Seamless Syntactic and Semantic Integration of Query Primitives over Relational and Graph Data

Authors: TigerGraph Inc Standard QL Working Group[1]
Status: Discussion paper
Date: October 31, 2018

This document presents TigerGraph's vision for simultaneous querying of graph and relational data in a fashion that brings out the commonalities between the two models. Towards easy adoption by SQL programmers, we advocate a syntax and semantics that start from standard SQL, extending it as seamlessly as possible to graph data. The extension is syntactically parsimonious, in the sense that we avoid the introduction of new keywords for concepts that already have an SQL counterpart.

The proposed language features

- The extension of the SQL FROM clause to support the specification of patterns to be matched against the graph, thus providing bindings for the pattern variables along with the standard SQL tuple variables.

- The uniform treatment of FROM variables in the remaining clauses, regardless of whether they were introduced as graph pattern variables or standard tuple variables

- Full compositionality in the sense that wherever standard SQL allows a nested subquery, this subquery may take graphs as input.

- Accumulators supporting 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 iterative graph algorithms.

The proposal is aligned with the requirements outlined in document sql-pg-2018-0011, "Oracle's Roadmap for SQL/PG 2020".

1. A Minimal Extension to the FROM Clause

In addition to standard SQL constructs, the FROM clause may mention atoms of general form

GraphName AS? pattern

where

[1] Current members are Alin Deutsch, Mingxi Wu and Yu Xu.

-- the AS keyword[2] 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 AS 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 detail our proposal for patterns in Section 6 but illustrate first with the following example.

Example 1. *Assume Company ACME maintains a human resource database stored in an RDBMS containing a relational table "Employee". It also has access to a graph named "LinkedIn" containing the professional network of LinkedIn users.*

The following query 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:

```
SELECT    e.email, e.name, count (outsider) as outsideConnections
FROM      Employee AS e,
          LinkedIn AS Person: p  -(Connection: c)-  Person: outsider
WHERE     e.email = p.email AND
          outsider.currentCompany NOT LIKE "ACME" AND
          c.since >= 2016
GROUP BY  e.email, e.name
```

Notice the pattern

Person: p -(Connection: c)- Person: outsider

to be matched against the LinkedIn graph (while numerous syntactic choices are possible for specifying patterns, in this document we adopt GSQL's style). The pattern variables are "p", "c" and "outsider", binding respectively to a vertex typed "Person", an edge typed "Connection" and a vertex typed "Person".

Once the pattern is matched, its variables can be used just like standard SQL tuple aliases. Indeed 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 sub-pattern

---

[2] Here, AS serves the same role as the MATCH keyword in the Discussion Paper sql-pg-2018-0042-initial-gql-proposal.pdf. In keeping with our parsimonious syntactic extension principle we opted for SQL's AS keyword.

-(Connection: c)-

denotes that the "Connection" relationship is symmetric, being represented by an undirected edge. (For the purpose of compatibility with the current status of this working group which has not yet tackled undirected edges, we can also interpret the arrow-less edge sub-pattern to match a directed edge regardless of its direction). □

The proposed syntax is aligned in spirit with the GRAPH_TABLE operator described in Change Proposal sql-pg-2018-0024r2, entitled "GRAPH_TABLE proposal" and with the MATCH operator described in Discussion Paper sql-pg-2018-0042-initial-gql-proposal.pdf. It preserves the semantics of the operators but drops the keywords, towards a seamless integration with SQL.

The proposed pattern syntax is also compatible with the proposal of specifying types as label sets, as in Discussion paper sql-pg-2018-0035. It is also compatible with the proposal (currently in working group discussions) to incorporate into patterns Boolean formulae over labels/types. One would write

(Employee | Intern) & !Secretary: p

to introduce a variable "p" ranging over vertices that are labeled/typed "Employee" or "Intern" but not "Secretary".

2. Multiple Patterns in FROM Clause Support Cross-Graph Joins

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

Example 2 (Cross-Graph and -Relation 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". Notice the join across the two graphs and the relational table in the query below:*

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

Also notice the arrowhead in the edge sub-pattern ranging over the Twitter graph,

-(tweets>)-

which matches only directed edges pointing from the "User" vertex to the "Tweet" vertex. □

3. Support for Nested Queries

The queries expressed using the extended FROM clause return tables, thus preserving compositionality. This yields the natural SQL semantics for nesting sub-queries wherever tables are expected.

We view Extension 1 above as key towards seamless integration of relational and graph querying capabilities. Extensions 2 and 3 follow naturally from Extension 1 and do not require the introduction of new keywords. Moreover, they enable a modular specification of the semantics based on standard relational tables.

4. 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.

The accumulator abstraction was introduced in the Green-Marl system [HCS+12] and it was adapted as high-level first-class citizen in GSQL. Accumulators support

- the concise specification of multiple simultaneous aggregations by distinct grouping criteria, and
- the computation of vertex-stored side-effects to speed up multi-pass and iterative algorithms.

Accumulators are GSQL's design choice for supporting the reduce operations of the map-reduce paradigm (or fold operations in the spirit of the classical map-fold paradigm from functional programming).

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. Alternatively, *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 composition operation $\oplus$: V x I $\rightarrow$ V. Accumulators implement two assignment operators. Denoting with Acc.val the current internal value of accumulator Acc,

- Acc = i sets Acc.val to the provided input i;
- Acc += i aggregates the input i into Acc.val using $\oplus$, i.e. sets Acc.val to Acc.val $\oplus$ i.

In the following we use a GSQL-inspired syntactic style to illustrate accumulator usage. For a comprehensive documentation on GSQL accumulators, see the developer's guide at http://doc.tigergraph.com. Here, we explain accumulators by example.

Example 3 (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 edge of type "Bought" from the "Customer"-typed vertex modeling c to the "Product"-typed 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.*

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.

Note that writing this query in standard SQL fashion requires performing two GROUP BY operations, one by customer and one by product. Towards conciseness (and to facilitate optimization), we propose the following GSQL-inspired alternative.

We define a vertex accumulator type for each kind of revenue. The revenue for product p will be aggregated at the vertex modeling p by vertex accumulator "revenuePerProduct", while the revenue for customer c will be aggregated at the vertex modeling c by the vertex accumulator "revenuePerCustomer". The total sales revenue will be aggregated in a global accumulator called "totalRevenue".
We declare these accumulators as follows:

SumAccum<float> @revenuePerProduct, @revenuePerCustomer, @@totalRevenue,

where 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).

Using accumulators, the multi-grouping query above is concisely expressible as

```
WITH
   SumAccum<float> @revenuePerProduct, @revenuePerCustomer, @@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.@revenuePerCustomer += salesPrice,
             p.@revenuePerProduct += salesPrice,
             @@totalRevenue += salesPrice;
END
```

Note the definition of the accumulators using the WITH clause in the spirit of standard SQL definitions.

Also note the novel ACCUM clause, which specifies 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.@revenuePerCustomer" refers to the vertex accumulator instance located at vertex "c".

While the query outputs the customer vertex ids, in this example we are interested in its side-effect of annotating vertices with the aggregated revenue values and of computing the total revenue using accumulators. □

The scope of the accumulator declaration may cover a sequence of queries, in which case the accumulated values computed by a query can be read (and further modified) by subsequent queries, thus achieving powerful side-effect composition.

Semantics. The semantics of queries with an ACCUM clause 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 once. 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 in order 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 3.[3]

We propose to include into the language a list of predefined accumulator types. TigerGraph's experience with the deployment of GSQL has yielded the short list from Appendix A that covers most use cases we have encountered in customer engagements.

We additionally propose allowing users to define their own accumulators by implementing a simple interface that declares the binary combiner operation used for aggregation of inputs into the stored value. This will lead to an extensible query language and will facilitate the development of accumulator libraries.

Exporting accumulators into tables. Towards compositionality with SQL queries, we also propose a way to output the accumulated values into tables.

If one desires to output the accumulator values in tuples, one can refer to them in the SELECT clause as done for regular attributes (using the syntax illustrated in the ACCUM clause in Example 5 below).

More flexibly, a GSQL query Q can be a sequence of blocks, each computing a table (possibly including accumulated values), and these tables can be exported to the context in which Q appears by defining these tables via an SQL-style WITH clause. The WITH clause has general form

[3] Input-order invariance holds for most predefined accumulators from the GSQL library (Sum<numeric>, Or, And, BitwiseOr, BitwiseAnd, Avg, Set, Map, Heap). The exceptions are List<T>, Array<T> and Sum<string> (string concatenation), which are sensitive to the order of operations.

WITH
  (T_1, T_2,..., T_n) AS { <Query Q computing tables T_1, T_2,... T_n> }
BEGIN
  <Query Q_c consuming tables T_1, T_2,..., T_n>
END

Example 4. *Suppose we wish to output the aggregates computed in Example 3 into a table CustRev associating customer names with the revenue from toy sales, a table ToyRev associating toy names with their revenue, and a table TotalRev containing the total revenue, so a query Q can consume these tables. We express this as*

```
WITH
  (CustRev, ToyRev, TotalRev) AS {
        Query from Example 3;

        SELECT c.name, c.@revenuePerCustomer INTO CustRev
        FROM   Customer:c;

        SELECT t.name, t.@revenuePerProduct INTO ToyRev
        FROM   Product t
        WHERE t.category="Toys";

        TotalRev = @@totalRevenue;
 }
 BEGIN
        Q
 END
```

□

Another prominent use of accumulators is in support of multi-pass and iterative graph algorithms. It involves computing per-vertex aggregates and storing them at their vertices to facilitate the next passes of the algorithm. See https://doc.tigergraph.com/GSQL-Tutorial-and-Demo-Examples.html for a concise specification of the classical PageRank alogorithm in GSQL, making crucial use of vertex accumulators that store the vertex rank at each iteration.

5. The POST-ACCUM Clause

The POST-ACCUM clause specifies per-vertex computation that takes as input the result of the ACCUM-specified aggregation.

Example 5
Assume we wish to write a simple toy recommendation system for customer 1. 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 customer c is weighted by the similarity of c to customer 1. This similarity is the standard log-cosine similarity, which reflects how many toys customers 1 and c 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 following query computes for each customer c their log-cosine similarity to customer 1, storing it in c'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 c's vertex accumulator @inCommon. The POST-ACCUM clause then computes the logarithm and stores it in c's vertex accumulator @lc.

```
WITH
    SumAccum<float> @lc, @inCommon, @rank;
BEGIN

    SELECT        DISTINCT  c  INTO OthersWithCommonLikes
    FROM          SalesGraph AS Customer:one -(Likes>)- Product:t -(<Likes)- Customer:c
    WHERE         one.id = 1 AND c.id <> 1 AND t.category = "Toys"
    ACCUM         c.@inCommon += 1
    POST-ACCUM  c.@lc = log (1 + c.@inCommon);

# We can next compute the rank of each toy t by adding up the similarities of all other customers besides
# 1 who like t. We output the top 10 recommendations:

    SELECT       t.name, t.@rank AS rank
    FROM         OthersWithCommonLikes AS r,
                 SalesGraph AS r.c -(Likes>)- Product:t
    WHERE        t.category = "Toy" AND c.id <> 1
    ACCUM        t.@rank += c.@lc
    ORDER BY   t.@rank DESC
    LIMIT        10;

END
```

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

6. Patterns Specified by Direction-Aware Regular Path Expressions (DARPEs)

We follow the tradition instituted in the mid-90s by a line of classic work on querying graph (a.k.a. semi-structured) data which yielded such reference query languages as WebSQL [MMM96], StruQL [FFLS97] and Lorel [AQM+97]. 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.

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

rpe → _ | EdgeType | ‘(‘ rpe ‘)’ | rpe’*’bounds? | rpe ‘.’ rpe | rpe ‘|’ rpe
bounds → N? ‘..’ N?

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 alternation. The ‘*’ terminal symbol is the standard Kleene star specifying several 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).

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

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 node);
- <E, denoting a hop along an incoming E-edge (from target to source node).

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>|<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.

A well-known semantic issue arises from the tension between RPE expressivity and well-definedness.

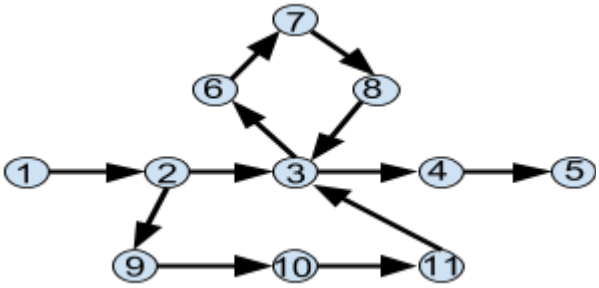
Regarding expressivity, applications need to sometimes specify reachability in the graph via RPEs comprising unbounded Kleene repetitions. They also need RPE patterns to be interpreted under bag semantics. That is, a pattern :s -(RPE)- :t should have as many matches of variables (s,t) to a given pair of nodes (n1,n2) as there are distinct paths from n1 to n2 satisfying the RPE. In other words, the multiplicity of the match of (s,t) to (n1,n2) should be the number of such distinct paths. This is needed to support graph analytics that aggregate data across paths via multiplicity-sensitive operators (e.g. count, sum, average).

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 the example Person:p1 -(Knows*)- Person:p2 in a social network with cycles involving the “Knows” relationship.

Legal paths Traditional solutions limit the kind of paths that are considered legal, so as to yield a finite number thereof. Two popular approaches allow only paths with no repeated vertices/no repeated edges. However under these definitions of path legality the evaluation of RPEs is in general notoriously intractable (even checking whether a pair of nodes is connected by a path that satisfies the RPE without counting such paths is NP-hard in the size of the entire graph [MW95,LMV16]). For this reason we suggest that the standard not mandate intractable semantics or at least, should these semantics be included, that they not be the only options.

We propose the *shortest-path legality* criterion. That is, among the paths from *s* to *t* satisfying a given DARPE, we consider legal only the shortest ones. The resulting RPE semantics is known to yield tractable evaluation [AAB+17], which extends to DARPEs.

Example 6. *To contrast the various semantic flavors, consider the graph topology below, assuming that all edges are typed "E" and all vertices are typed "V". Among all paths from source node 1 to target node*



5 that satisfy the DARPE "E>", there are*

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

Therefore, pattern V:s -(E>)- V:t will return the binding (s→ 1,t→ 5) with multiplicity 2, 3, or 1 under the non-repeated-vertex, non-repeated-edge respectively shortest-path legality criterion.* □

Bibliography

[AQM+97] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, Janet Wiener: *The Lorel Query Language for Semistructured Data.* Int. J. on Digital Libraries 1(1): 68-88 (1997)

[AAB+17] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc.
Foundations of Modern Query Languages for Graph Databases
ACM Comput. Surv. 50(5): 68:1-68:40 (2017)

[FFLS97] Mary F. Fernandez, Daniela Florescu, Alon Y. Levy, Dan Suciu:
A Query Language for a Web-Site Management System.
SIGMOD Record 26(3): 4-11 (1997)

[HCS+12] Sungpack Hong, Hassan Chafi, Eric Sedlar and Kunle Olukotun:
Green-Marl: a DSL for easy and efficient graph analysis.
ASPLOS 2012: 349-362

[LMV16] Leonid Libkin, Wim Martens, Domagoj Vrgoc.
Querying Graphs with Data.
Journal of the ACM 63(2): 1-53, 2016.

[MMM96] Alberto O. Mendelzon, George A. Mihaila, Tova Milo:
Querying the World Wide Web.
PDIS 1996: 80-91

[MW95] Alberto O. Mendelzon and Peter T. Wood.
Finding Regular Simple Paths in Graph Databases.
SIAM J. Comput. 24, 6 (1995), 1235–1258.

Appendix A: GSQL's Main Predefined Accumulator Types

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. Computes minimum value of its inputs of type O.
MaxAccum<O>	As above, for max aggregation.
AvgAccum<N>	where N is a numeric type. Computes average of its inputs of type N.
OrAccum	Aggregates its boolean inputs using logical disjunction.
AndAccum	Aggregates its boolean inputs using logical conjunction.
MapAccum<K,V>	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])

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 heap, and
- field_1 [ASC|DESC], field_2 [ASC|DESC], …, field_n [ASC|DESC]
  specifies a lexicographic order for sorting the tuples in the heap (each field may be used In ascending or descending order).

For details on GSQL's accumulators, see
https://doc.tigergraph.com/GSQL-Language-Reference-Part-2---Querying.html.