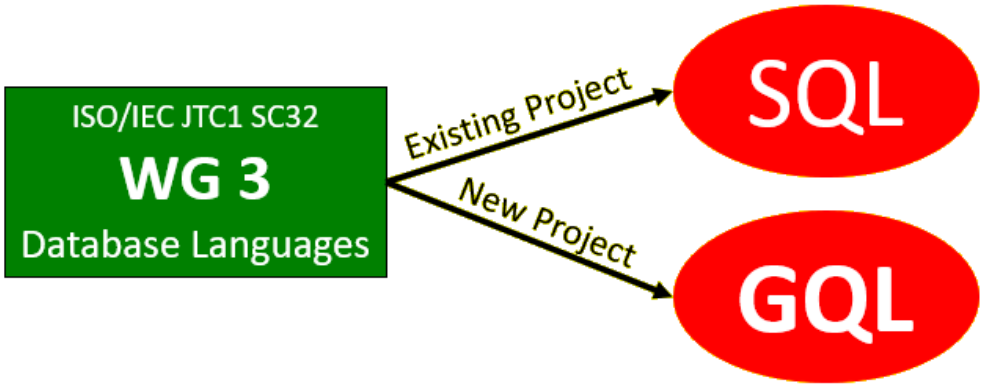

Fig. 1: GQL image (Source: Keith Hare)

GQL Scope and Features

Title: GQL Scope and Features
Authors: Neo4j Query Languages Standards and Research Team[1]
Status: Discussion Paper

Revisions: Revision 3, December 14, 2018
Subeditorial corrections; Added document numbers

Revision 2, November 29, 2018
Subeditorial corrections; Clarifications in 1.2 Summary of scope;
Added 3.6 Combinators; Additions to 4.2 Definitions;
Corrections in 3 Discussion, 4.4 Data types;
Include tables from [ERF-038] for 1.4 Concordances

Revision 1, November 12, 2018
Subeditorial corrections, including adding of references and related changes, and exchanged order of 4.7 and 4.8; Clarifications in 3.8 Design principles, 3.9 Motivation, 4.2 Definitions, 4.3 Type system, 4.6 Statements for graph pattern matching, 4.7 Statements for modifying graphs, 4.10.1 Nested procedures

Original, October 31, 2018

[1] Current members of the Neo4j Query Languages Standards and Research Team are: Alastair Green, Peter Furniss, Tobias Lindaaker, Petra Selmer, Hannes Voigt, Stefan Plantikow

Contents

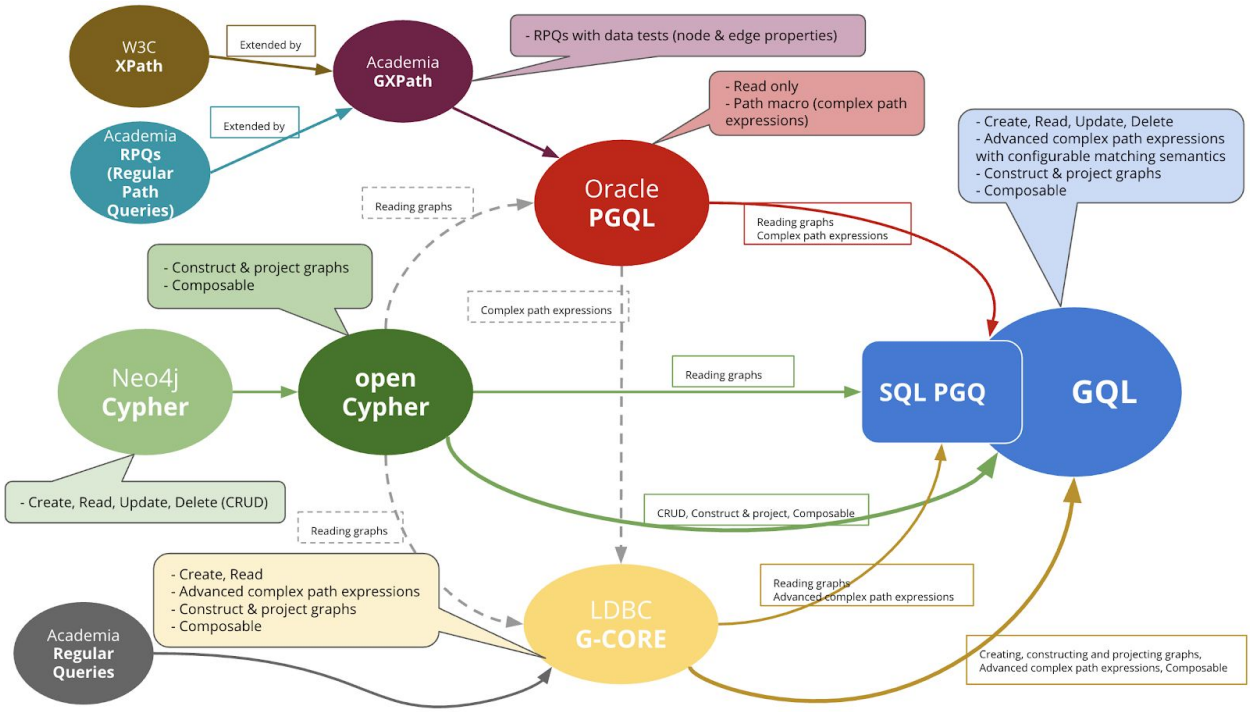

Fig. 2: Inputs to GQL

1 Introduction

The concept of a “standalone” or “native” property graph query language (GQL), as a complement to SQL, has been proposed independently in several contexts.

This document reflects the history described, and the approach proposed, in

[DM32.2-2018-00128] A. Green, “Working towards a New Work Item for GQL, to complement SQL PGQ”, July 2018

The diagram in [Fig. 2] helps to elucidate some of the roots of GQL in prior academic and industrial language designs. There are other angles of view, including prior practice in Apache Tinkerpop (Gremlin), existing standards for RDF graphs (RDF, OWL, SPARQL etc), and there is a growing interest in defining and expressing “graph schema” or “graph types”.

We propose an initial design for a GQL composable declarative database language for querying and maintaining property graphs. This document provides a high level overview of the content of the language, covering essential topics discussed, and the majority of the scope proposed in

[ERF-038] S. Plantikow, “Proposed GQL Scope and Landscape” ISO/IEC SC32/WG3:ERF-038 R1, October 2018

A concordance of [ERF-038] can be found in Section 1.4.2.

1.1 Inputs

As motivated in [DM32.2-2018-00128] and [YTZ-030r1], the primary basis underpinning the design of GQL are established industry property graph query languages such as openCypher [Cypher 9 Reference] and PGQL [PGQL], as well as related efforts, such as SQL:2020 Property Graph Querying (cf. [sql-pg-2017-0002] for an early discussion). These languages have shown the need for providing a standard property graph query language. Most importantly, they share a common foundation in

- their understanding of the property graph data model as a directed labeled multigraph whose elements have uniquely-named properties,
- the use of visual ("ASCII line art") syntax for pattern matching, and
- their aspiration to provide a composable language, especially by providing facilities for graph projection and construction.

We therefore consider GQL to be a natural outgrowth of these pre-existing languages that needs to honor established conventions regarding syntax and semantics, harmonize, complete and evolve existing features, as well as extend the provided functionality to ensure GQL is a viable, independent language.

Cypher and PGQL are not the sole inspiration for GQL. It also draws inspirations from other query languages such as SPARQL [SPARQL 1.1], G-CORE [G-CORE], GSQL [sql-pg-2018-0041], GXPath [GXPath], and JSON [JSON], as well as SQL.

The proposed design for GQL is intended to be built on a profile of the Framework [SQL Framework:2016] and Foundation of SQL [SQL Foundation:2016], and generally borrows from the look and feel of the SQL query language. However, GQL is an independent language, and does not include all constructs from SQL. GQL may deviate from SQL for reasons such as when it does not require supporting a feature that is outside the scope of GQL and already handled well by SQL. Moreover, GQL can be allowed to interoperate with SQL within the same query execution environment. GQL may also deviate where this makes sense for gaining a coherent and easy to explain mental model of the language and improving ease of use.

1.2 Summary of scope

GQL is intended to be independently practically useful languages that is capable of standing on its own. It is not intended to only be implemented by SQL vendors, but also by other kinds of database management systems. A design of GQL therefore needs to be fairly complete.

This document explores what this might require by covering a multitude of topics such as overall language structure, procedure composition, the type system, essential operations, and views. It also briefly touches on topics such as expressions, subqueries, catalog and schema, the execution model, the security model, and error handling. This list of topics could easily be extended but we've constrained it in this paper to items that we believe should be addressed as core features of GQL.

For now we have not addressed language modularization and conformance, sessions, transactions, cursors, constraints, and triggers, as well as support for bidirectional edges, the processing of streams or multidimensional data. These issues, and the features outlined in this paper may be the subject of future proposals.

A full scope of the proposed design is given in Section 4.1.

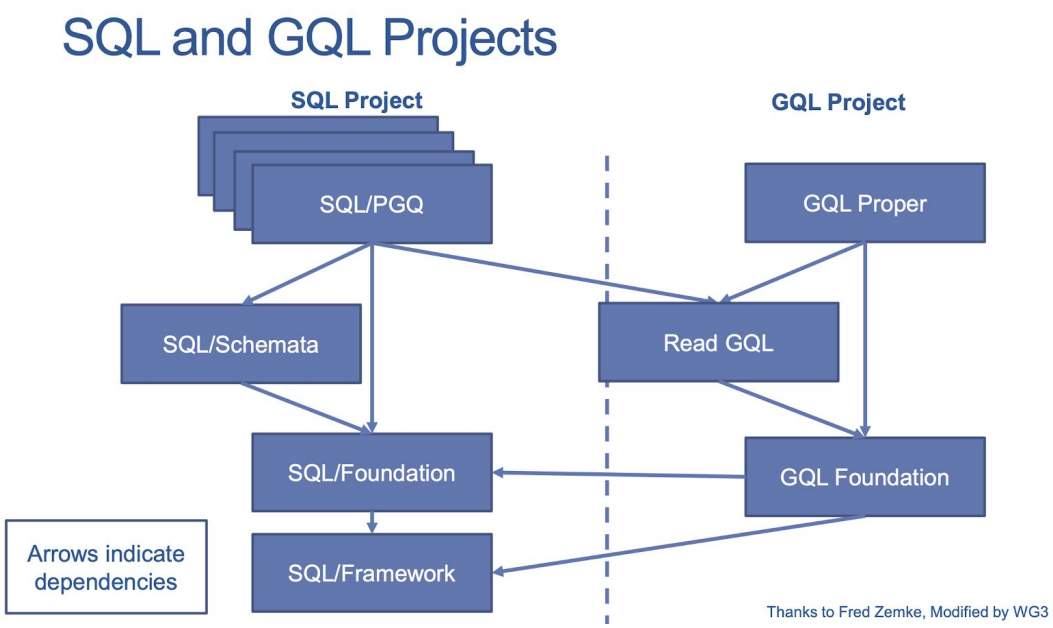

Fig. 3: SQL and GQL Projects (Source: Keith Hare and Fred Zemke, based on [ERF-037] by F. Zemke*)*

1.3 Next steps

A potential three-part structure of GQL was discussed at the last meeting of SC32/WG3 in Ilmenau, Germany (ref. [Fig. 3]). We would like to suggest the following alternative names for the three parts of GQL:

- GQL Language Definition (was: GQL Proper)

which incorporates by reference

- GQL Pattern Matching (was: Read GQL)
- GQL Profile of SQL (was: GQL Foundation)

We intend the content of this document to be mostly an input for the design of the GQL language. The proposed syntax for pattern matching belongs in GQL Pattern Matching. This proposal discusses new data types and expressions over and above what is provided by SQL. We would expect such additions to also belong to the GQL Language Definition atop a baseline provided by the GQL Profile of SQL.

Besides the need to further process the topics raised by this document in greater detail, it is our position that a viable next step on the road to GQL will be the creation of an outline of a GQL Language Definition base document. Such a base document should contain a GQL project scope and should generally follow the structure of SQL Foundation [SQL Foundation:2016].

1.4 Concordances

1.4.1 Concordance with [sql-pg-2018-0042]

The parallel discussion paper [sql-pg-2018-0042] has given a short initial proposal for a possible design of GQL, structurally following PGQL. We note many similarities but also differences with this paper, and here give a brief concordance.

Topic	*Section in each paper*	
● Both documents cover:	[sql-pg-2018-0042]	this paper
○ Scope	1	4.1
○ Query structure	2	4.3
○ Pattern matching and paths	3, 6	4.6
○ Tabular projection and aggregation	4	4.9
○ Data modification (Updates)	1.2	4.7
○ Subqueries	7	4.10
● [sql-pg-2018-0042] additionally covers:		
○ Joins	5	(4.12.2)
○ Grouping variables	6	
○ Path aggregation	6	
● This document additionally covers:		
○ Named, parameterized queries and views		4.3, 4.12
○ Type system		4.4
○ Expressions		4.5
○ Graph projection		4.8
○ Catalog and schema		4.11
○ Language interoperability		4.13
○ Security model		4.14
○ Error handling		4.15

We note that while both documents are in favor of providing a form of linear statement composition, there is not yet agreement about supporting a strict, top-to-bottom evaluation order for clauses. Our proposal advocates for such a design in detail in Section 3.9.3, following the tradition of languages directly or indirectly influenced by ALGOL. We think this approach is appropriate for a composable language as it is much more in line with existing programming languages.

Both documents re-use keywords and syntax from the spectrum of available input languages. Neither document branches out into more advanced topics such as graph analytics and a procedural sub language at this point. We note that if we pursue a composable approach as suggested by this document, these could easily be explored in other languages and integrated into GQL as procedure calls via the language integration mechanisms discussed in Section 4.13.

1.4.2 Concordance with [ERF-038]

This section incorporates the tables of features from [ERF-038] and annotates them with the sections of this document that covers those features.
In these tables features or language constructs targeting CIWD are highlighted in yellow, features or language constructs targeting GQL1 are highlighted in cyan, and features or language constructs targeting beyond are shown without highlighting.

Table 1.4.2.1 Proposed Scope for GQL Features

Feature	Minimal	Regular	Advanced
Property Graph Data Model	Multigraph with multi-labeled vertices and edges, each with uniquely-named properties **[from SQL/PGQ]** *implied*	Bidirectional edges *not covered*	Transient/temporary properties, Path objects *not covered*
High-level execution model	Session model *not covered*	Transaction semantics *not covered*	
Tables and graphs	Queries: return graph or table Views: return graph Schema: graphs only	Views: return graph or table Schema: graphs or tables, Graph snapshots	Queries: return multiple graphs Views: return multiple graphs, Triggers
	Sections 3.7, 3.8, 4.8, 4.10, 4.11, 4.12		*not covered*
Schema management	Graph metadata (schema) object creation in SQL catalog and DS/ IS views of those objects *Sections 4.11*	Schema evolution: graph metadata alteration *not covered*	Graph metadata graph model and graph queries added to SQL/Schemata: treat Schemata as standing above SQL and GQL *not covered*
Type system features	Basic data types **[partially from SQL],** incl. graph element types *Sections 4.4.1, 4.4.2, 4.4.3*	Union types for property data types, Label inference for graph elements *Sections 4.4.4, 4.4.5*	Composite types (e.g. structs, nested data), Path type, Schema-typed graphs, Schema-typed tables **[partially from SQL]** *Sections 4.4.2, 4.4.3, 4.4.4, 4.6.5, 4.11*

Modularization	Define language modules *not covered*	Explicit module imports *not covered*	
Graph and table read operations	Pattern matching (fixed and advanced), projection, filtering, sorting, slicing **[from SQL/PGQ]** *Sections 4.6, 4.9*	Unnest array, Graph and table set operations, aggregation/distinct **[partially from SQL]** *Section 4.8, 4.9*	
Data updates	Create Vertex/Edge **[from Cypher]** *Section 4.7*	Update/Delete Vertex/Edge **[from Cypher]** *Section 4.7*	Merge Vertex/Edge, Aggregating properties **[from Cypher]** *Section 4.7*
Graph projection	Basic without grouping **[from G-CORE, Cypher]** *Section 4.8*	Graph grouping **[from G-CORE, Cypher]** *not covered*	Advanced graph grouping **[from G-CORE, Cypher]** *not covered*
Access control		SQL compatible rules *Section 4.14*	Explore RBAC, ABAC, and VBAC *not covered*
Language interoperation (SQL, SPARQL, GraphQL, ...)	Define a mapping between GQL and SQL types *Section 4.4*	Foreign language interface for queries that return tabular data but not containing graph elements *Section 4.13*	Access to schema objects backed by a different data model (e.g. RDF graphs), Additional query formats (GraphQL-like queries) *Sections 4.13, 4.5*
Graph analytics	Call-out to analytics procedures *Section 4.13*	Basic procedural language for orchestrating multiple graph analytics procedures *not covered*	Language-native graph analytics computation model, Custom/imperative traversal language/API *not covered*

Table 1.4.2.2 Proposed Scope for GQL Language Constructs

Feature	Minimal	Regular	Advanced
Query structure	Multi-part queries consisting of linear chains of query parts *Sections 4.3.3, 4.3.4*	Multi-part queries consisting of trees of query parts *Sections 3.2, 3.6, 4.10*	Cross-statement composition *not covered*
Subqueries	Nested, optional *Section 4.10.1*	Scalar, existential, set operations *Sections 4.10.2, 3.6, 4.3.4.2*	Subquery aliases *Section 4.3.3.2, 4.12.4*
Parameters	Top-level query parameters (global) *Section 4.3.2*	Per-view query arguments (local) *Sections 3.7, 4.12*	Label and property name parameterization *Section 4.4.4*

Value types	Scalar types (incl. NULL, integer types etc), Graph element types (vertices and edges) **[partially from SQL]** *Sections 4.4.1, 4.4.3*	Composite types (e.g. Lists, Structs, Maps, ...) **[partially from SQL]** *Section 4.4.2*	Graphs and tables as values *Section 4.4.3*
Domain-specific value types	Time-based data types **[from SQL]** *Section 4.4.1*	Geospatial data types *not covered*	
Equality and comparability	Equality and comparison for basic types **[partially from SQL]** *Section 4.4*	Sorting union types, equality and comparison of composite types **[partially from SQL]** *Sections 4.4, 4.9*	
Expressions	Scalar functions and operators **[partially from SQL]** *Section 4.5*	Functions and operators over composite types **[partially from SQL]** *Sections 4.5, 4.10.2*	Functions and operators over advanced types *not covered*
Aggregating functions	Standard set, potentially extended (e.g. times) **[from SQL]** *Section 4.5*	Calling aggregating functions over composite types *Section 4.10.2*	
Error-handling	Well-defined error codes *Section 4.15.3*	Trapping of errors *not covered*	Provenance tracking error values *Section 4.15.1*
User-defined functions	Simple functions (non-updating, non-aggregating) *Section 4.13.1*	Aggregating functions *Section 4.13.1*	User-defined patterns backed by custom traversals *not covered*
User-defined procedures	Read-only procedures *Sections 4.12, 4.13*	Updating procedures *Sections 4.12, 4.13*	Schema-introspecting and modifying procedures *not covered*

2 References

[CIP2015-06-24] S. Plantikow, N. Small, T. Lindaaker, "Calling Procedures", openCypher CIP2015-06-24 (draft), version #093b6e5, Mar 2017

[CIP2015-08-06] T. Lindaaker, "Date and Time", openCypher CIP2015-08-06, version #4ae4ba8, Apr 2018

[CIP2015-10-27] A. Taylor, "State visibility between clauses", openCypher CIP2015-10-27, version #7237f4d, Jan 2018

[CIP2016-01-26] S. Plantikow, "The MANDATORY MATCH clause", openCypher CIP2016-01-26, version #7237f4d, Jan 2018

[CIP2016-06-14] M. Rydberg, S. Plantikow, "Definitions for Comparability and Equality, and Orderability and Equivalence", openCypher CIP2016-06-14, version #7237f4d, Jan 2018

[CIP2016-06-22] P. Selmer, S. Plantikow, "Nested, updating, and chained subqueries", openCypher CIP2016-06-22 (draft), version #077fb18, May 2018

[CIP2017-02-07] T. Lindaaker, "Map Projection", openCypher CIP2017-02-07, version #d01aadd, Apr 2017

[CIP2017-03-29] T. Lindaaker, "Scalar Subqueries and List Subqueries", openCypher CIP2017-03-29 (draft), version #d108a8c, Oct 2018

[CIP2017-06-18] S. Plantikow, A. Taylor, P. Selmer, "Querying and constructing multiple graphs", openCypher CIP2017-06-18 (draft), version #c5b8e42, May 2018

[CIP2018-05-03] S. Plantikow, A. Taylor, P. Selmer, "Creating and managing graphs and views", openCypher CIP2018-05-03 (draft), version #10b146b, May 2018

[Cypher for Apache Spark] openCypher, "Cypher for Apache Spark", https://github.com/opencypher/cypher-for-apache-spark, Nov 2018

[Cypher Formal] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, M. Schuster, P. Selmer, A. Taylor, "Formal Semantics of the Language Cypher", https://arxiv.org/pdf/1802.09984.pdf, Mar 2018

[Cypher 9 Reference] openCypher, "Cypher Query Language Reference", http://s3.amazonaws.com/artifacts.opencypher.org/openCypher9.pdf, Mar 2018, Originally published Sep 2017

[DM32.2-2018-00128] A. Green, “Working towards a New Work Item for GQL, to complement SQL PGQ”, ANSI INCITS DM32.2-2018-0128, Jul 2018

[ERF-015] F. Zemke, “SQL/PGQ skeleton”, ISO/IEC SC32/WG3:ERF-015, or ANSI INCITS DM32.2-2018-00150, or ANSI INCITS sql-pg-2018-0023, Aug 2018

[ERF-035] F. Zemke, "Fixed graph patterns", ISO/IEC SC32/WG3:ERF-035, or ANSI INCITS DM32.2-2018-0153r1, or ANSI INCITS sql-pg-2018-0029r1, Sep 2018

[ERF-037] F. Zemke, “Relating GQL and SQL”, ISO/IEC SC32/WG3:ERF-037, or ANSI INCITS DM32.2-2018-00170r1, or ANSI INCITS sql-pg-2018-0028, Aug 2018

[ERF-038] S. Plantikow, “Proposed GQL Scope and Landscape”, ISO/IEC SC32/WG3:ERF-038 R1, or ANSI INCITS DM32.2-2018-00171r1, or ANSI INCITS sql-pg-2018-0033r3, Oct 2018

[ERF-042] J. Michels, “The pure property graph data model”, ISO/IEC JTC1SC32/WG3:ERF-042, or ANSI INCITS sql-pg-2018-0035, Sep 2018

[ERF-043] P. Furniss, “SQL/PG graph schema and join syntax mapping examples”, ISO/IEC SC32/WG3:ERF-043, or ANSI INCITS DM32.2-2018-0176, or ANSI INCITS sql-pg-2018-0036r2, Oct 2018

[ERF-044] A. Green, “Property Graph Data Model Concepts and Terms” ISO/IEC SC32/WG3:ERF-044, or ANSI INCITS DM32.2-2018-0177, or ANSI INCITS sql-pg-2018-0037, Oct 2018

[G-CORE] R. Angles, M. Arenas, P. Barcelo, P. Boncz, G. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. Sequeda, O. van Rest, H. Voigt, "G-CORE: A Core for Future Graph Query Languages", https://arxiv.org/pdf/1712.01550.pdf, Dec 2017

[GQL Manifesto] A. Green, "The GQL Manifesto", https://gql.today/, Mar 2018

[GraphQL] Facebook, "GraphQL Specification, June 2018 Edition", https://facebook.github.io/graphql/June2018/

[GXPath] L. Libkin, W. Martens, D. Vrgoc, "Querying graph databases with XPath", http://homepages.inf.ed.ac.uk/s1058408/data/gxp.pdf, Mar 2013

[ISO14977] “Information technology -- Syntactic metalanguage -- Extended BNF”, Dec 1996

[JSON] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format, RFC 8259", http://tools.ietf.org/html/rfc8259, Dec 2017

[JSONPath] S. Goessner, “JSONPath - XPath for JSON”[2], https://goessner.net/articles/JsonPath/, Aug 2007

[MORPHEUS] Neo4j, Inc., "Morpheus User Guide", https://neo4j.com/docs/morpheus-user-guide/preview/, Nov 2018

[PARAM] Cesar Galindo-Legaria, "Parameterized Queries and Nesting Equivalences", MSR-TR-2000-31, Apr 2000

[PGQL] O. van Rest, "PGQL 1.1 Specification",http://pgql-lang.org/spec/1.1/, Sep 2017

[RDF] W3C, List of specifications, https://www.w3.org/standards/techs/rdf#w3c_all, Feb 2015

[SHACL] W3C, “Shapes Constraint Language (SHACL)”, Jul 2017

[SPARQL 1.1] S. Harris, A. Seaborne, "SPARQL 1.1 Query Language", https://www.w3.org/TR/sparql11-query/, Mar 2013

[SQL Foundation:2016] Jim Melton (ed), “ISO International Standard (IS) Database Language SQL - Part 2: SQL/Foundation”, ISO/IEC 9075-2:2016

[SQL Framework:2016] Jim Melton (ed), “ISO International Standard (IS) Database Language SQL - Part 1: SQL/Framework”, ISO/IEC 9075-1:2016

[sql-pg-2018-0041] Mingxi, "TigerGraph Overview", ANSI INCITS SQL-PG-2018-0041, Oct 2018

[sql-pg-2018-0042] O. van Rest, “Initial GQL Proposal”, ANSI INCITS SQL-PG-2018-0042, Oct 2018

[sql-pg-2018-0036] P. Furniss, "SQL/PG graph schema and join syntax mapping examples", ANSI INCITS SQL-PG-2018-0036, Sep 2018

[sql-pg-2017-0002] Jan Michels, "SQL Standards Presentation", ANSI INCITS SQL-PG-2017-0002, Apr 2017

[Unicode] Unicode Consortium, "Unicode 11.0.0 Specification", https://www.unicode.org/versions/Unicode11.0.0/, Jun 2018

[XPath] J. Robie, M. Dyck, J. Spiegel, “XML Path Language (XPath) 3.1”, https://www.w3.org/TR/xpath-31/, Mar 2017

[YTZ-029R1] T. Lindaaker, “An overview of the recent history of Graph Query Languages”, WG3 YTZ-029R1, or ANSI INCITS DM32.2-2018-00085R1, May 2018

[YTZ-030r1] S. Plantikow, “Summary Chart of Cypher, PGQL, and G-CORE”, WG3 YTZ-030r1, or ANSI INCITS DM32.2-2018-00086r1, May 2018

[2] *See also extensions suggested by the alternative implementation by D.Parker, described at* https://github.com/danielaparker/jsoncons/blob/master/doc/ref/jsonpath/jsonpath.md

3 Discussion

3.1 Overview

The proposed design for GQL understands programs written in GQL as graph procedures that take named graphs, tables, and values as input parameters, produce and return a table, a graph, a value, or nothing as output and may additionally perform side-effects on schema objects in the catalog.

GQL procedures consist of a top-to-bottom sequence of statements (e.g. for pattern matching and projection) that are composed implicitly in the order given using linear statement composition, or combinators that compose one or more such nested procedures[3].

As an example, consider the following query that matches persons from the same country that travelled together, are of the same age, and at some point lived in the same city, and returns the number of such pairings per city and age group.

```
FROM friends
MATCH (a:Person)-[:TRAVELLED_TOGETHER]-(b:Person)
WHERE a.age = b.age
  AND a.country = $country
  AND b.country = $country
FROM census($country)
MATCH SHORTEST (a)-[:BORN_IN|MOVED_TO*]->(p)<-[:BORN_IN|MOVED_TO*]-(b)
MATCH (p)-[:LOCATED_IN]->(c:City)
RETURN a.age AS age, c.name AS city, count(*) AS pairs GROUP BY age
```

The query may be easily read by just following the sequence of statements (**`FROM ... MATCH ... RETURN ... GROUP BY ...`**): The query is parameterized with a single string parameter `$country`, Graphs are selected using **`FROM`**, pattern matching with **`MATCH`** and predicate filtering with **`WHERE`** are used to query the graph, and a final result is produced using tabular projection with **`RETURN`** and aggregation with **`GROUP BY`**.

[3] This is explained in detail in Section 4 and motivated in Section 3.9

3.2 Complex queries

More complex queries may be formed through the use of local definitions at the beginning of a query:

```
QUERY foafs($a VERTEX) {
  MATCH ($a)-[:FRIEND_OF]-(x)-[:FRIEND_OF]-(b)
  WHERE $a <> x AND x <> b AND $a <> b
  RETURN count(DISTINCT b) AS num_foafs
}
...
```

Such local definitions may then be used[4]:

```
...
MATCH (a)
CALL foafs(a) YIELD num_foafs
RETURN a.name, num_foafs
 ORDER BY num_foafs DESC
 LIMIT 100
```

Alternatively, instead of local definitions, a subquery may be used:

```
...
MATCH (a)
CALL {
  MATCH (a)-[:FRIEND_OF]-(x)-[:FRIEND_OF]-(b)
  WHERE a <> x AND x <> b AND a <> b
  RETURN count(DISTINCT b) AS num_foafs
}
RETURN a.name, num_foafs
 ORDER BY num_foafs DESC
 LIMIT 100
```

CALL not only executes inline nested procedures as shown above, but also executes named graph procedures. These may be either declared locally inside a GQL procedure (Section 4.3.3.2) or may also be defined in the catalog (see Section 4.11), either written in GQL or as a means for language integration (see Section 4.13).

[4] **CALL** is probably not the keyword that should eventually be used, but serves to illustrate the example. See the discussion in Section 4.12.2 about table operands and joins.

3.3 Type system

We propose that GQL shares basic data types with SQL but expands the type system towards support for schema-extending graphs that only adhere to a partial (possibly empty) schema such that there can be data in the graph that is not covered by the schema[5], together with support for heterogeneous data (e.g. via union types), graph data types such as vertices, edges, and paths, as well as additional data types and literal syntax for collections and nested data.

Support for nested data will make it possible to use GQL for provisioning applications with document data derived from complex graph structures, combining the power of pattern matching with map projection [CIP2017-02-07], something like JSONPath [JSONPath], and GraphQL-inspired views [GraphQL]. This is discussed in more detail in Section 4.4.

3.4 Pattern matching

Pattern matching is central to a graph query language and we expect GQL to include rich facilities for it, including support for shortest and cheapest path matching, path patterns, and configurable morphisms. This is discussed in more detail in Section 4.6.

3.5 Modifying data

GQL is an independent language and therefore needs to provide its own mechanisms for insertion, modification, and deletion of data. We propose to build on the linear flow of statements through linear statement composition by allowing multi-part procedures that allow both reading and modifying data operations. This is both intuitive and natural in that it follows the established reading order of programming languages, and simplifies returning data that reflects the performed updates:

```
FROM customers
MATCH (a:Person) WHERE NOT EXISTS { (a)-[:HAS]->(:Contract) }
WITH a, a.email AS email
DETACH DELETE a
WITH DISTINCT email
CALL {
  FROM marketing
  MATCH (c:Contact) WHERE c.email = email
  UPDATE marketing
  DETACH DELETE c
}
RETURN email
```

[5] This would mean that GQL would support graphs that are not “closed”, by the SHACL definition of the term [SHACL], whereas SQL would only support “closed” graphs.

3.6 Combinators

GQL also support combinators that allow combining the results from multiple nested subqueries. Typical examples of combinators are set operations between views:

```
CALL {
  FROM view1
  MATCH (a:Person)-[:KNOWS]-(b:Person)
  RETURN *
  UNION
  FROM view2
  MATCH (a:Person)-[:KNOWS]-(b:Person)
  RETURN *
}
RETURN DISTINCT a, b WHERE a<>b
```

Nesting may also be used to specify the order in which subqueries are to be combined:

```
{
   ...
   INTERSECT
   ...
}
UNION
{
   ...
   EXCEPT
   ...
}
```

Please refer to the Grammar for the detailed syntax for nesting of subqueries.

3.7 Views

The ability to derive views from base tables is a cornerstone of the success of relational databases. For GQL, we propose to follow in these footsteps by providing graph views via graph projection and construction. Views in GQL are simply the graphs or tables returned by named graph procedures. Since graph procedures are parameterized, views may be parameterized too. Furthermore, graph views may share vertices and edges from the same underlying graphs or tables, leading to natural support for updatable views:

```
QUERY sameCityFriends {
  MATCH (a)-[e1:LIVED_IN]->(c:City)<-[e2:LIVES_ID]-(b)
  WHERE EXISTS (a)-[:KNOWS]-(b) AND e1.year = e2.year
  CONSTRUCT
  MERGE (a), (b)
  CREATE (a)-[:SAME_CITY_FRIEND]-(b)
  RETURN GRAPH
}
FROM sameCityFriends
MATCH (a)-[:SAME_CITY_FRIEND]-(x)-[:SAME_CITY_FRIEND]-(b)
WHERE a <> x AND x <> b AND a <> b
RETURN a.name, count(b) AS num_same_city_friend_of_a_friend
```

3.8 Catalog

GQL may access and manage multiple persistent schema objects such as graphs, graph types, and named graph procedures using a catalog. The proposal suggests the addition of necessary DDL procedures for manipulating the content of the catalog:

```
CREATE GRAPH myGraph WITH SCHEMA social_graph {
  <query that returns initial content of myGraph>
}
```

3.9 Design principles

We propose that the design of GQL aims to create a new, and independent language that respects the following principles:

- **GQL is a property graph query language**
  GQL follows established industrial property graph query languages in both syntax and semantics by relying on pattern matching with visual ASCII line art and linear statement composition.

- **GQL is a composable language**
  GQL procedures are composed of nested subprocedures and sequences of statements.

- **GQL is a declarative language**
  GQL is focused on describing the intention of the user, not aspects of execution.

- **GQL is an intuitive language**
  GQL is providing a consistent, and natural approach for working with property graphs.

- **GQL is a compatible language**
  GQL aims to be semantically compatible with a profile of SQL, and follows a similar syntax tradition in its re-use of keywords, operator symbols, and expression syntax where sensible.

We have tried to follow these principles in this proposal.

3.10 Motivation

Here we expand on the motivation for some of the design choices of the presented proposal.

3.10.1 Property graph query languages

The core of a property graph query language is the matching of patterns over graphs. Previous languages, while rich in their own ways, do not have this feature as their central concept, nor even as a convenient concept. Integrating graph pattern matching into another language will force pattern matching to be at best a second class citizen of that language, with unnecessary ceremony or inconvenient syntax as the result. While putting pattern matching first has clear benefits to the users of a property graph query language, it is also a desire for graph database vendors to have a query language that is focused on the primary tasks of the property graph database system without the addition of many other features. By these accounts, it is clear that a dedicated property graph query language, GQL, is desirable.

The space of property graph query languages has been explored by vendors for a couple of years. openCypher [Cypher 9 Reference] and PGQL [PGQL] have emerged as the most promising. While these two languages have a lot in common, they also differ in subtle ways. A *standard* property graph query language, GQL, based on these two languages, is desirable.

3.10.2 A system of composable procedures and queries

One important objective in the design of GQL is to have a system that allows the composition of queries, or more generally *procedures*, over graphs. It can be helpful to think of a query (a graph procedure without side-effects) as a function that accepts parameters that can include zero or more tables and zero or more graphs, and produces an output that is either a table or a graph.

An interesting side effect of this compositional ability is that it allows the definition of graph procedures in languages other than GQL to be composed with and by GQL within the same framework. This ability will allow for GQL to focus primarily on being a solid, comprehensive language for *querying and maintaining* property graphs, as other languages can be used for solving other related problems, such as expressing graph computation or queries over tables. This means that GQL will remain a relatively small language, and will moreover keep the design process focused.

3.10.3 The benefit of linear statement composition

Many years of industrial-grade Cypher usage have demonstrated that a simple way to express *linear statement composition*[6] is beneficial. *Linear statement composition* can be thought of as the output of one query being fed as input to another query.

It is our experience that linearly composed statements are easier for users to read and understand than the use of nested subqueries and explicit lateral joins. This is due to the fact that it allows the reader of the query to think about the data flowing from the top of the query text towards the bottom, proceeding from the first statement in the linear list of statements to the following statement and so on. In queries that mix nested subqueries and joins to a high degree, the reader is forced to shift their attention back and forth within the query text. Lateral joins between nested subqueries result in the dependency of data being “hidden” behind the fact that nested subqueries are not necessarily being laterally joined, and a reader of the query needs to know to look for it. By contrast, linear statement composition allows for one simple rule in the language that lets the user know that data from the prior query flows into the next query.

There are further advantages to allowing linear statement composition:

- A linear sequence of statements is a well established programming language concept.[7]
- It can be treated algebraically by a query optimizer using techniques that are also suitable for the unnesting of subqueries in a composable query language (e.g. such as those described in [PARAM]),
- It provides a natural mechanism for performing a query on aggregated results without having to use nested subqueries or even incurring processing external to the query language (such as a round trip over a network).
- It allows the natural interleaving of reads and updates. A query concluding with update statements can be followed by another query that reads the updated state. The semantics of this is made clear by the separation of a reading phase and a writing phase in each of the queries in sequence.
- The simplest possible benefit from the interleaving of reading and writing is the ability to return a result from a query that updates data.

In order to be able to compose queries in this way, the projection of results from a query needs to be placed syntactically at the end of the query, rather than at the beginning as is done in SQL. The clause order of SQL forces composition to be done through nested subqueries.

The overwhelming majority of users of Cypher with prior experience using SQL have shared that this manner of composing queries is beneficial, and that this order of writing queries is substantially easier to read.

---

[6] This has also been called linear composition or sequential composition.
[7] It also works quite well practically with cut-and-paste by minimizing needed manual formatting/indenting and with line-based version control systems such as git by minimizing change sets.

4 Proposal

4.1 Initial project scope

The GQL language is a composable declarative database language for querying and managing property graphs that is intended to be useable both as an independent language as well as in conjunction with SQL or other languages.

Graph procedures are evaluated over named parameters, producing values, graphs, and tables (or other matrix data), as well as performing updates. GQL procedures are graph procedures written using the GQL language and are formed through the composition of statements.

As a declarative language, GQL is intended to allow multiple, different implementation strategies (e.g. by pattern matching using relational algebra, linear algebra, or automata theory) using different storage representations (e.g. index-free adjacency storage, classic tabular storage techniques, matrix representations) and targeting various classes of workloads (e.g. OLTP, OLAP).

The GQL language contains facilities for

- querying, modifying, and projecting property graphs,
- querying, and constructing property graph views,
- basic transformation of tabular data,
- composition of parameterized graph procedures and subqueries,
- managing named schema objects like property graphs, named queries, constraints, and property graph types,
- managing users, roles, and their access control rights and privileges,
- managing user-defined functions, procedures, and similar constructs such as user-defined aggregator functions,
- interaction with other languages and systems, and
- the query execution model, including error handling.

The GQL project, if established, may over time address a wider scope, as needed in order to enable GQL to be a useful and implementable language.

4.2 Definitions

Catalog
A *catalog* is directory of named *schema objects* (see Section 4.11).

Driving table
A *driving table*[8] is the left-hand input of linear statement composition (i.e. it corresponds to the left input table of a left lateral join between two tables).

Expression function
An *expression function* is a *function* that returns a scalar value and may occur in an expression context in GQL. This is detailed further in Section 4.10.2 on subquery expressions.

Graph, and **graph elements** (**vertices** and **edges)**
A *graph* in the remainder of this document refers to a property graph, consisting of labeled *vertices* or *nodes* and *edges* or *relationships*. *Vertices* and *edges* are collectively referred to as *graph elements* and have independent identity and *properties* (i.e. multiple vertices with the same *properties* may exist in a graph).

Graph function
A *graph function* in this document, refers to a *graph procedure* that is free of side effects, i.e. does not modify any persistent data.

Graph procedure
A *graph procedure* is an executable unit of code that may optionally have side effects.

GQL-agent
A *GQL-agent* is the client that invokes a graph procedure in GQL with a set of *parameters*.

GQL procedure, and **GQL function**
A *GQL procedure* is a *graph procedure* written using the GQL language. A *GQL function* is a *graph function* written using the GQL language.

Graph processor
A *graph processor* is a system capable of executing a *graph procedure* that was submitted by a *GQL-agent* togeher with *paramaters* and return a result to the *GQL-agent* in accordance with the rules and definitions of the GQL language.

Graph type
A *graph type* describes the *vertices* and *edges* that may occur in a graph of a given *graph type* in terms of their *labels*, *properties*, and basic topology.

[8] The term "driving table" originated in Cypher: Most of Cypher's top-level clauses take a table as input, generate a subtable for each row of the input, and concatenate all these subtables to produce an output (similar to the "flatMap" operator in functional programming). The input table to a clause is called a driving table because it "drives" the evaluation of the clause, especially in DML.

Graph type definition
A graph type definition is a local definition that associates an identifier with a *graph type* for further use inside its enclosing *GQL procedure*.

Label, and **label name**
A *label* is used to classify *graph elements*. It consists of an identifier, the *label name*.

Named graph procedure, **named graph function**, or **named query**
A *named graph procedure* is a *graph procedure* that has been installed into the *catalog*, or that is provided by the system, and can be invoked from other procedures. Similarly a *named graph function* is a named graph procedure that is a *graph function*, and a *named query* is a named graph function that is a *query*.

Named pattern
A *named pattern* is a local definition that associates an identifier with a pattern for further use inside its enclosing *GQL procedure*.

(Procedure) **parameter**
A *parameter* is a name that is bound to either a value, a table, a graph, or the name of a *label* or a *property*. Not all kinds of *parameters* may be used in all contexts.

Property, **property name**, and **property value**
A property is an attribute of a *graph element*. A *property* consists of an identifier - the *property name* - and a value - the *property value* - that may not simply contain graph element references. A single *graph element* cannot have two *properties* with the same *property name.*

Query, **modifying query**, and **catalog-modifying query**
A *query* is a *graph function* defined using GQL, synonymous to a *GQL function*.
A *modifying query* is a *GQL procedure* that is not a *GQL function*.
A *catalog-modifying query* is a *GQL procedure* that modifies the *catalog*.

Table, **empty table**, and **unit table**
A *table* is a multiset of rows each of which have the same struct type. An *empty table* is a *table* with zero columns and zero rows. A *unit table* is a *table* with zero columns and one row.

Temporary graph
A *temporary graph* is created using a local definition that associates an identifier with a new, empty, modifiable graph for further use during the execution of its enclosing *GQL procedure*.

Value, and **symbol** (value)
A *value* is any scalar or composite *value* that may be processed by GQL that is not a *schema object* or a *symbol.* A *symbol* is either a *label name* or a *property name*.

View definition, and **view**
A *view definition* is a *named graph function* that returns a dynamically computed result that is called a *view*.

Schema object
A *schema object* is any DDL object that may be stored in a *catalog* (see Section 4.11).

4.3 Language structure

4.3.1 GQL-request

A valid GQL-request consists of either

- a graph procedure, or
- a catalog-modifying graph procedure.

A GQL-request is sent by an GQL-agent together with a parameter set to a graph processor and is processed by

- creating an execution environment,
- invoking the graph procedure or the catalog-modifying procedure in that execution environment using the parameter set provided by the GQL-agent,
- returning the result of execution to the GQL-agent.

4.3.2 Parameters and parameter sets

A parameter has a name and is one of the following:

- a value (that may or may not contain references to graph elements)
- a schema object such as a table or a graph
  (either by reference or in some cases by value)
- a symbol (e.g. a label name or a property name)

A parameter set is a set of uniquely-named parameters.

4.3.3 Graph procedure

4.3.3.1 Definition

A GQL-procedure consists of

- a (possibly empty) sequence of local definitions,
- a procedure body.

The result of executing a graph procedure for a given parameter set is the result of executing the procedure body using the given local definitions.

4.3.3.2 Local definitions

Local definitions are declarations of named schema objects that are only visible to the statements in the procedure body of a graph procedure, such as

- view definitions via (optionally parameterized) named queries using **`QUERY`** `name(<parameters>) { <query> }` *(from Cypher),*
- named patterns using **`PATH`** `<name> AS <pattern>` *(from PGQL),*
- temporary graphs using **`GRAPH`** `name`,
- value parameters using **`PARAM`** `$name AS { <query> }`, or
- graph type definitions *(following SQL PG)*.

4.3.3.3 Procedure body

A procedure body consists of a non-empty list of composite statements separated by combinators.

The result of executing a procedure body is the result of executing the operator tree specified by the composite statements and combinators of the procedure body.

4.3.4 Statements

4.3.4.1 Statement

A statement is an elementary operation of the language (such as e.g. **`MATCH`** for pattern matching, or **`CONSTRUCT`** for graph construction) that may be executed with an argument driving table (from linear statement composition) and a given parameter set and produces one of:

- a table (or other matrix data), or
- a graph, or
- a value, or
- nothing (e.g. when updating data).

A statement may refer to parameters using `$a`, `$b`, ... syntax (from Cypher).

4.3.4.2 Combinators

A combinator is an operation in the language whose execution combines the results from executing multiple composite statement arguments (e.g. it is similar to **`UNION`**, **`INTERSECT`**).

It is proposed that GQL should follow similar precedence rules as SQL regarding such combinators.

4.3.4.3 Composite statement

A composite statement is either

- a list of statements that are implicitly combined using linear statement composition, or
- a nested procedure (a GQL procedure enclosed in { and }).

The result of executing a composite statement is

- if the composite statement is a list of statements, then the result is the result of executing those statements effectively in the order given and by composing their intermediary results using linear statement composition
  (similar to SQL's lateral join and Cypher's linear driving table composition)
- otherwise, the result is the result of executing the nested procedure.

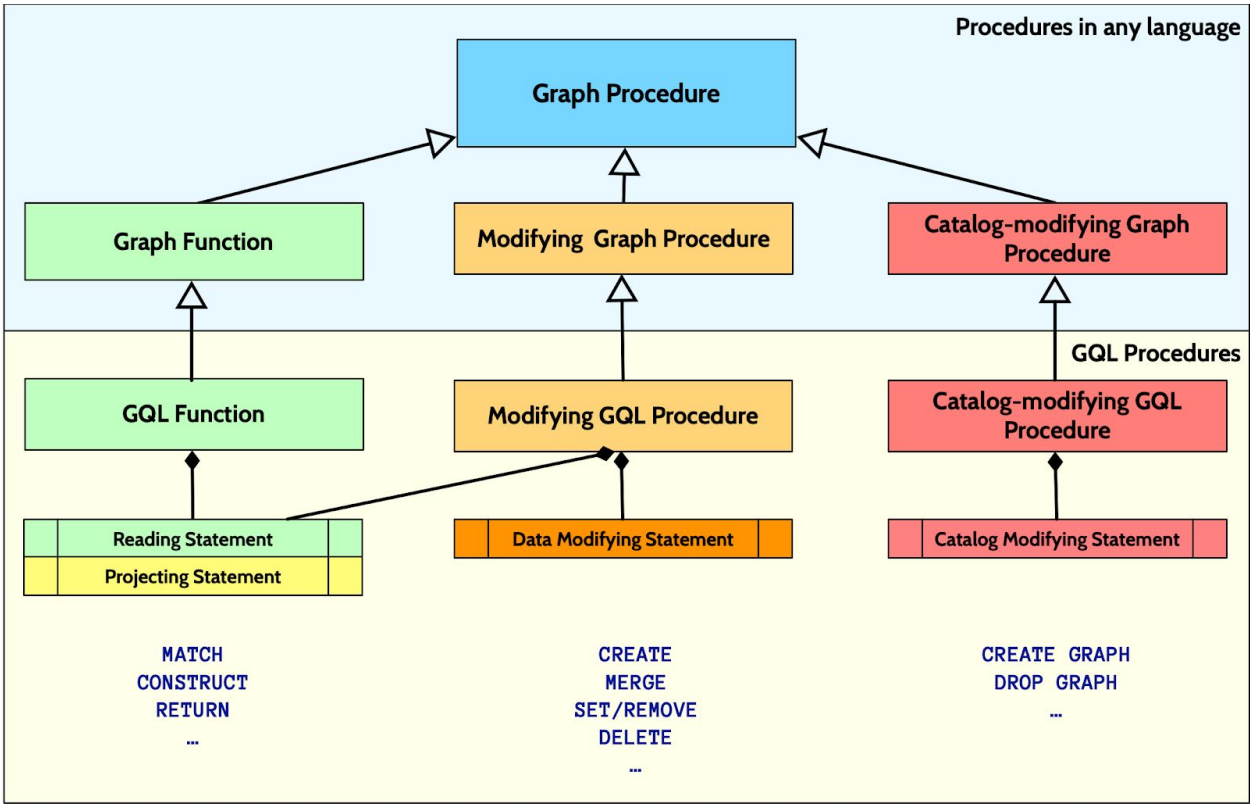

Fig. 4: Classification of graph procedures

4.3.5 Classification of graph procedures

A graph procedure may be classified according to its statements as one of the following:

- **Graph function**
  data reading and projecting statements
- **Modifying graph procedure**
  data reading, projecting, inserting, updating, and deleting statements
- **Catalog-modifying graph procedure**
  schema object inserting, updating, and modifying statements

We use the terms *GQL procedure*, *GQL function*, *modifying GQL procedure*, and *catalog-modifying GQL procedure* to indicate that a graph procedure is written in GQL (as opposed to be written in another language and exposed via language integration mechanisms).[9]

[Fig. 4] presents an overview of this classification terminology.

[9] For brevity we also use the terms *query* for GQL function, and *modifying query* for modifying GQL procedure.

4.4 Data types

We expect the GQL type system to share a large set of data types with the type system of SQL. GQL should support working with values of the data types discussed in the following subsections, including support for (i) comparison and equality, (ii) sorting and equivalence, and (iii) grouping of such values. Additional advanced type system features need to be provided for handling heterogenous data, graph elements, and graph schema information.

4.4.1 Scalar data types

- **`NUMERIC`** data types, comprising
    - fixed-width integer data types (both base 10 and base 2 precision)
    - fixed-width decimal data types (base 10 precision)
    - floating point data types (IEEE 754 32 bit, 64 bit, and 128 bit precision)
- **`STRING`** data type (restricted to the unicode character set)
- **`BOOLEAN`** data type
- temporal data types from SQL and [CIP2015-08-06]
- further data types from SQL as required

4.4.2 Collection data types

GQL should support the following collection data types:

- maps from arbitrary many string keys to values, each of the same data type with support for literal syntax such as
  `MAP { john: 12, sue: 13, billy: 6 }`
  (following JSON, Cypher)
- anonymous structs which are maps with a fixed set of typed fields and support for literal syntax such as
  `{ name: "GQL", type: "language", age: 0 }`
  (following JSON, Spark/Hive, Cypher)
- ordered sequences with duplicates (lists in Cypher, arrays in SQL) with support for literal syntax such as `[ 1, 2, 3 ]`
  (following SQL semantically, JSON, Spark/Hive, Cypher)
- unordered sequences with duplicates (multisets, bags) with support for literal syntax such as `BAG { 1, 2, 2, 3 }`
  (following SQL semantically)
- unordered sequences without duplicates (multisets, bags) with support for literal syntax such as `SET { 1, 2, 3 }`

4.4.3 Graph and graph element data types

- graph element reference types that reflect schema (e.g. the Person node type)
    - vertex (node) reference types
    - edge (relationship) reference types
- step type (values are vertex-edge-vertex reference triplets)
- path type (values may be understood as a list of alternating vertex and edge references or as a triplet of a start vertex reference, an end vertex reference, and a list of steps that connects them)
- graph types (tied to graph schema as discussed in Section 4.11, we envision both the use of static, pre-declared schema and open-world, dynamically-tracked schema, as well as mixed forms of partially static and partially dynamic schema)

4.4.4 Advanced data types

We may want to explore the following advanced type system features in the context of GQL, using and possibility adopting existing facilities of SQL where possible:

- union data types for expressing that a value may be one from a set of data types
- the null data type (which is the type of **`NULL`**)[10]
- recursive data type definitions for expressing nested documents,
  allowing to declare e.g. a "JSON" type as a
    - array of JSON documents, a map from strings to JSON documents, or a value
    - where a value is a number, a string, a boolean, or **`NULL`**.
- type aliases
- a symbol data type whose values are used for representing label or property names (e.g. for passing them as parameters)
- a data type for representing an arbitrary value (e.g. **`ANY`**, following Cypher and scala)
- row data types for handling tabular data
- user-defined data types
- user-defined enumeration data types
- user-defined domain data types

[10] This is introduced with the assumption that GQL will support union types. Union types are powerful enough to express nullability of a property at the type level, instead of having to use a **`NOT NULL`** constraint.

4.4.5 Type inference and checking

GQL should allow use of available information in type inference and type checking, including

- Use of schema information to infer the possible labels of graph elements
  (label inference, partially following Cypher 10)
- Use of data flow information (e.g. **`CASE`** with label test) to refine the possible label sets of a graph element
- Analyse the side-effects of updating statements to further constrain or relax type information

We believe it is also desirable to allow GQL execution using runtime type checking.

To realize both goals, GQL may be executed in two modes

- **strict mode**
  reject queries whose execution could fail according to static analysis
  (this mode might be more suitable for static compilation and application development).

- **dynamic mode**
  reject queries whose execution cannot succeed according to static analysis
  (this mode might be more suitable for interpreted execution and interactive exploration such as exploring a graph using a visualization tool).

The GQL specification may define different degrees of type inference that a conforming implementation must provide for each of these modes.

4.5 Expressions

GQL re-uses basic literal and expression syntax from SQL, possibly amended with the following additions/modifications:

- graph-type specific expressions
    - `source(e)`: getting the source vertex of an edge `e`
    - `target(e)`: getting the target vertex of an edge `e`
    - `handle(x)`: getting the implementation-defined handle (id) of a graph element
    - `inDegree(n)`: getting the number of incoming edges of a vertex `n`
    - `outDegree(n)`: getting the number of outgoing edges of a vertex `n`
    - `allDifferent(<elts>)`: for testing that the graph elements, denoted by `<elts>`, are non-overlapping *(following PGQL)*
    - operators and expression functions for working with paths (see Section 4.6.5)
- property access for graph elements and nested data
  `n.prop[2].otherProp`: static property access
  `n["dynamicPropertyName"]`: dynamic property access
  (following JSON, Hive, Cypher, PGQL, and many other languages and systems)

  It might be fruitful to explore extending property access with functionality that is similar to that of [JSONPath] or [XPath]. This would require introducing additional syntax, like `a(@.x<10), x[n:m], y[1,2]..prop`
- literal syntax for nested data
  ```
  {
    name: "GQL",
    parents: ["SQL/PG", "Cypher", "PGQL", "G-Core" ]
    age: 1
  }
  ```
  (following JSON, Hive, PGQL, Cypher)
- array indexing and slicing (e.g. `myList[4..5,8..8]`, or `myList[-1]`)
- map projections (see [CIP2017-02-07])
  ```
  <map-expr> { * -removed, existing: "new-value", extra: "value" }
  <map-expr> { one_included, another_included, extra: "value" }
  ```
  (following Cypher, inspired by GraphQL)

- additional aggregator functions
    - `times` - for e.g. multiplying probabilities along a path
    - `all_true` - for testing that a collection of booleans does not contain **`FALSE`**
    - `any_true` - for testing that a collection of booleans contains **`TRUE`**
    - `none_true` - for testing that a collection of booleans does not contain **`TRUE`**
    - `once_true` - for testing that a collection of booleans contains **`TRUE`** once
    - `alike` - which tests if all aggregated values are the same, equal value and are not **`NULL`**, and if this is true, returns that value, and otherwise returns **`NULL`**[11]
- aggregator functions for aggregating values into a collection using syntax such as e.g. `array_agg`, `bag_agg`, and `set_agg`
  (following SQL)
- syntax for computing aggregator functions over collections using syntax such as e.g. `count(`**`ELEMENTS OF`** `myArray)`

[11] this can be useful to discard conflicting values in data reconciliation

4.6 Statements for graph pattern matching

Pattern matching is central to GQL through being the means by which graphs are queried.

Pattern matching is undertaken using the **`MATCH`** statement

```
[ FROM, <graph> ], MATCH, <pattern>, { <comma>, <pattern> } ;
```

The first **`MATCH`** in a query is not allowed to omit **`FROM`**. If **`FROM`** is omitted, the matching is performed on the graph of the previous **`MATCH`**.

Pattern matching rests on a descriptive, visual syntax for graph patterns that is used for describing parts of a graph and is useful not only for pattern matching but also for modifying operations or selecting parts of a graph when specifying a constraint (not covered in this proposal).

A `<pattern>` specifies the shape of a path that is to be matched. Multiple patterns are separated by comma while individual patterns may be broken across multiple lines using whitespace. A `<pattern>` may optionally name a variable that a matching path is bound to. *(e.g. using Cypher's `<var>=<pattern>`, or PGQL's `<var>` **`AS`** `<pattern>`)*

The patterns in GQL are aligned with, and used the same syntax as, SQL PGQ (as described in [ERF-035]). In addition to the rigid patterns defined in [ERF-035], GQL supports, at a minimum, regular path queries through the use of grouping of patterns, alternation, optionality, Kleene star, and path macro definitions (as mentioned in Section 4.3.3.2 on local definitions). It is envisaged that the syntax for regular path patterns will be incorporated into SQL PGQ as well, as it will be developed by either the SQL PGQ project or the GQL project, or in collaboration between the projects.

Furthermore, GQL supports specifying the *cost* of the segments of a path in order to be able to match the *cheapest* paths. It is expected that this will also be incorporated into SQL PGQ, either by being developed by the SQL PGQ project or the GQL project, or in collaboration between the projects.

4.6.1 Modifiers to MATCH

Modifiers to **MATCH** specify the semantics of the result of this **MATCH** for all patterns:

- **OPTIONAL MATCH** - results are to be combined with the outer driving table in an left outer join
- **MANDATORY MATCH** - an error is to be raised if this **MATCH** does not match at least once
- **MATCH DIFFERENT** (**VERTICES**|**NODES**) - graph elements matched by the patterns of this **MATCH** should be isomorphic, i.e. that the same vertex may not be bound to more than one place in the pattern.
- **MATCH DIFFERENT** (**EDGES**|**RELATIONSHIPS**) - graph elements matched by the patterns of this **MATCH** should be edge isomorphic, i.e. not repeat the same edge in multiple places. This does allow the repetition of vertices.
- **MATCH UNCONSTRAINED** - graph elements matched by the patterns of this **MATCH** should be homomorphic, i.e. without restriction when vertices or edges may be repeated.

These modifiers may be combined as described by the following production:

```
[OPTIONAL|MANDATORY] MATCH
(UNCONSTRAINED|DIFFERENT (VERTICES|NODES|EDGES|RELATIONSHIPS))
```

4.6.2 Path pattern modifiers

Modifiers to path patterns specify which paths should be matched by a given path pattern (from all possible paths that would match the path pattern under homomorphism semantics):

- **SHORTEST** - only (one of) the shortest possible path(s), as measured by number of edges, should be matched by the path pattern. This may be non-deterministic.
- **TOP** <k> **SHORTEST** [**WITH TIES**] - only at most <*k*> of the shortest possible paths (of equal length), as measured by the number of edges, should be matched by the path pattern. This may be non-deterministic.
- **ALL SHORTEST** - only (all) the shortest possible paths (of equal length), as measured by the number of edges, should be matched by the path pattern.
- **CHEAPEST** - only (one of) the cheapest possible path(s), as measured by aggregation of the specified cost for each segment of the path, should be matched by the path pattern. This is may be non-deterministic.
- **TOP** <k> **CHEAPEST** [**WITH TIES**] - only at most <*k*> of the cheapest possible paths (of equal cost), as measured by aggregation of the specified cost for each segment of the path, should be matched by the path pattern. This may be non-deterministic.
- **ALL CHEAPEST** - only (all) the cheapest possible paths (of equal cost), as measured by aggregation of the specified cost for each segment of the path, should be matched by the path pattern.

- **REACHES** - only a single (arbitrary) possible path should be matched by the path pattern. This may be non-deterministic.
- **MAX** <k> - at most <*k*> possible paths should be matched by the path pattern. This may be non-deterministic.
- **ALL** - all possible paths should be matched by the path pattern.

4.6.3 Path eligibility modifiers

Path eligibility modifiers may be placed independently of after path pattern modifier in front of a pattern and may further limit possible paths that are considered to match the pattern. Semantically, they are applied before path pattern modifiers:

- **ACYCLIC** - specifies that a possible path should not revisit the same vertex.
- **SIMPLE** - specifies that a possible path should not revisit the same vertex, except allowing the first and the same vertex of the path to be the same.
- **TRAIL[S]** - specifies that a possible path should not revisit the same edge. Revisiting the same vertex is accepted.

4.6.4 Default modifiers

There seems to be no current consensus regarding a preferable default semantics for pattern matching: Some authors of G-Core have advocated for choosing non-deterministic shortest path semantics while other authors have rejected that notion. More importantly, in the industry we find both use of edge isomorphism (Cypher) and of shortest path (PGQL). We therefore think that in order to avoid confusion it is best to not mandate a default semantics for pattern matching and request that the user is explicit using the syntax like the modifier syntax outlined in this document. In particular, we are very skeptical of proposals that suggest to choose a non-deterministic default semantics for pattern matching.

We intend to explore this topic in greater detail in a future paper, along with a deeper discussion on the implications of combining certain modifiers.

4.6.5 Working with paths

Matched paths, held in a variable, can be operated on by expressions for:

- accessing the ordered sequence of *vertices* of a path
- accessing the ordered sequence of *edges* of a path
- accessing the ordered sequence of *steps* of a path
- accessing the *length* (measured in number of edges) of a path
- accessing the computed *cost* of a path

Path variables can be used in patterns for matching a pattern containing one or more paths matched by a separate pattern. This allows, for example, producing the concatenation of two paths (provided they share a vertex - otherwise the pattern will produce no matches).[12]

4.7 Statements for modifying graphs

GQL supports the following statements for modifying graphs:

- insertion (creation) of new vertices and edges using

  ```
  [UPDATE <graph>] INSERT <pattern>
  ```
  (following SQL, SPARQL)

- setting and removing graph element properties and labels

  ```
  [UPDATE <graph>] SET n.prop=<expr>
  [UPDATE <graph>] REMOVE n.prop=<expr>
  [UPDATE <graph>] SET n:NewLabel
  [UPDATE <graph>] REMOVE n:ExistingLabel
  [UPDATE <graph>] SET n=<expr>    /* replace all properties */
  ```
  (following Cypher and parts of G-Core graph construction syntax)

- deletion of existing graph elements

  ```
  [UPDATE <graph>] DELETE x        /* fails if vertex has edges */
  [UPDATE <graph>] DETACH DELETE n /* deletes vertex with edges */
  ```
  (following SQL, SPARQL, GSQL, Cypher)

- match-or-insert-with-optional-deduplication of vertices and edges using

  ```
  [UPDATE <graph>] MERGE ALL <pattern>        // no deduplication
  [UPDATE <graph>] MERGE [SAME] <pattern>     // deduplication
  ```

  *(Deduplication during insert has proven to be highly useful in practice. **MERGE** is proposed here to follow designs originally intended for Cypher 10. We have left a detailed presentation of its semantics for future discussion papers on modifying graphs)*

[12] It may also be useful to introduce a path concatenation operator and an operator for constructing paths manually. Such operators would potentially fail at runtime when trying to concatenate paths that do not coincide on their concatenated endpoints.

4.8 Statements for graph projection

GQL supports the following statements that project a graph:

- graph projection by construction that re-interprets the syntax for modifying statements to describe the content of the constructed graph:

  ```
  CONSTRUCT
  INSERT <pattern>          // add new vertices and edges
  MERGE <pattern>           // add existing and deduplicate elements
  MERGE GRAPH <graph>       // add all elements from <graph>
  DELETE <elements>         // delete elements
  DETACH DELETE <elements> // delete vertices with their edges
  ...
  RETURN [COPY OF] GRAPH    // return projected graph
  ```
  (following Cypher 10)

- graph combinators, e.g. for set operations between graphs:
  UNION [**ALL**], **INTERSECT**, **EXCEPT**
  (following G-Core, Cypher 10)

- shorthand variations of graph projection by construction
  - variable graph projection using

    RETURN GRAPH OF <variables>|<star>
    (following Cypher)

  - graph projection with graph grouping using

    [**ON** <graph>, ...] **CONSTRUCT** <pattern>
    (following G-Core, SPARQL)

4.9 Statements for transforming tables

GQL offers a limited set of statements for operating on driving tables:

- lateral tabular projection and selection over the driving table using **`WITH`**
  that supports filtering, sorting, grouping, and limiting
  *(inspired by SQL's **`SELECT`** and Cypher's **`WITH`**)*
- terminal tabular projection and selection over the driving table using **`RETURN`**
  that supports filtering, sorting, grouping, and limiting
  *(inspired by SQL's **`SELECT`** and Cypher's **`RETURN`**)*
- combinators between tables, e.g. for set operations:
  `UNION` [**`ALL`**], **`INTERSECT`**, **`EXCEPT`** *(following SQL)*
- collection unnesting using **`UNNEST`** *(combining SQL's `UNNEST` and Cypher's `UNWIND`)*

Cypher features an "implicit aggregation" where the use of an aggregator function in a projection triggers an inferencing rule for determining an implicit grouping key (all projected expressions that are not aggregator function calls). This simple form of aggregation has proven to be highly popular with users and we propose that a similar feature would be included in GQL.

For more advanced table transformation statements that are not directly needed for common use in GQL, the expectation is that a system would provide another language for table manipulation (most likely SQL) that integrates with GQL as named graph procedure
(see Section 4.11 *Language Interoperability*).

4.10 Nested procedures and queries

4.10.1 Nested procedures

A nested procedure is a GQL-procedure that is contained within an outer GQL-procedure and may access variables from the outer scope.GQL supports the following syntactic forms that have been inspired by [CIP2016-06-22] for calling nested table-valued procedures:

- direct nested procedure calls using
  ...
  CALL { <procedure> } [**YIELD** <non-empty variable list>]
  (following Cypher's **CALL***)*
  ***CALL** is a provisional keyword, see Section 4.12.2 Table operands and joins*
- left outer join using **OPTIONAL** { <procedure> }
  (following SPARQL, Cypher)
- error when not matching using **MANDATORY** { <procedure> }
  (following Cypher)

An inner <procedure> always explicitly specifies what is returned (e.g. table columns that correspond to variables, or a graph).

Syntactically, we suggest the use of { and } for enclosing inner queries in order no to overload and conflict with the use of (and) by patterns and expressions.

GQL also supports calling nested graph-valued procedures in the context of **FROM** in **MATCH**:

FROM { <procedure> } **MATCH** ...

A nested graph-valued procedure may not access variables from the outer scope.

The creation, modification, and access of temporary graphs may be allowed inside a nested procedure as long as neither the graph nor any graph elements from the graph are returned.

4.10.2 Subquery expressions

GQL supports the following subquery expressions for executing subqueries that produce values in an expression context:

- scalar subquery expressions using

  `( <subquery> )`
  (following SQL)

- list subquery expressions using

  `[ <subquery> ]`
  (following Cypher, and originally inspired by Python's list comprehensions)

- existential subquery expressions using

  `WHERE EXISTS` `{ <subquery> }`
  `WHERE EXISTS` `<pattern> (`**`IN`** `<graph>)`
  (following a mix of SQL and graph query languages)

A `<subquery>` is a limited form of `<procedure>`. In scalar subqueries, it is only allowed to return a single projected value or aggregation. In list subqueries, it is only allowed to return a sequence of projected values, i.e. a single column table
(This follows Cypher as documented in [CIP2017-03-29]).

4.11 Catalog and schema

A GQL-request is executed within a GQL-environment that provides a catalog of named schema objects, including

- **graphs**
  multigraph of multi-labeled vertices and edges with properties
- **graph types**
  Labels and their properties, valid vertex and edge types (label sets), as well as structural constraints of conforming graphs in terms of vertices and edge types in a way that is comparable to at least ER-models, and further check constraints.
  (as partially discussed in [sql-pg-2018-0036])
- **tables**
  multisets or sets of rows, all having the same number of typed columns, possibly supporting tables with no columns, not allowing multiple columns with the same name
- User-defined constructs such as types, functions, procedures
- Users and roles

All schema objects share the same namespace, i.e. it is not possible to have a graph and a named graph procedure with the same name.

GQL will provide the necessary DDL statements for creating, evolving, and reporting the structure of these objects, such as:

- **CREATE GRAPH** <name> [**WITH SCHEMA** <schema>] [**FROM** <subquery>]
- **CREATE QUERY** <name>(<params>) { ... }
- **CREATE SCHEMA** <schema-name> {
  ...
  }
- **ALTER** <schema-or-graph-name> ...
- **COPY** <schema-object> **AS** <new-schema-object-name>
  (e.g. for snapshotting a graph, copying a schema)
- **MOVE**|**RENAME** <schema-object> **TO** <new-schema-object-name>
- **ALIAS** <schema-object> **AS** <new-schema-object-name>
- **DROP** <name>
- **SHOW** <name>
  This reports the structure of the named schema object.
- ...

4.12 Views

4.12.1 Defining a view

A view is produced by a named graph function that returns either a graph or a table. The invocation of a function that returns a graph can be used as if it was a graph, and the invocation of a function that returns a table can be used as if it was a table.

Since graph procedures can accept parameters, views can be parameterized. It is important to note that graphs and tables are valid parameter types. This enables the powerful ability to apply the same view definition to different graphs, as long as they conform to the required graph type.

Views in GQL are created in the schema using something like a **`CREATE QUERY`**[13] statement:

```
CREATE QUERY <name> [<parameter list>] AS <subquery>
```

This follows the syntax of local query definitions (for declaring named subqueries in the context of another query), which takes the form of:

```
QUERY <name> [<parameter list>] AS <subquery>
```

It is also worth noting that view definitions are not limited to GQL functions, any other language that operates in the same system and uses the same type system for parameters and results could be used for defining a view:

Two illustrative examples would be:

- a view returning a graph:

```
CREATE QUERY atlantic_food_web AS {
  FROM fauna_graph
  MATCH (fish:Species)-[eat:EATS]->(prey)
  WHERE EXISTS {
    MATCH (fish)-[:LIVES_IN]->(habitat)
    WHERE habitat.name = "Atlantic Ocean"
  }
  RETURN GRAPH OF *
}
```

[13] A “query” is a GQL function, as per the definitions in Section 4.2

- a view that is parameterized with a graph (conforming to a specified graph type), and that is also returning a graph:

```
CREATE QUERY foaf($input SocialGraph) AS {
  FROM $input
  MATCH (a)-[:FRIEND]-()-[:FRIEND]-(b)
  CONSTRUCT (a)-[:FOAF]-(b)
}
```

4.12.2 Table operands and joins

GQL features means by which a projection of an existing table, whether it is a physical table, a view, or a table-valued subquery, can be brought into the driving table. This results in the new driving table being the cartesian product of the prior driving table and the projection of the table. This cartesian product is the basis for allowing simple joins to be performed.

Cypher uses the keyword **CALL** for the functionality that would be generalized for this purpose. **CALL** is probably not the right keyword for this feature though, but in order to express examples throughout this document, **CALL** has been used as a placeholder for the to-be-decided keyword.

4.12.3 Graph operands and pattern matching

Choosing a graph for querying is done through the `FROM` clause of `MATCH`. This is true whether that is a (base) graph, a view, or a parameterized query returning a graph.

- **FROM** `myGraph` **MATCH** … - match from the graph or view called *myGraph*
- **FROM** `myGraph()` **MATCH** … - same as above, providing an empty set of parameters to the procedure. For graph procedures that take no parameters, the parentheses may be omitted.
- **FROM** `myProc("alpha", $two)` **MATCH** … - match from the graph returned by the *myProc* graph procedure.

Formally the syntax is `[`**FROM** `<graph expr>]` **MATCH** … where `<graph expr>` can be either the name of a graph in the catalog, a parameter containing (a reference to) a graph, the invocation of a procedure that returns a graph, or a graph-valued subquery.

Invocations in the **FROM** clause of **MATCH** may only be passed arguments of a static nature, meaning literal value expressions and query parameters, but not variables from the surrounding query. This implies that graph subqueries may not be correlated. The reason for this restriction is that non-static arguments would imply querying a different graph per row in the driving table, the semantics of which can easily become complex to reason about. It is imaginable that this restriction might be lifted in a future version of GQL.

4.12.4 The relationship between subqueries and named local queries

A subquery is a syntactic shorthand for a named local query. This syntactic shorthand of a (table valued) subquery allows the direct use of variables from the outer scope. This can be thought of as these variables being implicitly declared parameters to the subquery, or as a linear statement composition (a lateral join). Inline subqueries have no means of declaring explicit parameters.

4.13 Language interoperability

4.13.1 Introduction

Interoperability with other languages is done simply by allowing graph procedures to be implemented in languages other than GQL and exposed to GQL, in the same way that GQL procedures are available to any graph procedure, written in GQL or another language.

Graph procedures, whatever the language of implementation, if composable in requests executed by a graph processor, populate a common namespace for that processor[14], and operate over a common type system. That means that procedures defined in other languages take the same types of parameters (centrally including graphs and tables) as graph procedures, and return either a table or a graph.

Other languages are also able to define custom expression functions that can be invoked in an expression context in GQL, and aggregator functions that are available as aggregators in GQL. In this proposal, custom expression functions and custom aggregator functions cannot be expressed using GQL. A custom expression function accepts zero or more parameters, in the same way that a procedure does, but returns a single value. A custom aggregator function accepts a stream of input values and produces a single output value.

The API for defining and exposing a procedure, expression function, or aggregator function is specific to each implementation language.

The set of supported implementation languages is defined by the implementing system. GQL does not mandate support for any specific other languages, only the interface for calling procedures defined in other languages, as well as the means for exposing procedures implemented in GQL to other languages.

[14] The common namespace bounded by a graph processor is the namespace also used by a user-visible catalog. Objects in the catalog are accessible to graph procedures

4.13.2 Integrating with SQL

- SQL (base) tables are available to GQL in the form of tables as catalog objects
- SQL views are available to GQL in the form of procedures that accept no arguments and return a table (this makes them indistinguishable from tables in GQL)
- GQL named queries that take no arguments are available to SQL
    - GQL named queries that return a graph are available to SQL PGQ as Graphs.
    - GQL named queries that return a table are available to SQL as Tables (technically as Views).
- GQL parameterized named queries are not available to SQL, since SQL does not have a way to pass parameters to views.

This allows passing data freely between GQL and SQL.

4.14 Security model

The security model of the GQL language deals with what kind of operations in the language are subject to permissions management, as well as what kinds of other access restrictions may be put in place.

4.14.1 Access to graphs and tables

These are the different access permissions that can be granted for a graph or a table:

- **Read** - for executing reading queries on that graph or table
    - Read permissions can be restricted for specific properties associated with particular label sets in graphs, or for particular columns in tables
- **Insert** - for adding new rows to a table
- **Insert Vertex** - for adding new vertices to a graph
- **Insert Edge** - for adding new edges to a graph
- **Delete** - for deleting rows from a table
- **Delete Vertex** - for deleting vertices from a graph
- **Delete Edge** - for deleting edges from a graph
- **Update** - for updating values of properties on vertices or edges, or fields of rows in tables

  Update permissions can be restricted for specific properties associated with particular labels sets in graphs, or for particular columns in tables.
- **Alter Schema** - for modifying the schema of the graph or table.

  Note that the schema is where permissions for access to individual properties or columns is granted. Permission to alter the schema thus implies permission to change those permissions. Permission to alter schema also implies full read and write access.

As can be seen in the list above it is possible to restrict access to properties of elements in the graph via **Read** and **Update** permissions for specific properties. Restricting access to the elements themselves is done by defining a view that only exposes the elements the user should be granted access to, and grant access to that view, while restricting access to the original graph itself.

For a graph procedure (either a GQL procedure, or otherwise), permission to **Execute** the procedure can be granted. Procedures execute with the permissions the procedure was installed with, not with the permissions of the user that executes it. Access to tables or graphs passed as parameters to a procedure is granted based on the permission of the context the parameter originates from. For the table or graph returned by the procedure, the procedure declarations may restrict the permissions granted to the receiver to a smaller set than what was granted to the procedure itself, but the receiver may never gain additional permissions beyond the permissions granted to the procedure.

The **Alter Schema** permission is never granted to a procedure, and may thus never be granted on a graph or table returned by a procedure. It may be feasible to lift this restriction for procedures that return nothing or only a value.

These kinds of access rules allow the creation of a setup where direct access to underlying data is restricted, but limited access can be provided through views (i.e. named graph procedures) for both reading and writing.

4.15 Error Handling

4.15.1 Error values

Error values can occur during the execution of a graph procedure and do not prevent the procedure from proceeding. The simplest (and only mandatory) error value is **`NULL`**. All error values behave like **`NULL`**, but a system with support for error values will provide additional details about why the value is an error value.

GQL provides means of converting an error value into a failure and terminating the execution of the query.

The level of support for error values is implementation defined. At a minimum an implementation must support the **`NULL`** value to represent any error value.
A sensible step up from that is to allow the immediate conversion of an error value into a failure to provide a specific error code for the error occurring at that location in the query, but not to be able to propagate the codes of prior error conditions that lead to this error, nor being able to hold error values (other than the **`NULL`** value) in variables.
The final step of support would be to have full traceability of how errors are caused by preceding errors, such as for example a property access producing an error due to it being evaluated on a vertex variable that held an error value which in turn held an error value because it originated from an **`OPTIONAL MATCH`** that did not match anything.
An implementation may even support different levels of errors in different execution scenarios or system configurations, such as only supporting full traceability in a debug execution mode.

4.15.2 Failures

A *failure* is when the execution of the query terminates and an error is returned instead of a result.

4.15.3 Error codes

The following is a codification of the error model that has been used for Cypher in Neo4j, there is still work to be done to relate this to other error code models, most notably SQL.

Error codes in GQL are textual and separated into two parts: a category and a code, separated by a period: *`<category>`*`.`*`<code>`*

The error code fully describes what the error is, there is never an accompanying message providing further description about what the error is.

There may however be parameters to the error specifying diagnostic details. If an error is caused by another preceding error, these details may contain the causing error. The error code completely specifies the parameters that may accompany that error code.

An example of such parameters could be for an error occuring when trying to access a property of a non-existing vertex:

- The name of the property being accessed
- (Optionally) Another error specifying why the vertex did not exist

Errors that occur at a specific place in the query should be accompanied by a textual position describing where in the query source text the error occurred.

The same category may not contain both transient and permanent errors.

A transient error is an error that occurred during a particular execution of the query, but would not necessarily occur if the same query is executed again. The same category of transient errors may not contain both data dependant and system dependant errors. A data dependent transient error is an error where the error might go away if the queried data was to change. A system dependent transient error is an error that depends on circumstances in the system that executes the query, such as for example the system rejecting the query due to it having reached its maximum processing capacity, or not being able to execute the query due to deadlocks with another query.

A permanent error indicates that something is wrong with the query. A syntax error is a simple example of a permanent error.

The category of errors determines whether the state of the query was rolled back, or if the whole transaction was rolled back. The same category may not contain errors with different rollback semantics.

The actual error codes (adhering to the rules stated above) are to be defined later.

5 Grammar

Here we provide a sketched outline of the high-level grammar of GQL, using ISO 14977 (Extended BNF) notation [ISO14977], and typesetting keywords in bold face.

```
<request> :=
  [<preamble>], ( <procedure> | <catalog procedure> ) ;

<procedure> := <local declarations>, <procedure body> ;

<local declarations> := { <local declaration> } ;

<procedure body> := <composite statement>,
                    { <combinator>, <composite statement> } ;

<composite statement> := <statement list> | <nested procedure> ;

<statement list>   := { <statement> }- ;

<nested procedure> := '{', <procedure>, '}' ;

<local declaration> := QUERY, <identifier>, [AS, '{', <procedure>, '}']
                     |  PATH, <identifier>, AS, <pattern>
                     |  GRAPH, <identifier>, [AS, '{', <procedure>, '}']
                     |  PARAM, <identifier>, AS, <expression>
                     |  ...
                     ;

<statement> := [FROM <identifier> | <named procedure call>],
                [OPTIONAL|MANDATORY] MATCH, <pattern>, [<WHERE>]
             |  CALL, <call arguments>, [ YIELD, <non-empty variable list> ]
             |  OPTIONAL, <call arguments>
             |  MANDATORY, <call arguments>
             |  WITH, <projection arguments>
             |  INSERT, <pattern>
             |  SET ...
             |  REMOVE ...
             |  [DETACH] DELETE, { <identifier> }-
             |  TRUNCATE, <identifier>
             |  RETURN, <projection arguments>
             |  ...
             ;

<where> := WHERE, <predicate> ;
```

```
<call arguments> := <nested procedure>
                  |  <named procedure call>
                  |  <table name> ;

<non-empty variable list> := <identifier> [AS <identifier>],
                             { ',', <identifier> [AS <identifier>] } ;

<named procedure call> :=
  <identifier>, '(',
    [ <expression> { ',', <expression> } ]
  ')' ;

<projection arguments> :=
  { <expression>, [AS, <identifier>] }-,
  [<where>]
  [<group by>],
  [<order by>],
  [<skip>],
  [<limit>] ;

<catalog procedure> := <catalog statement list> ;

<catalog statement list> := { <catalog statement> }- ;

<catalog statement> :=
    CREATE, QUERY, <identifier>, [AS, '{', <procedure>, '}']
 |  CREATE, PATH, <identifier>, [AS, '{', <pattern>, '}']
 |  CREATE, SCHEMA, <schema>
 |  CREATE, GRAPH, <identifier>, [AS, '{', <procedure>, '}']
 |  ALIAS, <identifier>, TO, <identifier>
 |  DROP, <identifier>,
 |  RENAME, <identifier>, TO, <identifier>
 ;
```

5.1 A note on `<preamble>` of a GQL request

A *GQL-preamble* is an optional construct that allows for the passing of settings that may influence the way in which the *GQL-request* is executed (e.g. query planner options) using syntax such as

```
[PROFILE|EXPLAIN] GQL option1=valueA, option2=valueB, ...
```

GQL may standardize some options; e.g. for specifying the expected minimal version of GQL (`version=202x`). The same option name may be repeated multiple times, depending on the meaning of that option.

The keywords **`PROFILE`** or **`EXPLAIN`** may be included in the preamble. If **`PROFILE`** is included in the preamble the query execution environment is instructed to record and report profiling information during the execution of the query. If **`EXPLAIN`** is included in the preamble the query execution environment is instructed to not execute the statement, but instead report a description of how it would execute the statement, a query plan. The format of the query plan reported if **`EXPLAIN`** is demanded, and the format of the profiling information reported if **`PROFILE`** is demanded are both implementation defined.

6 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 and like all such contributions is:

- End of Paper -