Aggregation Support for Modern Graph Analytics in TigerGraph

Alin Deutsch
deutsch@cs.ucsd.edu
UC San Diego and TigerGraph

Yu Xu
yu@tigergraph.com
TigerGraph

Mingxi Wu
mingxi.wu@tigergraph.com
TigerGraph

Victor E. Lee
victor@tigergraph.com
TigerGraph

ABSTRACT

We describe how GSQL, TigerGraph's graph query language, supports the specification of aggregation in graph analytics. GSQL makes several unique design decisions with respect to both the expressive power and the evaluation complexity of the specified aggregation. We detail our design showing how our ideas transcend GSQL and are eminently portable to the upcoming graph query language standards as well as to existing pattern-based declarative query languages.

CCS CONCEPTS

• **Information systems → Query languages for non-relational engines**; **Graph-based database models**.

KEYWORDS

Graph Databases; Graph Query Languages; Aggregation

ACM Reference Format:
Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3386144

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3386144

1 INTRODUCTION

The evolution of declarative query languages for graph data has recently reached an inflection point marking a proliferation of commercial systems [8, 26, 27], the G-CORE research manifesto [3], open-source projects [28], and two ISO/ANSI standardization projects: SQL/PGQ [22], a standard for extension of SQL with graph query capabilities and with the ability to view relational data as graphs, and GQL [23], a just-approved standard project aimed at standardizing a graph query language for arbitrary property graphs (not necessarily views of relational data).

The main focus of the research manifesto and the standardization process is on the paradigm of specifying queries declaratively via patterns, from the point of view of expressivity and semantics.

This submission focuses on a complementary topic, namely that of aggregating the values retrieved via pattern matching. Our interest is prompted by extensive experience with real customer use cases showing the importance of expressive and flexible aggregation capabilities for modern graph analytics support.

The manifesto, the standard, and the existing graph query languages treat aggregation conventionally, in the following sense. The query pattern yields a *match table* whose column names are given by the pattern variables, and in which each row provides a binding of these variables to graph elements (vertices, edges, paths), where the binding conforms to a match of the pattern against the graph. In all cases, the match table is aggregated using the equivalent of a conventional SQL GROUP BY clause.

In this work we describe an alternative paradigm for specifying aggregation in declarative queries, which we call the *accumulator-based aggregation paradigm*. The paper substantiates our claims that the accumulator-based specification of aggregation has both expressivity and performance advantages. From an expressivity point of view, the accumulator-based specification of aggregation

- supports a powerful composition effect by allowing query blocks to compute and attach state to the graph as a whole and to its individual vertices, and subsequent query blocks to read/modify it;
- supports the inclusion of aggregation results into the computed state, whose specification subsumes the expressive power of conventional aggregation;
- can be significantly more concise than conventional aggregation;
- synergizes with control flow primitives to express an important class of iterative graph algorithms that are left out by the current standard drafts and the representative languages informing them, and whose specification in state-of-the-art products complicates application development by forcing the addition of a client process, imperatively programmed using a low-level general-purpose programming language like C++/Java/Python.

Performance-wise, accumulator-based aggregation

- narrows the gap between declarative specification and efficient query plans that achieve single-pass aggregation of the same data according to multiple disjoint grouping criteria;
- supports iterative graph algorithms by enabling, within the same server process, cross-iteration query block composition via the contents of accumulators. Contrast this with the alternative of implementing the loop in a separate client process running a C++/Java/Python program that repeatedly emits the iterated query using a JDBC-style interface. This would require either the transmission of state between query server and client process (at high communication overhead per iteration), or making state persistent at the server between JDBC calls and re-joining vertices with their associated state on each JDBC call (at high server processing overhead).
- is particularly well-suited to parallel graph processing, enabling several graph traversal threads to proceed in parallel, synchronizing via the accumulators.

Our design is underpinned by the following concrete technical contributions:

- We show how to harmonize the paradigm of declarative pattern-based graph querying with the paradigm of accumulation-based aggregation by providing a declarative semantics of a pattern-based graph query language enriched with accumulator-based aggregation.
- We discuss the interference of pattern match and aggregation semantics, define a space of possible design choices, survey state of the art solutions, and present the one adopted uniquely by TigerGraph. Essentially, this semantics calls for limiting the paths matched by a pattern to *only the shortest ones* (in contrast to alternative state-of-the-art matching semantics, which also limit matched paths but use other criteria such as disallowing repeated vertices/edges).
- We identify a large and practically relevant class of queries that use accumulators and patterns but do not bind variables to entire paths in the graph (and don't simulate such variables by accumulating paths in accumulators). This class admits polynomial-time execution under TigerGraph's all-shortest-paths semantics. Counter-intuitively, this result holds even when the pattern matches exponentially many paths, which all-shortest-paths semantics allows. The result exploits the fact that it is possible to avoid the materialization of the exponentially many paths, instead counting them in polynomial time and using these counts to short-cut aggregation by generating appropriate inputs into the accumulators. This result is in stark contrast to the exponential-time evaluation yielded by the semantics of the other languages in circulation (a prominent representative is the Cypher language's default semantics, in which patterns match only paths with non-repeated edges).
- We confirm experimentally that TigerGraph evaluates queries from the tractable class in polynomial time, even in the worst case when exponentially many paths match the query pattern, and that alternative semantics such as Cypher's non-repeated edge semantics indeed yield exponential-time evaluation even when the legal paths – and therefore the query results – coincide under the two semantics. (We use Neo4j's engine as reference for the non-repeated edge semantics.)
- More surprisingly, our experiments show that Neo4j's engine still evaluates in exponential time even when we request matching under the tractable all-shortest-paths semantics (which Neo4j supports, but as it turns out, sub-optimally). This measurement, as well as personal communication with members of the standard committee, convinced us that the tractability result we present here is ignored by the industrial graph querying community. We seek to disseminate it in this forum as a public service.
- Keeping the matching semantics unchanged, we experimentally compare accumulator-based with conventional SQL-style aggregation on the reference benchmark provided by the leading independent graph benchmarking authority, the Linked Data Benchmarking Consortium (LDBC) [16]. We show that, on graph sizes ranging from 1GB to 1TB, accumulators speed up aggregation by a factor of up to 3x when compared to SQL-style aggregation.

Applicability beyond GSQL. While GSQL is the vehicle we use to illustrate the discussed aggregation-related concepts, these are eminently portable to the upcoming graph query language ANSI/ISO standards and particularly relevant to both of them, as they are currently under construction. TigerGraph is an active member of both standard bodies, to which we have submitted our ideas as official contributions. Our ideas are also relevant to existing languages which, together with GSQL, currently inform the standardization debate (Cypher [25], G-CORE [3], PGQL [29] and SparQL [14]).

2 DECLARATIVITY VIA PATTERNS

GSQL supports declarative patterns for specifying paths in the graph. In doing so, we follow the tradition instituted by a line of classic work on querying graph data (at the time known as semi-structured data), which yielded such reference query languages as WebSQL [18], StruQL [12] and Lorel [1]. Common to all these languages is the ability to specify patterns that match paths whose structure is constrained by a regular path expression (RPE). This tradition is so well established that it is preserved in current query language standards, such as the W3C SparQL 1.1 [14] standard for querying RDF graphs and the XPath [30] standard for querying XML graphs, in the upcoming SQL/PGQ [22] and GQL [23] standards, and in the G-CORE manifesto [3].

Example 1 (**Joining across 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 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. The lack of an arrowhead accompanying the edge subpattern -(Connected: c)- requires the matched "Connected" edge to be undirected.* □

Example 1 illustrates a pattern that matches paths consisting of single "Connected" edges. In general, patterns can specify multi-edge paths using regular path expressions.

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* (or *match*) 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.

Direction-Aware Regular Path Expressions. To date, only TigerGraph's and the upcoming GQL standard's data model support graphs that mix directed and undirected edges (the SQL/PGQ standard and the other products it is informed by work on directed graphs only). To the best of our knowledge, GSQL is the only product to feature an extension of the RPE formalism to support mixed-kind edges. We call this extension *Direction-Aware RPEs (DARPEs)*.[1]

DARPEs. DARPEs extend RPEs with the ability to specify the edge kind (directed vs undirected) and, for directed edges, to specify their orientation.

Example 2 (**Mixed edge kind DARPE**). *The DARPE*

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

matches paths that start by traversing an outgoing E-edge, followed by a sequence of zero or more traversals of either outgoing F-edges or incoming G-edges, next by the traversal of an undirected H-edge and finally ending in the traversal of an incoming J-edge. □

To specify the syntax of DARPEs, we reuse the RPE grammar above, allowing the terminal symbol EdgeType to range over the alphabet of direction-adorned edge types defined as follows: for each edge type E, the *direction-adorned alphabet* includes the symbols E, $E{>}$, and ${<}E$.

The notion of satisfaction of a DARPE by a path extends classical RPE satisfaction in the natural way: given a path p oriented from source node x to target node y, an E-edge e on p is adorned with

- the symbol $E{>}$ if e is directed in the direction of p,

[1] The upcoming GQL standard has recently decided that it will also support co-existence of directed and undirected, but since the initial design of patterns was inspired by the directed-only graph variety, at the time of this writing GQL pattern syntax has not yet been extended to treat undirected edges as first-class citizens. Since neither of the other languages supported by the vendors on the standard body supports such syntax, we have submitted our DARPE design as a proposal to the standard body and active debate on this topic is underway.

```
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 1, Illustrates DARPE-based Patterns and Joins Across Relational Table and Graph

- the symbol $<E$ if e is directed in the opposite direction, or
- the symbol E if e is undirected.

We say that path p satisfies a DARPE D if p's edge adornments spell out a word in the language accepted by D when viewed as a standard regular expression over the alphabet of direction-adorned edge types. When p satisfies D, we also say that p *is a match for* D, and that D *matches* p.

3 AGGREGATION VIA ACCUMULATORS

In Example 1, we specified aggregation in the conventional SQL-inspired style, which is adopted (modulo superficial syntactic variations) by virtually all major declarative graph query languages in circulation, whether they address unrestricted graphs (Neo4j's Cypher [25], Oracle's PGQL [29], W3C's SparQL [14]) or the special case of tree-shaped graphs corresponding to JSON-class nested data models (like Facebook's GraphQL [11], CouchBase's N1QL [7], UC San Diego's SQL++ [20], Amazon's PartiQL [2]).

We now introduce an alternative means to specify aggregation, via the concept of accumulators. As we show below, the benefits of doing so include the remarkably high-level and concise specification of (i) single-pass computation of multiple aggregations of the same data by distinct grouping criteria, (ii) the computation of vertex-stored side effects to support a large class of multi-pass and iterative algorithms, and (iii) straightforwardly parallelizable aggregation. [2]

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

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

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

[2]One may ask in what sense a declarative specification supports a certain kind of evaluation, given that an optimizer can in theory choose any query plan. Our answer stresses the difference between theory and practice.

Accumulators implement two assignment operators. Denoting with a.val the internal value of accumulator a,

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

Accumulator Types. GSQL offers several built-in accumulator types. TigerGraph's experience with the deployment of GSQL has yielded the short list below, that covers most use cases we have encountered in customer engagements. Notice that accumulator types are parameterized by the type of the data they aggregate, while the binary combinator operation is encoded in the name.

`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.

`Set-, Bag-, Array- , ListAccum<T>` insert their inputs of type `T` into a collection of corresponding kind.

`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_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).

For details on GSQL's accumulators and more supported types, see the online documentation.

Accumulator Declarations. Accumulator declarations define the accumulator instances to be created by a query, specifying their type and name. When prefixed by a single `@` symbol, a name denotes vertex-attached accumulators (one instance per vertex) while accumulator names prefixed by `@@` denote a global accumulator (a single instance per query).

Example 3 (Accumulator Declaration). *The accumulator declaration below defines a global accumulator called* `totalRevenue` *(a single instance), a family of vertex-attached accumulators called* `revenuePerToy` *(one instance per vertex), and a family of vertex-attached accumulators called* `revenuePerCust` *(one instance per vertex). They all share the same type, which specifies that all instances hold an internal floating point value and expect floating point inputs aggregated using the floating point addition operation.*

```
SumAccum<float> @@totalRevenue,
                @revenuePerToy,
                @revenuePerCust;
```

□

ACCUM Clause. We add this novel clause to SQL to specify assignment to accumulators. It consists essentially of a list of assignments of form $A+ = E$, where A is an accumulator and E an expression depending on the variables introduced by the FROM clause pattern. We describe its semantics by example first, detailing it in Section 4.3.

Example 4 (Single-Pass Multi-Aggregation 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.*

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.

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 2).*

Note the leading accumulator declaration, explained in Example 3. 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 computed once and aggregated twice, by input into both the `revenuePerCust` *accumulator instance attached to the* `Customer` *vertex denoted by variable c, and into the* `revenuePerToy` *accumulator instance attached to the* `Product` *vertex denoted by variable p. The "+=" operator expressed the input operation.* □

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;
       @@totalRevenue AS rev INTO Total
```

This query populates three tables, PerCust and PerToy and Total. □

Extensible Accumulator Library. In addition to pre-defined accumulators, 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 facilitates the development of accumulator libraries towards an extensible query language.

4 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 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 and parallel execution. See the extended version [9] for the formal GSQL semantics, described informally below.

```
SumAccum<float> @revenuePerToy, @revenuePerCust, @@totalRevenue;

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;
```

Figure 2: Query for Example 4, Illustrates Single-Pass Treeway Aggregation via Accumulators

```
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 3: Recommender Query for Example 6, Illustrates Composition via `@lc` Accumulator

```
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 4: PageRank Query for Example 7, Illustrates Cross-Iteration Composition via Accumulators

4.1 FROM Clause

The FROM clause defines a *binding table* whose columns are named after the variables occurring in the FROM clause patterns. Each row of the binding table corresponds to a *binding* β of the variables, i.e. a function from variables to vertices or edges, such that for each conjunct S :s $-$ (d) $-$ T :t of the FROM clause, where d is a DARPE mentioning edge variables $e_1, \ldots, e_k$, vertex $\beta(s)$ has type S, vertex $\beta(t)$ has type T, and $\beta(e_1), \ldots, \beta(e_k)$ correspond to edges on a path from $\beta(s)$ to $\beta(t)$ that satisfies d [9].

4.2 SQL-Borrowed Clauses

The clauses borrowed from SQL (SELECT, WHERE, GROUP BY, HAVING, ORDER BY, LIMIT) inherit SQL semantics

when interpreting each occurrence of a global accumulator @@a as a constant equal to the internal value of a, and each vertex-attached accumulator reference v.@a by interpreting v as a tuple alias and @a as its attribute named a.

4.3 ACCUM Clause

The effect of the ACCUM clause is to modify accumulator values. It is executed exactly once for every variable binding yielded by the WHERE clause (every row of the binding table). 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 *Map Phase*, all acc-executions compute accumulator inputs starting from the same accumulator value snapshot. After the Map Phase completes, the *Reduce Phase* aggregates the generated inputs into the accumulators they are destined for. The result of the Reduce Phase is a new snapshot of the accumulator values.

Order-invariance. The result of the Reduce 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 certainly the case for addition, which is used in Example 4. In general, order-invariance applies to the built-in accumulator types `Set-`, `Bag-`, `Heap-`, `Or-`, `And-`, `Max-`, `Min-`, `Sum-`, and even to `AvgAccum` (implemented in an order-invariant way by having the storing computing internally the pair of sum and count of inputs). Order-invariance also holds recursively for the `MapAccum` accumulator type if the nested accumulators are in turn order-invariant. The exceptions to order-invariance are the `List-`, `Array-` and `SumAccum<string>` accumulator types.

Potential for Parallelization. The snapshot semantics is compatible with bulk-synchronous parallel execution of GQL queries, a fact that we exploit in implementation for performance reasons while guaranteeing deterministic semantics in all order-invariant use cases. If the user decides to deploy the three order-dependent accumulator types, it is with the understanding that the result is in general non-deterministically ordered. Users typically accept this case in applications that sample a random solution among a large space of candidates (such as exhibiting any one among the multitude of paths connecting two given nodes, as proof of connectivity).

4.4 POST_ACCUM Clause

The purpose of the POST_ACCUM clause is to specify computation that takes place after accumulator values have been set by the ACCUM clause. 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. The POST_ACCUM clause does not add expressive power to GSQL, it is syntactic sugar introduced for convenience and conciseness.

5 COMPOSITION VIA ACCUMULATORS

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. This enables powerful composition effects that support the concise specification of multi-pass algorithms, with each pass specified declaratively.

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 [24], 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 3. 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-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 illustrates the POST_ACCUM clause (recall Section 4.4) as a convenient way to post-process accumulator values after the ACCUM clause finishes computing them.

Iterated Composition. GSQL's design reflects our philosophy that a graph query language should admit declarative pattern-based specification whenever possible, yet be sufficiently expressive to specify the sophisticated iterative algorithms required by modern graph analytics (e.g. PageRank [5], shortest-paths [13], weakly connected components [13], recommender systems, etc.).

Traditionally, these algorithms are coded in general-purpose programming languages like C++ and Java and offered as built-in function calls in existing languages. Consequently, whenever an application requires a variation of a built-in graph algorithm that is not expressible by manipulating the provided parameters, a new implementation needs to be developed outside the query language, as a user-defined function (UDF).

While GSQL supports UDFs written in C++, our customer engagements have shown that the need for them is greatly diminished by minimally extending the declarative core of the language with two control flow primitives (if-then-else and the while loop). We illustrate the loop by expressing the classic PageRank [5] algorithm in GSQL.

Example 7 (PageRank). *Figure 4 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.* □

6 INTERFERENCE OF PATTERN AND AGGREGATION 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 express 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 pair (n_1, n_2) in the bag of matches of the pattern.

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.

Example 8 (Infinitely many matching paths). *The pattern* Person: p1 -(Knows>*)- Person: p2 *matches an infinity of distinct paths in a social network with cycles through "Knows" edges.* □

Gremlin's default semantics [28] falls in this category, thus allowing developers to specify potentially non-terminating graph traversals.

6.1 Path Legality Flavors

State-of-the-art solutions limit the kind of paths that are considered legal, so as to yield a finite number of legal paths in any graph, and thus a finite number of possible pattern matches.

Well-defined, Aggregation-friendly, but Intractable. Two popular approaches allow only paths with *non-repeated vertices* (as illustrated in Gremlin tutorials [3]) or with *non-repeating edges* (this is the default semantics of Cypher patterns [25]). 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) is NP-hard, and counting such paths is #P-complete [17, 19]. Since these worst-case complexity lower bounds refer to the size of the data, the resulting semantics does not always scale to large graphs.

Well-defined, Tractable but Aggregation-Unfriendly. The SparQL [14] W3C-standardized query language for RDF graphs adopts the following semantics: SparQL regular path expressions that are Kleene-starred are interpreted as boolean *tests* whether any path exists between two vertices, without counting how many such paths exist. This yields a multiplicity of 1 on the pair of path endpoints. This design choice is a twist on a semantic variation introduced in [4] to guarantee tractability, namely the non-deterministic choice of one among the possible shortest paths connecting a given pair of vertices. These semantics however defeat the purpose of supporting aggregation, and we discuss them no further given our focus on aggregation.

Well-defined, Aggregation-friendly and Tractable. 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, where the length of a path is the count of its edges. There can of course exist several shortest paths between a given pair of vertices, due to ties. The advantage of this design choice is that checking existence of a shortest path that satisfies a DARPE and connects a given pair of vertices, and even counting all such shortest paths, is tractable, i.e. has polynomial data complexity (see Theorem 6.1 below).

Example 9 (Contrasting Legality Flavors). *To contrast the various path legality flavors, consider the graph G_1 in Figure 5, 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*

[3] While Gremlin's default semantics allows all unrestricted paths (and therefore possibly non-terminating graph traversals), virtually all the documentation and tutorial examples involving unbounded traversal use non-repeated-vertex semantics (by explicitly invoking a built-in `simplePath` predicate to filter paths).

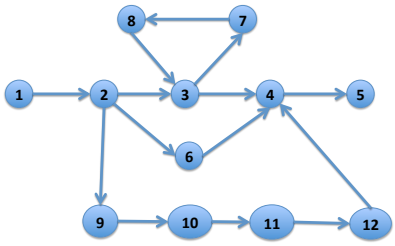

Figure 5: Graph G_1 for Example 9

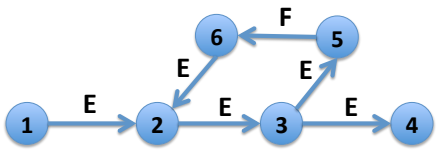

Figure 6: Graph G_2 for Example 10

- *Infinitely many unrestricted paths, depending on how many times they wrap around the 3-7-8-3 cycle (as per the default gremlin semantics);*
- *Three non-repeated-vertex paths, 1-2-3-4-5, 1-2-6-4-5, and 1-2-9-10-11-12-4-5 (as per the gremlin query style used in the Tinkerpop tutorials);*
- *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 (as per the default Cypher semantics);*
- *Two shortest paths, 1-2-3-4-5 and 1-2-6-4-5, (as per the default GSQL semantics).*

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 1.1 semantics, the multiplicity is 1. □

While in Example 9 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, as shown in Example 10 below, where shortest-path yields strictly more matches than the two non-repeating semantics.

Example 10 (More Shortest-Path than Non-Repeating Matches). *Consider Graph G_2 from Figure 6, and the pattern*

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

and note that it does not match any path from vertex 1 to vertex 4 under either non-repeated vertex or non-repeated edge semantics, while it does match one such path under shortest-path semantics: 1-2-3-5-6-2-3-4, which repeats both the vertices 2 and 3, and the edge between them. □

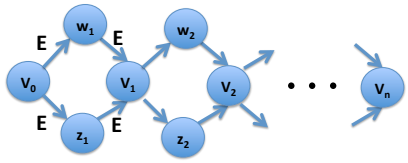

Figure 7: Diamond Chain Graph for Example 11

In general, the different classes of legal paths are incomparable with respect to inclusion [4]. However they all coincide on large classes of graphs, such as tree- and forest- shaped graphs, as well as on many other acyclic graphs. Moreover, all of them can define sets of paths of cardinality exponential in the size of the graph.

Example 11 (Coinciding Semantics, Exponential-Size Match Set). *Consider the well-known directed graph in Figure 7, which consists of a chain of diamond-shaped graphs that connect vertex v_i to vertex v_{i+1} via two paths, for each $0 \leq i \leq n$. Let DARPE d be E>∗, which denotes paths consisting exclusively of E edges, all oriented in the same direction. Observe that non-repeating vertex, non-repeating edge and all-shortest-paths semantics coincide on this graph, yielding the same set of paths. Also notice that for every $1 \leq k \leq n$, there are 2^k paths from v_0 to v_k and satisfying d.* □

Fixed-Unique-Length Patterns. The pattern match semantic flavors in circulation today constitute various ways to solve the problem of ensuring finitely many matches when the graph contains cycles and the pattern specifies variable-length paths via the Kleene star operator of unbounded repetition. Each flavor introduces its own limitation on the admissible paths, and in many applications this limitation is artificial. As shown above, the limitations are incomparable, in the sense that for each flavor there are graphs where one flavor admits paths that others do not.

The all-shortest-paths flavor stands out as the only one to not impose any restrictions on paths when the pattern belongs to a widely used class which we call *fixed-unique-length*. These are Kleene-free, and specify paths of a unique length that can be read off from the pattern. They are constructed by unrestricted application of the pattern concatenation operator ".", while the pattern disjunction operator "|" applies only to patterns with the same length. A typical, widely-used representative pattern subclass has general form $D_1.D_2.\cdots.D_n$ where each D_i is a possibly direction-annotated edge type-/wildcard (or a disjunction thereof). Notice that the length of all paths specified by such patterns is fixed uniquely to n.

For fixed-unique-length patterns, all-shortest-path semantics coincides with unrestricted semantics, since all satisfying paths are shortest. This means that all unrestricted paths are considered, *even if they form a cycle*, because not following the cycle leads to failure to match the pattern In contrast, both non-repeated-vertex and non-repeated-edge semantics rule out unrestricted paths that follow cycles.

Consider the pattern : $x - (\ A > .(B > |D >)._{>}.A >)- : y$. Clearly, it matches only paths of length 4. Apply this pattern against a cycle $v \xrightarrow{A} u \xrightarrow{B} w \xrightarrow{C} v$. All-shortest-paths finds the match $\{x \mapsto v, y \mapsto u\}$ as witnessed by a path starting from v, going around the cycle once and recrossing the A edge. Non-repeated-vertex finds no match, disqualifying this path because it revisits vertex v, while non-repeated-edge also finds no match, disqualifying the path because it revisits the A edge.

Tractability of All-Shortest-Paths Semantics. The remarkable property of all-shortest-paths semantics is that, despite that fact that (just like non-repeating vertex and non-repeating edge semantics) it may yield exponentially many paths in the size of the graph, it is the only one that admits polynomial-time *counting* of these paths by avoiding their enumeration.

Let's call the *Shortest DARPE Match Count (SDMC)* the problem of counting all shortest paths that satisfy a DARPE, where the path length is defined as edge count. Variations include:

- the *single-pair SDMC* flavor, where the source vertex s and target vertex t of the desired shortest paths are given, as well as the DARPE d they must satisfy, and $\text{SDMC}_d(s,t)$ denotes their count;
- the *single-source SDMC* flavor, where only the source vertex s and DARPE d are given and $\text{SDMC}_d(s)$ denotes the set

  $$\text{SDMC}_d(s) = \{(s,t,\text{SDMC}_d(s,t)) \mid t \text{ is a vertex}\};$$

- the *all-paths SDMC* flavor, in which only the DARPE d is given, and SDMC_d denotes

  $$\text{SDMC}_d = \bigcup_{s \text{ is a vertex}} \text{SDMC}_d(s).$$

Notice that the problem of checking the existence of a shortest path from vertex s to vertex t that satisfies DARPE d can be reduced to the test $\text{SDMC}_d(s,t) > 0$, and it thus inherits the complexity upper bound of single-pair SDMC.

The following result is known from semi-structured database research folklore, as corroborated in [4], but we could not find its proof published in the literature. For completeness, we provide the proof in the extended version [10].

Theorem 6.1. *The Single-Pair, Single-Source and All-Paths flavors of the SDMC problem are solvable in polynomial time in the size of the graph.*

Given the incomparability of matching semantics, as exposed by Examples 9 and 10 above, we subscribe to the goal of allowing users to select the desired matching semantics on

a per-query basis, coupled with that of educating users that alternative semantics are potentially intractable and that, for fixed-unique-length patterns all-shortest-paths considers all possible paths, without technically-driven restrictions. Since GSQL is Turing complete, it can implement any desired semantics, but it currently lacks the concise syntactic sugar for specifying semantic alternatives. We plan to support such syntactic sugar by the third quarter of 2020.

7 AVOIDING PATH MATERIALIZATION VIA PATH COUNTING

In the design of GSQL, we opted for the default pattern matching semantics based on the all-shortest-path legality criterion (recall Section 6) because

- it is the only one that amounts to unrestricted semantics for the practically relevant class of fixed-unique-length patterns
- it admits the polynomial-time evaluation of a very large class of queries that involves both DARPE-based patterns and accumulator-based aggregation, even in the case when the query pattern matches exponentially many paths in the size of the graph. This class of queries covers a majority of the use cases we have encountered in customer engagements.

The idea behind the efficient evaluation of queries from this tractable class is to avoid the materialization of the paths matching the query pattern, only counting them instead. The latter can be done in polynomial time by exploiting Theorem 6.1.

A Tractable Class of Queries. The tractable class disallows

- path variables bound to the matches of Kleene-starred DARPEs,
- vertex/edge variables bound in the scope of Kleene star, and
- accumulators of type `ListAccum<T>`, `ArrayAccum<T>` (for any type `T`), as well as `SumAccum<string>`.

Path variables can bind to an entire path, as opposed to individual edges or vertices. An example syntax (inspired by the Cypher, PGQL and Gremlin languages), is $p = :s - (E> \,.\, < D) - :t$, where p is a variable binding to each of the paths that satisfy the pattern on the right hand side of the assignment. Support for path variables was advocated by the G-CORE manifesto [3], it is provided by the above-mentioned languages, and it is adopted in the SQL/PGQ and GQL standard drafts. GSQL does not yet support them, but we plan to do so in early 2020 towards alignment with the standards. So far, we felt little pressure to introduce path variables due to the existence of accumulators in GSQL: they often allow one to avoid the manipulation of paths as first-class citizens and when unavoidable, `ListAccum`, `ArrayAccum` and `SumAccum<string>` accumulators can simulate them.

An example of variables inside the scope of Kleene stars is the pattern $:s - (E>: e) * - :t$, where edge variable e has multiple bindings for any satisfying path p, namely one for each edge of p.

Theorem 7.1. *Under all-shortest-paths semantics, the tractable class admits polynomial time evaluation in the size of the graph.*

Note that the *expression complexity* is exponential (i.e. evaluation may be exponential in the size of the query), but the query size is negligible compared to the graph size and can be treated as a constant. This is no worse than relational query evaluation, from which we inherit this lower bound. The proof is sketched in Appendix A.

7.1 All-Shortest-Path Semantics in Practice

We report on a simple experiment that illustrates the benefit of selecting the all-shortest-paths semantics and accumulator-based aggregation, showing that it is possible to exploit the potential for efficient evaluation given by Theorem 7.1. We also show that this potential is currently not exploited in state of the art.[4]

The experiment confirms that for the tractable class, TigerGraph query execution scales polynomially with the size of the graph even when accounting for exponentially many paths, because instead of materializing them it only counts them, which is a polynomial task according to Theorem 6.1. In contrast, non-repeating semantics (the edge flavor, as represented by the default semantics of the reference system Neo4j) scales exponentially.

Our findings also suggest an explanation for the fact that all-shortest-paths semantics has not yet emerged as the clear preference for a default semantics in either Neo4j's Cypher or in the standardization debate (which uses the experiences of Neo4j's developer community as a significant data point). Indeed in Neo4j's enterprise edition, all-shortest-paths semantics is supported if explicitly demanded by the developer, but with a warning that it might be expensive (no such warning is given for the default semantics). This gives developers

[4]We remark that the goal of this paper is not the comprehensive performance evaluation of TigerGraph, which depends on much more than its aggregation support and pattern semantics. Nor is it the comparison with its peer systems. Our focus is limited to describing aggregation support in TigerGraph and on showing how our design choices are applicable to any pattern-based declarative graph query language. See [21] for a third-party-conducted, comprehensive evaluation of TigerGraph and Neo4j using the Linked Data Benchmark Council's (LDBC) Social Network Benchmark (SNB) [16]. While TigerGraph is shown to generally outperform Neo4j, often significantly, the benchmark results do not isolate the contribution of aggregation performance.

the impression that non-repeated edge semantics is more efficient than all-shortest-paths, an impression strengthened by the actually observed running time performance: in our experiment, Neo4j always runs queries significantly faster under non-repeated edge semantics than under all-shortest-paths semantics.

Data. The experiment was conducted on a Diamond Chain graph like the one described in Example 11 and depicted in Figure 7. We loaded a 30-diamond graph, which is by all standards toy-sized: 91 vertices, 120 edges, with vertices carrying only a 'name' attribute of type string, and edges carrying no attributes. All involved vertices had type V ("label" V in Neo4j terminology) and all involved edges had type E.

Queries. We defined the family of queries $\{Q_n\}_{1 \le n \le 30}$, where Q_n counts the paths from v_0 to v_n. Recall from Example 11 that for this graph and for each query Q_n, the three semantics (non-repeated vertex/edge and all-shortest-paths) coincide and yield 2^n paths, thus facilitating the check for query evaluation correctness.

Varying n from 1 to 30, we measured the running time of Q_n on the diamond graph in Neo4j under the default non-repeated edge semantics, and compared it with the running time in TigerGraph under the all-paths-semantics, to confirm their the different growth rates predicted by their asymptotic complexity. We next ran the queries in Neo4j under all-shortest-path semantics, expecting to see a similar growth rate as in TigerGraph.

The query family is expressible in GSQL as a parameterized query, with the parameters providing the names of the desired source, respectively target vertex.

```
CREATE QUERY Qn(string srcName, string tgtName) {
  SumAccum<int> @pathCount;

  R = SELECT t
      FROM    V:s -(E>*)- V:t
      WHERE s.name = srcName AND t.name = tgtName
      ACCUM t.@pathCount += 1;

  PRINT R[ R.name, R.@pathCount ];
}
```

The Cypher queries are given below, instantiated for Q_n, with Q_n^{nre} denoting the query used to specify Q_n under non-repeated edge semantics, and with Q_n^{asp} the query used to specify Q_n under all-shortest-paths semantics.

Q_n^{nre} :
```
MATCH  (x0{name:'v0'})-[:E*]->(xn{name:'vn'})
RETURN xn.name, sum(1) as pathCount
```

Q_n^{asp} :
```
MATCH  allShortestPaths (
         (x0{name:'v0'})-[:E*]->(xn{name:'vn'})
       )
RETURN xn.name, sum(1) as pathCount
```

n	path count	Q_n^{nre} (ms)	Q_n^{asp} (ms)	n	path count	Q_n^{nre} (ms)	Q_n^{asp} (ms)
1	2	2	2	14	16384	161	206
2	4	2	2	15	32768	340	407
3	8	2	2	16	65536	661	870
4	16	2	3	17	131072	1356	1841
5	32	2	3	18	262144	2743	3911
6	64	2	3	19	524288	5631	8345
7	128	4	4	20	1048576	11618	17475
8	256	4	5	21	2097152	23119	42749
9	512	7	10	22	4194304	50189	–
10	1024	12	14	23	8388608	103970	–
11	2048	22	23	24	16777216	205575	–
12	4096	39	42	25	33554432	417026	–
13	8192	75	83				

Table 1: Neo4j running times in milliseconds

What We Measured. In all cases, we report the warm-cache running times observed after the initial loading of the graph into memory (where it fits easily), without flushing the cache between queries. We set the timeout at 10 minutes.

Results. To study the growth rate with n under non-repeated edge semantics, we ran each Q_n^{nre} in Neo4j. The observed running times start at 2 ms for Q_1, initially remaining in the range of a few milliseconds. Once n exceeds 8, for each increment of n we clearly observe a doubling of the running time, from 4 milliseconds for Q_8^{nre} to 6.95 minutes for Q_{25}. For $n \ge 25$, the queries timed out. This confirms the exponential growth rate of the evaluation algorithm under non-repeated edge semantics. See Table 1, column 3.

We repeated the experiment in TigerGraph, where the queries ran under all-shortest-paths semantics. All queries completed within 10 ms. This is not surprising, as careful analysis reveals that for this graph, our evaluation algorithm runs in linear time in the graph size.

We next returned to Neo4j, running the family Q_n^{asp} of queries under all-shortest-paths semantics. We observed that the Neo4j browser issues a warning that this is an expensive semantics and the recommendation to set a bound on the length of the desired path. Indeed when running the queries, we observed the curve in column 4, which grows significantly faster than for the non-repeating edge semantics, reaching timeout at $n = 22$, with Q_{21}^{asp} running in roughly 42 seconds. See Table 1, column 4.

Platform. The experiment was run on a dedicated MacBook Pro produced in 2018, with 1 Intel Core i5 processor with 4 cores, 16GB or RAM, running MacOS Mojave. We used a Neo4j Desktop 1.2.1 downloaded in October 2019, with Neo4j's Enterprise Edition Server 3.5.6 and Browser version: 3.2.20. We made sure to disable the maximum heap and page buffer limits, allowing Neo4j server to use the entire available RAM. We ran the pre-release version of TigerGraph 3.0, scheduled for release in early Q2 of 2020.

Large-Scale Experiments. We confirmed the observed trend by running experiments over the Social Network Benchmark (SNB) provided by the leading independent graph benchmarking authority, the Linked Data Benchmarking Consortium (LDBC) [16]. SNB provides a graph generator whose schema and statistics reflect real-life social networks. We ran the IC family of queries from the benchmark under both semantics (using TigerGraph for all-shortest-paths and Neo4j for non-repeated-edges). To observe the effect of the path length, we modified those queries that mention cross KNOWS edges between person, from the original 2 to 4 (queries were not modified it they didn't cross this edge type). The queries were run on graphs of scale factor 1 (1GB), 10 (10 GB) and 100 (100GB), on an Amazon EC2 instance of type r4.8xlarge.

TG	size	hops	ic3	ic5	ic6	ic9	ic11
	1	2	0.5s	0.52s	1.39s	1s	0.34s
		3	1.2s	1.03s	1.70s	3.5s	0.47s
		4	0.60s	1.03s	1.77s	3.49s	0.44s
	10	2	2.03s	2.13s	4.13s	4.71s	0.52s
		3	9.15s	7.51s	2m27s	42.63s	0.5s
		4	2.16s	7.6s	2m30s	43.80s	0.62s
	100	2	5.8s	5.15s	4.8s	8.88s	0.9s
		3	1m6s	1m26s	2m13s	4m21s	1.57s
		4	8.96s	2m20s	2m34s	6m55s	1.13s

Neo	size	hops	ic3	ic5	ic6	ic9	ic11
	1	2	1.6s	3.7s	1.9s	6.1s	1s
		3	3s	8.6s	3.8s	15.7s	1.7s
		4	31s	8.5s	43s	15.5s	1.4s
	10	2	8.44s	18.00s	1.9s	38.6s	1s
		3	28.78s	1m16s	3m52s	2m26s	1.8s
		4	5m56s	1m20s	9m22s	2m31s	5.5s
	100	2	4m52s	33.47s	36m57s	1m32s	1.12s
		3	-	12m49s	-	24m49s	5.88s
		4	-	14m23s	-	24m58s	39.73s

We observe that the results of the queries are the same under both semantics for this data set, yet the non-repeated-edge evaluation leads to repeated (60 minute) timeout on the largest graph, and in general to exponential trends on queries ic3 and ic11, which are those affected by the increased number of KNOWS hops.

8 ACCUMULATOR-BASED VS SQL-STYLE AGGREGATION

The main focus of this paper is not on comparing accumulator-based aggregation against relational SQL-style aggregation. Accumulators are meant as a means to achieve powerful and efficient composition results by adorning vertices and graphs with computed state shared between query blocks. However, it turns out that, as a side-effect, accumulators also provide the ability to express flexible aggregation which can be cumbersome and even inefficient to express in conventional SQL style. First, we substantiate our claim that, as far as expressive power is concerned, accumulator-based aggregation subsumes conventional one.

Simulating SQL-style Aggregation. Note that for each built-in aggregation function of SQL, GSQL features an accumulator type. Moreover, SQL group-by aggregation can be expressed using GSQL's `GroupByAccum` accumulator type, which specifies the values of the (possibly composite) group key and a list of nested accumulators, one for each of the desired aggregates.

Example 12 (SQL-Style Aggregation via Accumulators). *Consider a conventional SQL group-by aggregation specified as*

`SELECT` $k_1, k_2, k_3, sum(a_1), min(a_2), avg(a_3)$
...
`GROUP BY` k_1, k_2, k_3

with k_1, a_1, a_2, a_3 of floating point, k_2 of string and k_3 of timestamp type. In GSQL, this is achieved by the ACCUM clause

`ACCUM` $A \mathrel{+}= (k_1, k_2, k_3 \rightarrow a_1, a_2, a_3)$

where A is declared as an accumulator of type

```
GroupByAccum <float, string, ts,
        SumAccum<float>,
        MinAccum<float>, AvgAccum<float> >.
```

Multiple group-by aggregations, such as the CUBE, ROLLUP and GROUPING SET extensions of the SQL GROUP BY clause are also eminently expressible using accumulators. They each compute the aggregation for several subsets of the grouping attributes, outer unioning the results of each grouping. The GROUPING SETS extension is the most flexible one, allowing targeted selection of grouping attribute subsets. For instance, replacing the above GROUP BY clause with

`GROUP BY GROUPING SETS` $((k_1, k_2), (k_3))$

can be simulated in GSQL by

`ACCUM` $A \mathrel{+}= (k_1, k_2, null \rightarrow a_1, a_2, a_3)$,
$A \mathrel{+}= (null, null, k_3 \rightarrow a_1, a_2, a_3)$

Similarly, the CUBE (k_1, k_2, k_3) extension can be simulated with 8 accumulator assignments (one for each subset of $\{k_1, k_2, k_3\}$), and the ROLLUP (k_1, k_2, k_3) extension with 4 accumulator assignments, (one each for $\{k_1, k_2, k_3\}$, $\{k_1, k_2\}$, $\{k_1\}$, $\{\}$). □

Though the current SQL/PGQ and GQL standard drafts, as well as the other reference graph query languages they are informed by support only the equivalent of the unextended GROUP BY clause, Example 12 shows that accumulators facilitate the straightforward addition of the CUBE, ROLLUP and GROUPING SET keywords as syntactic sugar that preserves the intended single-pass execution.

Beyond SQL-style Aggregation. Certain accumulation-based aggregation classes cannot be expressed in SQL style, or, if expressible, doing so is inefficient as it turns single-pass

accumulator-based aggregation into multi-pass SQL-style plans. The inexpressible class includes GSQL queries that do not aggregate groups into scalars, instead storing them in nested collections using `SetAccum`, `BagAccum`, `MapAccum` or `HeapAccum` accumulators, possibly nested at arbitrary depth via `MapAccum` accumulators. The inefficiently expressible class involves single-pass queries that compute the result of grouping aggregation directly into separate tables, as described in Examples 4 and 5. Simulating these in SQL, even using the most flexible grouping control offered by GROUPING SETS, still requires the materialization and multi-pass post-processing of the resulting outer union table to separate it into the target tables.

We highlight another difference which, despite its apparent subtlety, has considerable impact on performance as shown experimentally below. Suppose we wish to aggregate according to several grouping sets, with different aggregations per grouping set, depositing the results in separate tables (accumulators). For SQL-style aggregation, we ignore for now the above-signaled inefficiency of having to post-process the resulting union table to separate results, and we focus on a new one. Notice that CUBE and ROLLUP semantics result in wastefully aggregating for many unwanted grouping sets, (GROUPING SETS avoids this problem). However, all aggregation flavors suffer from another inefficiency, namely having to wastefully compute for each grouping set all aggregates, including the ones destined for another grouping set.

Example 13 (**Wasteful Aggregation per Grouping Set**). *Revisiting Example 12, assume we wish to compute for grouping sets* $(k_1), (k_2), (k3)$ *a sum, min and avg aggregate, respectively. The GROUPING SET semantics would force the computation of all three aggregates (of which two are unwanted) per grouping set. In contrast, in GSQL we can dedicate a separate accumulator* A_i *to each grouping set i, each* A_i *performing only the desired aggregation:* `ACCUM` A_1 += $(k_1 \rightarrow a_1)$,A_2 += $(k_2 \rightarrow a_2)$,A_3 += $(k_3 \rightarrow a_3)$ □

Quantifying Wasteful Aggregation. For an experiment that quantifies the savings of wasteful aggregation due to GSQL-style accumulator-based aggregation, see Appendix B.

9 RELATED WORK

Accumulators. The precursor of GSQL's accumulator abstraction was pioneered in GreenMarl [15], a domain-specific language for programming graph algorithms. GreenMarl is an imperative language for specifying graph navigation via single-edge/breadth-first/depth-first traversal primitives. The nodes visited during the traversal can be adorned with fresh, mutable *node properties* into which values can be accumulated, towards a well-defined result when the same node is visited repeatedly during a traversal. The type of these node properties is restricted to primitive types and collections of vertices and edges. GSQL of course is highly declarative so it does not have any notion of "traversal" or "visited node" as these are inherently imperative. Our design contribution consists in adapting the concept to work with a declarative semantics based on pattern matching. An additional design contribution involves our generalization of the supported accumulator type to a richer type system including, beyond primitive types, all collection types supported by the ODL [6] object-oriented ODMG standard (sets, bags, lists, priority queues, maps of recursively nested accumulators) as well as user-defined accumulators.

Patterns. To date, only TigerGraph's and the upcoming GQL standard's data model support graphs that mix directed and undirected edges (SQL/PGQ and informing products work on directed graphs only). Our product is the only one to adapt the regular expression syntax accordingly, in the shape of DARPEs. When limiting focus to directed graphs, regular expression patterns are the same modulo superficial syntactic differences but deep semantic differences, as explained in Section 6.1. Our engine features a unique polynomial-time implementation of this semantic flavor.

10 CONCLUSIONS

We have described the unique design choices behind aggregation support in TigerGraph's query language GSQL, showing that pattern-based declarativity is compatible with accumulator-based aggregation. Moreover, when combined with all-shortest-paths semantics, it leads to tractability of evaluation of an important class of queries.

While the ideas related to the accumulation paradigm are presented using GSQL as vehicle, they port straightforwardly to any pattern-based query language in circulation. They are particularly relevant to the upcoming graph query language standards SQL/PGQ and GQL.

A PROOF OF THEOREM 7.1

Proof sketch. This result follows from two key observations.

1. Compressed representation of the binding table. While the binding table used to define the semantics of the FROM clause (Section 4.1) may conceptually have exponential size, it can always be represented in compressed form in polynomial space and computed in polynomial time. Indeed, for tractable-class queries, the number of distinct tuples of the binding table is bounded by a polynomial in the size of the graph since variables can bind only to individual vertices or edges (the polynomial's degree is the number of variables, hence the exponential expression complexity). The only source of exponentiality in the size of the graph is the *multiplicity* of a binding tuple β, since the former depends

on the count of distinct choices of shortest paths that witness β. This count, though worst-case exponential in the size of the graph, can be represented in polynomial space and, according to Theorem 6.1, it can be computed in polynomial time. One can therefore compute in polynomial time a compressed representation of the binding table that assigns to each distinct binding tuple its multiplicity.

2. Simulation of duplicate ACCUM clause executions in a single execution. Query semantics dictates that the ACCUM clause be executed once for each row of the binding table. For a binding β occurring with multiplicity μ, that means running μ identical executions of the ACCUM clause, with variables instantiated according to the same β. Since μ is potentially exponential in the size of the graph, we avoid this, running instead a single, modified execution. Instead of duplicating μ times the input operation of a value i into an accumulator A, the modified execution performs a single input, as follows: if A is multiplicity-insensitive (e.g. has type `Min-`, `Max-`, `Set-`, `Or-`, `AndAccum` or `MapAccum` with multiplicity-insensitive nested accumulators), i is inserted just once; if A has type `SumAccum` of numeric type, instead of i the modified execution inserts μi just once. □

B QUANTIFYING WASTEFUL SQL-STYLE AGGREGATION

Over the course of the graph traversal, the overhead wasted in unwanted aggregate computation can add up to measurable performance penalty. We illustrate this by an experiment that compares the performance of two styles of multi-aggregation: accumulator-based, and SQL GROUPING SET (this is the most efficient among the SQL options as it avoids computing undesirable grouping sets – though, like all others, it cannot avoid computing unwanted aggregates per grouping set).

The Data. Towards working with a scalable graph, we adopted LDBC SNB benchmark, using graphs ranging from size 1GB to 1TB.

The Queries. We adapted the SNB-provided queries to compute multiple aggregates. We report here on one such query (its measured behavior is representative for the others). The query navigates from persons to the city they live in and to the comments they liked, as long as published between 2010 and 2012 (persons, cities and comments are modeled as vertices, the relationships between them as edges). It computes three grouping sets, each with its own aggregations:

(i) per (comment publication year), it computes

- the 20 most recent comments, favoring the longest ones in tie breaks,
- the 20 earliest comments, favoring the longest,
- the 20 longest comments, favoring the most recent,
- the 20 shortest comments, favoring the most recent,
- the top 10 comments by oldest authors, favoring the longest,
- the top 10 comments by youngest authors, favoring the longest.

(ii) per (author's city, browser type, publication year, month, comment length), it counts the comments.

(iii) per (author's city, gender, browser type, publication year and month), it averages comment length.

We expressed the query in GSQL, conforming to two styles:

Q_{gs} mimics SQL GROUPING SET aggregation using accumulators as in Example 12. Conforming to GROUPING SET semantics, Q_{gs} computes all 8 aggregates for each of the three grouping sets. The query is published on GitHub.

Q_{acc} computes for each grouping set only the desired aggregates, using appropriate accumulators as shown in Example 13. The query is published on GitHub.

What We Measured. We measured the running times of Q_{gs} and Q_{acc} on graphs generated by SNB's generator, at scale factors SF-1 (1GB), SF-10 (10GB), SF-100 (110GB) and SF-1000 (1TB). For each graph, we ran each query 5 times, computing the median running time.

The Results. We observed the following running times for the queries (all expressed in seconds), showing that, on graph sizes ranging from 1GB to 1TB, accumulators speed up aggregation by a factor of up to 3x when compared to SQL-style aggregation.

scale factor	Q_{gs} median time	Q_{acc} median time	speedup
1	4.841	1.949	2.483
10	59.87	22.146	2.703
100	440.388	167.417	2.630
1000	2972.684	973.538	3.053

The Platform. We loaded the graphs (and ran the queries) at scale factors 1, 10 and 100 on an Amazon EC2 instance of type r4.8xlarge. For scale factor 1000, we used a Microsoft Azure 3-node cluster, with each node of type E64a v4 (both graph storage and query execution were distributed transparently by TigerGraph's engine, in the sense that the user query did not need to be edited).

REFERENCES

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

[2] Amazon. [n. d.]. Amazon QLDB PartiQL. https://docs.aws.amazon.com/qldb/latest/developerguide/ql-reference.html.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter A. Boncz, George H. L. Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan F. Sequeda, Oskar van Rest, and Hannes Voigt. 2018. G-CORE: A Core for Future Graph Query Languages. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*. 1421–1432. https://doi.org/10.1145/3183713.3190654

[4] Renzo Angles, Marcelo Arenas, Pablo Barcelo, Aidan Hogan, Juan Reutter, and Domagoj Vrgoc. 2017. Foundations of Modern Query Languages for Graph Database. *Comput. Surveys* 50, 5 (2017).

[5] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. 1998. What can you do with a Web in your Pocket? *IEEE Data Eng. Bull.* 21, 2 (1998), 37–47. http://sites.computer.org/debull/98june/webbase.ps

[6] R. G.G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez (Eds.). 2000. *The Object Data Management Standard: ODMG 3.0*. Morgan Kaufmann. ISBN 1-55860-647-5.

[7] CouchBase. [n. d.]. https://www.couchbase.com/products/n1ql.

[8] DataStax. [n. d.]. Gremlin. https://www.datastax.com/products/datastax-graph.

[9] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248

[10] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2020. Aggregation Support for Modern Graph Analytics in TigerGraph. *CoRR* (2020).

[11] FaceBook. [n. d.]. GraphQL. https://graphql.org/.

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

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

[14] W3C SparQL Working Group. 2018. SparQL.

[15] Sungpack Hong, Hassan Chafi, Eric Sedlar, and Kunle Olukotun. 2012. 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*. 349–362. https://doi.org/10.1145/2150976.2151013

[16] Linked Data Benchmark Consortium (LDBC). [n. d.]. Social Network Benchmark (SNB). http://ldbcouncil.org/developer/snb.

[17] Leonid Libkin, Wim Martens, and Domagoj Vrgoc. 2016. Querying Graphs with Data. *J. ACM* 63, 2 (2016), 14:1–14:53. https://doi.org/10.1145/2850413

[18] Alberto O. Mendelzon, George A. Mihaila, and Tova Milo. 1996. Querying the World Wide Web. In *PDIS*. 80–91.

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

[20] Kian Win Ong, Yannis Papakonstantinou, and Romain Vernoux. 2014. The SQL++ Semi-structured Data Model and Query Language: A Capabilities Survey of SQL-on-Hadoop, NoSQL and NewSQL Databases. *CoRR* abs/1405.3631 (2014). arXiv:1405.3631 http://arxiv.org/abs/1405.3631

[21] Florin Rusu and Zhiyi Huang. 2019. In-Depth Benchmarking of Graph Database Systems with the Linked Data Benchmark Council's (LDBC) Social Network Benchmark (SNB). UC Merced Technical Report, arXiv:1907.07405v1 [cs.DB]. https://arxiv.org/pdf/1907.07405.pdf.

[22] ISO SC32/WG3. [n. d.]. SQL Property Graph Query Extension, SQL/PGQ. Extension of ISO SQL standard.

[23] ISO SC32/WG3. 2019. Graph Query Language GQL. https://www.gqlstandards.org/.

[24] Amit Singhal. 2001. Modern Information Retrieval: A Brief Overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43. http://sites.computer.org/debull/A01DEC-CD.pdf

[25] Neo Technologies. [n. d.]. The Cypher Graph Query Language. https://neo4j.com/developer/cypher-query-language/.

[26] Neo Technologies. [n. d.]. Neo4j. https://www.neo4j.com/.

[27] TigerGraph. [n. d.]. TigerGraph. https://www.tigergraph.com/.

[28] Apache TinkerPop. 2018. The Gremlin Graph Traversal Machine and Language. https://tinkerpop.apache.org/gremlin.html.

[29] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, Redwood Shores, CA, USA, June 24 - 24, 2016*. 7. https://doi.org/10.1145/2960414.2960421

[30] World Wide Web Consortium (W3C). [n. d.]. XPath Language. https://www.w3.org/TR/xpath-31/.