GSQL 2.0: Seamless Querying of Relational and Graph Databases

Alin Deutsch Yu Xu Mingxi Wu Victor Lee

December 18, 2018

1 Introduction

TigerGraph is a graph database system built from the ground up to support massively parallel computation of queries and analytics. TigerGraph's high-level query language, GSQL, is designed for backwards compatibility with SQL, while simultaneously allowing NoSQL programmers to continue thinking in Bulk-Synchronous Processing (BSP) terms and reap the benefits of high-level specification. GSQL is sufficiently high-level to allow declarative SQL-style programming, yet sufficiently expressive to concisely specify the sophisticated iterative algorithms required by modern graph analytics and traditionally coded in general-purpose programming languages like C++ and Java.

2 GSQL DDL

Like SQL, GSQL is a strongly-typed language. This may seem surprising given that prior work on graph query languages traditionally touts schema freedom as a desirable feature (such data was not accidentally called "semi-structured"). However, this feature was historically motivated by the driving application at the time, namely integrating numerous heterogeneous, third-party-owned but individually relatively small data sources from the Web.

In contrast, TigerGraph targets enterprise applications, where the number and ownership of sources is not a concern, but their sheer size and resulting performance challenges are. In this setting, vertices and edges that model the same domain-specific entity tend to be uniformly structured and advance knowledge of their type is expected. Failing to exploit it for performance would be a missed opportunity.

GSQL's Data Definition Language (DDL) shares SQL's philosophy of defining in the same CREATE statement a persistent data container as well as its type. This is in contrast to typical programming languages which define types as stand-alone entities, separate from their instantiations.

GSQL's CREATE statements can define vertex containers, edge containers, and graphs consisting of these containers. The attributes for the vertices/edges populating the container are declared using syntax borrowed from SQL's CREATE TABLE command. In TigerGraph's model, a graph may contain both directed and undirected edges. The same vertex/edge container may be shared by multiple graphs.

Example 1 (DDL) *Consider the following DDL statements declaring two graphs: LinkedIn and Twitter. Notice how symmetric relationships (such as LinkedIn connections) are modeled as undirected edges, while asymmetric relationships (such as Twitter's following or posting) correspond to directed edges. Edge types specify the types of source and target vertices, as well as optional edge attributes (see the* `since` *and* `end` *attributes below).*

For vertices, one can declare primary key attributes with the same meaning as in SQL (see the `email` *attribute of Person vertices).*

```
CREATE VERTEX Person
   (email STRING PRIMARY KEY, name STRING, dob DATE)

CREATE VERTEX Tweet
   (id INT PRIMARY KEY, text STRING, timestamp  DATE)

CREATE DIRECTED EDGE Posts
   (FROM Person, TO Tweet)
```

```
CREATE DIRECTED EDGE Follows
   (FROM Person, TO Person, since DATE)

CREATE UNDIRECTED EDGE Connected
   (FROM Person, TO Person, since DATE, end DATE)

CREATE GRAPH LinkedInGraph
   (Person, Connected)
CREATE GRAPH TwitterGraph
   (Person, Tweet, Posts, Follows)
```

□

By default, the primary key of an edge type is the composite key comprising the primary keys of its endpoint vertex types.

Edge Discriminators. Multiple parallel edges of the same type between the same endpoints are allowed. To distinguish among them, one can declare *discriminator attributes* which complete the pair of endpoint vertex keys to uniquely identify the edge. This is analogous to the concept of weak entity set discriminators in the Entity-Relationship Model [5]. For instance, one could use the dates of employment to discriminate between multiple edges modeling recurring employments of a LinkedIn user at the same company.

```
CREATE DIRECTED EDGE Employment
   (FROM Company, TO Person, start DATE, end DATE)
   DISCRIMINATOR (start)
```

Reverse Edges. The GSQL data model includes the concept of edges being inverses of each other, analogous to the notion of inverse relationships from the ODMG ODL standard [4].

Consider a graph of fund transfers between bank accounts, with a directed Debit edge from account A to account B signifying the debiting of A in favor of B (the amount and timestamp would be modeled as edge attributes). The debiting action corresponds to a crediting action in the opposite sense, from B to A. If the application needs to explicitly keep track of both credit and debit vocabulary terms, a natural modeling consists in introducing for each Debit edge a reverse Credit edge for the same endpoints, with both edges sharing the values of the attributes, as in the following example:

```
CREATE VERTEX Account
   (number int PRIMARY KEY, balance FLOAT, ...)

CREATE DIRECTED EDGE Debit
   (FROM Account, TO Account, amount float, ...)
   WITH REVERSE EDGE Credit
```

3 GSQL DML

The guiding principle behind the design of GSQL was to facilitate adoption by SQL programmers while simultaneously flattening the learning curve for novices, especially for adopters of the BSP programming paradigm [17].

To this end, GSQL's design starts from SQL, extending its syntax and semantics parsimoniously, i.e. avoiding the introduction of new keywords for concepts that already have an SQL counterpart. We first summarize the key additional primitives before detailing them.

Graph Patterns in the FROM Clause. GSQL extends SQL's FROM clause to allow the specification of *patterns*. Patterns specify constraints on paths in the graph, and they also contain variables, bound to vertices or edges of the matching paths. In the remaining query clauses, these variables are treated just like standard SQL tuple variables.

Accumulators. The data found along a path matched by a pattern can be collected and aggregated into *accumulators*. Accumulators support multiple simultaneous aggregations of the same data according to distinct grouping criteria. The aggregation results can be distributed across vertices, to support multi-pass and, in conjunction with loops, even iterative graph algorithms implemented in MPP fashion.

```
SELECT e.email, e.name, count (outsider)
FROM   Employee AS e,
       LinkedIn AS Person: p -(Connected: c)- Person: outsider
WHERE  e.email = p.email and
       outsider.currentCompany NOT LIKE 'ACME' and
       c.since >= 2016
GROUP BY e.email, e.name
```

Figure 1: Query for Example 2, Joining Relational Table and Graph

Loops. GSQL includes control flow primitives, in particular loops, which are essential to support standard iterative graph analytics (e.g. PageRank [2], shortest-paths [7], weakly connected components [7], recommender systems, etc.).

Direction-Aware Regular Path Expressions (DARPEs).
GSQL's FROM clause patterns contain path expressions that specify constrained reachability queries in the graph. GSQL path expressions start from the de facto standard of two-way regular path expressions [3] which is the culmination of a long line of works on graph query languages, including reference languages like WebSQL [13], StruQL [6] and Lorel [1]. Since two-way regular path expressions were developed for directed graphs, GSQL extends them to support both directed and undirected edges in the same graph. We call the resulting path expressions *Direction-Aware Regular Path Expressions (DARPEs)*.

3.1 Graph Patterns in the FROM Clause

GSQL's FROM clause extends SQL's basic FROM clause syntax to also allow atoms of general form

<GraphName> AS? <pattern>

where the AS keyword is optional, <GraphName> is the name of a graph, and ¡pattern¿ is a pattern given by a regular path expression with variables.

This is in analogy to standard SQL, in which a FROM clause atom

<TableName> AS? <Alias>

specifies a collection (a bag of tuples) to the left of the AS keyword and introduces an alias to the right. This alias can be viewed as a simple pattern that introduces a single tuple variable. In the graph setting, the collection is the graph and the pattern may introduce several variables. We show more complex patterns in Section 3.4 but illustrate first with the following simple-pattern example.

Example 2 (Seamless Querying of Graphs and Relational Tables) *Assume Company ACME maintains a human resource database stored in an RDBMS containing a relational table "Employee". It also has access to the "LinkedIn" graph from Example 1 containing the professional network of LinkedIn users.*

The query in Figure 1 joins relational HR employee data with LinkedIn graph data to find the employees who have made the most LinkedIn connections outside the company since 2016:

Notice the pattern Person:p -(Connected:c)- Person:outsider to be matched against the "LinkedIn" graph. The pattern variables are "p", "c" and "outsider", binding respectively to a "Person" vertex, a "Connected" edge and a "Person" vertex. Once the pattern is matched, its variables can be used just like standard SQL tuple aliases. Notice that neither the WHERE clause nor the SELECT clause syntax discriminate among aliases, regardless of whether they range over tuples, vertices or edges.

The lack of an arrowhead accompanying the edge subpattern -(Connected: c)- requires the matched "Connected" edge to be undirected. □

To support cross-graph joins, the FROM clause allows the mention of multiple graphs, analogously to how the standard SQL FROM clause may mention multiple tables.

```
SELECT e.email, e.name, e.salary, count (other), count (t)
FROM   Employee AS e,
       LinkedIn AS Person: p -(Connected)- Person: other,
       Twitter AS User: u -(Posts>)- Tweet: t
WHERE  e.email = p.email and p.email = u.email and
       t.text CONTAINS e.company
GROUP BY e.email, e.name, e.salary
```

Figure 2: Query for Example 3, Joining Across Two Graphs

Example 3 (Cross-Graph and -Table Joins) *Assume we wish to gather information on employees, including how many tweets about their company and how many LinkedIn connections they have. The employee info resides in a relational table "Employee", the LinkedIn data is in the graph named "LinkedIn" and the tweets are in the graph named "Twitter". The query is shown in Figure 2. Notice the join across the two graphs and the relational table.*

Also notice the arrowhead in the edge subpattern ranging over the Twitter graph, $-(Posts >)-$, which matches only directed edges of type "Posts", pointing from the "User" vertex to the "Tweet" vertex. □

3.2 Accumulators

We next introduce the concept of accumulators , i.e. data containers that store an internal value and take inputs that are aggregated into this internal value using a binary operation. Accumulators support the concise specification of multiple simultaneous aggregations by distinct grouping criteria, and the computation of vertex-stored side effects to support multipass and iterative algorithms.

The accumulator abstraction was introduced in the GreenMarl system [9] and it was adapted as high-level first-class citizen in GSQL to distinguish among two flavors:

- *Vertex accumulators* are attached to vertices, with each vertex storing its own local accumulator instance. They are useful in aggregating data encountered during the traversal of path patterns and in storing the result distributed over the visited vertices.
- *Global accumulators* have a single instance and are useful in computing global aggregates.

Accumulators are polymorphic, being parameterized by the type of the internal value V, the type of the inputs I, and the binary *combiner* operation

$$\oplus : V \times I \to V.$$

Accumulators implement two assignment operators. Denoting with $a.\texttt{val}$ the internal value of accumulator a,

- $a = i$ sets $a.\texttt{val}$ to the provided input i;
- $a \mathrel{+}= i$ aggregates the input i into $acc.\texttt{val}$ using the combiner, i.e. sets $a.\texttt{val}$ to $a.\texttt{val} \oplus i$.

For a comprehensive documentation on GSQL accumulators, see the developer's guide at http://docs.tigergraph.com. Here, we explain accumulators by example.

Example 4 (Multiple Aggregations by Distinct Grouping Criteria) *Consider a graph named "SalesGraph" in which the sale of a product p to a customer c is modeled by a directed "Bought"-edge from the "Customer"-vertex modeling c to the "Product"-vertex modeling p. The number of product units bought, as well as the discount at which they were offered are recorded as attributes of the edge. The list price of the product is stored as attribute of the corresponding "Product" vertex.*

```
WITH
  SumAccum<float> @revenuePerToy, @revenuePerCust, @@totalRevenue
BEGIN

  SELECT c
  FROM   SalesGraph AS Customer: c -(Bought>: b)- Product:p
  WHERE  p.category = 'toys'
  ACCUM  float salesPrice = b.quantity * p.listPrice * (100 - b.percentDiscount)/100.0,
         c.@revenuePerCust += salesPrice,
         p.@revenuePerToy += salesPrice,
         @@totalRevenue += salesPrice;

END
```

Figure 3: Multi-Aggregating Query for Example 4

We wish to simultaneously compute the sales revenue per product from the "toy" category, the toy sales revenue per customer, and the overall total toy sales revenue. [1]

We define a vertex accumulator type for each kind of revenue. The revenue for toy product p will be aggregated at the vertex modeling p by vertex accumulator `revenuePerToy`*, while the revenue for customer c will be aggregated at the vertex modeling c by the vertex accumulator* `revenuePerCust`*. The total toy sales revenue will be aggregated in a global accumulator called* `totalRevenue`*. With these accumulators, the multi-grouping query is concisely expressible (Figure 3).*

Note the definition of the accumulators using the WITH clause in the spirit of standard SQL definitions. Here, `SumAccum<float>` *denotes the type of accumulators that hold an internal floating point scalar value and aggregate inputs using the floating point addition operation. Accumulator names prefixed by a single @ symbol denote vertex accumulators (one instance per vertex) while accumulator names prefixed by @@ denote a global accumulator (a single shared instance).*

Also note the novel ACCUM clause, which specifies the generation of inputs to the accumulators. Its first line introduces a local variable "salesPrice", whose value depends on attributes found in both the "Bought" edge and the "Product" vertex. This value is aggregated into each accumulator using the "+=" operator. `c.@revenuePerCust` *refers to the vertex accumulator instance located at the vertex denoted by vertex variable c.* □

Multi-Output SELECT Clause. GSQL's accumulators allow the simultaneous specification of multiple aggregations of the same data. To take full advantage of this capability, GSQL complements it with the ability to concisely specify simultaneous outputs into multiple tables for data obtained by the same query body. This can be thought of as evaluating multiple independent SELECT clauses.

Example 5 (Multi-Output SELECT) *While the query in Example 4 outputs the customer vertex ids, in that example we were interested in its side effect of annotating vertices with the aggregated revenue values and of computing the total revenue. If instead we wished to create two tables, one associating customer names with their revenue, and one associating toy names with theirs, we would employ GSQL's* multi-output SELECT clause *as follows (preserving the FROM, WHERE and ACCUM clauses of Example 4).*

```
SELECT c.name, c.@revenuePerCust INTO PerCust;
       t.name, t.@revenuePerToy INTO PerToy
```

Notice the semicolon, which separates the two simultaneous outputs. □

Semantics. The semantics of GSQL queries can be given in a declarative fashion analogous to SQL semantics: for each distinct match of the FROM clause pattern that satisfies the WHERE clause condition, the ACCUM clause is executed precisely once. After the ACCUM clause executions complete, the multi-output SELECT clause executes

[1] Note that writing this query in standard SQL is cumbersome. It requires performing two GROUP BY operations, one by customer and one by product. Alternatively, one can use the window functions' OVER - PARTITION BY clause, that can perform the groupings independently but whose output repeats the customer revenue for each product bought by the customer, and the product revenue for each customer buying the product. Besides yielding an unnecessarily large result, this solution then requires two post-processing SQL queries to separate the two aggregates.

```
CREATE QUERY TopKToys (vertex<Customer> c, int k) FOR GRAPH SalesGraph {

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

  SELECT DISTINCT o INTO OthersWithCommonLikes
  FROM   Customer:c -(Likes>)- Product:t -(<Likes)- Customer:o
  WHERE  o <> c and t.category = 'Toys'
  ACCUM  o.@inCommon += 1
  POST_ACCUM o.@lc = log (1 + o.@inCommon);

  SELECT t.name, t.@rank AS rank INTO Recommended
  FROM   OthersWithCommonLikes:o -(Likes>)- Product:t
  WHERE  t.category = 'Toy' and c <> o
  ACCUM  t.@rank += o.@lc
  ORDER BY t.@rank DESC
  LIMIT  k;

  RETURN Recommended;
}
```

Figure 4: Recommender Query for Example 6

each of its semicolon-separated individual fragments independently, as standard SQL clauses. Note that we do not specify the order in which matches are found and consequently the order of ACCUM clause applications. We leave this to the engine implementation to support optimization. The result is well-defined (input-order-invariant) whenever the accumulator's binary aggregation operation $\oplus$ is commutative and associative. This is certainly the case for addition, which is used in Example 4, and for most of GSQL's built-in accumulators.

Extensible Accumulator Library. GSQL offers a list of built-in accumulator types. TigerGraph's experience with the deployment of GSQL has yielded the short list from Section 4, that covers most use cases we have encountered in customer engagements. In addition, GSQL allows users to define their own accumulators by implementing a simple C++ interface that declares the binary combiner operation $\oplus$ used for aggregation of inputs into the stored value. This leads to an extensible query language, facilitating the development of accumulator libraries.

Accumulator Support for Multi-pass Algorithms. The scope of the accumulator declaration may cover a sequence of query blocks, in which case the accumulated values computed by a block can be read (and further modified) by subsequent blocks, thus achieving powerful composition effects. These are particularly useful in multi-pass algorithms.

Example 6 (Two-Pass Recommender Query) *Assume we wish to write a simple toy recommendation system for a customer c given as parameter to the query. The recommendations are ranked in the classical manner: each recommended toy's rank is a weighted sum of the likes by other customers.*

Each like by an other customer o is weighted by the similarity of o to customer c. In this example, similarity is the standard log-cosine similarity [14], which reflects how many toys customers c and o like in common. Given two customers x and y, their log-cosine similarity is defined as $log(1 + count\ of\ common\ likes\ for\ x\ and\ y)$.

The query is shown in Figure 4. The query header declares the name of the query and its parameters (the vertex of type "Customer" c, and the integer value k of desired recommendations). The header also declares the graph for which the query is meant, thus freeing the programmer from repeating the name in the FROM clauses. Notice also that the accumulators are not declared in a WITH clause. In such cases, the GSQL convention is that the accumulator scope spans all query blocks. Query `TopKToys` *consists of two blocks.*

The first query block computes for each other customer o their log-cosine similarity to customer c, storing it in o's vertex accumulator `@lc`*. To this end, the ACCUM clause first counts the toys liked in common by aggregating for each such toy the value* 1 *into o's vertex accumulator* `@inCommon`*. The POST_ACCUM clause then computes the logarithm and stores it in o's vertex accumulator* @lc*.*

Next, the second query block computes the rank of each toy t by adding up the similarities of all other customers o who like t. It outputs the top k recommendations into table `Recommended`*, which is returned by the query.*

Notice the input-output composition due to the second query block's FROM clause referring to the set of vertices `OthersWithCommonLikes` *(represented as a single-*

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

  MaxAccum<float> @@maxDifference;        // max score change in an iteration
  SumAccum<float> @received_score;        // 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      v
         FROM        AllV:v -(LinkTo>)- Page:n
         ACCUM       n.@received_score += v.@score/v.outdegree()
         POST-ACCUM  v.@score = 1-dampingFactor + dampingFactor * v.@received_score,
                     v.@received_score = 0,
                     @@maxDifference += abs(v.@score - v.@score');
  END;
}
```

Figure 5: PageRank Query for Example 7

column table) computed by the first query block. Also notice the side-effect composition due to the second block's ACCUM clause referring to the `@lc` *vertex accumulators computed by the first block. Finally, notice how the SELECT clause outputs vertex accumulator values (*`t.@rank`*) analogously to how it outputs vertex attributes (*`t.name`*).* □

Example 6 introduces the POST_ACCUM clause, which is a convenient way to post-process accumulator values after the ACCUM clause finishes computing their new aggregate value.

3.3 Loops

GSQL includes a while loop primitive, which, when combined with accumulators, supports iterative graph algorithms. We illustrate for the classic PageRank [2] algorithm.

Example 7 (PageRank) *Figure 5 shows a GSQL query implementing a simple version of PageRank.*

Notice the while loop that runs a maximum number of iterations provided as parameter `maxIteration`*. Each vertex v is equipped with a* `@score` *accumulator that computes the rank at each iteration, based on the current score at v and the sum of fractions of previous-iteration scores of v's neighbors (denoted by vertex variable n).* `v.@score'` *refers to the value of this accumulator at the previous iteration.*

According to the ACCUM clause, at every iteration each vertex v contributes to its neighbor n's score a fraction of v's current score, spread over v's outdegree. The score fractions contributed by the neighbors are summed up in the vertex accumulator `@received_score`*.*

As per the POST_ACCUM clause, once the sum of score fractions is computed at v, it is combined linearly with v's current score based on the parameter `dampingFactor`*, yielding a new score for v.*

The loop terminates early if the maximum difference over all vertices between the previous iteration's score (accessible as `v.@score'` *) and the new score (now available in* `v.@score`*) is within a threshold given by parameter* `maxChange`*. This maximum is computed in the* `@@maxDifference` *global accumulator, which receives as inputs the absolute differences computed by the POST_ACCUM clause instantiations for every value of vertex variable v.* □

3.4 DARPEs

We follow the tradition instituted by a line of classic work on querying graph (a.k.a. semi-structured) data which yielded such reference query languages as WebSQL [13], StruQL [6] and Lorel [1]. Common to all these languages is a primitive that allows the programmer to specify traversals along paths whose structure is constrained by a regular path expression.

Regular path expressions (RPEs) are regular expressions over the alphabet of edge types. They conform to the context-free grammar

$$\begin{aligned} rpe &\rightarrow \ \_ \mid EdgeType \mid \text{'('} \ rpe \ \text{')'} \mid rpe \ \text{'*'} \ bounds? \\ &\mid \ rpe \ \text{'.'} \ rpe \mid rpe \ \text{'|'} \ rpe \\ bounds &\rightarrow \ N? \ \text{'..'} \ N? \end{aligned}$$

where *EdgeType* and N are terminal symbols representing respectively the name of an edge type and a natural number. The wildcard symbol "_" denotes any edge type, "." denotes the concatenation of its pattern arguments, and "|" their disjunction. The '*' terminal symbol is the standard Kleene star specifying several (possibly 0 or unboundedly many) repetitions of its RPE argument. The optional bounds can specify a minimum and a maximum number of repetitions (to the left and right of the ".." symbol, respectively).

A path p in the graph is said to satisfy an RPE R if the sequence of edge types read from the source vertex of p to the target vertex of p spells out a word in the language accepted by R when interpreted as a standard regular expression over the alphabet of edge type names.

DARPEs. Since GSQL's data model allows for the existence of both directed and undirected edges in the same graph, we refine the RPE formalism, proposing *Direction-Aware RPEs (DARPEs)*. These allow one to also specify the orientation of directed edges in the path. To this end, we extend the alphabet to include for each edge type E the symbols

- E, denoting a hop along an undirected E-edge,
- $E>$, denoting a hop along an outgoing E-edge (from source to target vertex), and
- $<E$, denoting a hop along an incoming E-edge (from target to source vertex).

Now the notion of satisfaction of a DARPE by a path extends classical RPE satisfaction in the natural way.

DARPEs enable the free mix of edge directions in regular path expressions. For instance, the pattern

$$E> \ .(F> \mid <G) * . <H.J$$

matches paths starting with a hop along an outgoing E-edge, followed by a sequence of zero or more hops along either outgoing F-edges or incoming G-edges, next by a hop along an incoming H-edge and finally ending in a hop along an undirected J-edge.

3.4.1 DARPE Semantics

A well-known semantic issue arises from the tension between RPE expressivity and well-definedness. Regarding expressivity, applications need to sometimes specify reachability in the graph via RPEs comprising unbounded (Kleene) repetitions of a path shape (e.g. to find which target users are influenced by source users on Twitter, we seek the paths connecting users directly or indirectly via a sequence of tweets or retweets). Applications also need to compute various aggregate statistics over the graph, many of which are multiplicity-sensitive (e.g. count, sum, average). Therefore, pattern matches must preserve multiplicities, being interpreted under bag semantics. That is, a pattern :s -(RPE)- :t should have as many matches of variables (s, t) to a given pair of vertices (n_1, n_2) as there are distinct paths from n_1 to n_2 satisfying the RPE. In other words, the count of these paths is the multiplicity of the (n_1, n_2) in the bag of matches.

The two requirements conflict with well-definedness: when the RPE contains Kleene stars, cycles in the graph can yield an infinity of distinct paths satisfying the RPE (one for each number of times the path wraps around the cycle), thus yielding infinite multiplicities in the query output. Consider for example the pattern
Person: p1 -(Knows>)- Person: p2* in a social network with cycles involving the "Knows" edges.

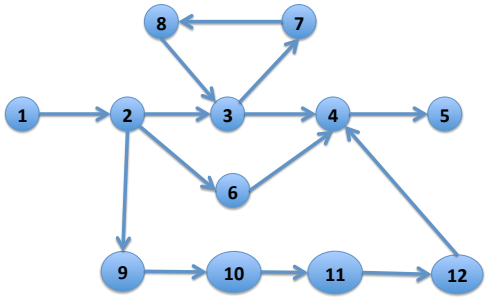

Figure 6: Graph G_1 for Example 8

Legal Paths. Traditional solutions limit the kind of paths that are considered legal, so as to yield a finite number in any graph. Two popular approaches allow only paths with non-repeated vertices/edges. [2] However under these definitions of path legality the evaluation of RPEs is in general notoriously intractable: even checking existence of legal paths that satisfy the RPE (without counting them) has worst-case NP-hard data complexity (i.e. in the size of the graph [12, 11]). As for the process of counting such paths, it is #P-complete. This worst-case complexity does not scale to large graphs.

In contrast, GSQL adopts the *all-shortest-paths* legality criterion. That is, among the paths from s to t satisfying a given DARPE, GSQL considers legal all the shortest ones. Checking existence of a shortest path that satisfies a DARPE, and even counting all such shortest paths is tractable (has polynomial data complexity).

For completeness, we recall here also the semantics adopted by the SparQL standard [8] (SparQL is the W3C-standardized query language for RDF graphs): SparQL regular path expressions that are Kleene-starred are interpreted as boolean *tests* whether such a path exists, without counting the paths connecting a pair of endpoints. This yields a multiplicity of 1 on the pair of path endpoints, which does not align with our goal of maintaining bag semantics for aggregation purposes.

Example 8 (Contrasting Path Semantics) *To contrast the various path legality flavors, consider the graph G_1 in Figure 6, assuming that all edges are typed "E". Among all paths from source vertex 1 to target vertex 5 that satisfy the DARPE "$E> *$", there are*

- *Infinitely many unrestricted paths, depending on how many times they wrap around the 3-7-8-3 cycle;*
- *Three non-repeated-vertex paths (1-2-3-4-5, 1-2-6-4-5, and 1-2-9-10-11-12-4-5);*
- *Four non-repeated-edge paths (1-2-3-4-5, 1-2-6-4-5, 1-2-9-10-11-12-4-5, and 1-2-3-7-8-3-4-5);*
- *Two shortest paths (1-2-3-4-5 and 1-2-6-4-5).*

Therefore, pattern $: s - (E> *) - : t$ will return the binding $(s \mapsto 1, t \mapsto 5)$ with multiplicity 3, 4, or 2 under the non-repeated-vertex, non-repeated-edge respectively shortest-path legality criterion. In addition, under SparQL semantics, the multiplicity is 1.

While in this example the shortest paths are a subset of the non-repeated-vertex paths, which in turn are a subset of the non-repeated-edge paths, this inclusion does not hold in general, and the different classes are incomparable. Consider Graph G_2 from Figure 7, and the pattern

$$: s - (E> *.F> .E> *) - : t$$

and note that it does not match any path from vertex 1 to vertex 4 under non-repeated vertex or edge semantics, while it does match one such path under shortest-path semantics (1-2-3-5-6-2-3-4). □

[2] Gremlin's [16] default semantics allows all unrestricted paths (and therefore possibly non-terminating graph traversals), but virtually all the documentation and tutorial examples involving unbounded traversal use non-repeated-vertex semantics (by explicitly invoking a built-in `simplePath` predicate). By default, Cypher [15] adopts the non-repeated-edge semantics.

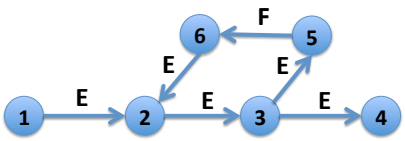

Figure 7: Graph G_2 for Example 8

3.5 Updates

GSQL supports vertex-, edge- as well as attribute-level modifications (insertions, deletions and updates), with a syntax inspired by SQL (detailed in the online documentation at https://docs.tigergraph.com/dev/gsql-ref).

4 GSQL's Main Built-In Accumulator Types

GSQL comes with a list of pre-defined accumulators, some of which we detail here. For details on GSQL's accumulators and more supported types, see the online documentation at https://docs.tigergraph.com/dev/gsql-ref).

`SumAccum<N>`, where N is a numeric type. This accumulator holds an internal value of type N, accepts inputs of type N and aggregates them into the internal value using addition.

`MinAccum<O>`, where `O` is an ordered type. It computes the minimum value of its inputs of type `O`.

`MaxAccum<O>`, as above, swapping max for min aggregation.

`AvgAccum<N>`, where `N` is a numeric type. This accumulator computes the average of its inputs of type `N`. It is implemented in an order-invariant way by internally maintaining both the sum and the count of the inputs seen so far.

`OrAccum`, which aggregates its boolean inputs using logical disjunction.

`AndAccum`, which aggregates its boolean inputs using logical conjunction.

`MapAccum<K,V>` stores an internal value of map type, where `K` is the type of keys and `V` the type of values. `V` can itself be an accumulator type, thus specifying how to aggregate values mapped to the same key.

`HeapAccum<T>(capacity, field_1 [ASC|DESC], field_2 [ASC|DESC], ..., field_n [ASC|DESC])` implements a priority queue where `T` is a tuple type whose fields include `field_1` through `field_n`, each of ordered type, `capacity` is the integer size of the priority queue, and the remaining arguments specify a lexicographic order for sorting the tuples in the priority queue (each field may be used in ascending or descending order).

5 Formal Syntax

In the following, terminal symbols are shown in bold font and not further defined when their meaning is self-understood.

5.1 Declarations

```
decl → accType gAccName (= expr)? ;
     | accType vAccName  (= expr)? ;
     | baseType var (= expr)? ;
```

```
gAccName → @@Id
vAccName → @Id


accType → SetAccum   <baseType>
        |  BagAccum   <baseType>
        |  HeapAccum<tupleType>(capacity (, Id dir?)+)
        |  OrAccum
        |  AndAccum
        |  BitwiseOrAccum
        |  BitwiseAndAccum
        |  MaxAccum   <orderedType>
        |  MinAccum   <orderedType>
        |  SumAccum   <numType|string>
        |  AvgAccum   <numType>
        |  ListAccum  <type>
        |  ArrayAccum <type> dimension+
        |  MapAccum   <baseType , type>
        |  GroupByAccum <baseType Id (, baseType Id)* ,
                        accType>

baseType → orderedType
         |  boolean
         |  datetime
         |  edge (< edgeType >)?
         |  tupleType

orderedType → numType
            | string
            | vertex (< vertexType >)?

numType → int
        |  uint
        |  float
        |  double

capacity → NumConst | paramName

paramName → Id

tupleType → Tuple < baseType Id (, baseType Id)* >

dir → ASC | DESC

dimension → [expr?]

type → baseType | accType
```

5.2 DARPEs

```
darpe → edgeType
      | edgeType>
      | <edgeType
      | darpe* bounds?
      | (darpe)
      | darpe (. darpe)+
      | darpe (| darpe)+

edgeType → Id

bounds → NumConst .. NumConst
       | .. NumConst
       | NumConst ..
       | NumConst
```

5.3 Patterns

```
pathPattern  →
   vTest (: var)?
|  pathPattern
     -(darpe (: var)?)- vTest (: var)?

pattern → pathPattern (, pathPattern)*

vTest  → _
       | vertexType (| vertexType)*

var → Id

vertexType → Id
```

5.4 Atoms

```
atom → relAtom | graphAtom

relAtom   → tableName AS? var
tableName → Id

graphAtom  → (graphName AS?)? pattern
graphName  → Id
```

5.5 FROM Clause

```
fromClause → FROM atom (, atom)*
```

5.6 Terms

```
term → constant
     |  var
     |  var.attribName
     |  gAccName
     |  gAccName'
     |  var.vAccName
     |  var.vAccName'
     |  var.type

constant → NumConst
         | StringConst
         | DateTimeConst
         | true
         | false

attribName → Id
```

5.7 Expressions

```
  expr → term
       | ( expr )
       | — expr
       | expr arithmOp  expr
       | not expr
       | expr logicalOp expr
       | expr setOp expr
       | expr between expr and expr
       | expr not? in expr
       | expr like expr
       | expr is not? null
       | fnName ( exprs? )
       | case
           (when condition then expr)+
           (else expr)?
         end
       | case expr
           (when constant then expr)+
```

```
          (else expr)?
        end
      | (exprs?)
      | [exprs?]
      | ( exprs -> exprs ) // MapAccum input
      | expr arrayIndex+  // ArrayAccum access

arithmOp → * | / | % | + | − | & | |

logicalOp → and | or

setOp → intersect | union | minus

exprs → expr(, exprs)*

arrayIndex  → [expr]
```

5.8 WHERE Clause

```
whereClause → WHERE condition

condition → expr
```

5.9 ACCUM Clause

```
accClause → ACCUM stmts

stmts → stmt (, stmt)*

stmt → varAssignStmt
     |  vAccUpdateStmt
     |  gAccUpdateStmt
     |  forStmt
     |  caseStmt
     |  ifStmt
     |  whileStmt

varAssignStmt → baseType? var = expr

vAccUpdateStmt → var.vAccName = expr
               |  var.vAccName += expr

gAccUpdateStmt → gAccName = expr
               |  gAccName += expr

forStmt → foreach var in expr do stmts end
        |  foreach (var (, var)*) in expr do stmts end
        |  foreach var in range (expr , expr) do stmts end

caseStmt → case
              (when condition then stmts)+
              (else stmts)?
           end
         |  case expr
              (when constant then stmts)+
              (else stmts)?
           end

ifStmt → if condition then stmts (else stmts)? end

whileStmt → while condition limit expr do body end
body      → bodyStmt (, bodyStmt)*
bodyStmt  → stmt
          |  continue
          |  break
```

5.10 POST_ACCUM Clause

```
pAccClause → POST_ACCUM stmts
```

5.11 SELECT Clause

```
selectClause → SELECT outTable (; outTable)*

outTable → DISTINCT? col (, col)* INTO tableName

col → expr (AS colName)?

tableName → Id

colName → Id
```

5.12 GROUP BY Clause

```
groupByClause → GROUP BY exprs (; exprs)*
```

5.13 HAVING Clause

```
havingClause → HAVING condition (; condition)*
```

5.14 ORDER BY Clause

```
orderByClause → ORDER BY oExprs (; oExprs)*

oExprs → oExpr(, oExpr)*

oExpr → expr dir?
```

5.15 LIMIT Clause

```
limitClause → LIMIT expr (; expr)*
```

5.16 Query Block Statements

```
queryBlock → selectClause
           |  fromClause
           |  whereClause?
           |  accClause?
           |  pAccClause?
           |  groupByClause?
           |  havingClause?
           |  orderByClause?
           |  limitClause?
```

5.17 Query

```
query → CREATE QUERY Id (params?)
        (FOR GRAPH graphName)? {
          decl*
          qStmt*
          (RETURN expr)?
        }

params → param (, param)*

param → paramType paramName

paramType → baseType
          |  set<baseType>
          |  bag<baseType>
          |  map<baseType , baseType>

qStmt → stmt ; | queryBlock ;
```

6 GSQL Formal Semantics

GSQL expresses queries over standard SQL tables and over graphs in which both directed and undirected edges may coexist, and whose vertices and edges carry data (attribute name-value maps).

The core of a GSQL query is the SELECT-FROM-WHERE block modeled after SQL, with the FROM clause specifying a pattern to be matched against the graphs and the tables. The pattern contains vertex, edge and tuple variables and each match induces a variable binding for them. The WHERE and SELECT clauses treat these variables as in SQL. GSQL supports an additional ACCUM clause that is used to update accumulators.

6.1 The Graph Data Model

In TigerGraph's data model, graphs allow both directed and undirected edges. Both vertices and edges can carry data (in the form of attribute name-value maps). Both vertices and edges are typed.

Let $\mathcal{V}$ denote a countable set of vertex ids, $\mathcal{E}$ a countable set of edge ids disjoint from $\mathcal{V}$, $\mathcal{A}$ a countable set of attribute names, $\mathcal{T}_v$ a countable set of vertex type names, $\mathcal{T}_e$ a countable set of edge type names.

Let $\mathcal{D}$ denote an infinite domain (set of values), which comprises

- all numeric values,
- all string values,
- the boolean constants **`true`** and **`false`**,
- all datetime values,
- $\mathcal{V}$,
- $\mathcal{E}$,
- all sets of values (their sub-class is denoted $\{\mathcal{D}\}$),
- all bags of values ($\{\!|D|\!\}$),
- all lists of values ($[\mathcal{D}]$), and
- all maps (sets of key-value pairs, $\{\mathcal{D} \mapsto \mathcal{D}\}$).

A graph is a tuple

$$G = (V, E, st, \tau_v, \tau_e, \delta)$$

where

- $V \subset \mathcal{V}$ is a finite set of vertices.
- $E \subset \mathcal{E}$ is a finite set of edges.
- $st : E \to 2^{V \times V}$ is a function that associates with an edge e its endpoint vertices. If e is directed, $st(e)$ is a singleton set containing a (source,target) vertex pair. If e is undirected, $st(e)$ is a set of two pairs, corresponding to both possible orderings of the endpoints.
- $\tau_v : V \to \mathcal{T}_v$ is a function that associates a type name to each vertex.
- $\tau_e : E \to \mathcal{T}_e$ is a function that associates a type name to each edge.
- $\delta : (V \cup E) \times \mathcal{A} \to \mathcal{D}$ is a function that associates domain values to vertex/edge attributes (identified by the vertex/edge id and the attribute name).

6.2 Contexts

GSQL queries are composed of multiple statements. A statement may refer to intermediate results provided by preceding statements (e.g. global variables, temporary tables, accumulators). In addition, since GSQL queries can be parameterized just like SQL views, statements may refer to parameters whose value is provided by the initial query call.

A statement must therefore be evaluated in a *context* which provides the values for the names referenced by the statement. We model a context as a map from the names to the values of parameters/global variables/temporary tables/accumulators, etc.

Given a context map ctx and a name n, $ctx(n)$ denotes the value associated to n in ctx [3]

$dom(ctx)$ denotes the *domain* of context ctx, i.e. the set of names which have an associated value in ctx(we say that these names are *defined* in ctx).

Overriding Context Extension When a new variable is introduced by a statement operating within a context ctx, the context needs to be extended with the new variable's name and value.

$$\{n \mapsto v\} \triangleright ctx$$

denotes a new context obtained by modifying a *copy* of ctx to associate name n with value v (overwriting any pre-existing entry for n).

Given contexts ctx_1 and $ctx_2 = \{n_i \mapsto v_i\}_{1 \le i \le k}$, we say that ctx_2 *overrides* ctx_1, denoted $ctx_2 \triangleright ctx_1$ and defined as:

$$\begin{aligned} ctx_2 \triangleright ctx_1 &= c_k \text{ where} \\ c_0 &= ctx_1, \text{ and} \\ c_i &= \{n_i \mapsto v_i\} \triangleright c_{i-1}, \text{ for } 1 \le i \le k \end{aligned}$$

Consistent Contexts We call two contexts ctx_1, ctx_2 *consistent* if they agree on every name in the intersection of their domains. That is, for each $x \in dom(ctx_1) \cap dom(ctx_2)$, $ctx_1(x) = ctx_2(x)$.

Merged Contexts For consistent contexts, we can define $ctx_1 \cup ctx_2$, which denotes the *merged context* over the union of the domains of ctx_1 and ctx_2:

$$ctx_1 \cup ctx_2(n) = \begin{cases} ctx_1(n), & \text{if } n \in dom(ctx_1) \\ ctx_2(n), & \text{otherwise} \end{cases}$$

6.3 Accumulators

A GSQL query can declare accumulators whose names come from Acc_g, a countable set of global accumulator names and from Acc_v, a disjoint countable set of vertex accumulator names.

Accumulators are data types that store an internal value and take inputs that are aggregated into this internal value using a binary operation. GSQL distinguishes among two accumulator flavors:

- *Vertex accumulators* are attached to vertices, with each vertex storing its own local accumulator instance.
- *Global accumulators* have a single instance.

Accumulators are polymorphic, being parameterized by the type S of the stored internal value, the type I of the inputs, and the binary *combiner* operation

$$\oplus : S \times I \to S.$$

Accumulators implement two assignment operators. Denoting with $a.\texttt{val}$ the internal value of accumulator instance a,

[3] In the remainder of the presentation we assume that the query has passed all appropriate semantic and type checks and therefore $ctx(n)$ is defined for every n we use.

- $a \ = i$ sets $a.\texttt{val}$ to the provided input i;
- $a \mathrel{+}= i$ aggregates the input i into $acc.\texttt{val}$ using the combiner, i.e. sets $a.\texttt{val}$ to $a.\texttt{val} \oplus i$.

Each accumulator instance has a pre-defined default for the internal value.

When an accumulator instance a is referenced in a GSQL expression, it evaluates to the internally stored value $a.val$. Therefore, the context must associate the internally stored value to the instannce of global accumulators (identified by name) and of vertex accumulators (identified by name and vertex).

Specific Accumulator Types We revisit the accumulator types listed in Section 4.

`SumAccum<N>` is the type of accumulators where the internal value and input have numeric type `N`/`string` their default value is 0/the empty string and the combiner operation is the arithmetic +/string concatenation, respectively.

`MinAccum<O>` is the type of accumulators where the internal value and input have ordered type `O` (numeric, datetime, string, vertex) and the combiner operation is the binary minimum function. The default values are the (architecture-dependent) minimum numeric value, the default date, the empty string, and undefined, respectively. Analogously for `MaxAccum<O>`.

`AvgAccum<N>` stores as internal value a pair consisting of the sum of inputs seen so far, and their count. The combiner adds the input to the running sum and increments the count. The default value is 0.0 (double precision).

`AndAccum` stores an internal boolean value (default **true**, and takes boolean inputs, combining them using logical conjunction. Analogously for `OrAccum`, which defaults to **false** and uses logical disjunction as combiner.

`MapAccum<K,V>` stores an internal value that is a map m, where `K` is the type of m's keys and `V` the type of m's values. `V` can itself be an accumulator type, specifying how to aggregate values mapped by m to the same key. The default internal value is the empty map. An input is a key-value pair $(k, v) \in K \times V$. The combiner works as follows: if the internal map m does not have an entry involving key k, m is extended to associate k with v. If k is already defined in m, then if V is not an accumulator type, m is modified to associate k to v, overwriting k's former entry. If V is an accumulator type, then $m(k)$ is an accumulator instance. In that case m is modified by replacing $m(k)$ with the new accumulator obtained by combining v into $m(k)$ using V's combiner.

6.4 Declarations

The semantics of a declaration is a function from contexts to contexts.

When they are created, accumulator instances are initialized by setting their stored internal value to a default that is defined with the accumulator type. Alternatively, they can be initialized by explicitly setting this default value using an assignment:

$$\mathit{type}\ @n \ = e$$

declares a vertex accumulator of name $n \in Acc_v$ and type *type*, all of whose instances are initialized with $[\![e]\!]^{ctx}$, the result of evaluating the expression in the current context. The effect of this declaration is to create the initialized accumulator instances and extend the current context ctx appropriately. Note that a vertex accumulator instance is identified by the accumulator name and the vertex hosting the instance. We model this by having the context associate the vertex accumulator name with a map from vertices to instances.

$$[\![\mathit{type}\ @n \ = e]\!](ctx) = ctx'$$

where

$$ctx' = \{@n \mapsto \bigcup_{v \in V} \{v \mapsto [\![e]\!]^{ctx}\}\} \triangleright ctx$$

Similarly,

$$\mathit{type}\ @@n \ = e$$

declares a global accumulator named $n \in Acc_g$ of type *type*, whose single instance is initialized with $[\![expr]\!]^{ctx}$. This instance is identified in the context simply by the accumulator name:

$$[\![\mathit{type}\ @@n \ = e]\!](ctx) = \{@@n \mapsto [\![e]\!]^{ctx}\} \triangleright ctx.$$

Finally, global variable declarations also extend the context:

$$[\![baseType\ var\ = e]\!](ctx) = \{var \mapsto [\![e]\!]^{ctx}\} \triangleright ctx.$$

6.5 DARPE Semantics

DARPEs specify a set of paths in the graph, formalized as follows.

Paths A *path* p in graph G is a sequence

$$p = v_0,\ e_1,\ v_1,\ e_2,\ v_2,\ \ldots,\ v_{n-1},\ e_n,\ v_n$$

where $v_0 \in V$ and for each $1 \leq i \leq n$,

- $v_i \in V$, and
- $e_i \in E$, and
- v_{i-1} and v_i are the endpoints of edge e_i regardless of e_i's orientation: $(v_{i-1}, v_i) \in st(e_i)$ or $(v_i, v_{i-1}) \in st(e_i)$.

Path Length We call n the *length* of p and denote it with $|p|$. Note that when $p = v_0$ we have $|p| = 0$.

Path Source and Target We call v_0 the *source*, and v_n the *target* of path p, denoted $\texttt{src}(p)$ and $\texttt{tgt}(p)$, respectively. When $n = 0$, $\texttt{src}(p) = \texttt{tgt}(p) = v_0$.

Path Hop For each $1 \leq i \leq n$, we call the triple (v_{i-1}, e_i, v_i) the *hop* i of path p.

Path Label We define the *label of hop i in p*, denoted $\lambda_i(p)$ as follows:

$$\lambda_i(p) = \begin{cases} \tau_e(e_i) \quad , & \text{if } e_i \text{ is undirected,} \\ \tau_e(e_i) >, & \text{if } e_i \text{ is directed from } v_{i-1} \text{ to } v_i, \\ < \tau_e(e_i), & \text{if } e_i \text{ is directed from } v_i \text{ to } v_{i-1} \end{cases}$$

where $\tau_e(e_i)$ denotes the type name of edge e_i, and $\tau_e(e_i) >$ denotes a new symbol obtained by concatenating $\tau_e(e_i)$ with $>$, and analogously for $< \tau_e(e_i)$.

We call *label* of p, denoted λ(p), the word obtained by concatenating the hop labels of p:

$$\lambda(p) = \begin{cases} \epsilon \quad , & \text{if } |p| = 0, \\ \lambda_1(p)\lambda_2(p)\ldots\lambda_{|p|}(p), & \text{if } |p| > 0 \end{cases}$$

where, as usual [10], ϵ denotes the empty word.

DARPE Satisfaction We denote with

$$\Sigma^{\{>,<\}} = \bigcup_{t \in \mathcal{T}_e} \{t, t >, < t\}$$

the set of symbols obtained by treating each edge type t as a symbol, as well as creating new symbols by concatenating t and $>$, as well as $<$ and t.

We say that path p *satisfies* DARPE D, denoted $p \models D$, if $\lambda(p)$ is a word in the language accepted by D, $\mathcal{L}(D)$, when viewed as a regular expression over the alphabet $\Sigma^{\{>,<\}}$ [10]:

$$p \models D \ \Leftrightarrow\ \lambda(p) \in \mathcal{L}(D).$$

DARPE Match We say that DARPE D *matches* path p (and that p *is a match for* D) whenever p is a shortest path that satisfies D:

$$p \text{ matches } D \Leftrightarrow \; p \models D \;\; \wedge \;\; |p| = \min_{q \models D} |q|.$$

We denote with $[\![D]\!]^G$ the set of matches of a DARPE D in graph G.

6.6 Pattern Semantics

Patterns consist of DARPEs and variables. The former specify a set of paths in the graph, the latter are bound to vertices/edges occuring on these paths.

A pattern P specifies a function from a graph G and a context ctx to

- a set $[\![P]\!]^{G,ctx}$ of paths in the graph, each called a *match* of P, and
- a family $\{\beta_P^p\}_{p \in [\![P]\!]^{G,ctx}}$ of bindings for P's variables, one for each match p. Here, β_P^p denotes the binding *induced* by match p.

Temporary Tables and Vertex Sets To formalize pattern semantics, we note that some GSQL query statments may construct temporary tables that can be referred to by subsequent statements. Therefore, among others, the context maps the names to the extents (contents) of temporary tables. Since we can model a set of vertices as a single-column, duplicate-free table containining vertex ids, we refer to such tables as *vertex sets*.

V-Test Match Consider graph G and a context ctx. Given a v-test VT, a *match* for VT is a vertex $v \in V$ (a path of length 0) such that v belongs to VT (if VT is a vertex set name defined in ctx), or v is a vertex of type VT (otherwise). We denote the set of all matches of VT against G in context ctx with $[\![VT]\!]^{G,ctx}$.

$$v \in [\![VT]\!]^{G,ctx} \Leftrightarrow \begin{cases} v \in ctx(VT), & \textit{if } VT \textit{ is a vertex set} \\ & \textit{defined in } ctx, \\ \tau_v(v) = VT, & \textit{if } VT \in dom(\tau_v) \; i.e. \\ & VT \textit{ is a defined vertex type} \end{cases}$$

Variable Bindings Given graph G and tuple of variables $\mathbf{x}$, a *binding* for $\mathbf{x}$ in G is a function β from the variables in $\mathbf{x}$ to vertices or edges in G, $\beta : \mathbf{x} \to V \cup E$. Notice that a variable binding (binding for short) is a particular kind of context, hence all context-specific definitions and operators apply. In particular, the notion of consistent bindings coincides with that of consistent contexts.

Binding Tables We refer to a bag of variable bindings as a *binding table*.

No-hop Path Pattern Match Given a graph G and a context ctx, we say that path p is a match for no-hop pattern $P = VT : x$ (and we say that P *matches* p) if $p \in [\![VT]\!]^{G,ctx}$. Note that p is a path of length 0, i.e. a vertex v. The match p *induces a binding* of vertex variable x to v, $\beta_P^p = \{x \mapsto v\}$. [4]

One-hop Path Pattern Match Recall that in a one-hop pattern

$$P = S : s - (D : e) - T : t,$$

D is a disjunction of direction-adorned edge types, $D \subseteq \Sigma^{\{>,<\}}$, S, T are v-tests, s, t are vertex variables and e is an edge variable. We say that (single-edge) path $p = v_0,\ e_1,\ v_1$ *is a match for* P (and that P *matches* p) if $v_0 \in [\![S]\!]^{G,ctx}$, and $p \in [\![D]\!]^G$, and $v_1 \in [\![T]\!]^{G,ctx}$. The *binding induced* by this match, denoted β_P^p, is $\beta_P^p = \{s \mapsto v_0, t \mapsto v_1, e \mapsto e_1\}$.

[4] Note that both VT and x are optional. If VT is missing, then it is trivially satisfied by all vertices. If x is missing, then the induced binding is the emtpy map $\beta_P^p = \{\}$. These conventions apply for the remainder of the presentation and are not repeated explicitly.

Multi-hop Single-DARPE Path Pattern Match Given a DARPE D, path p *is a match for* multi-hop path pattern $P = S : s - (D) - T : t$ (and P *matches* p) if $p \in [\![D]\!]^G$, and $\texttt{src}(p) \in [\![S]\!]^{G,ctx}$ and $\texttt{tgt}(p) \in [\![T]\!]^{G,ctx}$. The match induces a *binding* $\beta_P^p = \{s \mapsto \texttt{src}(p), t \mapsto \texttt{tgt}(p)\}$.

Multi-DARPE Path Pattern Match Given DARPEs $\{D_i\}_{1 \leq i \leq n}$, *a match for* path pattern

$$P = S_0 : s_0 - (D_1) - S_1 : s_1 - \ldots - (D_n) - S_n : s_n$$

is a tuple of segments $(p_1, p_2, \ldots, p_n)$ of a path $p = p_1 p_2 \ldots p_n$ such that $p \in [\![D_1.D_2.\ldots.D_n]\!]^G$, and for each $1 \leq i \leq n$: $p_i \in [\![S_{i-1} : s_{i-1} - (D_i) - S_i : s_i]\!]^{G,ctx}$. Notice that p is a shortest path satisfying the DARPE obtained by concatenating the DARPEs of the pattern. Also notice that there may be multiple matches that correspond to distinct segmenations of the same path p. Each match induces a binding

$$\beta_P^{p_1.\ldots.p_n} = \bigcup_{i=1}^{n} \beta^{p_i}_{S_{i-1}:s_{i-1}-(D_i)-S_i:s_i}.$$

Notice that the individual bindings induced by the segments are pairwise consistent, since the target vertex of a path segment coincides with the source of the subsequent segment.

Consistent Matches Given two patterns P_1, P_2 with matches p_1, p_2 respectively, we say that p_1, p_2 are *consistent* if the bindings they induce are consistent. Recall that bindings are contexts so they inherit the notion of consistency.

Conjunctive Pattern Match Given a conjunctive pattern $\mathbf{P} = P_1, P_2, \ldots, P_n$, a *match* of $\mathbf{P}$ is a tuple of paths $\mathbf{p} = p_1, p_2, \ldots, p_n$ such that p_i is a match of P_i for each $1 \leq i \leq n$ and all matches are pairwise consistent. $\mathbf{p}$ induces a binding on all variables occuring in $\mathbf{P}$,

$$\beta_{\mathbf{P}}^{\mathbf{p}} = \bigcup_{i=1}^{n} \beta_{P_i}^{p_i}.$$

6.7 Atom Semantics

Recall that the FROM clause comprises a sequence of relational or graph atoms. Atoms specify a pattern and a collection to match it against. Each match induces a variable binding. Note that multiple distinct matches of a pattern can induce the same variable binding if they only differ in path elements that are not bound to the variables. To preserve the multiplicities of matches, we define the semantics of atoms a function from contexts to binding tables, i.e. bags of bindings for the variables introduced in the pattern.

Given relational atom T AS x where T is a table name and x a tuple variable, the matches of x are the tuples t from T's extent, $t \in ctx(T)$, and they each induce a binding $\beta_x^t = \{x \mapsto t\}$. The semantics of T AS x is the function

$$[\![T \text{ AS } x]\!]^{ctx} = \biguplus_{t \in ctx(T)} \{\!|\beta_x^t|\!\}.$$

Here, $\{\!|x|\!\}$ denotes the singleton bag containing element x with multiplicity 1, and $\biguplus$ denotes bag union, which is multiplicity-preserving (like SQL's UNION ALL operator).

Given graph atom G AS P where P is a conjunctive pattern, its semantics is the function

$$[\![G \text{ AS } P]\!]^{ctx} = \biguplus_{p \in [\![P]\!]^{ctx(G),ctx}} \{\!|\beta_P^p|\!\}.$$

When the graph is not explicitly specified in the atom, we use the default graph DG, which is guaranteed to be set in the context:

$$[\![P]\!]^{ctx} = \biguplus_{p \in [\![P]\!]^{ctx(DG),ctx}} \{\!|\beta_P^p|\!\}.$$

6.8 FROM Clause Semantics

The FROM clause specifies a function from contexts to (context, binding table) pairs that preserves the context:

$$[\![\text{FROM } a_1, \ldots, a_n]\!](ctx)$$

$$= (ctx, \biguplus_{\substack{\beta_1 \in [\![a_1]\!]^{ctx}, \ldots, \beta_n \in [\![a_n]\!]^{ctx}, \\ ctx, \beta_1, \ldots, \beta_n \text{ pairwise consistent}}} \{\!|\bigcup_{i=1}^{n} \beta_i|\!\}).$$

The requirement that all β_is be pairwise consistent addresses the case when atoms share variables (thus implementing joins). The consistency of each β_i with ctx covers the case when the pattern P mentions variables defined prior to the current query block. These are guaranteed to be defined in ctx if the query passes the semantic checks.

6.9 Term Semantics

Note that GSQL terms conform to SQL syntax, extended to also allow accumulator- and type-specific terms of form

- $@@A$, referring to the value of global accumulator A,
- $@@A'$, referring to the value of global accumulator A prior to the execution of the current query block,
- $x.@A$, specifying the value of the vertex accumulator A at the vertex denoted by variable x,
- $x.@A'$, specifying the value of the vertex accumulator A at the vertex denoted by variable x prior to the execution of the current query block,
- $x.type$, referring to the type of the vertex/edge denoted by variable x.

The variable x may be a query parameter, or a global variable set by a previous statement, or a variable introduced by the FROM clause pattern. Either way, its value must be given by the context ctx assuming that x is defined.

A constant c evaluates to itself:

$$[\![c]\!]^{ctx} = c.$$

A variable evaluates to the value it is bound to in the context:

$$[\![var]\!]^{ctx} = ctx(var).$$

An attribute projection term $x.A$ evaluates to the value of the A attribute of $ctx(x)$:

$$[\![x.A]\!]^{ctx} = \delta(ctx(x), A).$$

The evaluation of the accumulator-specific terms requires the context ctx to also

- map global accumulator names to their values, and
- map each vertex accumulator name (and the primed version referring to the prior value) to a map from vertices to accumulator values, to model the fact that each vertex has its own accumulator instance.

Given context ctx, terms of form $@@A$ evaluate to the value of the global accumulator as recorded in the context:

$$[\![@@A]\!]^{ctx} = ctx(@@A).$$

Terms of form $@@A'$ evaluate to the prior value of the global accumulator instance, as recorded in the context:

$$[\![@@A']\!]^{ctx} = ctx(@@A').$$

Terms of form $x.@A$ evaluate to the value of the vertex accumulator instance located at the vertex denoted by variable x:

$$[\![x.@A]\!]^{ctx} = ctx(@A)(ctx(x)).$$

Terms of form $x.@A'$ evaluate to the prior value of the vertex accumulator instance located at the vertex denoted by variable x:

$$[\![x.@A']\!]^{ctx} = ctx(@A')(ctx(x)).$$

Analogously for global accumulators:

$$[\![@@A']\!]^{ctx} = ctx(@@A').$$

All above definitions apply when A is not of type `AvgAccum`, which is treated as an exception: in order to support order-invariant implementation, the value associated in the context is a (sum,count) pair and terms evaluate to the division of the sum component by the count component. If A is of type `AvgAccum`,

$$[\![@@A]\!]^{ctx} = s/c,\ where\ (s,c) = ctx(@@A)$$

and

$$[\![x.@A]\!]^{ctx} = s/c, where\ (s,c) = ctx(@A)(ctx(x)).$$

Finally, terms of form $x.type$ evaluate as follows:

$$[\![x.type]\!]^{ctx} = \begin{cases} \tau_v(ctx(x)) & if\ x\ is\ vertex\ variable \\ \tau_e(ctx(x)) & if\ x\ is\ edge\ variable \end{cases}$$

6.10 Expression Semantics

The semantics of GSQL expressions is compatible with that of SQL expressions. Expression evaluation is defined in the standard SQL way by induction on their syntactic structure, with the evaluation of terms as the base case. We denote with

$$[\![E]\!]^{ctx}$$

the result of evaluating expression E in context ctx. Note that this definition also covers conditions (e.g. such as used in the WHERE clause) as they are particular cases (boolean expressions).

6.11 WHERE Clause Semantics

The semantics of a where clause

WHERE $Cond$

is a function $[\![$WHERE $Cond]\!]$ on (context, binding table)-pairs that preserves the context. It filters the input bag B keeping only the bindings that satisfy condition $Cond$ (preserving their multiplicity):

$$[\![\text{WHERE } Cond]\!](ctx, B) = (ctx, \biguplus_{\substack{\beta \in B, \\ [\![Cond]\!]^{\beta \triangleright ctx} = true}} \{\!|\beta|\!\}).$$

Notice that the condition is evaluated for each binding β in the new context $\beta \triangleright ctx$ obtained by extending ctx with β.

6.12 ACCUM Clause Semantics

The effect of the ACCUM clause is to modify accumulator values. It is executed exactly once for every variable binding β yielded by the WHERE clause (respecting multiplicities). We call each individual execution of the ACCUM clause an *acc-execution*.

Since multiple acc-executions may refer to the same accumulator instance, they do not write the accumulator value directly, to avoid setting up race conditions in which acc-executions overwrite each other's writes non-deterministically. Instead, each acc-execution yields a bag of input values for the accumulators mentioned in the ACCUM clause. The cumulative effect of all acc-executions is to aggregate all generated inputs into the appropriate accumulators, using the accumulator's $\oplus$ combiner.

Snapshot Semantics Note that all acc-executions start from the same snapshot of accumulator values and the effect of the accumulator inputs produced by each acc-execution are not visible to the acc-executions. These inputs are aggregated into accumulators only after all acc-executions have completed. We therefore say that the ACCUM clause executes under *snapshot semantics*, and we conceptually structure this execution into two phases: in the *Acc-Input Generation Phase*, all acc-executions compute accumulator inputs (acc-inputs for short), starting from the same accumulator value snapshot. After all acc-executions complete, the *Acc-Aggregation Phase* aggregates the generated inputs into the accumulators they are destined for. The result of the acc-aggregation phase is a new snapshot of the accumulator values.

6.12.1 Acc-Input Generation Phase

To formalize the semantics of this phase, we denote with $\{\!|\mathcal{D}|\!\}$ the class of bags with elements from $\mathcal{D}$.

We introduce two new maps that associate accumulators with the bags of inputs produced for them during the acc-execution phase:

- $\eta_{gacc} : Acc_g \rightarrow \{\!|\mathcal{D}|\!\}$, a *global acc-input map* that associates global accumulators instances (identified by name) with a bag of input values.
- $\eta_{vacc} : (V \times Acc_v) \rightarrow \{\!|\mathcal{D}|\!\}$, a *vertex acc-input map* that associates vertex accumulator instances (identified by (vertex,name) pairs) with a bag of input values.

For presentation convenience, we regard both maps as total functions (defined on all possible accumulator names) with finite support (only a finite number of accumulator names are associated with a non-empty acc-input bag).

Acc-statements An ACCUM clause consists of a sequence of statements

$$\text{ACCUM } s_1, s_2, \ldots, s_n$$

which we refer to as acc-statements to distinguish them from the statements appearing outside the ACCUM clause.

Acc-snapshots An *acc-snapshot* is a tuple

$$(ctx, \eta_{gacc}, \eta_{vacc})$$

consisting of a context ctx, a global acc-input map η_{gacc}, and a vertex acc-input map η_{vacc}.

Acc-statement semantics The semantics of an acc-statement s is a function $[\![s]\!]$ from acc-snapshots to acc-snapshots.

$$(ctx, \eta_{gacc}, \eta_{vacc}) \xrightarrow{[\![s]\!]} (ctx', \eta'_{gacc}, \eta'_{vacc}).$$

Local Variable Assignment Acc-statements may introduce new *local variables*, whose scope is the remainder of the ACCUM clause, or they may assign values to such local variables. Such acc-statements have form

$$type?\ lvar\ =\ expr$$

If the variable was already defined, then the type specification is missing and the acc-statement just updates the local variable.

The semantics of local variable assignments and declarations is a function that extends (possibly overriding) the context with a binding of local variable $lvar$ to the result of evaluating $expr$:

$$[\![type\ lvar\ =\ expr]\!]_\eta(ctx, \eta_{gacc}, \eta_{vacc}) = (ctx', \eta_{gacc}, \eta_{vacc})$$

where

$$ctx' = \{lvar\ \mapsto [\![expr]\!]^{ctx}\} \triangleright ctx.$$

Input to Global Accums Acc-statements that input into a global accumulator have form

$$@@A \mathrel{+}= expr$$

and their semantics is a function that adds the evaluation of expression $expr$ to the bag of inputs for accumulator A, $\eta_{gacc}(A)$:

$$[\![@@A \mathrel{+}= expr]\!]_\eta(ctx, \eta_{gacc}, \eta_{vacc}) = (ctx, \eta'_{gacc}, \eta_{vacc})$$

where

$$\begin{aligned} \eta'_{gacc} &= \{A \mapsto val\} \triangleright \eta_{gacc} \\ val &= \eta_{gacc}(A) \uplus \{\!\{[\![expr]\!]^{ctx}\}\!\}. \end{aligned}$$

Input to Vertex Accum Acc-statements that input into a vertex accumulator have form

$$x.@A \mathrel{+}= expr$$

and their semantics is a function that adds the evaluation of expression $expr$ to the bag of inputs for the instance of accumulator A located at the vertex denoted by x (the vertex is $ctx(x)$ and the bag of inputs is $\eta_{vacc}(ctx(x), A)$):

$$[\![x.@A \mathrel{+}= expr]\!]_\eta(ctx, \eta_{gacc}, \eta_{vacc}) = (ctx, \eta_{gacc}, \eta'_{vacc})$$

where

$$\begin{aligned} \eta'_{vacc} &= \{(ctx(x), A) \mapsto val\} \triangleright \eta_{vacc} \\ val &= \eta_{vacc}(ctx(x), A) \uplus \{\!\{[\![expr]\!]^{ctx}\}\!\}. \end{aligned}$$

Control Flow Control-flow statements, such as

`if` *cond* `then` *acc-statements* `else` *acc-statements* `end`

and

`foreach` *var* `in` *expr* `do` *acc-statements* `end`

etc., evaluate in the standard fashion of structured programming languages.

Sequences of acc-statements The semantics of a sequence of acc-statements is the standard composition of the functions specified by the semantics of the individual acc-statements:

$$[\![s_1, s_2, \ldots, s_n]\!]_\eta = [\![s_1]\!]_\eta \circ [\![s_2]\!]_\eta \circ \ldots \circ [\![s_n]\!]_\eta.$$

Note that since the acc-statements do not directly modify the value of accumulators, they each evaluate using the same snapshot of accumulator values.

Semantics of Acc-input Generation Phase We are now ready to formalize the semantics of the acc-input generation phase as a function

$$[\![\text{ACCUM } s_1, s_2, \ldots, s_n]\!]_\eta$$

that takes as input a context ctx and a binding table B and returns a pair of (global and vertex) acc-input maps that associate to each accumulator their cumulated bag of acc-inputs. See Figure 8, where $\eta^{\{\!|\!\}}_{gacc}$ denotes the map assigning to each global accumulator name the empty bag, and $\eta^{\{\!|\!\}}_{vacc}$ denotes the map assigning to each (vertex, vertex accumulator name) pair the empty bag.

$$
\begin{aligned}
[\![\text{ACCUM } ss]\!]_\eta(ctx, B) &= (\eta'_{gacc}, \eta'_{vacc}) \\
\eta'_{gacc} &= \bigcup_{A \text{ global acc name}} \{A \mapsto \biguplus_{\substack{\beta \in B, \\ (\_, \eta_{gacc}, \_) = [\![ss]\!]_\eta(\beta \triangleright ctx, \eta_{gacc}^{\{\!\{\}\!\}}, \eta_{vacc}^{\{\!\{\}\!\}})}} \eta_{gacc}(A)\} \\
\eta'_{vacc} &= \bigcup_{v \in V,\ A \text{ vertex acc name}} \{(v, A) \mapsto \biguplus_{\substack{\beta \in B, \\ (\_, \_, \eta_{vacc}) = [\![ss]\!]_\eta(\beta \triangleright ctx, \eta_{gacc}^{\{\!\{\}\!\}}, \eta_{vacc}^{\{\!\{\}\!\}})}} \eta_{vacc}(v, A)\}.
\end{aligned}
$$

Figure 8: The Semantics of the Acc-Input Generation Phase Is a Function $[\![]\!]_\eta$

$$
\begin{aligned}
reduce_{all}(ctx, \eta_{gacc}, \eta_{vacc}) &= ctx' \\
ctx' &= \mu_{vacc} \triangleright (\mu_{gacc} \triangleright ctx) \\
\mu_{gacc} &= \bigcup_{A \in Acc_g} \{A' \mapsto ctx(A),\ \ A \mapsto reduce_{A.\oplus}(ctx(A), \eta_{gacc}(A))\} \\
\mu_{vacc} &= \bigcup_{A \in Acc_v} \{A' \mapsto \bigcup_{v \in V} \{v \mapsto ctx(A)(v)\},\ \ A \mapsto \bigcup_{v \in V} \{v \mapsto reduce_{A.\oplus}(ctx(A)(v), \eta_{vacc}(v, A))\}\}
\end{aligned}
$$

Figure 9: The Semantics of the Acc-Aggregation Phase Is a Function $reduce_{all}$

6.12.2 Acc-Aggregation Phase

This phase aggregates the generated acc-inputs into the accumulator they are meant for.

We formalize this effect using the *reduce* function. It is parameterized by a binary operator $\oplus$ and it takes as inputs an intermediate value v and a bag B of inputs and returns the result of repeatedly $\oplus$-combining v with the inputs in B, in non-deterministic order.

$$
reduce_{\oplus}(v, B) = \begin{cases} v & , \ \textit{if } B = \{\!\{\}\!\} \\ reduce_{\oplus}(v \oplus i, B - \{\!\{i\}\!\}), & \textit{if } i \in B \end{cases}
$$

Note that operator $-$ denotes the standard bag difference, whose effect here is to decrement by 1 the multiplicity of i in the resulting bag. Also note that the choice of acc-input i in bag B is not further specified, being non-deterministic.

Figure 9 shows how the *reduce* function is used for each accumulator to yield a new accumulator value that incorporates the accumulator inputs. The resulting semantics of the aggregation phase is a function $reduce_{all}$ that takes as input the context, the global and vertex acc-input maps, and outputs a new context reflecting the new accumulator values, also updating the prior accumulator values.

Order-invariance The result of the acc-aggregation phase is well-defined (input-order-invariant) for an accumulator instance a whenever the binary aggregation operation $a.\oplus$ is commutative and associative. This is the case for built-in GSQL accumulator types SetAccum, BagAccum, HeapAccum, OrAccum, AndAccum, MaxAccum, MinAccum, SumAccum, and even AvgAccum (implemented in an order-invariant way by having the stored internal value be the pair of sum and count of inputs). Order-invariance also holds for complex accumulator types MapAccum and GroupByAccum if they are based on order-invariant accumulators. The exceptions are the List-, Array- and StringAccm accumulator types.

6.12.3 Putting It All Together

The semantics of the ACCUM clause is a function on (context, binding table)-pairs that preserves the binding table:

$$[\![\text{ACCUM } s_1, s_2, \ldots, s_n]\!](ctx, B) = (ctx', B)$$

where

$$ctx' = reduce_{all}(ctx, \eta_{gacc}, \eta_{vacc})$$

with η_{gacc}, η_{vacc} provided by the acc-input generation phase:

$$(\eta_{gacc}, \eta_{vacc}) = [\![\text{ACCUM } s_1, s_2, \ldots, s_n]\!]_{\eta}(ctx, B).$$

6.13 POST_ACCUM Clause Semantics

The purpose of the POST_ACCUM clause is to specify computation that takes place after accumulator values have been set by the ACCUM clause (at this point, the effect of the Acc-Aggregation Phase is visible).

The computation is specified by a sequence of acc-statements, whose syntax and semantics coincide with that of the sequence of acc-statements in an ACCUM clause:

$$[\![\text{POST\_ACCUM } ss]\!] = [\![\text{ACCUM } ss]\!].$$

The POST_ACCUM clause does not add expressive power to GSQL, it is syntactic sugar introduced for convenience and conciseness.

6.14 SELECT Clause Semantics

The semantics of the GSQL SELECT clause is modeled after standard SQL. It is a function on (context, binding table)-pairs that preserves the context and returns a table whose tuple components are the results of evaluating the SELECT clause expressions as described in Section 6.10.

Vertex Set Convention We model a vertex set as a single-column, duplicate-free table whose values are vertex ids. Driven by our experience with GSQL customer deployments, the default interpretation of clauses of form SELECT x where x is a vertex variable is SELECT DISTINCT x. To force the output of a vertex bag, the developer may use the keyword ALL: SELECT ALL x.

Standard SQL bag semantics applies in all other cases.

6.15 GROUP BY Clause Semantics

GSQL's GROUP BY clause follows standard SQL semantics. It is a function on (context, table)-pairs that preserves the context. Each group is represented as a nested-table component in a tuple whose scalar components hold the group key.

6.16 HAVING Clause Semantics

GSQL's HAVING clause follows standard SQL semantics. It is a function from (context,table) pairs to tables.

6.17 ORDER BY Clause Semantics

GSQL's ORDER BY clause follows standard SQL semantics. It is a function from (context,table) pairs to ordered tables (lists of tuples).

6.18 LIMIT Clause Semantics

The LIMIT clause inherits standard SQL's semantics. It is a function from (context,table) pairs to tables. The input table may be ordered, in which case the output table is obtained by limiting the prefix of the tuple list. If the input table is unordered (a bag), the selection is non-deterministic.

6.19 Query Block Statement Semantics

Let qb be the query block

```
SELECT DISTINCT? s INTO t
FROM        f
WHERE       w
ACCUM       a
POST-ACCUM  p
GROUP BY    g
HAVING      h
ORDER BY    o
LIMIT       l.
```

Its semantics is a function on contexts, given by the composition of the semantics of the individual clauses:

$$
\begin{aligned}
[\![qb]\!](ctx) &= \{t \mapsto T\} \triangleright ctx' \\
&\textit{where} \\
(ctx', T) &= [\![\text{FROM } f]\!] \\
&\quad \circ\ [\![\text{WHERE } w]\!] \\
&\quad \circ\ [\![\text{ACCUM } a]\!] \\
&\quad \circ\ [\![\text{GROUP BY } g]\!] \\
&\quad \circ\ [\![\text{SELECT DISTINCT? } s]\!] \\
&\quad \circ\ [\![\text{POST\_ACCUM } p]\!] \\
&\quad \circ\ [\![\text{HAVING } h]\!] \\
&\quad \circ\ [\![\text{ORDER BY } o]\!] \\
&\quad \circ\ [\![\text{LIMIT } l]\!](ctx)
\end{aligned}
$$

Multi-Output SELECT Clause Multi-Output queries have the form

```
SELECT      s1 INTO t1; ... ; sn INTO tn
FROM        f
WHERE       w
ACCUM       a
GROUP BY    g1; ... ; gn
HAVING      h1; ... ; hn
LIMIT       l1; ... ; ln.
```

They output several tables based on the same FROM, WHERE, and ACCUM clause. Notice the missing POST_ACCUM clause. The semantics is also a function on contexts, where the output context reflects the multiple output tables:

$$\begin{aligned}
[\![qb]\!](ctx) &= \{t_1 \mapsto B_1, \ldots, t_n \mapsto B_n\} \triangleright ctx_A \\
&\textit{where} \\
(ctx_A, B_A) &= [\![\text{FROM } f]\!] \\
&\circ\ [\![\text{WHERE } w]\!] \\
&\circ\ [\![\text{ACCUM } a]\!](ctx) \\
& \\
& \textit{for each } 1 \leq i \leq n \\
(ctx_A, B_i) &= [\![\text{GROUP BY } g_i]\!] \\
&\circ\ [\![\text{SELECT } s_i]\!] \\
&\circ\ [\![\text{HAVING } h_i]\!] \\
&\circ\ [\![\text{LIMIT } l_i]\!](ctx_A, B_A)
\end{aligned}$$

6.20 Assignment Statement Semantics

The semantics of assignment statements is a function from contexts to contexts.

For statements that introduce new global variables, or update previously introduced global variables, their semantics is

$$[\![type?\ gvar\ = expr]\!](ctx) = \{gvar \mapsto [\![expr]\!]^{ctx}\} \triangleright ctx.$$

For statements that assign to a global accumulator, the semantics is

$$[\![@@gAcc\ = expr]\!](ctx) = \{@@gAcc \mapsto [\![expr]\!]^{ctx}\} \triangleright ctx.$$

Vertex Set Manipulation Statements In the following, let s, s_1, s_2 denote vertex set names.

$$[\![s = expr]\!](ctx) = \{s \mapsto [\![expr]\!]^{ctx}\} \triangleright ctx.$$

$$[\![s = s_1 \texttt{ union } s_2]\!](ctx) = \{s \mapsto ctx(s_1) \cup ctx(s_2)\} \triangleright ctx.$$

$$[\![s = s_1 \texttt{ intersect } s_2]\!](ctx) = \{s \mapsto ctx(s_1) \cap ctx(s_2)\} \triangleright ctx.$$

$$[\![s = s_1 \texttt{ minus } s_2]\!](ctx) = \{s \mapsto ctx(s_1) - ctx(s_2)\} \triangleright ctx.$$

6.21 Sequence of Statements Semantics

The semantics of statements is a function from contexts to contexts. It is the composition of the individual statement semantics:

$$[\![s_1 \ldots s_n]\!] = [\![s_1]\!] \circ \ldots \circ [\![s_n]\!].$$

6.22 Control Flow Statement Semantics

The semantics of control flow statements is a function from contexts to contexts, conforming to the standard semantics of structured programming languages. We illustrate on two examples.

Branching statements

$$[\![\texttt{if}\ cond\ \texttt{then}\ stmts_1\ \texttt{else}\ stmts_2\ \texttt{end}]\!](ctx)$$
$$= \begin{cases} [\![stmts_1]\!](ctx), & \textit{if } [\![cond]\!]^{ctx} = \textit{true} \\ [\![stmts_2]\!](ctx), & \textit{if } [\![cond]\!]^{ctx} = \textit{false} \end{cases}$$

Loops

$[\![\texttt{while}\ cond\ \texttt{do}\ stmts\ \texttt{end}]\!](ctx)$
$$= \begin{cases} ctx, & \textit{if } [\![cond]\!]^{ctx} = \textit{false} \\ [\![stmts]\!] \circ & \\ [\![\texttt{while}\ cond\ \texttt{do}\ stmts\ \texttt{end}]\!](ctx), & \textit{if } [\![cond]\!]^{ctx} = \textit{true} \end{cases}$$

6.23 Query Semantics

A query Q that takes n parameters is a function

$$Q : \{\mathcal{D} \mapsto \mathcal{D}\} \times \mathcal{D}^n \to \mathcal{D}$$

where $\{\mathcal{D} \mapsto \mathcal{D}\}$ denotes the class of maps (meant to model contexts that map graph names to their contents), and $\mathcal{D}^n$ denotes the class of n-tuples (meant to provide arguments for the parameters).

Consider query Q below, where the p_i's denote parameters, the d_is denote declarations, the s_is denote statements and the a_is denote arguments instantiating the parameters. Its semantics is the function

$[\![\ \texttt{create query}\ Q\ (ty_1\ p_1, \ldots, ty_n\ p_n)\ \{$
$\quad d_1; \ldots; d_m;$
$\quad s_1; \ldots; s_k;$
$\quad \texttt{return}\ e;$
$\}$
$]\!]\ (ctx, a_1, \ldots, a_n) = [\![e]\!]^{ctx'}$

where

$$\begin{aligned} ctx' \quad = \quad & [\![d_1]\!] \circ \ldots \circ [\![d_m]\!] \circ [\![s_1]\!] \circ \ldots \circ [\![s_k]\!] \\ & (\{p_1 \mapsto a_1, \ldots, p_n \mapsto a_n\} \triangleright ctx) \end{aligned}$$

7 Conclusions

GSQL represents a sweet spot in the trade-off between abstraction level and expressivity: it is sufficiently high-level to allow declarative SQL-style programming, yet sufficiently expressive to specify sophisticated iterative graph algorithms and configurable DARPE semantics. These are traditionally coded in general-purpose languages like C++ and Java and available only as built-in library functions in other graph query languages such as Gremlin and Cypher, with the drawback that advanced programming expertise is required for customization.

The GSQL query language shows that the choice between declarative SQL-style and NoSQL-style programming over graph data is a false choice, as the two are eminently compatible. GSQL also shows a way to unify the querying of relational tabular and graph data.

GSQL is still evolving, in response to our experience with customer deployment. We are also responding to the experiences of the graph developer community at large, as TigerGraph is a participant in current ANSI standardization working groups for graph query languages and graph query extensions for SQL.

References

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet Wiener. The lorel query language for semistructured data. *Int. J. on Digital Libraries*, 1(1):68–88, 1997.

[2] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a web in your pocket? *IEEE Data Eng. Bull.*, 21(2):37–47, 1998.

[3] Diego Calvanese, Giuseppe De Giacomo, Maurizio Lenzerini, and Moshe Y. Vardi. Containment of conjunctive regular path queries with inverse. In *KR 2000, Principles of Knowledge Representation and Reasoning Proceedings of the Seventh International Conference, Breckenridge, Colorado, USA, April 11-15, 2000.*, pages 176–185, 2000.

[4] R. G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Management Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[5] Peter Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[6] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, and Dan Suciu. A query language for a web-site management system. *ACM SIGMOD Record*, 26(3):4–11, 1997.

[7] A. Gibbons. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

[8] W3C SparQL Working Group. Sparql, 2018.

[9] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. Green-marl: a DSL for easy and efficient graph analysis. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March 3-7, 2012*, pages 349–362, 2012.

[10] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[11] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. Querying graphs with data. *J. ACM*, 63(2):14:1–14:53, 2016.

[12] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. *SIAM J. Comput.*, 24(6):1235–1258, December 1995.

[13] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. Querying the world wide web. In *PDIS*, pages 80–91, 1996.

[14] Amit Singhal. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[15] Neo Technologies. Neo4j, 2018. https://www.neo4j.com/.

[16] Apache TinkerPop. The gremlin graph traversal machine and language, 2018. https://tinkerpop.apache.org/gremlin.html.

[17] Leslie G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.