Graph Query Language For Navigating Complex Data [DEV5447]

Oskar van Rest - Principal Member of Technical Staff, Oracle
Vlad Haprian - Senior Researcher, Oracle
Sungpack Hong - Research Director at Oracle Labs, Oracle

October 22, 2018
Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing, and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.
Graph Data Model

- **What is a graph?**
  - Data model representing entities as **vertices** and relationships as **edges**
  - Optionally including:
    - Vertex/edge labels, typically for denoting types
    - Vertex/edge attributes ("properties")

- **Flexible** data model
  - No predefined **schema**, easily extensible
  - Particularly useful for sparse data
Main Benefits of Graph Model

- Powerful data analytics for business insight
- Visualization and interactive data browsing
- Easy query and fast navigation of complex data // focus of This talk

Covered in other talks (See related talks at the end)
PGQL – Property Graph Query Language

- **Familiar SQL-like syntax**
  - SELECT .. FROM .. WHERE ..
  - GROUP BY, HAVING, ORDER BY, etc.

- **Graph pattern matching**
  - Define a high-level pattern and match all the instances in the data graph

- **Recursive path expressions**
  - E.g. can I reach from vertex A to vertex B via any number of edges?

- **PGQL has been input to ongoing standardization efforts**
  - SQL/PG (SQL extension for property graphs)
  - GQL (standalone graph query language)

**Example query:**

```
SELECT p2.name AS friend_of_friend
FROM facebook_graph
MATCH (p1:Person) -[:friend_of{2}] -> (p2:Person)
GROUP BY ..
ORDER BY ..
LIMIT ..
OFFSET ..
```

---

Additional mention:

- PGQL has been input to ongoing standardization efforts
  - SQL/PG (SQL extension for property graphs)
  - GQL (standalone graph query language)

**http://pgql-lang.org/**
Example: Network Impact Analysis

- How does network disruption impacts reachability between electric devices?

```sql
PATH connects_to
AS (from) <-[c1]- (connector) -[c2]-> (to)
WHERE c1.status = 'OPEN'
AND c2.status = 'OPEN'
SELECT n.nickname, COUNT(m)
FROM Electric_Network
MATCH (n:Device) -[:connects_to]-> (m:Device)
WHERE java_regexp_like(n.nickname, 'Regulator')
AND n <> m
GROUP BY n
ORDER BY COUNT(m) DESC, n.nickname
```

<table>
<thead>
<tr>
<th>n.nickname</th>
<th>COUNT(m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regulator, VREG2_A</td>
<td>1596</td>
</tr>
<tr>
<td>Regulator, VREG4_B</td>
<td>1537</td>
</tr>
<tr>
<td>Regulator, VREG4_C</td>
<td>1537</td>
</tr>
<tr>
<td>Regulator, HVMV_Sub_RegA</td>
<td>3</td>
</tr>
<tr>
<td>Regulator, HVMV_Sub_RegB</td>
<td>3</td>
</tr>
</tbody>
</table>

Query: For each ‘Regulator’ device, show number of reachable devices following only ‘OPEN’ connections.
### PGQL compared to SQL

**Query:** Which devices are connected transitively to device 'Regulator, HVMV_Sub_RegB'?

**PGQL**

```sql
WITH temp(device_id, device_name) AS (
  -- Anchor member:
  SELECT device_id, name
  FROM Devices
  WHERE name = 'Regulator, HVMV_Sub_RegB'
  UNION ALL
  -- Recursive member:
  SELECT Devices.device_id, Devices.name
  FROM temp, Devices, Connections conn1,
       Connections conn2, Connectors
  WHERE temp.device_id = conn1.to_device_id
  AND conn1.from_connector_id = Connectors.connector_id
  AND Connectors.connector_id = conn2.from_connector_id
  AND conn2.to_device_id = Devices.device_id
  AND temp.device_id != Devices.device_id
CYCLE device_id SET cycle TO 1 DEFAULT 0
SELECT DISTINCT device_name
FROM temp
WHERE cycle = 0
AND device_name <> 'Regulator, HVMV_Sub_RegB'
)
```

**SQL**

```sql
PATH connects_to AS (from) <-> (connector) -> (to)
SELECT y.name
MATCH (x:Device) -[:connects_to*] - (y:Device)
WHERE x.name = 'Regulator, HVMV_Sub_RegB'
AND x <> y
```

**PATH**

- `connects_to` connects two devices transitively.

**SELECT**

- The name of the device `y`.

**MATCH**

- Match all devices `x` that are connected to device `y` transitively.

**WHERE**

- The device `x` must be named 'Regulator, HVMV_Sub_RegB' and different from `y`.

**PATH**

- The path connects `from` to `to` through one or more connectors.

**SELECT**

- The name of device `y`.

**MATCH**

- Match the device `x` as before.

**WHERE**

- The device `x` must be named 'Regulator, HVMV_Sub_RegB' and different from `y`.

**PATH**

- The path connects `from` to `to` through one or more connectors.

**SELECT**

- Distinct device names.

**FROM**

- The `temp` table.

**WHERE**

- The cycle is 0.

- The device name is different from 'Regulator, HVMV_Sub_RegB'.
PGQL: basic Graph Pattern Matching

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
FROM socialNetworkGraph
MATCH (v1:Person) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
WHERE v1.name = 'Amber'
```

Query: Find all people who are known by friends of ‘Amber’.
PGQL: basic Graph Pattern Matching

- Find all instances of a given pattern/template in the data graph

```
SELECT v3.name, v3.age
FROM socialNetworkGraph
MATCH (v1:Person) -[:friendOf]-> (v2:Person) -[:knows]-> (v3:Person)
WHERE v1.name = 'Amber'
```

**Query:** Find all people who are known by friends of ‘Amber’. 
PGQL: Regular Path Expressions

• Matching a pattern repeatedly
  – Define a PATH expression at the top of a query
  – Instantiate the expression in the MATCH clause
  – Match repeatedly, e.g. zero or more times (*) or one or more times (+)

```sql
PATH has_parent AS (child) -[:has_father|has_mother]-> (parent)
SELECT x.name, y.name, ancestor.name
FROM snGraph
MATCH (x:Person) -[:has_parent+]-> (ancestor)
  , (y) -[:has_parent+]-> (ancestor)
WHERE x.name = 'Peter' AND x <> y
```
PGQL: Regular Path Expressions

• Matching a pattern repeatedly
  – Define a PATH expression at the top of a query
  – Instantiate the expression in the MATCH clause
  – Match repeatedly, e.g. zero or more times (*) or one or more times (+)

```
PATH has_parent AS (child) -[:has_father|has_mother]-> (parent)
SELECT x.name, y.name, ancestor.name
FROM snGraph
MATCH (x:Person) -/[:has_parent+]-> (ancestor)
  , (y) -/[:has_parent+]-> (ancestor)
WHERE x.name = 'Peter' AND x <> y
```

<table>
<thead>
<tr>
<th>Result set</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.name</td>
</tr>
<tr>
<td>Peter</td>
</tr>
<tr>
<td>Peter</td>
</tr>
<tr>
<td>Peter</td>
</tr>
</tbody>
</table>

```
:Person
  name = 'Amber'
  age = 29

:Person
  name = 'Dwight'
  age = 15

:Person
  name = 'Paul'
  age = 64

:Person
  name = 'Retta'
  age = 43

:Person
  name = 'Peter'
  age = 12
```
PGQL: SELECT, FROM, MATCH, WHERE

• **SELECT** creates tabular projection of results
  – Alias names: `SELECT (x.prop1 + x.prop2) / 2 AS avg_prop1_prop2`
  – Remove duplicates: `SELECT DISTINCT x.prop`

• **FROM** and **MATCH** to specify the graph and the pattern to match on it
  – Example: `FROM g MATCH (n) -[e]-> (m)`

• **WHERE** (optional) contains a filter expression:
  – The filter expression typically involves properties of vertices/edges defined in the MATCH clause
    • e.g. `WHERE m.budget > 1000000 AND m.budget < 2000000 AND ...`

<table>
<thead>
<tr>
<th>Operator type</th>
<th>Operator</th>
<th>Example PGQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>+, -, *, /, %</td>
<td>1 + 1</td>
</tr>
<tr>
<td>Relational</td>
<td>=, &lt;&gt;, &lt;, &gt;, &lt;=, &gt;=</td>
<td>3 &lt; 4</td>
</tr>
<tr>
<td>Logical</td>
<td>AND, OR, NOT</td>
<td>true AND NOT false</td>
</tr>
</tbody>
</table>

Copyright © 2018, Oracle and/or its affiliates. All rights reserved.
# PGQL: patterns, labels, regular expressions

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v1)</td>
<td>Match vertex v1</td>
<td>-/:lbl/-&gt;</td>
<td>Matches if there exists a path that connects the source and destination by <strong>one</strong> match of the given pattern</td>
</tr>
<tr>
<td>-[e1]-&gt;</td>
<td>Match outgoing edge e1</td>
<td>-/:lbl*/-&gt;</td>
<td>Matches if there exists a path that connects the source and destination by <strong>zero or more</strong> matches of the given pattern</td>
</tr>
<tr>
<td>&lt;-[e1]-</td>
<td>Match incoming edge e1</td>
<td>-/:lbl+/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>one or more</strong> matches of the given pattern</td>
</tr>
<tr>
<td>-[e1]-</td>
<td>Match incoming or outgoing edge e1</td>
<td>-/:lbl?/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>zero or one</strong> matches of the given pattern</td>
</tr>
<tr>
<td>()</td>
<td>Match anonymous vertex</td>
<td>-/:lbl{2}/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>exactly 2</strong> matches of the given pattern</td>
</tr>
<tr>
<td>-&gt;</td>
<td></td>
<td>-/:lbl{2,}/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>at least 2</strong> matches of the given pattern</td>
</tr>
<tr>
<td>&lt;-</td>
<td></td>
<td>-/:lbl{2,3}/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>at least 2 and at most 3 (inclusive)</strong> matches of the given pattern</td>
</tr>
<tr>
<td>-</td>
<td></td>
<td>-/:lbl{,3}/-</td>
<td>Matches if there exists a path that connects the source and destination by <strong>at least 0 and at most 3 (inclusive)</strong> matches of the given pattern</td>
</tr>
<tr>
<td>(v1:lbl1</td>
<td>lbl2)</td>
<td>Match vertex v1 with label lbl1 or lbl2</td>
<td>(v1:lbl1</td>
</tr>
<tr>
<td>-[lbl1]/&gt;</td>
<td>Match anonymous outgoing/incoming/any-directional edge</td>
<td>(v1:lbl1</td>
<td>lbl2)</td>
</tr>
<tr>
<td>-/lbl1&gt;/-</td>
<td>Match anonymous outgoing edge with label lbl1</td>
<td>(v1:lbl1</td>
<td>lbl2)</td>
</tr>
</tbody>
</table>
PGQL: Grouping and Aggregation

• **GROUP BY** creates groups of intermediate solutions

• **COUNT, MIN, MAX, SUM, AVG** to compute aggregations over groups of solutions
  – Use **DISTINCT** to eliminate duplicates before aggregation
    • E.g. **COUNT(DISTINCT n.name)**
  – Omit **GROUP BY** to aggregate over the entire set of solutions (returns a single row)

• **HAVING** filters entire groups of solutions
  • E.g. **HAVING COUNT(*) > 10**

**Query:** "for each movie, return the time of the most recent click"

```
SELECT m.title, MAX(e.time)
FROM "Movies Graph"
MATCH (m) <-[e:click]- ()
WHERE m.type = 'movie'
GROUP BY m
```

<table>
<thead>
<tr>
<th>m.title</th>
<th>MAX(e.time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Yes Man&quot;</td>
<td>2011-08-25T20:19:40</td>
</tr>
<tr>
<td>&quot;Match Point&quot;</td>
<td>2012-09-29T02:49:09</td>
</tr>
</tbody>
</table>

**Movies Graph**

- **Clicks**
  - Click 1: 2011-08-25T20:19:40
  - Click 2: 2012-09-29T02:49:09
  - Click 3: 100

- **Users**
  - Maria
  - Lksana
  - Franklin
PGQL: ORDER BY, LIMIT, OFFSET

• ORDER BY
  – Sort result ascendingly (default): ORDER BY n.prop ASC, ...
  – Sort result descendingly: ORDER BY n.prop DESC, ...

• LIMIT and OFFSET
  – For returning the “top-k” results or for “pagination” of results
    • E.g. OFFSET 200 LIMIT 100 returns results 200 to 300

Query: “order customers by first name and return first two results”

```
SELECT n.first_name
FROM "Movies Graph"
MATCH (n)
WHERE n.type = 'customer'
ORDER BY n.first_name
LIMIT 2
```

<table>
<thead>
<tr>
<th>n.first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Franklin</td>
</tr>
<tr>
<td>Iksana</td>
</tr>
</tbody>
</table>

Movies Graph

<table>
<thead>
<tr>
<th>time</th>
<th>type</th>
<th>first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>2010-12-17T02:42:32</td>
<td>'movie'</td>
<td>'Match Point'</td>
</tr>
<tr>
<td>2010-12-17T02:42:32</td>
<td>'customer'</td>
<td>'Iksana'</td>
</tr>
<tr>
<td>2011-08-25T20:19:40</td>
<td>'movie'</td>
<td>'Yes Man'</td>
</tr>
<tr>
<td>2011-08-25T20:19:40</td>
<td>'customer'</td>
<td>'Maria'</td>
</tr>
<tr>
<td>2011-01-01T20:57:21</td>
<td>'customer'</td>
<td>'Franklin'</td>
</tr>
<tr>
<td>2011-01-01T20:57:21</td>
<td>'movie'</td>
<td>'No Man'</td>
</tr>
<tr>
<td>2012-09-29T02:49:09</td>
<td>'customer'</td>
<td>'Iksana'</td>
</tr>
<tr>
<td>2012-09-29T02:49:09</td>
<td>'movie'</td>
<td>'Yes Man'</td>
</tr>
<tr>
<td>2012-09-29T02:49:09</td>
<td>'customer'</td>
<td>'Franklin'</td>
</tr>
</tbody>
</table>
**PGQL: Data Types in the In-Memory Analyst (PGX)**
(note: for PGQL-to-SQL, the available data types correspond to those in the RDBMS)

<table>
<thead>
<tr>
<th>Data type</th>
<th>Name in PGX config</th>
<th>Example PGQL literal</th>
<th>Implicitly converts to (e.g. integer + long =&gt; LONG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING</td>
<td>string</td>
<td>'Franklin'</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>integer</td>
<td>-</td>
<td>LONG, FLOAT, DOUBLE</td>
</tr>
<tr>
<td>LONG</td>
<td>long</td>
<td>2000</td>
<td>FLOAT, DOUBLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>float</td>
<td>-</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>double</td>
<td>1.95</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>boolean</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>DATE</td>
<td>local_date</td>
<td>DATE '2018-08-18'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(note: don’t use deprecated date)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>time</td>
<td>TIME '20:15:00'</td>
<td>TIME WITH TIME ZONE</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>timestamp</td>
<td>TIMESTAMP '2018-08-18 20:15:00'</td>
<td>TIMESTAMP WITH TIME ZONE</td>
</tr>
<tr>
<td>TIME WITH TIME ZONE</td>
<td>time_with_timezone</td>
<td>TIME '20:15:00+08'</td>
<td>TIME</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>timestamp_with_timezone</td>
<td>TIMESTAMP '2018-08-18 20:15:00+08'</td>
<td>TIMESTAMP</td>
</tr>
</tbody>
</table>
PGQL: Explicit Type Conversion (CAST)

• Convert from string:
  – CAST('true' AS BOOLEAN) => true
  – CAST('2.50' AS DOUBLE) => 2.50
  – CAST('2018-08-18' AS DATE) => DATE '2018-08-18'

• Convert to string:
  – CAST(true AS STRING) => 'true'
  – CAST(2.50 AS STRING) => '2.50'
  – CAST(DATE '2018-08-18' AS STRING) => '2018-08-18'

• Narrowing numeric conversion
  – CAST(1.2 AS INTEGER) => 1

• Extract date or time portion from a timestamp:
  – CAST(TIMESTAMP '2018-08-18 20:15:00' AS DATE) => DATE '2018-08-18'
  – CAST(TIMESTAMP '2018-08-18 20:15:00' AS TIME) => TIME '20:15:00'

• Create a timestamp from a date or a time:
  – CAST(DATE '2018-08-18' AS TIMESTAMP) => TIMESTAMP '2018-08-18 00:00:00'
  – CAST(TIME '20:15:00' AS TIMESTAMP) => TIMESTAMP '2018-03-19 20:15:00'

It takes the current date to create a timestamp.
### PGQL: Built-in Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Return type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id(vertex/edge)</td>
<td>get the identifier of a vertex or edge</td>
<td>exact numeric / string</td>
</tr>
<tr>
<td>has_label(vertex/edge, lbl)</td>
<td>returns true if the vertex/edge has the label; false otherwise</td>
<td>boolean</td>
</tr>
<tr>
<td>labels(vertex)</td>
<td>returns the labels of a vertex</td>
<td>set&lt;string&gt;</td>
</tr>
<tr>
<td>label(edge)</td>
<td>returns the label of an edge</td>
<td>string</td>
</tr>
<tr>
<td>all_different(value1, value2, value3, ...)</td>
<td>returns true if the values are all different, e.g. all_different(1, 5, 2) → true e.g. all_different(1, 5, 1) → false</td>
<td>boolean</td>
</tr>
<tr>
<td>in_degree(vertex)</td>
<td>get outgoing number of edges</td>
<td>exact numeric</td>
</tr>
<tr>
<td>out_degree(vertex)</td>
<td>get incoming number of edges</td>
<td>exact numeric</td>
</tr>
<tr>
<td>java_regexp_like(string, pattern)</td>
<td>returns whether the string matches the pattern¹</td>
<td>boolean</td>
</tr>
</tbody>
</table>

¹ see [https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html)

---

Use `all_different` to avoid explosion of non-equality constraints:

**SELECT COUNT(*)**

MATCH (p1:Person) -[:likes]--> (friend:Person)
  , (p2:Person) -[:likes]--> (friend)
  , (p3:Person) -[:likes]--> (friend)
WHERE p1 <> p2 AND p1 <> p3 AND p2 <> p3

**SELECT COUNT(*)**

MATCH (p1:Person) -[:likes]--> (friend:Person)
  , (p2:Person) -[:likes]--> (friend)
  , (p3:Person) -[:likes]--> (friend)
WHERE all_different(p1, p2, p3)
PGQL: EXISTS subqueries

- **EXISTS** tests for the existence of a pattern, given the already obtained bindings

Query: “Find friends of friends of Peter”

SELECT DISTINCT friendOfFriend.name
FROM g
MATCH (a:Person) -[:friend_of]-> () -[:friend_of]-> (friendOfFriend)
WHERE a.name = 'Peter'
  AND NOT EXISTS (
    SELECT *
    MATCH (a) -[:friend_of]-> (friendOfFriend)
  )
**Static code analysis**: transitive function call from A to B such that no write to a global variable happens

```
PATH calls_but_no_global_write AS (:Function) -[:calls]-> (dst:Function)
  WHERE NOT EXISTS (
    SELECT *
    MATCH (dst) -[:calls*]-> (:Function) -[:writes]-> (:GlobalVariable)
  )
SELECT f2.name
FROM codeBaseGraph
MATCH (f1:Function) -[:calls_but_no_global_write*]-> (f2)
WHERE f1.name = 'A' AND f2.name = 'B'
```

No function along the path may lead to a write to a global variable.
Static code analysis: transitive function call from A to B such that no write to a global variable happens

```
PATH calls_but_no_global_write AS (:Function) -[:calls]-> (dst:Function) 
  WHERE NOT EXISTS ( 
    SELECT * 
    MATCH (dst) -[:calls*]-> (:Function) -[:writes]-> (:GlobalVariable) 
  )

SELECT f2.name 
FROM codeBaseGraph 
MATCH (f1:Function) -[:calls_but_no_global_write*]-> (f2) 
WHERE f1.name = 'A' AND f2.name = 'B'
```

From A to B:

- No function along the path may lead to a write to a global variable.
PGQL: Scalar Subqueries

- A scalar subquery is a subquery that returns a single value (single row + single column)

```sql
SELECT person.id ,
  COUNT(*) AS messageCount
FROM ldbcSnbGraph
WHERE tag.name = ?
GROUP BY person, tag
ORDER BY 1*messageCount + 2*replyCount + 10*likeCount DESC
LIMIT 100
```

Example query (from the LDBC graph benchmark)
PGQL: Cartesian Product

-Disconnected graph patterns result in Cartesian product

```
SELECT c1.first_name, c2.first_name
FROM "Movies Graph"
MATCH (c1), (c2) -[:click]-> (m)
WHERE c1.first_name = 'Maria'
AND m.title = 'Match Point'
```

Movies Graph

<table>
<thead>
<tr>
<th>c1.first_name</th>
<th>c2.first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maria</td>
<td>Iksana</td>
</tr>
<tr>
<td>Maria</td>
<td>Franklin</td>
</tr>
</tbody>
</table>

\[=\]

<table>
<thead>
<tr>
<th>c1.first_name</th>
<th>c2.first_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maria</td>
<td>Iksana</td>
</tr>
<tr>
<td>Maria</td>
<td>Franklin</td>
</tr>
</tbody>
</table>

\[=\]
Using the Groovy shell

```groovy
pgx> pg.queryPgql('"
pgx> SELECT x.name, y.name
pgx> MATCH (x) -[:leads]-> (y)
pgx> ORDER BY x.name, y.name
pgx> ""
```

<table>
<thead>
<tr>
<th>x.name</th>
<th>y.name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bobby Murphy</td>
<td>Snapchat</td>
</tr>
<tr>
<td>Ertharin Cousin</td>
<td>World Food Programme</td>
</tr>
<tr>
<td>Evan Spiegel</td>
<td>Snapchat</td>
</tr>
<tr>
<td>Google</td>
<td>Nest</td>
</tr>
<tr>
<td>Jack Ma</td>
<td>Alibaba</td>
</tr>
<tr>
<td>Jeff Bezos</td>
<td>Amazon</td>
</tr>
<tr>
<td>Pony Ma</td>
<td>Tencent</td>
</tr>
<tr>
<td>Pope Francis</td>
<td>The Vatican</td>
</tr>
<tr>
<td>Tony Fadell</td>
<td>Nest</td>
</tr>
</tbody>
</table>
```
Java API: execute PGQL queries and print results

• Execute a PGQL query:

```java
resultSet =
    graph.queryPgql("SELECT ... MATCH ...")
```

• Print the results:

```java
// prints all results to standard output
resultSet.print()

// prints the first 100 results to standard output
resultSet.print(100)
```
Java API: iterate through results and close result set

• Iterate through the result set - **Java iterator** way:

```java
for (PgqlResult result : resultSet) {
    String movie_title = result.getString("m.title");
    String movie_budget = result.getFloat("m.budget");
    ...
}
```

• Iterate through the result set - **JDBC** way:

```java
while (resultSet.next()) {
    String movie_title = resultSet.getString("m.title");
    String movie_budget = resultSet.getFloat("m.budget");
    ...
}
```

• Close the result set:

```java
resultSet.close()
```
Using Bind Variables

• Much like JDBC: use "?" for variable markers.

• Prepare the Statement:

```java
stmt = graph.preparePgql("SELECT ... MATCH ... WHERE ..."
```  

• Set bind values

```java
stmt.setString(1, "Alibaba") // first bind variable has index 1
stmt.setInt(2, 100)
stmt.setDouble(3, 2898.10)
...```

• Execute prepared statement

```java
rs = stmt.executeQuery()
```
Datetime data types in Java, and mapping to PGQL types

• Using the new Java 8 Date-Time library¹
  – java.time.*
  – New library has five main classes, one for each of the five PGQL date-time types
    • Note: types correspond but names differ (e.g. TIMESTAMP <---> LocalDateTime)

<table>
<thead>
<tr>
<th>Data type</th>
<th>Result Set API (Java)</th>
<th>Prepared Statement API (Java)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>LocalDate getDate(..)</td>
<td>setDate(LocalDate)</td>
</tr>
<tr>
<td>TIME</td>
<td>LocalTime getTime(..)</td>
<td>setTime(LocalTime)</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>LocalDateTime getTimestamp(..)</td>
<td>setTimestamp(LocalDateTime)</td>
</tr>
<tr>
<td>TIME WITH TIME ZONE</td>
<td>OffsetTime getTimeWithTimezone(..)</td>
<td>setTimeWithTimezone(OffsetTime)</td>
</tr>
<tr>
<td>TIMESTAMP WITH TIME ZONE</td>
<td>OffsetDateTime getTimestampWithTimezone(..)</td>
<td>setTimestampWithTimezone(OffsetDateTime)</td>
</tr>
</tbody>
</table>

¹ Java SE 8 Date and Time - http://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

alternatively, access a date/time/timestamp as a java.util.Date through getLegacyDate(..)
Graph Product Options

**Oracle Big Data Spatial and Graph**

- Available for Big Data platform/BDCS
  - Hadoop, HBase, Oracle NoSQL
- Supported both on BDA and commodity hardware
  - CDH and Hortonworks
- Database connectivity through Big Data Connectors or Big Data SQL
- Included in Big Data Cloud Service

**Oracle Spatial and Graph**

- Available with Oracle 18c/12.2/DBCS
- Using tables for graph persistence
- Graph views on relational data
- In-database graph analytics
  - Sparsification, shortest path, page rank, triangle counting, WCC, sub graphs
- SQL queries possible
- Included in Database Cloud Service
Graph Query (PGQL)

/* find friends of friends of Clara */
SELECT fof.name
FROM myGraph
MATCH (p:Person) -/:knows{2}/-> (fof:Person)
WHERE p.name = 'Clara'

In-memory Analyst / PGX
- Extreme performance, especially for analytical queries
- Can combine graph algorithms with graph queries
- Efficient property update through API (topology update only through Delta Update)

Analytical graph query

RDBMS (PGQL-to-SQL)
- Schema-less (the graph is stored as triples)
- Efficient property and topology update through API/SQL
- Can query very large data sets that don’t fit in the memory of a single machine

Transactional graph query

Oracle RDBMS

In-memory Analyst (PGX)

Delta Update
- Synchronizes an in-memory graph snapshot with graph changes from RDBMS
- Every x seconds/minutes/hours or upon request

Oracle RDBMS
Built-in graph analytic functions

\[ \text{graph} \Rightarrow \text{run analytic function} \Rightarrow \text{graph'} \Rightarrow \text{run PGQL graph query} \Rightarrow \text{etc.} \]

- Rich set of built-in (parallel) graph algorithms
  - Detecting Components and Communities: Tarjan’s, Kosaraju’s, Weakly Connected Components, Label Propagation (w/ variants), Soman and Narang’s Sparcification
  - Ranking and Walking: Pagerank, Personalized Pagerank, Betweenness Centrality (w/ variants), Closeness Centrality, Degree Centrality, Eigenvector Centrality, HITS, Random walking and sampling (w/ variants)
  - Evaluating Community Structures: Conductance, Modularity, Clustering Coefficient (Triangle Counting), Adamic-Adar
  - Path-Finding: Hop-Distance (BFS), Dijkstra’s, Bi-directional Dijkstra’s, Bellman-Ford’s
  - Link Prediction: SALSA (Twitter’s Who-to-follow)
  - Other Classics: Vertex Cover, Minimum Spanning-Tree (Prim’s)

- as well as parallel graph mutation operations
  - The original graph
  - Create Bipartite Graph
  - Create Undirected Graph
  - Filter-Expression
  - Sort-By-Degree (Renumbering)
  - Filtered Subgraph
  - Simplify Graph
  - Left Set: “a,b,e”
Example: Topic analysis in an Online Forum

• Analysis Goals:
  – Identify popular *topics* in on-line forum
  – Understand how these topics evolve
  – Detect expert users in certain topics

• Graph Approach
  – Create graph from postings and tags
  – Apply graph partitioning (community detection) algorithms

Comparing to traditional ML approach (e.g. LDA), this approach often results better quality of answer, with less susceptibility to hyper-parameters
Summary

• The **graph data model** enables easy and fast navigation of complex data
• **PGQL** is a query language for the graph data model
• **PGQL’s** syntax and semantic **aligns with SQL**
• In addition to basic graph pattern matching, **PGQL supports powerful concepts** like existential subqueries, reachability, and path queries.
Spatial and Graph at OOW and Code One 2018

View this list at tinyurl.com/SpatialGraphOOW18

## Sessions

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Title</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday, Oct. 22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9:00 am - 9:45 am</td>
<td>Graph Query Language For Navigating Complex Data [DEV5447]</td>
<td>Moscone West - 2016</td>
</tr>
<tr>
<td>11:30 am – 12:15 pm</td>
<td>Automate Anomaly Detection with Graph Analytics [DEV5397]</td>
<td>Moscone West - 2022</td>
</tr>
<tr>
<td>1:30 pm – 2:15 pm</td>
<td>How to Build Geospatial Analytics with Python and Oracle Database [DEV5185]</td>
<td>Moscone West - 2003</td>
</tr>
<tr>
<td>2:30 pm – 3:15 pm</td>
<td>Location-Based Tracking of Moving Objects with Apache Spark [DEV5355]</td>
<td>Moscone West – 2016</td>
</tr>
<tr>
<td>4:45 pm – 5:30 pm</td>
<td>Introduction to Graph Analytics and Oracle Graph Cloud Service [TRN4098]</td>
<td>Moscone West – 3004</td>
</tr>
<tr>
<td>5:45 pm – 6:30 pm</td>
<td>How to Analyze Data Warehouse Data as a Graph [TRN4099]</td>
<td>Moscone West – 3004</td>
</tr>
</tbody>
</table>
## Spatial and Graph at OOW and Code One 2018

View this list at [tinyurl.com/SpatialGraphOOW18](http://tinyurl.com/SpatialGraphOOW18)

### Sessions

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Title</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuesday, Oct. 23</td>
<td><strong>1:45 pm - 2:30 pm</strong> Machine Learning with R and Zeppelin on Oracle Big Data Solutions [PRO4054]</td>
<td>Moscone West - 3005</td>
</tr>
<tr>
<td></td>
<td><strong>4:00 pm – 4:45 pm</strong> Build Serverless Big Data and Graph Viz Web Apps with Spring Data and Core Java [DEV5479]</td>
<td>Moscone West - 2001</td>
</tr>
<tr>
<td></td>
<td><strong>5:45 pm – 6:30 pm</strong> Geo-Tagging, Geo-Enrichment, Geo-Fencing, and Tracking for Location-Enabled Apps [TRN4095]</td>
<td>Moscone West - 3003</td>
</tr>
<tr>
<td>Thursday, Oct. 25</td>
<td><strong>11:00 am – 11:45 am</strong> Using Location in Cloud Applications with Python, Node.js, and More [TRN4089]</td>
<td>Moscone West – 3001</td>
</tr>
<tr>
<td></td>
<td><strong>2:00 pm – 2:45 pm</strong> Analyzing Blockchain and Bitcoin Transaction Data as Graphs [DEV4835]</td>
<td>Moscone West – 2018</td>
</tr>
</tbody>
</table>
Meet the Experts

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Title</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tuesday, Oct. 23</td>
<td><strong>Location-Based Tracking and Geospatial Analytics for Database and Big Data Platforms [MTE6755]</strong></td>
<td>Moscone West – The Hub – Lounge B</td>
</tr>
<tr>
<td>11:00 am - 11:50 am</td>
<td>Location-Based Tracking and Geospatial Analytics for Database and Big Data Platforms [MTE6755]</td>
<td>Moscone West – The Hub – Lounge B</td>
</tr>
<tr>
<td>12:00 pm – 12:50 pm</td>
<td>Graph Analysis and Database Technologies [MTE6744]</td>
<td>Moscone West – The Hub – Lounge B</td>
</tr>
<tr>
<td>3:00 pm – 3:50 pm</td>
<td>Graph Analysis and Machine Learning [MTE6746]</td>
<td>Moscone West – The Hub – Lounge B</td>
</tr>
<tr>
<td>Wednesday, Oct. 24</td>
<td><strong>Graph Queries and Analysis [MTE6746]</strong></td>
<td>Moscone West – The Hub – Lounge A</td>
</tr>
<tr>
<td>11:00 am – 11:50 am</td>
<td>Graph Queries and Analysis [MTE6746]</td>
<td>Moscone West – The Hub – Lounge A</td>
</tr>
<tr>
<td>12:00 pm – 12:50 pm</td>
<td>Location-Based Tracking and Geospatial Analytics for Database and Big Data Platforms [MTE6755]</td>
<td>Moscone West – The Hub – Lounge B</td>
</tr>
</tbody>
</table>
### Demos

<table>
<thead>
<tr>
<th>Date/Time</th>
<th>Title</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monday 9:45 am – 5:45 pm</td>
<td>Oracle Spatial and Graph Database, Analytics, and Cloud Services</td>
<td>Moscone South Exhibit Hall (‘The Exchange’)</td>
</tr>
<tr>
<td>Tuesday 10:30 am – 5:45 pm</td>
<td></td>
<td>• Oracle Demogrounds: Cloud Platform &gt; Application Development area</td>
</tr>
<tr>
<td>Wednesday 10:30 am – 4:45 pm</td>
<td></td>
<td>• Kiosk # APD-WU3</td>
</tr>
</tbody>
</table>
Moscone South Exhibit Hall / Demogrounds ("The Exchange")

Oracle Spatial and Graph demopod (behind the Keynote Arena)
Kiosk #: APD-WU3

Intelligent Business Applications

Application Development

Cloud Platform
The Spatial & Graph SIG User Community

We are a vibrant community of customers and partners that connects and exchanges knowledge online, and at conferences and events.

Join us online

tinyurl.com/oraclespatialcommunity

LinkedIn  Google+  IOUG SIG

@oraspalialsig

oraclespatialsg@gmail.com
Call for Speakers now open!
Submit an abstract to share your use case or technical session

Analytics and Data Summit
All Analytics. All Data. No Nonsense.
March 12 – 14, 2019

Formerly called the BIWA Summit with the Spatial and Graph Summit
Same great technical content...new name!
www.AnalyticsandDataSummit.org