GraphScript: Implementing Complex Graph Algorithms in SAP HANA

Marcus Paradies, SAP SE
September 1, 2017

PUBLIC
SAP HANA Overview

<table>
<thead>
<tr>
<th>APPLICATION SERVICES</th>
<th>PROCESSING SERVICES</th>
<th>INTEGRATION &amp; QUALITY SERVICES</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web Server</td>
<td>Spatial</td>
<td>Data Virtualization</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Graph</td>
<td>ELT &amp; Replication</td>
</tr>
<tr>
<td>Fiori UX</td>
<td>Predictive</td>
<td>Data Quality</td>
</tr>
<tr>
<td>Graphic Modeler</td>
<td>Text Analytics</td>
<td>Hadoop &amp; Spark Integration</td>
</tr>
<tr>
<td>Application Lifecycle Management</td>
<td>Streaming Analytics</td>
<td>Remote Data Sync</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**DATABASE SERVICES**

- Columnar OLTP+OLAP
- Multi-Core & Parallelization
- Advanced Compression
- Multi-tenancy
- Multi-Tier Storage
- Data Modeling
- Openness
- Admin & Security
- High Availability & Disaster Recovery

- Offers advanced analytics features for graph, text, geospatial, and machine learning directly on business data
Graph Querying Paradigms in SAP HANA

**Graph Pattern Matching**

- Example Query: "Retrieve all suppliers of Company D"
- Language Interface: openCypher*

**Graph Analysis**

- Example Query: "Compute all communities in the graph"
- Language Interface: GraphScript

* Limited subset of language spec
Design Principles

Expressiveness & Simplicity
- Easy-to-use for graph algorithm implementers
- Support for a large variety of graph algorithm classes and workflows

Minimality & Orthogonality
- Limited but effective set of types and operations thereon
- Extensibility of built-in graph operators

Native Graph Abstraction
- Native exposure of graph-specific types
- Full exposure of graph data model
- Relational only for returning complex results

Tight Integration
- Pushdown of operations to relational store
- Reuse of dependency management
- Reuse of resource management

High Performance
- Desired performance close to hand-written code
- Explicit parallelization
- Effective Program Rewritings
GraphScript Type System

Graph Types
- Vertex
- Edge
- Path

Container Types
- Bag<T>
- Sequence<T>
- Table<T...>

SQL Scalar Types
- Int
- Text
- Timestamp
- ST_Point
Graph Data Exposure in GraphScript

Vertex Table

```sql
CREATE COLUMN TABLE MYSCHEMA.VERTICES (
  ID VARCHAR(100) PRIMARY KEY,
  TYPE VARCHAR(100),
  NAME VARCHAR(100),
  TITLE VARCHAR(100)
);
```

Edge Table

```sql
CREATE COLUMN TABLE MYSCHEMA.EDGES (
  ID INTEGER PRIMARY KEY,
  SRC VARCHAR(100) NOT NULL
    REFERENCES MYSCHEMA.VERTECES (ID),
  TRGT VARCHAR(100) NOT NULL
    REFERENCES MYSCHEMA.VERTECES (ID),
  TYPE VARCHAR(50)
);
```

Graph Workspace

```sql
CREATE GRAPH WORKSPACE MYSCHEMA.MY_GRAPH
  EDGE TABLE MYSCHEMA.EDGES
    SOURCE COLUMN SRC
    TARGET COLUMN TRGT
    KEY COLUMN ID
  VERTEX TABLE MYSCHEMA.VERTECES
    KEY COLUMN ID;
```
**Graph Data Exposure in GraphScript /2**

**Vertex Table View**

```
CREATE VIEW MYSHEMA.VERTEX_VIEW AS
SELECT * FROM MYSHEMA.VERTECES
WHERE TYPE = 'Person';
```

**Edge Table View**

```
CREATE VIEW MYSHEMA.EDGE_VIEW AS
SELECT * FROM MYSHEMA.EDGES
WHERE TYPE = 'knows';
```

**Graph Workspace**

```
CREATE GRAPH WORKSPACE MYSHEMA.MY_SUBGRAPH
EDGE TABLE MYSHEMA.EDGE_VIEW
  SOURCE COLUMN SRC
  TARGET COLUMN TRGT
  KEY COLUMN ID
VERTEX TABLE MYSHEMA.VERTEX_VIEW
  KEY COLUMN ID;
```
A Simple GraphScript Example

CREATE PROCEDURE "myGraphProc"(OUT numNeighbors BIGINT)
LANGUAGE GRAPH READS SQL DATA AS
BEGIN
  Graph g = Graph("myGraph");
  ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(BIGINT cnt = 0);
  FOREACH v IN Vertices(:g) {
    v.cnt = Count(Neighbors(:g, :v, 1, 3));
  }
  FOREACH v IN Vertices(:g) {
    numNeighbors += :v.cnt;
  }
END
Adjacency List Construction

<table>
<thead>
<tr>
<th>Adjacency List</th>
<th>Vertex Key Dictionary</th>
<th>Vertex Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2</td>
<td>C</td>
<td>C green</td>
</tr>
<tr>
<td>1 2 5</td>
<td>D</td>
<td>D green</td>
</tr>
<tr>
<td>2</td>
<td>E</td>
<td>E yellow</td>
</tr>
<tr>
<td>3 4 2</td>
<td>A</td>
<td>A green</td>
</tr>
<tr>
<td>4 0 2</td>
<td>B</td>
<td>B green</td>
</tr>
<tr>
<td>5 6</td>
<td>F</td>
<td>F yellow</td>
</tr>
<tr>
<td>6 2</td>
<td>G</td>
<td>G green</td>
</tr>
</tbody>
</table>

Key Color

Variants:
- Omit dictionary encoding for dense key domains
- Static/Dynamic/Compressed adjacency list
- Vertex/edge adjacency

Parallel Index Construction with up to 65 Mio. edge insertions/sec
### Inducing Subgraphs

<table>
<thead>
<tr>
<th>Description</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Induce a graph over all blue edges&quot;</td>
<td>Graph ( g = \text{Subgraph}(g, e \text{ IN Edges}(g) \text{ WHERE } e.\text{color} == 'blue') );</td>
</tr>
<tr>
<td>&quot;Induce a graph over all red edges that connect a green and a yellow vertex&quot;</td>
<td>Graph ( g = \text{Subgraph}(g, e \text{ IN Edges}(g) \text{ WHERE Source}(e).\text{color} == 'green' \text{ AND Target}(e).\text{color} == 'yellow' \text{ AND } e.\text{color} == 'red') );</td>
</tr>
<tr>
<td>&quot;Induce a graph overall all vertices that are reachable from vertex 4&quot;</td>
<td>Vertex ( v1 = \text{Vertex}(g, 4) ); ( g = \text{Subgraph}(g, v \text{ IN Vertices}(g) \text{ WHERE IS_REACHABLE}(g, :v1, :v) );</td>
</tr>
</tbody>
</table>
Integration with other Data Models/Scalar Types

Creation of Relational Output from GraphScript

```plaintext
Graph g = Graph("myWorkspace");
ALTER g ADD TEMPORARY VERTEX ATTRIBUTE(DOUBLE length = 0);
FOREACH v IN Vertices(:g) {
    Path p = Shortest_Path(:g, :v, Vertex(:g, 1));
    v.length = Length(:p);
}
outTab = SELECT :v.id, :v.length FOREACH v IN Vertices(:g);
```

Integration with Geospatial Processing

```plaintext
Graph g = Graph("myWorkspace");
ST_Geometry area = Vertex(:g, 'Munich').area;
Graph g1 = Subgraph(:g, v IN Vertices(:g) WHERE :v.type == 'Person'
AND ST_Within(:v.location, :area));
```
Conclusion

Language Constructs

• Rich type system with native graph types
• Powerful imperative constructs

Code Generation

• Generation of low-level code against internal Graph Storage interface
• Elimination of query processing on external keys
• Pushdown of filter conditions to relational engine

Future Work

• More language extensions towards fast traversals and user-defined function invocations
• More advanced GraphScript program rewritings and optimizations
Thank you.

Contact information:

Marcus Paradies
marcus.paradies@sap.com