Hewlett Packard Labs

VGL: Enabling Graph Computation over Enterprise Data

Alexander Kalinin, Alkis Simitsis, Kevin Wilkinson, Mahashweta Das

Hewlett Packard Labs
HPE-2016-55

Keyword(s):
Bi-directional connector; In-memory; Relational database; Vertica; Graph engine; GraphLab

Abstract:
We present VGL, an in-memory, bi-directional connector between a relational database and a graph engine that avoids the overhead of data serialization. This connector enables the relational engine to efficiently off-load computation to the graph engine and, conversely, enables the graph engine to efficiently access and use legacy relational data.

External Posting Date: July 5, 2016 [Fulltext]
Internal Posting Date: July 5, 2016 [Fulltext]

Hewlett Packard Enterprise

VGL: Enabling Graph Computation over Enterprise Data

Alex Kalinin
Brown University
Providence, USA
akalinin@cs.brown.edu

Alkis Simitsis
Hewlett Packard Labs
Palo Alto, USA
alkis@hpe.com

Kevin Wilkinson
Hewlett Packard Labs
Palo Alto, USA
kevin.wilkinson@hpe.com

Mahashweta Das
Hewlett Packard Labs
Palo Alto, USA
mahashweta.das@hpe.com

Abstract

We present *VGL*, an in-memory, bi-directional connector between a relational database, HPE Vertica, and a graph analytic engine, GraphLab, that avoids the overhead of data serialization. This connector enables the relational engine to efficiently off-load computation to the graph engine and, conversely, enables the graph engine to efficiently access and use legacy relational data.

1 Introduction

Enterprises use relational databases to store operational and historical data. They analyze this data using a variety of algorithms. But some complex algorithms are much more easily expressed and computed as graphs. In particular, iterative computations over enterprise data may involve multiple self-joins on tables of vertices and edges and updating a large portion of graph's vertices or/and edges (e.g., PageRank, Connected Components), and thus, they may consume a large fraction of the relational engine's resources and interfere with other analytic requests. In our work, we investigate using a graph analytic engine as a co-processor for a relational engine to off-load complex graph computations. Earlier work on coupling graph and relational engines shows the overhead of data transfer to be a limiting factor [8]. Here, we address how to optimize the connection between the two engines to enable engine cooperation.

In the last few years, several graph engines emerged and competed for market share in graph analytics. Among their sweet spots is the ease in representing and the speed in executing a certain class of analytics queries over a graph data model that are typically harder to express in traditional query languages like SQL. These engines however are built with a working assumption that the core of the data is stored as a graph in a graph engine, while all outside information needs to be imported to the graph engine first to become usable, i.e., queryable [6, 13]. And for that, most engines provide a number of methods, often suboptimal, to connect to external data stores, e.g., a database or filesystem.

To connect a database and a graph engines, we need to either replicate all the relational data in the graph engine or transfer data as needed per request; e.g., copy the data related to a product campaign to a graph engine and create a graph there, run analytics over the graph, potentially combine the results with data still stored in the database, and finally discard the graph when the campaign is over. For a number of reasons related to stability, robustness, performance, scalability, and support for legacy applications, it does not seem practical to replicate all data for an enterprise from the relational engine to the graph engine. We need to have both engines and a connection between them. But this is not trivial either since it requires moving the data from the relational database to the graph engine, computing the result, and moving it back to the relational database. The time required for this data movement can exceed the computation time and the resources required interfere with other concurrent requests on both systems.

In this paper, we present a method to reduce latency and resource usage for the data movement,

enabling better utilization of both the relational database and graph engine, and better service for end users. We present an efficient, bi-directional connector between a database and a graph engine that uses a shared memory buffer to greatly reduce the data shipping and function shipping overhead between the two engines. The core idea is that as we export the data from the database, it is transformed into the internal data structures of the graph engine and stored in a shared memory buffer, thus eliminating the overhead of a graph engine to both load the source data and convert it to a graph. This shared memory buffer enables a high-bandwidth, low-latency connector between the engines.

Our connector is faster and uses fewer compute resources than existing techniques that export the relational data and encode it as text or binary data, which is then ingested by the graph engine. Due to its efficiency, it allows the relational database to off-load more computation to the graph engine, thus providing better services for users of both systems: the graph requests are processed faster and, since the database has reduced load, database requests are processed faster too. It provides efficient mapping in both directions. In other words, our connector can be used to transfer data in three ways: (1) graph ingest to map from a relational database to a graph engine, (2) graph export from a graph engine to a database, (3) round-trip from a relational database to a graph engine, perform some computation on the graph engine, and pass the result back to the relational database. In some realizations, our connector can be used to embed the graph engine directly in the relational database by wrapping it as a user-defined database function. In some cases, this can provide significant performance advantages by eliminating context switches between systems.

Our solution enables new applications by providing an efficient connection between a graph engine and legacy data. One use case is predictive analytics where historical data is analyzed to generate a fast-access, predictive model that applications can then query in real-time. Our work facilitates use of graph methods for predictive analytics by enabling computations that rapidly integrate graph data and relational. Further, a variety of businesses use graph models and algorithms and can benefit from this invention, e.g., telecommunications, network security, IT asset management, national security, and scientific computing.

2 The *VGL* Connector

VGL is the initial implementation of our connector. It links HPE Vertica database and the GraphLab graph analytic engine [3, 11]. But, our method applies to most relational database engines and graph engines. We only require (i) that the database supports user-defined functions (UDFs) to execute arbitrary code within the database, and (ii) that the graph engine (perhaps with minor modification) can process data from shared memory. *VGL* currently assumes co-tenancy of the relational engine and the graph engine on a single node. This allows both engines access to a shared memory buffer. But, it limits scale-out of the engines. Future versions of *VGL* would overcome this by leveraging emerging technology for inter-node shared-memory, e.g., RVMA.

Background. Before presenting *VGL*, we briefly describe how the two engines store data.

Vertica. Vertica is a parallel, column-store, relational database that is tuned for analytics. For scale-out across multiple nodes, large tables may be segmented, i.e., physically stored across nodes in a cluster. Segments are replicated on different nodes for fault-tolerance. Within a node, a segment is physically stored as a number of containers (files) on disk. For scale-up within a node, containers of a segment can be read in parallel by threads on multiple cores. Vertica supports extensibility in a number of ways. UDFs can be created for data transformation or for data generation (e.g., to ingest data from another source, a file or a buffer). Vertica provides a very efficient bulk insert command and user-defined parsers may be used to ingest arbitrary data formats.

GraphLab. GraphLab runs as a single-node server (for implementation details see [3]). An application process accesses the server using a proprietary remote procedure call (RPC) interface. Application requests are serialized, sent to the server, the op-

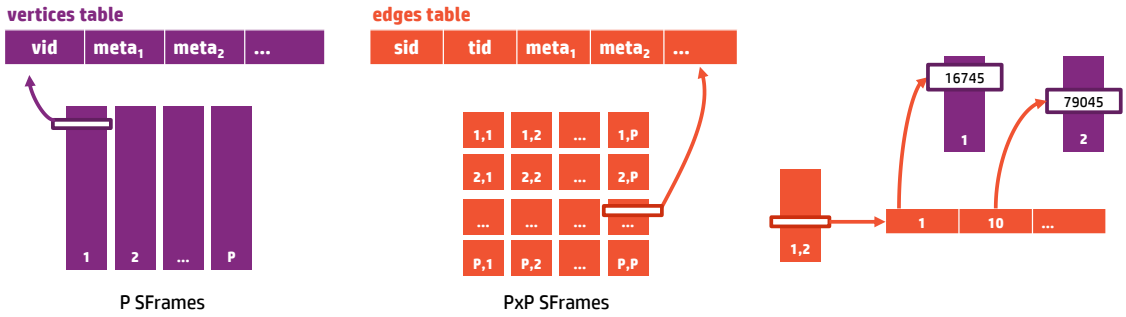

Figure 1: GraphLab storage: vertices table (left), edges table (center), and edge-to-vertices 'index' (right)

eration is performed, and results are serialized and returned to the client process. GraphLab stores a graph using two logical tables, a vertex table with schema: $(vid, meta_1, meta_2, \ldots)$ and an edge table with schema: $(sid, tid, meta_1, meta_2, \ldots)$, where *vid*, *sid*, *tid* are vertex identifiers and $meta_i$ are metadata containing key-value property pairs for a vertex or edge (see Figure 1).

The vertex table is divided into P partitions. P is a tunable parameter with a default value of 8. The best value depends on the CPU and the data. Each partition is stored as an SFrame (see below). The vertices are shuffled into partitions based on their *vid* values. The edge table is divided into P^2 partitions. Each edge partition is also stored as an SFrame. The edge partitions comprise a $P \times P$ matrix. An edge partition with matrix coordinates (X, Y) stores all edges that connect vertices in vertex partition X to vertices in vertex partition Y. For each edge, *sid* and *tid* values are replaced by the corresponding vertices' ordinal positions in their respective partitions. This basically creates an index: for each edge, its *sid* and *tid* point to the appropriate rows in the vertex partitions.

GraphLab's internal data structures (S-structures) are SArray, SFrame, and SGraph. SArray is an array of elements of the same type. It is divided into segments enabling parallel writing and reading. Each segment consists of a number of blocks storing compressed data. It consists of SArrays stored together. SGraph is GraphLab's representation of a graph. Logically, an SGraph comprises the vertex table and the edge table.

Physically, when the files storing S-structures are closed, they can be reopened only for reading not for appending. But at the logical level S-structures are only descriptors referencing the corresponding files. At a lowest level, they are built as SArray segments. And we can append to an SArray by appending segments to it (the new segment will be in a different file). We just need to change the descriptors, adding the reference to the new file, without any data movement. Thus, for adding a column to an SFrame, we add the SArray to the SFrame's descriptor. Appending to an SFrame involves just appending the SArrays of the second SFrame to the SArrays of the first one. And these operations being purely CPU operations are in general cheap.

GraphLab loads data from various sources (e.g., HDFS, S3, file system) and supports various formats (e.g., CSV, ODBC, and binary). For CSV, GraphLab uses a parallel parser that divides the file into multiple fragments, depending on the degree of parallelism, and then loads the result into an SFrame. Note that when creating an SGraph from a CSV, the CSV is treated as an edge table. This edge table is first loaded into an in-memory SFrame, which is then used to create the graph. The vertex table is created from the edge table by parsing source/destination ids. ODBC works as with CSV. Via the ODBC connection the specified table is retrieved and stored in an SFrame, which is then used to create an SGraph. Currently, GraphLab ODBC is single-threaded. Binary format is the native GraphLab format for SAr-

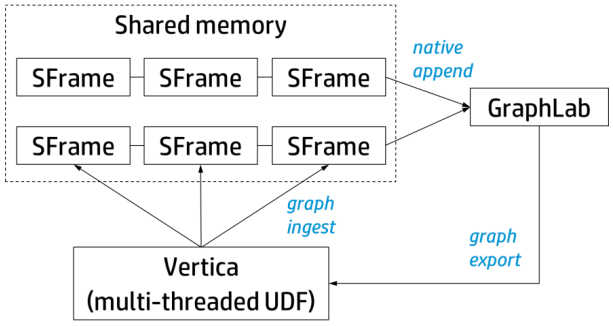

Figure 2: VGL architecture

rays/SFrames. Any S-structure can be saved into files for persistence or imported from them. GraphLab export works similarly; e.g., for storing a graph into an external file or into a database.

The *VGL* Connector. *VGL* is a bi-directional connector between Vertica and GraphLab. Conceptually, it works as follows. Let's assume that a database stores data in relational format, say R, and a graph engine stores data as a graph, say G, using a graph topology and additional storage for graph properties. *VGL* connects the database and graph engines and seamlessly transforms data from relational to graph and vice versa. To achieve this, it uses an intermediate data format, say SF for GraphLab SFrames, stored in shared memory. *VGL* applies a transformation $T_{R \to G}$: $R \to SF \to G$ and its reverse transformation $T_{G \to R}$: $G \to SF \to R$. *VGL* architecture is illustrated in Figure 2.

Graph ingest (database to graph connection). Assume a use case where we need to perform a graph computation on data stored in a relational database. To do this, we must extract the relational data into an intermediate form and then ingest that into a graph engine. This ingest consists of two generic phases: *data import* (load data into the engine) and *graph creation* (convert the data into a graph structure). For ease of presentation, let's assume that the relevant data is stored in two database tables, one for vertices and a second for edges.

Baseline. There are two baseline choices for graph ingest using existing GraphLab and Vertica functionality. First, if the graph engine supports a database connection (ODBC, JDBC, etc.), we can import the vertex and edge tables via a database connection to a common data structure (SFrame) and then create the graph from it. But ODBC is not highly optimized (single-threaded operation). In addition, ODBC itself introduces a non-trivial overhead. Second, we can export the data from a database into a file, e.g., CSV, and then load it using a native loader provided by the graph engine. In practice, this is very expensive.

These two approaches have several disadvantages on both engines. On a database, there is a non-trivial overhead for serializing the data into a common format and then converting it into a graph format. On a graph engine, there is the overhead of the import process. Typically, a graph engine partitions the graph into vertex and edge partitions. Thus, when loading a graph, the data must be shuffled into multiple vertex/edge partitions, which also includes the extra process of extracting the vertices from the edges and indexing both the vertices and the edges to their corresponding partitions.

VGL operation. In Vertica, we pre-shuffle the data into the appropriate number of vertex/edge partitions (explained shortly) and write these partitions directly in SFrame format. GraphLab gets the written edge and vertex partitions as inputs and creates proper SFrame descriptors from them, which is a fast, CPU-only operation. Then, it only has to index edges to point to rows in vertex partitions. For an additional performance boost, SFrames are transferred in and read from the shared memory (POSIX, /dev/shm).

To implement this functionality, we created the following Vertica C++ UDFs:

- gl_vertex_export(id, meta_1, meta_2, ...), which treats the input table as a vertex table with the column 'id' representing the vertex ids and 'meta_*' representing additional meta-data. The function takes the tuples and shuffles them into the appropriate number of vertex SFrames in shared memory. The function returns links to the created shared memory segments (e.g., /dev/shm/twitter_vertex_0.sf).
- gl_edge_export(sid, tid, meta_1, meta_2, ...), which is a similar function to create edge par-

titions. The 'sid' and 'tid' columns represent source and target vertex ids.

For example, suppose we have two Vertica tables with social network data describing people and their followers. We can export the data to GraphLab by executing the SQL commands below. The first Select exports graph vertices from the Persons table, where a vertex is identified by *person_id* and has properties *person_age* and *person_gender*. The second Select exports graph edges from the Followers table where each edge has a property *start_date*. In both cases, T specifies the number of UDF threads to spawn. The number of GraphLab partitions is not a parameter; it is a fixed value in these functions.

```
SELECT gl_vertex_export (
    person_id, person_age, person_gender )
    OVER ( PARTITION BY mod (
        hash ( person_id, <T> ) )
FROM Persons;

SELECT gl_edge_export (
    person_id, followed_by_id, start_date )
    OVER ( PARTITION BY mod (
        hash ( person_id, <T> ) )
FROM Followers;
```

We also extended GraphLab in a number of ways. First, we extended it with the ability to support shared-memory blocks in its cache. GraphLab supports SArrays that can be backed by in-memory files, referenced by special URLs. These files are represented by heap-memory blocks. We extended GraphLab to support not only heap, but also shared-memory cache blocks. These can be referenced by internal URLs in the form of "cache://shm/<file_name>". In GraphLab's cache manager, such shared-memory blocks are mapped into memory (mmap() call). The usual cache limit parameters apply to this, but if the cache manager tries to evict a shared-memory block, an exception is thrown to avoid performance degradation. This is, however, applicable to writes only. Memory mapping always succeeds when opened for reading.

We implemented a method parallel_append_sframes_urls() to support fast SFrame creation from URLs produced by the Vertica UDFs. This is especially important for parallel execution. Assume the gl_vertex_export() function creates P vertex SFrames. When this UDF runs in parallel as T threads in Vertica, each thread creates P vertex SFrames, resulting in $P \times T$ SFrames in total. So, parallel_append_sframes_urls() takes all these URLs and produces P proper vertex SFrames by appending the corresponding shared memory segments in parallel (T for each resulting SFrame). Note that these SFrame objects are just descriptors (so there is no data movement), which can be immediately used in other GraphLab API calls without modification.

We added a new GraphLab method, add_vertices_edges(). This takes vertex and edge partitions and appends them to the existing graph's data. Considering the function's knowledge about its parameters, this function is more optimal than the GraphLab add_edges() function in the following ways:

- Extracting vertices from edge partitions is unnecessary. Vertices are specified explicitly as a parameter.
- No shuffling is needed. Vertices and edges are already pre-shuffled.

However, the existing vertices are still scanned for possible vertex updates: if one of the new vertices has the same id as an existing one, the old vertex is replaced (its metadata, effectively). Also, the edge partitions indexing is performed as before, since it is required for query processing.

Hence, a typical graph ingest process follows these steps:

1. The client calls our Vertica vertex and edge export functions with appropriate parameters. This results in creating shared memory SFrames. The client gets segment URLs as the result.
2. The client passes the URLs to our GraphLab function parallel_append_sframes_urls(), which initiates SFrames.
3. The client creates an empty graph (SGraph() call) and calls our new add_vertices_edges() function to import the SFrames into the SGraph. At

this stage, effectively only the edge partition indexing is performed, as most of the ingest work is already done.

An optimization: stateful graph ingest. An optimization of the above method can be done through a 'stateful' ingest process, which performs the edge indexing at the Vertica side. To enable this, we added additional functions as described below.

We implemented a Vertica UDF gl_export_state_init() to create a shared state. This creates additional structures in memory that can be used between the vertex and edge export UDFs. When the vertex export function runs, it automatically detects the created state and switches to the stateful mode. In this mode, it not only shuffles the vertex tuples into SFrames, but it also creates vertex hash-maps (vertex_id $\rightarrow$ row number) in memory.

It is worth noting two implementation details. First, if the UDF is multi-threaded (PARTITION BY), the ordering of the created SFrames belonging to the same vertex partition is important, since the hash-maps reference partition-wide row numbers. Thus, each such SFrame is assigned an ordinal number. This number is used later to append SFrames from different threads in GraphLab in the correct order. For this, in addition to the partition id, the UDF outputs the ordinal numbers as well. Second, since creating and subsequently updating the hash-maps for the same partition requires locking in a multi-threaded environment (e.g., if two threads create same-partition SFrames at the same time), the UDF uses simple randomization so that different concurrent threads should not try to update hash-maps for the same partition.

Then, when a client runs the edge export function, it checks if the state exists and uses the previously created maps to index edges. After the function ends, the client has to clear the state using a gl_export_state_clear() function.

Graph export (graph to database connection). Assume a use case where we need to perform a graph computation on a graph engine and we need to send the result graph back to a database. Let's assume that the result graph is represented as an SFrame structure. For example, for PageRank the SFrame may have two columns: vertex id and the PageRank value.

Baseline. Similarly to graph ingest, there are two baseline choices for sending the resulting SFrame back to a database as a table using existing GraphLab and Vertica functionality: (a) propagate SFrame data (which effectively is a table) from the graph engine to an RDBMS using a standard database connection protocol (e.g., ODBC), and (b) export SFrame data to an intermediate file (e.g., CSV) and then load this file into a database (e.g., with a CSV parser). As we discussed in graph ingest, these approaches introduce considerable overhead. Both require some form of serialization, either into CSV or the ODBC format. And then, the data also needs to be ingested in the database.

VGL operation. The *VGL* connector overcomes these limitations. The basic idea behind the connector is to keep the resulting SFrame in shared memory and then import it into Vertica by using User-Defined Parser (UDP) and User-Defined Source (UDS). To implement this, we made the following extensions to GraphLab.

First, we introduced the idea of a 'shared SFrame/SGraph'. If all columns of an SFrame are in shared memory, then it is shared. SGraph is shared if all of its vertex and edge SFrames are shared. Then, when a query is being processed over a shared graph, it stores all intermediate results and the final result in shared memory. And finally, we introduced a new API function namely prepare_for_export() that moves, if necessary, columns of the SFrame into shared memory together with the resulting SFrame descriptor.

With these modifications it is now possible to keep any GraphLab object in shared memory, not only graph query results. Hence, for graph queries, a typical process follows these steps:

1. The graph is ingested via shared memory using our connector. Thus, the SGraph has the shared property.

2. After the query is done, the result is in shared memory already.

3. The function prepare_for_export() is invoked. This is needed to create the descriptor referencing the resulting shared-memory SFrame. Since the data is already in shared memory, the operation is almost instantaneous.

4. The function returns a single SFrame URL for the results in the form of cache: //shm/<some name>.

To ingest an SFrame from shared memory into Vertica, we created a UDS and a UDP.

User-Defined Source (UDS) called SFrame(): This source reads the SFrame metadata from shared memory and passes the corresponding data to the next component SFrameParser(). SFrame() instantiates multiple sources for disjoint segments of the SFrame (see the background discussion on GraphLab earlier in this section). The information for a segment (i.e., URL and the range of rows) is serialized into a small binary message to the parser.

User-Defined Parser (UDP) called SFrameParser(): The parser takes the information about a segment passed by the SFrame UDS, reads the corresponding data from shared memory and uses Vertica API to put the data into the Vertica table.

Graph compute round-trip. *VGL* also supports a round-trip process. In this scenario, relational data is sent to the graph engine for some graph computation and the result is sent back to the database as relational data. This is a simple composition of graph ingest and graph export with an intervening graph computation. This step also requires some orchestration to sequence the three phases: ingest, compute, and export.

3 Evaluation

In this section, we present an evaluation of the *VGL* connector.

Datasets and Configuration. We first describe our experimental setting.

Data sets. For our evaluation, we used two data sets: (a) Twitter graph data [2] and (b) LDBC graph (person knows person) generated using the LDBC benchmark generator at scale factor 1000 [10]. Table 1 summarizes the data sets characteristics. We intentionally chose data sets that fit in main memory, as we are mostly interested in main memory process here.

Hardware/Software. The experiments were performed on a single machine with: Intel Xeon E5-2660v2 (40 cores), 130GB memory, Debian 8.1. And we used Vertica v7.1.2-0 and the latest (as of Aug. 2015) version of open-source GraphLab [3].

Multi-threaded execution in Vertica. For multi-threaded execution we used the following statement:

```
SELECT <udf> OVER (
  PARTITION BY mod( hash(<id_cols>), <partCnt>)
)
FROM <table>
```

where `<id_cols>` are the columns used for vertex ids or edge source-destination ids and `<partCnt>` is the number of desired partitions. This causes Vertica to dynamically repartitioning the table into `<partCnt>` partitions and then run an instance of the vertex and export UDFs over each partition. Note that by 'partition' in Vertica we mean a table partition, which is the number of threads spawned to process the table; this should not be confused with 'partitions' (SFrames) in GraphLab.

Varying the number of table partitions. We used the Twitter data set to test how the number of dynamic Vertica table partitions affects the time to fully export vertex and edge tables into shared memory. We experimented with both fenced (safer, recommended for production) and unfenced (faster) UDF execution modes [7] and found that the trends are the same in both modes. We used unfenced execution in the rest of the experiments. Figure 3(left) shows that 16 partitions is a sweet spot for this particular machine. It seems that additional partitions result in more partitioning overhead (64 partitions cannot be actually completed in parallel, since we have only 40 cores).

Vertices and edges in the graph are shuffled into multiple partitions. Assuming P vertex partitions, then we have P^2 edge partitions. Figure 3(right) shows the time to export the graph to shared memory

Table 1: Data sets

Data set	Vertices	Edges	CSV size	SFrame binary size	Vertica tables size	Metadata
Twitter	52,579,682	1,963,263,821	37GB	8.2GB	52MB (vertex), 8.4GB (edge)	no
LDBC (SF 1000)	3,599,905	447,163,916	12GB	2.8GB	5MB (vertex), 2GB (edge)	no

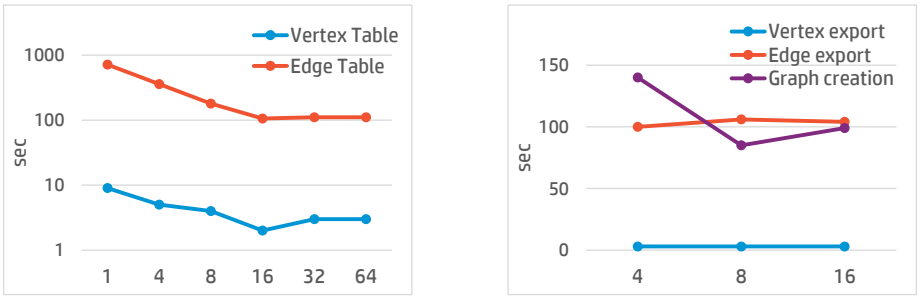

Figure 3: Varying the number of table partitions in Vertica (left - log scale) and vertex partitions in GraphLab (right)

and also to create it. The former is important, since the number of graph partitions each UDF thread produces is different (each vertex export UDF thread writes P and each edge export UDF thread writes P^2 partitions). The default GraphLab value is $P = 8$. For Vertica we used 16 table partitions in this experiment.

The number of shared memory segments written by Vertica does not affect export time much, as most processing is in-memory. Regarding graph creation, 4 vertex partitions do not perform well, as they results in 16 edge partitions and thus, more than half the cores are not used during graph creation. Increasing the number of partitions results in full core utilization. For example, 8 vertex partitions would result in 64 edge partitions and complete in two iterations: 40, and then 24. Each iteration runs in parallel. Note that graph creation includes combining URLs, produced by UDFs, into SFrames. And this is a relatively fast process. In the 16-case UDFs produce 16*16+256*16=4352 shared memory segments. In this case combining them using the aforementioned parallel_append_sframes_urls() function takes about 3 seconds. In the other cases, combining partitions finishes in under 1 second.

Database to graph connection. We experimented with four alternatives for graph ingestion:

- ODBC: Ingesting a graph via GraphLab ODBC. Internally, this is a two-stage process: ODBC → SFrame → SGraph.
- CSV: Importing from a CSV file using the native GraphLab loader. Internally, it is a two-stage process: CSV → SFrame → SGraph.
- SFrame: Importing a previously prepared SFrame, which is logically equivalent to the above CSV.
- *VGL*: Our *VGL* connector (with 16 Vertica threads and 8 vertex partitions).

Figure 4 shows the total time needed to load a graph to GraphLab. The total time is the time from the start of the import to the moment the graph is fully ready for analytical processing and consists of two main phases: data import and graph creation. The left plot of Figure 4 shows the results using the Twitter data set and the right, using the LDBC data set. The trend for graph ingest is the same in both plots. However, the scale is different due to LDBC being a smaller graph than Twitter.

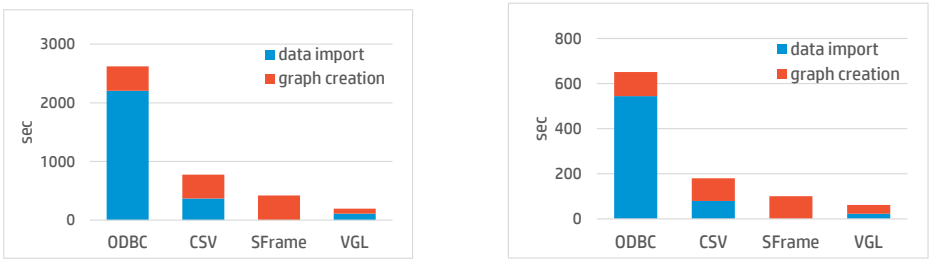

Figure 4: Graph ingestion, Twitter (left) and LDBC (right) graphs

Clearly, the *VGL* approach is the most efficient. In addition, *VGL* besides storing the graph in GraphLab, it *also* allows the data to be stored in a database too. On the other hand, ODBC demonstrates very poor performance. CSV is better than ODBC, but still not fast enough. SFrame demonstrates good performance, but it assumes that the SFrame has already been produced somehow. It is listed here for presentation reasons only. In practice, its total time is higher, depending on how the SFrame will be produced. For example, one way to produce an SFrame is through CSV that has a non trivial overhead.

Graph to database connection. We compared three alternatives: ODBC, CSV, *VGL*, for exporting graph data from GraphLab and importing it to Vertica. Figure 5 shows the results for the Twitter (left) and LDBC (right) graphs.

The ODBC overhead overcomes even the overhead of creating an intermediate CSV file. This indicates that there is room for significant optimization in the implementation of GraphLab ODBC. On the other hand, CSV file import/export is reasonably optimized on both systems. But again, our *VGL* connector performs best, as its only extra overhead comes from Vertica's API cost and it is the minimum possible without changing Vertica internals.

Query Performance. Next, we evaluated graph query execution using the Twitter and LDBC graphs, comparing two alternatives:

- Vertica: Running an analytics query on Vertica, using SQL.
- *VGL*: Ingesting the graph into GraphLab (data import), executing the query there (processing), and importing the result back in Vertica (result export).

We chose three representative graph analytics queries: PageRank (PR), Connected Components (CC), and Single-Source Shortest Path (SSSP). We executed each query on both data sets using both execution alternatives. The results are shown in Figure 6 for PR (top), CC (center), and SSSP (bottom). For *VGL*, we list the individual times of each execution stage: data import, processing, result export. For Vertica, we only report the query execution time.

As can be seen the benefits of *VGL* even for a single query are quite evident. The notable exception is SSSP, for which GraphLab provides inferior query processing times comparing to our SQL implementation in Vertica. We think this is because the SSSP implementation in GraphLab might not track 'active' vertices, doing a whole scan at every iteration. In addition, data import and export exacerbates the problem. We find this as a very interesting result. Although a graph engine is highly optimized for graph analytics queries, if the query implementation is not efficient, a traditional database engine can perform equally well (if not better). However, as the data set size increases, the performance of a pure database approach is affected as the database engine is not designed and optimized for such graph analytics queries.

Discussion. Massive parallelism approaches (e.g., GraphX/SPARK) scale well but have a high initial overhead; i.e., a single node graph engine often does better than a 4-8 node parallel graph engine. Many believe that most real-world (useful) graphs

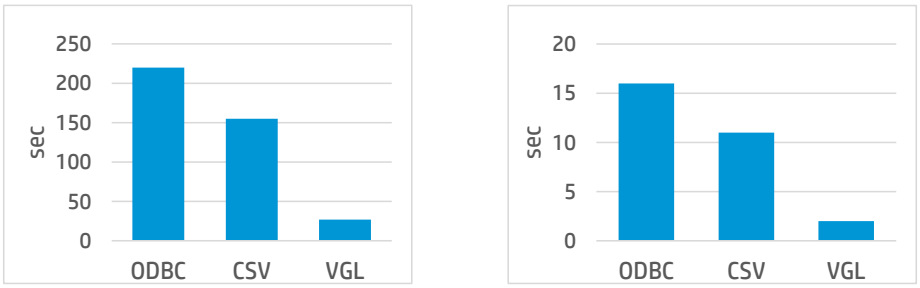

Figure 5: Graph export, Twitter (left) and LDBC (right) graphs

can be stored and analyzed using a modern single node server with large memory and many cores. In other words, we believe two specialized engines (relational and graph) could be more cost-effective and perform better than a general-purpose scale-out architecture.

4 Related Work

There exist several specialized large-scale graph processing systems such as GraphLab [11], Pregel [12], PowerGraph [6], Trinity [13], etc. that provide tailored optimizations for different kinds of algorithms and executions. But most enterprises continue to use relational databases for capturing, managing, and storing data [1, 5]. This has led to a growing interest in employing relational databases for graph analytics workloads by translating graph analytics queries to SQL equivalent [8, 15], designing user-friendly non-SQL query interface on top of relational databases [4, 9], and integrating graph analytics queries with SQL language [14], etc.

We propose a novel way of leveraging the power of both relational databases and graph engines in order to support a broader range of graph computations and algorithms. Unlike graph engines which cannot process analytics queries that require information outside the memory-resident data (e.g., [6]) or suffer from loading overheads (e.g., [11]), and unlike relational database solutions that require expressing complex analytic queries as SQL (e.g., [8]) or understanding a new framework (e.g., [4]), *VGL* uses a graph engine as a co-processor for a relational engine to offload analytic computational overheads while using the relational engine as a co-store for a graph engine to offload data movement overheads.

Connecting relational databases to specialized processing engines (e.g., R) is not new. To the best of our knowledge, using a memory buffer to pass engine-specific data structures for an efficient, bi-directional link has not been done before.

5 Conclusions

We presented *VGL*, an in-memory, bi-directional connector between a relational database, HPE Vertica, and a graph analytic engine, GraphLab, that avoids the overhead of data serialization. *VGL* is a fully functional solution. It requires minor GraphLab modifications to enable use of shared-memory.

There are several possible extensions. Adapting *VGL* to support multi-node Vertica instances would extend its range of applicability. This includes leveraging new low-latency, inter-node IPC as mentioned earlier. An interesting research question is when it is beneficial to off-load processing from the relational engine. As shown above, the benefit of off-loading graph computation is variable. Techniques for deciding when to do so are an interesting research challenge.

References

[1] N. Bronson et al. TAO: Facebook's distributed data store for the social graph. In *USENIX*, pages 49–60, 2013.

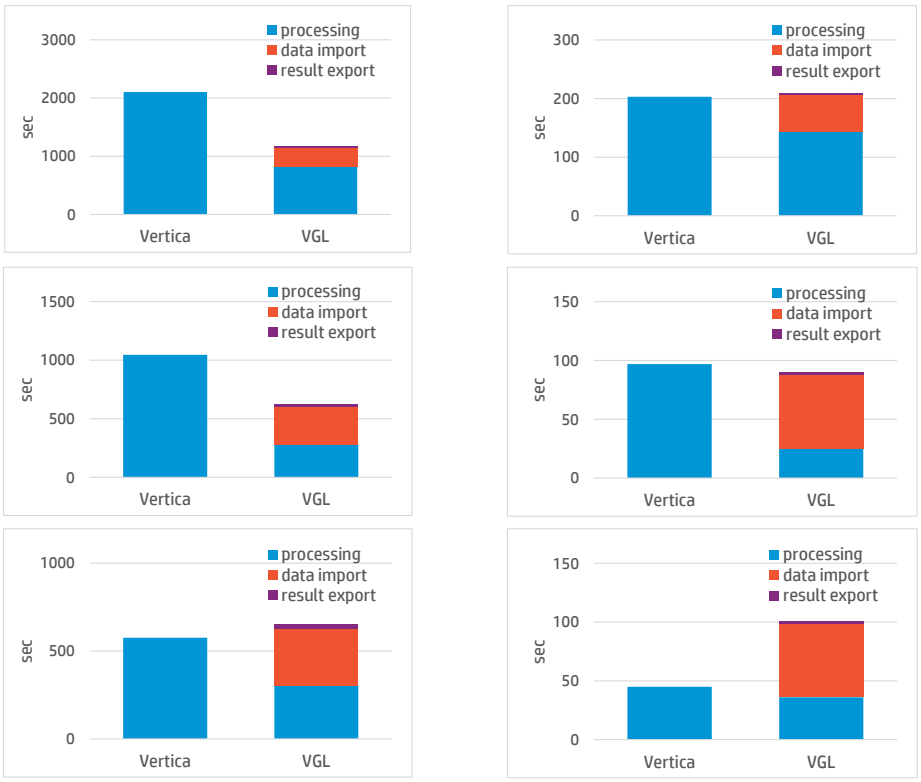

Figure 6: PR (top), CC (center), SSSP (bottom) on Twitter graph (left plots) and LDBC graph (right plots)

[2] M. Cha et al. Measuring User Influence in Twitter: The Million Follower Fallacy. In *ICWSM, URL: http://twitter.mpi-sws.org/*, 2010.

[3] Dato, GraphLab open source project. *URL: https://github.com/dato-code/Dato-Core*, 2015.

[4] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[5] FlockDB. *URL: http://github.com/twitter/flockdb/*, August 2011.

[6] J. E. Gonzalez et al. Powergraph: Distributed graph-parallel computation on natural graphs. In *USENIX*, pages 17–30, 2012.

[7] HPE Vertica. *Developing and Using User Defined Extensions, Programmer's Guide*, vers. 7.0.x, 2015.

[8] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph analytics using the Vertica relational database. *CoRR*, abs/1412.5263, 2014.

[9] A. Jindal et al. VERTEXICA: Your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[10] LDBC. *URL: http://ldbcouncil.org/*, August 2015.

[11] Y. Low et al. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[12] G. Malewicz et al. Pregel: a system for large-scale graph processing. In *SIGMOD*, pages 135–146, 2010.

[13] B. Shao et al.. Trinity: a distributed graph engine on a memory cloud. In *SIGMOD*, pages 505–516, 2013.

[14] D. E. Simmen et al. Large-scale graph analytics in aster 6: Bringing context to big data discovery. *PVLDB*, 7(13):1405–1416, 2014.

[15] A. Welc et al. Graph analysis: do we have to reinvent the wheel? In *GRADES*, page 7, 2013.