ORACLE

The SQL/PGQ Standard: SQL support for property graphs

Oskar van Rest

Product Development – Oracle Property Graph

18th LDBC TUC Meeting, 30 August 2024

These slides are adopted from the same-titled presentation at the Knowledge Graph Conference 2024 by Jan Michels (chair of INCITS Data Management (DM32), and the DM32 SQL/PGQ Expert Group)

Agenda

1. What is the SQL Standard?
2. What is SQL/PGQ?
3. Capabilities of SQL/PGQ
4. Possible future extensions

What is the SQL Standard? (SELECT … FROM … WHERE…)

Well-known de jure database language standard

- ISO standard (ISO/IEC 9075) developed collaboratively by a number of national bodies (USA, UK, Germany, Japan, etc.)
- National body standards identical to the ISO standard
- 11 parts:
  - Framework, Foundation, Schemata (specify “core” functionality; e.g., DDL, DML, etc.)
  - CLI, PSM, OLB, JRT, MED, XML, MDA, PGQ

Mature standard, but still evolving

- Initial version published in 1986 (US) and 1987 (ISO)
- Several revisions since: 1989/92/99, 2003/08/11/16
- Most recent: SQL:2023

Many implementations – with varying degrees of conformance

Large number of applications

Why integrate with SQL?

Data is already stored in relational databases

- Tables model vertices and edges
- Table columns model properties

SQL has powerful (analytical) functionality

- No need to duplicate functionality in a stand-alone property graph database
- E.g., GROUP BY, row pattern matching, window/analytics functions, etc.
- E.g., security model, transactions, manageability, metadata

Easily join graph data with relational (non-graph) data

- No need to ship data from graph system to relational system or vice versa

Graph definition in SQL dictionary (along with SQL schema definition)

What is SQL/PGQ?

Part of ANSI/ISO SQL standard: ISO/IEC 9075-16 (SQL/PGQ – Property Graph Queries)

Property graph - first class database object

- View-like object
- Created using DDL statements based on existing relational tables
  - Tables hold data representing vertices & edges
  - No restriction on the number of vertex and edge tables in a given graph
  - No restriction on the number of graphs in a database
- Queried using new GRAPH_TABLE operator

Primarily aimed at supporting graph querying over existing schemas

Data Model + DDL

Graph Pattern Matching + SQL query syntax

Integrates well with the remainder of SQL

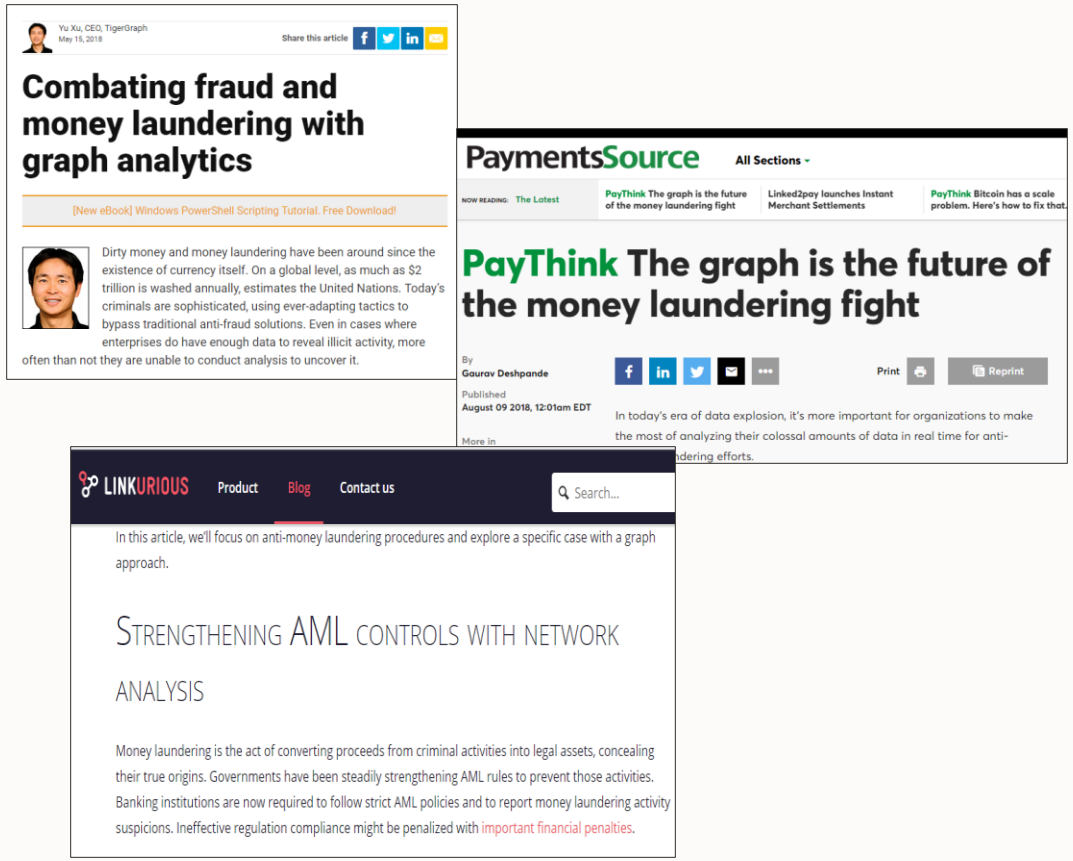
Graph Application Example: Anti-Money Laundering (AML)

Anti-Money Laundering (AML)

- A big application in financial domain
- Detect and report suspicious activity including offenses to money laundering

Graph is very useful in building AML solutions

- Money laundering activities are spread across many transactions between multiple entities
- Need to gather up actives that look suspicious individually and analyze inter-connections among those



Sample Graph Data – Underlying Tables

accounts

AID
10
20
30
40
50

customers

CID	NAME	CITY
100	Joe	San Jose
200	Jane	Santa Clara
300	Jeremy	San Francisco
400	Jessica	Redwood Shores
500	Fletcher	San Jose

owns

OID	CID	AID	SINCE
110	100	10	1/1/2019
220	200	20	2/2/2019
330	300	30	3/3/2019
440	400	40	4/4/2019
550	500	50	5/5/2019
510	500	10	1/1/2019

transfers

TID	FROM_ID	TO_ID	WHEN	AMOUNT
102001	10	20	1/1/2020	5000
103001	10	30	1/1/2020	15000
104001	10	40	1/1/2020	20000
105001	10	50	1/1/2020	25000
304001	30	40	1/2/2020	11000
305001	30	50	1/2/2020	4000
403001	40	30	1/3/2020	15000
305002	30	50	1/3/2020	14000

Property Graph Definition (1)

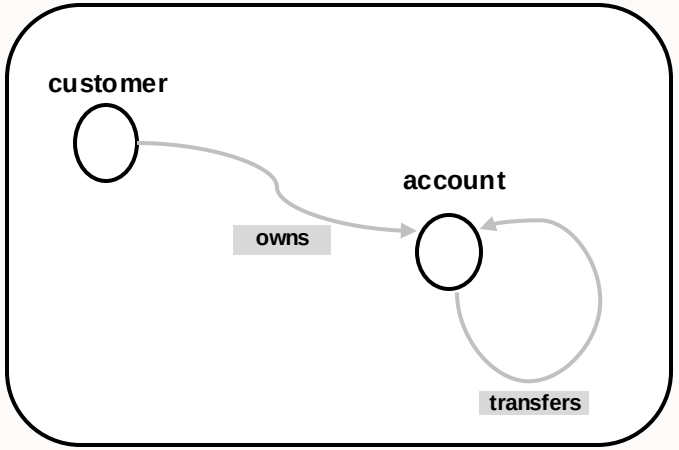
```
CREATE PROPERTY GRAPH aml
   VERTEX TABLES ( accounts AS account
                 , customers
                     LABEL customer PROPERTIES ( cid, name, city ))
       EDGE TABLES ( owns SOURCE customers DESTINATION account
                          PROPERTIES ( since )
                   , transfers
                      SOURCE KEY ( from_id ) REFERENCES account ( aid )
                      DESTINATION KEY (to_id) REFERENCES account ( aid )
                      LABEL transfers PROPERTIES ( when, amount) )
```

Defaults apply for label and all properties.

Explicit label and properties options for customer

Columns **when** and **amount** are exposed as properties. Columns tid, from_id, and to_id are **not**.

Vertex and edge labels



Property Graph Definition (2)

Existing tables (or views): customers, accounts, owns, transfers

User can specify options for

- Labels (1 or more per vertex/edge table)
- Properties (0 or more per label), can rename properties
- Keys (single or multi-column key)

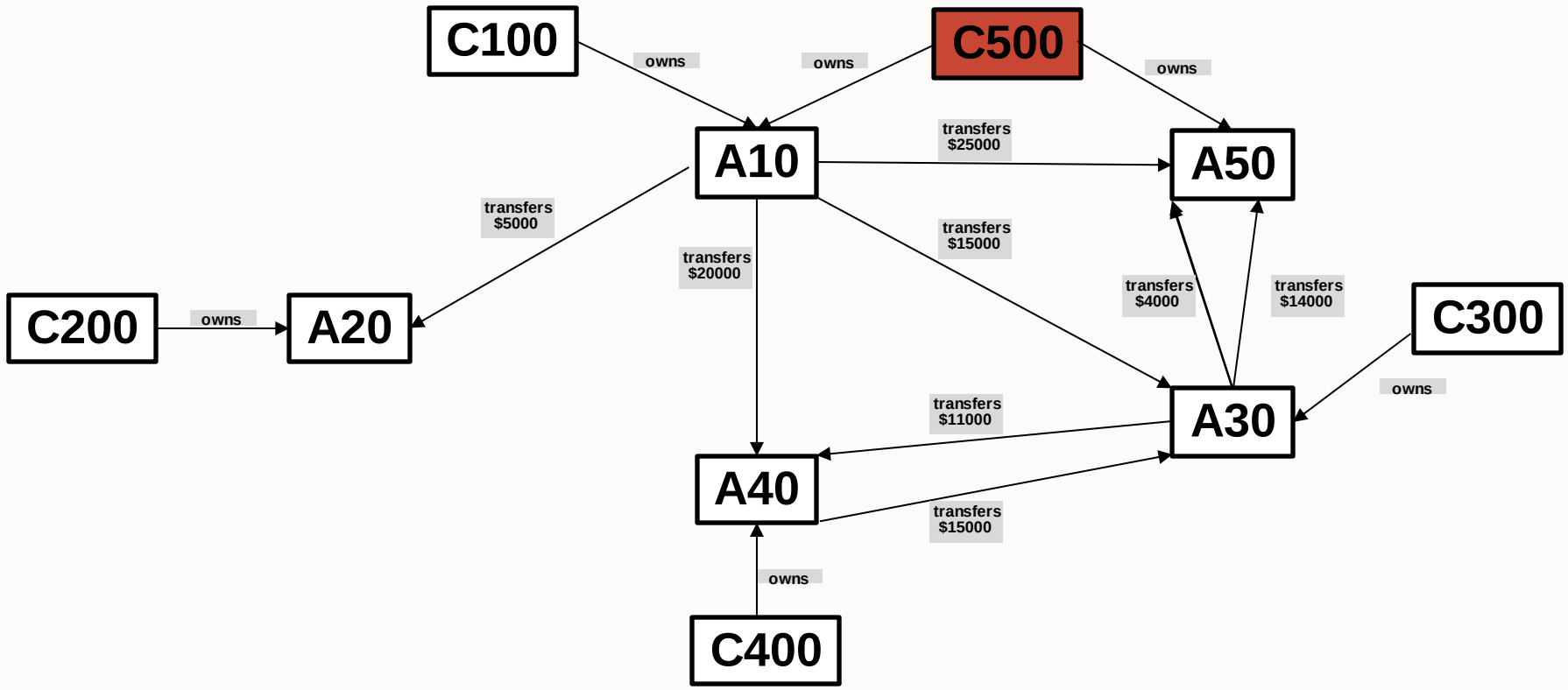
If not specified, defaults apply:

- Single label defaults to table name/alias
- All (non-hidden) columns are exposed as properties for a given label
- Keys are inferred from primary/foreign keys of underlying tables.
- PK-FK determines connection between vertices via edges (e.g., customer –[owns]-> account)

User can mix and match within a single PG definition:

- Explicit options, and
- Implicit defaults

Sample Graph Data



Querying PGs – Example 1

Retrieve the info of all customers who got more than $10,000 from customer 100.

New operator applied to graph named aml, returns table

```
SELECT gt.cid, gt.name, gt.city, gt.amount
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]->
        ( IS account ) <-[ IS owns ]- ( c2 IS customer )
  WHERE c1.cid = 100
    AND t1.amount > 10000
  COLUMNS ( c2.cid
          , c2.name
          , c2.city
          , t1.amount )
 ) gt
```

Edge pattern enclosed in -[]->

Vertex pattern enclosed in ()

COLUMNS defines the shape of the output table. Properties projected out of the MATCH.

Example 1 - Output

CID	NAME	CITY	AMOUNT
300	Jeremy	San Francisco	15000
400	Jessica	Redwood Shores	20000
500	Fletcher	San Jose	25000

SQL/PGQ Pattern Matching Cheat Sheet – Vertex Pattern (1)

Matches a single vertex

Enclosed in parentheses ()

- ASCII art for a circle drawn around a vertex

Three (optional) components:

- Vertex graph pattern variable
- Label expression (introduced by the keyword IS)
- (local) WHERE clause to specify a condition (an SQL predicate)

SQL/PGQ Pattern Matching Cheat Sheet – Vertex Pattern (2)

()

- None of the three optional components is present
- Empty vertex pattern matches any vertex without restrictions

(C)

- Vertex pattern has only a vertex graph pattern variable: C
- Pattern matches any vertex, which can then later be referred to by the pattern variable

(IS customer)

- Vertex pattern has only a label: customer
- Pattern matches only those vertices that have the label customer

(C IS customer)

- Vertex pattern has a vertex graph pattern variable: C
- Vertex pattern has a label: customer
- Pattern matches any vertex that has a label customer, can then later be referred to by the pattern variable C

(V WHERE V.id = 12345)

- Vertex pattern has a vertex graph pattern variable: V
- Vertex pattern has a WHERE clause that specifies a condition
- Pattern matches any vertex whose property id equals 12345

(C IS customer WHERE C.first_name = 'Joe')

- Vertex pattern has a vertex graph pattern variable: C
- Vertex pattern has a label: customer
- Vertex pattern has a WHERE clause that specifies a condition
- Pattern matches any vertex that has a label customer and whose property first_name equals ‘Joe’

SQL/PGQ Pattern Matching Cheat Sheet – Edge Pattern (1)

Matches a single edge

Enclosed in arrow tokens

Three (optional) components (same as in vertex pattern):

- Edge graph pattern variable
- Label expression (introduced by the keyword IS)
- (local) WHERE clause to specify a condition (an SQL predicate)

Two (x3) variants:

	directed pointing to the right	directed pointing to the left	Any edge (pointing right or left)
Brackets to enclose optional components	-[ ]->	<-[ ]-	-[ ]-
No filler	->	<-	-

SQL/PGQ Pattern Matching Cheat Sheet – Edge Pattern (2)

->

- Edge pattern that matches any (outgoing) edge
- None of the three optional components can be specified

-[]->

- Same as above but any of the four optional components could be specified between the brackets

-[E]->

- Matches any (outgoing) edge and specifies pattern variable E
- One can refer to this edge later in the query using the pattern variable E

-[IS knows]->

- Matches any (outgoing) edge that has a label knows

-[E IS owns]->

- Matches any (outgoing) edge that has a label owns, can then later be referred to by the pattern variable E

-[E WHERE E.since = '1999']->

- Matches any (outgoing) edge whose property since equals ‘1999’.

-[E IS knows WHERE E.since = '1999']->

- Matches any (outgoing) edge that has a label knows and whose property since equals ‘1999’.

Variable-length Path Patterns

Quantification is used to express variability/repetition in the pattern.

Uses postfix quantifiers:

- ? — 0 or 1 iterations
- { n } — exactly n iterations (n > 0)
- { n, m } — between n and m (inclusive) iterations (0 <= n <= m, 0 < m)
- { , m } — between 0 and m (inclusive) iterations (m > 0)

Additionally:

- { n, } — n or more iterations (n >= 0)
- * — 0 or more iterations
- + — 1 or more iterations

  Dangerous, if graph contains cycles
  - Query has the potential to not terminate and/or produce infinite results
  - Need to put in safeguards to ensure termination

Querying PGs – Example 1a

Retrieve the info of all customers who got more than $10,000 from customer 100 via 1 intermediary.

```
SELECT gt.cid, gt.name, gt.city, gt.amount1, gt.amount2
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]->
        ( IS account ) -[ t2 IS transfers ]->
        ( IS account ) <-[ IS owns ]- ( c2 IS customer )
  WHERE c1.cid = 100
    AND t1.amount > 10000
    AND t2.amount > 10000
  COLUMNS ( c2.cid
          , c2.name
          , c2.city
          , t1.amount AS amount1
          , t2.amount AS amount2 )
 ) gt
```

Output

CID	NAME	CITY	AMOUNT1	AMOUNT2
400	Jessica	Redwood Shores	15000	11000
300	Jeremy	San Francisco	20000	15000
500	Fletcher	San Jose	15000	14000

Querying PGs – Example 2

Retrieve the info of all unique customers who are connected to fraudster 500 within four hops via money transfers (incoming, outgoing, or both).

```
SELECT DISTINCT gt.cid, gt.name, gt.city
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ IS transfers ]- {1,4}
        ( IS account ) <-[ IS owns ]- ( c2 IS customer )
  WHERE c1.cid = 500
  COLUMNS ( c2.cid
          , c2.name
          , c2.city )
 ) gt
```

Repeating edge (1 to 4 times)

Querying PGs – Example 2a

Retrieve the info of all customers who are connected to fraudster 500 within four hops via money transfers (incoming, outgoing, or both), and their closest distance from 500.

```
SELECT gt.cid, gt.name, gt.city, MIN(gt.hops)
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]- {1,4}
        ( IS account ) <-[ IS owns ]- ( c2 IS customer )
  WHERE c1.cid = 500
  COLUMNS ( c2.cid
          , c2.name
          , c2.city
          , COUNT (t1.when) AS hops )
 ) gt
GROUP BY gt.cid, gt.name, gt.city
```

Aggregate function inside COLUMNS clause.

Querying PGs – Example 3

Find the (shortest) loops where money flows back to the original sender (not necessarily the same account).

```
SELECT DISTINCT gt.cid, gt.name, gt.city, gt.min_hops
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]->*
        ( IS account ) <-[ IS owns ]- ( c1 )
  KEEP ANY SHORTEST
  COLUMNS ( c1.cid
          , c1.name
          , c1.city
          , COUNT(t1.when) AS min_hops)
 ) AS gt
```

Kleene * means 0 to unbounded iterations

SHORTEST makes an unbounded upper limit computable.

SQL/PGQ Pattern Matching Cheat Sheet – KEEP clause

Path selector

- ALL (default)
- ANY
- ANY k
- ALL SHORTEST
- ANY SHORTEST
- SHORTEST k

Path restrictor

- WALK (default) — no filtering taking place
- TRAIL — filters paths with repeated edges
- ACYCLIC — filters paths with repeated vertices
- SIMPLE — filters paths with repeated vertices unless repeated vertices are the first and last in path

Any combination of path selector plus path restrictor is permitted.*

For example:

- KEEP ANY TRAIL
- KEEP SHORTEST 10 ACYCLIC PATHS

Optionally suffixed with PATH/PATHS keyword.

* With necessary safeguards to guarantee termination in case of unbounded quantifiers.

SQL/PGQ Number of Rows Per Match

ONE ROW PER VERTEX and ONE ROW PER STEP allow for unnesting/normalizing of path data.

Graph Table Rows Clause

- ONE ROW PER MATCH — the default
  produces one row per match
- ONE ROW PER VERTEX **(v)** — declares a single iterator vertex variable
  produces one row per vertex
- ONE ROW PER STEP **(v1, e, v2)** — declares an iterator vertex variable, an iterator edge variable, and another iterator vertex variable
  produces one row per step (= a vertex-edge-vertex triple)

Querying PGs – Example 4

Find all simple paths from fraudster 500 back to itself. Output all account numbers and transaction amounts along paths.

```
SELECT *
FROM GRAPH_TABLE ( aml
  MATCH
   ( c1 IS customer ) -[ IS owns ]->
        ( IS account ) -[ t1 IS transfers ]->+
        ( IS account ) <-[ IS owns ]- ( c1 )
  KEEP ALL SIMPLE PATHS
  WHERE c1.cid = 500
  ONE ROW PER STEP ( v1, e, v2 )
  COLUMNS ( MATCHNUM() AS matchnum
          , ELEMENT_NUMBER(e) AS elemnum
          , v1.aid AS from_acct
          , e.amount
          , v2.aid AS to_acct)
 )
WHERE amount IS NOT NULL /* gets rid of 'owns' edges */
ORDER BY matchnum, elemnum
```

SIMPLE means no repeated vertices allowed other than first+last (**c1** in this case)

WHERE clause after KEEP clause specifies a postfilter*

ONE ROW PER STEP outputs one row for each step in a path (a step is a vertex-edge-vertex triple)

* For this query the result is the same no matter if the filter is a prefilter or a postfilter, but that is not generally the case.

Example 4 - Output

	MATCHNUM	ELEMNUM	FROM_ACCT	AMOUNT	TO_ACCT
path #1 has 2 transfers edges	1	4	10	15000	30
	1	6	30	4000	50
path #2 has 2 transfers edges	2	4	10	15000	30
	2	6	30	14000	50
path #3 has 3 transfers edges	3	4	10	20000	40
	3	6	40	15000	30
	3	8	30	4000	50
path #4 has 3 transfers edges	4	4	10	2000	40
	4	6	40	15000	30
	4	8	30	14000	50
path #5 has 1 transfers edge	5	4	10	25000	50

Possible future extensions to SQL/PGQ

There are ideas and some initial write-ups for standardizing additional features.

Additional path selectors

- For example: cheapest, minimal, maximal path

Conditions that cross iterations of a quantifier

- LDBC Financial Benchmark example: find a path of transactions where the timestamp keeps increasing for each two consecutive edges

Optional pattern matching

- Like left outer join

Exporting vertices, edges, paths, or entire matches to JSON

- Example use case: graph visualization based on graph elements returned from a query

Thank you

ISO Standards Sectors About ISO News Taking part Store

Free learning opportunity: Join us online in September! →

Read sample

ISO/IEC 9075-16:2023

Information technology — Database languages SQL

Part 16: Property Graph Queries (SQL/PGQ)

Published (Edition 1, 2023)

General information

Status : Published
Publication date : 2023-06
Stage : International Standard published [60.60]

Edition : 1
Number of pages : 269

Technical Committee : ISO/IEC JTC 1/SC 32
ICS : 35.060

RSS updates