Graph Storage: How good is CSR really?

Mahammad Valiyev

December 5, 2017

Abstract

In the last decade, the data size is growing exponentially and processing those data is becoming a difficult problem. Nowadays researchers and industry have been interested in the analysis of the graph to get deep understanding of social networks. Many of these graphs used in industry have become very large, containing hundreds of millions of nodes and edges. In this paper, I propose Compressed Sparse Row(CSR) which is a fast graph container. My CSR implementation also supports updates on graph. On a static graph, CSR gives 1.2-3x better performance over Adjacency List Implementation(AL) on Depth-First-Search(DFS) and Breadth-First-Search(BFS) and up to 2-3x better performance on Dijkstra algorithm(Shortest path).

Keywords: Graph databases, Compressed Sparse Row

1 Introduction

Graph analysis have become more popular in the last few years. Graphs are usually used to represent social networks in which vertices indicate users and edges indicate friendship between users. Facebook has 1.39 billion active users as of 12/2014 with more than 400 billion edges.[1]. Therefore, processing big graphs is consuming more time as the number of users and relationships significantly increased. The proposed approach(CSR) gives slightly better performance over the other two approaches on DFS algorithm. BFS algorithm in general is faster than DFS as it doesn't have an additional overhead of recursion. CSR gives a bit much better on BFS. In the following subsections, I will give some details about CSR and two variants of AL. In the following, I will give some details about existing approaches in Section 2, then followed by implementation details(Section 3) and an evaluation(Section 4). In Section 5 I will give conclusion about how good CSR is really.

2 Related Works

3 Implementation

I have implemented a nice graph interface for all of three graph containers in which graph algorithms can easily be built upon on that. Thus, graph algorithms are independent of the type of graph container. Moreover, modifications on graph containers don't affect the algorithms. For the future work, new algorithms can easily be implemented using the methods of graph containers. In the following subsections, I will briefly explain all three graph containers and show how the edges are actually stored in the memory for sample graph in Figure 1.

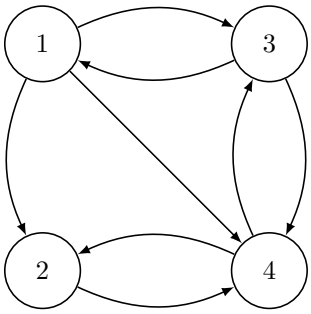

Figure 1: Small sample graph with four nodes

3.1 Compressed Sparse Row

In CSR format, all edges are stored in the same array called `edges` and an addditional `offsets` array is used to get the first neighbor of a node. For each node, the number of neighbors for a node `n` can be calculated `offsets[n+1]-offsets[n]`. Figure 1 shows a small graph with four nodes with few edges and Figure 2 shows the corresponding CSR.

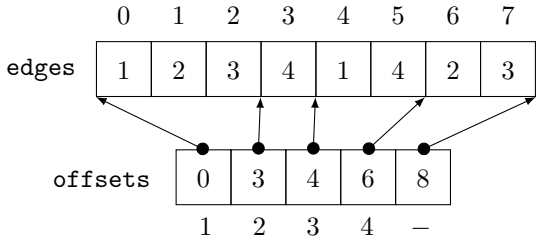

Figure 2: CSR format of the sample graph

All edges are kept densely in the main memory. While iterating over all neighbors of all nodes, the common task in graph algorithms is mostly memory accesses which are usually sequential accesses. As it is a sequential access, the accesses to `edges` and `offsets` arrays are also sequential. It is indeed a beneficial memory access pattern as CPU can easily predict and prefetch. Therefore most of the memory accesses hit the cache. In case of non-sequential access, CSR is not cache-friendly.

3.2 Ajdacency List with `std::list`

In this format, all edges which are outgoing are stored in `std::list`, thus at the beginning for each node a `std::list` is kept. According to the structure of `std::list`, the elements of `std::list` are not located sequentially in the memory. Moreover, it is not cache-friendly at all.

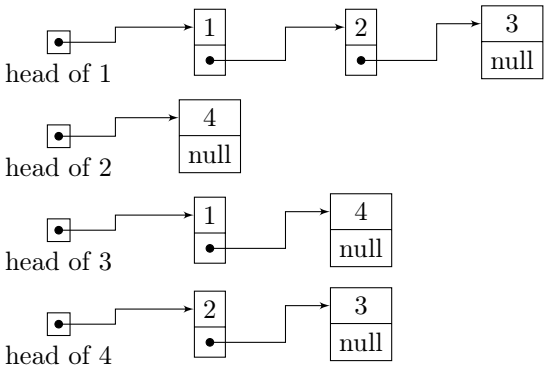

Figure 3: The storage structure of the graph on Figure 1 using `std::list`.

As shown from Figure 3, `std::list` doesn't show any cache friendly structure. Thus, usage of it only gives us better update latency.

3.3 Ajdacency List with `std::vector`

Second approach of Adjacency List uses `std::vector` structure to keep outgoing edges for each node. `std::vector` also uses an array structure to keep the elements, however whenever it hits the capacity, it creates new doubled size array and copies the contents of the old one including the newly added element, then removes the old one. It may seem to be like a big overhead, but in reality it only does `3*n` additional operations for `n` inserts into it.

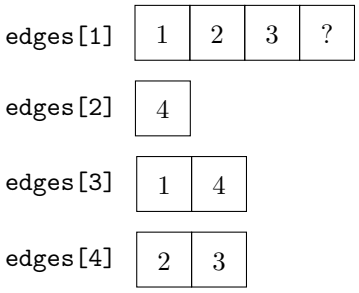

Figure 4: The storage structure of the graph on Figure 1 using `std::vector`.

As it is shown in Figure 4, `std::vector` implementation also has a cache-fiendly structure.

4 Evaluation

All three approches are tested on different datasets which are generated randomly. The number of users differs from 10,000 to 10,000,000. To make it as a real social network, the number of neighbors just differs from 50 to 200. In the following subsections, I discuss in which scenario CSR gives better performance in terms of latency and memory consumption. My test machine is a single socket server computer equipped with Intel(R) Core(TM) i7-3930K CPU 6-core @ 3.20GHz and 64 GB of main memory. This server has 12 hardware cores (with hyperthreading enabled).

4.1 Experimental platform

4.2 Algorithm comparisons

4.3 Complex Scenario

4.4 Memory consumption

4.5 Cache friendliness

5 Conclusion

As it is shown from the results, CSR is not that superior to other two approaches. But considering the fact that social networks are growing so fast nowadays, a slight better approach significantly matters. While most analysis still focuses on static graphs, in the future there will be huge demand on temporal graphs. As CSR itself is not update-friendly, more investigation should be done on the variants of CSR. The CSR library, together with two Adjacency List implementation and generation of graphs, is available online at `https://github.com/mehemmedv/DB_Imp_Seminar`

References

[1] Ching, A., Edunov, S., Kabiljo, M., Logothetis, D., Muthukrishnan, S. (2015). One trillion edges: Graph processing at facebook-scale. Proceedings of the VLDB Endowment, 8(12), 1804-1815