SCALABLE GRAPH DATA MANAGEMENT AND ANALYTICS WITH GRAODOOP

MARTIN JUNGHANNS, ANDRE PETERMANN, KEVIN GOMEZ, ERHARD RAHM

www.scads.de
Facebook
ca. 1.3 billion users
ca. 340 friends per user

Twitter
ca. 300 million users
ca. 500 million tweets per day

Internet
ca. 2.9 billion users

Gene (human)
20,000-25,000
ca. 4 million individuals
Patients
> 18 millions (Germany)
Illnesses
> 30,000

World Wide Web
ca. 1 billion Websites

LOD-Cloud
ca. 31 billion triples
- Relational database systems, e.g., SAP HANA, Vertexica
  - store vertices and edges in tables
  - static schemas, expensive joins

- Graph database system, e.g., Neo4J, OrientDB
  - use of property graph data model & dedicated graph storage
  - focus on online transactions and simple analytical queries

- Parallel graph processing systems, e.g., Google Pregel, Apache Giraph
  - in-memory processing of generic graphs in shared nothing cluster
  - recent approaches (Spark, Flink): analysis workflow with graph operators and general purpose data operators
  - little support for semantically expressive graphs
  - no end-to-end approach for graph analytics
- **Integrate data** from one or more sources into a dedicated graph storage with common graph data model

- Definition of **analytical workflows** from operator algebra

- Result representation in **meaningful way**
An end-to-end framework and research platform for efficient, distributed and domain independent graph data management and analytics.
GRADOOP CHARACTERISTICS

- Hadoop-based framework for graph data management and analysis
- Graph storage in scalable distributed store, e.g., HBase
- Extended property graph data model
  - operators on graphs and collections of (sub) graphs
  - support for semantic graph queries and mining
- Leverage powerful components of Hadoop ecosystem
  - MapReduce, Giraph, Spark, Flink, ...
- New functionality for graph-based processing workflows and graph mining
  - Frequent Subgraph Mining, Graph Pattern Matching ...
HIGH LEVEL ARCHITECTURE

Data flow

Control flow

Workflow Declaration

Visual

GrALa DSL

Representation

Workflow Execution

Operator Implementations

Data Integration

Graph Analytics

Representation

Extended Property Graph Model

HBase Distributed Graph Store

HDFS Cluster
HIGH LEVEL ARCHITECTURE

- Workflow Declaration
  - Visual
  - GrALa DSL

- Operator Implementations
  - Data Integration
  - Graph Analytics
  - Representation

- Extended Property Graph Model

- HBase Distributed Graph Store

- HDFS Cluster

Data flow
Control flow
1. Simple but powerful
   • intuitive graphs are flat structures of vertices and binary edges

2. Logical graphs
   • support of multiple, possibly overlapping graphs in one database is advantageous for analytical applications

3. Attributes and type labels
   • type labels and custom properties for vertices, edges and graphs

4. Parallel edges and loops
   • allow multiple relations between two vertices and self-connected relations
EXTENDED PROPERTY GRAPH MODEL

**Vertex space**

\[ \mathcal{V} = \{v_0, \ldots, v_n\} \]

**Edge space**

\[ \mathcal{E} = \{e_0, \ldots, e_m\} \]

\[ e_i = \{ v_i, v_j \mid v_i, v_j \in \mathcal{V} \} \]

**Type labels**

\[ \tau : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{G}) \rightarrow T \]

**Logical graphs**

\[ \mathcal{G} = \{ G_{DB}, G_0, \ldots, G_p \} \]

\[ G_i = \langle V, E \mid V \subseteq \mathcal{V} \land E \subseteq \mathcal{E} \rangle \]

**Properties**

\[ \kappa : (\mathcal{V} \cup \mathcal{E} \cup \mathcal{G}) \times K \rightarrow A \]

\[ DB_{EPGM} = \langle \mathcal{V}, \mathcal{E}, \mathcal{G}, T, \tau, K, A, \kappa \rangle \]
## GRAPH OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>GrALa notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>unary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pattern</td>
<td>$\mu_{G^*,\varphi} : G \rightarrow G^n$</td>
<td><code>graph.match(patternGraph,predicate) : Collection</code></td>
</tr>
<tr>
<td>Matching</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aggregation</td>
<td>$\gamma_{\alpha} : G \rightarrow G$</td>
<td><code>graph.aggregate(propertyKey,aggregateFunction) : Graph</code></td>
</tr>
<tr>
<td>Projection</td>
<td>$\pi_{\nu,\epsilon} : G \rightarrow G$</td>
<td><code>graph.project(vertexFunction,edgeFunction) : Graph</code></td>
</tr>
<tr>
<td>Summarization</td>
<td>$\varsigma_{\nu,\epsilon} : G \rightarrow G$</td>
<td>`graph.summarize(vertexGroupKeys, vertexAggregateFunction,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>edgeGroupKeys,edgeAggregateFunction) : Graph</td>
</tr>
<tr>
<td>binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combination</td>
<td>$\sqcup : G^2 \rightarrow G$</td>
<td><code>graph.combine(otherGraph) : Graph</code></td>
</tr>
<tr>
<td>Overlap</td>
<td>$\sqcap : G^2 \rightarrow G$</td>
<td><code>graph.overlap(otherGraph) : Graph</code></td>
</tr>
<tr>
<td>Exclusion</td>
<td>$\neg : G^2 \rightarrow G$</td>
<td><code>graph.exclude(otherGraph) : Graph</code></td>
</tr>
</tbody>
</table>
1: pattern = new Graph("(a)<-d-(b)-e->(c)")
2: predicate = (Graph g => g.V[$a][:type] == “Person” &&
                   g.V[$b][:type] == “Forum” &&
                   g.V[$c][:type] == “Person” &&
                   g.E[$d][:type] == “hasMember” &&
                   g.E[$e][:type] == “hasMember”)
3: result = db.match(pattern, predicate)
1: pattern = new Graph("(a)<-d-(b)-e->(c)")
2: predicate = (Graph g => g.V[$a][:type] == "Person" &&
           g.V[$b][:type] == "Forum" &&
           g.V[$c][:type] == "Person" &&
           g.E[$d][:type] == "hasMember" &&
           g.E[$e][:type] == "hasMember")
3: result = db.match(pattern, predicate)
1: `personGraph = db.G[0].combine(db.G[1]).combine(db.G[2])`
2: `vertexGroupingKeys = {"type", "city"}`
3: `edgeGroupingKeys = {"type"}`
4: `vertexAggFunc = (Vertex vSum, Set vertices => vSum["count"] = |vertices|)`
5: `edgeAggFunc = (Edge eSum, Set edges => eSum["count"] = |edges|)`
6: `sumGraph = personGraph.summarize(vertexGroupingKeys, edgeGroupingKeys, vertexAggFunc, edgeAggFunc)`
1: `personGraph = db.G[0].combine(db.G[1]).combine(db.G[2])`
2: `vertexGroupingKeys = {type: "city"}`
3: `edgeGroupingKeys = {type}`
4: `vertexAggFunc = (Vertex vSum, Set vertices => vSum["count"] = |vertices|)`
5: `edgeAggFunc = (Edge eSum, Set edges => eSum["count"] = |edges|)`
6: `sumGraph = personGraph.summarize(vertexGroupingKeys, edgeGroupingKeys, vertexAggFunc, edgeAggFunc)`
## COLLECTION OPERATORS

<table>
<thead>
<tr>
<th>Operator</th>
<th>Definition</th>
<th>GrALa notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>collection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection</td>
<td>$\sigma_\varphi : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td><code>collection.select(predicate) : Collection</code></td>
</tr>
<tr>
<td>Distinct</td>
<td>$\delta : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td><code>collection.distinct() : Collection</code></td>
</tr>
<tr>
<td>Sort by</td>
<td>$\xi_{k,d} : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td>`collection.sortBy(key, [:asc</td>
</tr>
<tr>
<td>Top</td>
<td>$\beta_n : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td><code>collection.top(limit) : Collection</code></td>
</tr>
<tr>
<td>Union</td>
<td>$\cup : (\mathcal{G}^n)^2 \rightarrow \mathcal{G}^n$</td>
<td><code>collection.union(otherCollection) : Collection</code></td>
</tr>
<tr>
<td>Intersection</td>
<td>$\cap : (\mathcal{G}^n)^2 \rightarrow \mathcal{G}^n$</td>
<td><code>collection.intersect(otherCollection) : Collection</code></td>
</tr>
<tr>
<td>Difference</td>
<td>$\setminus : (\mathcal{G}^n)^2 \rightarrow \mathcal{G}^n$</td>
<td><code>collection.difference(otherCollection) : Collection</code></td>
</tr>
<tr>
<td>auxiliary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apply</td>
<td>$\lambda_o : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td><code>collection.apply(unaryGraphOperator) : Collection</code></td>
</tr>
<tr>
<td>Reduce</td>
<td>$\rho_o : \mathcal{G}^n \rightarrow \mathcal{G}$</td>
<td><code>collection.reduce(binaryGraphOperator) : Graph</code></td>
</tr>
<tr>
<td>Call</td>
<td>$\eta_{a,p} : \mathcal{G}^n \rightarrow \mathcal{G}^n$</td>
<td>`[graph</td>
</tr>
</tbody>
</table>
2: predicate = (Graph g => |g.V| > 3)
3: result = collection.select(predicate)
2: predicate = (Graph g => |g.V| > 3)
3: result = collection.select(predicate)
1. **Social Network Analysis “Summarized Communities”**
   - Find communities by label propagation
   - Summarize vertices per community and edges between community members

2. **Business Intelligence “Top Revenue Subgraph”**
   - Find the common subgraph of the top 100 revenue business transaction graphs
// define pattern to extract persons and their “knows” relations
1: pattern = new Graph( "(a)-c->(b)" )
2: predicate = ( Graph g =>
    g.V[$a][:type] == "Person" &&
    g.V[$b][:type] == "Person" &&
    g.E[$c][:type] == "knows"")
// find all matches inside the database
3: friendships = db.match( pattern , predicate )
// combine all matches to a single graph
4: knowsGraph = friendships.reduce( Graph g, Graph f => g.combine(f) )
// remove properties
5: knowsGraph = knowsGraph.project( Vertex v =>
    new Vertex(v[:type], {}), new Edge(e[:type], {}))
// extract communities, store community at vertex property “community”
6: knowsGraph = knowsGraph.callForGraph(
    :CommunityDetectionAlgorithm , {"propertyKey":"community"})
// summarize vertices based on their community
// count edges inside and between communities
7: summarizedCommunities = knowsGraph.summarize(
    {"community"},
    ((Vertex vSum, Set vertices) => vSum["count"] = |vertices|),
    {},
    ((Edge eSum, Set edges) => eSum["count"] = |edges|))
// compute logical graphs
1: btgs = db.callForCollection( :BusinessTransactionGraphs , {} )
// define predicate function (graph contains invoice)
2: predicate = (Graph g => g.V.select(Vertex v =>
    v[:type] == “SalesInvoice”).count() > 0)
// define aggregate function (revenue per graph)
3: aggRevenue = (Graph g => g.V.values(“revenue”).sum())
// apply predicate and aggregate function
4: invBtgs = btgs.select(predicate).apply(Graph g =>
    g.aggregate(“revenue”, aggRevenue))
// sort graphs by revenue and return top 100
5: topBtgs = invBtgs.sortBy( “revenue“ , :desc ).top( 100 )
// compute overlap to find master data objects (e.g., Employees)
6: topBtgOverlap = invBtgs.reduce( Graph g, Graph h => g.overlap(h))
GRADOOP

- end-to-end framework for graph data management and analytics
- leverages Hadoop ecosystem including graph processing systems
- extended property graph model (EPGM) with powerful operators
- Gradoop graph store based on HBase
- initial implementation running (using MapReduce and Giraph)
OUTLOOK

- complete processing framework
  - implementation for all operators
  - implement more mining algorithms on EPGM (FSM, ...)
  - workflow execution layer (Tez, Spark, Flink, ...)
  - Visualization

- evaluate different storage layouts / solutions (e.g., Cassandra)
- automatic optimization of analysis workflows
- optimized graph partitioning approaches
- graph-based data integration (DeDoop)
GRADOOP TEAM

- Graph Store / Workflow Execution / Graph Pattern Matching: Martin Junghanns (wiss. MA)
- BIIIG / Workflow Execution / Frequent Subgraph Mining: Andre Petermann (wiss. MA)
- RDF Graph Analytics: Markus Nentwig (wiss. MA)
- Gradoop + Flink: Niklas Teichmann (SHK)
- Graph Partitioning: Kevin Gómez (SHK/BA)
- Visual Workflow Definition: Simon Chill (MA)
- Graph Pattern Matching: Andreas Krause (MA)
- Frequent Subgraph Mining: Thomas Döring (MA)
- Graph Visualization: Ngoc Ha Tran (MA)


A. Petermann, M. Junghanns, R. Müller, E. Rahm: BIIIG : Enabling Business Intelligence with Integrated Instance Graphs. Proc. 5th Int. Workshop on Graph Data Management (GDM 2014)

A. Petermann, M. Junghanns, R. Müller, E. Rahm: Graph-based Data Integration and Business Intelligence with BIIIG. Proc. VLDB Conf., 2014

Petermann, A.; Junghanns, M.; Müller, R.; Rahm, E.: FoodBroker - Generating Synthetic Datasets for Graph-Based Business Analytics. Proc. 5th Int. Workshop on Big Data Benchmarking (WBDB), 2014

Jindal, A. et.al.: Vertexica: your relational friend for graph analytics!. PVLDB 7(13), 2014

Rudolf, M. et.al.: The Graph Story of the SAP HANA Database. BTW, 2013
Thank you!

www.gradoop.com