture) -- a reminder that this is optional
        FROM POSTS pst
          JOIN ON pst."PostID" = pub."post"
)
```

8.5.9 Using keys

In this section we show by means of an incrementally-constructed example the use and utility of keys. Newly-added declarations appears in **black font**, and declarations shown in a previous example - and thus already described - appear in **gray font**.

8.5.9.1 Element type with a key

We declare a `Person` element type with an associated key, `person_key`, which mandates that the combination of the `country` and `idNumber` properties must be unique across all instances of `Person` element types.

```
CREATE PROPERTY GRAPH students_2018
(
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName   VARCHAR(100) NOT NULL,
     country    VARCHAR(100),
     idNumber   VARCHAR(100) NOT NULL),
  KEY person_key (country, idNumber)
)
```

8.5.9.2 Element type using inheritance with a key

The element type `Student` inherits `Person`, which means that a `Student` not only has the properties `studentNumber` etc, but also all the properties defined for `Person`, such as `firstName`.

As an element type also inherits the *keys* from its supertypes, this means that not only must the combination of `country` and `idNumber` be unique across all instances of `Student` element types (just as for `Person`), but also - thanks to `student_key` - that the combination of `studentNumber` and `institutionCode` must be unique across all `Student` instances.

```
CREATE PROPERTY GRAPH students_2018
(
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName  VARCHAR(100) NOT NULL,
     country   VARCHAR(100),
     idNumber  VARCHAR(100) NOT NULL),
  KEY person_key (country, idNumber),

  Student EXTENDS Person
    (studentNumber  VARCHAR(100) NOT NULL,
     yearOfStudy   INT,
     institutionCode INT NOT NULL),
  KEY student_key (studentNumber, institutionCode),

  Teacher EXTENDS (Person),

  TEACHES
    (courseNumber VARCHAR(50) NOT NULL,
     supervisoryRoleType VARCHAR(50))
)
```

8.5.9.3 Vertex type with a key

A vertex type inherits all keys - along with properties - of its included element type. This means that the vertex type (`Student`) inherits `student_key` from the `Student` element type (and also `person_key` from the `Person` supertype). Therefore, all `Student` vertices will be subject to exactly the same rules regarding uniqueness as the `Student` element type (as given in [Example 8.5.9.2](#)).

```
CREATE PROPERTY GRAPH students_2018
(
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName  VARCHAR(100) NOT NULL,
```

```

        country      VARCHAR(100),
        idNumber     VARCHAR(100) NOT NULL),
KEY person_key (country, idNumber),

Student EXTENDS Person
  (studentNumber  VARCHAR(100) NOT NULL,
   yearOfStudy   INT,
   institutionCode INT NOT NULL),
KEY student_key (studentNumber, institutionCode),

Teacher EXTENDS (Person),

TEACHES
  (courseNumber  VARCHAR(50) NOT NULL,
   supervisoryRoleType VARCHAR(50)),

(Student),
(Teacher)
)

```

8.5.9.4 Edge type with a key

In this example, the key `one_teacher_per_course_per_student` is defined for the edge type given by `(Teacher)-[TEACHES]->(Student)`.

Here, the key properties are drawn from:

1. the property `idNumber` from the head vertex type `(Teacher)` which is qualified by 'TAIL',
2. the property `courseNumber` (defined as a property on the `TEACHES` edge), and
3. the property `studentNumber` from the tail vertex type `(Student)`, which is qualified by 'HEAD'.

```

CREATE PROPERTY GRAPH students_2018
(
  Person
    (firstName  VARCHAR(100) NOT NULL,
     lastName   VARCHAR(100) NOT NULL,
     country    VARCHAR(100),
     idNumber   VARCHAR(100) NOT NULL),
  KEY person_key (country, idNumber),

  Student EXTENDS Person
    (studentNumber  VARCHAR(100) NOT NULL,
     yearOfStudy   INT,
     institutionCode INT NOT NULL),
  KEY student_key (studentNumber, institutionCode),
)

```

Teacher EXTENDS (Person),

TEACHES

(courseNumber VARCHAR(50) NOT NULL,
supervisoryRoleType VARCHAR(50)),

(Student),

(Teacher),

(Teacher)-[TEACHES]->(Student),

KEY one_teacher_per_course_per_student

(TAIL idNumber, courseNumber, HEAD studentNumber)

)

9 Checklist

Interactions with other concurrent proposals identified and editorial assistance given	no known interaction with other concurrent proposals.
Concepts	Yes
Access Rules	No
Conformance Rules	No
Lists of SQL-statements by category	No
Table of identifiers used by diagnostics statements	No
Collation coercibility for character strings	No
Closing Possible Problems	No
Any new Possible Problems clearly identified	Yes
Reserved and non-reserved keywords	No
SQLSTATE tables and Ada package	No
Information and Definition Schemas	No, noted required in PP
Implementation-defined and -dependent Annexes	No
Incompatibilities Annex	No
Embedded SQL and host language implications	No
Dynamic SQL issues: including descriptor areas	No
CLI impact	No
PSM impact	No
MED impact	No
OLB impact	No
Schemata impact	Yes
JRT impact	No
SQL/XML impact	No

SQL/PGQ impact	Yes
Any TR impact	Yes, in future

10 Revision details

10.1 Changes in sql-pg-2018-0056r1 (WG3:BNE-022)

Grouped all prefatory material in first section, reserving Rationale for technical description.

Recast property graph type as a conceptual schema object included in property graph, rather than as a first-class schema object in its own right, therefore

Revised Rationale text, diagrams and examples; revised all example in part 8

Removed <property graph type definition> and modified referencing text. Moved <property graph definition> to be first sub-clause of Rules.

Removed subclause numbering from Format, Syntax rules etc.

Removed use of “deemed”

Removed most uses of “shall” in General Rules (issue remains on how to ensure consistency to types)

Changed text that invokes “subroutine subclauses” to standard style

11 An ITI, ISO and openCypher/GQL contribution from Neo4j Inc.

This contribution is a Deliverable under the terms of clause 2.2.1 of the Agreement for Membership in the InterNational Committee for Information Technology Standards (“INCITS”), a Division of the Information Technology Industry Council (“ITI”) to which Neo4j Inc. is a party.

It is also a contribution to the [openCypher community](https://www.opencypher.org/)⁷ and like all such contributions is:

Copyright © 2018 Neo4j Inc.

**Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.**

You may obtain a copy of the License at <https://www.opencypher.org/>

⁷ <https://www.opencypher.org/>

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Apache License, Version 2.0, Attribution Notice

This document is a contribution by Neo4j's Query Languages Standards and Research Team to the openCypher project (opencypher.org) and to the SQL standard development process.

It is also a contribution to the GQL standard project incubation community (gqlstandard.org).

