Modern Graph Analytic Support in GSQL, TigerGraph's GQL

Alin Deutsch
TigerGraph Chief Scientist
Professor, UC San Diego
The Age of the Graph Is Upon Us (Again)

• Early-mid-90s: semi- or un-structured data research was all the rage
  – data logically viewed as graph
  – initially motivated by modeling WWW (page=vertex, link=edge)
  – query languages expressing constrained reachability in graph

• Late 90s-late 2000s: special case XML (graph restricted to tree shape)
  – Mature: W3C standard ecosystem for modeling and querying (XQuery, XPath, XLink, XSLT, XML Schema, …)

• Since mid 2000s: JSON and friends (also restricted to tree shape)
  – Mongodb, Couchbase, SparkSQL, GraphQL, AsterixDB, …

• Present: back to unrestricted graphs
  – Initially motivated by analytic tasks in social networks
  – Now universal use (most interesting data is linked, after all)
The Traditional Graph Data Model

• Nodes correspond to entities

• Edges correspond to binary relationships

• Edges may be directed or undirected (asymmetric, resp. symmetric relationships)

• Nodes and edges may be labeled/typed

• Nodes and edges annotated with data – both have sets of attributes (key-value pairs)
Example: Customers Buy Products
Key Traditional Language Ingredients

• Pioneered by academic work on relational query extensions for graphs (since ‘87)

– Path expressions (PEs) for navigation
– Variables for referring to and manipulating data found during navigation
– Stitching multiple PEs into complex navigation patterns → conjunctive path queries
– Constructors for new nodes and edges
Path Expressions

• Express reachability via constrained paths

• Early graph-specific extension over conjunctive queries

• Introduced initially in academic prototypes in early 90s
  – StruQL (AT&T Research - Fernandez, Halevy, Suciu)
  – WebSQL (U Toronto - Mendelzon, Mihaila, Milo)
  – Lorel (Stanford - Widom et al)

• Supported by modern languages
  – SparQL, Cypher, Gremlin, GSQL
Path Expression Examples (1)

• Pairs of customer and product they bought:

  \(-Bought->\)

• Pairs of customer and product they were involved with (bought or reviewed)

  \(-Bought|Reviewed->\)

• Pairs of customers who bought same product (lists customers with themselves)

  \(-Bought->.<-Bought-\)
Path Expression Examples (2)

- Pairs of customers involved with same product (like-minded)

  -$Bought|Reviewed->$.-$<Bought|Reviewed>-$

- Pairs of customers connected via a chain of like-minded customer pairs

  ($-Bought|Reviewed->.$<-Bought|Reviewed-)$*)
• Path expressions as atomic building blocks

• Explicitly introduce variables binding to source and target nodes of path expressions.

• Variables can be used to stitch multiple path expression atoms into complex patterns.
CRPQ Examples

• Pairs of customers who have bought same product (do not list a customer with herself):

   \[ Q1(c1,c2) :- c1 \neg \text{Bought}->.\neg\text{Bought}-\ c2, c1 \neq c2 \]

• Customers who have bought a product and also reviewed it:

   \[ Q2(c) :- c \neg \text{Bought}-> p, c \neg \text{Reviewed}-> p \]
Key Language Ingredients Needed in Modern Applications

– All primitives inherited from past
  • path expressions + variables + conjunctive patterns + node/edge construction

&

– Support for large-scale graph analytics
  • Aggregation of data encountered during navigation
    → requires bag semantics for pattern matches
  • Control flow support for class of iterative algorithms that converge in multiple steps
    – (e.g. PageRank-class, recommender systems, shortest paths, etc.)
Aggregation
Aggregation in Modern Graph QLs

• PGQL, Gremlin and SparQL use an SQL-style GROUP BY clause

• Cypher’s RETURN clause uses similar syntax as aggregation-extended CQs

• GSQL uses aggregating containers called “accumulators”
  – (soon to add above solutions as syntactic sugar, but accumulators remain strictly more versatile)
GSQQL Accumulators

- GSQL traversals collect and aggregate data by writing it into *accumulators*

- Accumulators are containers (data types) that
  - hold a data value
  - accept inputs
  - aggregate inputs into the data value using a binary operator

- May be built-in (sum, max, min, etc.) or user-defined

- May be
  - global (a single container)
  - Vertex-attached (one container per vertex)
Vertex-Attached Accumulator Example: Revenue per Customer and per Product

- customer
- product
- thisSaleRevenue

- @cSales
- @pSales

- discount
- quantity
- price

- bought
Vertex-Attached Accumulator Example: Revenue per Customer and per Product
Vertex-Attached Accumulator Example: Revenue per Customer and per Product

SumAccum<float> @cSales, @pSales;

SELECT c
FROM Customer :c –(Bought :b)-> Product :p
ACCUM thisSaleRevenue = b.quantity*(1-b.discount)*p.price,
    c.@cSales += thisSaleRevenue,
    p.@pSales += thisSaleRevenue;

Groups are distributed, each node accumulates its own group

Same sale revenue contributes to two aggregations, each by distinct grouping criteria
Recommended Toys Ranked by Log-Cosine Similarity

\[
\text{SumAccum}<\text{float}> @\text{rank}, @\text{lc}; \\
\text{SumAccum}<\text{int}> @\text{inCommon}; \\
\]

\[
\text{Me} = \{\text{Customer. 1}\}; \\
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{p} \text{ INTO ToysILike, } \text{o} \text{ INTO OthersWhoLikeThem} \\
\text{FROM} & \quad \text{Me: c -(Likes)-> Product: p <-(Likes)- Customer: o} \\
\text{WHERE} & \quad \text{p.category} == \text{“Toys” and o != c} \\
\text{ACCUM} & \quad \text{o.@inCommon += 1} \\
\text{POST-ACCUM} & \quad \text{o.@lc = log (1 + o.@inCommon)};
\end{align*}
\]

\[
\text{ToysTheyLike} = \quad \begin{align*}
\text{SELECT} & \quad \text{t} \\
\text{FROM} & \quad \text{OthersWhoLikeThem: o -(Likes)-> Product: t} \\
\text{WHERE} & \quad \text{t.category} == \text{”toy”} \\
\text{ACCUM} & \quad \text{t.@rank += o.@lc};
\end{align*}
\]

\[
\text{RecommendedToys} = \text{ToysTheyLike} - \text{ToysILike};
\]
Control Flow Primitives
Loops Are Essential

• Loops (until condition is satisfied)

  – Necessary to program iterative algorithms, e.g. PageRank, recommender systems, shortest-path, etc.

  – They synergize with accumulators. This GSQL-unique combination concisely expresses sophisticated graph analytics

  – Can be used to program unbounded-length path traversal under various semantics
CREATE QUERY pageRank (float maxChange, int maxIteration, float dampingFactor) {

MaxAccum<float> @@maxDifference = 9999; // max score change in an iteration
SumAccum<float> @received_score = 0;   // sum of scores received from neighbors
SumAccum<float> @score = 1;           // initial score for every vertex is 1.

AllV = {Page.*};                      // start with all vertices of type Page

WHILE @@maxDifference > maxChange LIMIT maxIteration DO
@maxDifference = 0;

S= SELECT s
       FROM AllV:s -(Linkto)-> :t
       ACCUM t.@received_score += s.@score/s.outdegree()
       POST-ACCUM s.@score = 1-dampingFactor + dampingFactor * s.@received_score,
                      s.@received_score = 0,
                     @@maxDifference += abs(s.@score - s.@score');

END;
}

Takeaway

Serendipitous synergy of

flexible aggregation + loops

from point of view of both

expressive power (conciseness, naturalness) performance