Graph Processing: Clash of engines

BigData processing project.

Clash of Engines
8-13-2018

Contents

The Problem

In the recent years there has been a great increase in the importance and sizes of graph data. Billion and trillion sized graphs need to be processed to give some crucial insights in different domains and applications. How to represent, store, and process this data is still a vital question.

On the other side there has been a corresponding increase in the number of the offered solutions and graph processing platforms. According to some surveys more than 60 graph processing platforms have emerged since 2003. These platforms are diverse, each group uses a different architecture, storage, communication, and computation model. Each works very well with some types of applications and fail with others.

In this project we're trying to answer two questions:

- Do we need all these platforms?
- How to choose?

From all the available platforms we choose 2 platforms to compare, one general purpose database system (MonetDB), and one specialized graph processing engine (Giraph).

We then define an evaluation and comparison criteria. Then we report the results of the experiments done with these two platforms. Finally, we give some recommendations to guide anyone interested in making a choice between the two systems for practical purposes.

Platforms:

MonetDB:

MonetDB was designed to serve analytical purposes, to work on big workloads efficiently on modern hardware. MonetDB stores data by column which allows it to use **storage optimizations** not possible for row-oriented databases. MonetDB also process data column by column and exploits **bulk processing** and **late materialization**[1].
Other key points of MonetDB architecture:

- it uses processing model, that minimize CPU cache misses rather than IOs.
- It's a **dynamic**, **adaptive** system where index selection, creation and maintenance are automatic.
- It uses **full materialization**[2]**.**

[1] "Late materialization allows columns to be kept in a compressed representation in memory, whereas creating wider tuples generally requires decompressing them first."
[2] "For example, if a select operator consumes its complete input column in one go, then it needs to materialize a result which represents all qualifying tuples, resulting in a significant overhead especially as we scale to bigger data inputs. Together, these aspects make MonetDB vulnerable to swapping when its working set starts exceeding RAM."

To handle updates, MonetDB uses a collection of pending updates columns for each base column in a database. Every update action affects initially only the pending updates columns, i.e., every update is practically translated to an append action on the pending update columns. Every query **on-the-fly merges updates** by reading data both from the base columns and from the pending update columns. Periodically, pending columns are merged with their base columns.

Giraph:

Giraph uses the Bulk Synchronous Parallel (BSP) model, a parallel programming model. BSP is a vertex-centric programming model where the computation on vertices are represented as a sequence of **supersteps** with synchronization between the supersteps. In particular, in each iteration, every vertex that is involved in computation,
1) receives its neighbors updated values from previous iteration.
2) the vertex finishes computation using received values, and possibly updates its value.
3) sends its updated value to its adjacent vertices, that will be available to them in the next superstep.
Each superstep ends with a waiting phase, synchronization barrier, that ensures that messages sent from one superstep are correctly delivered to the subsequent step.
In each superstep, a vertex may vote to **halt** (inactive status) if it does not receive any message and it can also be **re-activated** once it receives a message at any subsequent superstep. Thus, in each superstep, only the **active vertices** are involved in the computation process results in significant reduction in the communication overhead. The whole graph processing operation terminates when all vertices are inactive and no more messages are in transit between the vertices of the graph.

The Evaluation Criteria:

To be able to compare two different platforms we need to define what are the most important measures for us. Many factors affect the performance of a platform on a given graph processing task. Among these there are: the type of algorithm, the properties of the graph to be processed, and the available processing environment. Here we try to capture the effect of each of each combination of these factors.

Algorithms:

There is a great number of graph processing algorithms. To be able to compare them and come up with results applicable to any new problem we need to classify them into categories each having common features. The graph processing algorithms can be classified according multiple classification criteria. Some of them classify the algorithms according to the type of task.

But the classification that might be the most effective in the parallel graph processing scheme, is the classification according the amount of communication, and computation done by the algorithm.

Weakly Connected Components (WCC):

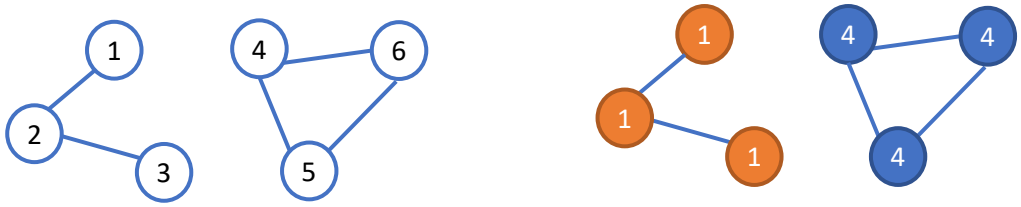


Figure 1. WCC applied to a simple graph. The graph has two connected components. Within each component the smallest node id is propagated until all the nodes in the component receive this value. So, the first component gets id 1 and the second gets id 4.

Two vertices are said to be connected if there is a path between them; i.e. they are reachable from each other. In an undirected graph, a connected component is a subgraph in which any two vertices are reachable from eachother. A connected graph has exactly one connected component which is the graph itself. A vertex with no edges forms a connected component by itself. Finding the connected components in a graph has many applications. For example, finding connected components in a social network helps to identify the different communities of people who share the same interests. Other applications include clustering and graph indexing.
The algorithm works by propagating the smallest id available in each connected component to all the nodes in this component. At each superstep all the active vertices receive the messages sent from the adjacent nodes, find the minimum id in these messages, compare it to its current value, if smaller it updates the value and send new messages to its neighbors, otherwise it will vote to halt.

The computation continue until all vertices are inactive.

Breadth First Search (BFS):

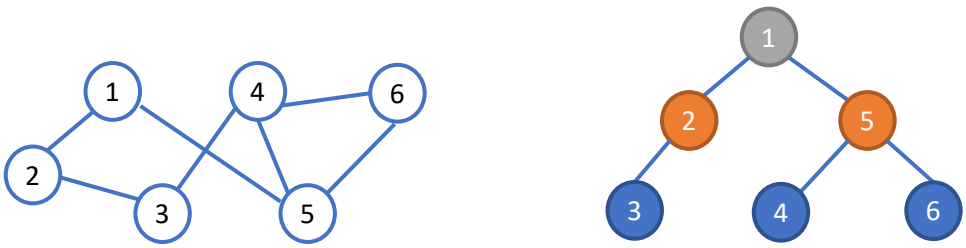


Figure 2. BFS applied to a simple graph. The algorithm starts at node 1 and proceed level by level until it explores the whole graph. On the right we see vertices within the same level with the same color.

Breadth-First Search starts from a source node, explores the traversal tree of the graph one level atime moving from each node to its children, and give the distance to a specific target node, or to all the nodes in the graph. In unweighted graphs, BFS gives the shortest distances from the source node.

In the first iteration every single vertex that is not the source node should vote to halt (become inactive). The source node is recognized by its vertex Id. So, the only vertex that will do any communication in the first superstep is the source node.

The source node, at the end of the first superstep, sends messages to all adjacent nodes and then vote to halt. Receiving a message will make the receiving nodes become active again.

Each vertex value is assigned the maximum possible value in the first superstep. When that vertex becomes active, that means it received a message. The now active vertex checks its current value. If it is the maximum value, it changes that value to the current superstep number – since in each superstep, a new depth/level is explored – otherwise it just votes to halt because it has already been explored in a previous superstep.

How the two algorithms are different:

- **Vertex activation:** BFS vertices are active for one iteration only, the one corresponding to their level in the traversal tree, then after updating their values and going inactive they never become active again.
  WCC vertices can go active/inactive multiple times with no clear pattern to be expected.
- **Worst case scenario; Maximum number of iterations:** In BFS the maximum number of iterations is the diameter of the component of the source node, i.e. depends on the local structure around the source node.
  While WCC's maximum number of iterations is the largest diameter of all components in the graph, so it depends on the global structure of the graph.
- **Complexity of communication**:
  BFS vertices don't do any computation, just update their values according to the superstep number, and send messages to the neighbors. Also, since the number of nodes in traversal tree levels grow exponentially in the number of level, BFS may cause, at one iteration, an explosion in the number of messages and active nodes.
  While WCC communication is quite balanced and doesn't increase or decrease radically between the iterations.
- **Complexity of computation and the effect of density:**
  At any step WCC vertex iterate through all the messages received from the adjacent vertices and compare them to find the minimum. So, it does a linear time computation that grows with the number of the node's neighbors. That's why WCC is severely affected by the average density of the graph.

Datasets:

The chosen datasets:

Dataset	*Edges*	*Nodes*
Livejournal	86M	4.8M
Orkut	234M	3M
Friendster	3.5B	50M

Data format and preparation:

The data of each graph is originally represented as an **edges list**. Each edge is a line of two numbers: source_node_id and target_node_id. The graph is **symmetrized** by adding a reverse edge from target to source node corresponding to each original edge. This way the graph is processed as an undirected graph.

Since Giraph reads the raw files at every run of the algorithm, after some experiments I found that representing the graph as **adjacency list**[3], instead of edges list, save space and loading time considerably (about 50%). However, this format can't be used for loading data into MonetDB, so it was only used for Giraph.

In MonetDB the graph was represented as two tables: nodes table, and edges table with structure in fig-1.

1. Node Table: Node_id and CC is the id of the component of this node, DIST is the distance computed using BFS. Edge Table: SRC and DST are the IDs of the the endpoints of the edge.

Node_id	CC	DIST
3000	1	40
2345	2	20

SRC	DST
3000	3321
2345	1020

Metrics:

The metrics I measured are listed in the table below.

[3] https://en.wikipedia.org/wiki/Adjacency_list

Metric	*Description*
Computation time	The time spent in pure computation. (Sec)
IO Time	The overhead time of reading/writing data (Sec)
Memory	Memory utilization, RAM and Virtual (GB)
CPU	Average CPU utilization %

Experiments:

The Environment:

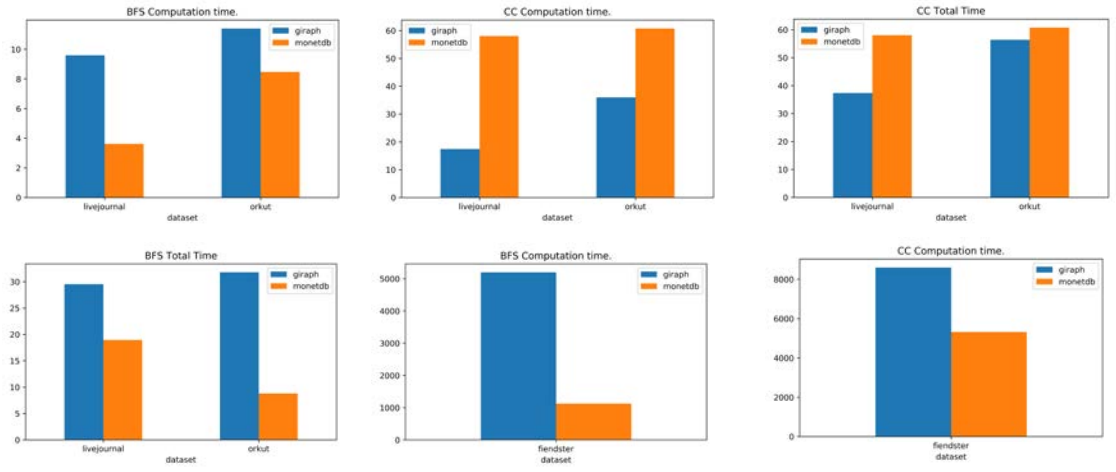
For running the experiments, we used AWS EC2 machine 2.4 GHz Intel Xeon® E5-267 processor, with 64 GB memory and 16 cores. The operating system is Ubuntu 14. Disk space exceeds 200GB.

The software used is mainly Giraph version 1.2, Hadoop 2.0.2, MonetDB 11.29.7.

For processing data, I used python 2.7, and the data processing stack: pandas, numpy, scipy, snap, matplotlib.

Results:

Processing Time:



In the figures above the computation time of the two algorithms are shown in the first two plots. Then the next two plots show the total time, i.e computation time + input/output and any other overhead. Finally, the last two plots show the time for Friendster dataset, plotted separately because of the difference of scale. While Giraph computation time is quite fast, it has a huge overhead that limits the advantages of the short processing time. When the computation time is so long, we can afford to ignore such a big overhead. Otherwise, it's a real disadvantage for short/moderate processing tasks.

The pure computation for MonetDB on BFS is well below that of Giraph, while it jumps suddenly to exceed Giraph time with CC algorithm. The next section tries to explain the reason for this.

The effect of algorithm properties:

As you can see in the figure above the computation time of MonetDB for BFS is smaller than Giraph time. But then there's a great increase in the time difference when we check the CC plots.

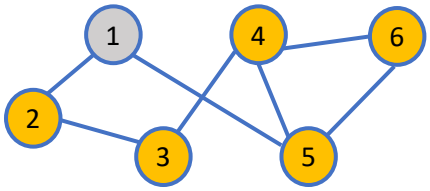
I propose that this is the effect of the algorithms properties and how these interplay with the platform properties.

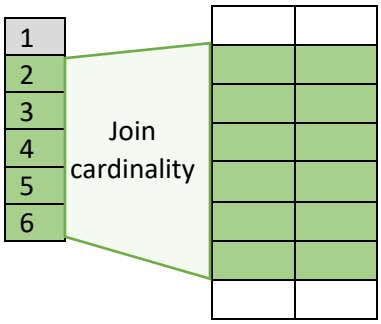
The CC algorithm works by propagating the smallest ID in each connected component of the graph. Most of the vertices are always active, sending messages, and updating their values. The number of messages from iteration to iteration is fairly stable and doesn't change much. **There are two reasons for the computation cost: join cost and update cost.**

MonetDB implements graph algorithms using three-way-join between the node, edge, node tables. If the vertices are always active that means that the join cardinality at each iteration is high. Since The join operation is **expensive**[4], the computation time for CC increases more in the case of MonetDB than Giraph. For example, in the figure below we see the second iteration of CC on a simple graph. All the vertices in yellow are active at this point, while the ones in grey are inactive and the number decreases slowly.

Also, there's the cost of the updates at each iteration. MonetDB is **read-optimized** platform, that means that transactions performance is not a priority. So, making a lot of updates at each iteration is inefficient with such a platform. **One common optimization** is to drop the nodes table and create a new table at each iteration, since writing the table sequentially might be faster than making updates to compressed data and merging these iteratively. This optimization has added a small percentage improvement in performance, so it might need additional optimization. But it doesn't change the performance characteristics extensively.

[4] "Unordered positional lookups are problematic since extracting values from a column in this unordered fashion requires jumping around storage for each position, causing significant slowdown since most storage devices have much slower random access than sequential."
For complete explanation of the cost of the join operation in a column-oriented database check:
The Design and Implementation of Modern Column-Oriented Database Systems.





On the other side, in the BFS algorithm because of traversing the graph like a tree starting from one source every vertex is active only for one iteration and the number of active vertices grows (and decays) exponentially with the number of iteration. That means that for most iterations the join cardinality for MonetDB is small, and great only for a few iterations. That also mean that in these few iterations where there're a huge number of messages, Giraph sometimes crash when there's no free memory to hold this sudden flood of messages.

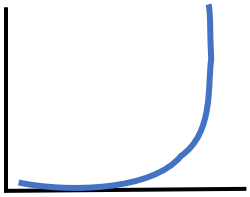
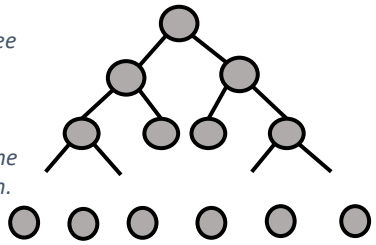


Figure 4. BFS number of active vertices with iterations

Figure 3 traversal tree of the graph, the number of active vertices grows exponentially with the tree level or iteration.



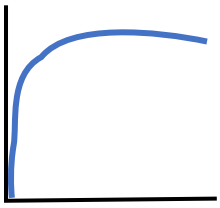


Figure 5 WCC number of active vertices with iterations

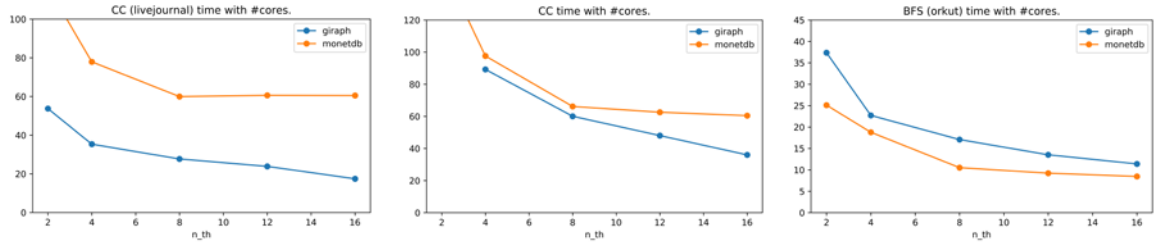
The effect of dataset properties:

Some of the experiments also indicated the effect of the graph **diameter**[5] and **density**[6] on the WCC algorithm. The average density affects the join cardinality and increasing graph diameter increases the number of iterations and the total time.

The dataset properties usually change with the domain of application at hand, e.g. social network, scientific, etc. For example, social network graphs are known to be much denser than the other types. So, each domain can have different measures for choosing the best platform for a give processing platform.

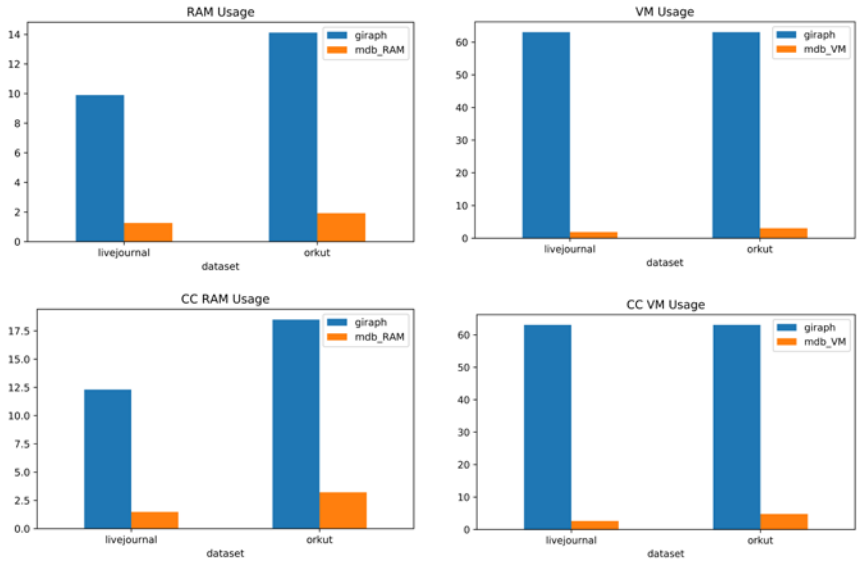
[5] http://mathworld.wolfram.com/GraphDiameter.html
[6] https://en.wikipedia.org/wiki/Dense_graph

Scaling Up:



The plots above show the change of the computation time with the number of cores. The plots and the numbers of average CPU utilization suggest that Giraph scales up better, consuming most the resources available and parallelizes the computation well. The limitation of MonetDB scaling is probably not an inherent feature, but an effect of the kind of queries used. That means that MonetDB put more responsibility on the developer to choose and tune the queries, while Giraph's computation model seems to make things easier. By defining the computation unit as the vertex and its neighborhood, the model makes computation naturally parallelizable.

Memory:



As the plots above show the memory usage of Giraph greatly exceeds that of MonetDB. At any point in time Giraph keeps everything in memory, the graph data, the messages, etc. While MonetDB keeps only the parts of data it's working on and the intermediate queries results. This limited usage of MonetDB is also because of the **compression** techniques it uses both when storing and processing data.

Out-Of-Memory:

Giraph was originally a pure in-memory platform, but that gave it a bad reputation for failing whenever the graph size is close to or exceeds the available memory. That also didn't allow it to process huge graphs. So, a new feature has been introduced, **Out-Of-Core**[7] processing which permits spilling data to disk when needed. I tried to use this feature with Friendster dataset, since it's the only dataset exceeding the available memory. The last two figures of the processing times (in previous section) show that MonetDB BFS time for this dataset was about 20 minutes, while Giraph takes about 2 hours. For CC computation it takes almost 3 hours. By looking at these numbers it's obvious that performance has suffered serious degradation because of exceeding the limited resources available. The OOC feature is an important addition to the platform of course, but it seems the feature is still immature.

Scaling out:

Giraph has well established features to manage the distributed processing of a graph across multiple nodes. However, when I searched for relevant distributed processing features in MonetDB the one feature I found is what is called **Merge Table**[8]**.** This is a logical abstraction around a set of partitions of a table distributed to multiple machines. Partitioning the data and distributing it is the responsibility of the developer and MonetDB manages the queries. Also, this feature allows only for exploring data, but not updating it. So, this feature doesn't provide a solid ground to implement any processing algorithm on top of it, the developer is asked to do most of the job and implement a thick abstraction layer to manage most of the process.

For these reasons I wasn't able to compare the scaling out capabilities of both platforms.

Conclusions:

- MonetDB is fairly efficient database system for graph processing tasks with performance comparable, some times exceeds, a specialized graph engine like Giraph.
- Giraph is easy, flexible platform with programming model familiar to any one experienced with graph processing. It's also efficient and scalable.
- The final performance doesn't depend only on the platform, other factors might be effective as well.

[7] http://giraph.apache.org/ooc.html
[8] https://www.monetdb.org/Documentation/Cookbooks/SQLrecipes/DataPartitioning

Use cases Recommendations

1. **Big graph, limited resources:**
    - If the size of the graph is big relative to the available resources, and if the task at hand is small/medium and intended to run as **stand-alone** on one big machine then MonetDB might be the best choice. Because of how it's capable of managing limited memory space to process big data reliably and fairly quickly, then MonetDB is better choice. While using Giraph means risking failures and low performance because of its inefficiency with limited resources.
    - On the other side if the limited resources available is a **small cluster of machines**, then MonetDB might not be the best choice. Because of its limited and immature distributed processing features, MonetDB will ask the developer to take great responsibility in managing the process including data partitioning, distributing, and updating. Unless the other advantages of MonetDB provide good reasons for taking that responsibility, Giraph is the easier and more flexible choice.

2. **Big graph, Big resources:**
   In that case there might be a need to spend some time to analyze the possible performance of the two platforms and compare it with the required performance priorities.
    - If the priority is **memory or saving resources**, then Giraph might not be the best choice. MonetDB is a better choice for its compact representation of data both in disk and in memory, and its ability to keep only the crucial part of data in memory.
    - If the priority is **computation time** then the performance depends on some other factors not just the platform. It might be beneficial to analyze the properties of the algorithm and the graph and how they affect the performance of each platform.
      An algorithm that involves a big fraction of active vertices at any step doesn't work well with MonetDB. Also, dense or wide graphs degrade MonetDB performance and limit its scalability.
    - Finally, if there's some reasons to prefer using a relational database to take advantage of the other features of data exploration and transformation that go beyond the limited model of graph processing, then MonetDB is a very good choice with performance that competes with both general-purpose databases and specialized graph engines.

extensions:

Possible extensions of this project is to enhance the distributed processing features available in MonetDB to use it for graph processing. Also, testing the interplay between the properties of other algorithms and datasets. Finally, evaluating Giraph distributed performance could add useful advices and recommendations.

References:

1. *The Design and Implementation of Modern Column-Oriented DBs*
2. *Thinking Like a Vertex: a Survey of Vertex-Centric Frameworks for Distributed Graph Processing.*
3. *The Case Against Specialized Graph Analytics Engines.*
4. *Join Processing for Graph Patterns.*
5. *MonetDB: Two Decades of Research.*
6. *An Architecture for Recycling Intermediates.*
7. *Database Cracking - Towards Auto-tunning Database Kernels.*
8. *Graph analytics using vertica relational database.*
9. *The Vertica analytic database: c-store 7 years later.*
10. *How Well do Graph-Processing Platforms Perform.*
11. *A Survey of Parallel Graph Processing Frameworks.*
12. *An Experimental Comparison of Pregel-like graph processing systems.*