Query Languages for Unrestricted Graph Data

Alin Deutsch

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: special case XML (graph restricted to tree shape)
- 2000s: JSON and friends (also tree shaped)
- ~2010 to present: back to unrestricted graphs
  - Initially motivated by analytic tasks in social networks,
  - Now universal use (data is linked in all scenarios)

The Unrestricted Graph Data Model

- Nodes correspond to entities
- Edges are binary, correspond to relationships
- Edges may be directed or undirected
- Nodes and edges may carry labels
- Nodes and edges annotated with data
  - both have sets of attributes (key-value pairs)
- A schema is not required to formulate queries

Example Graph

Vertex types:

- Product (name, category, price)
- Customer (ssn, name, address)

Edge types:

- Bought (discount, quantity)
- Customer c bought 100 units of product p at discount 5%:

modeled by edge

c -- (Bought {discount=5%, quantity=100})→ p

Expressing Graph Analytics

- Two Different Approaches
  - High-level query languages à la SQL
  - Low-level programming abstractions
    - Programs written in C++, Java, Scala, Groovy…
- Initially adopted by disjoint communities (recall NoSQL debates)
- Recent trend towards unification

High-Level Query Languages

Some Modern Graph QLs We Will Discuss

There is a host of them! Spectrum includes

- Datalog with aggregation (LogicBlox)
- Cypher (neo4j)
  - declarative, highly similar to StruQL (and hence CRPQs)
- Gremlin (Apache and commercial projects)
  - dataflow programming model: graph annotated with tokens ("traversers") that flow through it according to user program
- New arrival: GSQL (TigerGraph)
  - Inspired by SQL + BSP, extended for more flexible grouping/ aggregation

Key Ingredients for High-Level Query Languages

- Pioneered by academic work on Conjunctive Query (CQ) extensions for graphs (since ‘87)
  - Path expressions (PEs) for navigation
  - Variables for manipulating data found during navigation
  - Stitching multiple PEs into complex navigation patterns → conjunctive regular path queries (CRPQs)

- Beyond CRPQs, needed in modern applications:
  - Aggregation of data encountered during navigation
    ➔ support for bag semantics as prerequisite
  - Intermediate results assigned to nodes/edges
  - Control flow support for class of iterative algorithms that converge to result in multiple steps
    - (e.g. PageRank-class, recommender systems, shortest paths, etc.)

Path Expressions

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 (Mendelzon, Mihaila, Milo)
  - Lorel (Widom et al)
- Today supported by languages of commercial systems
  - Cypher, SparQL, Gremlin, GSQL

Path Expression Syntax

Notations vary. Adopting here that of SparQL W3C Recommendation.

```
path → edge label
        |   _                   // wildcard, any edge label
        |   ^ edge label        //  inverse edge
        |   path . path         // concatenation
        |   path | path         // alternation
        |   path*               // 0 or more reps
        |   path*(min,max)      // at least min, at most max
        |   (path)
```

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 Expression Semantics

- In most academic research, the semantics are defined in terms of *sets of node pairs*
- Traditionally specified in two ways:
  - Declaratively, based on satisfaction of formulae/ patterns
  - Procedurally, based on algebraic operations over relations
- These are equivalent

Classical Declarative Semantics

- Given:
  - graph G
  - path expression PE

- the meaning of PE on G, denoted PE(G) is

  the set of node pairs (src, tgt)
  s.t. there exists a path in G from src to tgt
  whose concatenated labels spell out a word in L(PE)

L(PE) = language accepted by PE when seen as regular expression over alphabet of edge labels

Classical Procedural Semantics

PE(G) is a binary relation over nodes, defined inductively as:

- E(G) = set of s-t node pairs of E edges in G
- _(G) = set of s-t node pairs of any edges in G
- ^E(G) = set of t-s node pairs of E edges in G
- P1.P2(G) = P1(G) o P2(G)
- P1|P2(G) = set union (P1(G), P2(G))
- P*(G) = reflexive transitive closure of P(G)

relational composition

finite due to saturation

Conjunctive Regular Path Queries

- Replace relational atoms appearing in CQs with path expressions.
- Explicitly introduce variables binding to source and target nodes of path expressions.
- Allow multiple path expression atoms in query body.
- 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 –Bought.^Bought-> c2, c1 != c2

- Customers who have bought and also reviewed a product:

Q2(c) :- c –Bought-> p, c –Reviewed-> p

CRPQ Semantics

- Naturally extended from single path expressions, following model of CQs
- Declarative
  - lifting the notion of satisfaction of a path expression atom by a source-target node pair to the notion of satisfaction of a conjunction of atoms by a tuple
- Procedural
  - based on SPRJ manipulation of the binary relations yielded by the individual path expression atoms

Limitation of Set Semantics

- Common graph analytics need to aggregate data
  - e.g. count the number of products two customers have in common
- Set semantics does not suffice
  - baked-in duplicate elimination affects the aggregation
- As in SQL, practical systems resort to bag semantics

Path Expressions Under Bag Semantics

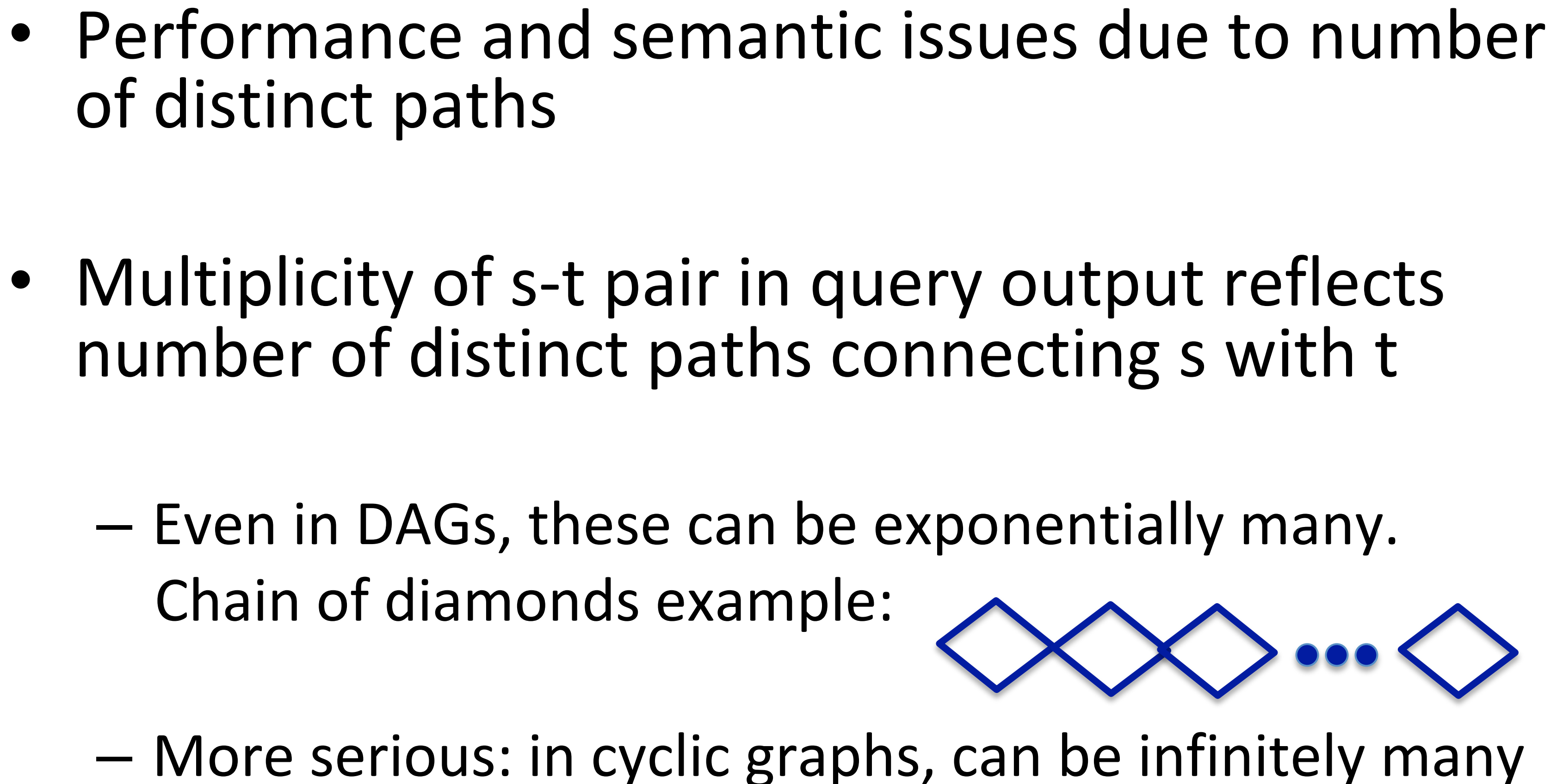
PE(G) is a bag of node pairs, defined inductively as:

- E(G) = ~~set~~ ***bag*** of s-t node pairs of E edges in G
- _(G) = ~~set~~ ***bag*** of s-t node pairs of any edges in G
- ^E(G) = ~~set~~ ***bag*** of t-s node pairs of E edges in G
- P1.P2(G) = P1(G) o P2(G)
- P1|P2(G) = ~~set~~ ***bag*** union (P1(G), P2(G))
- P*(G) = ~~reflexive transitive closure~~ of P(G)

relational composition for ***bags***

Not necessarily finite under bag semantics!

Issues with Bag Semantics



Solutions In Practice: Bound Traversal Length

- Upper-bound the length of the traversed path
  - Recall bounded Kleene construct *(min,max)
  - Bounds length and hence number of distinct paths considered
  - Supported by Gremlin, Cypher, SparQL, GSQL, very common in tutorial examples and in industrial practice

Solutions In Practice: Restrict Cycle Traversal

- No repeating vertices (simple paths)
  - Rules out paths that go around cycles
  - Recommended in Gremlin style guides, tutorials, formal semantics paper
  - Gremlin’s simplePath () predicate supports this semantics
  - Problem: membership of s-t pair in result is NP-hard

- No repeating edges
  - Allows cyclic paths
  - Rules out paths that go around same cycle more than once
  - This is the Cypher semantics

Solutions In Practice: Mix Bag and Set Semantics

- Bag semantics for star-free fragments of PE
- Set semantics for Kleene-starred fragments of PE
- Combine them using (bag-aware) joins

- Example:

  p1.p2*.p3(G)

  treated as

  p1(G) o (distinct (***p2****(G))) o ***p3***(G)

- This is the SparQL semantics (in W3C Recommendation)

Solutions In Practice: Leave it to User

- User explicitly programs desired semantics
- Path is first-class citizen, can be mentioned in query
- Can simulate each of the above semantics, e.g. by checking the path for repeated nodes/edges
- Could lead to infinite traversals for buggy programs
- Supported by Gremlin, GSQL
  - also partially by Cypher (modulo restriction that only edge non-repeating paths are visible)

One Semantics I Would Prefer

- Allow paths to go around cycles, even multiple times
- Achieve finiteness by restriction to *pumping-minimal* paths
  - in the sense of Pumping Lemma for Finite State Automata (FSA)
  - PE are regular expressions, they have an equivalent FSA representation (unique up to minimization)
  - As path is traversed, FSA state changes at every step
  - Rule out paths in which a vertex is repeatedly reached in the same FSA state
- Can be programmed by user in Gremlin and GSQL (costly!)

A Tractable Semantics: Shortest Paths

- For pattern

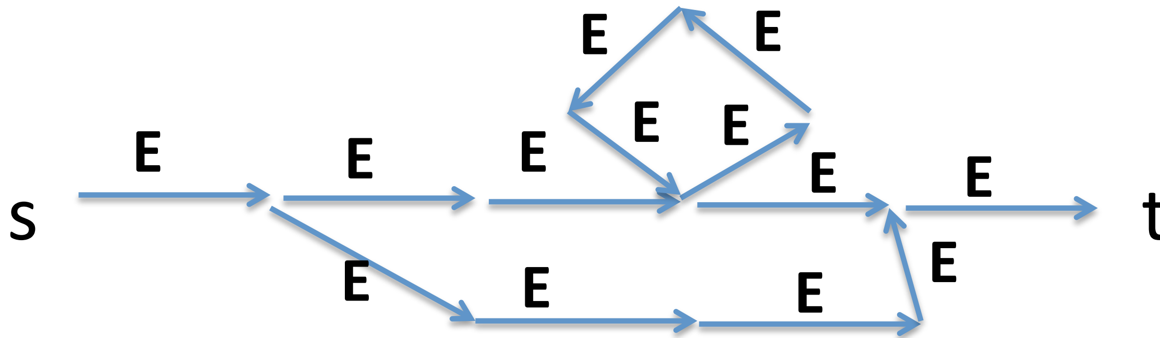
  x –Pattern-> y,

  vertex pair (s,t) is an answer iff there is a path p from s to t s.t.
  - word spelled by edge labels of p is in L(Pattern)
  - p is *shortest* among all such paths from s to t

- Multiplicity of (s,t) in answer is the count of such shortest paths

Contrasting Semantics

- pattern E* over graph:



- s-t is an answer under all semantics, but
  - Simple-path: s-t has multiplicity 2
  - Unique-edge: s-t has multiplicity 3
  - Shortest-path: s-t has multiplicity 1

Aggregation

Let’s See it First as CQ Extension

- Count toys bought in common per customer pair

Q(c1, c2, count (p)) :- c1 –Bought-> p, c2 –Bought-> p,
p.category = “toys”, c1 < c2

- c1, c2: composite group key - no explicit group-by clause
- Standard syntax for aggregation-extended CQs and Datalog
- Rich literature on semantics
  - (tricky for Datalog when aggregation and recursion interleave).

Aggregation in Modern Graph QLs

- Cypher's RETURN clause uses similar syntax as aggregation-extended CQs
- Gremlin and SparQL use an SQL-style GROUP BY clause
- GSQL uses aggregating containers called "accumulators"

Flavor of Representative Languages

Running Example in CRPQ Form

- Recall:

  count toys bought in common per customer pair

Q(c1, c2, count (p)) :- c1 –Bought-> p, c2 –Bought-> p,
p.category = “toys”, c1 < c2

SparQL

- Query language for the semantic web
  - graphs corresponding to RDF data are directed, labeled graphs
- W3C Standard Recommendation

Running Example in SparQL

```
SELECT   ?c1, ?c2, count (?p)

WHERE   { ?c1 bought ?p.
           ?c2 bought ?p.
           ?p  category ?cat.

           FILTER (?cat == “toys” && ?c1 < ?c2) }

GROUP BY ?c1, ?c2
```

SparQL Semantics by Example

- Coincides with CRPQ version

Q(c1, c2, count (p)) :- c1 –Bought-> p, c2 –Bought-> p,
p.category = “toys”, c1 < c2

Cypher

- The query language of the neo4j commercial native graph db system
- Essentially StruQL with some bells and whistles
- Also supported in a variety of other systems:
  - SAP HANA Graph, Agens Graph, Redis Graph, Memgraph, CAPS (Cypher for Apache Spark), ingraph, Gradoop, Ruruki, Graphflow

Running Example in Cypher

```
MATCH   (c1:Customer) –[:Bought]-> (p:Product)
          <-[:Bought]- (c2:Customer)

WHERE   p.category = “Toys” AND c1.name < c2.name

RETURN  c1.name AS cust1,
        c2.name AS cust2,
        COUNT (p)  AS inCommon
```

c1.name, c2.name are composite group key
– no explicit group-by clause, just like CQ

Cypher Semantics by Example

- Coincides with CRPQ version

Q(c1, c2, count (p)) :- c1 –Bought-> p, c2 –Bought-> p, c1 < c2

- Modulo non-repeating edge restriction
  - no effect here since repeated-edge paths satisfying the two PE atoms would necessarily have c1 = c2

Gremlin

- Supported by major Apache projects
  - TinkerPop and JanusGraph

- Also by commercial systems including
  - TitanGraph (DataStax)
  - Neptune (Amazon),
  - Azure (Microsoft),
  - IBM Graph

Gremlin Semantics

- Based on ***traversers***, i.e. tokens that flow through graph binding variables along the way

- A Gremlin program adorns the graph with a set of traversers that co-exist simultaneously

- A program is a pipeline of steps, each step works on the set of traversers whose state corresponds to this step

- Steps can be
  - map steps (work in parallel on individual traversers)
  - reduce steps (aggregate set of traversers into a single traverser)

Gremlin Semantics by Example

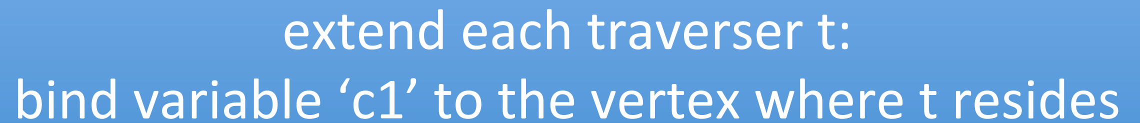
V()

Gremlin Semantics by Example

V().**hasLabel**('Customer')

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')



Gremlin Semantics by Example

V().**hasLabel**(‘Customer’).**as**(‘c1’)

.**out**(‘Bought’)

Traversers flow along out-edges of type ‘Bought’.

If multiple such edges emanate from a Customer vertex v, the traverser at v ***splits*** into one copy per edge, placed at edge destination.

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')

.**out**('Bought').**hasLabel**('Product').**has**('category','Toys')

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')

.**out**('Bought').**hasLabel**('Product').**has**('category','Toys').**as**('p')

extend surviving traversers with binding of variable 'p' to their location vertex.

now each surviving traverser has two variable bindings: c1, p

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')
.**out**('Bought').**hasLabel**('Product').**has**('category','Toys').**as**('p')
.**in**('Bought')

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')
  .**out**('Bought').**hasLabel**('Product').**has**('category','Toys').**as**('p')
  .**in**('Bought').**hasLabel**('Customer').**as**('c2')

.**select** ('c1', 'c2','p').**by**('name')

Gremlin Semantics by Example

V().**hasLabel**('Customer').**as**('c1')

.**out**('Bought').**hasLabel**('Product').**has**('category','Toys').**as**('p')

.**in**('Bought').**hasLabel**('Customer').**as**('c2')

.**select** ('c1', 'c2','p').**by**('name')

.**where** ('c1', **lt**('c2'))

Gremlin Semantics by Example

```
V().hasLabel(‘Customer’).as(‘c1’)
    .out(‘Bought’).hasLabel(‘Product’).has(‘category’,’Toys’).as(‘p’)
    .in(‘Bought’).hasLabel(‘Customer’).as(‘c2’)

.select (‘c1’, ‘c2’,‘p’).by(‘name’)
.where (‘c1’, lt(‘c2’))

.group().by(select(‘c1’,’c2’)).by(count())
```

group tuples

first by() specifies group key

second by() specifies group aggregation

GSQL

- The query language of TigerGraph, a native parallel graph db system

- A recent start-up founded by UCSD DB lab's PhD alum Yu Xu

- Full disclosure: I have been involved in design

GSQL 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 operation

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

- May be
  - global (a single container accessible from all traversal steps)
  - local (one per node, accessible only when reached by traversal)

Running Example in GSQL

```
GroupByAccum  <string cust1, string cust2,
                  SumAccum<int> inCommon>    @@res;

SELECT    _
FROM      Customer:c1 -(Bought>)- Product:p –(<Bought)- Customer:c2
WHERE     p.category == “Toys”  AND  c1.name < c2.name
ACCUM    @@res += (c1.name, c2.name -> 1);
```

Global accum

cust1, cust2 form composite group key

inCommon is group value (a sum aggregation)

aggregate this input into accumulator

create input associating value 1 to key (c1.name, c2.name)

GSQL Semantics by Example

```
GroupByAccum  <string cust1, string cust2,
                    SumAccum<int> inCommon>    @@res;

SELECT    _
FROM      Customer:c1 -(Bought>)- Product:p −(<Bought)- Customer:c2
WHERE     p.category == "Toys"  AND  c1.name < c2.name
ACCUM     @@res += (c1.name, c2.name -> 1);
```

For every distinct path satisfying FROM pattern and WHERE condition…

…execute ACCUM clause

Why Aggregate in Accumulators Instead of Select-Group By Clauses?

```
GroupByAccum  <string cust, SumAccum<float> total>  @@cSales;
GroupByAccum  <string prod, SumAccum<float> total>  @@pSales;

SELECT     _
FROM       Customer:c -(Bought>:b)- Product:p
ACCUM      float thisSalesRevenue = b.quantity*(1-b.discount)*p.price,
           @@cSales += (c.name -> thisSalesRevenue),
           @@pSales += (p.name -> thisSalesRevenue);
```

revenue per customer

revenue per product

local variable, this is a let clause

multiple aggregations in one pass, even on different group keys

Local Accumulators

- Minimize bottlenecks due to shared global accums, maximize opportunities for parallel evaluation

```
SumAccum<float> @cSales, @pSales;

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

local accums, one instance per node

groups are distributed, each node accumulates its own group

Role of SELECT Clause? Compositionality

- queries can output set of nodes, stored in variables
- used by subsequent queries as traversal starting point:

```
S1 = SELECT t
       FROM S0:s – pattern1 – T1:t
       WHERE … ACCUM …

S2 = SELECT t
       FROM S1:s – pattern2 – T2:t …
       WHERE … ACCUM …

S3 = SELECT t
       FROM S1:s – pattern3 – T3:t …
       WHERE … ACCUM …
```

Variable S1 stores set of nodes reached in traversal

S1 used in subsequent traversals (query chaining)

Recommended Toys Ranked by Log-Cosine Similarity

```
SumAccum<float> @rank, @lc;
SumAccum<int>    @inCommon;

I = {Customer.1};
ToysILike, OthersWhoLikeThem =
      SELECT        p, o
      FROM          I:c -(Likes>)- Product:p  -(<Likes)- Customer:o
      WHERE         p.category == "Toys" and o != c
      ACCUM         o.@inCommon += 1
      POST-ACCUM    o.@lc = log (1 + o.@inCommon);

ToysTheyLike =   SELECT    t
                 FROM      OthersWhoLikeThem:o –(Likes>)- Product:t
                 WHERE     t.category ==  "toy"
                 ACCUM     t.@rank += o.@lc;

RecommendedToys = ToysTheyLike – ToysILike;
```

Control Flow Primitives

Loops Are Essential

- Loops (until condition is satisfied)
  - Explicitly supported in Gremlin and GSQL
  - Necessary to program iterative algorithms like PageRank, recommender systems, shortest-path, etc.
  - Can be used to program match of Kleene-starred path expressions under various semantics
- If-then-else, case constructs
  - Supported by all QLs in some way

PageRank in GSQL

```
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;
}
```

Low-level, NoSQL-style Programming for Parallel Graph Analytics

Think-Like-a-Vertex (TLAV) aka Vertex-Centric

- Parallel computing abstraction
- Conceptually, each vertex is a processor
- Vertices execute a vertex program in parallel
- Instances of vertex programs communicate via messages to neighbors
- Vertices typically execute in lockstep (via synchronization barriers)

Pregel: A TLAV Programming Abstraction

- Bulk-synchronous parallel computing abstraction

- Introduced by Pregel System (Google)
- Supported in open-source systems
  - e.g. Giraph (Apache), GraphX (Apache Spark)

- Pregel program executes in lockstep a series of supersteps

- During each superstep, vertices (in parallel)
  - receive inbound messages sent in previous superstep,
  - compute a new value for the vertex data
  - send messages to neighboring vertices (received in next superstep)

Gather-Apply-Scatter (GAS)

- Isomorphic with Pregel when vertices evaluate in lockstep
- Also supports asynchronous evaluation

- Introduced by GraphLab system (an open-source project)

- Each vertex program step is organized in three phases:
  - **Gather**: may directly access information from its one-hop neighborhood, aggregating it with user-defined function
  - **Apply**: vertex value is updated by incorporating this sum
  - **Scatter**: neighborhood values updated using result of apply phase

- Communication abstraction: shared memory, not messaging

PowerGraph

- Refinement of GAS abstraction to process ***edges*** in parallel
  - for load balancing in presence of high-degree vertices
- Gather phase executes a function that maps over edges
- Results of edge map are reduced by a user-defined Sum function
- Apply phase uses the reduced result
- Only edges incident on *active vertices* work.
  - Vertices can be explicitly activated during scatter phase.

GSQL's Edge-Map/Vertex-Reduce (EM/VR)

- Extends PowerGraph for flexibility
  - user can define *multiple* independent reducers via accumulators
  - accums can *be local or global*
  - accums are *first-class citizens*
    - persist across steps, can be mentioned by future steps
  - parallel map over edges generates accum inputs
  - reduce phase updates each accum value by aggregating all inputs into it

GSQL As High-level EM/VR Program

```
SumAccum<float> @cSales, @pSales;

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

local accums implement two independent reducers

SELECT clause specifies vertex to activate next

FROM clause filters edges undergoing map

ACCUM clause executes per edge, generates accum inputs

Summary

- We discussed representative high-level graph QLs
  - from point of view of expressive power and semantics
  - de-emphasizing syntax
- We have seen NoSQL-style low-level parallel graph programming abstractions
- No need to choose between high-level and low-level (false choice claimed by prior NoSQL-related debates)
  - abstraction levels can be harmonized (as shown for GSQL)

Topics Not Covered Here

- Creating/modifying vertices and edges
  - As opposed to just returning tables of variable bindings
- Non-scalar vertex and edge properties (these can be lists/arrays and other containers)
- Behavior when a vertex/edge property does not exist (options are comprehensively laid out in Part A on hierarchical graph model)
- Graph schemas