# GRAPHiQL: A Graph Intuitive Query Language for Relational Databases

Alekh Jindal
CSAIL, MIT

Samuel Madden
CSAIL, MIT

***Abstract*—Graph analytics is becoming increasingly popular, driving many important business applications from social network analysis to machine learning. Since most graph data is collected in a relational database, it seems natural to attempt to perform graph analytics within the relational environment. However, SQL, the query language for relational databases, makes it difficult to express graph analytics operations. This is because SQL requires programmers to think in terms of tables and joins, rather than the more natural representation of graphs as collections of nodes and edges. As a result, even relatively simple graph operations can require very complex SQL queries. In this paper, we present GRAPHiQL, an intuitive query language for graph analytics, which allows developers to reason in terms of nodes and edges. GRAPHiQL provides key graph constructs such as looping, recursion, and neighborhood operations. At runtime, GRAPHiQL compiles graph programs into efficient SQL queries that can run on any relational database. We demonstrate the applicability of GRAPHiQL on several applications and compare the performance of GRAPHiQL queries with those of Apache Giraph (a popular 'vertex centric' graph programming language).**

## I. INTRODUCTION

Graphs are everywhere — from the web to social networks to communication topologies to online games to shopping, logistics, and transportation — graph structured data is a part of our everyday life. These applications all yield massive, evolving graphs that capture user activity, intent, and interactions. Analyzing these massive graph datasets is critical to deriving key insights for businesses and organizations. As a result, a plethora of systems for graph analytics have been proposed in the past few years. These include vertex-centric systems, e.g. Pregel [1], Giraph [2], GraphLab [3] and its extensions [4], [5], [6], GPS [7], Trinity [8], GRACE [9], [10], Pregelix [11]; neighborhood-centric systems, e.g. Giraph++ [12], NScale [13], [14]; datalog-based systems, e.g. Socialite [15], [16], GrDB [17], [18]; SPARQL-based systems, e.g. G-SPARQL [19]; RDF stores, e.g. Jena [20] and AllegroGraph [21]; key-value stores, e.g. Neo4j [22], HypergraphDB [23]; and others such as TAO [24] and FlockDB [25].

These purpose-built graphs systems are typically used in conjunction with a storage system such as a relational database. A common usage pattern involves collecting raw data, e.g., about sales transactions, friend relationships, locations visited, etc in a relational table, exporting this relational data into a graph system, running some analysis, and then (possibly) reloading the data into the database for presentation or aggregation. Two systems are typically used because of the perception that 1) it is difficult or impossible to express many graph analytics in SQL, 2) even if it is possible to express graph analytics in SQL, relational databases are inefficient at the kind of iterative algorithms that comprise many graph analytics (e.g., page rank, shortest paths), and 3) graphs systems lack features that make them usable as primary data stores, such as the ability to efficiently perform in-place updates, provide transactions, or efficiently subset or aggregate records.

As a result, most users of graph systems run two different platforms. This is cumbersome because users must put in significant effort to learn, develop, and maintain two separate environments. Thus, a natural question to ask is how bad it would be to simply use a SQL-based relational system for both storing raw data and performing graph analytics. Related work suggests there may be some hope: for instance, researchers have demonstrated very good performance of triangle counting on Vertica [26], [27], comparable or better performance than specialized graph systems for finding the shortest paths on Oracle [28], and query optimization techniques for efficient subgraph pattern matching in PostgreSQL [29]. Others have looked at the utility of combining graph analysis with relational operators [30], [31]. However, none of these efforts tried to build a truly usable, general purpose graph processing layer on top of relational systems.

In this work we describe a new, general purpose graph processing language we have developed that can be readily compiled into efficient SQL. We chose to develop a new language because it is awkward to express graph analytics in SQL, as many operations involve multiple self-joins on tables of nodes and edges. Such queries are verbose and hard to read, and require programmers to think in terms of tables and joins, rather than the more natural representation of nodes and edges. Complex SQL queries can result even for relatively simple graph operations, e.g. multiple joins for simple neighborhood accesses. The user also needs to tune the relational engine, e.g. create physical designs such as sort orders and indexes, in order to extract good performance.

To illustrate, we consider the problem of implementing single source shortest paths in SQL. Assume that we have a `node` table storing node ids and node values, and an `edge` table storing source and destination node ids for each edge. We could compute the path distance at a node by looking at the distance stored in the neighboring nodes, and updating the distance whenever we see a smaller distance, i.e.

```
UPDATE node SET node.value=node_update.value
 FROM(
  SELECT e.to_node AS id, min(n1.value+1) AS value
      FROM node AS n1, edge AS e, node AS n2
      WHERE n1.id=e.from_node AND n2.id=e.to_node
      GROUP BY e.to_node, n2.value
      HAVING min(n1.value+1) < n2.value
 ) AS node_update
 WHERE node.id=node_update.id;
```

Clearly, coming up with the above query expression requires expert knowledge and experience of SQL, moving the programmer far from the actual graph analysis task.

The above example shows that ease-of-use is a crucial requirement for graph analytics. In fact, this is one of the reason for the popularity of vertex-centric graph processing models and the wide range of graph processing systems based on it, including Pregel [1], Giraph [2], GraphLab [3], GPS [7], Trinity [8], GRACE [9], [10], and Pregelix [11]. Using a vertex-centric programming model, the developer only deals with the computation logic at each vertex of the graph, without worrying about other messy details. However, procedural vertex-centric programs require programmers to write custom code, rather than declarative expressions as in SQL. Furthermore, the vertex-centric model is not particularly flexible and several seemingly simple graph operations are very difficult to express, e.g. triangle counting and 1-hop analysis (see Section II-A).

In this paper, we present GRAPHiQL (pronounced *graphical*), an intuitive *declarative* graph query language and compiler that compiles into efficient SQL programs that can run on any relational database. GRAPHiQL is inspired by PigLatin [32] in the sense that it aims to *fit in a sweet spot between the declarative style (SQL) and low-level procedural style (vertex-centric) approaches*. GRAPHiQL provides first class support for graphs, allowing programmers to reason in terms of nodes and edges. Furthermore, GRAPHiQL offers key graph constructs such as looping, recursion, and neighborhood access, thereby making it a more natural query language for expressing graph analytics. As a result, users can focus on their actual analysis task instead of struggling with crafting SQL expressions. The GRAPHiQL compiler also applies query optimization techniques to tune the performance, and runs them on a standard relational engine. Thus, GRAPHiQL combines efficient and easy to program graph analytics within relational engines, which make it possible to perform transactional operations, execute fine grained updates, and run a variety of declarative queries to filter, aggregate and join graph data.

**Contributions:** In summary, our major contributions are:

**(1.)** We contrast two major query languages for graph analytics, a procedural query language and a structured query language. We discuss the pros and cons of these two approaches and motivate the need for a new language. (Section II).

**(2.)** We present GRAPHiQL, an intuitive graph query language for relational databases. We introduce *Graph Tables*, a data abstraction for manipulating graphs. We describe how users can create as well as manipulate Graph Tables. (Section III)

**(3.)** We discuss several case studies, including typical applications such as PageRank and shortest path, as well as more advanced node-centric analysis such as triangle counting, discovering strong ties and finding weak overlaps. (Section IV)

**(4.)** We describe the GRAPHiQL compiler, which first parses a GRAPHiQL query into an operator graph, and then compiles Graph Tables to relational tables and Graph Table manipulations to relational operators. GRAPHiQL applies several standard optimization techniques to the resulting relational algebra expression. (Section V)

**(5.)** We show the performance of GRAPHiQL queries and compare it with Apache Giraph, a popular system for large scale graph analytics. Our result shows that GRAPHiQL queries can match or even outperform specialized graph processing systems. (Section VI).

## II. LANGUAGES FOR GRAPH ANALYSIS

In this section, we motivate the need for a new graph query language. We start by looking at a leading graph language, Pregel, and discuss its strength and weaknesses for graph analytics. We then contrast Pregel with SQL, a very different query language from Pregel, for graph analytics. Our goal is to show the problems with the leading graph languages and describe how GRAPHiQL addresses them.

### *A. Pregel: Procedural Graph Language*

Like MapReduce, procedural graph languages require programmers to implement one or more procedures that perform the core of the graph operations, while the graph system takes care of executing the program (often in a distributed manner). The canonical procedural languages are so called "vertex-centric languages", such as Pregel [1]. These requires programmers to write a vertex compute function, which is executed once for each vertex of the graph (possibly over several iterations or rounds). Listing 1 shows the vertex-centric implementation of PageRank in Pregel [1]. The vertex compute function takes a list of messages as input and sends one or more messages to its neighbors.

```
void compute(vector<float> messages){
  // compute the PageRank
  float value;
  if( iteration >= 1){
    float sum = 0;
    for( vector<float>:: iterator it=messages.begin(); it!=messages.end(); ++it)
      sum += *it;
    value = 0.15/NUM_NODES + 0.85*sum;
  }
  else
    value = 1.0/NUM_NODES;
  modifyVertexValue(value);
  // send messages to all edges if there are still more iterations
  if( iteration < NUM_ITERATIONS){
    vector<int> edges = getOutEdges();
    for( vector<int>:: iterator it = edges.begin(); it != edges.end(); ++it)
      sendMessage(*it, value/edges.size());
  }
  voteToHalt();   // halt
}
```

Listing 1. PageRank in Pregel.

For many programs, such as PageRank, the vertex-centric abstraction is quite intuitive for programmers. However, vertex-centric programs requires expert programmers whereas analysts are typically more comfortable with SQL-style declarative languages. Furthermore, vertex-centric programming model is not ideal for many analyses. Consider triangle counting (computing the number of "triangles" in a graph, where a triangle is any set of three nodes, $A$, $B$, and $C$, such that there is an edge from $A \leftarrow B$, from $B \leftarrow C$, and from $C \leftarrow A$). To implement this as vertex program requires every vertex to have access to its neighbors' neighbors. To do this, we might use the first superstep to ship the neighborhood information and the second superstep to actually count the triangles, as shown in Listing 2. This leads to significant communication overhead and poor performance as large amounts of neighborhood information are sent throughout the network.

```
void compute(vector<float> messages){
  if( iteration ==0){  // send neighborhood as messages
    vector<int> edges = getOutEdges();
    for( vector<int>:: iterator  it1 =edges.begin(); it1 != edges.end(); ++it1)
      for( vector<int>:: iterator  it2=edges.begin(); it2 != edges.end(); ++it2)
        sendMessage(*it1, *it2);
  }
  else{
    multiset<float> ids;
    for( vector<float>:: iterator  it=messages.begin(); it != messages.end(); ++it)
      ids. insert (* it );
    vint  triangleCount = 0;
    vector<int> edges = getOutEdges();
    for( vector<int>:: iterator  it=edges.begin(); it != edges.end(); ++it)
      triangleCount += ids.count(* it );
    modifyVertexValue( triangleCount );
  }
  voteToHalt();
}
```

Listing 2. Per-node Triangle Counting in Pregel.

Figure 1(a) and 1(b) show the performance of Giraph (open source implementation of Pregel) and Vertica (SQL) on PageRank and triangle counting respectively. We ran these experiments over varying graph sizes (shown on the x-axis) on a single node. Giraph runs PageRank over all graph sizes, however its performance is much worse than Vertica, by a factor ranging from 14 to 6.5. Hence, although these vertex centric languages are intuitive for some programs, they don't perform well. Additionally, Giraph can run triangle counting only for smaller graphs and runs out of memory for larger graphs, even with 12GB of RAM for each of the four workers.

## B. *SQL: Structured Query Language*

SQL is a well known language used for a variety of data analysis (including graph analytics). As we saw in Figure 1(a), SQL-based systems can have very good performance for graph queries. This is because programmers can leverage heavily optimized built-in operators, compared to custom vertex functions in Pregel. Furthermore, programmers do not have to provide elaborate custom code, but can simply compose their queries from the rich vocabulary of SQL operators. However, SQL is awkward for some graph analysts and it is difficult for analysts to craft the right SQL queries for their analysis. For example, Listing 3 shows how one could implement PageRank using SQL [33], over two tables: node(id,value) and edge(from_node,to_node).

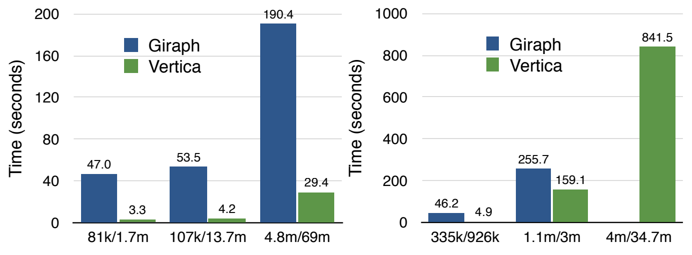
```
CREATE TABLE node_outbound AS
  SELECT id, value/COUNT(to_node) AS new_value
    FROM edge e, node n
    WHERE e.from_node=n.id
    GROUP BY id, value;
CREATE TABLE node_prime AS
  SELECT to_node AS id, 0.15/N + 0.85*SUM(new_value) AS value
    FROM edge e, node_outbound n
    WHERE e.from_node=n.id
    GROUP BY to_node;
```

Listing 3. PageRank in SQL.

```
SELECT e1.from_node,count(*)/2 AS triangles
 FROM edge e1
  JOIN edge e2 ON e1.to_node=e2.from_node
  JOIN edge e3 ON e2.to_node=e3.from_node
    AND e3.to_node=e1.from_node
 GROUP BY e1.from_node
```

Listing 4. Per-node Triangle Counting in SQL.

While the above PageRank implementation has good performance, it is not particularly intuitive as it requires reasoning



(a) PageRank (b) Triangle Counting

Fig. 1. Pregel and SQL languages over varying graph sizes (nodes/edges).

in terms of joins of tables. Likewise, Listing 4 shows triangle counting using SQL.

Again, we see that it is not trivial to come up with the above SQL statement, as users are forced to think in terms of tables and joins instead of graph elements such as nodes and edges. Furthermore, the analyst needs to be aware of system specific optimizations and craft the query for best performance, e.g. in the PageRank example, since the algorithm updates all nodes, in most SQL databases it is better to create a new table compared to updating an existing table in-place.

To bridge the gap between these two approach, in this paper we propose a new graph query language that has powerful syntax and expressibility (as in SQL) while allowing programmers to think in terms of graphs (as in vertex-centric languages). Furthermore, our language supports both SQL-style aggregates as well as the Pregel-style vertex-programs, thereby combining the best of both worlds. And finally, our language runs all graph analysis on the relational database engine itself, harnessing the efficiency of modern analytic database engines. Thus, our goal is to strike the right balance between usability, expressibility, and performance. We describe our language in detail below.

## III. GRAPHIQL

In this section, we introduce GRAPHiQL, our new query language for graph analytics. The core of GRAPHiQL is a new data abstraction and a number of operators supported by it. We describe these below.

## A. *Graph Tables*

Typically, one would store a graph in a relational database as a table of nodes and a table of edges. However, with this data model, the programmers are left to deal with complicated (self) joins when performing data analysis. Such joins are difficult for many programmers to think about, and can introduce performance bottlenecks that require careful tuning of SQL or physical database structures (e.g., indexes and materialized views). As an alternative, we believe it is more natural for programmers to think of graphs in terms of a recursive data structure. This recursive nature helps users to easily access neighborhoods of nodes and edges, a key operation in almost all graph analyses. Finally, the notion of neighborhood is equally important for nodes as well as edges, i.e. an analyst should be able to access the nodes or edges that are adjacent to a given node or edge. To address these, we propose a new data abstraction, called *Graph Tables*, to allow programmers to easily express their graph analytics. A Graph Table has the following three key features:

*1) **Collection of Graph Elements:*** A *Graph Table* is a set of *graph elements*, which together describe a graph. A graph element can be either a node or an edge. Each graph element has a unique `id` as well as a `type` indicating whether it is a node or an edge.

*2) **Graph Element Properties:*** A graph element can have any number of *properties* associated to it, e.g. the *name* of the graph element, or its *weight* denoting its importance. A property is a key/value pair which describes the graph element. `Id` and `type` are default properties of every element.

*3) **Recursive Association:*** Each graph element (or sets of elements) has access to its *outgoing* and *incoming* graph elements, e.g. outgoing edges. Graph elements have this support for neighborhood access via two recursive associations:

- `out` — points to outgoing graph elements
- `in` — points to incoming graph elements

For undirected graphs, `out` and `in` are identical sets of graph elements. This neighborhood access allows programmers to reason about the graphs more naturally.

Thus, Graph Table provides first class support for graphs and is more intuitive to reason about. By utilizing a combined data structure for both nodes and edges, GRAPHiQL takes care of combining their physical tables. Essentially, Graph Tables act as a logical abstraction over the physical representation of the graph. As we describe in Section V, Graph Tables can be easily compiled into relational tables. In this paper, we stick to node lists and edge lists as the underlying physical representation. However, one could imagine other physical representations as well.

## *B. Graph Table Definition*

In the following, let us first look at the data definition language (DDL) for Graph Tables.

*1) **Create:*** We introduce a `GRAPHTABLE` keyword to create a Graph Table.

```
CREATE GRAPHTABLE g AS
  NODE (p1,p2,..)
  EDGE (q1,q2,..)
```

This means that we create a Graph Table as a set of nodes and edges, with properties `p1,p2,..` and `q1,q2,..` respectively. Node and edge properties can have different or same name, i.e. `p1,p2,..` and `q1,q2,..` are not disjoint. Also, note that in this paper, we restrict ourselves to traditional graph elements, i.e. nodes and edges. However, we could also extend Graph Table to other types of graph elements, e.g. neighborhoods. Exploring this in more detail would be a part of future work.

*2) **Load:*** Once created, we can load a Graph Table with graph elements as follows:

```
LOAD g AS
  NODE FROM graph_nodes DELIMITER d
  EDGE FROM graph_edges DELIMITER d
```

Keywords `NODE` and `EDGE` identify the type of graph element and files *graph_nodes* and *graph_edges* contain the data for nodes and edges respectively. Different graph element types can have different key value pairs. GRAPHiQL automatically adds an *id* to each graph element, in case it is not provided.

*3) **Drop:*** We can drop a Graph Table as:

```
DROP GRAPHTABLE g
```

## *C. Graph Table Manipulation*

Let us now look at the Graph Table manipulation operators provided by GRAPHiQL. Overall, GRAPHiQL provides five Graph Table operators which are powerful enough to express a large variety of graph analytics. These are iterate, filter, retrieve, update, and aggregate. We discuss these below.

*1) **Iterate:*** We can access individual graph elements of a Graph Table using `FOREACH..IN` statement as follows:

```
FOREACH item IN graph_element_set
```

Here the set of graph elements can be the original Graph Table or a Graph Table referred to by recursive association from the starting element (i.e. *out*, *in*). Additionally, we can execute a fixed number of iterations using `FOREACH`, e.g.:

```
FOREACH i IN [1,5]
```

We can also iterate conditionally using `WHILE` statement:

```
WHILE condition
```

*2) **Filter:*** We can filter a Graph Table on one or more key/value pairs to get a subset of the Graph Table, i.e. given a Graph Table $g$, its subset $g'$ filtered on key/value pairs $k_1/v_1....k_n/v_n$ is given as:

```
g' = g(k1=v1,k2=v2,...,kn=vn)
```

In addition, we can filter the graph to retrieve just nodes or edges, and to select the neighborhood of a node. For example, the set of nodes (i.e. graph elements of type node) in $g$ is given as $g$(type=N) and the set of edges in $g$ is given as $g$(type=E). While accessing individual graph elements (using `FOREACH` statement), we can access the set of outgoing edges of each graph element $e$ as $e.out$(type=E). Likewise, the set of outgoing nodes of $e$ is given as $e.out$(type=N), and so on.

We see that the GRAPHiQL syntax for filter operation is very compact, thus allowing an analyst to quickly narrow down his analysis to only the interesting portions of the Graph Table.

*3) **Retrieve:*** We can conditionally retrieve graph elements or algebraic expressions over elements as follows:

```
GET expression1,expression2,...,expressionk
WHERE condition
```

Here, *expression* can be any algebraic expression over a set of graph elements or each graph element in a `FOREACH` construct. For example, we can retrieve the *value* of all nodes as follows:

```
GET g(type=N).value
```

Similarly, we can retrieve nodes with value greater than 10 as:

```
FOREACH n in g(type=N)
  GET n
  WHERE n.value>10
```

*4) **Update:*** GRAPHiQL also allows updating of graph elements subject to a condition, as follows:

```
SET variable TO expression
WHERE condition
```

We can update all elements in a Graph Table, e.g. set the weight of all edges to 1, as follows:

```
SET g(type=E).weight TO 1
```

We can also do the above update using a `FOREACH` statement:

```
FOREACH e IN g(type=E)
  SET e.weight TO 1
```

The update statement returns the number of graph elements successfully updated.

*5) **Aggregate:*** Finally, GRAPHiQL allows programmers to aggregate properties from a set of graph elements. GRAPHiQL has several aggregate functions including *SUM*, *COUNT*, *MIN*, *MAX*, *AVG*, by default. For example, the number of outgoing edges of each node in Graph Table $g$ is given as:

```
FOREACH n IN g(type=N)
  GET n.id, COUNT(n.out(type=E))
```

## D. Nested Manipulations

Graph Table manipulations can be nested, i.e. the analysis over the recursive Graph Tables can be nested within each other. Updates, however, cannot be nested, i.e. we cannot apply another Graph Table manipulation after applying an update. Table I shows all valid nesting for Graph Table manipulations.

| outer \ inner | Iterate | Aggregate | Retrieve | Update |
|---|---|---|---|---|
| Iterate | ✓ | ✓ | ✓ | ✓ |
| Aggregate | ✓ | ✓ | ✓ | |
| Retrieve | ✓ | ✓ | | |
| Update | ✓ | ✓ | | |

TABLE I. VALID NESTING FOR GRAPH TABLE MANIPULATIONS.

As an example, we can retrieve all pairs of nodes as follows:

```
FOREACH n1 IN g(type=N)
  FOREACH n2 IN g(type=N)
    GET n1.id, n2.id
```

Likewise, we can compute the number of 1-hop neighbors (neighbors of neighbors) of each node in a Graph Table $g$ (assuming undirected graph) as follows:

```
FOREACH n IN g(type=N)
  GET n.id, SUM(
                FOREACH n' IN n.out(type=N)
                  COUNT(n'.out(type=N))
               )
```

By nesting the Graph Table manipulations, programmers can easily compose arbitrarily complex graph analysis.

## E. User Defined Functions

GRAPHiQL supports user defined functions (UDFs) to allow programmers to inject custom functionality. These UDFs are stateless functions which take in a set of parameters, perform some computation, and output the result. For example, we can define a UDF with some computation for every vertex of the graph. Likewise, we can also define a UDF having computation for every edge of the graph. Programmers can create a UDF by defining its signature. For instance, a vertex UDF could be created as follows:

```
CREATE UDF vertex(int id, int value, g edges)
RETURNS int
```

Thereafter, we can run the UDF *vertex* over a Graph Table $g$ in a Pregel-style manner as follows:

```
WHILE c>0
 FOREACH n IN g(type=N)
  c = SET n.value TO vertex(n.id,n.value,n.out)
```

UDFs are powerful tools to extend the functionality of GRAPHiQL.

## IV. APPLICATIONS

In this section, we present several case studies to demonstrate the applicability of GRAPHiQL. We consider both typically graph analysis, such as PageRank and shortest path, as well as more complex analysis, such as triangle counting, finding strongly overlapping vertices, and detecting weak ties.

## A. PageRank

We first consider PageRank. The PageRank of a vertex is the aggregate of the PageRanks contributed by its incoming neighbors, i.e. the PageRank of each vertex $n$ in a Graph Table $g$ is given as:

```
FOREACH n IN g(type=N)
  SET n.value TO 0.15/N+0.85*SUM(pr_neighbors)
```

Here, `pr_neighbors` are the PageRanks contributed by the incoming neighbors. In the following we show how to compute `pr_neighbors` as a GRAPHiQL expression. Each vertex distributes its PageRank equally among its outgoing neighbors, i.e. the PageRank contributed by a vertex $n'$ to each neighbor is given as:

```
n'.value/COUNT(n'.out(type=N))
```

Therefore, the total PageRank of a node (with `pr_neighbors` expanded out) can be written as:

```
FOREACH n IN g(type=N)
  SET n.value TO 0.15/N+0.85*SUM (
    FOREACH n' IN n.in(type=N)
      GET n'.value/COUNT(n'.out(type=N))
  )
```

We can run multiple iterations of PageRank by nesting the above query in another loop. For example, we can run 10 iterations of PageRank as follows:

```
FOREACH i IN [1,10]
  FOREACH n IN g(type=N)
    SET n.value TO 0.15/N+0.85*SUM (
      FOREACH n' IN n.in(type=N)
        GET n'.value/COUNT(n'.out(type=N))
    )
```

The three key features that makes writing the above expression extremely easy and intuitive are: (i) allowing users to easily nest multiple sub queries, (ii) quick neighborhood access via recursive association of graph elements, and (iii) the ability to perform ad-hoc filtering of graph elements.

### B. Shortest Path

Single Source Shortest Path (SSSP) is an important operation in several applications, such as social networks and transportation networks. To compute the shortest distance to a vertex, we look at the distances to each of its neighbors, find the minimum, and increment that by 1. We update the vertex distance if the newly found distance is smaller. This is expressed in GRAPHiQL as follows:

```
FOREACH n IN g(type=N)
 SET n.value TO MIN(n.in(type=N).value+1) As v'
 WHERE v'<n.value
```

Of course, we need to initialize the distance of source vertex to 0 and all other vertices to infinity. And we continue updating the distances as long as any vertex finds a smaller distance, i.e. as long as we have any updates to the Graph Table. The entire SSSP query in GRAPHiQL looks as follows:

```
SET g(type=N).value TO INF
SET g(type=N,id=startNode).value TO 0
WHILE updates>0
  FOREACH n IN g(type=N)
    updates = SET n.value TO
                 MIN(n.in(type=N).value+1) As v'
                 WHERE v'<n.value
```

Thus, we see that the above path-style BFS query is easy to express in GRAPHiQL.

### C. Triangle Counting

We now look at the problem of counting the number of triangles in graph. Triangle counting is an important step in algorithms such as computing clustering coefficients. Using GRAPHiQL, the total number of triangles in a Graph Table $g$ can be counted as:

```
COUNT(
 FOREACH e1 IN g(type=E)
  GET e1.out(type=E).out(type=E,to_id=e1.from_id)
)
```

Essentially, in the above query, we are looking for patterns such that two successive outgoing edges of an edge (i.e. `out(type=E).out(type=E)`) form a triangle. Note that in this query, we deal only with the edges because that is more natural when thinking of triangles. This is in contrast to vertex-centric languages where users are constrained to think in terms of vertices.

Instead, of getting the global count, we can also count the triangles for each node in the graph (useful for finding local clustering coefficient). This could be done by iterating over all vertices in the graph and looking for three successive edges that form a triangle, as follows:

```
FOREACH n IN g(type=N)
 GET n.id, COUNT(
               n.out(type=E)
                .out(type=E)
                .out(type=E,to_id=n.id)
              )
```

Thus, we see that in addition to path-style queries, GRAPHiQL can also be easily used for pattern-style queries to quickly navigate and find desired portions of the graph.

### D. Strong Overlap

Let us now look at a slightly more complex graph analysis, namely finding all nodes which have strong overlap between them. Here we define overlap as the number of common neighbors between two nodes. Other measures are of course possible as well. Such a query can be very tricky to express in other query languages, particularly in SQL. However, in GRAPHiQL, we can make use of two nested loops over the nodes in the graph and count the overlap between them. Given two nodes, $n1$ and $n2$, an overlap between them occurs when $n2$ is the 2-hop outgoing neighborhood of $n1$, i.e.:

```
n1.out(type=N).out(type=N,id=n2.id) = n2
```

Thereafter, we simply count these occurrences for every pair of nodes and filter those which are above a threshold:

```
FOREACH n1 in g(type=N)
  FOREACH n2 in g(type=N)
    GET n1.id,n2.id,COUNT(
      n1.out(type=N).out(type=N,id=n2.id)
    ) AS overlap
    WHERE overlap > Threshold
```

Notice that in the above example, we have two nested iterators over unrelated (i.e. not recursively associated) sets of graph elements. Thus, the iterator nesting in GRAPHiQL is not just limited to neighborhood access. Rather, we can nest any two sets of graph elements.

### E. Weak Ties

Finally, let us consider an even more complex graph analysis of finding all nodes which frequently act as ties between two otherwise disconnected pair of nodes. This is more complicated because it involves finding the absence of edge between two vertices. Using GRAPHiQL, we can detect whether two nodes $n1$ and $n2$ have an edge between them by checking whether `COUNT(n1.out(type=N,id=n2))` is 0 or not. Thus, we can get the neighboring node-pairs of $n$ which are pair-wise disconnected as follows:

```
FOREACH n1 IN n.out(type=N)
  FOREACH n2 IN n.out(type=N)
    GET COUNT(n1.out(type=N,id=n2.id))==0 ? 1:0
```

The '`?`' operator above is similar to `CASE` expression in SQL, i.e. it returns the first or the second value depending on the condition before '`?`'. We find all nodes $n$ which have a sufficient number (greater than threshold) of mutually disconnected neighbor-pairs, i.e. they act as ties between them, as follows:

```
FOREACH n IN g(type=N)
 GET n.id, SUM(
  FOREACH n1 IN n.out(type=N)
   FOREACH n2 IN n.out(type=N)
    GET COUNT(n1.out(type=N,id=n2.id))==0 ? 1:0
 ) AS ties
 WHERE ties > Threshold
```

Thus, we see that even though GRAPHiQL has a restricted vocabulary, it is powerful enough to express fairly complex graph analysis. More importantly, it brings the user focus back to their analysis instead of struggling with the query language.

## V. Compilation

### A. Query Parser

The GRAPHiQL query parser parses GRAPHiQL query statements into a parse tree of Graph Table operators, consisting of Filter, Iterate, Retrieve, Update, and Aggregate operators. The parser is responsible for checking the query syntax against the syntax of the five Graph Table manipulations, as defined in Section III-C. The parser also checks query nesting in two respects:

- the symbols in a nested statement must be *defined*, either in the same statement or any of the outer statements.
- the nesting must be *valid*; Table I summarizes the valid nestings in GRAPHiQL.

For user-defined functions, GRAPHiQL checks that the function has been defined and its input/output matches with its definition. After parsing and checking the query statements, GRAPHiQL produces a logical query representation. This is then used to build the relational plan, as described below.

### B. Plan Builder

GRAPHiQL compiles the user queries into a relational operator tree. In the following, we describe how GRAPHiQL compiles Graph Tables and its manipulations to relational tables and relational operators over them.

*1)* ***Graph Table Manipulations to Relational Operators:*** Let us first look at how GRAPHiQL compiles the Graph Table manipulations to relational operators (still over Graph Tables for the moment).

*Filter.* Filtering operations over a Graph Table are simply mapped to selection predicates ($\sigma$). For example, filtering over keys $k_1, k_2, ..$ is mapped as follows:

$$g(\texttt{type=N}, k_1 = v_1, k_2 = v_2, ..) \mapsto \sigma_{k_1=v_1,k_2=v_2,..}(\texttt{g(type=N)})$$

The only exception here is the filter by *type*, which is used when compiling Graph Tables to relational tables.

*Scalar Iterators.* Iterators over scalar values, e.g. `FOREACH i IN [1,10]`. indicate a set of iterations for the analysis. We map these to an outside driver loop to run the actual SQL. The compilation of iterators over graph elements, in contrast, depend on the type of manipulation they contain, i.e. retrieve, update, or aggregate. We describe these below.

*Retrieve.* Retrieve operations are essentailly projections ($\pi$) over the Graph Table. Retrieve operations with or without an iterator are compiled identically (the iterator only improves readability.) For example, consider the following retrieve operation which returns the values of all vertices:

```
FOREACH n IN g(type=N)
  GET n.value
```

Apart from retrieval, this query also involves renaming `g(type=N)` to $n$. Therefore, GRAPHiQL compiles it to $\pi_{n.value}(\rho_n(\texttt{g(type=N)}))$, where $\rho$ is the rename operator.

*Update.* An update operation modifies entries in the Graph Table. We denote an update as: $g \leftarrow g'$. Again, updates within or without an iterator translate identically.

*Aggregate.* An aggregate operation computes statistics (e.g. sum, count, etc.) over the Graph Table. For example, `COUNT(g(type=N))` compiles to $\gamma_{count}(\texttt{g(type=N)})$. Encapsulating an aggregator in an iterator loop introduces a group-by, i.e. the aggregate is computed for each item in the loop. For example, consider the following statements to compute the out-degree of each vertex:

```
FOREACH n IN g(type=N)
  COUNT(n.out(type=E))
```

This will be compiled in GRAPHiQL as follows:

$$\gamma_{\text{count}(n.out(type=E))}(\Gamma_{n.id}(\rho_n(\texttt{g(type=N)})))$$

GRAPHiQL treats the recursive associations of a Graph Table, e.g. $n.out, n.in$, as projections of the Graph Table. Thus, we can rewrite the above query by pushing down projections as follows:

$$\gamma_{\text{count}}(\Gamma_{n.id}(\pi_{n.id,n.out(type=E)}(\rho_n(\texttt{g(type=N)}))))$$

The above query first applies the projection ($n.id$ and $n.out$(type=E)) before applying the group-by and the count. We can further decompose the projection in the above query into a join between two views on the Graph Table, as follows:

$$\gamma_{\text{count}}(\Gamma_{n.id}(\pi_{n.id,n'}(\rho_n(\texttt{g(type=N)}) \bowtie \rho_{n'}(\texttt{g(type=N).out(type=E)}))))$$

Thus, aggregates in GRAPHiQL can be compiled to a combination of project, group-by, aggregate, and join operators.

We see that after compiling the Graph Table manipulations to relational operators, a query looks very similar to a relational query. The only missing piece is to translate the Graph Table expressions to relational tables. We describe these next.

*2)* ***Graph Tables to Relational Tables:*** GRAPHiQL physically stores a Graph Table as relational tables in a database system. By default, it creates two tables, a list of nodes $N$ and a list of edges $E$, i.e. $N$ and $E$ are two relational views of a Graph Table. However, we can create other views as well, e.g. $N \bowtie E$. When compiling a query, GRAPHiQL translates all Graph Table expressions into manipulations over the relational tables We show the typical Graph Table mappings below:

$$\begin{aligned}
&\texttt{g(type=N)} \mapsto N\\
&\texttt{g(type=E)} \mapsto E\\
&\texttt{g(type=N).out(type=E)} \mapsto N \bowtie E\\
&\texttt{g(type=E).out(type=E)} \mapsto E \bowtie E\\
&\texttt{g(type=N).out(type=N)} \mapsto N \bowtie E \bowtie N\\
&\texttt{g.out.in} \mapsto \texttt{g.in}\\
&\texttt{g.in.out} \mapsto \texttt{g.out}
\end{aligned}$$

With these, $\rho_n(\texttt{g(type=N)})$ in the above query is simply $N$ and $\rho_{n'}(\texttt{g(type=N).out(type=E)})$ can be denoted as $N' \bowtie E$. The entire query can then be written as follows:

$$\gamma_{\text{count}}(\Gamma_{n.id}(\pi_{n.id,n'}(N \bowtie (N' \bowtie E))))$$

This could be further simplified to:

$$\gamma_{\text{count}}(\Gamma_{n.id}(\pi_{n.id}(N \bowtie E)))$$

The above query expression is standard relational algebra (easily translatable into SQL) and could be readily executed on any relational database of choice.

*3)* ***Tuning Relational Algebra Expressions:*** The previous sections showed that we can compile GRAPHiQL queries to relational algebra expressions. A major advantage of doing this is that we can now readily apply several standard relational query optimization techniques in order to improve

| GRAPHiQL Analysis | Compiled Relational Algebra Expression |
|---|---|
| PageRank | $l_{i=1}^{10}(n.value \leftarrow \gamma_{\text{sum}}(\Gamma_{n.id}((N \bowtie E \bowtie N') \bowtie (\gamma_{\text{count}}(\Gamma_{id}(N'' \bowtie E))))))$ |
| Shortest Path | $l^{\text{updateCount}>0}(n.value \leftarrow \sigma_{n.value>v'}(\gamma_{\text{min(n'.value)+1}}(\Gamma_{n.id}(N \bowtie E \bowtie N'))))$ |
| Triangles (global) | $\gamma_{\text{count}}(\sigma_{e_1.\text{from}=e_3.\text{to}}(E_1 \bowtie E_2 \bowtie E_3))$ |
| Triangles (local) | $\gamma_{\text{count}}(\Gamma_{e_1.\text{from}}(\sigma_{e_1.\text{from}=e_3.\text{to}}(E_1 \bowtie E_2 \bowtie E_3)))$ |
| Strong Overlap | $\sigma_{\text{count}>t}(\gamma_{\text{count}}(\Gamma_{e_1.\text{from},e_2.\text{from}}(E_1 \bowtie E_2)))$ |
| Weak Ties | $\sigma_{\text{count}>t}(\gamma_{\text{count}}(\Gamma_{e_1.\text{to}}((E_1 \bowtie E_2) \ltimes E_3)))$ |

TABLE II. COMPILED RELATIONAL ALGEBRA EXPRESSIONS FOR GRAPHIQL QUERIES.

query performance. In the following, we outline some of these techniques. For a more detailed overview on optimizing relational queries, please see [34].

*De-duplication of Graph Elements.* For undirected graphs, recursive access to the Graph Table can create duplicate graph elements. For example, `g(type=E).out(type=E)` will include every pair of edges twice, one in each direction, as well as include cycles. GRAPHiQL avoids that by introducing an additional selection predicate as follows: $\sigma_{e_1.from<e_1.to \wedge e_2.from<e_2.to}(E_1 \bowtie E_2)$.

*Selection Pushdown.* As in standard relational queries, GRAPHiQL pushes down the selections as much as possible, i.e. only the relevant graph elements are passed to the higher level operators. For instance, with selection pushdown `g(type=E).out(type=E)` would be compiled as follows: $(\sigma_{e_1.from<e_1.to}(E_1)) \bowtie (\sigma_{e_2.from<e_2.to}(E_2))$.

*Cross-product As Join.* A cross product followed by a selection can be mapped to a join operator. For example, we can get the overlap between all pairs of nodes as follows:

```
FOREACH n1 in g(type=N)
  FOREACH n2 in g(type=N)
    GET n1.id,n2.id,COUNT(
       n1.out(type=N).out(type=N,id=n2.id)
    )
```

This would be compiled to:

$$\gamma_{\text{count}}(\Gamma_{n1.id,n2.id}(\sigma_{n_1.id=n_2.id}(\rho_{n_1}(E \bowtie E) \times \rho_{n_2}(N))))$$

However, the above statement could be optimized to:

$$\gamma_{\text{count}}(\Gamma_{n1.id,n2.id}(\rho_{n_1}(E \bowtie E) \bowtie \rho_{n_2}(N)))$$

*Redundant Joins.* Finally, GRAPHiQL prunes several redundant joins. For instance, if in $N \bowtie E$ no attribute of $N$ other than its *id* is used, then GRAPHiQL removes the redundant join and simplifies it to $E$. Likewise, $N \bowtie N$ is simplified to $N$, given that its join key is also the primary key.

Thus, we see that GRAPHiQL queries can be compiled as well as tuned to relational algebra expressions. Table II shows the compiled algebraic expressions for the six application queries from Section IV.

## VI. PERFORMANCE

In this section, we study the performance of GRAPHiQL on several real graphs running on a relational engine. Specifically, we ran GRAPHiQL on Vertica [35] and compared it against Apache Giraph [2], a popular system for graph analytics. We tested the five applications discussed in Section IV, namely PageRank, shortest path, triangle counting (both global and local), strong overlap, and weak ties. Note that PageRank and shortest path are suited for directed graphs whereas the remaining make more sense over undirected graphs. We chose both a small and a large dataset for both directed and undirected scenarios, summarized as follows:

- *Directed*: small Twitter dataset (81,306 nodes and 1,768,149 edges) and large LiveJournal dataset (4,847,571 nodes and 68,993,773 edges), and
- *Undirected*: small Amazon dataset (334,863 nodes and 925,872 edges) and large LiveJournal dataset (3,997,962 nodes and 34,681,189 edges)

All datasets are available online [36]. We ran the above analysis on a machine having a 2GHz 2x6 Xeon (24 total hardware threads with hyper-threading), 48GB of memory, 1.4T disk, running RHEL Santiago 6.4. We ran all experiments with a cold disk cache and report the average of three runs.

The top part of Table III shows the performance of GRAPHiQL compared to Giraph on the small graph. We can see that GRAPHiQL outperforms Giraph significantly, by a factor ranging from 6.3 for weak ties to 15.2 for global triangle counting. The main reason for this huge difference is that Giraph suffers from a significant initialization cost and MapReduce overheads, even for small sized graphs. On the other hand, GRAPHiQL compiles user queries into efficient relational operators with no such overheads.

Table III also shows the performance of GRAPHiQL and Giraph on the large graph. GRAPHiQL outperforms Giraph by 6.5 times on PageRank and 2.1 times on shortest path. Furthermore, Giraph could not run the remaining queries, namely triangle counting, strong overlap, and weak ties, on the large graph since it runs out of memory. This is because these queries involving passing the 1-hop neighbors in the first superstep and incurring significant data shuffling overhead. GRAPHiQL, on the other hand, translates these queries into efficient join operations which are highly tuned in Vertica. As a result, it does not suffer from these issues.

Of course, these results are not exhaustive; evaluating GRAPHiQL on larger graphs, multiple nodes, and against multiple systems remains an important future step, but we believe that these encouraging preliminary results, combined with the simplicity of the GRAPHiQL language suggest the merits of using relational engines for graph query execution, coupled with a compiled language like GRAPHiQL.

## VII. OTHER RELATED WORK

Apart from Pregel and SQL, a number of other query languages have been considered for graph analytics. Below we review the major ones.

**Imperative languages.** Imperative languages provide full control to the programmers on how to perform a given task by specifying the sequence of instructions to run. Green Marl [37], for instance, allows programmers to write elaborate C-style

| Graph | System | PageRank | Shortest Path | Triangles (global) | Triangles (local) | Strong Overlap | Weak Ties | *Average* |
|---|---|---|---|---|---|---|---|---|
| *Small* | Giraph | 46.96 | 43.67 | 47.12 | 46.22 | 42.10 | 50.51 | 46.10 |
| | GRAPHiQL | 3.31 | 2.95 | 3.10 | 4.93 | 3.49 | 8.00 | 4.30 |
| | Improvement | 14.2 | 14.8 | 15.2 | 9.4 | 12.1 | 6.3 | **11.99** |
| *Large* | Giraph | 190.42 | 115.46 | out of memory | out of memory | out of memory | out of memory | 152.94 |
| | GRAPHiQL | 29.42 | 54.42 | 287.77 | 841.51 | 1211.73 | 1483.31 | 651.36 |
| | Improvement | 6.5 | 2.1 | - | - | - | - | **4.30** |

TABLE III. THE QUERY PERFORMANCE OF GRAPHIQL COMPARED TO GIRAPH OVER SIX APPLICATION QUERIES.

programs for analyzing graphs, i.e. programmers have control over the entire graph. However, writing such elaborate code for every graph analysis could quickly become very messy. The likely workflow for a graph analyst is to perform an analysis, analyze the results, and move on to yet more analyses. Such users would want to quickly analyze the graph and not burdened with writing and maintaining elaborate code.

**XPath.** XPath is generally considered a pattern matching query language for XML. However, some recent graph languages, such as Cypher [38] and Gremlin [39], also make use of XPath for graph traversals. The beauty of XPath is that it allows programmers to express their queries succinctly, compared to verbose and clumsy implementations in imperative languages. However, in spite of being succinct, it is not easy to formulate such path expressions in the first place. Additionally, XPath-style languages constrain programmers to think in terms of path expressions, whereas graph analysts are often interested in broader graph analysis.

**Datalog.** Datalog is a declarative language in which queries are expressed as logical rules. There is a renewed interest in Datalog for graph analysis. This is because Datalog has natural support for recursion, a key characteristic of several graph analysis. Socialite [15], for instance, is a recently proposed Datalog-based system for graph analytics. Unfortunately, in spite of being a powerful language, Datalog has not been the language of choice for analysts. This is because Datalog is primarily an experts language and Datalog queries are often too complex to be understood by non-expert users.

**SPARQL.** SPARQL is semantic web language designed for web applications that involve reasoning such as inferring new relationship triples. Although primarily used for subgraph retrieval, SPARQL also supports post-processing the subgraph, e.g. computing aggregates. As a result, SPARQL increasingly looks some kind of a vanilla SQL without stored procedures. Still, SPARQL works well primarily for extracting specific points in the graph based on their relationships, and not good for graph analytics [40]. As a result, it is very difficult to express queries such as PageRank in SPARQL.

## VIII. CONCLUSION

In this paper, we presented GRAPHiQL, a graph intuitive query language for relational databases. GRAPHiQL allows developers to reason about more natural graph representations, i.e. nodes and edges, rather than dealing with relational tables and joins. GRAPHiQL provides several key graph constructs, such as looping and neighborhoods. As a result, GRAPHiQL shifts the focus of the developer back to his analysis. At runtime, GRAPHiQL compiles user queries to optimized SQL queries, which can run on any relational engine. Our experiments show that GRAPHiQL queries run significantly faster on a relational database compared to Giraph: 12x faster on small graph and 4.3x faster on large graph. Essentially, GRAPHiQL combines the best of both worlds — ease-of-use of procedural languages and good performance of declarative languages. In our future work, we will build an optimizer to tune the performance of GRAPHiQL queries to the underlying RDBMS.

## REFERENCES

[1] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," *SIGMOD*, 2010.

[2] "Apache Giraph," http://giraph.apache.org.

[3] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, 2012.

[4] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," *OSDI*, 2012.

[5] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC," *OSDI*, 2012.

[6] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation," http://ipads.se.sjtu.edu.cn/ projects/powerswitch/PowerSwitch-IPADSTR-2013-003.pdf, 2014.

[7] S. Salihoglu and J. Widom, "GPS: A Graph Processing System," *SSDBM*, 2013.

[8] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud," *SIGMOD*, 2013.

[9] G. Wang, W. Xie, A. Demers, and J. Gehrke, "Asynchronous Large-Scale Graph Processing Made Easy," *CIDR*, 2013.

[10] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke, "Fast Iterative Graph Computation with Block Updates," *PVLDB*, vol. 6, no. 14, pp. 2014–2025, 2013.

[11] "Pregelix," http://hyracks.org/projects/pregelix.

[12] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson, "From "Think Like a Vertex" to "Think Like a Graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[13] A. Quamar, A. Deshpande, and J. Lin, "NScale: Neighborhood-centric Large-Scale Graph Analytics in the Cloud," http://arxiv.org/abs/1405.1499, 2014.

[14] ——, "NScale: Neighborhood-centric Analytics on Large Graphs," *VLDB*, 2014.

[15] M. S. Lam, S. Guo, and J. Seo, "SociaLite: Datalog Extensions for Efficient Social Network Analysis," *ICDE*, 2013.

[16] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis," *VLDB*, 2013.

[17] W. E. Moustafa, H. Miao, A. Deshpande, and L. Getoor, "A System for Declarative and Interactive Analysis of Noisy Information Networks," *SIGMOD*, 2013.

[18] W. E. Moustafa, G. Namata, A. Deshpande, and L. Getoor, "Declarative Analysis of Noisy Information Networks," *GDM*, 2011.

[19] S. Sakr, S. Elnikety, and Y. He, "G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs," *CIKM*, 2012.

[20] "Apache Jena," http://jena.apache.org.

[21] "AllegroGraph," http://franz.com/agraph/allegrograph.

[22] "Neo4j," http://www.neo4j.org.

[23] “HyperGraphDB,” http://www.hypergraphdb.org.

[24] N. Bronson *et al.*, “TAO: Facebook’s Distributed Data Store for the Social Graph,” *USENIX ATC*, 2013.

[25] “FlockDB,” http://github.com/twitter/flockdb.

[26] “Counting Triangles with Vertica,” http://www.vertica.com/2011/09/21/counting-triangles.

[27] S. Lawande, L. Shrinivas, R. Venkatesh, and S. Walkauskas, “Scalable Social Graph Analytics Using the Vertica Analytic Platform,” *BIRTE*, 2011.

[28] A. Welc, R. Raman, Z. Wu, S. Hong, H. ChaÞ, and J. Banerjee, “Graph Analysis Ð Do We Have to Reinvent the Wheel?” *GRADES*, 2013.

[29] J. Huang, K. Venkatraman, and D. J. Abadi, “Query Optimization of Distributed Pattern Matching,” *ICDE*, 2014.

[30] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, “GraphX: A Resilient Distributed Graph System on Spark,” *GRADES*, 2013.

[31] A. Jindal, P. Rawlani, and S. Madden, “Vertexica: Your Relational Friend for Graph Analytics!” *VLDB*, 2014.

[32] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig Latin: A Not-so-foreign Language for Data Processing,” *SIGMOD*, 2008.

[33] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “The HaLoop Approach to Large-Scale Iterative Data Analysis,” *VLDB J.*, vol. 21, no. 2, pp. 169–190, 2012.

[34] S. Chaudhuri, “An Overview of Query Optimization in Relational Systems,” *PODS*, 1998.

[35] “Vertica,” http://www.vertica.com.

[36] “SNAP datasets,” http://snap.stanford.edu/data.

[37] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: A DSL for Easy and Efficient Graph Analysis,” *ASPLOS*, 2012.

[38] “Cypher,” http://www.neo4j.org/learn/cypher.

[39] “Gremlin,” https://github.com/tinkerpop/gremlin.

[40] “Graph Analytics,” http://blogs.teradata.com/data-discovery/tag/graph-analytics.