CS224W Final Project Report

Jonathan D. Ellithorpe
jdellit@stanford.edu

ABSTRACT

Many tools exist today for generating and processing graphs, including NetworkX, iGraph, and SNAP [4, 3, 5]. These tools come pre-packaged with a library of common graph algorithms that make executing graph computations often as simple as one line of python. These tools, however, are limited to working on graph datasets that fit in local memory. Pregel, Grappa, PowerGraph, and GraphX [8, 9, 7, 11] are examples of systems that address this scaling issue, but only provide the user with high level programming primitives, thus leaving the burden of algorithm implementation to the user. The focus of this work is to explore combining the best of both worlds by extending the SNAP graph analysis library to operate on large-scale graph datasets stored in RAMCloud, a low-latency, fault-tolerant, and scalable distributed key-value store.

1. INTRODUCTION

Graph datasets are continuing to grow in size. Already the Facebook social graph has reached approximately 150 billion edges [6]. At 4 Bytes needed per edge the storage needed for a graph that size is approximately 600GB. Extrapolating to a model of the whole world at 7 billion people and 100 friends per person the storage requirements jump to roughly 5.6TB using 8B edge encoding. In order to process graphs of this magnitude, distributed graph computation frameworks are needed. Pregel, Grappa, PowerGraph, and GraphX [8, 9, 7, 11] are all examples of an attempt to meet that need, however each presents its own high level primitives and force researchers to re-implement the same algorithms over again (e.g. PageRank, BFS, Triads, Clustering, etc.). Graph libraries do exist, like NetworkX, iGraph, and SNAP, however these libraries are constrained to work on a single machine operating on a graph stored in local memory, and are therefore unable to scale.

In this paper we present SNAPCloud, a series of simple extensions to SNAP enabling the platform to store, generate, and process large scale graphs in RAMCloud [10], a low-latency, fault-tolerant, and scalable key-value store. By storing the graph in RAMCloud, SNAPCloud has access to any single element in the entire dataset in microsecond latencies. Multiread and multiwrite features of RAMCloud also enable data streaming for performing efficient bulk computations. Additionally, as a shared storage medium, it opens the possibility for multiple users (or systems) to be interacting with and modifying the same dataset(s) at the same time. Thus, with SNAPCloud, we hope to take a step towards building a tool for researchers that enables simple real-time graph data analysis on large graph datasets that may additionally be shared with other users or systems, and may even be evolving over time.

The paper is broken down in the following sections. In Section 2 we describe the details of the extensions that were made to SNAP for performing graph generation and computation using RAMCloud, and in Section 3 we describe the peformance evaluation of our implementation compared to regular SNAP. Finally in the last section we address the overall current status of the project, challenges that we face, and future work.

2. IMPLEMENTATION

At the core of SNAP is a set of graph objects for representing different types of graphs, including undirected and directed graphs, as well as directed multi-graphs (TUNGraph, TNGraph, and TNEGraph, respectively). Graph objects define an inner Node class storing the node's unique integer ID, as well as a list of its neighbors. The graph object itself stores a hash table of all the nodes in the graph that is keyed by node ID, and provides a set of functions for adding and deleting nodes and edges to this hash table (where edges are implicitly represented as the two neighbors appearing in each other's neighbor lists). In addition, graph objects also define two more inner classes: a node iterator, TNodeI, and an edge iterator, TEdgeI, for iterating over all the nodes and edges in the graph, respectively.

For the purpose of this work we chose to focus on undirected graphs and wrote a new graph type called `TUNRCGraph` which internally uses RAMCloud to store the graph remotely, while still implementing the standard interface for graphs in SNAP. This new RAMCloud-enabled undirected graph class has the following structure:

```
class TUNRCGraph {
  class TNode {
    uint64_t Id;
    vector<uint64_t> Neighbors;
    TUNRCGraph* G; // parent graph ptr
    // Accessor methdos
  }
  class TNodeI {
    TNode Node; // cache the current node
    RAMCloud::TableEnumerator tableEnum;
    TUNRCGraph* G;
    // ++ operator
  }
  class TEdgeI {
    TNodeI CurNode, EndNode;
    uint64_t CurEdge;
    TUNRCGraph* G;
    // ++ operator
  }
  RAMCloud::RamCloud RamCloud;
  uint64_t AdjTable_Id;
  // Methods for modifying and
  // accessing the graph
}
```

The TNode inner class remains the same, and acts as a cache of the currently pointed-to node by a node iterator. Node iterators now have a RAMCloud TableEnumerator object which holds the current state of enumeration in a RAMCloud table. Edge iterators remain the same, keeping a start and end node iterator as well as the current index into the current node's neighbor list to describe the current edge. Finally, the TUNRCGraph object itself contains a RamCloud object through which various RAMCloud operations are performed (connecting to the coordinator, creating and deleting tables, and reading and writing data). The `AdjTable_Id` variable keeps track of the assigned table ID (at table creation time) for storing the graph, which is organized as a table of adjacency lists keyed by node ID.

Instances of these different graphs can be passed to template methods for performing computations on the graph, including counting triads, calculating the diameter of the graph, returning a BFS tree from a source node, etc.

Note that at the current stage of the project, not all elements of the above class and contained inner classes have been implemented. For instance, because RAMCloud's tableEnumerator object is not currently copy and assignable, the `operator=` operator of node iterators is not implemented. Further, because RAMCloud only support instantiating a `TableEnumerator` object pointing to the beginning of a table (and not an arbitrary key in the table), TUNRCGraph's `TNodeI GetNI(const int& NId)` is not implemented. Filling out the full SNAP graph interface in TUNRCGraph is a work in progress.

2.1 Graph Generation

2.1.1 N x M Grid

At the current stage of development we have been able to successfully run the `GenGrid(int N, int M)` grid generation library function on a `TUNRCGraph`. The code for this is reproduced below, generating a 100 x 100 sized grid:

```
PUNRCGraph G;
G = TSnap::GenGrid<PUNRCGraph>(100, 100, false);
```

While the performance of this method was slow as expected (see Evaluation section), we experimented incrementally with three performance improvements, listed in Table 1.

Table 1: Stages of Performance Optimization to GenGrid

Function Name	Description
GenGrid	Vanilla SNAP method
RCGenGrid	Performs one RAMCloud write/node
RCGenGridMW	Utilizes multiwrites
RCGenGridMWMT	Utilizes multithreading

We noted that the vanilla GenGrid method performs 5 reads and 5 writes to RAMCloud (one of the reads is due to a sanity check to make sure that the node does not already exist in the graph) due to the way that the implementation utilizes the graph's `AddNode()` and `AddEdge()` methods, when only 1 write to RAMCloud is necessary per node (when sanity checks are turned off). Building on top of this, since writes to RAMCloud are synchronous, the second performance improvement batches writes together into what's called a multiwrite, which sends a single RPC to RAMCloud containing all the writes and thus ammortizes the overheads of RPC processing and system latencies. Finally, the last performance improvement parallelizes the algorithm by splitting the grid across N threads for writing.

2.1.2 Edros-Renyi Graph

Erdos-Renyi graphs come in two flavors. In the `G(N, M)` model, a graph is chosen uniformly at random from the set of all graphs that have N nodes and M edges [2]. In the `G(N, p)` model, each edge appears in the graph with probability p, or is omitted from the graph with probability (1-p). While the `G(N, M)` model allows one to exactly select the total number of edges, the `G(N, p)` model generates a number of edges according to a binomial distribution:

$$\Pr(M = m) = \binom{E}{m} p^m (1-p)^{E-m}$$

Where E is the total number of possible edges in the graph ($E = \frac{N(N-1)}{2}$) and the expected number of edges is therefore:

$$\mathbb{E}[M] = \frac{pN(N-1)}{2}$$

For the scope of this work we implemented the `G(N, M)` model, and in the section on evaluation explore how to take advantage of RAMCloud's multiReads and multiWrites to improve its performance.

2.2 Graph Processing

2.2.1 Average Clustering Coefficient

The clustering coefficient of a node in a graph measures the connectedness of the node's neighbors. It is defined as the ratio of the total number of edges that connect the neighbors to the total number of possible edges that could exist between the neighbors [1]. Thus, the clustering coefficient for a node `v` is defined mathematically as:

$$C_v = \frac{e_v}{\frac{N_v(N_v-1)}{2}}$$

where e_v denotes the total number of edge between the neighbors of v and N_v denotes the number of neighbors of

Table 2: Cluster Configuration

Part	Specification
CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 1333 MHz
NIC	Mellanox ConnectX-2 Inniband HCA
Switches	5x 36-port Mellanox InniScale IV (4X QDR)

Table 3: RAMCloud Configuration

Parameter	Value
Servers	8
Replicas	3
Backups	8
Total Master Memory	80%
Master Service Threads	4

v. It follows that the average clustering coefficient of a graph is the average taken over all the nodes in the graph.

We implemented this function in SNAPCloud and evaluate its performance in section 3, and explore some ways to improve its performance, and also look at the scaling properties of the algorithm for very large graphs.

2.2.2 *Breath First Search Tree*

A breadth first search starts from a given starting node and explores the graph breadth first moving progressively farther away from the starting node. The algorithm is typically used for finding shortest paths in a graph, or the shortest distance between two nodes, which is a cornerstone algorithm for other algorithms, including calculating the betweenness centrality of an edge in a graph, and finding the diameter of a graph.

We implemented the `GetBfsTree` function in SNAPCloud that returns a directed tree graph starting from a source node. It generates the tree as a separate table in RAMCloud and returns a pointer to that tree to the user.

3. EVALUATION

For the following performance evaluations we used a cluster of 80 machines, each with the hardware configuration listed in Table 2 [10]. A RAMCloud cluster was then configured with the parameters listed in Table 3. Here servers refers to the number of physical masters in the system that can store RAMCloud data in RAM, while backups refer to the number of physical machines that store replicated data. These sets can overlap, and in this case we have set the number of masters equal to the number of backups so that every physical machine acts as both a master and a backup at the same time. The replica parameter is the number of backups to which a segment of data in RAMCloud is stored in case of failure. Finally, we set the master to use up 80% of its local memory for storing RAMCloud data, and set the number of threads for a master to service requests to 4.

3.1 Graph Generation

3.1.1 *N x M Grid*

Here we evaluate the performance of our `GenGrid` code and RAMCloud-aware performance improvements. For these experiments we used a small test graph size of 50 million nodes (10,000 by 5,000 node grid, approximately 2GB in size) to

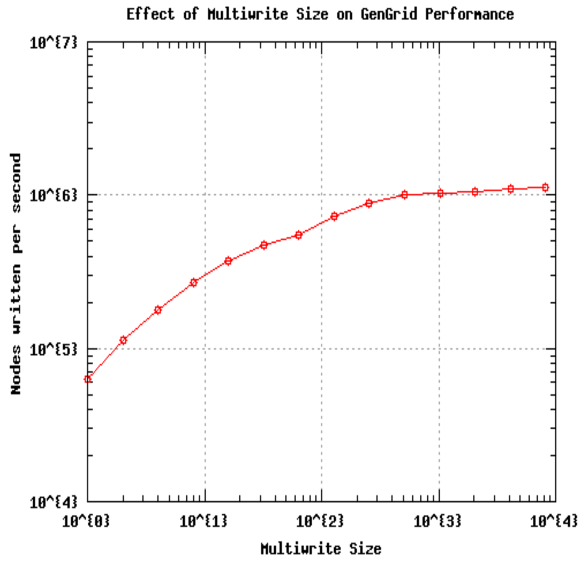

Figure 1: Effect of Multiwrite Size on GenGrid Performance

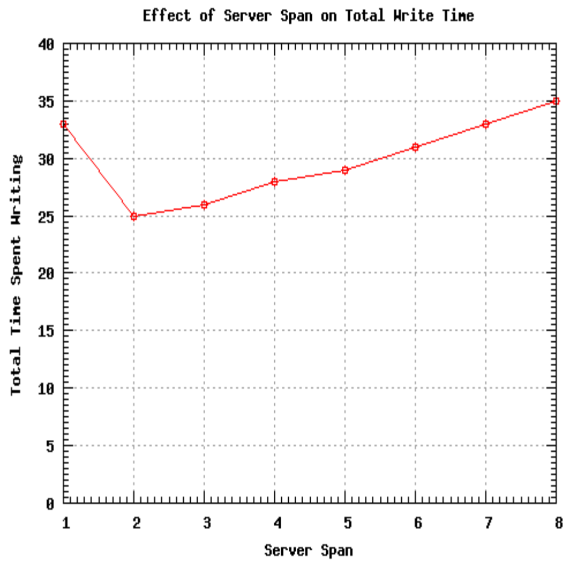

Figure 2: Effect of Server Span on Total Write Time Using Multiwrite Size of 8192

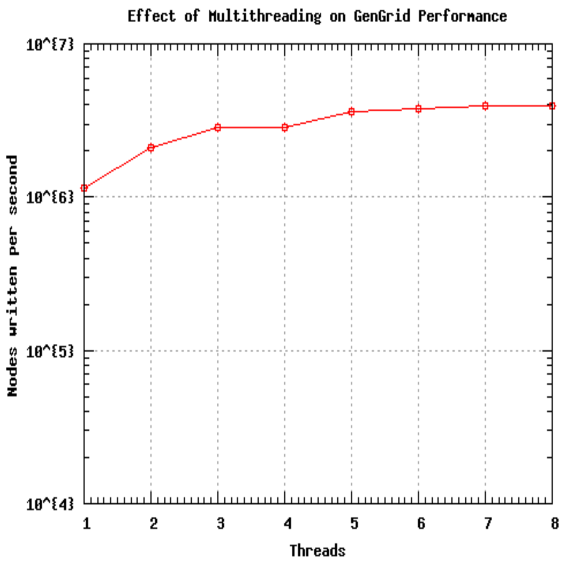

Figure 3: Effect of Multithreading on GenGrid Performance Using Multiwrite Size of 8192 and 6 Servers

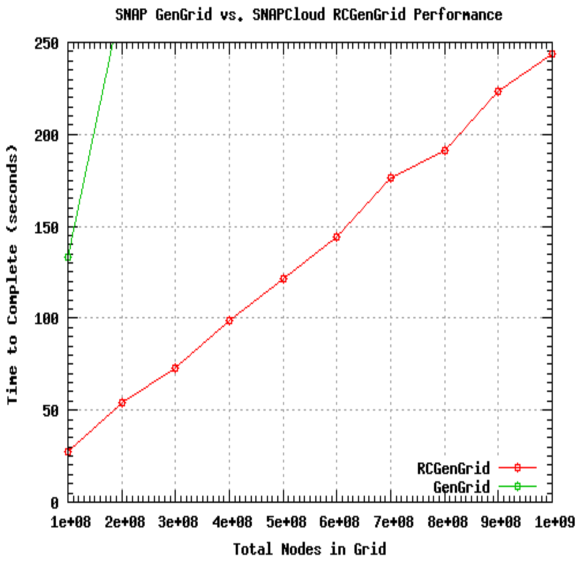

Figure 4: SNAP GenGrid vs. SNAPCloud RCGenGrid Performance

test our algorithms and related performance improvements. For our performance metric we use a simple `nodes written per second` measure, to guage how fast we are able to generate the graph in RAMCloud.

Figure 1 shows the results of exploring the effect of using multiwrites to boost the overall write rate into RAMCloud. Multwrites ammortize the RPC overheads incurred on both the client and the server, and we see deminishing returns past roughly 1,024 writes in a multiwrite. For the GenGrid workload a node has on average 4 neighbors (the majority of nodes in a grid are in the middle of the grid, not on the corners or edges), and since each nodeId is encoded as an unsigned 64-bit integer, on average the neighbor list of node is 32B. Adding onto that the nodeId of the node itself gives a total of 40B for writing a single node of the grid into RAMCloud. For a multiWrite of size 8192 shown in Figure 1 we achieve a `nodes/second` rate of 1,116,445 n/s, which equates to an overall bandwidth of 44.7 MB/s. Digging a little further we found that of the 45 seconds it took to write the graph, 33 seconds were spent performing multiWrites to RAMCloud and 12 seconds were spent constructing the write.

Since we found that 73% of the time was spent writing to RAMCloud, we then turned to trying increasing the server span of the table that stores the graph. For a server span of N, this effectively spreads a multiWrite over the N servers using a hashing function over the set of keys. In expectation, with a multiwrite of size M_s each server will get $\frac{M_s}{N}$ of the keys in the multiWrite. We therefore expected that with a server span of N we should see the total time spent doing writes be divided by N. Figure 2 shows our findings, which were surprising. We see that increasing the server span from 1 to 2 only reduces the total write time by 24%, and increasing the server span further actually increases the total time spent writing. There are a few ways to explain this. One possible explanation is that each write has a certain amount of overhead that comes from constructing the underlying RPCs that are sent to the servers, and therefore increasing the server span is only helpful towards reducing the fraction of time actually spent waiting for the RPCs to return from the servers. As the server span increases, overheads of bucketing the keys and constructing more RPC packets may be contributing to the increase in write time. Lastly, the total time for a write RPC to return to the client is bottlenecked by the slowest server, since the client must wait for all replies to return before returning to the application. Therefore as we increase the number of servers, the more likely we are to be slowed down by a sluggish server.

Noting that in our original measurement with a server span of 1, 27% of the time was spent constructing the nodes and neighbor lists and preparing them for writing into RAMCloud, we decided to explore multithreading the `GenGrid` function. Figure 3 shows our findings, where we fix the server span to be 6. The total write workload is spread evenly among N threads by giving each thread $\frac{1}{N}$ of the rows. We saw that with 8 threads we were able to boost the overall write performance from 1,150,384 to 3,969,322 Node/s for a 3.5x improvement, and a total write bandwidth of 158 MB/s.

Finally, since we used a small 2GB 10,000 by 5,000 node grid for our performance measurements discussed above, we experimented with writing a 40GB 1 billion node grid (10,000 by 100,000 nodes) into RAMCloud, and did so successfully across 6 servers using 6 threads for a total time of 244 seconds and a total write speed of 4,103,627 Nodes/second. Figure 4 shows a comparison between total completion times for graph generation between vanilla SNAP `GenGrid` and SNAPCloud's `RCGenGrid` methods. For grid sizes larger than 200 million nodes the machine runs out of memory (24GB total system memory) and begins to fall over, whereas RAMCloud is able to scale to much larger graph sizes.

3.1.2 *Erdos-Renyi Graph*

For Erdos-Renyi graph generation, the basic algorithm taken from the SNAP library and modified to use an undirected RAMCloud graph object looks like the following:

```
PUNRCGraph RCGenRndGnm(Nodes, Edges) {
  PUNRCGraph Graph = new TUNRCGraph();
  for(uint64_t node = 0; node < Nodes; node++)
    Graph->AddNode(node);

  for(uint64_t edge = 0; edge < Edges; ) {
    uint64_t SrcNId = Rnd(Nodes);
    uint64_t DstNId = Rnd(Nodes);
    if (SrcNId != DstNId)
      if(Graph->AddEdge(SrcNId, DstNId)) {
        // is new edge
        edge++;
    }
  }
  return Graph;
}
```

The algorithm first adds all the nodes to the graph (total of 1 write per node), and then adds edges one at a time by picking two random nodes and adding an edge between them if an edge does not already exist. For an undirected graph this means reading both nodes out of RAMCloud, adding the source and destination node to the destination and source neighbor lists respectively, and then writing the two nodes back to RAMCloud for a total of 2 reads and 2 writes.

We measured the performance of first adding all the nodes to the graph, and found that on average each `AddNode()` call takes approximately 14.4us, which writes an 8B key and a 0B object to RAMCloud. We then measured the performance of adding each edge to the graph and found that on average each `AddEdge(src, dst)` call takes approximately 42.6us, where each write takes approximately 15.4us and each read takes about 5.9us. So in approximation the total time for the algorithm is $T = 14.4us(Nodes) + 42.6us(Edges)$ in the case where $Edges << \frac{Nodes(Nodes-1)}{2}$, meaning that it is unlikely to pick an edge that already exists. If an edge is picked which already exists, this costs us two reads in the `AddEdge(src, dst)` call, or equivalently 11us, each time this happens.

To speedup the first step of the algorithm we wrote a bulk AddNode method to `TUNRCGraph` called `AddNodes(Nodes)` which creates a contiguous set of nodes with IDs 0 to Nodes-1. Its implementation utilizes RAMCloud multiwrites and multithreading, and is able to write nodes at a rate of about 4,300,000 Nodes/second using 6 threads and 6 servers to spread the load across. This changes our above equation to $T = 0.23us(Nodes) + 42.6us(Edges)$.

To reduce the overhead of adding edges we wrote a new SNAP `TUNRCGraph` method called `AddEdges( Edges )` which takes in a list of edges to add to the graph and adds them all at once. The method first takes the set of all nodes touched in the list of edges and performs a RAMCloud multiRead on that set, effecitively caching all of the nodes in local memory. Then the edges from the input edge list are added one by one to the cached set of nodes, while at the same time keeping track of which nodes have been modified since it is possible that the edge list contains edges that already exist in the graph. Finally, a multiWrite is performed onto only the nodes that have been modified. We observed that by adding 1024 edges at a time (caching approximately 2048 nodes at a time from RAMCloud on a graph with nodes much greater than 2048) we are able to achieve about 9.3us on average per edge add, giving us a new performance estimate of $T = 0.23us(Nodes) + 9.3us(Edges)$.

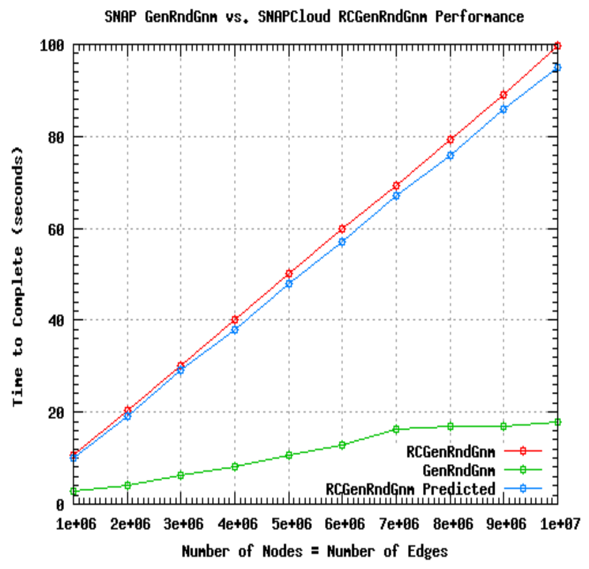

Figure 5: SNAP GenRndGnm vs. SNAPCloud RC-GenRndGnm Performance

Figure 5 shows the performance comparison between vanilla SNAP `GenRndGnm` and SNAPCloud `RCGenRndGnm`. For this performance comparison we set the number of edges equal to the number of nodes and observe the total running time of the method.

We can also see in the figure that our predictor for total completion time is quite accurate due to the scaling properties of RAMCloud (that is, the speed of reads and writes are not affected by the amount of data stored in RAMCloud). Although due to time limitations we did not run the experiment for larger graphs, we expect that it will continue to scale as predicted.

3.2 Graph Processing

3.2.1 *Average Clustering Coefficient*

To calculate the clustering coefficient of a given node in the graph we must read out that node's neighbor list as well as the neighbor list of all its neighbors. The neighbor lists are then searched to see how many edges exist between the neighbors of the given node. Since the time to complete this operation is dominated by read times to RAMCloud, the

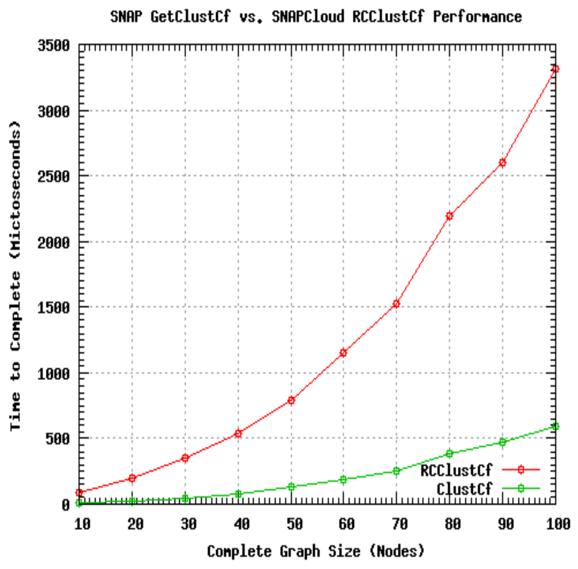

Figure 6: SNAP GetClustCf vs. SNAPCloud RC-ClustCf Performance

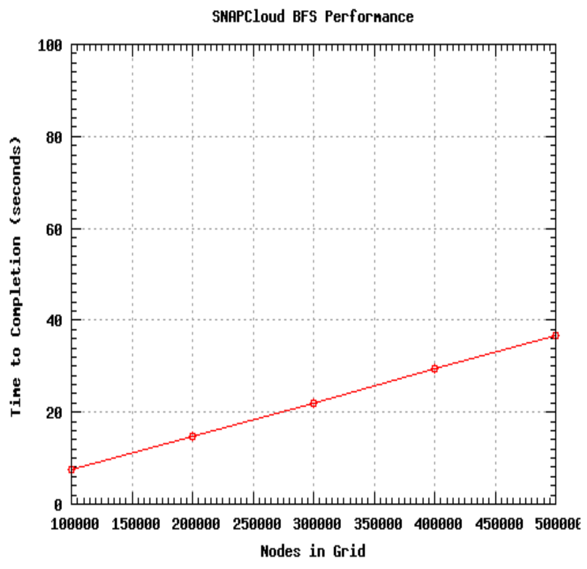

Figure 7: SNAP BFS Performance

overall time to completion can be accurately estimated as the time per read multiplied by the number of reads.

In Figure 6 we evaluate the performanc of the vanilla SNAP and SNAPCloud versions of this function on a single node in a complete graph. We chose to use a complete graph in this performance evaluation to control for the number of neighboring nodes, the factor which most affects the performance of the clustering coefficient algorithm. In a complete graph of size N, every node has N-1 neighbors. Therefore the clustering coefficient algorithm needs to read every node out of the graph for a total of N reads. Every node has the same length neighbor list of size $(N-1)8B$. Since not only are the number of reads inceasing, but also the size of each read proportional to N, we see the quadratic curve in the figure.

While SNAPCloud's performance may not be comparable to SNAP's in-memory hash talbe lookup, it should be noted that 1) multiReads on the set of neighbors has the potential to bring down SNAPCloud's curve by a factor of 10, and 2) the performance SNAPCloud's clustering coefficient algorithm depends only on the number and storage size of the target node's neighbors, while SNAP as an in-memory graph library will eventually begin paging to disk as the graph size exceeds that of local memory.

3.2.2 *Breath First Search Tree*

For our breadth first search algorithm with implemented the standard textbook queue and seen-list based algorithm. It reads nodes out from the graph starting at a source node and explores the neighbors that have yet to be seen, adding them into the FIFO queue. Along the way the breadth first search tree is constructed in a separate RAMCloud table. The algorithm therefore has performance that scales with the number of nodes that are in the graph, as the algorithm reads each node once. Figure 7 shows the results of our performance measurements on a grid structured graph as we increase the size of the grid.

4. SUMMARY OF CURRENT STATUS AND CHALLENGES

At the current stage of development the RAMCloud library has been successfully integrated with the SNAP library, and the basic features of a SNAP graph object have been implemented in `TUNRCGraph` that stores the graph entirely in a RAMCloud table, used for storing adjacency lists in the graph. Some features such as edge iteration and returning a node iterator to a specified node in the graph have not yet been implemented due to current limitations with table enumeration in RAMCloud, which prevents certain functions in SNAP that use this funcionality from working out of the box.

The GenGrid, GenRndGnm, and also GenFull graph generating functions have been successfully implemented in RAMCloud, and performance optimizations to those functions were explored. While for most cases when the graph fits in local memory SNAP's in-memory hash table outperforms RAMCloud's remote key-value store, in some cases we are able to perform comparably, and for all cases where the graph exceeds the size of local memory using local-memory becomes too slow and SNAPCloud's methods are preferred.

Further, we successfully implemented a clustering coefficient algorithm in SNAPCloud and explored its performance on a complete graphs of varying size. While multiRead performance optimization have yet to be implemented, it is expected from previous measurements that such performance optimizations could potentially bring down the total time to completion by a factor of 5-10x. Lastly, an algorithm for finding the breadth first search tree from a given source node was implemented, returning the tree in a separate table in RAMCloud to the user.

The robustness challenges that we faced for much of the project did get resolved inadvertently, and mysteriously, over the course of a few patches to the code base. This allowed us to overcome our initial limitation of only being able to load roughly 2GB of data into RAMCloud, and to date we have been able to successfully generate a 1 billion node, 40GB graph in RAMCloud using SNAPCloud. One challenge we remained to face with RAMCloud, however a bit strange, was the time it took to drop a table. For a 40GB graph we measured that it took 10 minutes to drop the graph table, which had the effect of delaying some of the experiments on large graph sizes.

5. FUTURE WORK

There still remains many algorithms to be implemented in order to call SNAPCloud a true graph library. Further, there are many performance optimizations that have yet to be explored, including using multReads to increase the time to completion for calculating clustering coefficients, parallelizing the breadth first search code, and further parallelizing the Erdos-Renyi graph creation code through the use of conditional writes to resolve race conditions between threads modifying a single node in the graph.

There are a number of useful features to be added to RAMCloud as well that would make it a more useful system for SNAPCloud. One improvement would be to make table dropping more efficient. Another would be to implement snapshot and revert features, so that large 100GB-1TB graphs could be loaded in parallel directly on the RAMCloud servers, instead of relying on the client to bulk-load graph data into the system. This would be very helpful and time saving for running experiments on extremely large graphs.

Lastly, as a part of this work, it is important to evaluate the relative performance to other systems besides SNAP, but also PowerGraph, GraphLab, GraphChi, Pregel, GPS, NetworkX, GraphX, and others. In what cases does RAMCloud make more sense that other systems? When does it make less sense. These are important questions that were beyond the scope of this work, and are left for future work.

6. REFERENCES

[1] Clustering coefficient wiki article. http://en.wikipedia.org/wiki/Clusteringcoefficient.

[2] Erdos-renyi wiki article. http://en.wikipedia.org/wiki/ErdosRenyimodel.

[3] igraph. http://igraph.sourceforge.net/.

[4] Networkx. http://networkx.github.io/index.html.

[5] Snap: Stanford network analysis platform. https://snap.stanford.edu/.

[6] United state securities and exchange commission, facebook, inc. form 10-k. December 2012.

[7] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 17–30, Berkeley, CA, USA, 2012. USENIX Association.

[8] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[9] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin. Crunching large graphs with commodity processors. In *Proceedings of the 3rd USENIX conference on Hot topic in parallelism*, pages 10–10. USENIX Association, 2011.

[10] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Commun. ACM*, 54(7):121–130, July 2011.

[11] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 2:1–2:6, New York, NY, USA, 2013. ACM.