What is the database language GQL?

A new standard for a property graph database language, ISO/IEC 39075 *Information technology — Database languages — GQL*, has just been published[1]. But what is GQL and what is in the standard?

This new standard was developed by the international standards committee, ISO/IEC JTC1 SC32 WG3 Database Languages. Ignoring the standards hierarchy acronyms, SC32 WG3 is the international committee that is also responsible for developing and enhancing the SQL database language. The participants (individual experts) in SC32 WG3 are delegated by the standards processes in various countries around the world.

When the standards committees started discussing a new standard for a property graph database language, we chose the name “GQL.” GQL might be an acronym for Graph Query Language, although the name was primarily chosen to parallel the SQL database language.

The initial version of SQL international standard (ISO/IEC 9075 - Database languages — SQL) was published in 1987. Since then, the SQL standard has been revised and has incorporated a variety of new technologies[2]. In 2019, a new project was approved to produce a parallel standard focusing on property graph databases, the Database Language GQL. This is significant both for the standards body and for the industry, insofar as it is the first time in more than 35 years that ISO will be releasing a new database query language.

Property graph databases came to recent prominence in the big-data, NoSQL technology spaces. A distinguishing feature of property graph data is that relationships are created in the data so that a user can write declarative queries without having to specify the relationships. Unlike the relational model that structures data into tables, the property graph model structures data inside of the database as a graph[3]. This enables new kinds of pattern-based analysis, and also the flexibility to add new kinds of data with minimal overhead. Real-world uses range from digital twin to anti money-laundering to drug discovery to supply-chain analysis, knowledge graphs for GenAI, and far more. The combined promise of the new model & language, and significant & growing use in the field, were in large part the basis for a new ISO sibling language to SQL.

Property graph databases store and retrieve nodes (vertexes) and edges (relationships).
The declarative language specified by the GQL standard is influenced both by existing property graph database products and by the SQL standard.

The GQL standard is a full database language supporting creating, reading, updating, and modifying property graph data. The property graph data can either be schema-free or constrained by a full Graph Type – a property graph schema.

---

[1] The GQL standard was officially published 2024-04-11.
[2] 11 parts of ISO/IEC 9075 - *Database Languages – SQL* were published in June, 2023.
[3] Graph theory - Wikipedia

The GQL standard specifies a rich variety of data types to support character and byte string, fixed point numeric, and floating point numeric, as well as native nested data. The Graph Pattern Matching (GPM) language used in GQL queries is very powerful, allowing a GQL user to write relatively simple queries for doing complex data analysis.

GQL is a database language, not just a graph query language. Like SQL it defines a runtime environment that is initialized from a persistent and extensible catalog. The catalog enumerates stored data objects that are accessed via authenticated sessions using transactional units of work. It supports the insertion, updating, deletion and reading of property graphs which are created and referenced by entries in the catalog. The contents of these graphs can be unconstrained, or may be prescribed by administrator-defined data models called graph types. These graph types are specified as part of what GQL (following SQL) calls a "GQL-schema": a specialized container in the catalog for metadata and data definitions, which also enable the rudiments of a security model, by defining the ownership of data by an authenticated principal (user).

GQL subsumes many features of existing commercial graph query languages, such as openCypher (originally developed by Neo4j Inc.), GSQL (Tigergraph Inc.), and PGQL (Oracle Inc.), as well as LDBC's research language, G-Core. It also shares a Cypher-derived graph pattern-matching sub-language (informally called GPML[4]) with SQL/PGQ[5], which is a read-only extension to SQL allowing tabular data accessed through a specialized view to be treated as a property graph. GPML matches a pattern graph against a data graph, returning a tabular representation of a subgraph of the data graph.

Property Graph Use Cases

Property graphs are used by both private and public financial institutions, where there is a desire (and sometimes a legal requirement) to identify potential crimes such as money laundering. For example, a sequence of money transfers from person A to person B to person C to person A might indicate money laundering. Likewise, with assessing portfolio risk & lender and borrower solvency, anti-fraud, customer 360, and a variety of other use cases.

Property graphs are also used to model utility infrastructure such as water distribution, electrical distribution, and telecommunications networks. Typical queries here include identifying the potential impact and benefits of adding new loads or distributed power generation such as wind or solar at particular points in the distribution network, and designing, maintaining, troubleshooting, and security the network.

[4] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, et al. Graph pattern matching in GQL and SQL/PGQ. In *Proceedings of the 2022 International*
Conference on Management of Data, pages 2246–2258, 2022.

[5] ISO/IEC 9075-16:2023 Information technology – Database languages SQL – Part 16: Property Graph Queries (SQL/PGQ). Standard, International Organization for Standardization, Geneva, CH, June 2023.

Additional use cases include (but are not limited to) social networks, bill of materials, access control, product recommendations, citation networks, routing, influencer detection, protein interaction networks, impact analysis, and generative AI.

GQL Examples

The following examples provide a brief introduction to GQL capabilities using examples:

- Queries and Graph Pattern Matching
- Add, Modify, and Delete data
- Transactions
- Schema-free and Fixed-schema

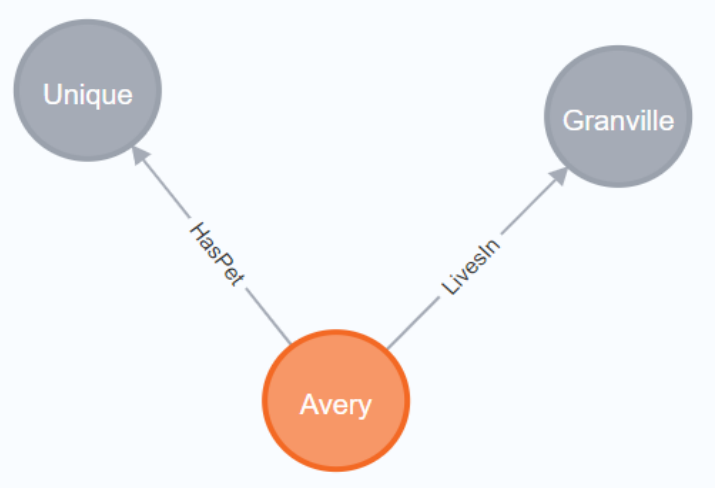
Queries and Graph Pattern Matching

GQL queries use a rich Graph Pattern Matching (GPM) language. The following example finds all nodes with a one-hop relationship to a node with a `firstname` of 'Avery':

```
MATCH (a {firstname: 'Avery'})-[b]->(c)
RETURN a, b, c
```

Note that the variables *a*, *b*, and *c* are defined in the MATCH statement and can be referenced later in the query, in this case the RETURN statement. The query writer does not have to know the relationships between 'Avery' and the other nodes.

The GQL standard does not specify how the returned data is displayed to the user. One way would be to display the data as a graph:



A graph visualization tool could show the relationship between the data retrieved and expand to show additional details as needed.

Another way of representing the data is as text. Note that this version of the query returns less data so that the example fits in the page width.

```
MATCH (a {firstname: 'Avery'})-[b]->(c)
RETURN a.firstname, b, c.name
```

```
+-----------------------------------------------------------+
| a.firstname | b                                 | c.name      |
+-----------------------------------------------------------+
| "Avery"     | [:LivesIn {since: 1980-07-15}] | "Granville" |
| "Avery"     | [:HasPet]                         | "Unique"    |
+-----------------------------------------------------------+
```

GQL GPM supports more complex queries such as quantified path patterns. For example:

```
/* Simple QPP */
MATCH ((a)-[r]->(b)){1,5}
RETURN a, r, b
```

This example will find paths where one node knows another node, up to five hops long. More complex quantified path patterns are also possible.

Add, Modify, and Delete

The GQL data in the previous examples was created using the INSERT statement. For example:

```
/* Insert one node */
INSERT (:Pet {name: 'Unique', pettype: 'Dog'})
```

In this example, “Pet” is a label and “name” and “pettype” are properties. Labels are identifiers that are either present or not present. Properties are pairs of names and values. Both nodes (vertexes) and edges (relationships) can have labels and properties.

Nodes are enclosed in parenthesis while edges are enclosed in square brackets.

GQL insert statements can use complex graph patterns. The following statement inserts two nodes and an edge between the nodes.

```
/* Insert two nodes and an edge */
INSERT (:Person {firstname: 'Avery'
                ,lastname: 'Stare'
                ,joined: date("2022-08-23")})
      -[:LivesIn {since: date("2022-07-15")}]->
       (:City {name: 'Granville'
              ,state: 'OH'
              ,country: 'USA'})
```

Inserts can operate on the results of a MATCH statement:

```
/* Create an edge between Avery and their dog. */
MATCH (a {firstname: 'Avery'})
     ,(d {name: 'Unique'})
INSERT (a)-[:HasPet]->(d)
```

In this example, “a” and “d” are aliases. They are defined in the MATCH clause and survive until the end of the GQL statement. The result of the MATCH is the cartesian product of the nodes returned by the two node expressions. Because each node expression returns only one node, the INSERT will insert only one edge.

GQL data is modified by identifying the nodes or edges to be updated and then setting or removing properties. For example:

```
MATCH (d:Pet) where d.name="Unique"
SET d.weight_kg=26

MATCH (e {lastname: 'Stare'})
REMOVE e.joined
```

GQL data is deleted by identifying nodes, detaching them to delete relationships, then deleting the nodes. For example:

```
/* Delete Avery and related nodes */
MATCH (a {firstname: 'Avery'})-[b]->(c)
DETACH DELETE a, c
```

Transactions

GQL supports serializable transactions and allows for additional implementation-defined transaction modes. Transactions are initiated with an explicit or implicit START TRANSACTION statement and terminated with either a COMMIT or ROLLBACK. A GQL implementation may also provide automatic transaction starts and commits.

Schema-free and Fixed-schema Graphs

GQL supports both schema-free graphs with no restrictions on the data entered and fixed-schema graphs where a graph can be created with a graph type that specifies the restrictions on the types of nodes and relationships that can exist in a graph created using that graph type.

A schema-free graph[6]will accept any data that is inserted. This allows for quick startup but leaves the control of the data with the application developer(s) and/or users.

The GQL standard also supports the ability to create a graph type. A graph type is a template that restricts the contents of a graph by fully specifying the types of nodes and edges that can be contained in a fixed-schema graph created from a particular graph type. A fixed-schema graph can contain only the node types and edge types specified in the graph type.

An administrator can create a graph inside a GQL-schema, inside a hierarchy of catalog directories, based on a graph type. Multiple graphs can be created based on a single graph type.

The following graph type defines two node types and two edge types.

```
/* Examples adapted from Alastair Green 2024-03-26 */

CREATE GRAPH TYPE /MyFolder/control/fraud_TYPE // DDL
      (customer:Customer => {id::STRING, name::STRING}),
      (account:Account => {no::STRING, acct_type::STRING }),
```

[6] Or “graph of the open graph type.”

```
      (customer)-[:HOLDS]->(account),
      (account)-[:TRANSFER {amount::INTEGER}]->(account)
```

Creating a graph using a graph type will constrain the contents of the graph to the node types and edge types described in the graph type.

```
CREATE GRAPH /MyFolder/control/fraud  /* graph name is "fraud" */
  TYPED /MyFolder/control/fraud_TYPE  /* graph type is fraud_TYPE */
```

When a data modification statement completes, the contents must fit the restrictions specified by the graph type. If the data does not fit the graph type restrictions, the statement results in an exception and the transaction is rolled back.

```
USE GRAPH fraud
INSERT // Data Manipulation Language
      (d:Customer {id: 'AB23', name: 'Doe'}),
      (r:Customer {id: 'CH45', name: 'Reiss'}),
      (a1:Account {no: '25673890', type: 'C'}),
      (a2:Account {no: '05663981', type: 'C'}),
      (d)-[:HOLDS]->(a1),
      (r)-[:HOLDS]->(a2),
      (a1)-[:TRANSFER {amount: 5000}]->(a2)


USE GRAPH fraud
MATCH // Data Query Language
      (c1:Customer)-[:HOLDS]->(a1:Account)
      -[t:TRANSFER]->
      (a2:Account)<-[:HOLDS]-(c2:Customer)
RETURN c1.name, a1.no, t.amount, c2.name, a2.no

/*
      'Doe', '25673890', 5000, 'Reiss', '05663981'
      1 row returned
*/
```

How do nodes and edges compare to tables?

In an SQL database, data is stored as records in tables where each table has a fixed shape. Relationships between tables can be enforced with constraints, but the person (or program) who writes the query has to understand the relationships between tables.

In a property graph database, the level of abstraction is raised, allowing whole sets of tables to be treated as a unit ("data product").

Nodes with a particular label (or sets of labels) are approximately equivalent to a table. However, a MATCH query can access nodes with particular characteristics regardless of the label(s).

Edges can be inserted to describe the relationships between nodes. The person (or program) who writes the query can query relationships without having to know what relationships exist.

Summary

This document provides a brief introduction to the capabilities of the GQL database language. The full GQL language includes support for additional capabilities that are beyond the scope of this document. The GQL database language is designed to allow for future growth, by being ready to incorporate additional forms of data in the catalog and by having been prepared for returning non-tabular results (e.g., graphs).

Author:
Keith W. Hare, JTC 1/SC 32/WG 3 Database Languages