Partition: A Greedy Repartitioning System for Distributed Graphs (CS244B Final Project)

Samar, Anshul
asamar@stanford.edu

Eto, Naoki
naokieto@stanford.edu

December 12, 2017

1 Overview

We present *Partition*, a greedy repartitioning system for distributed graphs. *Partition* operates in a decentralized fashion, reorganizing vertices and edges across server nodes, without the need of a master.

In Section 2, we discuss motivation, related work, and the underlying model. In Section 3, we describe our repartitioning algorithm and the Paxos consensus protocol. In Section 4, we present our implementation. We also present experimental results, most notably, a 40% reduction in the the average inter-node edge count for Erdos Renyi graphs on 25 vertices and around 30 edges. We also demonstrate more than a 50% inter-node reduction for preferential attachment graphs on 25 vertices. In Section 5, we discuss our development process and conclude.

2 Motivation and Related Work

With the rise of large distributed data stores, exploiting patterns and workload history to shard data across nodes is critical for reducing transaction latency.

Facebook's large social network offers one significant use case. At Facebook, servers receive queries from clients requiring them to get data from external databases (queries such as: who are my friend's friends?). Because clients are largely interested in their own sub-networks, having a single node deal to all queries from a sub-network can allow nodes to keep common data in cache and reduce cache misses. Effective partitioning of such a graph - where friends are vertices and friendships are edges - ensures that queries from friends are dealt with by the same node. Facebook implemented their centralized distributed repartitioning algorithm, lowering cache misses by 50% [1].

Schism [2], on the other hand, uses repartitioning to minimize the cost of distributed transactions. As the authors write, the consensus protocol needed to execute such a transaction "adds network messages, decreases throughput, increases latency, and potentially leads to expensive distributed deadlocks." To prove this, the authors conduct a simple experiment in which clients attempt to read two random rows of a database sharded across multiple servers. Distributed transactions are shown to take twice as long to complete, along with a significant hit to throughput. To reduce need for distributed transactions, Schism repartitions the database. Specifically, Schism creates a fully complete subgraph for every transaction (each tuple, for example, may be represented by a vertex). Vertices that represent the same tuple across transactions are connected by an edge. Edges between vertices of the same tuple are given weights, with higher weights to those involved in more write transactions. By partitioning this graph with a min-cut, vertices sharing transactions get pushed to the same partition, keeping the number of distributed transactions small. Because replicated vertices that require updating have higher weight edges, the min-cut prefers to keep them in the same partition, thus reducing the overhead of writing to replicated nodes.

This project attempts to solve the general problem of repartitioning but unlike Facebook, which uses a master to coordinate repartioning, *Partition* is decentralized. While our underlying algorithm is similar to Facebook's, we developed it independently before learning about their solution. While our implementation stops after a certain number of repartitioning rounds, it is designed to run forever.

We propose a system that would not only work in the above use cases, but also in large organic networks that have no centralized authority and are dynamically changing. As distributed ledgers such as blockchain become unwieldy and are used to record trillions of transactions, having nodes store only portions of the graph depending on their geographic location and clusters of transactions may be a viable solution (for example, more money may be exchanging hands in one area than across two).

3 The Partition Model

Before we introduce the *Partition* model, we offer some terminology and assumptions. Note that to reduce scope we do not implement every feature here in our final system (see implementation section).

3.1 Terminology

1. **Graph**: A graph consists of vertices and edges sharded across nodes. Vertices and nodes are described by ids.
2. **Nodes**: A node contains vertices and edges, but is restricted in the number of vertices it can hold. It also has data structures to hold vertex neighbors and mappings from vertex to node. Due to the large size of the graph, we assume that nodes only keep track of their own vertices and the nodes that vertex neighbors are on. We refer to the entire collection of nodes as the cluster.
3. **Transaction**: A transaction is a message that specifies a single vertex, the node that is was on, and the node that it is being moved to - it also includes a list of neighbors of that vertex and the nodes on which they reside.
4. **Instance**: Our system progresses in a series of instances. Each instance, one transaction is committed by the cluster.
5. **Consensus**: We use Basic Paxos [4] [3] to ensure safety and consensus among nodes. Paxos provides safety with up to half nodes failing under asynchronous network conditions. It is important for our system to be consistent at the end of every instance - i.e. if vertex 1 moves from node 0 to node 2, all other nodes should know of the change, most especially node 2. This is required not only to keep node information about the global graph up to date, but also in case underlying data needs to be moved.
6. **Proposal**: Each proposal consists of the instance, current round number within that instance, node id, and transaction message. Proposals within an instance are ordered by round number and then node id.

3.2 Normal Operation

The *Partition* model runs through a series of instances of Basic Paxos. At the start of every instance, every node queries its own vertices and edges and determines a transaction it wishes to propose. Using a series of PREPARE and ACCEPT messages part of the Paxos protocol, it attempts to get its transaction accepted by a majority of nodes in the cluster (including itself). At the end of every instance i, a transaction is selected and then executed by all nodes. Note that not all transactions change each node - if a transaction relates to vertices not on a node, then it becomes a no-op for that node. Nodes are not allowed to work on future Paxos instances until they figure out the transaction chosen in the current instance.

3.3 Determining a Transaction

We propose a simple greedy algorithm. Abstractly, this algorithm proceeds by nodes acting greedily and pushing vertices they do not want to other nodes.

To determine what transaction to propose, every node randomly picks a vertex from its vertex set and attempts to move v to wherever it has the most amount of neighbors. If that node has space, it starts a Paxos proposal to get this transaction confirmed. If it does not, the node proposes a NONE transaction.

While we only implemented the greedy approach above, here are other repartioning algorithms for future models:

1. **Load Balancing**: Let $CAP(M)$ represent the percentage of remaining system capacity present in node M. Here, rather than suggest a random vertex, a node n weights each vertex v by the following: $max_m \frac{EDGES(v,m)}{EDGES(v)} * CAP(m)$ where $EDGES(v, m)$ is the number of edges v has on node m and $EDGES(v)$ is the degree of vertex v. Nodes propose transactions for vertices with high weights.

2. **Group Transactions**: Using information gained from Paxos messages, nodes create a fuller view of the graph and fill in missing edges probabilistically based on average degree counts of the current network. Then, they propose transactions that span multiple nodes. Specifically, nodes iterate through all clusters X of size $4, 5, 6$, weighting them with $CC(X) * INE(X)$, where CC is the clustering coefficient and INE is the percentage of inter-node edges. Transactions are proposed to move these clusters to empty nodes. This weighting gives priority to tight clusters that are spread out across multiple nodes.

3. **Direct Connection**: Here, nodes do not attempt consensus and instead swap vertices directly with other nodes. This saves us node messaging and Paxos overhead - but at the cost of shared data structure consistency.

3.4 Consensus and Failure

The Paxos consensus protocol is used to pick a transaction for a given instance and to ensure that everyone agrees to the same transaction. We implement Paxos based on a Paxos lecture, using similar variable names to keep consistency for ease of debugging [3]. See paxos.py for our implementation (along with server code). We defer an in depth treatment of Paxos to the original paper.

Nodes log both their graph data structures as well as Paxos state to disk. When a node comes back alive, it starts Paxos as normal - if any other node sees PREPARE or ACCEPT messages for old instances, it replies with information about the transaction that was chosen in that instance to help sender node get up to speed.

We deal with the following cases:

1. *Node dies in the middle of a paxos round*: nodes store what they have accepted when they start up again.

2. *All nodes except the proposer die before receiving prepare and reply messages*: No retransmission is necessary. The proposer simply waits. Other nodes will eventually propose transactions with higher round numbers.

3. *Node dies before executing transaction*: The instance number of paxos is only updated after a transaction has been logged. Thus the node begins a paxos round all over again. Other nodes see that it is attempting a round for an old instance and send update messages to inform it of the last transaction. Because we are only dealing with vertices and edges (which are stored as part of the transaction

messages), we avoid issues of safely transferring underlying blobs.

4 Implementation and Experiments

We did two implementations for the project. In Implementation 1 (see master branch), nodes are assumed to always be alive. We used this to produce the examples seen in this paper. In Implementation 2 (see ft branch), nodes can all die and start back up again and continue where they left off. However, we need a majority of nodes alive for any repartioning to occur due to consensus.

Note to teaching team: the second implementation has at least one weird concurrency deadlock we've found under a certain run. Thorough testing of this implementation is still needed.

Here are the various threads in each server node:

1. **Server**: Listens for incoming messages and adds them to a queue.
2. **Worker**: Responds to CHOSEN, PREPARE, PREAPREREPLY, ACCEPT, ACCEPTREPLY messages and executes Paxos logic.
3. **Proposer**: Proposes transaction repeatedly until it gets chosen or it is informed of another chosen transaction. If a proposer gets rejected in Basic Paxos, we add a random sleep before it starts again (in case two proposers get repeatedly caught in prepare/accept cycles - see page 7 of [4]).
4. **Main**: Controls worker and proposer to execute together, one instance at a time. In between instances, executes the chosen transaction and determines what next transaction to propose. It also sleeps for a time proportional to how many of its transactions have been chosen by Paxos, to allow fairness to other nodes in the system.

In the second implementation, we also have an 'update worker' which checks for update messages from other nodes about instances it might have missed when dead.

Details on using the system are on our github. Briefly, users can create files specifying vertices and edges (they can choose to shard randomly or specify a starting configuration). There are also scripts to create random Erdos Renyi graphs [6] and Preferential Attachment graphs [5]. The *run.py* script creates folders for each node. The user starts up every node - these need to be done together as we assume nodes are all live - and the system goes through a set number of Paxos instances. We then gather data from all the nodes to determine which vertices are where and use our pre-existing edge file to draw the final cluster setup (using the graphviz library). We tested the system by running each node as a separate process on one machine.

We now show a series of examples growing in complexity. Note that some examples - especially the ones after example 5 - may be done with slightly different versions of the system (i.e. a version with an extra sleep call, allowing vertices with in and out edges equal to also be moved, etc). The examples are in the appendix.

1. **Example 1** In Example 1, we demonstrate the common case of multiple subgraphs or clusters within a larger graph. Note that before repartitioning, the clique made by $(v1, v2, v3)$ are on three separate nodes. The two other cliques are over two separate nodes respectively. After a series of random repartitions, we find that two of the cliques are on unique nodes and only one of them spans two. The number of edges in between nodes shrinks from 7 to 2.
2. **Example 2** In Example 2, we 'mimic' a preferential attachment graph. While this is not generated by a preferential attachment process (as our later graphs are), we wanted to demonstrate what happens when a small subset of nodes have high degrees. Both vertices $(v1, v2)$ are placed on separate clusters. We go from 10 edges between nodes to 4.

3. **Example 3** This subgraph was generated from a Erdos Renyi model on 15 vertices. The number of edges between nodes shrunk 15 to 6. Here, notice that there are two vertices that do not have any neighbors. One of the drawbacks of our system is that it does not make any assumptions about vertices that have yet to have neighbors. In a social network example, it may make sense to place vertices without any friends on nodes corresponding to similar geographic locations.

4. **Example 4 and 5** Example 4 has a perfect repartioning of three triads. Example 5 has a perfect repartioning of three larger clusters.

4.1 Example 6

We conduct a series of experiments on Erdos Renyi graphs on 25 vertices. Specifically, we randomly assign vertices to nodes and then add an edge between two vertices with some probability p. Note that because Erdos Renyi graphs are random, we expect to not achieve a high degree of success - as any clusters are due to variance - but we use this as an easy way to generate test graphs.

We also conducted a series of experiments on preferential attachment graphs on 25 vertices. We built this using the model in [5] - the only difference is our graphs are undirected. In the referenced algorithm, p is the probability that a new vertex connects randomly to an existing node and $1 - p$ is the probability that it picks a random vertex and connects randomly to one of its neighbors.

For both, we determined the average percentage decrease of inter-node edges. An inter-node edge is an edge that spans two server nodes. The percentage decrease for high p is understandably low (there are too many edges to partition), but note the average successes for both Erdos Renyi with low probabilities and all preferential attachment graphs. Here are our results:

Type	Probability	Average Percent Decrease (over 10 runs)	Standard Dev
Erdos Renyi	0.1	39.07	15.43
	0.5	11.42	5.508
	0.9	1.827	1.258
Preferential Attachment	0.25	56.69	18.52
	0.5	50.31	9.189
	0.75	54.39	13.45

5 Conclusion

As part of this project, we did an extensive amount of experimenting and prototyping that did not make it into the final project. These include work on:

1. **Simulation**: A simulation to test our partitioning algorithm. Here, nodes swapped vertices all within a single process. See prototypes/simulation.py.

2. **Algorithms**: Exploring other algorithms for graph partitioning as well - for example, the randomized Karger algorithm or a max flow min cut solution.

3. **rpyc**: Distributed model using rpyc remote procedure calls (proof of concept). See prototypes/rpyc folder.

4. **Direct Connection**: Prototyping model in which nodes connect to other nodes directly (rather than broadcast transaction details to the entire cluster). See the original/ branch.

In conclusion, we described and implemented a greedy decentralized repartitioning system for distributed graphs. We proposed multiple algorithms for repartitioning and demonstrated experimental results using greedy repartitioning and Paxos. Most notably, we showed significant decreases in inner-node edges after running graphs through this system. Future work would include adding more repartioning algorithms

involving node to node messaging about various views of the network.

Acknowledgements and Notes

This is a joint project between Anshul Samar (CS244B/CS224W) and Naoki Eto (CS244B). Thanks to David Mazeries, Jure Lescovec, Peter Bailis, Anunay Kulshrestha, Seo Jin Park, and Michael Chang for helpful conversations and comments.

Small note on our in class demo: We realized later that the live repartitioning example that we showed was likely screwed up due to a bug. We've put the same example in the paper above, run correctly with our first implementation.

References

[1] Alon Shalita and Igor Kabiljo. *Using Graph Partitioning in Distributed Systems Design.* Talk at Scale (published 2014).

[2] Calro Curino, Evan Jones, Yang Zhang, and Sam Madden. *Schism: a Workload-Driven Approach to Database Replication and Partitioning.* Proceedings of the VLDB Endowment, Vol. 3, No. 1.

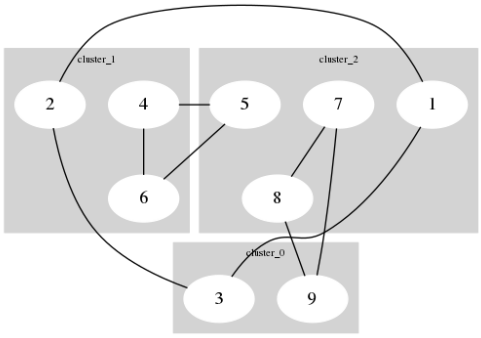
[3] John Ousterhout and Diego Ongaro. *Implementing Replicated Logs with Paxos.* 2013.

[4] Leslie Lamport. *Paxos Made Simple.* 2001.

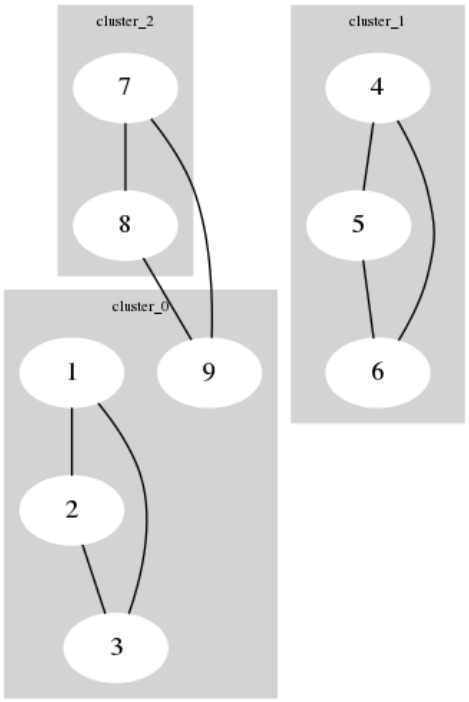
[5] Leskovec, Jure. *Lecture 2: Web as a Graph, Measuring Networks, and the Random Graph Model.* Stanford University CS224W.

[6] Leskovec, Jure. *Lecture 11: Network Formation Processes: Power-law degrees and Preferential Attachment.* Stanford University CS224W.

Appendix

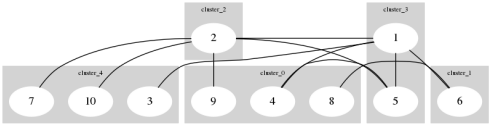


(a) Before

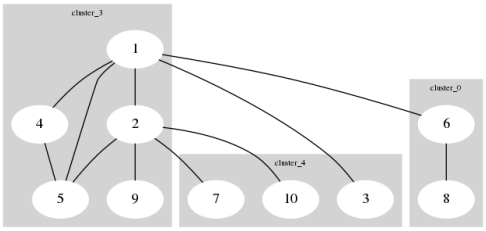


(b) After

Figure 1: Example 1

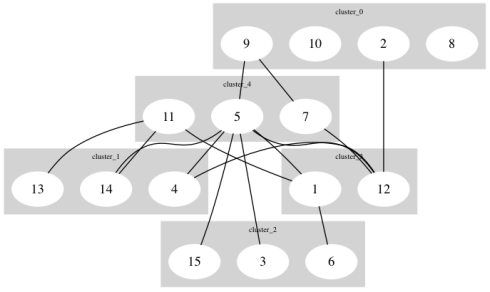


(a) Before

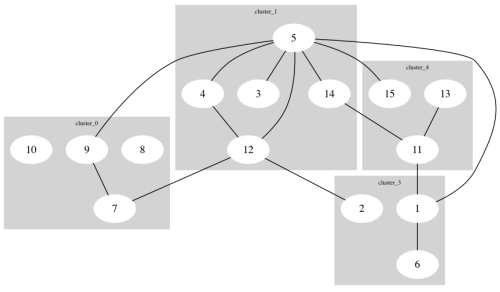


(b) After

Figure 2: Example 2

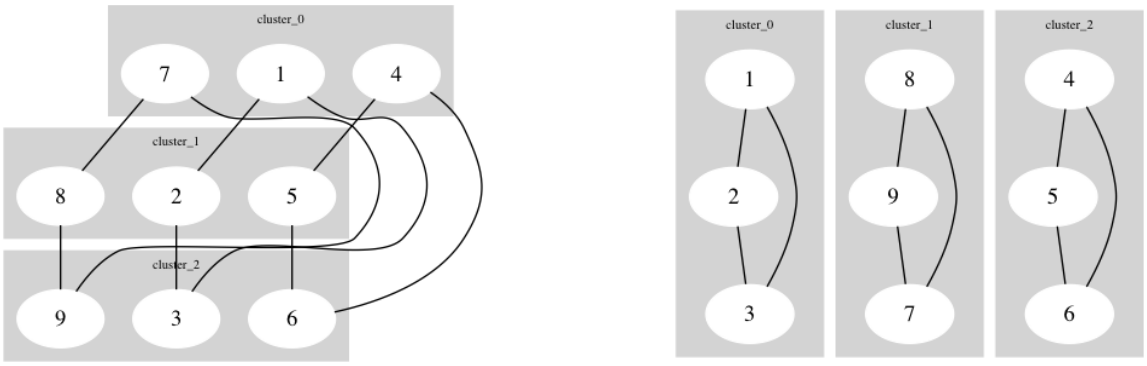


(a) Before



(b) After

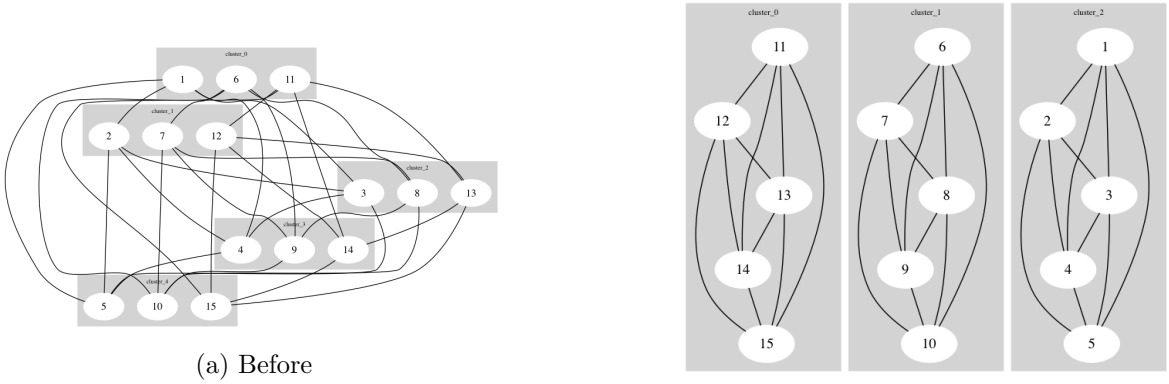
Figure 3: Example 3



(a) Before

(b) After

Figure 4: Example 4



(a) Before

(b) After

Figure 5: Example 5