OAEP-2023-01 **DOI: 10.54285/ldbc.QZSK3559**

Published on LDBC's website April 2023

Originally published privately for members of ISO/IEC JTC 1/SC32 WG3,
May to October 2018

SQL/PGQ data model and graph schema

Neo4j 2018 Contributions to WG3 (YTZ, ERF)
on SQL/PGQ data model and graph schema

These papers were originally submitted to WG3 for discussion at its YTZ (Toronto, Canada) and ERF (Ilemanau, Germany) meetings in May and September/October 2019[1]. This document contains three original papers, reproduced without modification.

1	ISO/IEC JTC1/SC32 WG3 **YTZ-034**	"Property Graph Data Model for SQL"	Neo4j SQL working group	r2 2 May 2018
2	ISO/IEC JTC1/SC32 WG3 **ERF-043**	"SQL/PG graph schema and join syntax mapping examples"	Peter Furniss, Individual Expert Contribution (member of Neo4j SQL working group)	27 September 2018
3	ISO/IEC JTC1/SC32 WG3 **ERF-044r1**	"Property Graph Data Model Concepts and Terms"	Alastair Green, U.S.A. National Expert (member of Neo4j SQL working group)	r1 30 October 2018

[1] Revisions to WG3 papers may be dated later than the meeting concerned

LDBC Open Access to External Papers

In this series, Linked Data Benchmark Council makes papers published originally for a restricted audience available for open access.

These papers are of interest to our members and the public, and are concerned with topics that relate to the work of LDBC. They are published with the permission of their copyright holders, which may have been given by a licence grant.

This collection of three papers is relevant for the work of the LEX (LDBC Extended GQL Schema) Working Group.

These papers have the character of technical reports: they have not been submitted to or accepted via peer review by an established scholarly publication.

Permission to publish is given by the grant of an ASL 2.0 licence for each of these papers by the copyright holder Neo4j Inc. This grant is included in the text of each paper as reproduced here.

The attribution statements for each paper is included in its text, and the reader's attention is drawn to those statements.

Property Graph Data Model for SQL

Title	Property Graph Data Model for SQL
Authors	Neo4j SQL working group[1]
Status	Outline partial draft of initial working document on SQL PGQ
Date	24 April 2018
Date of r1	24 April 2018 Sub-editorial changes
Date of r2	2 May 2018 Correct over-restriction on sharing of property names in Label Sets, strengthens References and comparison of PGDM model variants.

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

Contents

[1] Current members of the Neo4j SQL working group are: Tobias Lindaaker, Stefan Plantikow, Petra Selmer, Peter Furniss, Alastair Green.

1. Summary

We present a revised version of the section **"SQL Property Graph Data Model"** in [sql-pg-2017-0029].

The meta-metamodel of such a Graph Schema and of a Graph Object, as new entities in the SQL Information Schema, is shown in UML.

This contribution forms the basis for a second (forthcoming) paper on mapping existing tabular data in a SQL store into a Graph Object.

Readers may also find the graph metamodel representation proposed in [openCypher-CIR-2018-307] interesting and useful in understanding the model presented in this paper.

That "ASCII Art" representation is another view of the kind of **Graph Schema** or **Property Graph Data Model** defined here**.** It reflects a separate proposal documented in [openCypher-CIR-2018-311], which allows use of Cypher patterns in SQL without the need for syntax "quoting".)

The ASCII Art schema representation allows the definition of Label Sets, which is an important step in forming a Graph Schema.

2. References

[Foundation:2016]	Jim Melton (ed), “ISO International Standard (IS) Database Language SQL- Part 2: SQL/Foundation”, ISO/IEC 9075-2:2016
[sql-pg-2017-0029]	Peter Furniss, Alastair Green, “SQL Property Graph Representations”, July 2017
[openCypher-CIR-2018-307]	Neo4j Cypher Language Group, “‘ASCII Art’ graph schema”, March 2018
[sql-pg-2017-0036]	Alastair Green, “Remarks on [sql-pg-2017-0032r1] ‘Comments on sql-pg-2017-0029’”, July 2017
[sql-pg-2017-0047r1]	Peter Furniss, Alastair Green, Petra Selmer, “SQL Graph Query Procedures” (with corrigenda), August/October 2017
[openCypher-CIR-2018-311]	Neo4j SQL Working Group, “Syntax modifications to Cypher path patterns to permit use in SQL”, April 2018
[Cypher 9]	Neo4j Cypher Language Group, “Cypher Query Language Reference, Version 9”, 2018
[PGQL 1.1]	Oracle, “PGQL 1.1 Specification”, 2018
[GCORE]	Angles et al., “G-CORE: A Core for Future Graph Query Languages”, pre-print of paper accepted for SIGMOD 2018

3. Purpose of a Property Graph Object in SQL

The purpose of a property graph object is to provide an operand or a result of a property graph function. Such a function, however surfaced in SQL syntax, is an operation that accepts as input one or more graphs.

In [sql-pg-2017-0047r1] we proposed one such function, GRAPH_TABLE, that can be used to illustrate this point.

Drafting Note

We also proposed that graph objects be exposed as cell values with a type called GRAPH_REFERENCE.

We have subsequently come to the view that the simplest way to create a reference to a graph is to create a named Information Schema object, analogous to a table. (We referred to this possibility in Appendix B, p30 of the cited document, where we also suggested the syntactic impact that might have on GRAPH_TABLE.)

In the interests of making progress on initial property graph extensions to SQL we are focussing on the option of named graph objects, and that is reflected in this paper.

4. The SQL Property Graph Data Model

Simplifying, constraining and making fully logical the data model defined in the section **“SQL Property Graph Data Model”** in [sql-pg-2017-0029], we define a **Property Graph Data Model** for SQL as follows.

1) A **Property Graph** is a directed multigraph[2], whose vertices or nodes form a set of **Nodes**, and whose edges or arcs or relationships form a set of **Edges**. The Nodes and Edges of a graph are, taken together, the **Entities** of the graph.
2) Each Edge has one **Start Node** and one **End Node**.
3) The Direction of an Edge is from Start Node to End Node: synonymously, the Start Node is the tail and the End Node is the head.
4) The term **Graph** is used hereafter as a synonym for Property Graph. A Graph is more than a directed multigraph in the following respects.
5) A Graph has a **Name,** which is an identifier that follows the rules for naming user-defined Tables. A Graph Name can be qualified by the database and schema in which the Graph is defined, giving a triple-element name like that of a Table.
6) Every Entity has one to many **Labels** (which are string identifiers). An implementation must define the maximum number of Labels allowed for Nodes and for Edges, which in each case may be a finite positive integer, or infinity.
7) Each Label can have zero to many **Properties** associated with it.
8) An Entity cannot have two Labels each of which has a Property of the same name, unless each of those properties has the same data-type[3].
9) A **Property** is a named typed value, whose type is any valid SQL type, and which may be optional or mandatory for any given Node or Edge.
10) A **Non-Null Property** is an optional property which has a null value.
11) A **Property Name** must be unique within the set of Property Names associated with a Label.
12) A Label can have one or more **Entity Keys**. An **Entity Key** is a set of mandatory Properties whose values, taken together, form a set, an **Entity Key Value**.
13) If a Node has a Label, and that Label has an Entity Key, then that Entity Key acts as a **Node Key**. The Entity Key Value of a Node Key for a given Node cannot equal the

[2] A *multigraph* is one where there can be multiple edges between two nodes and where an edge can connect a node to itself (a *loop*). Every edge in a *directed graph* has a tail and a head, and the direction flows from the tail to the head.

[3] For this to be useful, the properties should have the same application semantic.

Entity Key Value of any other Node with the same Label[4]. An implementation may choose not to support Node Keys.

14) If an Edge has a Label, and that Label has an Entity Key, then that Entity Key acts as an **Edge Key**. The Entity Key Value of an Edge Key for a given Edge cannot equal the Entity Key Value of any other Edge with the same Label. An implementation may choose not to supports Edge Keys.

15) If an Entity has more than one Label then all of its Labels are called a **Label Set**. The union of all of the Properties of all the members of a Label Set is a set called a **Label Set Properties**. It follows that each Property of such an Entity has a name which is unique. To achieve this, if two or more Labels in the Labet Set have a property of the same name, then (if the data-type of each such property is identical) the Label Set Properties will have only one property with that name[5].

16) The set of Non-Null Properties which is a subset of the Label Set Properties of an Entity (of a Node, or of an Edge) is a set called the **Non-Null Properties**. The set of values of all the members of the Properties Present is the **Non-Null Properties Values**.

17) For any two Nodes which have the same Label, the corresponding Non-Null Properties may be equal or different. If equal, then the corresponding Non-Null Properties Values for the same two Nodes may be equal or different. The set of Nodes in a Graph having a given Label therefore maps to a multiset of **Node Non-Null Properties** and to a multiset of **Node Non-Null Properties Values**.

18) For any two Edges which have the same Label, the corresponding Non-Null Properties may be equal or different. If equal, then the corresponding Non-Null Properties Values for the same two Edges may be equal or different. The set of Edges in a Graph having a given Label therefore maps to a multiset of **Edge Non-Null Properties** and to a multiset of **Edge Non-Null Properties Values**.

[4] Node and Edge Keys are candidate or unique keys. The concepts of primary key and foreign key to do not exist in the property graph data model, where relationships are expressed explicitly by edges. They can be relevant in assisting *mappings* between the relational and property graph data models.

[5] The coalescence of properties that have the same meaning in more than one Label is equivalent to the effect of creating a “mix-in” interface in Java, to take an analogy.

5. Remarks on the Property Graph Data Model

5.1. Graph Entities without Properties

Graphs, which can express structure without data values, may have no properties on a particular kind of Edge or Node. This is common and unremarked for Edges, and less common for Nodes, but Nodes which are linked to form purely structural models are found in the wild.

A Table in SQL must have at least one column. This extends from base tables to views (given the same restriction on the results of SELECT).

Labels, for this reason are logically distinct from Tables, and cannot always be represented by a Table, unless the model (artificially) forces at least one property to be added to every Label.

A View in SQL cannot have a primary key or uniqueness constraint. Therefore, not all Tables can have unique keys. Labels should have the ability to have candidate keys (unique keys). This is a second reason for distinguishing Labels from Tables.

Therefore, it is, in our view, preferable to express a Graph Object as being conformant with a Graph Schema, which implements a metamodel consistent with the overall Data Model (schema metamodel) described here, and to express optional mappings from Tables to Labels to give a Graph view over existing data, than to force the use of Tables as the building blocks of a Graph Schema, not least because this would preclude the creation of Graph Objects that do not derive from nor are directly mapped to Table data.

5.2. Explicit Entity Identifiers Removed

Node identifiers and Edge Identifiers (Entity Identifiers) have been removed in this revised version of the data model. Their presence in **[sql-pg-2017-0029]** was a mistake, influenced by implementation tactics in existing SQL property graph implementations like SQLServer.

[Cypher 9] also contributes to this erroneous thinking by making a function available which gives a unique edge identifier or node identifier for an entity accessible via a binding variable, and by defining the identity of two node references in terms of identifier equality.

[PGQL 1.1] does not define identifiers as an inherent part of the data model. Conversely, **[GCORE]** makes identifiers central to its data model.

However, at a logical level, each Node and each Edge is a member of a set (like each row in a table is a member of a set of rows), and is therefore unique, whatever the number or value of its properties. Two entities with identical Label Sets and identical values of the Label Set Properties are still two distinct entities, just as two rows in a Table are distinct tuples.

At the same time the same two entities, viewed in terms of their Label Sets and values of their Label Set Properties, form a multiset. SQL uses this angle of view when it describes the data content of a Table as a “multiset of rows” **[Foundation:2016**].

It is not necessary for the specification of graph query semantics, or for the definition of Labels, or for the definition of the data model in terms of structural relationships (edges between nodes), to use the concept of Entity Identifier. It is only necessary to define a function, that may or may not be available in the syntax of any graph manipulation sub-language, that allows for the comparison of two Entity references and evaluates to either “identical: the same Entity”, or “not identical: different Entities”.

Row identifiers are not part of the core SQL tabular data model, for the same reasons.

5.3. Closed-world Schema

Note that the data model defined above

a) prevents data being held in a property graph in Properties whose type is not defined by the Information Schema

b) prevents data in a Property for a given Label from having more than one type.

These constraints corresponds to the fact that SQL Information Schema is a closed-world model, which extends (modulo Polymorphic Table Functions) to column-typing. This allows schema-based type inference during the course of projection or expression evaluation.

Unlike Cypher, SQL property graphs will have mandatory schema (will be “schema strict”), and will not permit heterogeneous property typing.

Cypher’s data model is “schema optional” and does allow types of a given property name to vary. The SQL Property Graph Data Model defined here is therefore a proper subset of Cypher’s[6]. In particular a Cypher read or data update query can operate unchanged over a graph conforming to this SQL Property Graph Data Model.

5.4. Label Sets and Inheritance

If the relationship between a type (class) of Start Node, a type of Edge and a type of End Nodes is defined in terms of Label Sets (i.e., where the Label Set is the type), then it is possible to say things like: “All Persons are either Residents or Visitors, for the purpose of a census; Any Person may be present in a Town, on the night of the census.”

[6] There is a proviso: openCypher has not yet formalized the concepts of Node Keys and Relationship Keys. Node Keys are present in Neo4j Cypher. Neo4j favours introducing both kinds of Entity Key in Cypher, but that depends on a more general expansion of schema features in Cypher.

It therefore follows that the relationship or edge **`[PRESENT_IN]`** can connect nodes of type **`(Person)`** and **`(Town)`**, and that it is not possible to be a **`(Person)`** without being a **`(Resident | Visitor)`**. This allows the semantic effect of a supertype, like **`(Person)`**, with respect to to a subtype, like **`(Resident)`** or **`(Visitor)`**, to be expressed without explicitly describing it as an inheritance relationship.

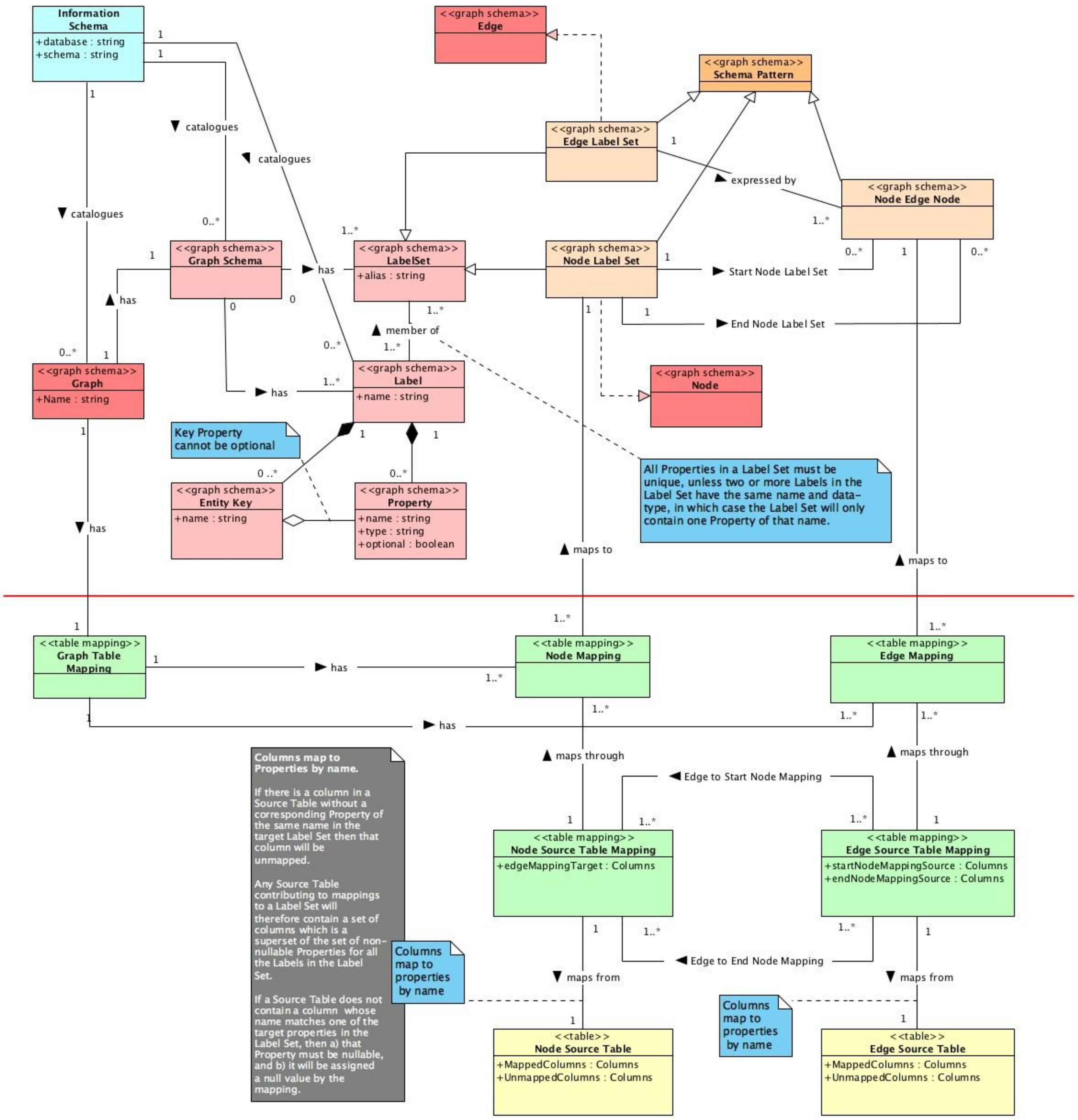
An “accidental”, coincidental combination of labels, where some nodes of type **`(Audited)`** are also nodes of type **`(Person)`**, but not all nodes of type **`(Person)`** are **`(Audited)`**, can also be expressed in this model.

This is achieved by defining Label Sets and defining “Node Edge Node” (NEN) relationships in terms of those Label Sets.

[openCypher-CIR-2018-307] is a means of stating those definitions and NEN relationships (*mutatis mutandis* for the likely differences in Cypher and SQL DDL syntax).

The ability to state these Graph Schema constraints without regard to mappings from Tables to Label Sets is useful in factoring different concerns in the design and programming phases of creating Graph Objects.

6. UML Graph Schema Metamodel (Property Graph Data Model)


Information Schema
+database : string
+schema : string
<<graph schema>> Edge
<<graph schema>> Schema Pattern
<<graph schema>> Edge Label Set
<<graph schema>> Node Edge Node
expressed by
catalogues
catalogues
catalogues
<<graph schema>> Graph Schema
<<graph schema>> LabelSet
+alias : string
<<graph schema>> Node Label Set
Start Node Label Set
End Node Label Set
has
has
member of
has
<<graph schema>> Graph
+Name : string
<<graph schema>> Label
+name : string
<<graph schema>> Node
<<graph schema>> Entity Key
+name : string
<<graph schema>> Property
+name : string
+type : string
+optional : boolean
has
maps to
maps to
<<table mapping>> Graph Table Mapping
<<table mapping>> Node Mapping
<<table mapping>> Edge Mapping
has
has
maps through
maps through
Edge to Start Node Mapping
<<table mapping>> Node Source Table Mapping
+edgeMappingTarget : Columns
<<table mapping>> Edge Source Table Mapping
+startNodeMappingSource : Columns
+endNodeMappingSource : Columns
Edge to End Node Mapping
maps from
maps from
<<table>> Node Source Table
+MappedColumns : Columns
+UnmappedColumns : Columns
<<table>> Edge Source Table
+MappedColumns : Columns
+UnmappedColumns : Columns
1
1
1
1
1..*
1..*
0..*
1..*
0..*
1
0..*
1
0
0
1
1
1..*
0..*
1..*
0..*
1
1..*
1
1
0 ..*
0..*
1
1..*
1
1..*
1..*
1..*
1..*
1..*
1
1..*
1..*
1
1
1..*
1..*
1
1
1

Key Property cannot be optional

All Properties in a Label Set must be unique, unless two or more Labels in the Label Set have the same name and data-type, in which case the Label Set will only contain one Property of that name.

Columns map to Properties by name.

If there is a column in a Source Table without a corresponding Property of the same name in the target Label Set then that column will be unmapped.

Any Source Table contributing to mappings to a Label Set will therefore contain a set of columns which is a superset of the set of non-nullable Properties for all the Labels in the Label Set.

If a Source Table does not contain a column whose name matches one of the target properties in the Label Set, then a) that Property must be nullable, and b) it will be assigned a null value by the mapping.

Columns map to properties by name

Columns map to properties by name

7. An ITI and openCypher 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[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/
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 SQL working group to the openCypher project and to the SQL standards formation process.

[7] https://www.opencypher.org/

SQL/PG graph schema and join syntax mapping examples

Author: Peter Furniss, Neo4j
Source: Individual Expert Contribution
Status: Discussion paper
Date: 27 September 2018
Revision R2: (since first sql-pg-2018-0036)
Expanded introduction with diagram showing relationship between concepts, added text on label set aliasing, examples showing M:M mapping cases

This paper shows some examples of Graph SQL DDL for defining the Graph Schema and mapping a graph from relational data, as currently implemented in Neo4j's proto-product Morpheus (which uses Cypher of Apache Spark). The mapping uses the "join syntax" which Neo4j now prefer. It is an evolution of proposals previously submitted to the SQL/PG Ad Hoc.

1 References

[**YTZ-033**] Jan Michels, "The Pure Property Graph Data Model", ISO/IEC JTC1/SC32 WG3:YTZ-033 = ANSI INCITS DM32.2-2018-00091 / sql-pg-2018-0002

[LDBC SNB] "LDBC Social Network Benchmark (SNB) - 0.3.1", http://ldbc.github.io/ldbc_snb_docs/ldbc-snb-specification.pdf

[YTZ-034] Neo4j SQL working group, "Property Graph Data Model for SQL", ISO/ IEC JTC1/SC32 WG3:YTZ-034 = ANSI INCITS DM32.2-2018-00092 / sql-pg-2018-0003r2

[ERF-042] Jan Michels, "The Pure Property Graph Data Model", ANSI INCITS sql-pg-2018-0035, ISO/IEC JTC1/SC32 WG3-ERF-042

2 Introduction

In parallel with discussions on the property graph data model and its possible representation, Neo4j has developed an implementation of proposed syntax for "pure property graph" definition, in the context of Cypher for Apache Spark, where Cypher co-exists with Spark SQL. This graph metadata or graph schema capability is relevant for SQL PGQ and for a future standalone GQL.

The PPG definition can, in addition, be used as the target for loading tabular data into graph objects (which are in effect graph views over SQL tables). The mapping from SQL tables is expressed in extended SQL DDL based on proposals previously introduced by the Neo4j SQL working group in papers submitted to the SQL/PG ad hoc.

This paper shows some examples of the latest designs for pure property graph metadata definition (based on labels with properties, and how those labels are used in label sets to define the permitted semantic structure of a graph).

It also shows our proposal for a join-inspired DDL syntax to express table-to-graph mappings.

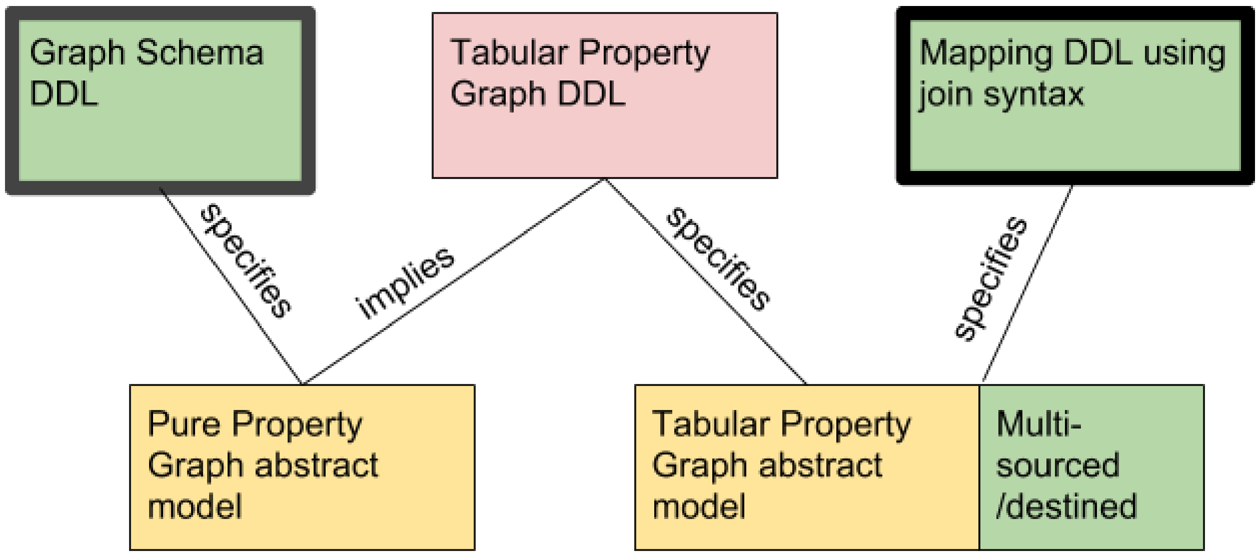
Some of the guiding principles in the design have been:

a) If data manipulations are needed to put the data into the right "shape" for loading the graph, these manipulations should be done by creating SQL views over the existing database structure (tables and views).

b) The structure of the property graph (the Pure Property Graph, in the sense of [ERF-042]) is declared prior to and independently of the specification of which SQL tables (/views) provide the data (the mapping).

These are to some extent related. For example, since the properties of elements (nodes, edges) with a particular label set are specified by the label definitions, there is no need to explicitly subset the columns of a mapped table provided the names match.

The following diagram attempts to summarise the relationship between the concepts described in [ERF-042], [YTZ-033] and this paper. The Tabular Property Graph DDL is assumed to be an updated version of the DDL shown in [YTZ-033]


Graph Schema DDL
Tabular Property Graph DDL
Mapping DDL using join syntax
specifies
implies
specifies
specifies
Pure Property Graph abstract model
Tabular Property Graph abstract model
Multi-sourced /destined

This document shows examples of the two boxes with heavy borders. The graph schema DDL defines the labels and their properties, the permitted label sets for nodes and for edges and the permitted “edge triplets”. The mapping DDL specifies which tables are drawn on to provide the data of the graph.

The examples below do not show any use of primary or foreign key in the underlying tables. This is partly because the relational datasource most commonly used with Morpheus is Hive, which does not support these. In addition, if data is being loaded from views rather than base tables, primary and foreign key will not (normally) be available.

Since Cypher is case-sensitive, the implementation will preserve case for any “graph object” name - graph names, label names, property names even if they are not quoted. Similarly, when it is matching column names to property names, it will match a mixed-case property name to an all-upper column name, if there is no case-sensitive match. It will also handle quoted names in a SQL standard fashion.

3 Label declarations

The implementation currently has two alternative styles for declaring labels and their properties - a SQL-like style and a Cypher-like style. The former is modelled on SQL table declarations, the latter on Cypher patterns. Both have the same capabilities, and can even be used side-by-side.

In the examples, the Cypher-style declarations use the Cypher type system for the property types, the SQL-style the SQL type system. In fact, either can be used in either place - an appropriate conversion will be performed if necessary at the point of loading.
The two styles differ in how an optional (nullable) property is shown - with opposite defaults.

3.1 SQL-style label declaration

```
LABEL "Message"
    PROPERTIES
        ("creationDate"        TIMESTAMP NOT NULL,
         "browserUsed"         VARCHAR(100) NOT NULL,
         "locationIP"          VARCHAR(100) NOT NULL,
         "content"             VARCHAR(2000),
         "length"              INTEGER NOT NULL)
```

3.2 Cypher-style label declaration

```
    LABEL ( Message
        { creationDate         : TIMESTAMP,
          browserUsed         : STRING,
          locationIP          : STRING,
          content             : STRING?,
          length              : INTEGER}
      )
```

4 LDBC Social Network example (SNB)

This is the same example as used in [YTZ-033] 3.4, to the extent of recreating the input tables as used there, but loading data from one of the outputs of the data generation tool (“DATAGEN”) described in [LDBC SNB]. following the same assumptions about the input data made there. Since DATAGEN creates single files “place” and for “organisation”, views were created to separate the subtypes. As in [YTZ-033], all but one of the relationships (edges) are derived from link tables. The exception is the (Forum)-[CONTAINER_OF]->(Post) relationship, where the table of posts has a column that references the forums table.

Note that the graph schema - labels, properties, label sets and triplets, with their cardinality - is a very close transcription of the data schema diagram (Figure 2.1) in [LDBC SNB].

4.1 Graph Schema DDL

The graph schema is declared within the graph specification, and uses Cypher-style label declarations.

The schema contains one or more lists of label declarations, node label set and edge label set declarations and edge triplet declarations. Label declarations are preceded by the keyword “LABEL”, the others can be distinguished by their syntax.

```
CREATE GRAPH snb
     -- graph schema definition
     -- (DDL describing a pure property graph)
     -- (within the graph declaration in this case

  WITH GRAPH SCHEMA (
     -- labels used for nodes
    LABEL ( Person
        { creationDate         : TIMESTAMP,
          firstName            : STRING,
          lastName             : STRING,
          gender               : STRING,
          birthday             : DATE,
           -- these next two should be arrays, according to the
           -- SNB model. This is supported but not shown here.
          email                : STRING,
          speaks               : STRING,
          browserUsed          : STRING,
          locationIP           : STRING}
      ),
      -- organisation has two sub-types, but the properties
      -- are the same for both
    LABEL ( Organisation
        { name                 : STRING,
          url                  : STRING}
      ),
    LABEL ( Company ),
    LABEL ( University ),
      -- Message has two subtypes, Post has extra (optional)
properties
    LABEL ( Message
        { creationDate         : TIMESTAMP,
          browserUsed          : STRING,
          locationIP           : STRING,
          content              : STRING?,
          length               : INTEGER}
      ),
    LABEL ( Comment ),
    LABEL ( Post
        { language             : STRING?,
          imageFile            : STRING?}
      ),
    LABEL ( Forum
        { title                : STRING,
          creationDate         : TIMESTAMP}
      ),
    LABEL ( TagClass
        { name                 : STRING,
          url                  : STRING}
      ),
    LABEL ( Place
        { name                 : STRING,
          url                  : STRING}
      ),
    LABEL ( City ),
    LABEL ( Continent ),
    LABEL ( Country ),
    LABEL ( Tag
        { name                  : STRING,
          url                   : STRING}
      ),
      -- labels used for edges (following cypher convention of
      -- upper-case for edge labels, but in principle a label can
      -- be used for either or both
    LABEL ( HAS_TYPE ),
    LABEL ( HAS_TAG ),
    LABEL ( IS_SUBCLASS_OF ),
    LABEL ( HAS_MODERATOR ),
    LABEL ( HAS_CREATOR ),
    LABEL ( REPLY_OF ),
    LABEL ( HAS_INTEREST ),
    LABEL ( CONTAINER_OF ),
    LABEL ( IS_PART_OF ),
    LABEL ( IS_LOCATED_IN ),
      -- some labels with properties used for edges
    LABEL ( KNOWS
        { creationDate          : TIMESTAMP}
      ),
    LABEL ( HAS_MEMBER
        { joinDate              : TIMESTAMP}
      ),
    LABEL ( WORK_AT
        { workFrom              : INTEGER}
      ),
    LABEL ( STUDY_AT
        { classYear             : INTEGER}
      ),
    LABEL ( LIKES
        { creationDate          : TIMESTAMP}
      ),


-- node label set declarations. This is a “closed-world”
specification
-- no other node label sets are permitted/appear in the graph
    (Message, Post),
    (Message, Comment),
    (Continent, Place),
    (Country, Place),
    (City, Place),
    (University, Organisation),
    (Company, Organisation),
    (Tag),
    (Person),
    (Forum),
    (TagClass),

-- edge label set declarations. Since all of these are single
labels,
-- and all of them appear in at least one edge triplet, this list
-- could be omitted. This is not the case for the above node label
-- sets, as some of them have multiple labels, and so all must be
-- specified.
-- The use of brackets is strongly preferred. Although the parser
can
-- cope with an unadorned label name, that is visually confusing to
-- the human reader.
    [IS_SUBCLASS_OF],
    [LIKES],
    [KNOWS],
    [STUDY_AT],
    [HAS_INTEREST],
    [WORK_AT],
    [IS_LOCATED_IN],
    [HAS_MEMBER],
    [REPLY_OF],
    [HAS_MODERATOR],
    [HAS_CREATOR],
    [IS_PART_OF],
    [HAS_TYPE],
    [CONTAINER_OF],
    [HAS_TAG],

    -- edge triplets
          -- the default cardinality <0..*> is omitted
    ("Country")       - [IS_PART_OF] ->     <1>      ("Continent"),
    ("Forum")         - [HAS_TAG] ->                 ("Tag"),
    ("Person")        - [IS_LOCATED_IN] ->  <1>      ("City"),
    ("Comment")       - [REPLY_OF] ->       <1>      ("Message"),
    ("University")    - [IS_LOCATED_IN] ->  <1>      ("City"),
    ("Person")        - [HAS_INTEREST] ->            ("Tag"),
    ("TagClass")      - [IS_SUBCLASS_OF] -> <1>      ("TagClass"),
    ("City")          - [IS_PART_OF] ->     <1>      ("Country"),
    ("Person")        - [WORK_AT] ->                 ("Company"),
    ("Forum")         - [HAS_MODERATOR] ->  <1>      ("Person"),
    ("Forum")         - [HAS_MEMBER] ->     <1..*> ("Person"),
    ("Message")       - [HAS_CREATOR] ->    <1>      ("Person"),
    ("Tag")           - [HAS_TYPE] ->       <1>      ("TagClass"),
    ("Company")       - [IS_LOCATED_IN] ->  <1>      ("Country"),
    ("Message")       - [HAS_TAG] ->                 ("Tag"),
    ("Message")       - [IS_LOCATED_IN] ->  <1>      ("Country"),
    ("Person")        - [STUDY_AT] ->                ("University"),
    ("Person")    <1> - [LIKES] ->                   ("Message"),
    ("Forum")     <1> - [CONTAINER_OF] ->   <1..*> ("Post"),
    ("Person")        - [KNOWS] ->                   ("Person")
-- end of graph schema declaration
    )
```

4.2 Mapping DDL

(this would follow on in the same file as the previous DDL block - the heading is present only to aid reader navigation)

```
     -- DDL describing table to property graph mappings

     -- mappings for nodes, showing the node label set and input
     -- table for each. In this case, each node label set is loaded
     -- from a single table. (A single table can only supply entries
     -- for one node label set, though it can also be “re-read” to
     -- to supply relationships. In the (peculiar ?) case that some
     -- table is supposed to supply to sets of nodes with different
     -- label sets (but identical cardinality), a SELECT * FROM …
     -- view must be used.)

     -- Since all the input tables for the node label sets are
     -- referenced in at least one relationship label set, some or
     -- all of these could be omitted, and inferred from the
     -- relationship mappings.
    NODE LABEL SETS (
        (University, Organisation)
             FROM "University",

        (Country, Place)
             FROM "Country",

        (Comment, Message)
             FROM "Comment",

        (Company, Organisation)
             FROM "Company",

        (Continent, Place)
             FROM "Continent",

        (City, Place)
```

```
            FROM "City",

        (Tag)
            FROM "Tag",

        (TagClass)
            FROM "TagClass",

        (Person)
            FROM "Person",

        (Message, Post)
            FROM "Post",

        (Forum)
            FROM "Forum"
    )
```

-- mappings for edges

```
    RELATIONSHIP LABEL SETS (
           -- this first label set differs from the others, in that
           -- one of the node tables has a reference column to the
           -- other.
        (CONTAINER_OF)
            FROM "Post" edge
                START NODES
                    LABEL SET (Forum)
                    FROM "Forum" start_nodes
                        JOIN ON start_nodes.ID = edge."forum"
                END NODES
                    LABEL SET (Message, Post)
                    -- edge table is also start table,
                    -- each row joining to itself
                    FROM "Post" end_nodes
                        JOIN ON end_nodes.ID = edge.ID,

           -- the “isSubclassOf” table is (effectively) a link table
           -- with references to the nodes at either end.
        (IS_SUBCLASS_OF)
            FROM "isSubclassOf" edge
                START NODES
                    LABEL SET (TagClass)
                    FROM "TagClass" start_nodes
                        JOIN ON start_nodes.ID = edge."subTagClass"
                END NODES
                    LABEL SET (TagClass)
                    FROM "TagClass" end_nodes
                        JOIN ON end_nodes.ID = edge."superTagClass",

           -- the IS_PART_OF relationship applies to two pairs
           -- of node label sets
        (IS_PART_OF)
            FROM "cityIsPartOf" edge
                START NODES
                    LABEL SET (City, Place)
                    FROM "City" start_nodes
                        JOIN ON start_nodes.ID = edge."city"
                END NODES
                    LABEL SET (Country, Place)
                    FROM "Country" end_nodes
                        JOIN ON end_nodes.ID = edge."country",
            FROM "countryIsPartOf" edge
                START NODES
                    LABEL SET (Country, Place)
                    FROM "Country" start_nodes
                        JOIN ON start_nodes.ID = edge."country"
                END NODES
                    LABEL SET (Continent, Place)
                    FROM "Continent" end_nodes
                        JOIN ON end_nodes.ID = edge."continent",

        (HAS_TYPE)
            FROM "hasType" edge
                START NODES
                    LABEL SET (Tag)
                    FROM "Tag" start_nodes
                        JOIN ON start_nodes.ID = edge."tag"
                END NODES
                    LABEL SET (TagClass)
                    FROM "TagClass" end_nodes
                        JOIN ON end_nodes.ID = edge."tagClass",

        (KNOWS)
            FROM "knows" edge
                START NODES
                    LABEL SET (Person)
                    FROM "Person" start_nodes
                        JOIN ON start_nodes.ID = edge."person1"
                END NODES
                    LABEL SET (Person)
                    FROM "Person" end_nodes
                        JOIN ON end_nodes.ID = edge."person2",
        (HAS_TAG)
            FROM "postHasTag" edge
                START NODES
                    LABEL SET (Message, Post)
                    FROM "Post" start_nodes
                        JOIN ON start_nodes.ID = edge."post"
                END NODES
                    LABEL SET (Tag)
                    FROM "Tag" end_nodes
                        JOIN ON end_nodes.ID = edge."tag",
            FROM "commentHasTag" edge
                START NODES
                    LABEL SET (Comment, Message)
                    FROM "Comment" start_nodes
                        JOIN ON start_nodes.ID = edge."comment"
                END NODES
                    LABEL SET (Tag)
                    FROM "Tag" end_nodes
                        JOIN ON end_nodes.ID = edge."tag",
            FROM "hasTag" edge
                START NODES
                    LABEL SET (Forum)
                    FROM "Forum" start_nodes
                        JOIN ON start_nodes.ID = edge."forum"
                END NODES
                    LABEL SET (Tag)
                    FROM "Tag" end_nodes
                        JOIN ON end_nodes.ID = edge."tag",

        (HAS_INTEREST)
            FROM "hasInterest" edge
                START NODES
                    LABEL SET (Person)
                    FROM "Person" start_nodes
                        JOIN ON start_nodes.ID = edge."member"
                END NODES
                    LABEL SET (Tag)
                    FROM "Tag" end_nodes
                        JOIN ON end_nodes.ID = edge."interest",

        (HAS_MODERATOR)
            FROM "hasModerator" edge
                START NODES
                    LABEL SET (Forum)
                    FROM "Forum" start_nodes
```

```
                JOIN ON start_nodes.ID = edge."forum"
        END NODES
            LABEL SET (Person)
            FROM "Person" end_nodes
                JOIN ON end_nodes.ID = edge."moderator",

(REPLY_OF)
    FROM "replyOfPost" edge
        START NODES
            LABEL SET (Comment, Message)
            FROM "Comment" start_nodes
                JOIN ON start_nodes.ID = edge."reply"
        END NODES
            LABEL SET (Message, Post)
            FROM "Post" end_nodes
                JOIN ON end_nodes.ID = edge."post",
    FROM "replyOfComment" edge
        START NODES
            LABEL SET (Comment, Message)
            FROM "Comment" start_nodes
                JOIN ON start_nodes.ID = edge."reply"
        END NODES
            LABEL SET (Comment, Message)
            FROM "Comment" end_nodes
                JOIN ON end_nodes.ID = edge."comment",

(LIKES)
    FROM "likesPost" edge
        START NODES
            LABEL SET (Person)
            FROM "Person" start_nodes
                JOIN ON start_nodes.ID = edge."person"
        END NODES
            LABEL SET (Message, Post)
            FROM "Post" end_nodes
                JOIN ON end_nodes.ID = edge."post",
    FROM "likesComment" edge
        START NODES
            LABEL SET (Person)
            FROM "Person" start_nodes
                JOIN ON start_nodes.ID = edge."person"
        END NODES
            LABEL SET (Comment, Message)
            FROM "Comment" end_nodes
                JOIN ON end_nodes.ID = edge."comment",
(STUDY_AT)
    FROM "studyAt" edge
        START NODES
            LABEL SET (Person)
            FROM "Person" start_nodes
                JOIN ON start_nodes.ID = edge."person"
        END NODES
            LABEL SET (University, Organisation)
            FROM "University" end_nodes
                JOIN ON end_nodes.ID = edge."university",

(IS_LOCATED_IN)
    FROM "personIsLocatedIn" edge
        START NODES
            LABEL SET (Person)
            FROM "Person" start_nodes
                JOIN ON start_nodes.ID = edge."person"
        END NODES
            LABEL SET (City, Place)
            FROM "City" end_nodes
                JOIN ON end_nodes.ID = edge."city",
    FROM "companyIsLocatedIn" edge
        START NODES
            LABEL SET (Company, Organisation)
            FROM "Company" start_nodes
                JOIN ON start_nodes.ID = edge."company"
        END NODES
            LABEL SET (Country, Place)
            FROM "Country" end_nodes
                JOIN ON end_nodes.ID = edge."country",
    FROM "universityIsLocatedIn" edge
        START NODES
            LABEL SET (University, Organisation)
            FROM "University" start_nodes
                JOIN ON start_nodes.ID = edge."university"
        END NODES
            LABEL SET (City, Place)
            FROM "City" end_nodes
                JOIN ON end_nodes.ID = edge."city",
    FROM "postIsLocatedIn" edge
        START NODES
            LABEL SET (Message, Post)
            FROM "Post" start_nodes
                JOIN ON start_nodes.ID = edge."post"
        END NODES
            LABEL SET (Country, Place)
```

```
                FROM "Country" end_nodes
                    JOIN ON end_nodes.ID = edge."country",
        FROM "commentIsLocatedIn" edge
            START NODES
                LABEL SET (Comment, Message)
                FROM "Comment" start_nodes
                    JOIN ON start_nodes.ID = edge."comment"
            END NODES
                LABEL SET (Country, Place)
                FROM "Country" end_nodes
                    JOIN ON end_nodes.ID = edge."country",

    (HAS_MEMBER)
        FROM "hasMember" edge
            START NODES
                LABEL SET (Forum)
                FROM "Forum" start_nodes
                    JOIN ON start_nodes.ID = edge."forum"
            END NODES
                LABEL SET (Person)
                FROM "Person" end_nodes
                    JOIN ON end_nodes.ID = edge."member",

    (WORK_AT)
        FROM "workAt" edge
            START NODES
                LABEL SET (Person)
                FROM "Person" start_nodes
                    JOIN ON start_nodes.ID = edge."person"
            END NODES
                LABEL SET (Company, Organisation)
                FROM "Company" end_nodes
                    JOIN ON end_nodes.ID = edge."company",

    (HAS_CREATOR)
        FROM "postHasCreator" edge
            START NODES
                LABEL SET (Message, Post)
                FROM "Post" start_nodes
                    JOIN ON start_nodes.ID = edge."post"
            END NODES
                LABEL SET (Person)
                FROM "Person" end_nodes
                    JOIN ON end_nodes.ID = edge."creator",
        FROM "commentHasCreator" edge
            START NODES
```

```
                    LABEL SET (Comment, Message)
                    FROM "Comment" start_nodes
                        JOIN ON start_nodes.ID = edge."comment"
                END NODES
                    LABEL SET (Person)
                    FROM "Person" end_nodes
                        JOIN ON end_nodes.ID = edge."creator"
    )
```

5 Label set aliasing

In the graph schema DDL, an alias can be assigned to a node or edge label set, and this alias can then be used in edge triplets and in the mapping ddl as a synonym for the label set. This can be viewed as giving a single name to a type. For example, in the SNB example above, the two “sub-types” of Message could be defined in the graph schema with:

```
    (Message, Post) as PostMsg,
    (Message, Comment) as CommentMsg
```

The edge triplet for CONTAINER_OF could then be shown as

```
    ("Forum")     <1> - [CONTAINER_OF] ->   <1..*> ("PostMsg"),
```

And in the mapping DDL, the mapping for nodes from table Post could be

```
        (PostMsg)
             FROM "Post"
```

And similarly in the edge mappings, such as

```
    (HAS_CREATOR)
            FROM "postHasCreator" edge
                START NODES
                    LABEL SET (PostMsg)
                    FROM "Post" start_nodes
                        JOIN ON start_nodes.ID = edge."post"
                END NODES
                    LABEL SET (Person)
                    FROM "Person" end_nodes
                        JOIN ON end_nodes.ID = edge."creator"
```

6 Example with multiple input tables for a node label set

This example was created to show and test the case where one node label set is created from more than one input table. In many cases, this could be handled by making a UNION ALL of the tables, but in this case the distinguishing columns (primary keys) of the input tables are different. Loading these as one would require creation of a new synthetic key. Since this is what the loading will do anyway, creation of a UNION view is unnecessary. The use of the join syntax for mappings makes clear what is intended.

Scenario: Persons are observed in towns. Persons can be settlers, who have unambiguous idNumbers, or nomads, who do not have id numbers, but their personal name is unique within their clan. Both have first and last names, which is all that is wanted for the graph - which may mean there will be more than one node in the graph with the same first and last name as another, but representing a different person. Note that the “clan” name for nomads is part of the “key” used to identify the node in loading, but is not retained in the graph data.

The example also shows an exception to the “use views whenever possible” In the “nomads” table, the column that will be mapped to property “first_name” is call “personal_name”. The implementation allows simple renaming.

Both node input tables have a reference column to the town the person was observed in, so in both cases, the node table is also the edge table.

6.1 Table creation DDL

```
(to show what the graph will be loaded from)

CREATE TABLE towns
     (town_name  VARCHAR(32) PRIMARY KEY,
      population INTEGER );

CREATE TABLE settlers
(
     first_name           VARCHAR(32) NOT NULL,
     last_name            VARCHAR(32) NOT NULL,
     idNumber             INTEGER  PRIMARY KEY,
     town                 VARCHAR(32) REFERENCES towns(town_name)
);

CREATE TABLE nomads
(
     personal_name        VARCHAR(32) NOT NULL,
     last_name            VARCHAR(32) NOT NULL,
     clan                VARCHAR(32) NOT NULL,
     town                VARCHAR(32) REFERENCES towns(town_name),
          CONSTRAINT nomads_pk  PRIMARY KEY (clan, personal_name)
);
```

6.2 Graph schema DDL

The labels and the schema are declared independently of each other and the graph mapping. The label declarations use the SQL-style syntax. All graph object names that use mixed case are quoted.

```
CREATE LABEL "Town"
    PROPERTIES
        ("town_name"           VARCHAR(32) NOT NULL)

CREATE LABEL "Person"
    PROPERTIES
        ("first_name"          VARCHAR(32) NOT NULL,
         "last_name"           VARCHAR(32) NOT NULL)

CREATE LABEL SEEN_IN

CREATE GRAPH SCHEMA "Observation"
         -- both the permitted node label sets and edge label sets
         -- can inferred from the edge triplets, so need not be
         -- stated explicitly

    ("Person")    <0 .. *>    - [SEEN_IN] ->     <0 .. *>   ("Town")
```

6.3 Mapping DDL

```
CREATE GRAPH DESERT WITH GRAPH SCHEMA Observation
     -- the graph schema is referenced by name

    NODE LABEL SETS (
        (Town)
             FROM TOWNS,

        (Person)
             FROM SETTLERS,
             FROM NOMADS
                  (PERSONAL_NAME AS "first_name")
    )

    RELATIONSHIP LABEL SETS (

        (SEEN_IN)
            FROM SETTLERS edge
                START NODES
                    LABEL SET (Person)
                    -- edge table is also start table,
                     -- each row joining to itself
                    FROM SETTLERS start_nodes
                        JOIN ON start_nodes.IDNUMBER = edge.IDNUMBER
                END NODES
                    LABEL SET (Town)
                    FROM TOWNS end_nodes
                        JOIN ON end_nodes.TOWN_NAME = edge.TOWN,
            FROM NOMADS edge
                START NODES
                    LABEL SET (Person)
                    -- edge table is also start table,
                     -- each row joining to itself
                    FROM NOMADS start_nodes
                            (PERSONAL_NAME AS "first_name")
                        JOIN ON
                      start_nodes.PERSONAL_NAME = edge.PERSONAL_NAME
                        AND start_nodes.CLAN = edge.CLAN
                END NODES
                    LABEL SET (Town)
                    FROM TOWNS end_nodes
                        JOIN ON end_nodes.TOWN_NAME = edge.TOWN
    )
```

7 Example showing M:M relationship

This example shows a mapping from relational data to a graph where the representation of the relationships in the tables is many-to-many rather than one-to-one as in the previous examples. One-to-many is also possible.

The input data has two tables - “officers” and “subordinates”. Both have just two fields, a name, which is unambiguous, and a department name. Several officers and several subordinates can . be in the same department. The graph is required to have a node for each officer and each subordinate, with an OBEYS relationship from each subordinate to all the officers in the department.

7.1 Graph DDL

The SQL-style is used fo the label declarations, with an in-line graph schema

```
CREATE GRAPH HIERARCHY
     -- describe the pure property graph
     WITH GRAPH SCHEMA (
               LABEL OBEYS
                   PROPERTIES
                       ("department"          VARCHAR(30) NOT NULL)

               LABEL "Subordinate"
                   PROPERTIES
                       ("name"                VARCHAR(30) NOT NULL)

               LABEL "Officer"
                   PROPERTIES
                       ("name"                VARCHAR(30) NOT NULL)

         -- both the permitted node label sets and edge label sets
         -- can inferred from the edge triplets, so need not be
         -- stated explicitly

         ("Subordinate") - [OBEYS] ->  ("Officer")
     )
```

7.2 Mapping DDL

(again, this follows on directly from the graph schema DDL above)

```
     -- describe the table to property graph mappings
    NODE LABEL SETS (
        (Subordinate)
             FROM SUBORDINATES,

        (Officer)
             FROM OFFICERS
    )

    RELATIONSHIP LABEL SETS (

        (OBEYS)
            FROM SUBORDINATES edge
                START NODES
                    LABEL SET (Subordinate)
                    FROM SUBORDINATES start_nodes
                      JOIN ON start_nodes.NAME = edge.NAME
                END NODES
                    LABEL SET (Officer)
                    FROM OFFICERS end_nodes
                      JOIN ON end_nodes.DEPARTMENT = edge.DEPARTMENT
    )
```

Note that the start (source) node table is also the edge table and the join for the start nodes is just re-identifying the node that was created from that row of the table, as in the previous example.

For the end (destination) side, the row from the edge table is matched against all of the rows in the end table with the same department.

This processing can be described as a variation of the processing described in [ERF-042] 1.5.2 b). Instead of creating an edge, with an assigned identifer directly from each row of the edge table, a "potential" edge (a template) is created. The template then becomes zero, one or more edges when matched against the node tables, rather than flag the edge as incomplete or multi-sourced.

7.3 Example data

Given the following input tables

Table OFFICERS:

NAME	**DEPARTMENT**
Arthur	alpha
Angela	alpha
Brian	beta
Gustav	gamma
George	gamma
Gertrude	gamma
David	delta
Diana	delta

Table SUBORDINATES

NAME	**DEPARTMENT**
Adams	alpha
Atkinson	alpha
Brown	beta
Emmett	epsilon
Erskine	epsilon
Gardner	gamma

Gershwin	gamma
Goddard	gamma

Then, the equivalent of the Cypher query

```
MATCH (s:Subordinate)-[d:OBEYS]->(o:Officer)
     RETURN s.name AS subordinate, o.name AS officer,
           d.department AS department
```

Produces the following output

subordinate	**officer**	**department**
Atkinson	Arthur	alpha
Atkinson	Angela	alpha
Adams	Arthur	alpha
Adams	Angela	alpha
Brown	Brian	beta
Gershwin	Gustav	gamma
Gershwin	George	gamma
Gershwin	Gertrude	gamma
Goddard	Gustav	gamma
Goddard	George	gamma
Goddard	Gertrude	gamma
Gardner	Gustav	gamma
Gardner	George	gamma
Gardner	Gertrude	gamma

8 Self-referencing M:M example

A variation on the example above can be created where there is only a one set of nodes and the relationship is just which of them share some attribute. We can re-use the data above, but with a SAME_DEPT relationship among the subordinates.

If the mapping is defined with

```
(SAME_DEPT)
    FROM SUBORDINATES edge
        START NODES
            LABEL SET ("Subordinate")
            FROM SUBORDINATES start_nodes
              JOIN ON start_nodes.NAME = edge.NAME
        END NODES
            LABEL SET ("Subordinate")
            FROM SUBORDINATES end_nodes
```

```
                                  JOIN ON end_nodes.DEPARTMENT = edge.DEPARTMENT
```

Then there will be an edge from every node back to itself. If this is not desired, the mapping for the end nodes can exclude such loop backs (highlighted):

```
   (SAME_DEPT)
            FROM SUBORDINATES edge
                START NODES
                    LABEL SET ("Subordinate")
                    FROM SUBORDINATES start_nodes
                      JOIN ON start_nodes.NAME = edge.NAME
                END NODES
                    LABEL SET ("Subordinate")
                    FROM SUBORDINATES end_nodes
                      JOIN ON end_nodes.DEPARTMENT = edge.DEPARTMENT
                            AND end_nodes.NAME != edge.NAME
```

Possibly the not-equals condition should reference the start_nodes correlation name rather than the edge, although these are the same table in this case. If they are not the same, this additional mechanism probably is not needed.

Property Graph Data Model Concepts and Terms

Title	Property Graph Data Model Concepts and Terms
Author	Alastair Green, U.S.A National Expert[1]
Status	Discussion Paper
Date	1 October 2018
r1	30 October 2018

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

Contents

[1] Member of of the Neo4j SQL working group (Alastair Green, Hannes Voigt, Peter Furniss, Petra Selmer, Stefan Plantikow, Tobias Lindaaker) whose other members have contributed comments during the preparation of this paper. We don't all agree on all of what is said, but we all feel the thrust is correct and the points raised are important for refining the planned PGQ IWD.

1. References

[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 PG DM]	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]	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
[Graph DDL]	ISO/IEC JTC1/SC32 WG3:ERF-043 *ANSI INCITS sql-pg-2018-0036r2* Peter Furniss, **“SQL/PG graph schema and join syntax mapping examples”**, September 2018

2. Introduction

This paper proposes *adding* some concepts (element keys, label expression/set types, label and label set tables/datasets) to the pure property graph abstract data model described in Jan Michel’s paper [PPG data model], and *revising* some of the terms used to describe existing concepts in that model.

The syntax and descriptions used in Peter Furniss’ [Graph DDL] would also be affected by the changes proposed in this paper.

Those changes are not viewed as antagonistic to the mental model or resulting designs in those two papers.

The resulting amended metamodel for a pure property graph and for a tabular data model is shown as a revised version of the UML diagram presented first in [SQL PG DM].

The paper in part reflects prior submissions by Neo4j to the SQL PG Ad Hoc; in part responds to comments raised by Romans Kasperovics of SAP and Mingxi Wu of TigerGraph in the 25 September 2018 meeting of the INCITS D32.2 Ad Hoc for SQL Property Graph Extensions; and in part reflects implementation work in this area using the open source Cypher for Apache Spark project.

Terms in bold like **elements** are defined terms and intended descriptors.

3. Revised Terms

3.1. Graph Schema

The structure and the values of a particular graph at a particular point in logical time (a **graph instance**) can be thought of as being circumscribed by a **graph schema**, which is metadata that describes the permitted structure and values of that graph.

The consensus of the SQL PG Ad Hoc is that no property of any element in the graph shall exist unless the graph schema specifies that property as belonging to a label which is part of a vertex label set or an edge label set, as defined in [PPG data model]. This restriction, which is not present in schema-optional languages like Cypher, PGQL and G-CORE, means that SQL PGQ conforms to the closed-world schema model of SQL (modulo Polymorphic Table Functions).

The term “graph schema” has occasionally caused confusion. From an SQL perspective, it can be seen to clash with the notion of a catalog schema, in whose scope other objects like tables and views are defined. On the other hand it is a well-established term in the property graph community, being a synonym for graph metadata.

The SQL standard itself seems to embed this confusion, apparently using the term “schema” in two ways. First, to define a catalog subdivision/container of metadata as in

```
mydatabase.myschema.mytable_or_view
```

And second, as a generic term for metadata objects, as we can see from this BNF fragment, which treats catalog schema definition and table definition as both being “schema definition” activities:

```
<SQL schema definition statement> ::=
  <schema definition>
```

```
      | <table definition>
      | <view definition> ...
```

However, if we view all objects within a schema as part of the schema, then the apparent ambiguity of the term, as used in the standard, resolves.

3.2. Better: Graph Type

An alternative term for graph schema would be **graph type**.

By analogy, one can define a table, and in the course of that definition lay out an anonymous row type (or implicitly create a row type if one is not explicitly defined).

```
CREATE TABLE my_table (ROW (a CHAR(2),
                            b INTEGER,
                            c ROW (ca CHAR(2),
                                   cb CHAR(2))))
```

You could also define (SQL standard) a named user-defined structured type, or (taking PostgreSQL as an example of an implementation extension) a named row type. Using PostgreSQL syntax:

```
CREATE TYPE my_row (a CHAR(2),
                    b INTEGER,
                    c ROW (ca CHAR(2),
                           cb CHAR(2))))

CREATE TABLE my_table (content my_row)
```

Turning back to graphs: in the syntax examples shown in [Graph DDL] we see this kind of graph definition:

```
CREATE PROPERTY GRAPH snb
  WITH [[PROPERTY] GRAPH] SCHEMA
    (LABEL "Person"
       PROPERTIES (creationDate TIMESTAMP NOT NULL,
                   firstName    VARCHAR(255),
                   lastName     VARCHAR(255),
                   gender       CHAR(1))
```

If we thought of this as declaring an anonymous type in-line with creating a graph (like ROW) then we could instead see:

```
CREATE [PROPERTY] GRAPH snb
  WITH [[PROPERTY] GRAPH] TYPE
```

```
(LABEL "Person" ...
 /*
    more labels
    vertex label sets
    edge label sets
    edge triplets using
      source vertex, destination vertex and edge label subsets
 */
)
```

This is interesting, again by analogy with existing SQL, in that the metadata for a table can be thought of as a complex row type (whether specified or inferred) and constraints.

4. Additional Concepts and Features

4.1. Graph Identifier

A graph cannot exist other than as an object viewable in the Information Schema of a catalog [Schemata:2020]. It must be defined as a part of a user-defined schema, which may contain many graphs. The schema-qualified name of a graph object viewed by the Information Schema of a catalog is a unique identifier for a graph within that catalog.

4.2. Elements

A graph is made up of two sets: the set of nodes (vertices) and the set of edges. The union of those two sets can be termed a set of **elements**.[2] An **element** is therefore a term for a node or edge, and is an object that may have labels and therefore may have properties. We can say that there are two **sorts** of elements, nodes and edges.

4.3. Label Expressions and Label Sets

Label sets allow zero or more labels to be combined (conjoined) to define a set of label identifiers and their possible properties and their names and types. Every label is considered to be a label set with one member.

A label set is a special case of a **label expression** (a concept introduced in the context of pattern matching, see [Graph patterns]), in which labels are simply conjoined. (A label set is therefore a term for a label conjunction). A label expression allows potentially recursive conjunction, disjunction and negation of labels and of resulting label expressions.

[2] There is no well-established term in the research literature for this set, which is not interesting to graph theorists. It is a useful term for discussing characteristics which are common to nodes and edges in the graph data model, for example labels and properties. It is the term used by Apache Tinkerpop and by PGQL, and seems better than the term “entity” used by openCypher, which could be confused with an entity in an entity-relationship model (which corresponds to a type of node in a property graph).

For example, given labels `Person`, `Resident`, `Visitor`, `Dog`, `Pet`, `WorkingAnimal`, `Licensed` we could see several useful label expressions, like the following cases:

```
(Person & Resident)
(Person & Visitor)
(Dog & (Pet | Working Animal))
(Dog & WorkingAnimal & !Licensed)
```

The conjunction of labels is disallowed if a label in the proposed set has a property with the same name, but a different datatype, as a property in another member of the proposed set. If a label in the proposed set has the same name and the same datatype as a property in another member of the proposed set then the conjunction is allowed.

4.4. Label Expression Types

A label expression (including a label set "conjoining" one label) can be associated with a **label expression type**. A label expression type is either an intersection type, in which all properties of all conjoined label expressions are included (apart from properties from different labels with the same name, and therefore also with the same data type, which are coalesced into a single property of that name and type), or a union type.

All label expressions can be expanded into "conjunctive normal form" (i.e. into a label set), giving a set of label expressions that are conjunctions, and where all the associated label types are intersection types. These can equally be termed **label set types**. A union type is therefore a set of intersection or label set types.

The use of label expressions to define types enables mapping inheritance relationships without explicitly defining type hierarchies, but also enables relationships that are arbitrary or partial to be expressed (not all `Person` node modifications are `Audited`, for example).

A label expression type can be associated with an element sort, in which case that type becomes a **vertex type** or an **edge type** (collectively, **element types**).

A label expression type used for defining edge triplets would be termed a **triplet type**. A triplet type used to define a triplet source vertex type or destination vertex type must, in conjunctive normal form (viewed as a label set), be a subset of a vertex type; one used to define a triplet edge type must be a subset of a label set edge type.

It is possible to distinguish all label expression types by their label expression definition. However, it would be helpful to allow the naming of label expression types (which, in a form, is shown as aliasing in [Graph DDL]). This would enable naming of empty label sets, which would give us "marker interfaces", and it would also allow typedef-ing.

It would also be helpful to define a **label expression supertype** which could be used in label expression types of the following forms:

```
Person & !(LABEL_EXPRESSION & !Person)
Person & !(& !Person)
```

which allows us to say “Person and only Person”.

Such a supertype could also be used to defining edge triplets where the nature of the nodes is irrelevant, but the set of permitted edge types is restricted, or conversely where the nature of the edges is uninteresting:

```
()-[REGISTERED_BY]-(Municipality) -- dogs, people, cars ...
(Dog)-[]-() -- dogs without relationships are not allowed ...
```

4.5. Label Tables or Datasets

If the set of properties of a label is called the **label properties**, then the values of the label properties for an element with that label is a **label row or record**. The label rows of all the elements of the same sort for a given label constitute an **imputed label table or dataset**. A label table is a multiset if there is no label key, but a set if there are one or more label keys for that label.

An element may have multiple labels, and therefore its properties may have many (potentially intersecting) subsets, each of which is a label row in a distinct label table.

4.6. Label Set Tables or Datasets

An element has one most-specific label expression type (the intersection type that corresponds to a label set or conjunctive normal form label expression). The defining label set of such a type can be referred to as an **element label set** and if the set of properties of all of the labels in its label set is called its **element properties**, then the values of the element properties for an element is an **element row or record**. The element rows of all the elements of the same sort with the same element label set constitute an **imputed label set table**.

A label set table is a set if all the members of the element label set that gives rise to the table have at least one label key, but otherwise is a multiset.

A property graph is made up of a finite set of label set tables, which correspond to the vertex types and edge types defined as metadata. It is also made up of a larger set of label tables.

The *metamodel* says that a graph is a set of vertex labels sets and a set of edge label sets (which may be empty), and their associated types; the data *model* or *schema* for a concrete graph says that those sets have defined members; the data graph or *instance* has a table for each defined element type.

4.7. Imputed Tables and the Catalog

Note that the label and label set tables described above are imputed or inferred: they need not be catalog objects and they may never be materialized in the implementation, or projected as a result.

It would be conceivable to allow these tables to be named and used as inner objects of the complex graph object in the catalog. However, as Fred Zemke has pointed out, a similar effect would be created if an SQL view were constructed over a GRAPH_TABLE with a `MATCH (IS a & IS b)` graph query, taking the example of a node label set `(IS a & IS b)`.

4.8. Node and Edge Keys

Cypher implementations (Neo4j Database, Cypher for Apache Spark) have found it useful to define **keys** for sets of nodes or sets of edges. An **element key** is an optional set of mandatory (non-null) properties, which are a subset of the properties of an element. An element may have more than one key.

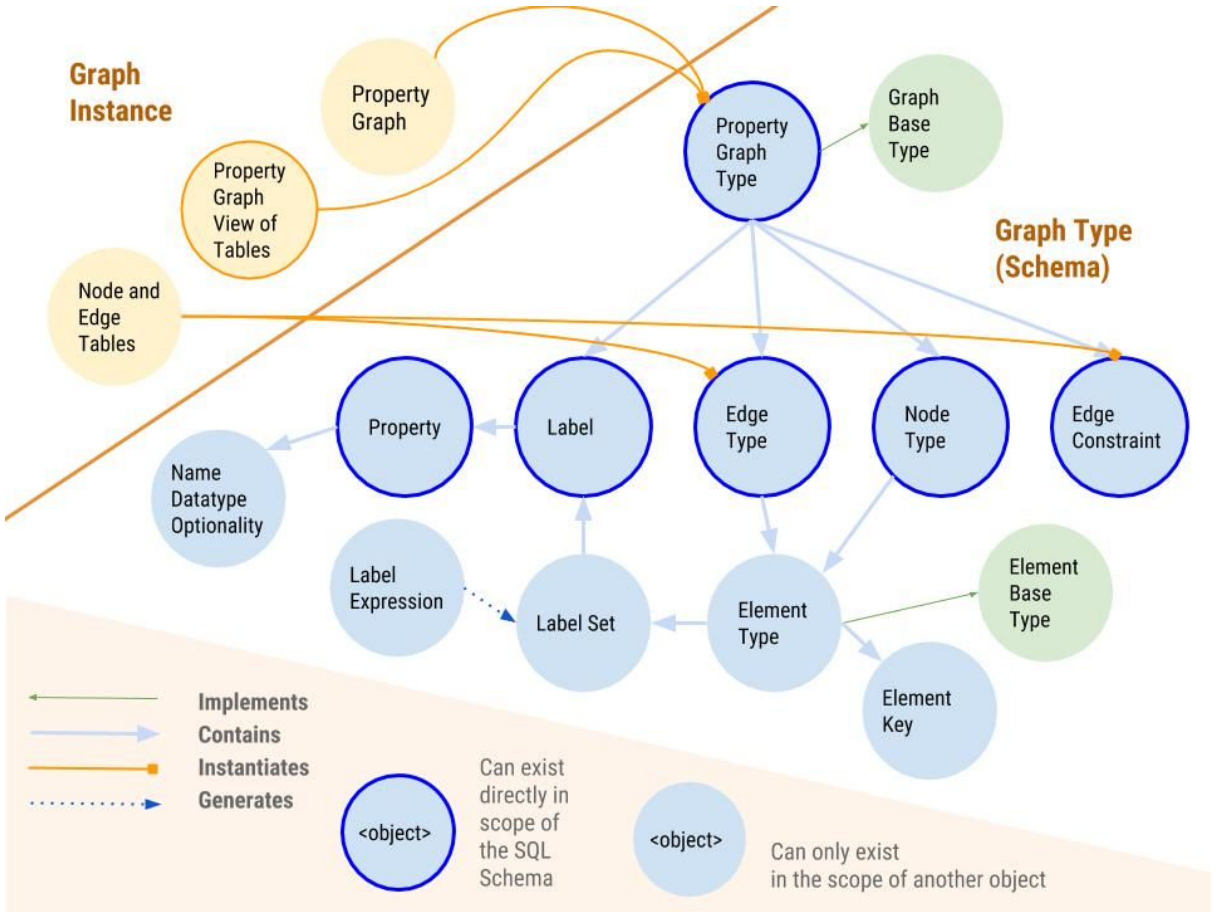
If an element of a graph has a key, then the value of the key for that element must differ from its value for all other elements in that graph with the same element type.

A key is therefore a candidate key for a label set table. When a label set is applied to vertices, creating a vertex type, and one or more keys are defined for that label set, then its keys are **vertex keys**, when applied to edges, creating an edge type, they are **edge keys**, together these are of course **element keys**.

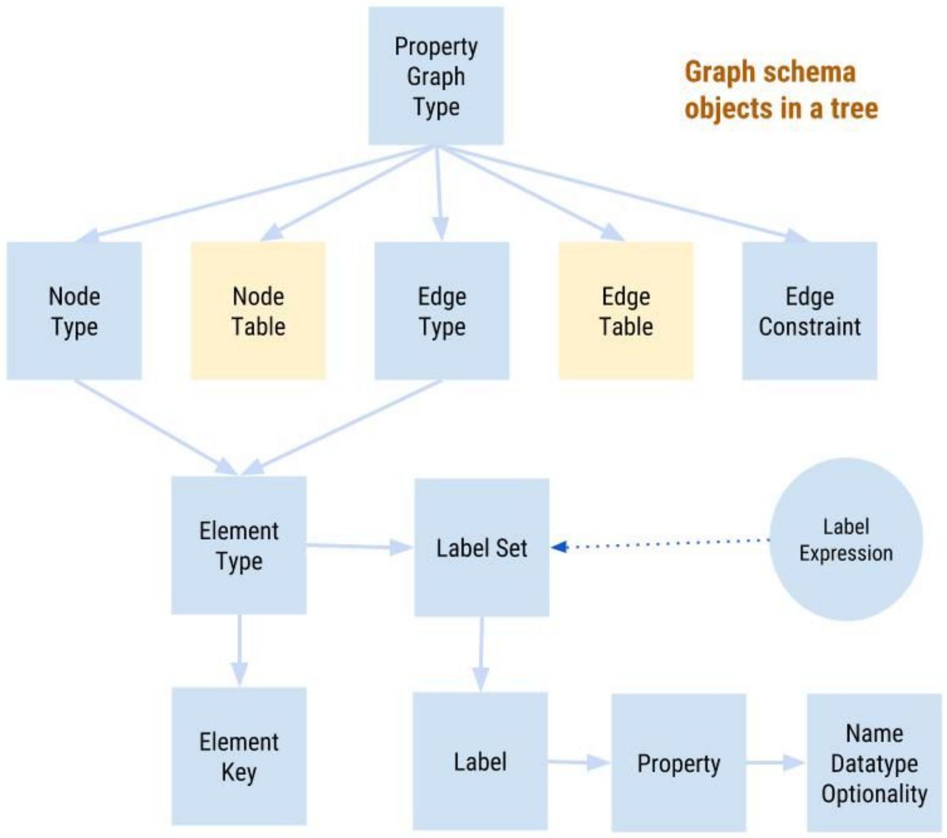
The properties that constitute an element key must be a subset of the mandatory properties of the element type.

5. Graph Types, Edge Types, Node Types

The diagram below summarizes the revised terminology (and relationships) first suggested in this paper.



6. Tree view of DDL declaration of a graph type



7. Named (Nominal) and Anonymous (Structural) Types

In the Neo4j SQL working group we are comfortable with the idea of named types being registered in the catalog's Definition Schema, for any of the type objects mentioned above, including graph types, as well as for edge triplet constraint objects.

These objects could then be attached to concrete data objects by reference during the definition of such objects, and could be used freely to create new complex types where appropriate. This is the approach we would prefer to see for the native, standalone GQL.

For SQL PGQ we are equally happy to start (and maybe finish) with anonymous types, including with anonymous graph types that must be defined within a graph object definition, in line with the approach taken for table definition (when user-defined structure types are not employed).

In the absence of names for inner objects like element types, the use of LIKE would have to be reserved for the highest level graph type:

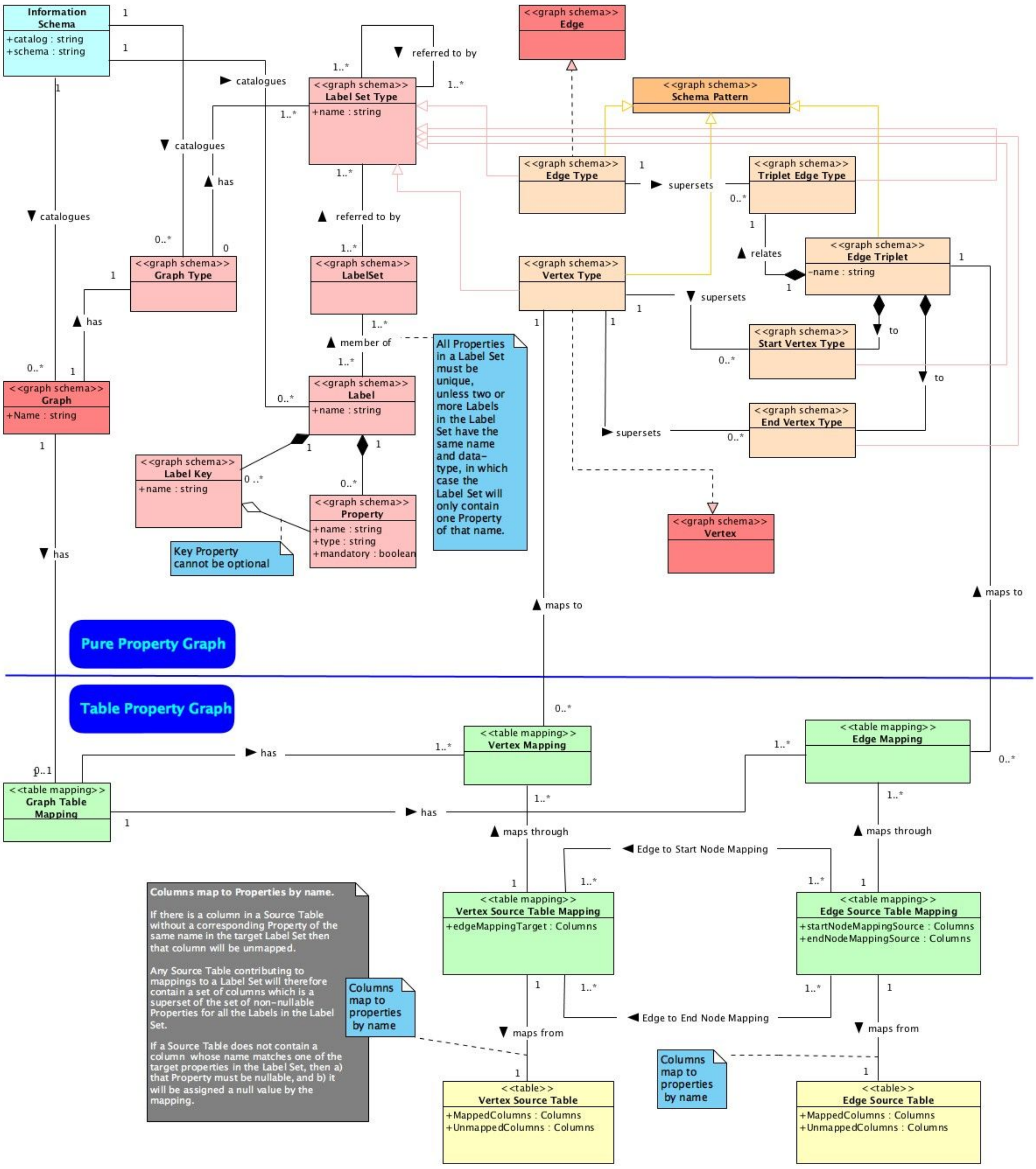
```
CREATE [PROPERTY] GRAPH snb_today
  LIKE [[PROPERTY] GRAPH] snb_yesterday
```

Allowing types to be named would allow the full, recursive system of label expression types to be expressed more economically. This could be viewed as a purely syntactic concern: it does not affect the model. In the absence of named label set types, it would only be possible to define element or triplet types as conjunctions, disjunctions or negations of label expressions themselves.

The complex label type system suggested here has no effect on the pattern matching sub-language used in MATCH, although it does provide information that would be of potential value to an optimizing implementation. It is possible to deduce much of the information in this model from the mappings represented by the table property graph described in [PPG data model], which in turn can be inferred from the join syntax for mapping tables to the pure property graph outlined in [Graph DDL]. With small extensions of the explicit graph schema or type definitions shown in the latter paper, all of the metadata described here could be defined.

We should note that it is necessary to have user-defined label definitions that are additive to existing table definitions to enable any mapping approach, and that this may militate in favour of allowing label set type definitions (including in terms of other label set type definitions), to be defined independently of the mappings.

8. UML Metamodel for the Pure and Table Property Graphs


Information Schema
+catalog : string
+schema : string
<<graph schema>> Label Set Type
+name : string
<<graph schema>> Edge
<<graph schema>> Schema Pattern
<<graph schema>> Edge Type
<<graph schema>> Triplet Edge Type
<<graph schema>> Edge Triplet
-name : string
<<graph schema>> Graph Type
<<graph schema>> LabelSet
<<graph schema>> Vertex Type
<<graph schema>> Start Vertex Type
<<graph schema>> End Vertex Type
<<graph schema>> Graph
+Name : string
<<graph schema>> Label
+name : string
<<graph schema>> Label Key
+name : string
<<graph schema>> Property
+name : string
+type : string
+mandatory : boolean
<<graph schema>> Vertex
catalogues
referred to by
has
supersets
relates
to
member of
maps to
Pure Property Graph
Table Property Graph
<<table mapping>> Vertex Mapping
<<table mapping>> Edge Mapping
<<table mapping>> Graph Table Mapping
<<table mapping>> Vertex Source Table Mapping
+edgeMappingTarget : Columns
<<table mapping>> Edge Source Table Mapping
+startNodeMappingSource : Columns
+endNodeMappingSource : Columns
<<table>> Vertex Source Table
+MappedColumns : Columns
+UnmappedColumns : Columns
<<table>> Edge Source Table
+MappedColumns : Columns
+UnmappedColumns : Columns
maps through
Edge to Start Node Mapping
Edge to End Node Mapping
maps from
1
1..*
0..*
0..1
0
0 ..*

All Properties in a Label Set must be unique, unless two or more Labels in the Label Set have the same name and data-type, in which case the Label Set will only contain one Property of that name.

Key Property cannot be optional

Columns map to Properties by name.

If there is a column in a Source Table without a corresponding Property of the same name in the target Label Set then that column will be unmapped.

Any Source Table contributing to mappings to a Label Set will therefore contain a set of columns which is a superset of the set of non-nullable Properties for all the Labels in the Label Set.

If a Source Table does not contain a column whose name matches one of the target properties in the Label Set, then a) that Property must be nullable, and b) it will be assigned a null value by the mapping.

Columns map to properties by name

Columns map to properties by name

9. 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[3] 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/

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 SQL working group to the openCypher project and to the SQL standard development process. It is also a contribution to the GQL standard project incubation community.

[3] https://www.opencypher.org/