The case against specialized graph analytics engines

Jing Fan
University of Wisconsin
fanj@cs.wisc.edu

Adalbert Gerald Soosai Raj
University of Wisconsin
gerald@cs.wisc.edu

Jignesh M. Patel
University of Wisconsin
jignesh@cs.wisc.edu

ABSTRACT

Graph analytic processing has started to become a nearly ubiquitous component in the enterprise data analytics ecosystem. In response to this growing need, various specialized graph processing engines have been created in recent years. Sadly, the use of relational database management systems (RDBMSs) for graph processing is largely ignored in most enterprise settings. This oversight is surprising since in most enterprise settings, RDBMSs are already present and used for a variety of other analytic tasks. This situation then begs the question of whether the use of RDBMS for graph processing is fundamentally lacking in some respect compared to the specialized graph processing engines. In this paper, we aim to address this question both from the programmer productivity perspective and from the performance perspective. We present Grail – a syntactic layer for querying graph in a vertex-centric way in an RDBMS, which can be compiled to translate graph queries to SQL. In a single node setting, we also compare Grail to GraphLab and Giraph, and examine the performance implications of using Grail, showing that the RDBMS engine is competitive to these specialized engines. Given that RDBMSs are ubiquitous in enterprise settings, and have a robust and mature technology that has been hardened over decades, and are part of existing administrative methods in place, we argue that it is time to reconsider if specialized graph engines have a role to play in most enterprises.

1. INTRODUCTION

Graph problems have now become mainstream instead of being an esoteric class of computation. In response to the growing popularity for graph analyses, a large number of specialized graph engines have emerged, including Pregel [12], GraphLab [11], Giraph [1], and GPS [15]. These specialized graph engines sit in a broader enterprise ecosystem where there are typically already some existing relational data processing platforms (e.g. an RDBMS) for executing "traditional" data analysis tasks. The idea of using the relational platform for graph analysis has largely been ignored in favor of using these newer graph platforms.

The central question we concern ourselves with in this paper is if this idea of "one size fits one" makes sense for graph analytics. A key feature that these specialized graph engines have going for them is that they provide a vertex-centric way of graph programming, which is intuitive for the end (graph analytics) application developer to use. We concede that the graph engines have this advantage, but it is possible to provide much of this programmer convenience using a syntactic layer on top of SQL that can be run on an RDBMS. We present a method for **Gr**aph **A**nalysis **i**n **L**egacy Systems (i.e. **Grail**) that presents such a syntactic mapping layer. We present how graph analytics queries using the popular vertex-centric approach can be expressed in Grail, and also present how Grail queries can be translated to vanilla SQL. In addition, we also present techniques to optimize the generated SQL code to improve the execution of the translated SQL queries.

We compare Grail to two popular specialized graph engines, namely GraphLab and Giraph, and demonstrate the Grail approach is competitive in performance. The Grail approach generally also allows more graceful scaling to larger datasets, while the other two systems fall over fairly easily when dealing with datasets that don't fit in memory.

Finally, we note that – similar to the arguments made for XML (e.g. [17]), column-stores (e.g. [6]), and JSON (e.g. [8]) – the advantages of extending an RDBMS for "non-core relational" processing has the usual benefits of leveraging existing robust mature technology that has already been hardened, has various well-known manageability tools, and administrators that know how to deploy the technology at scale. In addition having one fewer system in the enterprise ecosystem (i.e. a new graph engine/platform) reduces the overall cost of managing and administrating the entire enterprise ecosystem as it reduces the administration overhead that tends to grow quickly when one has specialized engines for each application category.

The remainder of this paper is organized as follows: Sections 2 and 3 describe the systems and the algorithms that are used in this paper. The Grail approach is described in Section 4. Section 5 contains experimental results. Section 6 describes related work, and Section 7 contains our concluding remarks and points to some directions for future work.

This article is published under a Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/), which permits distribution and reproduction in any medium as well allowing derivative works, provided that you attribute the original work to the author(s) and CIDR 2015.
7th Biennial Conference on Innovative Data Systems Research (CIDR '15) January 4-7, 2015, Asilomar, California, USA

2. SYSTEM TESTED

In this section, we describe the three engines that we consider in this paper. These engines adopt the general Bulk Synchronous Parallel (BSP) programming model [20]. The engines that we consider are representative of three distinct approaches to graph analysis, namely RDBMS-based, Hadoop-based, and a specialized native graph engine.

2.1 RDBMS-based Approach: SQL Server

In this study, we use a single-node Microsoft SQL Server 2014 as the RDBMS. It provides T-SQL as the SQL language surface. We pick SQL Server because it is a representative mature RDBMS product, but our results should generalize to other RDBMSs. Like other commercial RDBMSs, SQL Server has built-in buffer management system to enable efficient disk-RAM data exchange.

RDBMS engines also scale-out across nodes, and exploring that performance aspect is a good topic for follow-on work, and not considered in this initial position paper.

2.2 Hadoop-based approach: Giraph

Apache Giraph is an open-source iterative graph processing system. Giraph runs jobs on Hadoop and uses HDFS to store both input and output data. Giraph includes out-of-core computation, spilling data to disk as needed.

2.3 Native Graph Engine: GraphLab

GraphLab is an open-source graph analytic platform based on a Gather-Apply-Scatter model. GraphLab is a key example of a specialized graph engine.

3. ALGORITHMS

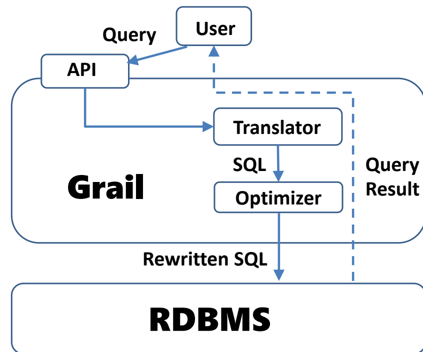
In this section, we describe three common graph analytic algorithms, and use them to make our argument.

3.1 Single Source Shortest Path (SSSP)

SSSP finds the minimum cost path between a given source vertex and all other vertices. A parallel variant of the Bellman-Ford algorithm is commonly used to evaluate SSSP. Initially, the source vertex distance is set to 0, and it is the only active vertex; all other vertices have the initial distance set to ∞. In each iteration, every active vertex v sends a message to each node that is directly connected to the vertex v. Each such neighbor node, u, receives a message containing its current minimum distance plus the weight of the edge. In the next iteration, the vertices that have received messages become active, and mutate their (vertex) value to the minimum of its received messages. Updated vertices then send messages to their neighbors. The program terminates when no vertices are active and no messages remain in the system.

As an example, consider the case when the weight of each edge is 1. Then, the shortest path value of a vertex v is updated as: $SP(v) = min(SP(v), min_{u \in N(v)}(SP(u) + 1))$, where $SP(v)$ is the shortest path value of v, and $N(v)$ is the set that has an edge pointing to the vertex v.

Figure 1: Architecture of Grail



3.2 PageRank

The PageRank algorithm was designed to rank web pages, counting the number and quality of links to a page to determine an estimate of how important each page is. In each iteration, the PageRank of a vertex v is updated as:

$$PR(v) = d \times \Sigma_{u \in N(v)} \frac{PR(u)}{L(u)} + (1 - d)$$

where d is the damping factor, $PR(v)$ is the pagerank value of v, $L(u)$ is the out-degree of the vertex u. A typical default value for d is 0.85, and we use that default value.

3.3 Weakly Connected Component (WCC)

WCC computes the maximal subgraph in a directed graph, such that every pair of vertices in the subgraph is mutually reachable when replacing the directed edges with undirected edges. In WCC, all the vertices are initially active and use their own vertex id as the component id value. In each iteration, each active vertex updates its component id value if a smaller component id value is found. Each updated vertex then send messages containing its current component id value to its neighbors. Execution stops when all the vertices are inactive and there are no new messages.

4. GRAIL

A key feature that the specialized graph engines tout is a (graph) programmer-friendly API. RDBMS engines provide SQL, which is not a natural way to express graph analysis tasks. In this section we present Grail – a syntactic layer on top of SQL that provides a programmer-friendly interface for expressing graph analytic queries.

We note that even this area of designing APIs and/or languages for graph analytics is an active area of research, with a number of recent Datalog-inspired proposals, such as [16, 18, 7]. But these previous efforts largely end up building a new data processing platform. We also note that in the future, it may make sense to generate other APIs/languages like Grail to map to SQL engines. A goal of this paper is simply to show that this mapping can be done fairly easily with vanilla SQL.

4.1 Overview

The architecture of the Grail approach is shown in Figure 1. There are three parts: an API, a translator, and an opti-

mizer. The API provides a vertex-centric way to express the query, the translator maps the query to a runnable SQL script, which is T-SQL in our case. An optimizer component is used to generate "efficient" T-SQL queries (note that this optimizer optimizes the translation to suit the specific characteristics of the underlying RDBMS engine and is complementary to the query optimizer that sits inside the RDBMS).

4.2 Data Model

In Grail, the graph data is stored in the underlying DBMS using a set of tables. In this section we introduce the underlying data model.

First, we note that conceptually both Giraph and Graphlab use separate classes to define the graph using two classes: Vertex and Edge.

```
class Vertex {
    I id;
    V data;
    List<Edge> edges;
}
```

```
class Edge {
  I target;
  E data;
}
```

We mirror this model, and store the edges and the vertices as tables in a RDBMS, using the schema defined in Table 1. In Table 1 there are "permanent" tables to store the base data, and "intermediate" tables that are used as temporary relations during the graph computation. The final result is stored in the next table.

The list above is not complete and in some cases additional tables are needed based on the variables used in user's query. For example, the out_cnts table can be created to record out-degrees of vertices, and the updated table is needed to record the updated vertices in the current iteration.

4.3 Computation Model

The computation models of Giraph, GraphLab and Grail differ in the way that vertices communicate with each other using messages/signals, and the ability/inability of vertices to extract information from other vertices. Here we briefly describe the computational models of the two specialized graph engines and Grail.

4.3.1 Giraph

Giraph is a vertex-centric graph system that uses a message passing model for communication. Computation is organized into *supersteps* (i.e. iterations).

More specifically, in Giraph vertices communicate directly with each other by sending messages. In every *superstep*, a vertex can execute a user-defined function to receive messages, do local computation, and send messages. For example, the single source shortest path algorithm works as shown in Figure 2. Here, we are computing the shortest path from vertex A to every other vertex in the graph. In iteration (#1), vertex A sends messages to vertices B and C, and updates its own shortest path distance to be zero. Then, in iteration (#2), vertices B and C use the messages that it received (at the beginning of this iteration from vertex A) to modify their shortest path distance from the vertex A. In this same iteration, vertex B also sends a message to vertex D. This message contains the destination vertex, and the sum of the value of the source vertex and the weight of the edge between them. Similar processing happens at vertex C in this iteration (#2), resulting in a message being sent from vertex C to vertex D. Finally in iteration (#3), vertex D uses the messages it received from vertices B and C to update its value (of the shortest path distance from A). Since vertex D doesn't have any out-going edges, it doesn't send any messages, and the algorithm terminates. Now every vertex has the distance of the shortest path from vertex A. It is important to note that in Giraph the data is moved to the computation, and the computation is carried out at the vertices.

4.3.2 GraphLab

GraphLab is based on the Gather-Apply-Scatter model. In each iteration, GraphLab performs Gather, Apply, and Scatter functions serially for every vertex.

In contrast to Giraph, rather than communicating using messages, GraphLab employs a signaling mechanism for communication. In each iteration, vertices receiving the signals get activated, and GraphLab performs Gather, Apply, and Scatter functions serially for every active vertex, in each iteration. GraphLab uses a shared-memory view of computation, which means that data on adjacent vertices and edges are directly accessible by an active vertex, and there is no need to use messages to transfer these data values. (In a cluster environment, each vertex has a master node, and multiple vertices could be mastered by a single node. Furthermore, ghost vertices are created for all vertices that are adjacent to vertices that are mastered at each node. Then, signals sent to a ghost vertex results in sending a message to the corresponding master node.) For example, in Figure 2, to find the shortest path distance from vertex A to all other vertices, in iteration (#1), vertex A performs its gather, apply and scatter functions. This iteration results in the vertex A signaling the vertices B and C. In iteration (#2), the vertices B and C become activated. Then, they read values from their incoming edges, and also data values from the vertex A (in the gather phase). Finally, they use these gathered values to update their own vertex values during the apply phase. A (barrier-based) synchronization between the gather and the apply phases ensures that the updates in the apply phase strictly happen after the reads have completed in the gather phase. The last step in iteration (#2), is the scatter phase (which is also super-seeded by a barrier synchronization), in which the vertices scatter their updated values by signaling their neighbors; in this case vertex D is signaled. Finally in iteration (#3), vertex D updates its shortest path distance (from the vertex A) based on the data at vertices B and C and the information about the incoming edges. It is important to note that in GraphLab the data is not copied and sent to where it is used for computation, but rather the data values are simply accessed when they are needed for computation.

Asynchronous Execution in GraphLab. The above description for GraphLab was for its *synchronous* mode of computation. GraphLab can also execute in an *asynchronous* mode where the PowerGraph [9] engine can execute computation for active vertices as and when processor and network resources become available. To address the problem of non-determinism that may arise due to asynchronous execution,

Table 1: Schema Definitions: In some systems (e.g. SQL Server) next, message and update are reserved keywords and can be substituted for other names. In the interest of readability, we use these keywords here.

Schema		Definition
Permanent	`edge(src, dest, data, val)`	Original directed graph edges. src and dest are the source vertex id and the destination vertex id of this edge respectively, data contains some properties of this edge that is irrelevant to the computation (e.g. edge establishment time), and val represents some property of this edge that is relevant to the computation (e.g. edge weight).
	`vertex(id, data, val)`	Original vertices. id represents the vertex id of the vertex, data contains some property of the vertex that is irrelevant to the computation (e.g. vertex description), and val contains some property of the vertex that is relevant to the computation (e.g. indegree of a vertex).
Intermediate	`next(id, val)`	Values for the vertices in the next iteration. id is the vertex id of the vertex, and val is the relevant value of the vertex that would be useful in the next iteration.
	`cur(id, val)`	Values for the gathered messages. id is the vertex id of the vertex, and val is the aggregated value for the vertex.
	`message(id, val)`	Messages passed to neighbors. id is the vertex id of the target vertex, and val is the message value.

Table 2: From Vertex-centric Verbs to Relational Algebra

Vertex Centric	Relational Algebra	Meaning
Receive Messages	cur $\longleftarrow \gamma_{id,F_0(val)}(message)$	Group by and aggregate
Mutate Value	$next \overset{u}{\longleftarrow} \pi_{next.id,F_1(other.val)} other \bowtie_{id} next$	oldvalue ← newvalue
Send Messages	$\pi_{edge.B,F_2(other.val,edge.val)} other \bowtie_{other.id=edge.A} edge$	Join other table and edge in the sending direction

PowerGraph uses a locking-based protocol to provide strong serializability guarantees.

4.3.3 Grail

Grail's overall computational model structure is similar to the message passing model in Giraph. The difference is that in an RDBMS, vertex data, message data, and edge data are all stored in relational tables. Furthermore, the basic execution paradigm needs to leverage set-oriented processing. For example, while executing the single source shortest path analysis in Grail, the values of the next and message tables for the three iterations are shown in Figure 3. The values of the vertex and the edge tables remain the same across all iterations, and they are also shown in Figure 3. The vertex table contains the vertex id and the initial value of each vertex. The edge table contains the source id, destination id and the value of the corresponding edge. In iteration 1, vertex A updates its own value and sends messages to vertices B and C. This step can be seen in the next and message tables shown for iteration (#1). The value for vertex A is updated to zero since the distance from vertex A to itself is zero. The values for other vertices B, C, and D, remain at 100 (which represents infinity in this example). The message table in iteration (#1), shows that there are two messages that are generated during this iteration: one for the vertex B with value 1 and another for the vertex C with value 2. In iteration (#2), the vertices B and C update their values in the next table to 1 and 2 respectively, using the messages that they received from the vertex A, at the start of this iteration. The message table shows the two messages that are generated by the vertices B and C during this iteration. Finally, in iteration (#3), the vertex D updates its value in the next table using the messages it received from the vertices B and C at the start of this iteration. The vertex D has no out-going edges, and so no messages are sent during this iteration, and the algorithm terminates.

4.3.4 Cost Evaluation

In this section, we compare the computational model that is used in Giraph, GraphLab, and Grail. Giraph and Grail use the same message passing-like computational model, while GraphLab adopts the Gather-Apply-Scatter model. However, GraphLab needs to send signals to its neighbors, which functionally is similar to sending messages, except that the data associated with a vertex is not transferred. Suppose that in iteration i, Giraph and Grail receive M_i messages at the beginning of the iteration. Similarly, the number of signals received by GraphLab should also be M_i. Assume that the number of vertices in this iteration that need to do work (i.e. the "active" vertices) is A_i. Furthermore, assume that M_{i+1} new messages or signals are generated for the next iteration, and that the average number of neighbors that are touched in the Gather phase in GraphLab is k. Then, we can model the time that is used by these three systems in iteration i as shown below. The unit time cost to handle, gather, compute and send messages are shown in Table 3. Of course, each of these steps can have different physical costs based on the implementation, creating opportunities for implementation-based optimizations.

Let the execution time for Grail, Giraph, and GraphLab (in synchronization mode) be denoted as T_{Gr}, T_{Gi}, T_{GL} respectively. Then:

$$\begin{aligned} T_{Gr} &= handle * M_i + compute * A_i + gen * M_{i+1} \\ T_{Gi} &= handle * M_i + compute * A_i + gen * M_{i+1} \\ T_{GL} &= handle * M_i + gather * A_i * k \\ &\quad + compute * A_i + gen * M_{i+1} \end{aligned}$$

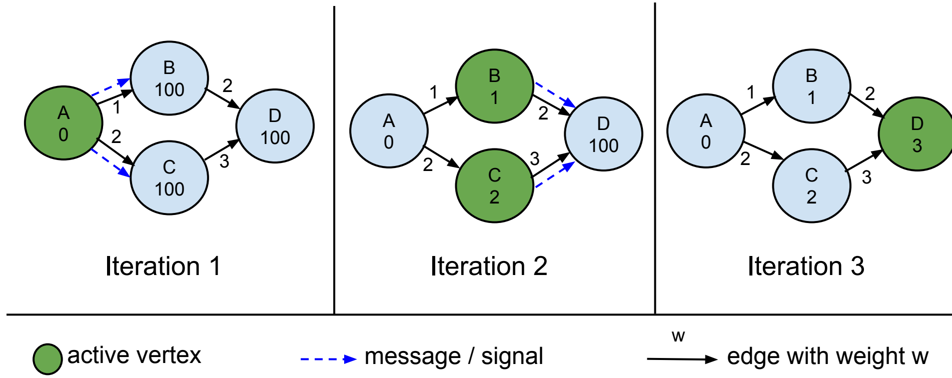


Figure 2: Execution of the Single Source Shortest Path algorithm with Giraph and GraphLab. The distance between the connected vertices are shown as labels on the edges. The distance of the shortest path of each vertex from the source vertex A is written below the name of the vertex. A distance value of 100 is assumed to represent infinity.

vertex

id	val
A	100
B	100
C	100
D	100

edge

src	dest	val
A	B	1
A	C	2
B	D	2
C	D	3

Iteration 1

next

id	val
A	0
B	100
C	100
D	100

message

id	val
B	1
C	2

Iteration 2

next

id	val
A	0
B	1
C	2
D	100

message

id	val
D	3
D	5

Iteration 3

next

id	val
A	0
B	1
C	2
D	3

message

id	val

Figure 3: The tables used by Grail for the Single Source Shortest Path algorithm are shown here. The vertex and the edge tables are shown only once since they have the same values in all the three iterations. The instances of the next and the message tables are shown for each iteration.

Name	Meaning
handle	Time taken to handle a message/signal
gather	Time taken to gather a value from a neighbor
compute	Time taken to do local computation
gen	Time taken to generate & send a message/signal
check	Time taken to check whether to send a signal

Table 3: Unit Time Definition

GraphLab can reduce the number of signals in some circumstances. For example, in the shortest path algorithm, in the scatter stage, a vertex can compare the neighbor value with its value plus the edge value; then, if the former value is smaller, the vertex can choose not to signal the neighbor. Let us assume that this optimization can reduce the number of signals in iteration i by M_i'. Thus, one can reduce at most M_i' active vertices in a current iteration, but this optimization will require M_i' more checks in the previous iteration. So, the GraphLab's time cost becomes:

$$\begin{aligned} T_{GL} \geq\ & handle * (M_i - M_i') \\ & + gather * (A_i - M_i') * k + compute * (A_i - M_i') \\ & + gen * (M_{i+1} - M_{i+1}') + check * M_{i+1}' \end{aligned}$$

Thus, Grail and Giraph have the same high-level cost model, while GraphLab's cost model has some variations. GraphLab can trade off some check time for number of messages. But the total complexity of all three models is similar.

4.4 Grail: Translating to SQL

In Giraph, in every *superstep* (iteration), each vertex does three things: Receive messages, Mutate values, and Send messages. These *verbs* enable vertex-centric programming. Our approach is to map these primitives to relational algebra (RA) using the transformation rules that are shown in Table 2. With the RA mappings, its fairly easy to generate the

corresponding SQL statement[1]. Thus, with Grail we can map vertex-centric programming to SQL.

Notice that in Table 2 the current result set is stored in the table cur. Updated values in each iteration are stored in the table next. A table join is used to generate new messages and "pick the direction" in which to send the messages. There are three options for the direction – namely, to out-going edges, to in-coming edges, and to all edges. This direction is chosen by picking the "right predicate" for the join operation (the Send Messages operation shown in Table 2). If messages are sent through out-going edges, then the join should be on the src attribute of the edge table; if messages are sent through in-coming edges, then the join should be on the dest attribute of the edge table; if messages are sent through all edges, then we should use a "union all" operation to union the messages sent through out-going edges and in-coming edges.

Next, we illustrate and compare Grail with the traditional vertex-centric method for the SSSP analytics. Listing 1 shows the supersteps in Giraph, and the corresponding RA transformation is shown in Listing 2. Here the table update is used as flow control, and we join the update and the edge tables using $update.id = edge.src$ as we want to send messages to the vertices that are connected by the outgoing edges. (The program in the Grail API is discussed below.)

4.5 Grail API

The basic Grail API is shown in Table 4. The language is simple and intuitive as users are freed from specifying low-level operations like joins, and simply defining the task in each iteration from a vertex-centric perspective.

Listing 1: Pseudocode for SSSP in Giraph

```
// Combine Message
foreach (int msgVal : messages) {
    minDist = Math.min(minDist, msgVal);
}
if (minDist < vertexValue) {
    // Mutate Value
    mutateValue(minDist);
    // Send Message
    foreach (edge e : outEdges) {
        sendMessage(e.target, minDist + e.val);
    }
}
```

Listing 2: Relational Algebra for SSSP in Grail

$$cur \leftarrow \gamma_{id,MIN(val)}(message)$$
$$update \leftarrow \pi_{cur.id,cur.val}(cur \bowtie_{cur.id=next.id\ AND\ cur.val<next.val} next)$$
$$next \xleftarrow{u} \pi_{next.id,update.val} update \bowtie_{id} next$$
$$message \leftarrow \pi_{edge.dest,update.val+edge.val}(update \bowtie_{update.id=edge.src} edge)$$

[1]The translation shown in Table 2 is simplified. There are some nuance as the "Mutate Values" and "Send Messages" component may require additional joins if the query requires a more complex "MutateAndSend" function that is described in Table 4. Thus, the actual Grail translation is slightly more complicated.

4.6 Example

SSSP can be expressed in Grail as follows:

```
VertexValType: INT
MessageValType: INT
InitiateVal : INT_MAX
InitialMessage : (1, 0)
CombineMessage: MIN(message)
UpdateAndSend: update=cur.val<getVal()
               if (update) {
                 setVal(cur.val)
                 send(out, cur.val+1)
               }
End: NO_MESSAGE
```

The Grail generated T-SQL is shown in Listing 3.

Listing 3: T-SQL Ouput for SSSP

```
1  DECLARE @flag int;
2  SET @flag = 1;
3
4  SELECT vertex.id, 2147483647 AS val
5  INTO next
6  FROM vertex;
7
8  CREATE TABLE message(
9      id int,
10     val int
11 );
12 INSERT INTO message values(1,0);
13
14 WHILE (@flag != 0)
15 BEGIN
16     SELECT message.id AS id,
17            MIN(message.val) AS val
18     INTO cur
19     FROM message
20     GROUP BY message.id;
21
22     DROP TABLE message;
23
24     SELECT cur.id AS id,
25            cur.val AS val
26     INTO update
27     FROM cur, next
28     WHERE cur.id = next.id
29     AND cur.val <  next.val;
30
31     UPDATE next
32     SET next.val = update.val
33     FROM update, next
34     WHERE  next.id = update.id;
35
36     SELECT edge.dest AS id,
37            update.val + 1 AS val
38     INTO message
39     FROM update, edge
40     WHERE edge.src = update.id;
41
42     DROP TABLE cur;
43
44     DROP TABLE update;
45
46     SELECT @flag = COUNT(*)
47     FROM message;
48  END
```

Table 4: Grail API

Name	Description	Example	Meaning	Usage
VertexValType	The vertex data type	INT	The vertex data type is int	N/A
MessageValType	The message data type	INT	The message data type is int	N/A
InitiateVal	The initial value of the vertex at the beginning of the computation	id	All vertices should be initiated with vertex id	Initiate
InitialMessage	Initial message	(1, 0)	Sending message to vertex 1 with message value 0	Initiate
		(ALL, 0)	Sending messages to all vertices with message value 0	
CombineMessage	The aggregate function to combine messages	MIN(message)+1	Use the MIN aggregation operator on all incoming messages	Combine message

MutateAndSend	The operations to update the vertex in this iteration, and determine the messages to send for the next iteration	`update=cur.val < getVal()`
		`if (update) {`
		`setVal(cur.val)`
		`send(out, cur.val/out_cnts)`
		`}`
			Create the table `update` with two attributes `cur.id` and `cur.val`. Update the values for each vertex, and send messages to all out-going neighbors	Mutate Value
				Send Messages

End	End condition	(ITER, 10)	Stop after 10 iterations	Loop control
		NO_MESSAGE	Stop when there are no new messages.	

4.7 Optimization

There are optimizations that can be performed during the code-generation part of Grail to produce an "efficient" T-SQL program. We describe these optimizations next.

4.7.1 Index Creation

The generated (SQL) code has joins involving the tables `next` and `edge`. To speed up the join, we create an index on the `id` attribute of the `next` table, and an index on the `edge` table based on the direction in which the message is to be sent. When the message is sent to neighbors with an out-going edge, the index should be created on `edge(src, dest)`; for neighbors on the in-coming edge the index should be created on `edge(dest, src)`; for neighbors on both types of edges, both indices are created. These indices allow the SQL server optimizer to consider index-based query plans, and generally speed up the query processing. (As part of future work, it would be good to make this optimization cost-driven either in the code generation component of Grail, or directly inside the RDBMS optimizer.)

4.7.2 Avoiding Materializing the Messages Table

The default code generation results in materializing the `message` table (e.g. lines 36–40 in Listing 3), and then running an aggregation operation on this table (e.g. lines 16–20 in Listing 3). We can merge these two parts as follows:

```
SELECT edge.dest AS id, MIN(update.val+1) AS val
INTO cur
FROM update, edge
WHERE edge.src = update.id
GROUP BY (edge.dest)
```

4.7.3 Reduce the Update Cost

In the original query, we use tuple updates to mutate values. For algorithms that mutate all vertex values in each iteration (such as PageRank), the updates can be expensive. For such algorithms, Grail inserts new records into the table `cur` updating table `next`. After the insertions, we rename the table `cur` to `next` as follows:

```
INSERT INTO cur
SELECT *
FROM next
WHERE NOT EXISTS(
    SELECT * FROM cur
    WHERE cur.id = next.id
)
EXEC sp_rename "cur", "next"
```

4.8 Extensions

Extensions to the discussion above is needed to support some complex graph analytics algorithms, as described below.

4.8.1 UDT and UDAF

The expressiveness of Grail is dependent on the available data types and the aggregate functions. Thus, User-defined Data Types (UDTs) and User-Defined Aggregate Functions (UDAF) are needed for some graph analytics. For example, to compute the vertex value as $val = \prod_{m \in messages} m.val$, we need to define a corresponding UDAF.

UDAFs can also be generalized to consider multi-attribute aggregation. For example, the following two (Giraph-based) functions require multi-attribute aggregation functions on the attributes `msgVal` and `vertexVal`. (We note that these examples aim to illustrate the difference in the expressive power of Giraph and SQL with traditional UDAFs. However, we do not know of any real graph analytics that would require such multi-attribute UDAFs.)

```
foreach (int msgVal : messages) {
    mutateValue(getValue()/msgVal+1);
}
```

```
int sum = 0;
foreach (int msgVal : messages) {
  if (msgVal * getValue() > 0) {
    sum += msgVal * getValue();
  }
}
mutateValue(sum);
```

4.8.2 *Edge Mutations*

To avoid modifying the original vertex table, we create the table next as a copy, and use it for computation. If we want to allow for edge mutations, then we also need to make a copy of the edge table.

5. EXPERIMENT SETUP

In this section, we present results from an empirical evaluation.

5.1 System Setup

Each experiment was run on a single server. We use two servers with the same hardware configurations, one running Windows Server 2012 R2 (for SQL Server) and the other running Ubuntu 14.04 LTS with Linux kernel 3.13.0-24-generic (for GraphLab and Giraph). Each machine has 16 physical cores spread across two Intel(R) Xeon(R) CPU E5-2450L 1.80GHz processors. The machine has 96GB of main memory. We use Giraph v. 1.1.0, GraphLab v. 2.2, and SQL Server 2014. The Linux system runs Hadoop 1.2.1, jdk1.7.0_55. Giraph reads input from and writes output to HDFS, and Graphlab uses the local disk. SQL Server reads and writes directly from disk-based tables.

In the performance test, we set the max parallelism for SQL Server to 16. The log level is set to “Simple” to avoid log writes slowing down the query. For Giraph, we can start multiple workers and multiple threads for each worker. A worker is more expensive than a working thread as it has to keep additional data structures. However, multiple threads can lead to contention, and thus slow down the program. In the experiment, we use different settings for different datasets for Giraph, picking the optimal point for each (Giraph is quite brittle in this way in terms of finding its optimal setting). We also set parameters to allow Giraph to spill to disk if needed, and disable checkpointing. GraphLab has similar issues with parameters, and we report the optimal time that we observed below (the key parameter in GraphLab is the degree of parallelism). We note, that such tuning of Giraph and GraphLab for each task is likely to be a hinderance in practice.

5.2 Datasets

Table 5 shows the datasets that we use, obtained from [2, 3, 4, 5]. The datasets grow from small to “large.” We note that both Giraph and GraphLab insist on loading the dataset before running each program (they do not have a way to cache or pre-load the data). They also need a large amount of space to load the data as the raw data is prepared (i.e. internal data structures optimized) for the graph analysis. For example, for the 24GB Twitter dataset, GraphLab needs about 90GB of working memory to prepare the data. Giraph has a similar mode of operation.

Table 5: Datasets

Dataset	# Nodes	# Edges	Size
web-Google (GO)	9K	5M	71MB
com-Orkut (OR)	3.0M	117M	1.6GB
twitter-10 (TW)	41.6M	1.46B	24GB
uk-2007-05 (UK)	100M	3.3B	56GB

5.3 Queries

Our queries are the operations described in Section 3 . For SQL Server, we use the optimized query generated by Grail. For Giraph, we use their existing PageRank, SSSP and WCC implementations. For Graphlab, we run experiments under both synchronous mode and asynchronous mode and report both results. We use their existing SSSP and WCC implementations, and the fixed-number-iterations version of PageRank. All PageRank programs are terminated after 10 supersteps/iterations.

5.4 Experiment Results

The key results are presented in Figure 4 where there is one figure for each of the four datasets. For these experiment, we note that the GraphLab asynchronous mode is generally more expensive that the synchronous mode, due to the lock contention in the asynchronous mode. From this figure we also observe that sometimes the specialized engines are faster than the Grail-on-SQL Server approach – for example, with the GO dataset the GraphLab (synchronous) approach is 2.5X faster than the Grail approach for the SSSP analysis (though the Grail approach is 2.5X faster than Giraph on that same problem). As the dataset size increases, the Grail approach catches up; for example, compare the GO and the OR results. Furthermore, as the dataset size increases even further (to the TW and the UK datasets), the other systems start to fail. In fact, neither Giraph or GraphLab can run the UK dataset which is only 56GB in size – both these systems need far larger amount of space (for the data representation and intermediate data structures) than the original data, and they do not scale gracefully when there isn’t enough memory. The Grail approach is far more robust.

5.5 On Other Related Reported Experiments

In a recent experiment for Pregel-like systems by Han [13], they use 32 EC2 machines to run the algorithms that we use here. For the TW dataset, on 32 machines Giraph took 1-3 minutes to setup, and 10 minutes to run 30 iterations of PageRank. GraphLab took 3 minutes to setup and about 10 minutes to run. Taking the number of iterations into consideration, the cluster runs 10+ times faster than a single node SQL Server. But 32 machines are used in that experiment. If we take into consideration all the costs such as hardware amortization cost, power cost, administration cost, SQL Server may be a better choice for graph analysis at this scale. Thus, while a lot of attention has focused on scale-out behavior of specialized graph engines, and the fact that papers report cluster numbers when the datasets are smaller than what can fit in typical single node today, we note that systems like RDBMSs that can deal with out-of-memory scenarios well are far more versatile and likely far more cost-effective than methods that demand an always-in-memory approach.

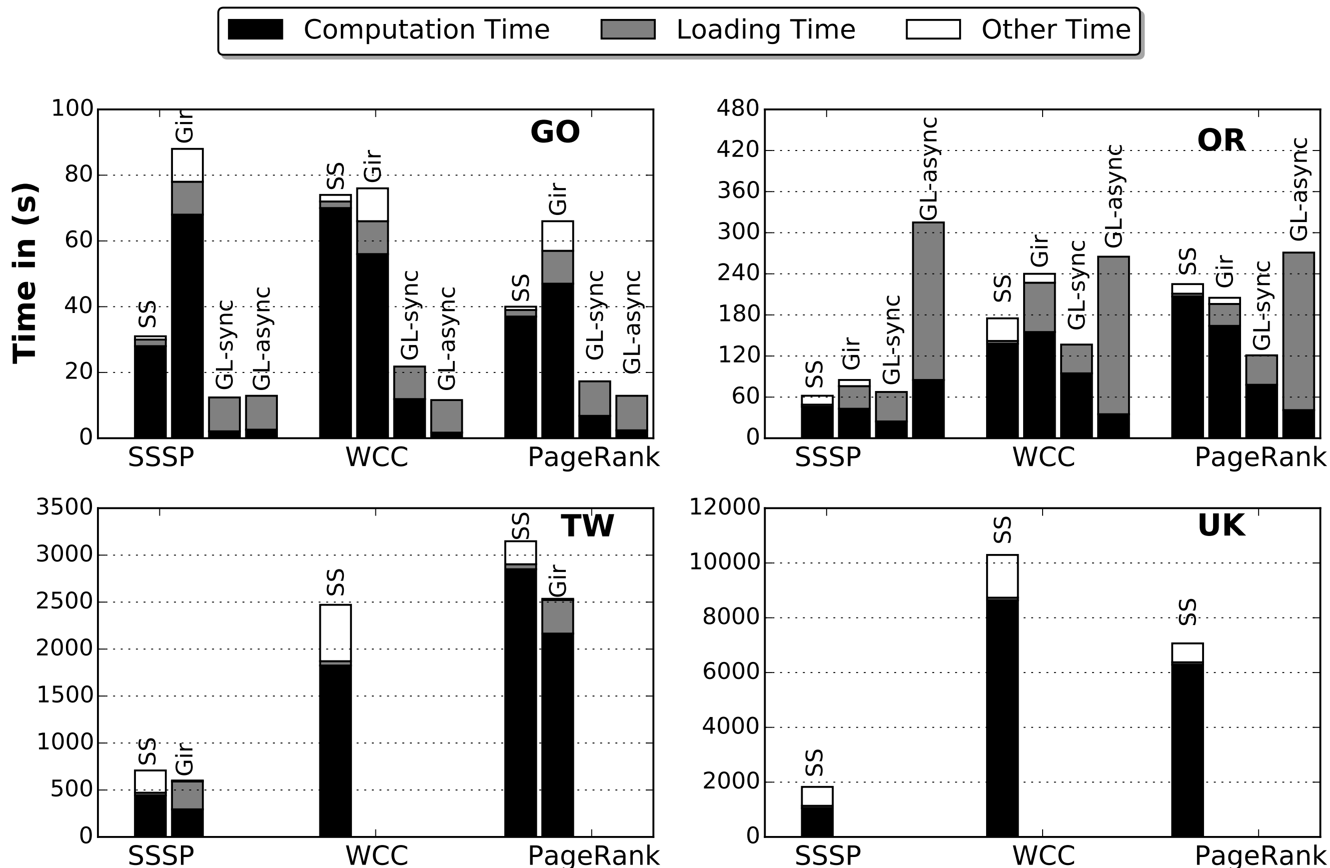


Figure 4: Comparison of the three graph engines, on the four datasets (GO, OR, TW and UK) with one figure for each dataset. SS stands for SQL Server, Gir for Giraph, GL-sync for GraphLab synchronous mode, and GL-async for GraphLab asynchronous mode. For each algorithm the time is broken down into the "Computation", "Loading," and "Other" times. Giraph and GraphLab need to load the data each time, where as SQL server can benefit from a warm buffer pool. The load times for SQL Server reported here are warm numbers, but the load time only goes up by about 10% if the buffer pool is cold. The "Other" component includes the cost of building indices (in SQL Server), the shutdown and the setup time (for Giraph), while this component does not exist for GraphLab. For the TW and the UK datasets, some bars are missing because the corresponding graph engines don't finish/crash on those tasks.

6. RELATED WORK

There has been a flurry or recent work on graph analysis using database systems. Han et al. [13] have experimentally compared Pregel-like graph processing systems and have shown that Giraph and GraphLab's synchronous mode execution has good all-round performance, but they do not consider an RDBMS-based approach.

Marc et al. [14] have studied different platforms for graph processing including a relational approach, but do not consider a graph programmer-friendly API. They also did not compare with popular engines like Giraph and GraphLab that have been created more recently.

Teradata Aster 6.0 (Aster 6) [19] introduced support for large-scale graph analytics. They have a specialized graph engine similar to Pregel [12]. Aster 6 provides a SQL-like interface where the graph analytic functions can be accessed and executed using SQL queries. Their graph analytics functions can also operate on relational tables, just as Grail does. Alekh et al. [10] also have an approach of exposing vertex-centric APIs on top of a relational DBMS. They expose the same APIs as Pregel [12], and sketch out at a high level how the APIs work. However, these previous works have not considered GraphLab. They have also not explored the parallels between the relational engines and specialized graph engines to draw out the similarities and differences in the execution models, and build from that analysis to make a case for the use of relational database engines for graph analytics. Collectively this body of work does points to increasing awareness and interest in using relational database management systems for graph analytics, instead of using specialized graph engines.

7. CONCLUSION

The use of RDBMS as a platform for graph analytics has largely been ignored, encouraging a flurry of specialized graph engines/platforms. We argue that this specialization does

not make sense for two reasons. First, the programming convenience offered by these systems can easily be mirrored using a syntactic layer on SQL and a code generation tool that converts the graph program to SQL. In fact, we present such a method called Grail in this paper. Second, the performance of the Grail-based RDBMS approach is comparable to the specialized engines, and the Grail approach allows handling of datasets that are large (e.g. don't fit entirely in memory), and brings production-quality systems that have been hardened over time making actual deployment far more manageable and cost-effective. Thus, the case for specialized graph analysis engines in most enterprises (that also have other analytical needs) is tenuous.

There are a number of directions for future work, including determining if there are ways to tune RDBMS engines to improve their performance for Grail queries (e.g. by creating special set-oriented operators that optimize the creation of messages for each iteration), exploring far larger datasets and scale-out behavior with these datasets, and considering applications holistically in an enterprise which typically need to run both graph analytic queries and other queries on the same data.

8. ACKNOWLEDGMENTS

This research was supported in part by a grant from the Microsoft Jim Gray Systems Lab (GSL), and by the National Science Foundation under grants IIS-0963993 and IIS-1250886.

9. REFERENCES

[1] Apache Giraph. http://giraph.apache.org/.

[2] Snap: Stanford network analysis project. http://snap.stanford.edu/.

[3] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711–726, 2004.

[4] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, pages 587–596, 2011.

[5] P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web (WWW)*, pages 595–601, 2003.

[6] N. Bruno. Teaching an old elephant new tricks. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[7] Y. Bu, V. R. Borkar, M. J. Carey, J. Rosen, N. Polyzotis, T. Condie, M. Weimer, and R. Ramakrishnan. Scaling datalog for machine learning on big data. *CoRR*, 2012.

[8] C. Chasseur, Y. Li, and J. M. Patel. Enabling JSON document stores in relational systems. In *Proceedings of the 16th International Workshop on the Web and Databases (WebDB)*, pages 1–6, 2013.

[9] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In C. Thekkath and A. Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–30, 2012.

[10] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. VERTEXICA: your relational friend for graph analytics! *Proceedings of the VLDB Endowment (PVLDB)*, 7(13):1669–1672, 2014.

[11] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new parallel framework for machine learning. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, July 2010.

[12] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, pages 135–146, 2010.

[13] H. Minyang, D. Khuzaima, A. Khaled, Ã. M. Tamer, W. Xingfang, and J. Tianqi. An experimental comparison of pregel-like graph processing systems. *Proceedings of the VLDB Endowment (PVLDB)*, 7(12):1047–1058, 2014.

[14] M. Najork, D. Fetterly, A. Halverson, K. Kenthapadi, and S. Gollapudi. Of hammers and nails: An empirical comparison of three paradigms for processing large graphs. In *5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[15] S. Salihoglu and J. Widom. Gps: A graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 22:1–22:12, 2013.

[16] J. Seo, J. Park, J. Shin, and M. S. Lam. Distributed socialite: A datalog-based language for large-scale graph analysis. *Proceedings of the VLDB Endowment (PVLDB)*, 6(14):1906–1917, 2013.

[17] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[18] A. Shkapsky, K. Zeng, and C. Zaniolo. Graph queries in a next-generation datalog system. *Proceedings of the VLDB Endowment (PVLDB)*, 6(12):1258–1261, 2013.

[19] D. E. Simmen, K. Schnaitter, J. Davis, Y. He, S. Lohariwala, A. Mysore, V. Shenoi, M. Tan, and Y. Xiao. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *Proceedings of the VLDB Endowment (PVLDB)*, 7(13):1405–1416, 2014.

[20] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.