GRAPHITE: An Extensible Graph Traversal Framework for Relational Database Management Systems

Marcus Paradies*, Wolfgang Lehner*, and Christof Bornhoevd° | SSDBM’15

*TU Dresden       °Risk Management Solutions, Inc.
Graph Processing on Enterprise Data

Relational + Application Logic

- Recursive Queries
- Chained Joins

✓ Data already in RDBMS
✗ SQL as the only interface/no graph abstraction
✗ Data transfer to application

Relational + Graph + Application Logic

✓ Efficient processing in GDBMS
✗ Processing on replicated data
✗ Data transfer to application
✗ No combination with other data models possible
Integration of Graph Processing into an RDBMS

How could a deep integration of graph functionality into an RDBMS look like?

Graph operators can be seamlessly combined with other plan operators.
Columnar Graph Storage

(a) Example graph.

(b) Vertex column group.

(c) Edge column group.

All available data types can be used as vertex/edge attributes
Graph Traversal Workflow

- S – set of start vertices
- \( \phi \) – edge predicate
- c – collection boundary
- r – recursion boundary
- d – traversal direction

Pluggable physical traversal kernels as implementations of a logical traversal operator
Graph Traversal Formalism

FORMAL DESCRIPTION (SET-BASED)

- A traversal operation is a totally ordered set $P$ of path steps
- Each path step $p_i$ receives a vertex set $D_{i-1}$ discovered at level $(i-1)$ and returns a set of adjacent vertices $D_i$ ($1 \leq i \leq r$, $r$ is recursion boundary)
- Initially, $D_0 = \{ S \}$

$$D_i = \{ v \mid \exists u \in D_{i-1}: e = (u,v) \in E \land \text{eval}(e, \varphi) \}$$

- The final output $R$ is defined as

$$R = (\bigcup_{i=c}^r D_i) \setminus (\bigcup_{i=0}^{c-1} D_i)$$

Target vertices

Visited vertices
Graph Traversals by Example

<table>
<thead>
<tr>
<th>Traversal Configuration</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ { A }, &quot;type='a' OR type='b'&quot; }, 2, 2, \rightarrow }</td>
<td>{ E, F }</td>
</tr>
<tr>
<td>{ { A }, &quot;type='a'&quot; }, 1, \infty, \rightarrow }</td>
<td>{ B, C, D, F }</td>
</tr>
</tbody>
</table>

Root vertex

Discovered vertex
Level-Synchronous (LS) Traversal

**Scan-based Graph Traversal**

- **Distribute**: Distribute the graph data
- **Scan**: Scan partitions (dictionary-encoded)
- **Materialize**: Fetch neighbors by position
- **Merge**: Clustering by edge type
  - Clustering by source vertex
  - Clustering by edge type

**Edge Clustering**

<table>
<thead>
<tr>
<th>$V_s$</th>
<th>$V_t$</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>F</td>
<td>a</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>a'</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>a</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>a</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>b</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>b</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>b</td>
</tr>
</tbody>
</table>
Fragmented-Incremental (FI) Traversal

- Partition column into fragments
- Track dependencies between fragments in index structure
- Goal: Minimize number of fragment reads

For a fragment size equal to |E|, FI-traversal degenerates to LS-traversal
Experimental Evaluation

**Evaluated Real-World Data Sets**

- Six real-world data sets with different topology characteristics

| ID | $|V|$ | $|E|$ | $\bar{d}_{out}$ | $\max(d_{out})$ | $\bar{\delta}$ | Size (MB) |
|----|-----|-----|--------------|---------------|-------------|----------|
| CR | 1.9 M | 2.7 M | 2.8 | 12 | 495.0 | 143 |
| LJ | 4.8 M | 68.5 M | 28.3 | 635 K | 6.5 | 1617 |
| OR | 3.1 M | 117.2 M | 76.3 | 32 K | 5.0 | 3066 |
| PA | 3.7 M | 16.5 M | 8.7 | 793 | 9.4 | 397 |
| SK | 1.7 M | 11.1 M | 13.1 | 35 K | 5.9 | 305 |
| TW | 40.1 M | 1.4 B | 36.4 | 2.9 M | 5.4 | 32686 |

**Evaluated Systems**

- Implementation in main-memory column store prototype (C++)
- Graph database (Neo4j)
- RDF DBMS (Virtuoso 7.0 with columnar storage layout)
- Commercial columnar RDBMS (via chained self-joins, with and without index support)
Experimental Evaluation

**Comparison of LS-Traversal and FI-Traversal**

Scan-based traversal outperforms fragmented traversal depending on traversal depth and graph topology.

Different performance characteristics for different fragment sizes.

Traversal performance depends on the traversal depth and the topology.
Experimental Evaluation

**System-Level Benchmark**

Combination of LS and FI-traversal outperforms native graph systems by up to two orders of magnitude.
Summary

**GRAPHITE**

- Graph processing tightly integrated into RDBMS
- Extensions of core components by graph extensions (operators, cost model, index structures)
- Topology characteristics-aware traversal operators

**GRAPH-SPECIFIC DATA STATISTICS AND ALGORITHMS**

- Diverse graph topologies demand different algorithmic design decisions
- Index scan versus full column scan decision also applies for graph traversals

**FUTURE WORK**

- Integration with temporal, spatial, and text data
- Language extensions for custom code executed during graph traversal
Contact

Marcus Paradies, TU Dresden
m.paradies@sap.com
https://wwwdb.inf.tu-dresden.de/team/external-members/marcus-paradies/
Backup Slides
Experimental Evaluation

Effect of Fragment Size and False Positive Rate

Elapsed time of FI-traversal more sensible to a change of the fragment size than a change to the false positive rate.
Experimental Evaluation

**MEMORY CONSUMPTION**

- Memory footprint can be reduced by increasing the fragment size or the false positive rate.
- Smaller memory footprint for very sparse graphs.
- No significant improvement for very sparse graphs.
Cost Model

- Models based on the total number of accessed edges including a constant access cost
- Goal is to minimize the number of edges to read

**Level-Synchronous**

\[ C_{LS} = \min\{r, \tilde{\delta}\} \cdot |E| \cdot C_e \]

**Fragmented-Incremental**

\[ C_{FI} = \min\{r, \tilde{\delta}\} \sum_{i=0}^{\min\{r, \tilde{\delta}\}} (1 + p)(\bar{d}_{out})^i \cdot \xi \cdot C_e \]