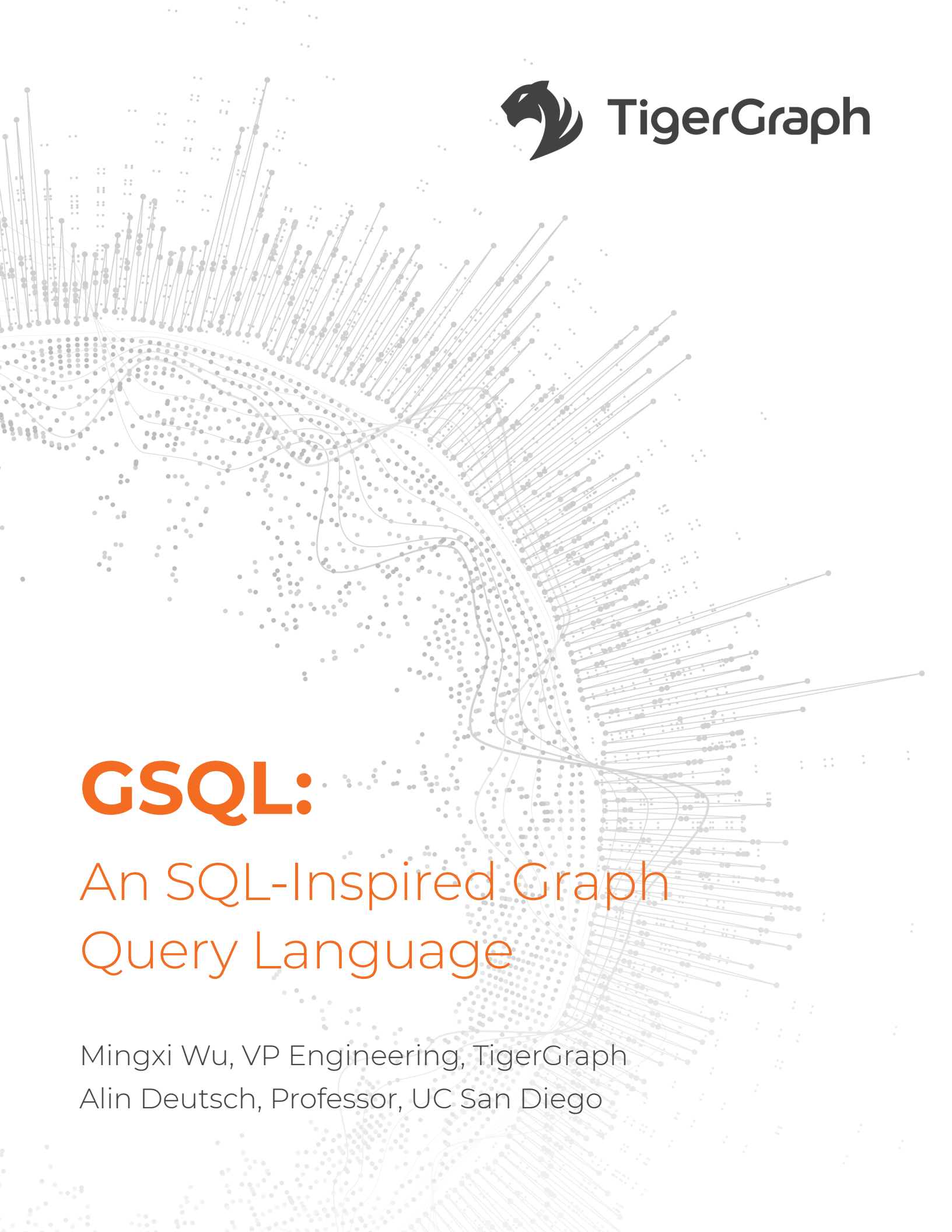

GSQL:

An SQL-Inspired Graph Query Language

Mingxi Wu, VP Engineering, TigerGraph
Alin Deutsch, Professor, UC San Diego

1

INTRODUCTION

We present GSQL, a high-level yet Turing-complete query language for expressing graph analytics. GSQL features an SQL-like syntax that reduces the learning curve for SQL programmers while simultaneously supporting a Map-Reduce interpretation that is preferred by NoSQL developers and that is compatible with scalable, massively parallel evaluation. The GSQL team at TigerGraph has benefited immensely from studying the design principles and implementation of multiple technologies including Gremlin from Apache TinkerPop framework, Hadoop MapReduce, SQL, Cypher and SPARQL. GSQL is the conceptual descendent of these technologies and TigerGraph's team has worked on incorporating the best elements of these predecessors, streamlining the final product for developer productivity and maximum performance for loading, queries and complex graph analytics.

2

GSQL OVERVIEW

GSQL expresses queries that traverse the graph, gather data along the way (as collections or in aggregated form), and store it either centralized or distributed over the visited vertices.

The main building blocks of GSQL queries are the ***vertex*** and ***edge blocks***. These specify computation carried out independently by an ***input vertex set***, its incident edges, and its adjacent vertices and that satisfy a Boolean condition we shall refer to as a ***guarding condition***. The independence of this computation creates high potential for parallel implementation.

To preserve this parallelization potential, the data generated by the vertex/edge computation is stored in ***accumulators***, which are data types able to aggregate a collection of values written to them in parallel. The aggregation is carried out by repeatedly invoking a binary operator += to incorporate the righthand operand (a new value) into the lefthand operand (the accumulator's current value).[1] Accumulators come in two flavors. ***Global accumulators*** gather results in a centralized location, while ***vertex accumulators*** support gathering results distributed across the vertices involved in the block's computation, as well as any other remote vertices whose id is known to the computation.

Blocks specify an ***output vertex set*** which can serve as input vertex set to subsequent blocks. This allows the chaining of GSQL blocks, and more generally the organization of GSQL queries into a directed acyclic graph of blocks.

The expressive power of GSQL is enhanced by standard control flow primitives such as if-then-else style branching and while loops, guarded by conditions that may involve the value of global accumulators. That is, the guard conditions may depend on data computed dynamically during the query evaluation.

A very powerful feature of GSQL is its capability to define ***named parameterized queries***, which can be invoked from other queries by passing arguments for the parameters (recursive invocations are supported). GSQL's query-calling-query capability is analogous to Oracle PL/SQL and Microsoft SQL Server stored procedure capabilities.

The interplay between the versatile accumulator library, the accumulator-based control flow conditions and the query-calling-query capability renders GSQL SQL-complete (defined in Section 4) and indeed Turing-complete.

1 "GSQL comes with a rich set of pre-defined accumulator types, from basic ones corresponding to SQL's aggregation operators (sum, count, min, max, avg), to collection builders for lists, sets, bags, maps, heaps, and arrays.

GSQL BY EXAMPLE

Consider for our running examples a graph database whose vertex types include ***Product*** and ***Customer***. ***Product*** vertices are annotated with attributes for the product name, category, and list price per unit. ***Customer*** vertices carry attributes for the customer SSN, name, and address. In this example, the fact that customer ***c*** bought a product ***p*** is modeled as an edge from ***c*** to ***p***, of type ***Bought***. Edges of type ***Bought*** carry, among others, a price discount attribute and a quantity of product units bought.

Example 3.1 Suppose we are interested in finding, for each customer c, the total revenue earned from sales of products from the "toy" category to c. For this purpose, we need to find all out-edges b of type ***Bought*** whose source is the ***Customer*** vertex c and whose target is a ***Product*** vertex p from the "toy" category. For each such edge b, we determine the sale price by computing the discounted price per product unit ((1- b.discount/100.0)*p.price) multiplying it by the number of product units bought (b.quantity). Notice how this computation can be performed independently (in parallel) for each edge. The revenue is obtained by summing up the sale prices over all b edges, which is achieved by each edge writing to the vertex accumulator located at c. The query prints the name and revenue for each customer.

```
SumAccum<float> @revenue;                                                    (1)
Start = {Customer.*};  (2)
Cust =    select    c                                                        (3)
          from      Start:c -(Bought:b)-> Product:p                          (4)
          where     p.category == "toy"                                      (5)
          accum     c.@revenue += p.price*(1-b.discount/100.0)*b.quantity;   (6)
print Cust.name, Cust.@revenue;                                              (7)
```

The query starts by defining the vertex accumulators that will hold the computed revenue. According to Line (1), each vertex will be equipped with an accumulator named 'revenue' that aggregates floating point values into a sum (implemented as the arithmetic + function).

Lines (3) through (6) define an edge block whose FROM clause (Line 4) specifies the edges that emerge from the Start vertex set (initialized in Line (2) to comprise all ***Customer*** vertices), end in a ***Product*** vertex and are labeled ***Bought***. The FROM clause also introduces variables: ***b*** binds to the edge itself, ***c*** to the source vertex of ***b***, and ***p*** to the target vertex of ***b***.

The WHERE clause (Line 5) specifies a boolean guard condition that determines which edges take part in the block's computation (namely those with a target vertex whose category attribute has the value "toy"). We call these the allowed edges.

The ACCUM clause (Line 6) is executed for each allowed edge. It writes to the revenue accumulator the value of the right-hand-side expression, which computes the revenue from the individual sale corresponding to the edge.

The SELECT clause (Line 3) is also executed for each allowed edge. It specifies the vertex to add to the query's output vertex set. The block in the example returns the set of ***Product*** vertices appearing as targets of the allowed edges. The output vertex set is assigned to variable Cust.

Finally, the PRINT statement prints the name attribute and the contents of the revenue accumulator for each of the vertices in the set denoted by Cust.

Complex Accumulators So far, we have shown accumulators that aggregate numeric data into a single scalar result value. The expressive power of GSQL is significantly enhanced by its support for complex-valued accumulators, which contain for instance collections of tuples, possibly partitioned into groups, and optionally aggregated per-group.

Example 3.2 We refine the revenue aggregation to show, for each customer c, the revenue by product category and the revenue by discount.

```
GroupByAccum<string categ, SumAccum<float> revenue> @byCategory;                    (1)
GroupByAccum<int disc, SumAccum<float> revenue> @byDiscount;                        (2)
Start = {Customer.*};  (3)
Cust =  select   c                                                                  (4)
        from     Start:c -(Bought:b)-> Product:p                                    (5)
        accum    c.@byCategory += (p.category -> p.price*(1-b.discount/100.0)*b.quantity),  (6)
                 c.@byDiscount += (b.discount -> p.price*(1-b.discount/100.0)*b.quantity);   (7)
```

Notice the declaration (in Line 1) of vertex accumulators called 'byCategory'. These are group-by accumulators that define the string attribute 'categ' as the group key and that associate to each group a sum accumulator called 'revenue', which is in charge of aggregating the group's values. Line (6) shows the syntax for writing the value *p.price*(1-b.discount/100.0)*b.quantity* into the 'revenue' sum accumulator corresponding to the group of key *p.category* in the *byCategory* accumulator located at vertex c.

A similar group-by accumulator is defined in Line 2 to group sales by discount value, summing up the sales in each group.

To support the manipulation of accumulator values of collection type, GSQL also includes a *foreach* primitive that iterates over the elements of the collection, binding a variable to them. See the ***tutorial***.

Multi-hop Traversals with Intermediate Result Flow GSQL supports the specification of analytics that involve traversing multi-hop paths in the graph while carrying along intermediate results computed along the way. The following example illustrates how one can concisely express a simple recommender system in GSQL.

Example 3.3 The following GSQL query generates a list of toy recommendations for *customer 1*. The recommendations are ranked in the classical manner of recommender systems: 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* already like in common.

More formally, for each customer c we define a vector $likes_c$ of bits, such that bit i is set if and only if customer c likes toy i. The log-cosine similarity measure between two customers x and y is defined as

$$lc(x, y) = \log(1 + likes_x \odot likes_y)$$

where $\odot$ denotes the dot product of two vectors. Note that $likes_x \odot likes_y$ simply counts the number of toys liked in common by customers x and y.

In the following, we assume that the fact that customer c likes product p is modeled as a unique edge connecting *c* to *p*, labeled *Likes*.

```
SumAccum<float> @rank, @lc;                                          (1)
SumAccum<int> @inCommon;                                             (2)
Start = {Customer.1};      (3)
ToysILike =                                                          (4)
        select      p                                                (5)
        from        Start:c -(Likes)-> Product:p                     (6)
        where       p.category == "toy";                             (7)
OthersWhoLikeThem =                                                  (8)
        select      o                                                (9)
        from        ToysILike:t <-(Likes)- Customer:o                (10)
        where       o.id != 1                                        (11)
        accum       o.@inCommon += 1                                 (12)
        post-accum  o.@lc = log (1 + o.@inCommon);                   (13)
ToysTheyLike =                                                       (14)
        select      t                                                (15)
        from        OthersWhoLikeThem:o -(Likes)-> Product:t         (16)
        where       t.category == "toy"                              (17)
        accum       t.@rank += o.@lc;                                (18)
RecommendedToys = ToysTheyLike MINUS ToysILike;                      (19)
print RecommendedToys.name, RecommendedToys.@rank;                   (20)
```

The query specifies a traversal that starts from customer 1, then in a first block follows ***Likes*** edges to the toys she likes (Line 6), storing them in the ***ToysILike*** vertex set (Line 4). In the second block, the traversal continues from these toys, following inverse ***Likes*** edges to the other customers who also like some of these toys (Line 10). These customers are stored in the vertex set ***OthersWhoLikeThem***. The third block continues from these customers, following ***Likes*** edges to toys they like (Line 16), storing these in the vertex set ***ToysTheyLike***. Notice how blocks are chained together by using the output vertex set of a block as the input vertex set of the successor block.

The recommended toys are obtained as the difference of the ***ToysTheyLike*** and the ***ToysILike*** vertex sets, as we do not wish to recommend toys customer 1 already likes.

Along the traversal, the following aggregations are computed. The second block counts how many toys each other customer o likes in common with customer 1, by writing the value 1 into o's vertex accumulator o.@inCommon for every toy liked by o (Line 12). These unit values are aggregated into a sum by the vertex accumulator, whose final value represents the desired count. In the post-accumulation phase, when the count value is final, Line 13 computes the log-cosine similarity of customer o with customer 1. The third block now writes to each toy t the similarity value o.@lc of every customer o who likes t (Line 18). At the end of the accumulation phase, t.@rank contains the sum of these values, which is precisely the definition of the rank.

Note how intermediate results computed in a block are accessible to subsequent blocks via the accumulators (e.g., customer o's log-cosine similarity to customer 1 is stored by the second block into the vertex accumulator @lc, which is read by the third block to compute the toy's rank).

Control Flow GSQL's global control flow primitives comprise if-then-else style branching (including also the standard ***case*** switches borrowed from SQL), as well as while loops, both guarded by boolean conditions involving global accumulators, global variables, and query parameters. This is particularly useful for expressing iterative computation in which each iteration updates a global intermediate result used by the loop control to decide whether to break out of the loop. A prominent example is the PageRank algorithm variant that iterates until either a maximum iteration number is reached or the maximum error in rank computation over all vertices (aggregated in a global

Max accumulator) is lower than a given threshold (provided for instance as parameter to the query). See the online ***tutorial*** for an implementation of PageRank in GSQL.

Loops can be arbitrarily nested within each other, with the code of the innermost loop being a directed acyclic graph of blocks (the looping of course permits cyclic data flow through the graph).

Query Calling Query GSQL's support for queries calling named, parameterized queries is detailed ***online***.

Updating the Graph GSQL supports vertex-, edge- as well as attribute-level modifications (insertions, deletions and updates), with a syntax inspired by SQL (detailed ***online***).

GSQL Tutorial A comprehensive GSQL tutorial is available ***online***.

Syntactic Sugar Supported in the Future One of the planned extensions to GSQL involves adding the ability to specify several multi-hop paths within the same FROM clause. We note that this is just concise syntactic sugar, since the corresponding traversals can already be expressed via multiple blocks.

Example 3.4 We illustrate by revisiting Example 3.3, whose Lines (4) through (13) would be rewritten more concisely as:

```
ToysILike, OthersWhoLikeThem =
            select      p, o
            from        Start:c -(Likes)-> Product:p <-(Likes)- Customer:o
            where       p.category == "toy" and o != c
            accum       o.@inCommon += 1
            post-accum  o.@lc = log (1 + o.@inCommon);
```

4

COMPARISON TO OTHER LANGUAGES

We compare GSQL to other prominent graph query languages in circulation today. The comparison seeks to transcend the particular syntax or the particular way in which semantics are defined, focusing on expressive power classified along the following key dimensions.

1. **Accumulation** What is the language support for the storage of (collected or aggregated) data computed by the query?
2. **Multi-hop Path Traversal** Does the language support the chaining of multiple traversal steps into paths, with data collected along these steps?
3. **Intermediate Result Flow** Does the language support the flow of intermediate results along the steps of the traversal?
4. **Control Flow** What control flow primitives are supported?
5. **Query-Calling-Query** What is the support for queries invoking other queries?
6. **SQL completeness** Is the language SQL-complete? That is, is it the case that for a graph-based representation G of any relational database D, any SQL query over D can be expressed by a GSQL query over G?
   Turing completeness Is the language Turing-complete?

4.1

Gremlin

Gremlin is a graph query language that specifies graph analytics as pipelines of operations through which objects (called "traversers") flow, evolving and collecting information along the way. Operations can for instance replace (within the traverser) a vertex with its in- or out-neighbor, its in- or out-edge, its attribute, etc., while also extending the traverser with bindings of variables to values computed from the contents of the traverser. The semantics of Gremlin is specified operationally, in functional programming style.

Gremlin is a Turing-complete language and as such as expressive as GSQL. We detail below how it covers the expressivity dimensions we articulated, referring to the ***Apache TinkerPop3 v3.2.7 documentation***.

1. **Accumulation** Gremlin supports the collection of data (into lists or tables), as well as their aggregation using either user-defined aggregation functions or the usual SQL built-in aggregates. Group-by aggregation is supported as in SQL, by defining a relational table, specifying its grouping attributes and aggregated columns.
    - **Gremlin inherits a limitation from SQL: like SQL queries, Gremlin queries can group a table by one group-by attribute list or by another, but not simultaneously by both.** Achieving two different group-by aggregations of the same collected data would require a verbose query that defines the same table of variable bindings twice, grouping each table accordingly. Contrast this with the ease with which GSQL specifies the simultaneous computation of the two groupings of the same data by simply writing each value into two distinct grouping accumulators.
    - Intermediate results computed during the traversal can be stored either in global variables via the "side-effect" operator (akin to GSQL's global accumulators) or into attributes (aka properties) of vertices/edges.
2. **Multi-hop Path Traversal** Gremlin's design was motivated by the desideratum to concisely specify multi- hop linear paths as a pipeline of individual traversal step operations.
3. **Intermediate Result Flow** Intermediate results can be transmitted down the pipeline by storage into the traverser object via a variable binding (in the same spirit in which GSQL passes data via accumulators).
4. **Control Flow** Gremlin features both if-then-else style branching and loops, whose guard conditions do not only specify global intermediate results but also data local to the traversers.
5. **Query-Calling-Query** This is supported as Gremlin allows users to define functions that can invoke each other, including recursively.
6. **SQL and Turing Completeness** Gremlin is Turing-complete and thus SQL-complete.

A Note on Programming Style A characteristic of Gremlin is that the attributes (properties) of vertices and edges are modeled as a *property graph* that needs to be navigated in order to reach the values of properties. To access several attributes, one needs to perform branching navigation in the property graph (from a given vertex or edge, we need to follow one navigation branch to each of its attribute values).

Another characteristic is that its syntax was optimized to concisely express linear (non-branching) path navigation. Branching navigation is achieved in a verbose way, by binding a variable to the vertex at the branching point, following the side branch in another linear navigation, then jumping back to the branching point to resume following the main branch.

The interaction of these two characteristics leads to convoluted navigation and verbose query expressions in queries that need to simultaneously access various attributes of the same edge/vertex. We illustrate by translating Example 3.1 to Gremlin.

```
toys =
      V().hasLabel('Customer').as('c')                              (1)
      .values('name').as('name')                                    (2)
      .outE('Bought').as('b')                                       (3)
      .values('discount').as('disc')                                (4)
      .select('b')                                                  (5)
      .values('quantity').as('quant')                               (6)
      .select('b')                                                  (7)
      .outV()                                                       (8)
      .has('Product', 'category', eq('toy'))                        (9)
      .as('p')                                                      (10)
      .values('price').as('price')                                  (11)
.select('price', 'disc', 'quant')                                   (12)
.map{it.get().price*(1-it.get().disc/100.0)*it.get().quant}         (13)
      .as('sale_price')                                             (14)
.select('name','sale_price')                                        (15)
.group().by('name').by(sum())                                       (16)
```

The above program performs the following steps. (1) Check that the vertex has label 'Customer', if so bind variable 'c' to it. (2) Go to its 'name' attribute, bind variable 'name' to it. (3) Go to outgoing 'Bought' edge, bind variable 'b' to it. (4) Go to value of 'discount' property of 'b', bind variable 'disc'. (5) Go back to edge 'b'. (6) Go to value of b's property 'quantity', bind variable 'quant'. (7) Go back to edge 'b'. (8) Go to target vertex of edge 'b'. (9) Check that it has label 'Product', and a property 'category' whose value is 'toy'. (10) Bind variable 'p' to this 'Product' vertex. (11) Go to the value of the 'price' property, bind variable 'price' to it. (12) Output tuples of variable bindings with components for the variables 'price', 'disc' and 'quant'. (13) For each such tuple, referred to by the built-in variable 'it', compute the price per sale by executing the function given as argument to the map. (14) Bind variable 'sale price' to the value computed at step 13. (15) Output tuples of variable bindings with components for the variables 'name' and 'sales price'. (16) Group these tuples by 'name', summing up the 'sales price' column.

Note that the navigation to the values of the 'price', 'discount', and 'quantity' attributes is branching and involves repeated returns to edge ***b*** (e.g., in Lines (6) and (8)). [2]

Also note that the ability to bind variables is crucial to expressing this query (and all others in our running example) in Gremlin. **Some current systems, such as Amazon's Neptune, only support variable-free Gremlin expressions** (see ***the user guide***), **precluding the specification of this kind of query.**

Finally, note that translating Example 3.2 to Gremlin encounters an additional difficulty: like SQL queries, Gremlin queries can group a table by one group-by attribute list or by another, but not simultaneously by both. Therefore, the grouping by category and the grouping by discount performed in Example 3.2 require a verbose query that defines the same table of variable bindings twice, each followed by its own grouping, and then joins or, more efficiently in terms of result size, unions the two. Contrast this with the ease with which GSQL specifies the simultaneous computation of the two groupings of the same data by simply writing each value into two distinct grouping accumulators (Lines 6 and 7 in Example 3.2).

2 Quoting from **TinkerPop3 documentation** (3/2018): "In previous versions of Gremlin-Groovy, there were numerous syntactic sugars that users could rely on to make their traversals more succinct. Many of these conventions made use of Java reflection and thus, were not performant. In TinkerPop3, these conveniences have been removed in support of the standard Gremlin-Groovy syntax being both in line with Gremlin-Java8 syntax as well as always being the most performant representation. However, for those users that would like to use syntactic sugar at the price of performance loss, there is SugarGremlinPlugin (a.k.a. Gremlin-Groovy-Sugar)." Among the syntactic sugar eliminated in TinkerPop3 is the one allowing the programmer to simply write v.name wherever the value of the name attribute of vertex v is needed, instead of v.values('name').

4.2

Cypher

Cypher is a declarative graph query language based on patterns with variables, such that each pattern match yields a tuple of variable bindings. The collection of these tuples is a relational table that can be manipulated as in SQL.

1. **Accumulation** Cypher's support for accumulation is limited. It supports the collection of data (into lists or tables), and, as in Gremlin, group-by aggregation is computed in SQL style, by defining a relational table and specifying its grouping attributes and aggregated columns. **Intermediate results computed during the traversal can be stored into attributes (aka properties) of vertices/edges, but only as long as they are of scalar or list-of-scalar type, since these are the only attribute types in Cypher.** In contrast, GSQL can store complex-valued data at vertices (e.g., collections of tuples, be they sets, bags, lists, heaps, or maps) using accumulators.

In particular, the query in Example 3.2, which needs to store multiple tuples at the customer vertex (one for each category and one for each discount value) has no exact Cypher analogy. For each of the two groupings, the closest approximation would be a Cypher query that, instead of storing the accumulated complex values at each customer vertex, constructs a table that associates the customer vertex's id with each grouping. This Cypher approximation suffers from another limitation, inherited from SQL: like SQL queries, Cypher queries can group a table by one group-by attribute list or by another, but not simultaneously by both. Therefore, the grouping by category and the grouping by discount require a query that defines the same table of variable bindings twice, each grouped appropriately, and then joins the two:

```
MATCH   (c:Customer) -[b:Bought]-> (p:Product)
WITH    c, p.category as cat,
        sum (p.price*(1-b.discount/100.0)*b.quantity) as rev
WITH    c, collect({category:cat, revenue:rev}) as byCateg
MATCH   (c) -[b:Bought]-> (p:Product)
WITH    c, byCateg, b.discount as disc,
        sum (p.price*(1-b.discount/100.0)*b.quantity) as rev
RETURN c, byCateg, collect({discount:disc, revenue:rev}) as byDisc
```

Contrast this with the ease with which GSQL simultaneously computes the two groupings of the same data by simply writing each value into two distinct grouping accumulators. Note that towards efficient evaluation, Cypher's optimizer would have to identify the fact that the two MATCH clauses can reuse matches, instead of redundantly recomputing them.

2. **Multi-hop Path Traversal Cypher** can specify multi-hop patterns with variables, in the spirit of the upcoming GSQL syntactic sugar illustrated in Example 3.4.
3. **Intermediate Result Flow** Intermediate results can be accessed along the traversal path by referring to the variables bound to them.
4. **Control Flow** Cypher's control flow includes if-then-else style branching. Loop control is limited, supporting only loops over lists of elements computed prior to the iteration's start. This can be used to simulate

loops of given number of iterations, by first creating a corresponding list of integers: *unwind range(1,30) as iter* specifies a 30-iteration loop with iteration variable *iter* and *unwind L as e* specifies an iteration over the elements of the list *L*, binding variable *e* to each. Recalling the PageRank version that iterates until the maximum error falls below a threshold, note that it relies on loops whose guard condition depends on an intermediate result updated during the iteration. This kind of loop is not supported in Cypher and the corresponding class of algorithms is not expressible (e.g., PageRank is provided as built-in primitive implemented in Java).

5. **Query-Calling-Query** Cypher allows users to define functions but these must be implemented in Java, as per the Neo4j ***developer manual***. This design introduces two drawbacks. First, the programmer must be fluent in both Cypher and Java simultaneously. Second, user-defined functions are black boxes to the neo4j optimizer and therefore cannot be optimized in the context of the calling query.
6. **SQL and Turing Completeness** Cypher is SQL-complete but not Turing-complete. The loop control limitations mentioned above are one of the reasons precluding Turing completeness. This statement ignores arbitrary user-defined functions, whose implementation in Java is allowed by Neo4j, and in whose presence the question becomes meaningless since Java is Turing-complete.

5

DISCUSSION

We highlight here one more feature of GSQL that sets it apart from other graph query languages, due to the support for optimization and security: **GSQL's data model definition assumes pre-defined schema/metadata information, which is stored once and for all, factored out of the vertex/edge/attribute representation**.

This leads to increased performance not only due to space savings but also due to the enhanced ability to improve evaluation plans at optimization time. Exploiting pre-defined schemas at optimization time is a tried-and-true database technique that has time and again led to performance advantage.

Additionally, the pre-defined schema is exploited for security and privacy purposes to allow users to query only the graphs (and parts thereof, as defined by graph views) that they have access permissions to. This functionality is critical in real-life applications as it enables multiple tenants (e.g., multiple departments of a company) to use the same graph database with distinct access permissions. Considering all graph database vendors, this is the industry's first and only such support so far.

6

CONCLUSIONS

We have presented the main features of GSQL, a new high-level but Turing-complete graph query language that facilitates parallel evaluation. We have identified key dimensions that classify expressivity and used them to compare with other graph query languages. GSQL matches or surpasses the expressive power of other graph query languages. While the underlying programming paradigm is a matter of taste, GSQL has been designed for compatibility with both SQL and NoSQL's Map-Reduce programming paradigm, thus appealing to both programmer communities.