Graph Storage: How good is CSR really?

Mahammad Valiyev

December 10, 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 compare Compressed Sparse Row(CSR) as a fast graph container to Adjacency List(AL) in terms of latency and memory consumption. My CSR implementation also supports updates on graph. On a static graph, CSR gives up to 3x better performance over AL Implementation on Depth-First-Search(DFS) and Breadth-First-Search(BFS) and sometimes 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 approach 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

STINGER [2] is a mutable graph container which is capable of handling updates which can be inserted parallel. it uses a fixed-size chained buckets to keep edges. Due to its loosely synchronized parallel updates that only maintains physical consistency of the data structure, a query may see the graph in a state that never existed. Each bucket contains only edges of the same type.
LLAMA [3] is a new extention to CSR with the support of versioning. It provides consistent snapshots on the graph and allow multiple accesses to them. At first, updates are buffered in a change set and after some period of time creates a new snapshots with the updates on change set. The main drawback is that the method usually has many existing snapshots and accessing to the newer snapshots consume more time. it merges old snapshots and apply a delta to CSR.

ASGraph [4] is also a new extention to which reduce update time to very small latency by breaking up the edge list of CSR into multiple chunks and that support efficient mutation and employing an append-only scheme for updates.

3 Implementation

I have implemented a nice graph interface for CSR graph container and two variants of AL 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.

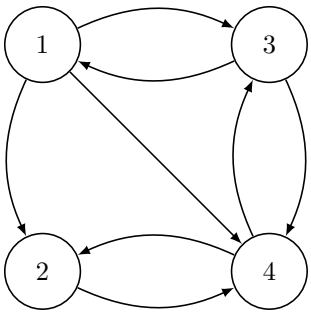


Figure 1: Small sample graph with four nodes

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.

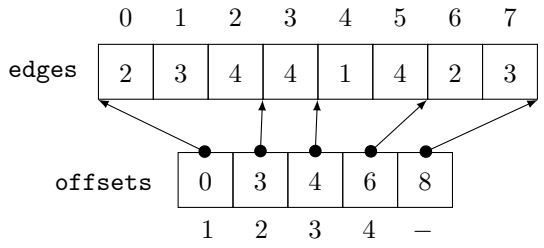


Figure 2: CSR format of the sample graph

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.

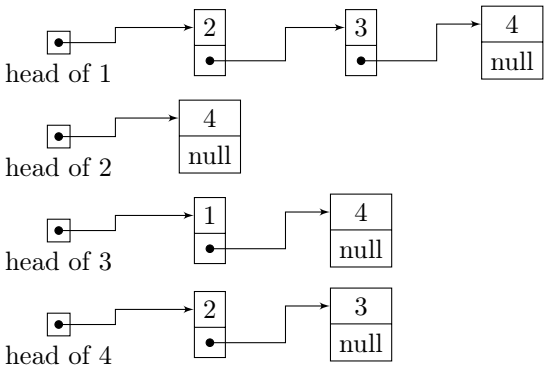


Figure 3: The storage structure of the graph on Figure 1 using `std::list`.

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.

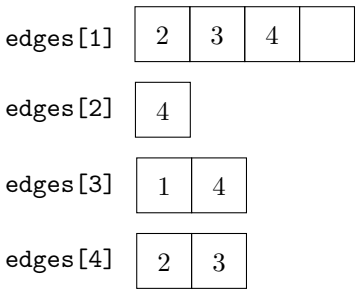


Figure 4: The storage structure of the graph on Figure 1 using `std::vector`.

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 is 1,000,000. To make it as a real social network, the number of neighbors differs from 50 to 200. The neighbors are actually selected

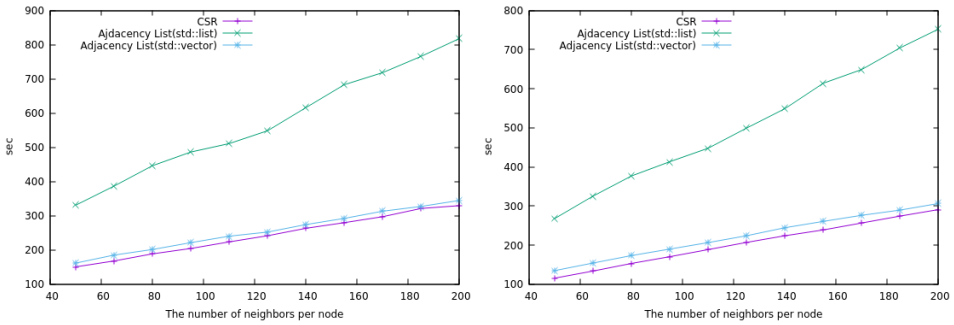
randomly. The graph that I generated is directed and weighted, as I need to run Dijkstra Algorithm. In the following subsections, I discuss in which scenario CSR gives better performance in terms of latency and memory consumption. My test machine is a single socket server computer equipped with Intel(R) 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 Algorithm comparisons

To get a feeling of how fast CSR is compared to Adjacency list, I evaluate three different widely used and simple algorithms that cover a wide-ranging graph access patterns.

Depth-first search and Breadth-first search

DFS is a mainly read-only graph search algorithm which usually reads them sequentially. However on BFS, it also read and write to an additional queue. Theoretically both algorithms should be *O(n)*. However depending on the depth of tree for DFS and the size of queue for BFS, one of the algorithms may perform better than the other one. In the following section, due to the property of sequential read, CSR performs better than Ajdacency List.



(a) Result for DFS

(b) Result for BFS

Figure 5: Results of DFS and BFS for the graph with 1,000,000 nodes.

I test the system with 200 DFS and BFS in which it starts from a random node. CSR is up to 3 times faster than `std::list` approach. For `std::vector`, CSR finishes it at least 15 secs earlier.

Dijkstra Algorithm

Dijkstra algorithm solves the shortest path problem between two nodes in the graph. The complexity of the algorithm is *O(nlogn)*. Despite the fact that it heavily reads the data from main memory, the most time consuming operation is insertion into priority queue which takes *O(logn)*. Thus in the evaluation of Dijkstra, it is difficult to interpret anything from the result.

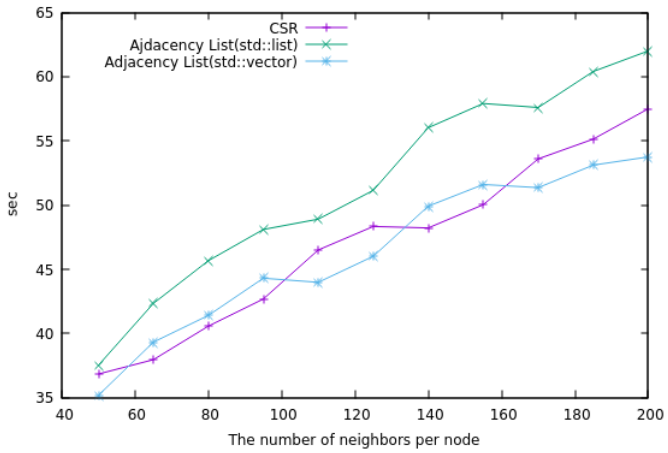


Figure 6: Results of Dijkstra for the graph with 1,000,000 nodes.

CSR is still better on Dijkstra algorithm than `std::list` approach. It is difficult to say whether CSR or `std::vector` is better than the other one.

4.2 Complex Scenario

I have implemented two different update structures for CSR. As it is seen from the structure of Adjacency List on Section 3, updates on Adjacency List is super fast operation which just appends the newly added edge to the corresponding list. It takes much less than a milli second. For CSR, it is too heavy operation to handle. In the following subsection, I will explain how both update structures works and how fast they are.

Simple Update

Assume that the directed graph has V nodes and E edges. To keep the structure of CSR, we need to allocate a new array with $E+1$ edges for each newly added edge. Let's call the new array as `edges'` and the new edge is $e=(v,u)$. First of all, I copy all the content of the array `edges` from `offsets[1]` to `offsets[v+1]` and the rest content of the array `edges` to the array `edges'` starting from `offsets[v+1] + 1` index. Then I set `edges'[offsets[v+1]]=u`. At the end, I increment all the elements from `offsets[v+1]` to `offsets[V+1]`. To copy the content of array, I use `std::memcpy` function which is better than `std::memmove` function.

Light Update

As it is shown from the results of ***Simple Update***, it is the most time consuming operation for CSR format. A new update structure -***Light Update***- is a much faster update method compared to ***Simple Update***.
The new approach is built on top of the previous approach. As an addition, the structure of CSR format should be modified in this method. Assume that we are given a directed graph which has V nodes and E edges. `edges` array should have more than $|E|$, which I call it as *additional space*. For each new upcoming edge, CSR doesn't need to allocate a new big `edges` array to store a new one. Let's call the new edge as $e=(v,u)$. It does a new allocation whenever `edges` array hits the capacity. If it is already full, the update is exactly the same as in ***Simple Update***. If the `edges` array still has some space to store new edges, I copy all the array elements from `offsets[v+1]` into `edges` array starting from `offsets[v+1] + 1` index. As there is intersection between source and destination, I use `std::memmove`

function to do this copy operation. Then I set the corresponding element to `u` and increment corresponding elements of `offsets` array as it is in ***Simple Update***. As it is shown, CSR doesn't do new allocation and copy all the array for each update. The speed-up comes along with having very few new allocations and copying. The latency for ***Light Update*** can simply be twice as less if the graph is unweighted, as I am also updating the weights array for each new edge.

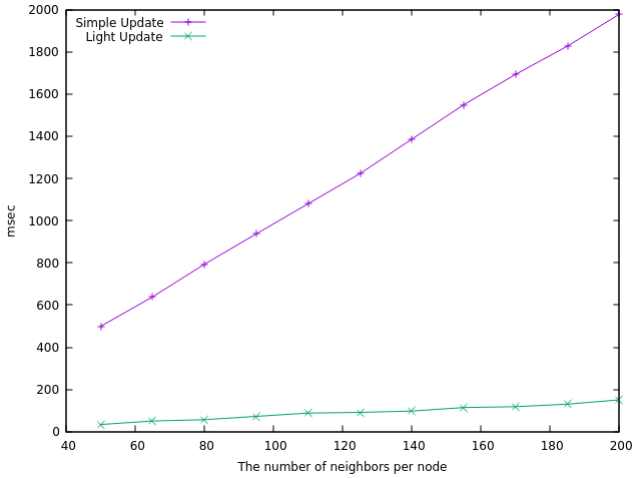


Figure 7: Latencies of both updates for the graph with 1,000,000 nodes.

As it is shown from Figure 7, ***Light Update*** always performs 13-15x faster than ***Simple Update***. This gives us a possibility to use CSR for temporal graphs as well.

Real World Example

I evaulate the system in a mixed workload in which every 20th operation of queries is an update query(***Light Update***) which adds a new edge to the graph. Therefore, it gives us an overview of how good CSR is really on temporal graphs.

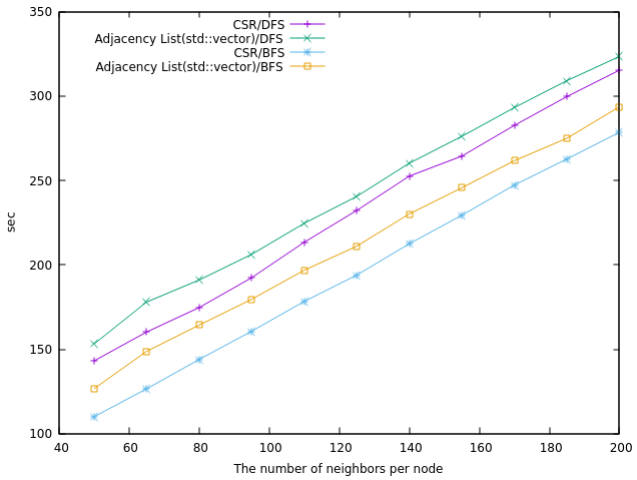
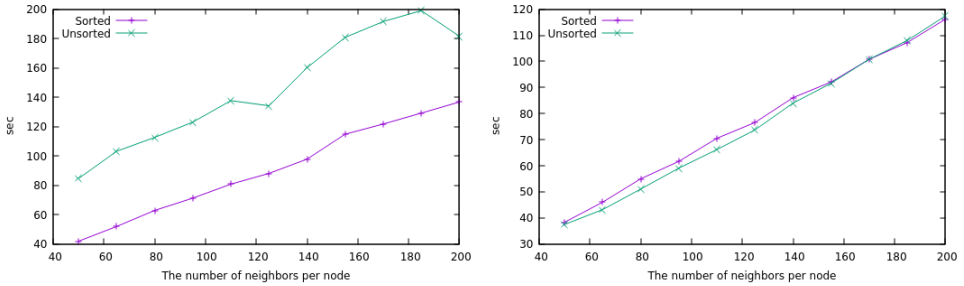


Figure 8: Latencies for mixed workload for the graph with 1,000,000 nodes.

Figure 8 shows that CSR is still performing better in the case of having every 20th operation as update. Thus, CSR can be an option for temporal graphs. However, ***Light Update*** is too heavy operation for much bigger graphs. Further investigation should be done on bigger graphs.

An Interesting Approach to Real World Example

Real-world graphs are usually clustered. Therefore, the likelihood that a neighbor is already in the cache is higher if you access them in a sorted manner. One might label nodes in such a way that neighbors have similar ids. I have generated a graph with 1,000,000 nodes in which each node has neighbors between 50 and 200 and all neighbors have similar ids.



(a) Result for DFS

(b) Result for BFS

Figure 9: Results of DFS and BFS for the graph stores neighbors in sorted.

For DFS, keeping neighbors in sorted way gives us 1.6-2.1x less latency. However, for BFS it doesn't really decrease the time. One can interpret from here that it is always better to keep neighbors sorted in CSR. In order to keep sorted structure, additional overhead is just to do binary search to find the corresponding index for new edge which is just *O(logn)*.

4.3 Memory consumption

So far we only compared CSR with other two approaches in terms of latency. For the real world use, memory consumption should be considered as one of the main property.

# neighbors per node	mem(std:list)÷mem(CSR)	mem(std:vector)÷mem(CSR)
50	7.33	1.41
65	7.46	2.01
80	7.55	1.66
95	7.62	1.42
110	7.67	1.23
125	7.7	1.15
140	7.73	1.85
155	7.76	1.68
170	7.78	1.54
185	7.79	1.42
200	7.81	1.31

Table 1: Ratio of memory consumption of CSR with other approaches.

`std::list` is usually implemented as doubly linked list. Because of this structure it uses much more memory. But according to the structure of `std::vector`, each of them is at least half full for each node, whereas there are some empty places. However CSR has only one memory overhead which is `offsets` array. As shown from Table 1, CSR is always using less memory than the other approaches, at least 7 times less than `std::list` and around 1.15 times less than `std::vector`.

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

[2] Ediger, D., McColl, R., Riedy, J., Bader, D. A. (2012, September). Stinger: High performance data structure for streaming graphs. In High Performance Extreme Computing (HPEC), 2012 IEEE Conference on (pp. 1-5). IEEE.

[3] Macko, Peter, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. "LLAMA: Efficient graph analytics using large multiversioned arrays." In Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pp. 363-374. IEEE, 2015.

[4] Haubenschild, Michael, Manuel Then, Sungpack Hong, and Hassan Chafi. "AS-Graph: a mutable multi-versioned graph container with high analytical performance." In Proceedings of the Fourth International Workshop on Graph Data Management Experiences and Systems, p. 8. ACM, 2016.