Aggregation Support for Modern Graph Analytics in TigerGraph

Alin Deutsch
UCSD faculty & TigerGraph Chief Scientist

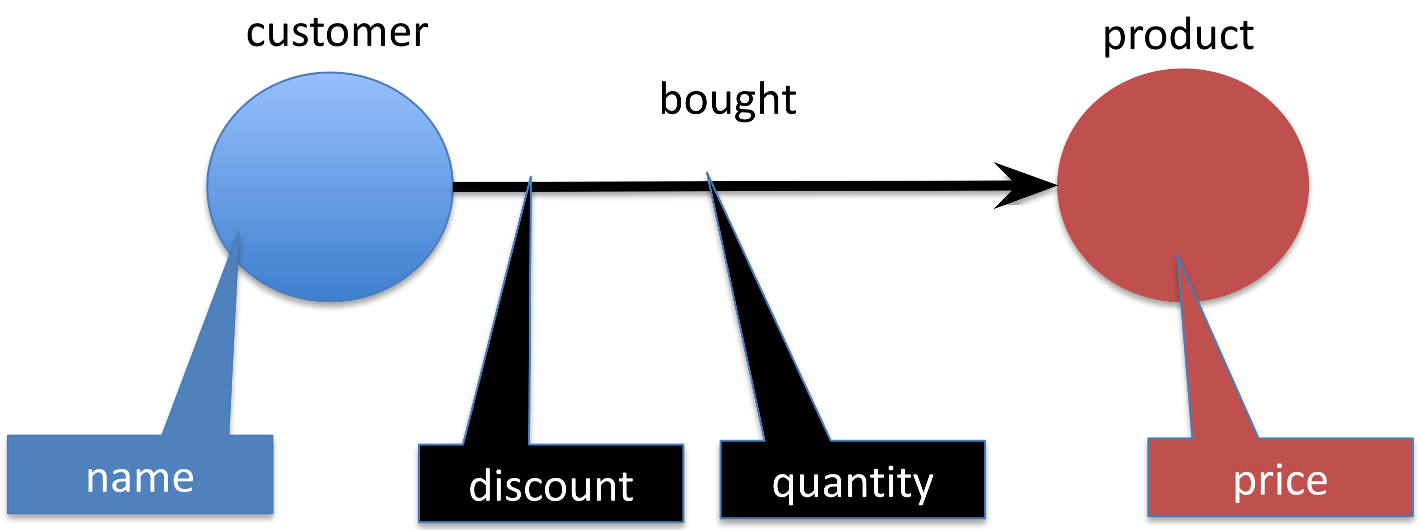
The Age of the Graph Is Upon Us (Again)

- Mid-Late 90s: semi-structured 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)
  - 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
  - Cypher, Gremlin, SparQL, more recently TigerGraph's GSQL
  - Two ANSI/ISO standards coming up: SQL/PGQ extension & GQL

The Traditional Graph Data Model

- Nodes correspond to entities
- Edges correspond to binary relationships
- Edges may be undirected or directed
  (modeling asymmetric, resp. symmetric relationships)
- Nodes and edges may be labeled/typed
- Nodes and edges annotated with data
  - both have sets of attributes, aka properties (key-value pairs)

Example: Customers Buy Products



Key Language Ingredients Required by Modern Applications

- All primitives inherited from classical academic work
  (first prototypes as early as 1987)
  - path expressions + variables + conjunctive patterns
    + node/edge construction (de facto standard, soon de jure)

  [not the focus of this talk]

&

- Support for large-scale graph analytics
  - Aggregation of data encountered during navigation
  - Control flow support for algorithms that iterate to convergence
    - PageRank-class, recommender systems, shortest paths, etc

  [this talk]

Aggregation

Aggregation in Modern Graph QLs

- Conventional (SQL-style):
  - Compute table of pattern matches, next partition it into groups
  - PGQL, Gremlin and SparQL use explicit GROUP BY clause
  - Cypher's implicit GROUP BY has same syntax as aggregation-extended conjunctive queries
- GSQL (TigerGraph's QL): alternate paradigm based on aggregating containers called "accumulators"
  - advantages for both naturality of specification and performance
  - (recently added conventional style as syntactic sugar, but accumulators remain strictly more versatile)

GSQL Accumulators

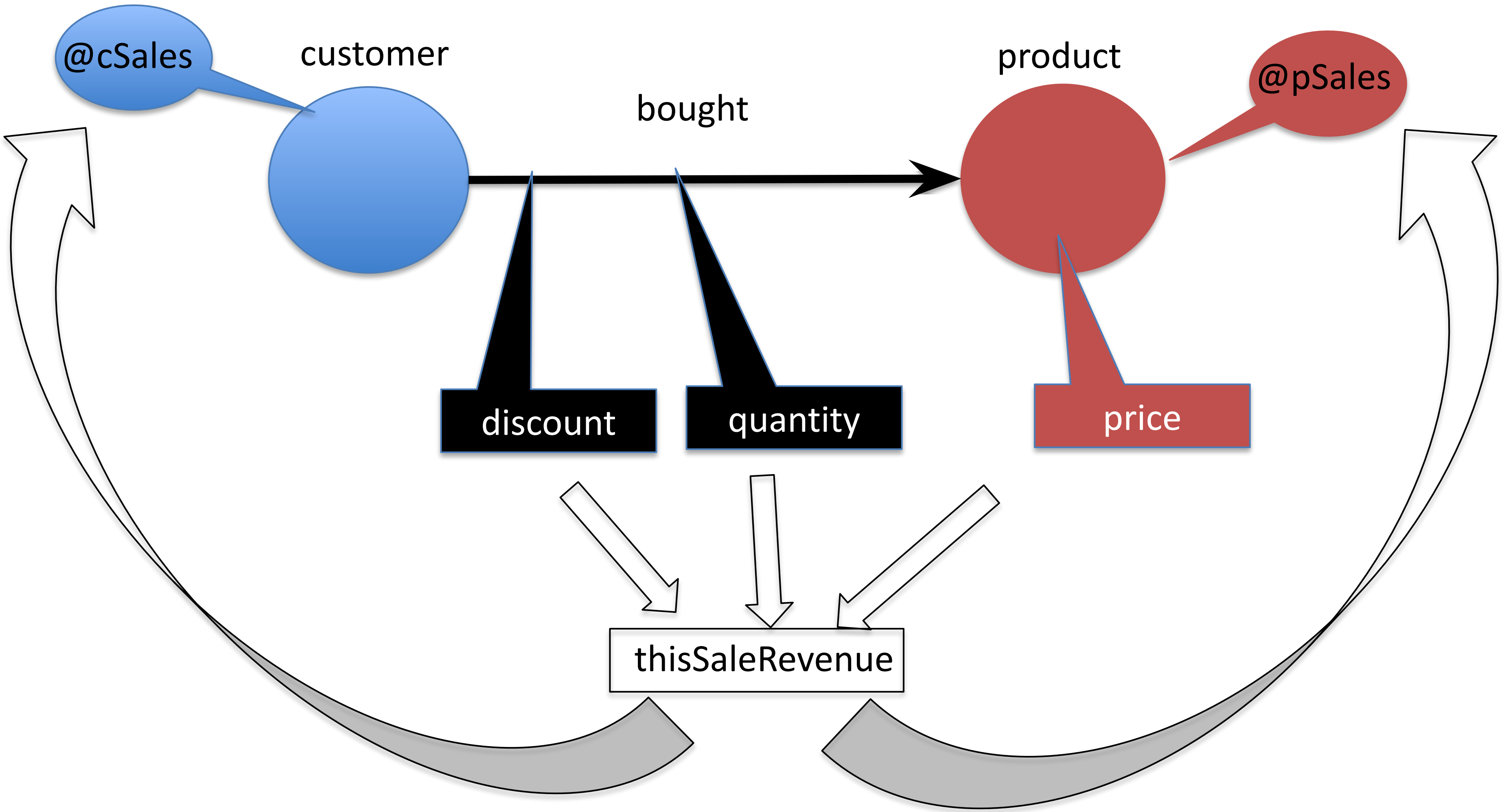
- GSQL traversals collect and aggregate data by writing it into *accumulators*

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

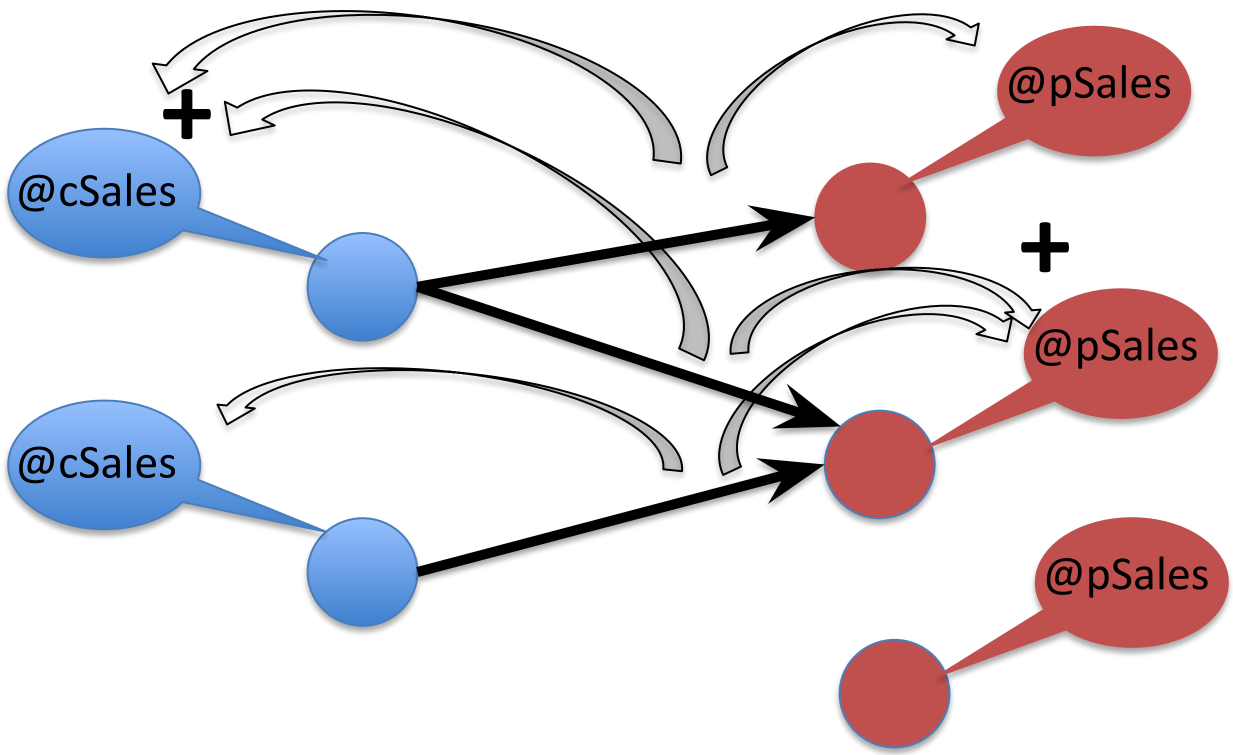
- May be built-in (sum, max, min, etc.) or user-defined

- May be
  - global (a single container per query)
  - vertex-attached (one container instance per vertex)

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



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;
```

accumulator declaration

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

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

Me = {Customer.1};

SELECT          p INTO ToysILike, o INTO OthersWhoLikeThem
FROM            Me: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);

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

RecommendedToys = ToysTheyLike – ToysILike;
```

Benefits of Accumulator-based Aggregation (Transcend Graph Model)

- It subsumes SQL-style aggregation
  - just implemented SQL's GROUP BY as syntactic sugar

- Specifies queries whose evaluation is naturally parallelizable

- Facilitates specification of single-pass multi-aggregation (by different grouping criteria)
  - currently unsupported in GQL 1.0 standard draft or other graph QLs
  - only partially supported even in SQL:
  - Its most sophisticated aggregation primitives (PARTITION OVER, CUBE, ROLLUP) result in wasteful aggregation (may compute more aggregates than user wants)
  - Experiments show up to 3x speedup of accumulator-based over conventional (SQL-style) aggregation (see SIGMOD 2020 paper)

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 algorithms
    - within the language!

      → no need to modify built-in algorithms programmed in Java/C++/Python…
  - Can be used to program unbounded-length path traversal under various semantics

PageRank in GSQL

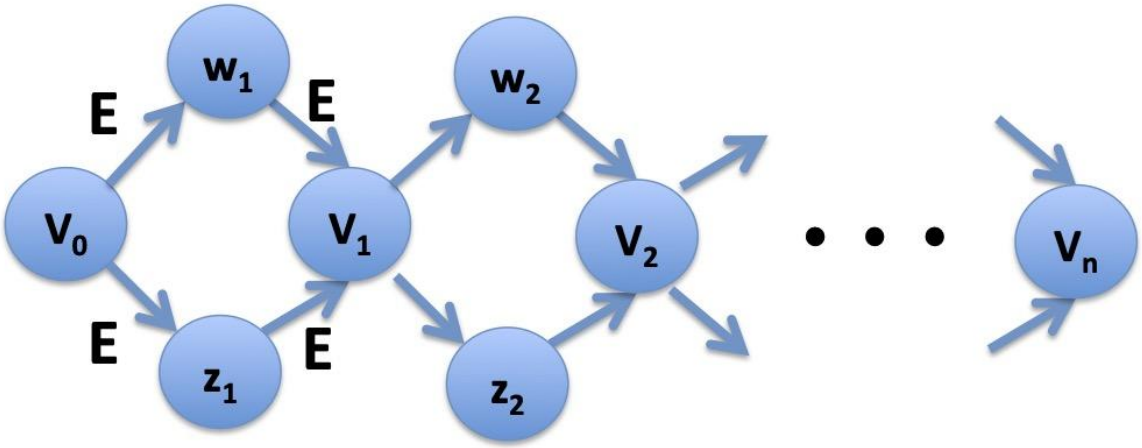
```
CREATE QUERY pageRank (float maxChange, int maxIteration, float dampingFactor) {

  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;
}
```

Exploring the Design Space for Aggregation Semantics

Aggregation Requires Bag Semantics, which Clashes with Finiteness

- Common graph analytics need to aggregate data
  - e.g. count the number of products two customers like in common
- Set semantics (the tradition in academic work) does not suffice
  - baked-in duplicate elimination affects the aggregation
- As in SQL, in practice systems resort to bag semantics
- BUT they encounter a new, graph QL-specific challenge:
  - Bag semantics clashes with finiteness of query answer
- Multiplicity of s-t pair in query output reflects number of distinct paths connecting s with t
  - Even in acyclic graphs, can be exponentially many (in the graph size!)
  - Worse: in cyclic graphs, can be infinitely many

The Chain-of-Diamonds Graph



Ensuring Finite Query Results in State of the Art: Restricting Legal Paths

- No restriction
  - non-terminating queries possible (Gremlin)
- No repeated nodes, aka simple paths (Gremlin tutorial examples)
  - Aggregation-friendly, intractable (existence of simple path is NP-hard)
- No repeated edges, aka trails (Cypher default semantics)
  - Aggregation-friendly, intractable
- Transitive closure patterns as Boolean reachability tests (SparQL)
  - Aggregation-unfriendly, tractable
- Shortest paths (TigerGraph default semantics)
  - Aggregation-friendly, tractable

Aggregation-Friendly but Intractable Designs: 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 intractable (NP-hard)

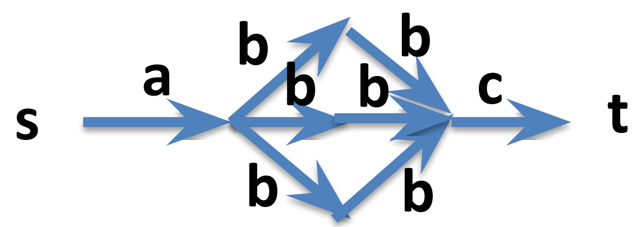
- No repeating edges (trails)
  - Allows cyclic paths
  - Rules out paths that go around same cycle more than once
  - This is the default Cypher semantics
  - Problem: membership of s-t pair in result still NP-hard

Tractable Yet Aggregation-Unfriendly: Mix Bag and Set Semantics

- Bag semantics for star-free fragments of PE
- Set semantics for Kleene-starred fragments of PE
- This is the semantics of the SparQL WC3 standard
- Tractable complexity but aggregation-unfriendly

- Example:

    a.b*.c



multiplicity of (s,t) in answer is 1, as if there were only one path connecting s to t

$\Rightarrow$ path counting, or aggregating data from the path meaningless

Aggregation-Friendly & Tractable: Shortest Paths

- For pattern

  x –(PE)-> y,

  vertex pair (s,t) is a match iff there is a path p from s to t such that

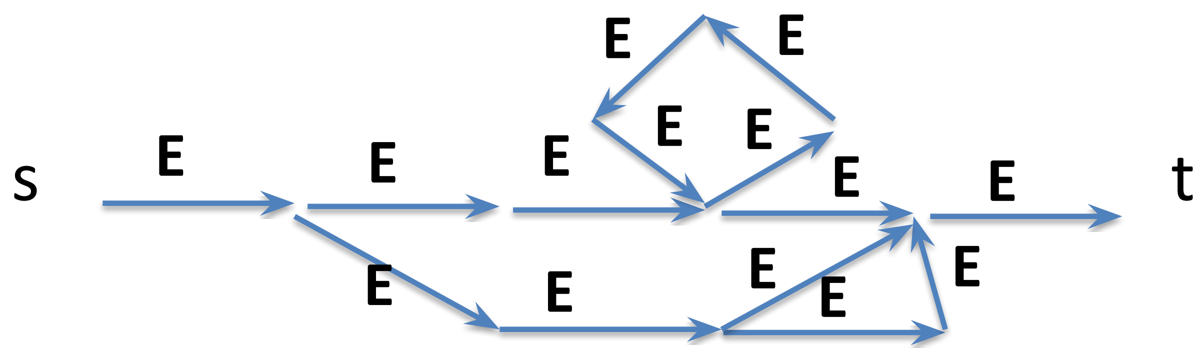
  – PE matches p, and
  – p is *shortest* among all matching paths from s to t

- Multiplicity of (s,t) in result is the count of all shortest paths

- Default semantics in GSQL (as of TG 2.4)

Contrasting Semantics

- pattern E* over graph:



- s-t is an answer under all semantics, but
  - Unrestricted paths: s-t has multiplicity infinite (Gremlin)
  - Simple-path: s-t has multiplicity 3 (Gremlin recommended)
  - Unique-edge: s-t has multiplicity 4 (Cypher)
  - Shortest-path: s-t has multiplicity 2 (GSQL)

Accumulators + Shortest Paths = Performance (Computational Complexity)

Two well-known facts:

- Can count shortest paths in polynomial time, even exponentially many, because no need to materialize them

- Same holds for paths satisfying a path expression

⇒ A key fragment of GSQL (covering a majority of TG's use cases) has PTIME data complexity

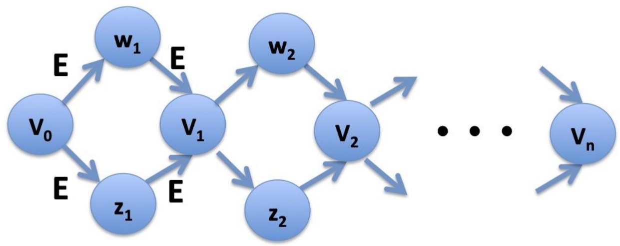
Restriction:

– do not bind variables to entire paths
– do not bind variables in scope of Kleene star
– do not use List and String accumulators

Proof sketch in SGMOD 2020 paper

Accumulators + Shortest Paths = Performance (Experiments)

- a family of DAGs with exponential number of paths between source and sink



- query counts these paths
- non-repeated edge and shortest-path semantics coincide
- increasing graph size, we measured running time and observed
  - exponential trend for non-repeated-edge evaluation
    - reference system for trail semantics Neo4j (timeout at 10 minutes for chain of 25 diamonds),
  - linear trend for shortest-path evaluation
    - TigerGraph (all runs within a few tens of ms)

Takeaway

- flexible aggregation via accumulators yields
  expressive power (conciseness, naturalness of specification)
  and
  performance (due to support for parallel one-pass multi-aggregation, and for iterative algorithms)

- accumulators + shortest-paths semantics
  yields large tractable GSQL fragment

Looking Ahead

- Due to its control primitives and accumulators, GSQL is Turing complete

- Will achieve conformance to standard by translating to GSQL

- Will continue to maintain a library of graph algorithms implemented in GSQL (standard GQL not expressive enough)
  => users can tweak them, no need to go to lower-level languages

- TigerGraph sits on both standard working groups and is an active contributor. Two-way street:
  - GSQL is influencing the standards and in turn it is evolving to align

Thank You!