Graph Analytics using Vertica Relational Database

Alekh Jindal
MIT CSAIL

Samuel Madden
MIT CSAIL

Malú Castellanos
Vertica, HP Software

Meichun Hsu
Vertica, HP Software

***Abstract*—Graph analytics is becoming increasingly popular, with a number of new applications and systems developed in the past few years. In this paper, we study Vertica relational database as a platform for graph analytics. We show that vertex-centric graph analysis can be translated to SQL queries, typically involving table scans and joins, and that modern column-oriented databases are very well suited to running such queries. Furthermore, we show how developers can trade memory footprint for significantly reduced I/O costs in Vertica. We present an experimental evaluation of the Vertica relational database system on a variety of graph analytics, including iterative analysis, a combination of graph and relational analyses, and more complex 1-hop neighborhood graph analytics, showing that it is competitive to two popular vertex-centric graph analytics systems, namely Giraph and GraphLab.**

I. Introduction

Recent years have seen growing interest in the area of graph data management. This focus on graphs arises from their use in a number of new applications, including social network analytics, transportation, ad and e-commerce recommendation systems, and web search. Given the popular demand for graph analytics, a natural question is whether or not traditional database systems really are a bad fit for these graph analytics workloads? This question arises because, in many real-world scenarios, graph data is collected and stored in a relational database in the first place and it is expensive to move data around. Given this, if it is avoidable, users may prefer not to export their data from the relational database into a graph database. Rather, they would like to perform the graph analytics (with comparable performance) directly with the relational engine, without the expensive step of copying data into a file system (or distributed storage system like HDFS), in order to be processed by a graph system, and then (possibly) back into the relational system for further storage and analyses.

Apart from the need to avoid copying data in and out of file systems, graph engines suffer from another limitation. As graphs get larger and larger, frequently the users want to (or will have to) select a subset of a graph before performing analysis on it. For example, it is unlikely that a user will run a single-source shortest path query on the entire trillion node Facebook graph — this would be prohibitively slow on any system. Rather, it is more likely that users will run several shortest paths queries over different subsets of the graph (e.g., the N-hop neighbors of some particular user.) Furthermore, real-world graphs have vertices and edges accompanied by several other attributes. For example, edges in a social network may be of different types such as friends, family, or classmates. Similarly nodes may have several attributes to describe the properties of each person in the social network, e.g., their username, birthdate, and so on. Given such metadata, an analyst would typically do some ad-hoc relational analysis in addition to the graph analysis. For instance, the analyst may want to preprocess and shape the graph before running the actual graph algorithm, e.g., filtering edges based on timestamp or limiting the graph to just close friends of a particular user. Similarly, he may want to analyze the output of the graph analysis, e.g., computing aggregates to count the number of edges or nodes satisfying some property, or other statistics. Such pre- and post- processing of graph data requires relational operators such as selection, projection, aggregation, and join, for which relational databases are highly optimized.

In addition to combining relational analysis, several graph analyses compute aggregates over a larger neighborhood. For example, counting the triangles in a graph requires every vertex to access its neighbors' neighbors (which could potentially form the three vertices of the triangle). Likewise, finding whether a vertex acts as a bridge (weak ties) between two disconnected vertices requires every vertex to check for the presence of edges between its neighbors, i.e. the 1-hop neighborhood. Vertex-centric interfaces like Pregel [1] are tedious for expressing the above queries, as they require sending neighborhood information to all its neighbors in the first superstep and then performing the actual analysis in the second superstep. SQL, on the other hand, is a much more powerful and general purpose language for capturing such analyses. For example, in Vertica, we can express triangle counting as a three-way self-join over the edge table very efficiently [2]. Similarly, we can detect weak ties using two inner joins (to get the two vertices on either side of the bridge) and one outer join (to make sure that the two vertices are not connected by an edge). Thus, by simply adding more joins, SQL is more flexible at expressing such graph analyses.

In this paper, we describe four key aspects necessary to build high-performance graph analytics in the Vertica column-oriented database. First, we look at how we can translate the logical query plans of vertex-centric graph queries into relational operators and run them as standard SQL. Although vertex compute functions can be rewritten into pure SQL in some cases, we find that table UDFs (offered by many relational databases, including Vertica) are sufficient to express arbitrarily complex vertex functions as well. Second, we show several query optimization techniques to tune the performance of graph queries on Vertica. These include considering updating vs replacing the nodes table on each iteration, incremental evaluation of queries, and eliminating redundant joins. Third, we outline several features specific to a column-store like Vertica that makes it well suited to run graph analytics queries. Finally, we show how Vertica can be optimized using table UDFs to run iterative graph analytics in-memory, which significantly reduces the disk I/Os (and overall query time) at the cost of higher memory footprint.

Contributions. Our key contributions are as follows:
(1.) We take a closer look at vertex-centric graph processing,

using Giraph (a popular graph analytics system) as an example. We show that vertex-centric graph processing can be expressed as a query execution plan, which in the case of Giraph is a fixed plan that is used to run all Giraph programs. We then show that this plan can be expressed as a logical query plan that can be optimized using a relational query optimizer (Section III).

(2.) We show how we can translate this vertex-centric plan into SQL, which can be run on standard relational databases. We describe several query optimizations to improve the performance of vertex-centric queries and describe Vertica specific features to run these queries efficiently. As a concrete example, we discuss the physical query execution plan of single source shortest path on Vertica. Lastly, we show how Vertica can be extended to run the entire unmodified vertex-centric query in-memory and as a single transaction (Section IV).

(3.) We provide an extensive experimental evaluation of several typical graph queries on Vertica. We compare it with two popular vertex-centric graph processing systems, GraphLab and Giraph. Our key findings are: (i) Vertica has comparable end-to-end performance to these popular vertex-centric systems, (ii) Vertica has a much smaller memory footprint than other systems, at the cost of higher disk I/O, (iii) We can extend Vertica to trade an increased memory footprint for faster runtimes, comparable to that of GraphLab, (iv) relational engines naturally excel at combining graph analysis with relational analysis, and (v) column-stores can efficiently implement more complex 1-hop neighborhood graph analyses (Section V).

II. Related Work

Existing graph data management systems address two classes of query workloads: (i) low latency online graph query processing, e.g. social network transactions, and (ii) offline graph analytics, e.g. PageRank computation. Typical examples of systems for online graph processing include RDF stores (such as Jena [3]), key value stores (such as HypergraphDB [4]), and native graph stores (such as Neo4j [5]). Other graph processing systems wrap around relational databases to provide efficient online query processing, e.g., TAO [6] and FlockDB [7], which wrap around MySQL to build a distributed and scalable graph processing system.

Graph analytics, on the other hand, is seen as completely different from traditional data analytics, typically due to the iterative nature and the perceived awkwardness of expressing graph analytics as SQL queries (which typically involves multiple self-joins on tables of nodes and edges). Examples queries include computing statistics and other metrics over graphs, such as PageRank or shortest paths. As a result, a plethora of graph processing systems have been recently proposed. These include vertex-centric systems, e.g. Pregel [1], Giraph [8], GraphLab [9] and its extensions [10], [11], [12], GPS [13], Trinity [14], GRACE [15], [16], Pregelix [17]; neighborhood-centric systems, e.g. Giraph++ [18], NScale [19], [20]; datalog-based systems, e.g. Socialite [21], [22], GrDB [23], [24]; SPARQL-based systems, e.g. G-SPARQL [25]; and matrix-oriented systems, e.g. Pegasus [26].

Recent works have also looked at specific graph analysis using relational databases. Examples include triangle counting [2], shortest paths [27], [28], subgraph pattern matching [29], [30], and social network analysis [31], [32]. Others have looked at the utility of combining graph analysis with relational operators [33]. Data flow systems such as GraphX [34], Naiad [35], and epiC [36] further support more general end-to-end data processing. Grail provides a layer for expressing vertex-centric queries and compiling them into SQL, and the authors run it on a single node row-store [37].

In this work, we have a particular emphasis on showing how vertex-centric graph processing can be translated and optimized on Vertica, and compare it to specialized graph systems in a distributed setting.

III. Background

In this section, we first recap the popular vertex-centric programming model. Then, to understand the graph processing in a typical vertex-centric system, we analyze the execution pipeline in Giraph, an open-source implementation of Pregel, and express it as a logical query plan. Other Pregel-like systems use a similar static query plan, though some may use different scheduling strategies, e.g. GraphLab.

Vertex-centric Model. In the vertex-centric programming model, the user provides a UDF (the *vertex program*) specifying the computation that happens at each vertex of the graph. The UDFs update the vertex state and communicate by sharing messages with neighboring vertices. For example, to implement single source shortest paths (SSSP), each vertex compares its current shortest distance to the distance reported by each of its neighbors, and if a shorter distance is found, updates its distance and propagates the updated distance to its neighbors. The underlying execution engine may choose to run the vertex-centric programs synchronously, as a series of *supersteps* with synchronization between them, or asynchronously, where threads update a representation of the graph in shared memory. Programmers do not have to worry about details such how the graph is partitioned across nodes/threads, how it is distributed across multiple machines, or how message passing and coordination works. Each vertex may be on the same physical machine or a different, remote machine.

Giraph Execution Pipeline. We now provide a detailed study of execution workflow used in Giraph. Giraph executes the vertex-centric query as a map-only job (the *GiraphMapper*) on Hadoop MapReduce, i.e., the mappers are simply the containers for Giraph workers. These map jobs run for the duration of the job, repeatedly executing the compute UDF and communicating with other mappers over sockets. To illustrate this data flow, Figure 1 shows the physical execution of Giraph with four workers W_1 to W_4. The execution in Giraph is organized into *supersteps*, wherein each worker operates in parallel during the superstep and the workers synchronize at the end of the superstep. During the *InputSuperstep*, the system splits the input graph into a list of vertices V and

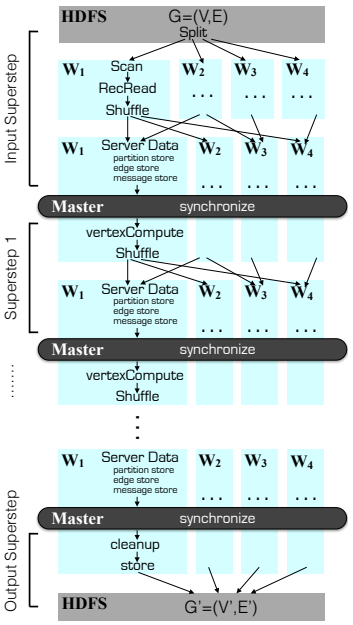


Fig. 1. Giraph Physical Plan.

list of edges E as it reads data from HDFS. Each worker reads the split assigned to it, parses it into vertices and edges, and partitions them across all workers, typically using a hash-based partitioner. Each worker then builds its *ServerData*, consisting of three components: (i) the *partition store* to keep the partition vertices and related metadata, (ii) the *edge store* to keep the partition edges and related metadata, and (iii) the *message store* to keep the incoming messages for this partition. At the end of the InputSuperstep, i.e. when all workers have finished creating the ServerData, the workers are ready to perform the actual vertex computation. In each superstep after the InputSuperstep, the workers run the *vertexCompute* UDF for the vertices in their respective partition and shuffle the outgoing messages across all workers. The workers then update their respective ServerData and wait for everyone to finish the superstep (the *synchronize* barrier). Finally, when there are no more messages to process, the workers store the output graph back in HDFS during the *OutputSuperstep*. Thus, we see that similar to MapReduce execution in Hadoop [38], Giraph has a static hard-coded query execution pipeline.

Giraph Logical Query Plan. The above Giraph physical execution pipeline can also be represented as a logical query plan, consisting of relational operators and the vertexCompute UDF. Figure 2 shows such a simplified logical query plan.

Assuming that the graph structure itself remains unchanged[1], the Giraph execution pipeline is essentially a distributed vertex update query. That is, it takes the set of vertices V (each having an id and a value), edges E (each having a source and a destination vertex id) and messages M (each having destination vertex id and the message value), runs the vertexCompute UDF for each vertex, and produces the set of output vertices (V') and messages (M').

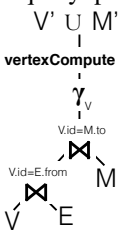


Fig. 2. Logical Plan.

The downside of the above vertex-centric query execution in Giraph is that all graph analysis is forced to fit into a fixed query plan. This is not desirable for several analyses, e.g., triangle counting, which requires a three-way join over the edges table. Moreover, the Giraph logical query plan is not really implemented as a composition of query operators, making it very difficult to modify, extend, or add functionality to the execution pipeline. For instance, the join with M is implemented as a sort merge join; changing to another join implementation would require several deep changes in the system. Furthermore, even if one could extend or modify the physical execution pipeline, e.g. switch merge join to hash join, Giraph cannot make dynamic decisions regarding the best physical plan. For example, hash join may be suitable for very large numbers of intermediate messages and merge join better for small numbers of messages. Giraph does not have this flexibility. Finally, Giraph is a custom built query processor restricted to a specific type of graph analysis. It cannot be used for broader types of queries, e.g. multi-hop analysis, or end-to-end graph analysis, e.g. analyzing the output of graph analysis, or combining multiple graph analyses.

In the rest of the paper, we show how relational databases can overcome many of these limitations and, in particular, how Vertica is highly suited for a variety of graph analytics.

[1]This is true for several typical vertex-centric graph analysis, such as PageRank, shortest paths, connected components, etc.

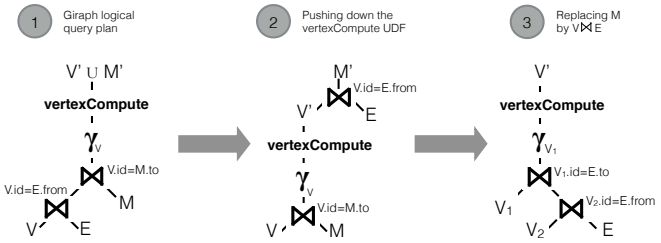


Fig. 3. Rewriting Giraph Logical Query Plan.

IV. Graph Analytics using Vertica

In this section, we describe how we can: (i) translate vertex-centric graph analyses to standard SQL queries, (ii) apply several query optimizations to improve the performance of graph analyses, and (iii) leverage key features of Vertica for efficiently executing these graph analytics queries.

A. Translation to SQL

In the following, we describe how we rewrite and translate the Giraph logical query plan to standard SQL.

1) Eliminating the message table: Consider again the Giraph logical query plan, shown on the left in Figure 3. The relation M in this plan is an intermediate output and an artifact of message passing, a consistency mechanism in Giraph. Since relational databases take care of consistency and allow us to operate directly on the relational tables, we can get rid of M. Note that the relation M is used to communicate the new values of vertices to its neighbors. Therefore, we can push down the vertex compute function and obtain the new messages M' by joining the new vertex values (V') with the outgoing edges (E), as shown in the middle of Figure 3. Finally, we can replace M with $V \bowtie E$ and get rid of relation M completely, as shown on the right in Figure 3. This simplified query plan deals only with relations V and E as the input and produces modified relation V' as output.

2) Translating the vertex compute functions: The *vertexCompute* in the Giraph logical query plan (Figure 2) can be an arbitrary user defined function, similar to *map*/*reduce* in the MapReduce framework. However, for many graph analytics, the vertex function involves relatively simple and well defined aggregate operations, which can be expressed directly in relational algebra/SQL. For example, the vertex function for SSSP finds the MIN of the neighboring distances and applies the filter for detecting smaller distances, i.e.:

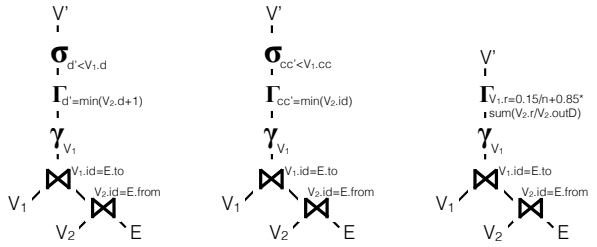
$$\text{SSSP} : \textit{vertexCompute} \longmapsto \sigma_{\text{d'}<V_1.d}(\Gamma_{\text{d'=min}(V_2.d+1)})$$

Figure 4(a) shows the resulting logical query plan. Similarly, the vertex function for connected components finds the minimum vertex ID amongst its neighbors and filters for new minimum found (Figures 4(b)), whereas the vertex function for PageRank combines the PageRank of its neighbors (Figure 4(c)), i.e.:

$$\text{CC} : \textit{vertexCompute} \longmapsto \sigma_{\text{cc'}<V_1.cc}(\Gamma_{\text{cc'=min}(V_2.id)})$$

$$\text{PageRank} : \textit{vertexCompute} \longmapsto \Gamma_{V_1.r=\frac{0.15}{n}+0.85*sum(\frac{V_2.r}{V_2.outD})}$$

By rewriting the vertex functions as relational expressions, the resulting query plans become purely relational and can be implemented completely in standard SQL, without using user-defined function features in the database system at all (we describe a more general implementation based on UDFs



(a) SSSP (b) CC (c) PageRank

Fig. 4. Logical query plans for three vertex-centric queries: (i) single source shortest path (SSSP), (ii) connect components (CC), and PageRank.

in Section IV-D below). For instance, we could write SSSP from Figure 4(a) as the following SQL expression:

```
SELECT v1.id, MIN(v2.d+1) AS d
    FROM vertex AS v1, edge AS e, vertex AS v2
    WHERE v2.id = e.from_node AND v1.id = e.to_node
    GROUP BY e.to_node, v1.d
    HAVING MIN(v2.d+1) < v1.d
```

The above SQL query computes the minimum neighboring distance of every vertex and filters, via the HAVING clause, distances that are smaller than the already known distance. The resulting vertices can then be used to update the `vertex` relation, as shown in Listing 1.

```
UPDATE vertex AS v SET v.d=v'.d
  FROM (
    SELECT v1.id, MIN(v2.d+1) AS d
      FROM vertex AS v1, edge AS e, vertex AS v2
      WHERE v2.id = e.from_node AND v1.id = e.to_node
      GROUP BY e.to_node, v1.d
      HAVING MIN(v2.d+1) < v1.d
    ) AS v'
    WHERE v.id=v'.id;
```

Listing 1. Shortest Path in SQL.

Finally, a driver program (run via a stored procedure) repeatedly runs the above shortest path query as long as at least one of the vertices finds a shorter distance.

B. Query Optimizations

The advantage of expressing graph analysis as relational queries is that we can apply several relational query optimizations, i.e. we have the flexibility to optimize the queries in several different ways in order to boost performance, in contrast to the hard-coded execution pipeline in Giraph. In the following, we present three such query optimizations that can be used to tune the performance. We use vertex-centric single source shortest paths as the running example. However, of course, these optimizations are applicable in general to SQL-based graph analytics.

1) Update Vs Replace: Graph queries often involve updating large portions of the graph over and over again. However, large number of updates can be a barrier to good performance, especially in read optimized systems like Vertica. To overcome this problem, we can instead replace the vertex or edge table with a new copy of tables containing the updated values. For instance, the single source shortest path involves updating all vertices that find a smaller distance in an iteration. As we explore the graph in parallel, the number of such vertices can quickly grow very large. Therefore, instead of updating the vertices in the existing vertex relation, we can create a new vertex relation (`vertex_prime`) by joining the updated vertices with the non-updated vertices:

```
CREATE TABLE vertex_prime AS
  SELECT v.id, ISNULL(v'.d, v.d) AS d
    FROM vertex AS v LEFT JOIN (
      SELECT v1.id AS id, MIN(v2.d+1) AS d
        FROM vertex AS v1, edge AS e, vertex AS v2
        WHERE v2.id=e.from_node AND v1.id=e.to_node
        GROUP BY e.to_node, v1.d
        HAVING MIN(v2.d+1) < v1.d
    ) AS v'
    ON v.id = v'.Id;
```

Listing 2. Shortest Path with Replace instead of Update.

Afterwards, we replace `vertex` with `vertex_prime`. This replacement is quite fast, and, in general, creating a new table is faster than updating because it allows new records to be written sequentially to the table, rather than performing random I/O to update-in-place or recording large delete lists in Vertica. One downside of this approach is that we lose the physical design (i.e., indexes) on the original table, and physical designs are expensive to create during query execution. However, many graph analyses, including PageRank and SSSP, update only the smaller vertex table and therefore the physical designs on the larger edge table can be preserved. Still, update-in-place may be more efficient for algorithms that perform small numbers of updates. For instance, the parallel graph exploration in single source shortest path updates very few vertices in the first few iterations. Therefore, a more sophisticated approach is to apply *updates* in the first few iterations before switching to *replace*. In this work, we experimentally determine a fixed threshold to switch from *updates* to *replace*. Eventually, of course, a cost-based optimizer could be use to determine when to switch.

2) Incremental Evaluation: Typically, iterative queries process different portions of the data in different iterations. As a result, there is an opportunity for incremental query evaluation. This is applicable to iterative graph queries as well. For example, in single source shortest path, we do not need to explore the entire graph in every iteration. We need to only explore the neighbors of vertices that found a smaller distance in the previous iteration. This introduces the overhead of keeping track of such vertices from previous iteration, but allows us to benefit by only joining the incrementally updated vertices table (`v_update`) with its neighbors. To achieve this, we initialize `v_update` with the `startNode` since that is the only vertex that updated its distance to 0. Thereafter, in each iteration, we get the new set of updated vertices (`v_update_prime`) from the existing set (`v_update`). Although we need to materialize additional intermediate output, we are able to exploit it to significantly reduce the join cardinalities by expanding only the neighbors of the updated vertices. We can then replace `v_update` by `v_update_prime` and get the updated set of vertices. Listing 3 shows the incrementally evaluated single source shortest path query. Note that Giraph actually employs a similar optimization as it only computes updates for active vertices in each superstep.

3) Join Elimination: Several graph analysis perform neighborhood access without reading the metadata associated with the neighboring vertices. This means that even though the logical query plan may have a join between the vertex and the edge table, we read only the vertex id from the vertex table. Thus, the join is redundant and can be elimi-

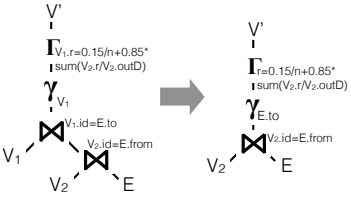


Fig. 5. Join Elimination in PageRank.

```
CREATE TABLE v_update_prime AS
  SELECT v1.id, MIN(v2.d+1) AS d
    FROM v_update AS v2, edge AS e, vertex AS v1
    WHERE v2.id=e.from_node AND v1.id=e.to_node
    GROUP BY e.to_node, v1.d
    HAVING MIN(v2.d+1) < v1.d;

DROP TABLE v_update;
ALTER TABLE v_update_prime RENAME TO v_update ;

CREATE TABLE vertex_prime AS
  SELECT v.id, ISNULL(v_update.d, v.d) AS value
    FROM vertex AS v LEFT JOIN v_update
    ON v.id = v_update.id ;

DROP TABLE vertex; ALTER TABLE vertex_prime RENAME TO vertex;
```

Listing 3. Shortest Path with Incremental Evaluation.

nated. For example, in the logical query plan for PageRank in Figure 4(c), we read only the vertex id from V_1. Therefore, the join with V_1 is redundant and can be eliminated as shown in Figure 5. Eliminating one of the joins in the above query will result in much better performance because the output of $V_2 \bowtie E$, which is as big as the number of edges itself, does not need to be re-partitioned again to perform the second join.

In summary, vertex-centric graph analyses can be translated to SQL, enabling several optimization techniques to tune the performance. In the following section, we look at the actual query execution of these graph queries and describe what makes Vertica a good choice for such analyses.

C. Query Execution

In the previous section, we saw that graph analyses typically involve full table scans and joins over the vertex and the edge tables. We now look at some of the features that makes Vertica well-suited for executing such queries. Specifically, we describe four key features that Vertica provides: (i) optimized physical database design, (ii) join optimizations, (iii) query pipelining, and (iv) intra-query parallelism. Thereafter, we walk through the query execution plan of single source shortest path in Vertica and contrast it with that of Giraph.

1) Physical Design: Vertica provides rich support for creating physical designs in order to boost query performance. For instance, it allows creating projections, sort orders, and segmentations within and across different nodes, as well as several encoding and compression schemes. See [39] for more details on physical design using Vertica. Although the columnar data representation is not useful for projections over narrow vertex and edge tables with just a few columns, it is useful for efficiently compressing these tables and saving disk I/O. As a result, we can create multiple table projections over these tables, in order to boost the performance of queries, while still not exceeding the raw data size. For instance, we can create two projections over the edge table, one segmented on `from_node` and the other on `to_node`, in order to perform different self-joins over the edge table locally. Likewise, we can sort the projections on different attributes for performing merge join instead of hash join as well as for evaluating selection predicates.

Thus, using Vertica, developers can efficiently encode and compress their graph data, create multiple sort orders and partitionings, and based on the physical design, leverage the query optimizer to automatically pick the best physical query operators for their analysis at run time.

2) Join Optimizations: Graph analyses when written in SQL make heavy use of joins. Vertica is highly optimized to efficiently execute such joins over large tables. For example, it can perform joins directly on compressed data without decoding it, apply type dependent just-in-time compilation of the join condition in order to avoid branching, and use sideways information passing (SIP) to push down the join condition as selection predicate over the outer input and thus filter tuples early on [40]. Furthermore, databases are not limited to a specific join implementation. Rather, the optimizer can choose between hash or merge joins, or even dynamically switch between the two.

Efficient join processing is a key feature that makes graph analysis possible in Vertica, by allowing developers to quickly traverse and manipulate large graphs via repeated self-joins.

3) Query Pipelining: Vertica supports pipelined query execution, which avoids materializing intermediate results that would otherwise require repeated access to disk. This is important because graph queries involve join operations that can have large intermediate results which can benefit dramatically from pipelining. For instance, in each iteration, the single source shortest path joins a vertex with its incoming edges and incoming nodes, thereby resulting in an intermediate result with cardinality equal to the number of edges. We can induce pipelining for such queries by creating sort orders on join and group by attributes. Additionally, we can express graph operations as nested queries, allowing the query optimizer to employ pipelining between the inner and outer query when possible. This is in contrast to Giraph, which blocks the execution and materializes all intermediate output before running the vertex compute function.

Thus, pipelining allows Vertica to avoid materializing large intermediate outputs, which are typical in graph queries. This reduces memory footprint and improves performance.

4) Intra-query Parallelism: Vertica includes capabilities that allow it employ multiple cores to process a single query. To allow Vertica to explore graphs in parallel as much as possible, we rewrite graph exploration queries that involve a self-join on the edges table by adding a GROUP BY clause on the edge id, and let Vertica partition the groups across CPU cores to process subgraphs in parallel. Though such parallel graph exploration ends up doing more work in each iteration, it still reduces the number of joins and results in much better performance.

5) Example SSSP Query Execution on Vertica: We now look at the specific example of physical query execution plan for single source shortest path (SSSP) on Vertica. Figure 6 shows the plan for running SSSP over the Twitter graph (41 million nodes, 1.4 billion edges). The query involves two joins, one hash and the other sort merge. To perform the hash join, the system broadcasts the smaller node relation (shown in the middle). While scanning the large edge relation, the query applies two SIP filters, one for the hash join condition and other for the merge join condition, in order to filter unnecessary tuples at the scan step itself. The hash join blocks on the smaller node relation. However, once the hash table is built, the remainder of the query is fully pipelined, including the merge join, the group by, and writing the final output. This is possible because the merge join and the group by are on the

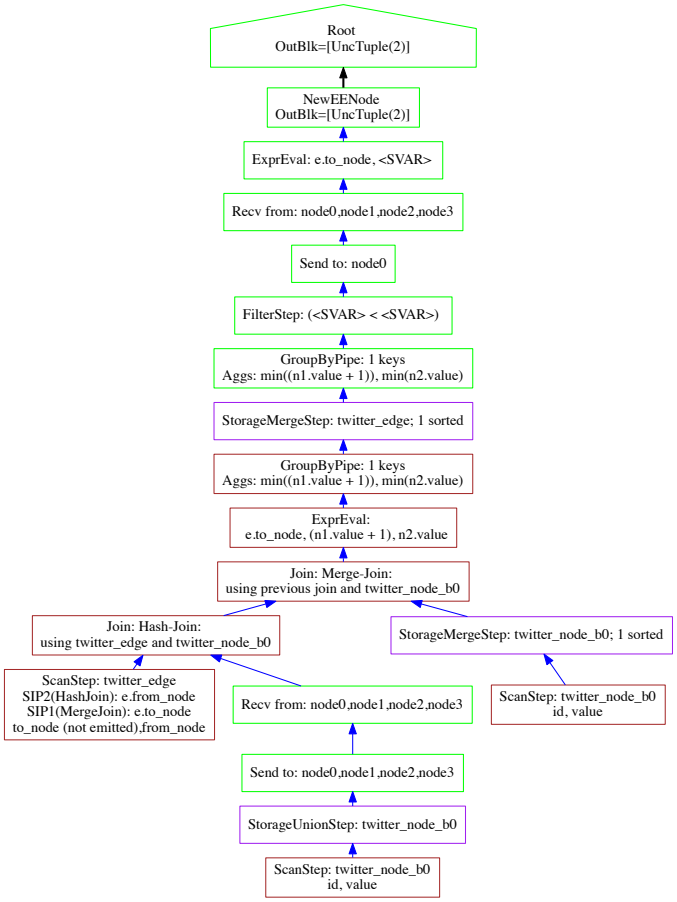


Fig. 6. Query execution plan of shortest paths in Vertica.

same key. As a result, data does not need to be re-segmented for the group by and the system performs 1-pass aggregation locally on each machine.

The above query execution plan is different from the Giraph query execution pipeline of Figure 1 in three ways: (i) it filters the unnecessary tuples from the large edge table as early as possible by using sideways information passing, (ii) it fully pipelines the query execution as opposed to blocking the data flow at the vertex function in Giraph, and (iii) it picks the best join execution strategies and broadcasts the data wherever required as compared to the static hard-coded join implementation in Giraph. As a result, Vertica is able to produce better execution strategies for such graph queries.

D. Extending Vertica

Relational database are extensible by design via the use of UDFs. We see how we can extend Vertica to address two issues: (i) how to run unmodified vertex programs without translating to SQL, and (ii) avoiding the expensive intermediate disk I/O in iterative graph queries.

1) Running Unmodified Vertex Programs: We saw in Section IV-A that several common vertex functions can be rewritten as relational operators. However, certain algorithms, such as collaborative filtering, have more sophisticated vertex function implementations which cannot easily be mapped to SQL operators. We can, however, still run vertex functions as table UDFs in Vertica without translating to relational operators. The middle of Figure 7 shows such a logical query plan. We first

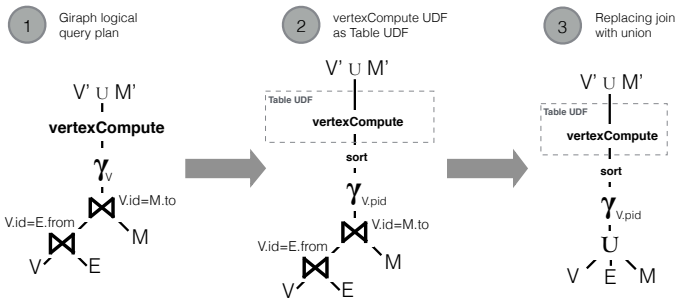


Fig. 7. Rewriting Giraph Query Plan using Table UDFs.

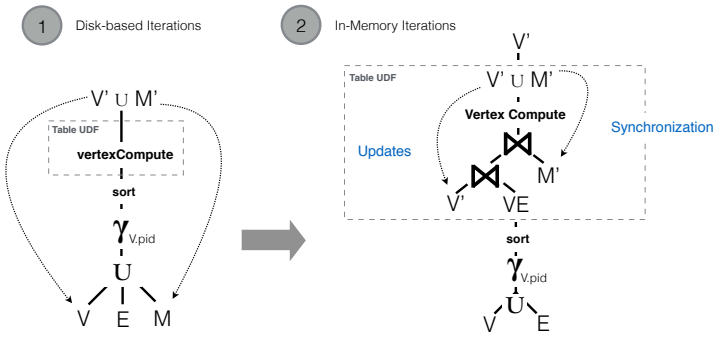


Fig. 8. In-memory Vertex-centric Query Execution in Vertica.

partition the vertices, then sort the vertices in each partition, and finally invoke the table UDF for each partition. The table UDF iterates over each vertex, invokes the *vertexCompute* function over it, and outputs the union of updated vertices (V') and outgoing messages (M'). As an optimization, by batching several vertices in each table UDF, we can significantly reduce the UDF overhead in relational databases. This query plan can be further improved by replacing the table joins with unions, as shown in the right of Figure 7. The table UDF is then responsible for segregating the tuples from different tables before calling the vertex function.

2) Avoiding Intermediate Disk I/Os: Iterative queries generate a significant amount of intermediate data in each iteration. Since relational databases run iterative queries via an external driver program, the output of each iteration is spilled to disk, thereby resulting in substantial additional I/O. This I/O also happens when running vertex functions as table UDFs. However, we can implement a special table UDF in which the UDF instances load the entire graph at the beginning and store the graph in shared memory, without writing the output of each iteration to disk (emulating the Giraph-like map-only behavior using table UDFs in Vertica). The right side of Figure 8 shows such a query plan. Of course this approach has less I/O at the cost of a higher memory footprint. And since the entire graph analysis runs as a single transaction, many of the database overheads such as locking, logging, and buffer lookups are further reduced. However, the UDF is now responsible for materializing and updating $V \bowtie E$, as well as propagating the messages from one iteration to the other (the synchronization barrier). Still, once implemented[2], the shared memory extension allows users to run unmodified vertex programs (or those which are difficult to translate to SQL). It can also yield a significant speed-up (up to 3 times) over even native SQL variants (as shown in our experiments).

[2]Our current implementation runs on multiple cores on a single node. Future work will look at distributing it across several nodes.

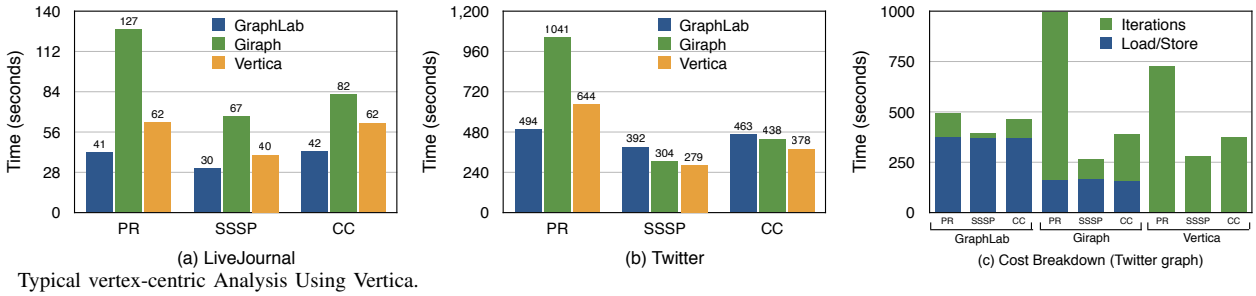


Fig. 9. Typical vertex-centric Analysis Using Vertica.

V. Experiments

In this section, we analyze and benchmark the performance of Vertica (version 6.1.2) on graph analytics, and compare it to dedicated graph-processing systems (Giraph and GraphLab). We organize our experiments as follows. First, we look at the performance of typical graph queries over large billion-edge graphs. Second, we dig deeper and look at the memory footprint and disk I/Os incurred and analyze the differences. Third, we evaluate our in-memory table UDF implementation of vertex-centric programs over different graph analyses. Fourth, we study end-to-end graph processing, comprising of graph algorithms combined with relational operators to sub-select and project portions of the graph prior to running analytics, and perform aggregations and joins after the graph analysis completes. Finally, we look at more complex graph analytics beyond vertex-centric analysis, namely 1-hop analysis, and evaluate the utility of Vertica on these operations.

Setup. Our test bed consists of a cluster of 4 machines, each having 12 (6x2) 2GHz Xeon cores, running 24-threads with hyper-threading, 48GB of memory, 1.4T disk, running on RHEL Santiago 6.4. We ran all experiments with cold cache and report the average of three runs. We compare Vertica with two popular vertex-centric graph processing engines, Giraph (version 1.0.0 running on Hadoop 1.0.4 with 4 workers per-node) and GraphLab (version 2.2 running on 4 nodes via MPI and using all available threads). We ran our benchmarks on a variety of datasets of varying sizes, including both directed as well as undirected graphs. Table I shows the different datasets used in our evaluation. All datasets are publicly available at *http://snap.stanford.edu/data*.

Type	Name	Nodes	Edges
Directed	Twitter-small	81,306	1,768,149
	LiveJournal	4,847,571	68,993,773
	Twitter	41,652,230	1,468,365,182
Undirected	YouTube	1,134,890	2,987,624
	LiveJournal-undir	3,997,962	34,681,189

TABLE I. The datasets used in the evaluation.

Data Preparation. The queries in our experiments read data from an underlying data store before running the analysis. While Vertica reads data from its internal data store, Giraph and GraphLab read the data from HDFS. All datasets are stored as a list of nodes and a list of edges. For GraphLab, we further split the data files into 4 parts, such that each node can load (ingress=grid) the graph in parallel during analysis. Table II summarizes the data preparation costs for the three systems. We can see that Giraph and GraphLab simply copy the raw files to HDFS and load faster than Vertica. However, Vertica has significantly less disk usage, due to compression and encoding, compared to Giraph and GraphLab.

Metric	Dataset	Vertica	GraphLab	Giraph
Upload Time (sec)	LiveJournal	45.927	15.621	12.049
	Twitter	916.421	472.358	267.799
Disk Usage (GB)	LiveJournal	0.423	3.030	3.030
	Twitter	9.964	73.140	73.140

TABLE II. Data preparation over two datasets.

A. Typical Vertex-centric Analysis

We first look at the performance of three typical graph queries, namely PageRank (PR), SSSP, and connected components (CC), on Vertica and compare it with Giraph and GraphLab. Then, we break the total query time into the time to load/store from disk and the actual graph analysis time. We used the built-in PageRank, SSSP, and connected component algorithms for Giraph (provided as example algorithms) and GraphLab (provided in the graph analytics toolkit). For Vertica, we implemented these three algorithms as described below:

PageRank. We implemented PageRank query shown in Figure 5 as a combination of two SQL statements on the Vertex (V) and Edge (E) tables: (i) to compute the outbound PageRank contributed by every vertex:

```
CREATE TABLE V_outbound AS
  SELECT  id, value/Count(to_id) AS new_value
    FROM E,V
    WHERE E.from_id=V.id
    GROUP BY id,value;
```

and (ii) to compute the total PageRank of a vertex as a sum of incoming PageRanks:

```
CREATE TABLE V_prime AS
  SELECT to_node AS id, 0.15/N+0.85*SUM(new_value) AS value
    FROM E,V_outbound
    WHERE E.from_id=V_outbound.id
    GROUP BY to_id;
```

After each iteration, we replace the old vertex table V with V_prime, i.e. drop V and rename V_prime to V.

Single Source Shortest Path (SSSP). We implemented SSSP on Vertica using the query shown in Listing 3, i.e. we incrementally compute the distances and replace the old vertex table with the new one, unless the number of updates is less than 5000, in which case we update the vertex table in-place.

Connected Components. We implemented the HCC algorithm [26] in Vertica (this is the same algorithm used in Giraph). HCC assigns each node to a component identifier. We initialize the vertex values with their ids and in each iteration the vertex updates are computed (similar to SSSP) as follows:

```
CREATE TABLE v_update_prime AS
  SELECT v1.id, MIN(v2.value) AS value
    FROM v_update AS v2, edge AS e, vertex AS v1
    WHERE v2.id=e.from_node AND v1.id=e.to_node
    GROUP BY v1.id, v1.value
    HAVING MIN(v2.value) < v1.value;
```

As in shortest paths, we apply these updates either in-place or by replacing the vertex table, depending upon the number of updates. Additionally, we apply two optimizations: (1) we do not perform incremental computation at first; rather, we update all vertices in the first few iterations, i.e. we use the entire vertex table instead of v_update in the above query, (2) since the component ids can be propagated in either edge direction, we propagate them in opposite directions over alternate iterations, i.e. we change the join condition in the above query to: `v2.id=e.to_node AND v1.id=e.from_node`. These optimizations help to significantly speed up the convergence of the algorithm, since component ids can be propagated quickly.

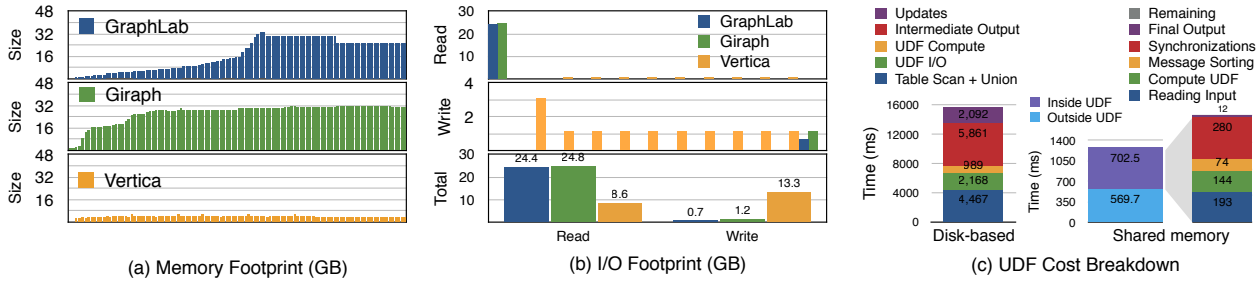


Fig. 10. The Resource Consumption of Different Systems.

Figures 9(a) and 9(b) show the query runtime of PageRank (10 iterations), SSSP, and Connected Components on GraphLab, Giraph, and Vertica over directed graphs using 4 machines. We can see that Vertica outperforms Giraph and is comparable to GraphLab on these three queries, both on the smaller LiveJournal graph as well as the billion edge Twitter graph. The reason is that these queries are full scan-oriented join queries for which Vertica is heavily optimized.

Finally, Figures 9(c) shows the runtime breakdown of the GraphLab, Giraph, and Vertica on the Twitter graph. We can see that the total runtime of GraphLab is dominated by the load/store costs. In fact, the analysis time for SSSP in GraphLab is just 25 seconds, a very small fraction of the overall cost. In contrast, the runtimes of Giraph are dominated by the analysis time. Vertica, on the other hand, pipelines the data from disk and therefore there is no distinct load/store phase. However, efficient data storage and retrieval makes Vertica competitive in terms of the total runtime.

B. Resource Consumption

We now study the resource consumption, i.e., the memory footprint and disk I/O, of Vertica, Giraph, and GraphLab. Figure 10(a) shows the per-node memory consumption of GraphLab, Giraph, and Vertica when running PageRank on the Twitter graph using 4 machines. We can see that out of the total 48GB memory per node, both GraphLab and Giraph has a peak memory usage of close to 32GB, i.e., 66% of the total memory. In contrast, Vertica has a peak usage of only 5.2GB, i.e. 11% of total memory. Thus, Vertica has a much smaller memory footprint.

The picture changes completely if we look at the disk I/O. Figure 10(b) shows the number of bytes read and written to disk by GraphLab, Giraph, and Vertica in each PageRank iteration over the Twitter graph. We can see that GraphLab and Giraph have high read I/O in the input step and no disk reads thereafter, whereas Vertica has no upfront read but incurs disk reads in each iteration. Likewise, Vertica incurs high write I/O in each iteration, whereas GraphLab and Giraph incur writes only in the output step. Thus, in total, while Vertica incurs less read I/O than Giraph and GraphLab (due to better encoding and compression), it incurs much more write I/O (due to materializing the output of each iteration to disk).

This expensive I/O for writing the iteration output is also true when running vertex functions as table UDFs. To illustrate this, Figure 10(c) shows the cost breakdown when running PageRank as table UDFs. We can see that writing intermediate output (shown in red) is the major cost in the query runtime. To test whether we can avoid the expensive intermediate I/O in Vertica, we implemented the shared memory UDFs as described in Section IV-D2. The right side of Figure 10(c) shows that the intermediate IO is indeed reduced significantly and developers can sacrifice memory footprint for better I/O performance in Vertica.

C. In-memory Graph Analysis

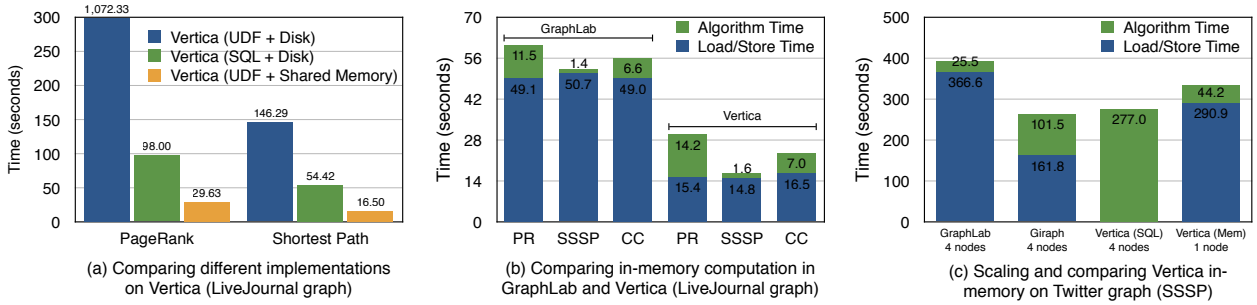
Figure 11(a) shows the runtime of the shared memory table UDF, on the Twitter-small graph. We can see that the shared memory implementation is almost 9 times faster than the disk-based table UDF and even more than 3 times faster than the SQL implementation. Figure 11(b) shows the shared memory Vertica extension on the LiveJournal graph and compares it with GraphLab on a single node. We can see that the actual graph analysis time (Algorithm Time) of GraphLab and Vertica are very similar now. However, GraphLab still suffers from the expensive loading time while Vertica benefits from more efficient data storage. As a result, the performance gap between GraphLab and Vertica widens. Finally, we scale SSSP to the billion edge Twitter graph on the shared memory Vertica extension (single node), as shown in Figure 11(c). The single node algorithm runtime in this case is 44.2 seconds, which is just 1.7 times that of GraphLab on 4 nodes[3]. Thus, we see that we can extend Vertica to exhibit similar performance characteristics as main-memory graph engines.

D. Mixed Graph & Relational Analyses

Finally, we consider situations when users want to combine graph analysis with relational analysis. We extended the graph datasets in our experiment with the following metadata. For each node, we added 24 uniformly distributed integer attributes with cardinality varying from 2 to 10^9, 8 skewed (zipfian) integer attributes with varying skewness, 18 floating point attributes with varying value ranges, and 10 string attributes with varying size and cardinality. For each edge, we added three additional attributes: the weight, the creation timestamp, and an edge type (friend, family, or classmate), chosen uniformly at random. These attributes are meant to model additional relational metadata that would be associated with properties of users in a social media context. The total size of the Twitter graph with this metadata is 66 GB.

We consider three end-to-end graph analysis: *(i) Sub-graph Projection & Selection* – extract subgraph by projecting just the node ids and selecting nodes with weight 4 and connected by edges of type 'Family', before running PageRank and SSSP. *(ii) Graph Analysis Aggregation* – gather distributions (equi-width histograms) of nodes importance and distance values after running PageRank and SSSP respectively. *(iii) Graph Joins* – combine the output of PageRank and SSSP to emit

[3]Single node GraphLab runs out of memory on the Twitter graph.



(a) Comparing different implementations on Vertica (LiveJournal graph)

(b) Comparing in-memory computation in GraphLab and Vertica (LiveJournal graph)

(c) Scaling and comparing Vertica in-memory on Twitter graph (SSSP)

Fig. 11. Improving I/O Performance in Vertica with In-memory Graph Analysis.

Query	Dataset	Vertica	Giraph	SpeedUp
Sub-graph Projection & Selection	PR	55.6	954.6	17.2
	SSSP	101.3	405.5	4.0
Graph Analysis Aggregation	PR	643.9	1089.7	1.7
	SSSP	279.8	349.9	1.3
Graph Joins	PR+SSSP	927.0	1435.9	1.5

TABLE III. COMBINING GRAPH AND RELATIONAL ANALYSIS.

those nodes which are either very near (path distance less than a given threshold) or are relatively very important (PageRank greater than a given threshold). We compare against Giraph, which allows users to provide custom input/output formats that could be used to perform the projection and selection. We write additional MapReduce jobs for the aggregation and join. Table III shows the result on the Twitter dataset over 4 nodes.

We can see that the performance difference between Vertica and Giraph is much higher now. For example, Vertica is 17 times faster on PageRank and 4 times faster on SSSP, as compared to 1.6 and 1.08 earlier, when combining these analysis with sub-graph selection. Performance on Vertica could be further improved by creating sort orders on the selection attributes. These massive performance differences are because Vertica is highly optimized to perform selections and exploit late materialization to access the remaining attributes of only the qualifying records. In contrast, Giraph incurs a complete sequential scan of data and cannot exploit the fact that part of the analysis is relational in nature. Rather, developers are required to stitch Giraph programs with programs in another data processing engine such as Hadoop MapReduce or Spark. Similar results are obtained on aggregation and join queries.

E. Beyond Vertex-centric Analyses

Let us now look at more advanced graph analytics. Consider the following two 1-hop neighborhood graph queries.

(i) Strong Overlap. Find all pairs of nodes having a large number of common neighbors, between them (overlap). We could also extend the algorithm to include other definitions of overlap. Such an analysis to find the strongly overlapping nodes in a graph could be useful in detecting similar entities. To implement such a query using SQL we do a self-join on the edge table followed by a group by on the two leaf nodes and a count of the number of common neighbors between them. Finally, all those pairs of nodes which have less than the *threshold* number of common neighbors are filtered.

```
SELECT e1.from_node as n1,e2.from_node as n2, count(*)
 FROM edge e1
 JOIN edge e2 ON e1.to_node=e2.to_node
  AND e1.from_node<e2.from_node
 GROUP BY e1.from_node,e2.from_node
 HAVING count(*) > THRESHOLD
```

(ii) Weak Ties. Find all nodes that act as a bridge (*weak ties*) between a threshold number of otherwise disconnected node-pairs. This is a slightly more complicated query because we need to test for disconnection between node pairs. Using SQL, this could be implemented as a three-way join, with the second join being a left join, and counting all cases when the third edge does not exist.

Query	Dataset	Vertica	Giraph
Strong Overlap	Youtube	259.56	230.01
	LiveJournal-undir	381.05	out of memory
Weak Ties	Youtube	746.14	out of memory
	LiveJournal-undir	1,475.99	out of memory

TABLE IV. 1-HOP NEIGHBORHOOD ANALYSIS.

```
SELECT e1.to_node AS Id,
 sum(CASE WHEN e3.to_node IS NULL THEN 1 ELSE 0 END)/2 AS C
 FROM edge e1
 JOIN edge e2 ON e1.to_node=e2.from_node
  AND e1.from_node<>e2.to_node
 LEFT JOIN edge e3 ON e2.to_node=e3.from_node
  AND e1.from_node=e3.to_node
 GROUP BY e1.to_node
 HAVING C > THRESHOLD
```

While the above queries are straightforward to implement and run on Vertica, they are tedious to implement in vertex-centric graph processing systems. For instance, in Giraph, each vertex needs to broadcast its neighborhood in the first superstep in order to access the 1-hop neighborhood. Thereafter, we perform the actual analysis in the subsequent supersteps. Unfortunately, this results in replicating the graph several times in memory. As an optimization in Giraph, we can reduce the number of messages by sending messages only to those neighbors with a higher node id. In addition, we can reduce the size of messages by sending only the ids which are smaller than the id of the receiving vertex. Still, passing neighborhoods as messages is not natural in the vertex-centric model and incurs the overhead of serializing and deserializing the nodes ids.

Table IV shows the performance of Vertica and Giraph over the two 1-hop neighborhood analyses running on 4 nodes. We can see that Giraph runs out of memory when scaling to larger graphs (even after allocating 12GB to each of the 4 workers on each of the 4 nodes). This is because the graphs are quite dense and sending the entire 1-hop neighborhood results in memory usage proportion to the graph size times the average out-degree of a node. Vertica, on the other hand, does not suffer from such issues and works for larger graphs as well.

VI. LESSONS LEARNED & CONCLUSION

This paper demonstrates that efficient and scalable graph analytics is possible within Vertica relational database system. We implemented a variety of graph analyses on Vertica as well as two popular vertex-centric graph analytics system. Our results show that Vertica has comparable end-to-end runtime performance, without requiring the use of a purpose-built graph engine. In summary, there are three key takeaways from our analysis and evaluation in this paper.

First, graph analytics can be expressed and tuned in relational databases. In particular, Vertica can be effectively tuned to offer good performance on graph queries, yielding performance that is competitive with dedicated graph stores. This is because these queries typically involves full scans, joins, and aggregates over large tables, for which Vertica is heavily optimized. These optimizations include efficient physical data representation (columnar layout, compression, sorting, segmentation), pipelined query execution, and an optimizer to automatically pick the best physical plan. We showed the effectiveness of graph analytics on Vertica both with declarative SQL queries, for expert users, as well as with procedural vertex functions, for non-expert users.

Second, developers can trade higher memory footprint to significantly reduced the I/O costs in iterative graph analytics on Vertica. This is done by loading the relevant graph into a shared memory and running the entire graph analysis as a single transaction, essentially emulating specialized graph processing systems. We implemented this shared memory graph processing on Vertica using the table UDFs and run on multiple cores in parallel. Such an implementation allows developers to load and extract graph (from relational tables) only once, avoid many of the database overheads such as locking, logging, and buffer lookups in each iteration, and achieve main-memory runtimes very close to those of specialized engines.

Third, relational databases, and column-stores in particular, provide the advantage of efficiently combining graph analysis with relational analysis. This is important because graph analysis is typically accompanied by relational analysis (selection, projection, aggregation, join), either as a preparatory step or as a final reporting step. Column stores such as Vertica provide several features, including efficient layouts, sort orders, and statistics, to perform these operations efficiently. Likewise, advanced 1-hop graph analytics are difficult to program and inefficient to run in vertex-centric graph systems, whereas they are easy to run via efficient self-joins (to gather the neighborhood) in column-stores.

Finally, relational databases come with many features such as update handling, transactions, checkpointing and recovery, integrity constraints, type checking, ACID etc., that are not present (yet) in the next generation of graph processing systems. One might think of stitching multiple systems together and coordinating between them to achieve these features, e.g. using HBase for transactions and Giraph for analytics, but then we have the additional overhead of stitching these systems together. Of course, with time, graph-engines may evolve these features, but our results suggest that these graph systems should be layered on RDBMS, not built outside of them, which would enable them to inherit these features for free without giving up performance.

REFERENCES

[1] G. Malewicz *et al.*, "Pregel: A System for Large-Scale Graph Processing," *SIGMOD*, 2010.

[2] "Counting Triangles with Vertica," vertica.com/2011/09/21/counting-triangles.

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

[4] "HyperGraphDB," http://www.hypergraphdb.org.

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

[6] N. Bronson *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," *USENIX ATC*, 2013.

[7] "FlockDB," http://github.com/twitter/flockdb.

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

[9] Y. Low *et al.*, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud," *PVLDB*, 2012.

[10] J. E. Gonzalez *et al.*, "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs," *OSDI*, 2012.

[11] A. Kyrola *et al.*, "GraphChi: Large-Scale Graph Computation on Just a PC," *OSDI*, 2012.

[12] C. Xie *et al.*, "SYNC or ASYNC: Time to Fuse for Distributed Graph-parallel Computation," *PPoPP*, 2014.

[13] S. Salihoglu *et al.*, "GPS: A Graph Processing System," *SSDBM*, 2013.

[14] B. Shao *et al.*, "Trinity: A Distributed Graph Engine on a Memory Cloud," *SIGMOD*, 2013.

[15] G. Wang *et al.*, "Asynchronous Large-Scale Graph Processing Made Easy," *CIDR*, 2013.

[16] W. Xie *et al.*, "Fast Iterative Graph Computation with Block Updates," *PVLDB*, vol. 6, no. 14, pp. 2014–2025, 2013.

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

[18] Y. Tian *et al.*, "From "Think Like a Vertex" to "Think Like a Graph"," *PVLDB*, vol. 7, no. 3, pp. 193–204, 2013.

[19] A. Quamar *et al.*, "NScale: Neighborhood-centric Large-Scale Graph Analytics in the Cloud," *VLDB Journal (to appear)*, 2015.

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

[21] M. S. Lam *et al.*, "SociaLite: Datalog Extensions for Efficient Social Network Analysis," *ICDE*, 2013.

[22] J. Seo *et al.*, "Distributed SociaLite: A Datalog-Based Language for Large-Scale Graph Analysis," *VLDB*, 2013.

[23] W. E. Moustafa *et al.*, "A System for Declarative and Interactive Analysis of Noisy Information Networks," *SIGMOD*, 2013.

[24] ——, "Declarative Analysis of Noisy Information Networks," *GDM*, 2011.

[25] S. Sakr *et al.*, "G-SPARQL: A Hybrid Engine for Querying Large Attributed Graphs," *CIKM*, 2012.

[26] U. Kang *et al.*, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," *ICDM*, 2009.

[27] J. Gao *et al.*, "Relational Approach for Shortest Path Discovery over Large Graphs," *PVLDB*, vol. 5, no. 4, 2011.

[28] A. Welc *et al.*, "Graph Analysis Ð Do We Have to Reinvent the Wheel?" *GRADES*, 2013.

[29] J. Huang *et al.*, "Query Optimization of Distributed Pattern Matching," *ICDE*, 2014.

[30] M. Rudolf *et al.*, "SynopSys: Large Graph Analytics in the SAP HANA Database Through Summarization," *GRADES*, 2013.

[31] S. Lawande *et al.*, "Scalable Social Graph Analytics Using the Vertica Analytic Platform," *BIRTE*, 2011.

[32] R. Angles *et al.*, "Benchmarking Database Systems for Social Network Applications," *GRADES*, 2013.

[33] A. Jindal *et al.*, "Vertexica: Your Relational Friend for Graph Analytics!" *VLDB*, 2014.

[34] J. Gonzalez *et al.*, "GraphX: Graph Processing in a Distributed Dataflow Framework," *OSDI*, 2014.

[35] D. G. Murray *et al.*, "Naiad: A Timely Dataflow System," *SOSP*, 2013.

[36] D. Jiang *et al.*, "epiC: an Extensible and Scalable System for Processing Big Data," *VLDB*, 2014.

[37] J. Fan *et al.*, "The case against specialized graph analytics engines," *CIDR*, 2015.

[38] J. Dittrich and al., "Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)," *PVLDB*, vol. 3, no. 1, 2010.

[39] "Physical Design Automation in Vertica," www.vertica.com /2014/07/21/physical-design-automation-in-the-hp-vertica-analytic-database.

[40] A. Lamb *et al.*, "The Vertica Analytic Database: C-store 7 Years Later," *PVLDB*, vol. 5, no. 12, 2012.