VERTEXICA: Your Relational Friend for Graph Analytics!

Alekh Jindal* Praynaa Rawlani* Eugene Wu*
Samuel Madden* Amol Deshpande⋆ Mike Stonebraker*

*CSAIL, MIT ⋆University of Maryland

ABSTRACT

In this paper, we present VERTEXICA, a graph analytics tools on top of a relational database, which is user friendly and yet highly efficient. Instead of constraining programmers to SQL, VERTEXICA offers a popular vertex-centric query interface, which is more natural for analysts to express many graph queries. The programmers simply provide their vertex-compute functions and VERTEXICA takes care of efficiently executing them in the standard SQL engine. The advantage of using VERTEXICA is its ability to leverage the relational features and enable much more sophisticated graph analysis. These include expressing graph algorithms which are difficult in vertex-centric but straightforward in SQL and the ability to compose end-to-end data processing pipelines, including pre- and post- processing of graphs as well as combining multiple algorithms for deeper insights. VERTEXICA has a graphical user interface and we outline several demonstration scenarios including, interactive graph analysis, complex graph analysis, and continuous and time series analysis.

1. INTRODUCTION

Graph analytics is getting increasingly popular with several new application domains such as social networks, transportation networks, ad networks, e-commerce, and web search. These graph analytics workloads are seen as quite different from traditional database analytics, largely due to the iterative nature of many of these computations, and the perceived awkwardness of expressing graph analytics as SQL queries (which typically involves multiple self-joins on tables of nodes and edges). As a result, many new storage and querying systems optimized for graph algorithms have been proposed. In particular, a number of so-called "vertex-centric" systems (e.g., Pregel [5], Giraph [3], GraphLab [4], GPS [6], and Trinity [7]) have been proposed in recent years.

Relational databases, on the other hand, are completely missing from this wave of new graph processing systems. Note that in many real-world scenarios, data is collected and it resides in a relational database in the first place, e.g. [1, 2]. Using a different system for graph analytics would mean that the users now need to dump their data from the relational database to a graph database. This involves stitching different systems together and is highly undesirable. Furthermore, users might find several features from relational databases, such as transactions, checkpointing and recovery, fault tolerance, durability, integrity constraints, etc., hard to forego. Interestingly, relational databases have been highly optimized for relational data analytics in recent years, in particular column-oriented data storage and query processing has emerged as an attractive technology. A natural question, therefore, is whether these techniques can be leveraged for efficient graph analytics as well?

In this paper, we present VERTEXICA, a relational database system which is friendly yet efficient over graph analytics. There have been some early efforts to implement graph queries in relational databases [8]. However, these works constrain the graph analysts to SQL, the de facto query interface in relational databases. Unfortunately, implementing graph queries in SQL can be tricky and time consuming. This is because with SQL, the programmer must manipulate relational tables rather than graphs. In contrast, VERTEXICA provides a vertex-centric query interface, wherein the programmers writes his graph queries as vertex-compute functions and system takes care of running it on the underlying relational engine. As a result, VERTEXICA allows programmers to easily express several graph analytic queries such as PageRank, shortest path, connected components, stochastic gradient descent, random walk with restart, and other message passing algorithms. Apart from efficiently executing vertex-centric programs, VERTEXICA also allows users to leverage and combine the traditional strengths of the relational database. For example, VERTEXICA allows ad-hoc mutations to the graph as well as the associated metadata, which is simply impossible to do in many new graph processing systems such as Giraph. Likewise, VERTEXICA allows users to easily combine graph algorithms with relational operators, thereby facilitating more advanced graph queries e.g. localized PageRank. Overall, our goal in this paper is to demonstrate how VERTEXICA enables efficient, easy-to-use, and rich graph analytics right on top of a relational engine.

In the following, we first show the details of VERTEXICA. Thereafter, we discuss the use-cases, for which VERTEXICA is a useful tool (Section 3). Finally, we sketch our demonstration proposal describing the setup and demonstration scenarios, and highlight how the users can interact and play with VERTEXICA (Section 4).

2. VERTEXICA

The core idea of VERTEXICA is to bring user-friendly and high-performance graph analytics capability to a relational database. VERTEXICA does this by injecting data storage, query processing, and query interface support to enable efficient vertex-centric graph analytics. Currently, VERTEXICA sits on top of an industry strength column-oriented database system. However, it could be extended to other relational databases as well. Below, we discuss the vertex-centric interface, the implementation details, and the optimization techniques employed in VERTEXICA.

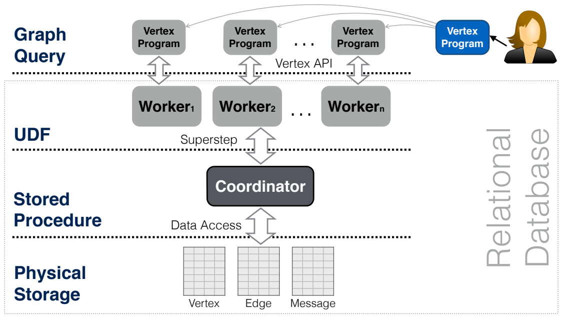


Figure 1: Vertexica Architecture

2.1 Vertex-centric Interface

Relational databases have SQL query interface and although one can imagine implementing graph queries in SQL, crafting and tuning the SQL can be tedious and time consuming. This is because manipulating relational tables is not really intuitive for expressing graph queries. In contrast, the concept of *vertex-centric programming* allows programmers to think in terms of graphs. This interface allows programmers to specify their computations as UDFs, which are executed as *supersteps* on the graph vertices. The UDFs communicate by sharing messages with neighboring nodes. In most implementations, this sharing is done after a synchronization barrier at the end of every superstep. As a result of the vertex-centric interface, programmers do not have to worry about details such as partitioning the graph, distributing the computing across multiple machines, and coordinating the message passing. This is analogous to MapReduce where the programmers simply provide the *map* and *reduce* functions and the framework takes care of the system details.

Vertexica provides a vertex-centric interface on top of SQL. The interface exposes the same API as in Pregel and manages parallelizing the vertex compute function, the passing of messages, and synchronizing the supersteps. As in Pregel, programmers simply provide their vertex compute function, and Vertexica takes care of running it as standard SQL (with UDFs) in an unmodified relational database. Thus, with the vertex-centric interface, Vertexica allows programmers to actually *think* in terms of a graph while still running their queries in a relational engine.

2.2 Implementation Details

Let us now look at the implementation details of Vertexica. Figure 1 shows the architecture of Vertexica. It has four major components: (1) the physical graph storage, (2) the coordinator to drive the vertex-computations, (3) the workers which run the vertex-computations, and (4) the actual compute function provided by the user. We describe each of these below.

Physical Storage. Vertexica stores all data in three relational tables: (1) the vertex table to store the vertex id, the vertex value, and the vertex state, (2) the edge table to store the edge source and the edge destination, and (3) the message table to store the sender vertex, the receiver vertex, and the message value.

Coordinator. The coordinator is the driver program that manages the supersteps. It is responsible for running the vertex computations in parallel and passing messages from one superstep to the next superstep. We implement the coordinator as a stored procedure; it runs as long as there is any message for the next superstep. As shown in Figure 1, in each superstep, the coordinator invokes a set of worker UDFs to run the vertex program.

Worker. The worker is the container for the vertex-compute function. It is responsible for sending and receiving messages with the coordinator. Additionally, the worker exposes similar API methods as in Pregel, e.g. getVertexValue(), getMessages(), getOutEdges(), modifyVertexValue(), sendMessage(), and voteToHalt(). The workers run as database UDFs and typically there are as many parallel workers as the number of cores on the machine or nodes in the cluster. Conceptually, the worker is analogous to *mapper*, which is the container for *map* function in MapReduce. As shown in Figure 1, the workers invoke the actual vertex program provided by the user as well as propagate the output of the vertex program, such as new vertex values and the messages, to the coordinator.

Vertex Computation. Finally, the vertex computation is the user provided graph query that runs once per superstep for every vertex that has at least one incoming message. Typically, the vertex-compute function gets a set of incoming messages, performs some computation, modifies the vertex state, and outputs a set of outgoing messages. This is a simple yet powerful model for graph analytics.

2.3 Optimizations

Vertexica applies several optimizations to make the vertex-centric computations efficient. Let us look at some of these below.

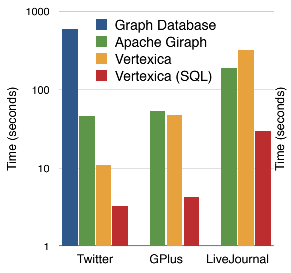
Table Unions. In order to run the vertex computations, Vertexica needs to read all vertices having incoming messages, the corresponding outgoing edges, and the messages themselves. Traditional database wisdom will tell us to simply join the vertex, edge, and message tables. However, for large graphs and/or for large number of messages (every vertex could send a message to every other vertex in the worst case), this three-way join could be very expensive and kill the performance. Since we only need to extend the message table with the vertex values and edges of every receiving vertex, we can union the three tables instead of joining them. These three inputs are renamed to a common schema and the union is then fed to the workers, which are responsible for parsing and identifying the message, vertex, and edge tuples from each other.

Parallel Workers. Vertexica exploits multiple cores or multiple machines by running multiple instances of the worker in parallel and speed-up the computation. In practice, we have as many workers as the number of cores. After each superstep, Vertexica synchronizes the workers during the synchronization barrier.

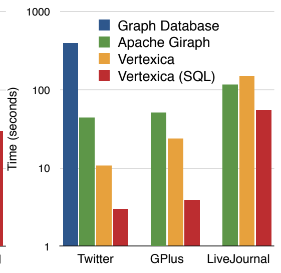
Vertex Batching. The extreme case could be to run each active vertex in a different worker. However, this leads to many UDFs calls, which are relatively expensive in most database systems. Therefore, Vertexica batches several vertices together and runs them serially on a worker, i.e. it tries to balance serial and parallel execution of vertex-programs. To do this, Vertexica hash partitions the table union on the vertex id into a fixed number of partitions. Each partition is sorted on the vertex id. The worker is responsible for identifying different vertices in the partition and executing the vertex program on each of them serially.

Update Vs Replace. Vertexica involves two kinds of updates: (1) updating vertices, e.g. vertex value, and (2) updating messages, e.g. deleting old messages and inserting new ones from the current superstep. However, vertex-centric computations could easily incur large number of updates and can slow down the performance significantly. To handle this, instead of updating the vertices and messages in the existing tables, Vertexica creates new vertex and message tables. This is done by left joining the old vertex/message table with the new vertex/message values from the current superstep and replacing the old table with the new one. Such modifications via *replace* are much faster. Still, if the number of updated tuples is below a fixed threshold, then Vertexica updates the existing tables.

Let us now compare the performance of Vertexica with Apache Giraph [3], a popular vertex-centric graph processing system, and a transactional graph database system. Figure 2 shows the performance of graph database, Giraph, and Vertexica for two popular



(a) PageRank



(b) Shortest Paths

Figure 2: Vertex-centric Graph Algorithm Performance.

graph algorithms, PageRank and Shortest Paths, over three different graphs, Twitter (81*K* nodes, 1.7*M* edges), GPlus (107*K* nodes, 13.6*M* edges), and LiveJournal (4.8*M* nodes, 68*M* edges).

We can see from Figure 2 that the graph database runs only for the smallest graph and both Giraph and VERTEXICA outperform it significantly. Furthermore, VERTEXICA outperforms Giraph by more than 4 times on the small graph and has very similar performance as Giraph on larger graphs. Figure 2 also shows the performance of VERTEXICA(SQL), the hand-coded and meticulously optimized SQL implementations of graph algorithms in VERTEXICA. We can see that the SQL implementation in VERTEXICA significantly outperforms all other approaches. Thus, VERTEXICA combines the best of both worlds, an easy-to-use vertex-centric interface having comparable performance as Giraph as well a high performance SQL interface which significantly outperforms all other approaches.

VERTEXICA facilitates analysts to perform a large variety of graph analytics. We outline several such use cases below.

3. USE CASES

Let us now look at the use-cases supported by VERTEXICA.

3.1 Vertex-centric Graph Queries

VERTEXICA supports several vertex-centric graph analysis algorithms, including: (i) **PageRank** – a ranking algorithm to compute the relative importance of every vertex, (ii) **Single Source Shortest Path** – a reachability query to compute the shortest path from a given vertex to every other vertex, (iii) **Connected Components** – also a reachability query to find subgraphs in which any two vertices are connected to each other, and (iv) **Collaborative Filtering** – a recommendation technique to predict the edge weights in a bipartite graph. In general, VERTEXICA supports any graph algorithm which can be expressed as message-passing iterative vertex-computations.

3.2 Hybrid Graph Queries

While the vertex-centric computations works well for graph queries accessing local neighborhood, they do not work very well, if at all, for queries which involve 1-hop neighborhood. This is because the vertex-centric approaches needs to first collect the 1-hop neighborhood, which is expensive both in terms of time and memory, before doing the actual analysis.

VERTEXICA allows analysts to easily combine vertex-centric analysis with 1-hop analysis. For example, the analyst may want to find all nodes which act as ties between otherwise disconnected nodes and have PageRank greater than a threshold, i.e. find sufficiently important nodes which act as bridges. Similarly, the analyst may want to compute the single source shortest path with the source node being the node with the maximum local clustering coefficient, i.e. find the distance from the most clustered node to every other node. VERTEXICA supports several 1-hop algorithms for such analysis:
Triangle Counting: count the number (total or per-node) of triangles. This could be used for computing clustering coefficients.
Strong Overlap: find pairs of nodes having strong overlap between them. *Overlap* could be defined as number of common neighbors.
Weak Ties: find nodes which act as bridges between otherwise disconnected pair of nodes.

Such analysis is very difficult or even not possible on traditional graph processing systems.

3.3 Dynamic Graph Analyses

Graphs are not static in nature. Over time, we need to add, remove, or modify the nodes and edges in the graph. Similarly, we may need to update the metadata associated with the nodes and edges. VERTEXICA is naturally suited to handle updates and therefore allows for dynamic graph analysis. However, graph processing systems, such as Giraph, have no clear method of updating the graphs it analyzes. One might think of using HBase for updates and Giraph for analytics, but then we have the additional overhead of stitching two systems together and coordinating between them.

Using VERTEXICA, an analyst can easily do graph mutations, metadata updates, as well as perform temporal analysis. For example, the analyst may be interested in knowing the nodes whose PageRanks have changed over last one year, or to see all node-pairs whose shortest paths have decreased by at least a threshold. Such dynamic graph analysis allows VERTEXICA users to treat graph analytics as a continuos process rather than an offline one-time activity.

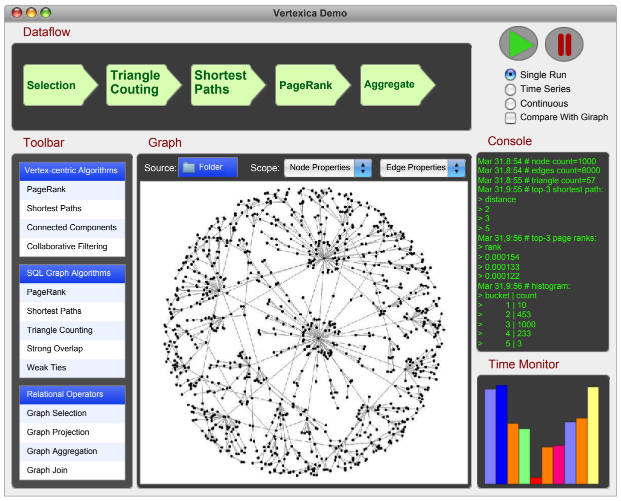
3.4 Richer Graph Analytics

Real-world graphs have vertices and edges accompanied by rich metadata, e.g. vertices may describe each person in the social network and edges may be of types family, friends, or classmates. 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 select a subset of the graph before running the actual graph algorithm. Similarly, he may want to compute aggregates or build histograms over the output of the graph analysis. Furthermore, in many cases, the graphs may be implicit in the relational data and need to be extracted in the first place.

Above pre- and post- processing of graph data would typically require relational operators such as selection, projection, aggregation, join, something which VERTEXICA inherits by default. As a result, VERTEXICA allows analysts to easily combine graph analysis with relational analysis and produce more complete data processing pipelines. Thus, graph analytics on VERTEXICA is not just running a particular graph algorithm on the bare graph skeleton, rather it includes the end-to-end data processing, starting from raw data and right up to deriving meaningful insights.

4. DEMONSTRATION

In this section, we describe our demonstration proposal and show how the audience can play with VERTEXICA.
Hardware. The demonstration runs on 4 machines, each with 2GHz Xeon with 2-socket, 12-core, 24-threads, with Hyper-threading, 48GB of memory, 1.4T disk, and running on RHEL Santiago 6.4.
Datasets. We use several social network graphs of different sizes and both directed as well as undirected from `http://snap.stanford.edu/data/`. These include graphs from Twitter, GPlus, LiveJournal, Amazon, and Youtube, with sizes varying from a few million edges to a billion edges.
Metadata. We also extend the graph datasets 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 distribution) 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



```
Mar 31,8:54 # node count=1000
Mar 31,8:54 # edges count=8000
Mar 31,8:55 # triangle count=57
Mar 31,9:55 # top-3 shortest path:
> distance
> 2
> 3
> 5
Mar 31,9:56 # top-3 page ranks:
> rank
> 0.000154
> 0.000133
> 0.000122
Mar 31,9:56 # histogram:
> bucket | count
>        1 | 10
>        2 | 453
>        3 | 1000
>        4 | 233
>        5 | 3
```

Figure 3: VERTEXICA Graphical User Interface.

type (friend, family, or classmate), chosen uniformly at random.

4.1 Audience Interaction

In our demonstration, we let the audience interact with VERTEXICA via a graphical user interface. Figure 3 shows a snapshot of the interface. It has six main components:

Graph Visualization. The interface allows users to load any raw graph data and visualize the nodes and edges in the graph. Users can interact with the visualization by clicking on the nodes and edges and performing actions such as viewing and/or modifying the properties, e.g. node value, edge weight, etc., of a node or edge. Users can also customize the visualization to show one or more properties for all nodes and edges by default.

Scope of Analysis. Apart from visualizing, users can also select portions of the graph for analysis, i.e. they can interactively define the scope of their graph analysis. This includes visual selection by clicking on one or more nodes or by drawing a minimum bounding rectangle. Alternatively, users can also apply filters based on node/edge metadata, e.g. select all edges of type "Family". By defining the scope of analysis, users can easily specify their regions of interest.

Toolbar. We allow the users to play with several graph algorithms and operators. The *toolbar* provides four vertex-centric algorithms (PageRank, shortest paths, connected components, collaborative filtering), five SQL graph algorithms (PageRank, shortest paths, triangle counting, strong overlap, weak ties), and four relational operators (selection, projection, aggregation, and join). These algorithms and operators allow for a very rich graph analysis.

Graph processing pipelines. Users can not only run a standalone algorithm, but they can also compose a set of algorithms and operators into a graph processing pipelines. The *Dataflow* panel in Figure 3 is used for designing the graph pipelines. Users can drag and drop the algorithms/operators, chain and combine them, and even inspect and modify the code of each of them.

Running mode. After selecting the graph and picking the algorithm (or pipeline), users can run their analysis as: (i) a single run, (iii) time series run to see how the results change over time, and (ii) continuous run to monitor how the analysis changes in response to changes such as graph mutations. Also, the users can compare the query runtimes with Giraph, in case of vertex-centric algorithm.

Output Display. Finally, the interface produces two outputs from the user interactions. First, we show the actual output of the graph analysis, e.g. outputting the shortest path, triangle counts, etc., on the *console*. Second, we plot the time taken to run the analysis in the *time monitor*. In case of continuous analysis, both the console and the time monitor show running results.

4.2 Demonstration Scenarios

We invite the audience to play with several demonstrations scenarios of VERTEXICA. We describe some of these below.

4.2.1 Interactive Graph Analysis

Since VERTEXICA has very good runtime performance, it allows the audience to perform interactive graph analysis on small to medium-sized graphs, i.e. around 10s of millions of edges. The audience can interactively select the regions of interest in the graph by click on nodes or by drawing a rectangle. They can also set the scope of their analysis by considering only those nodes and edges that have specific properties (e.g. edge types). Once a region or a subgraph is selected, users can interactively run the graph algorithms as well. For example, users can click on a node and ask for its PageRank, or the number of triangles that the node participates in. Internally, this triggers a PageRank or a triangle counting query over the region of interest and produces the result in the console. Similarly, users can click on two nodes and ask for the shortest path between them or check if the node-pairs have strong overlap between them.

4.2.2 Complex Graph Analysis

In addition to running the graph algorithms, the audience can pre- or post- process the graphs, such as applying selection predicates to select ad-hoc subgraphs, assigning weights and/or labels to nodes and edges, and computing statistics on the output of graph algorithm. For example, the users might be interested in looking at the distribution of PageRank values. Additionally, the audience can combine multiple graph analyses. For example, users might combine PageRank and shortest path to emit nodes which are either very near (path distance less than a threshold) or are relatively very important (PageRank greater than a threshold). Similarly, the users can ask for global clustering coefficient (combining triangle counting with weak ties). By combining and composing different graph algorithms and operators, VERTEXICA allows users to play with much more sophisticated graph analysis.

4.2.3 Continuous & Time Series Analysis

We invite the audience to describe a graph analysis task and run it in a continuous mode. Thereafter, the users can change the scope of the graph analysis, e.g. change the edge filter from "Family" to "Classmates", and observe how runtimes and the console output changes. Furthermore, users can also click and modify nodes and edges and observe the impact of change on the graph analysis.

Similarly, we invite the audience to compare how the graph analysis has evolved over time. For example, users might ask how the PageRank of a given node has changed in the last 5 years. Or, which nodes have come *closer* (smaller path distance) in the last one year. Such analysis will internally trigger graph algorithms on different versions of nodes and edges. VERTEXICA makes these possible.

5. REFERENCES

[1] N. Bronson and al. TAO: Facebooks Distributed Data Store for the Social Graph. *USENIX ATC*, 2013.
[2] FlockDB, http://github.com/twitter/flockdb.
[3] Apache Giraph, http://giraph.apache.org.
[4] Y. Low and al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 2012.
[5] G. Malewicz and al. Pregel: A System for Large-Scale Graph Processing. *SIGMOD*, 2010.
[6] S. Salihoglu et al. GPS: A Graph Processing System. *SSDBM*, 2013.
[7] B. Shao and al. Trinity: A Distributed Graph Engine on a Memory Cloud. *SIGMOD*, 2013.
[8] A. Welc and al. Graph Analysis Do We Have to Reinvent the Wheel? *GRADES*, 2013.