Partition: A Repartitioning System for Distributed Graphs

Samar, Anshul
asamar@stanford.edu

Eto, Naoki
naokieto@stanford.edu

December 11, 2017

1 Overview

We present *Partition*, a repartitioning system for distributed graphs. *Partition* operates in a decentralized and dynamic fashion, reorganizing vertices and edges across server nodes, without the need of a master. In Section 1, we discuss motivation, related work, and the underlying model. In Section 2, we describe our repartitioning algorithm and the Paxos consensus protocol. In Section 3, we then present an implementation. Note that while Section 2 discusses a complete algorithm, our implementation makes some simplifying assumptions to reduce scope. We also present experimental results, most notably, a 40% reduction in the the average inter-server edge count for Erdos Renyi graphs on 25 vertices and around 30 edges. We also demonstrate more than 50% inter-edge reduction for preferential attachment graphs on 25 vertices. In Section 4, we discuss our development process and future work.

1.1 Motivation and Related Work

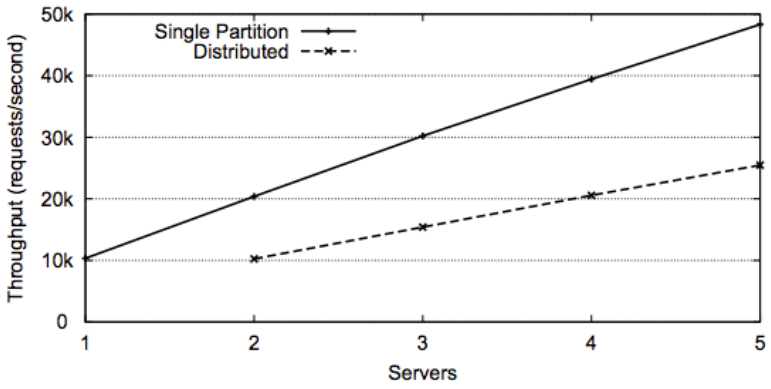
With the rise of large distributed data stores, smartly sharding data across nodes is critical for reducing transaction latency.

Facebook's centralized distributed repartioning solution [1] offers one example. 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 reduce cache misses and transaction latencies. Effective partitioning of such a graph - where friends are vertices and friendships are edges - ensures that queries from friends are routed to the same node. Their solution led to a drop of cache misses by 50%.

As another example, Schism [2] uses repartitioning to minimize the cost of distributed transactions. As 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 where clients attempt to read two random rows of a database sharded across multiple servers. Transactions which were distributed took twice as long to complete. Throughput was also significantly affected:

To reduce need for distributed transactions, Schism offers a repartioning solution. 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

Figure 1: Figure 1 from Schism: a Workload-Driven Approach to Database Replication and Partitioning



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 (i.e. via a costly protocol such as two phase commit).

A third, more general application, is in any setting in which data blobs have semantic relationships. If one models transactions as a series of read/write requests on blobs, the probability of a blob being read/written may strongly depend on whether blobs it is 'related' to were read or written in the past. Alternately, note that if the probability of a blob being accessed at time T has no bearing on the blobs accessed before T (i.e. they follow a Markov property), repartitioning a data store may not have much utility aside from load balancing. Many real world applications, however, have strong semantic relationships between blobs. In a research analysis workload for example, having an entire cluster of vertices on a single node could reduce time otherwise spent in accessing the data and allow for analysis/operation performed locally (i.e. computing the clustering coefficient). By placing related blobs together, we can reduce processing time.

This project attempts to solve the general problem of repartitioning with two additional constraints. First, 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 and did so in a decentralized setting. Second, unlike Schism, we allow for repartitioning to be done dynamically by the nodes themselves. This way, the system does not have to be stopped and started to repartition.

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 (e.g. more money may be exchanging hands in one area than across two).

2 The Partition Model

Before we introduce the *Partition* model, we offer some terminology and assumptions. We discuss our approach in its entirety - note we do not implement every feature here in our final system.

2.1 Terminology

1. **Graph**: Graph G consists of V vertices and E edges sharded across N 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 while nodes can keep track of their own vertices and the nodes that vertex neighbors are on, each node has only a 'local' view of the entire graph. If needed, nodes can discard information about neighbors that are on other nodes. It is possible that much of the node is being used for actual payload/underlying data and thus space for the graph itself is limited. All graph related information is kept in main memory.
3. **Cluster**: 'Cluster' refers to the entire collection of nodes.
4. **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.
5. **Instance**: Our system progresses in a series of instances. Each instance, one transaction is committed by the cluster.
6. **Consensus**: We use Basic Paxos [3] [4] 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.
7. **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.

2.2 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.

2.3 Determining a Transaction

Here, we propose a simple greedy algorithm (noting that strict partitioning is not required for performance gains). 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. Specifically, v should

be moved to node m where $m = \text{argmax}_m OUT(v, m) - IN(v, n)$. The current node n checks its local data structures (kept updated up to the most recent instance seen by n) to determine if m has space. If it does, it starts a Paxos proposal to get this transaction confirmed. If it does want to move any vertex, it can also propose an empty NONE transaction.

2.4 Paxos

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 discuss and implement Paxos based on a Paxos lecture, using similar variable names to keep consistency, as it became tricky to debug [4].

The central idea is that each node sends PREPARE messages to other nodes in the cluster with a proposal. When a node receives a PREPARE message it replies back with a PREPAREREPLY specifying any proposal that it may have accepted (or an empty proposal if hasn't accepted any). If the proposal it received in the PREPARE is higher than any proposal it has seen so far, it sets its 'minproposal' variable to that.

Once a node receives PREPAREREPLY message from a majority of the nodes (including itself), it sends ACCEPT messages. Once again, this message includes the proposal, replacing the transaction with whatever transaction it found in the PREAPREREPLY corresponding to the highest proposal number of non empty proposals. When a node receives an ACCEPT message, it replies with the minimum proposal it has seen so far, and accepts the proposal only if it is greater than or equal to the minimum proposal it has seen.

After having sent all the ACCEPT messages, it waits for a majority of ACCEPTREPLY messages, and chooses a value as long as all minimum proposals are less than or equal to the one it gave. Otherwise, it starts the process all over with a higher round number (and thus greater proposal value). Once a node chooses a value, it broadcasts this to all other nodes, and moves to the next Paxos instance.

Note that it is ok for a node to reply to accept even though it has not received a prepare. However, a node cannot send ACCEPT messages until it has received a majority of PREPAREREPLY messages.

Intuitively, we ensure that once a transaction has been accepted, all other nodes that send prepares will see that transaction and accept it and will be forced to accept one of the accepted transactions. We defer an in depth treatment of Paxos to the paper and lecture slides linked below.

2.5 Failure

We propose the following steps to ensure progress and safety with up to half the nodes failing (note this is not implemented). While there are some other minor details, these are the main components:

1. **Retransmission**: All messages come with ACKs. Thus proposing nodes can retransmit messages for the current instance, until the instance is decided.
2. **Node death**: Node comes back alive: Here, a node that comes back alive, asks nodes in the cluster for a history of all committed transactions. Once it applies these, it can start responding to messages for the current instance.
3. **Paxos death**: Paxos state and messages are logged, so in case a node dies in the middle of a Paxos round (such that no majority is left), if it comes back alive, it can resume its role as before.

4. **Recipient Node**: One of the messages in both the prepare and accept phase for a proposing node must be from the node that the transaction pertains to - upon receiving an accept for a transaction that pertains to itself, it must log necessary information and receive and save any payload to disk in case of crash.

3 Implementation and Experiments

In our implementation, we assume that all nodes are live and that all messages eventually reach. We thus implement a modified version of Basic Paxos (slide 12 [4]) without logging and fault tolerance. We also do not enforce any strict requirements on node memory and capacity outside of the maximum number of vertices it can hold. To see our code, please see github.com/anshulsamar/partition.

Here are the various threads in each process:

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 [3]).
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.

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 [5] and Preferential Attachment graphs [6]. 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 user defined number of paxos instances. We then gather data from all the nodes to determine which vertices are where and use our edge file to draw the final cluster setup (using the graphviz library).

We now discuss a series of examples growing in complexity - both in support of our system, but also in critique. 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). To run each example, we specified a number of paxos instances (for example, 80) and maximum capacity for a node. Note that only a small fraction of the paxos instances are actually transactions as nodes are allowed to propose empty transactions as well.

3.1 Example 0

This is a baby example to demonstrate proof of concept. This demonstrates the simple example of a 'dangling' vertex which has no neighboring vertices on its node.

3.2 Example 1

This is another simple graph over 5 vertices and 3 nodes. Note that the number of inter-node edges goes from 2 to 1, as a vertex is shifted.

3.3 Example 2

Here we attempted to repartition a complete graph over five vertices (all vertices fully connected to one another). With a high enough capacity, notice how all vertices were repartitioned to be on the same node. This is optimal behavior, but is a warning to us to be careful about setting node capacities. The best cut is not cut at all, but this is impractical as it requires a node with a more than practical amount of memory or storage.

3.4 Example 3

In Example 3, 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.

3.5 Example 4

In example 4, 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.

3.6 Example 5

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.

3.7 Example 6

This subgraph consists of three triads. It does not fully separate them - this can be a function of the number of paxos rounds (since nodes choose vertices randomly, perhaps the vertex that needed to be moved wasn't chosen yet) or the capacity of a node. In this case, the only vertex that needs to move is vertex 3, however, if node 0 has capacity of 5 it clearly cannot. This brings up another weakness of the system. Node 0 should be able to move multiple vertices together.

3.8 Example 7

A perfect repartioning of three triads.

3.9 Example 8

A perfect repartioning of three larger clusters.

3.10 Example 9

We conducted a series of experiments on Erdos Renyi graphs on 25 vertices. Specifically, we randomly assigned vertices to nodes and then added an edge between two vertices with some probability p.

We also conducted a series of experiments on preferential attachment graphs on 25 vertices. We built this using the model in [6] - 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 $1 - p$ that it picks a random vertex and connects randomly to one of its neighbors.

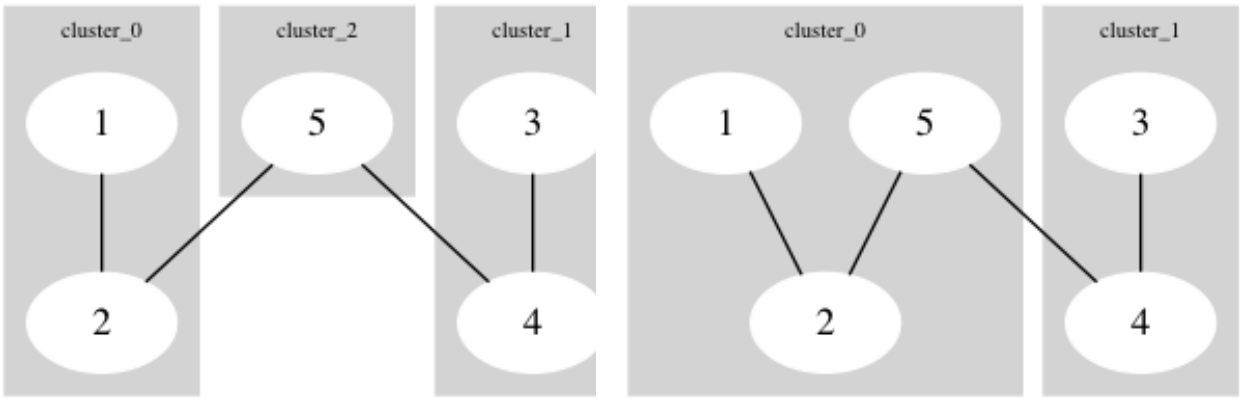
For both, we determined the average percentage decrease of inter-node edges.

Here are our results:

Type	Probability	Average Percent Decrease (over 10 runs)
Erdos Renyi	0.1	39.07
	0.5	11.42
	0.9	1.827
Preferential Attachment	0.25	56.69
	0.5	50.31
	0.75	54.39

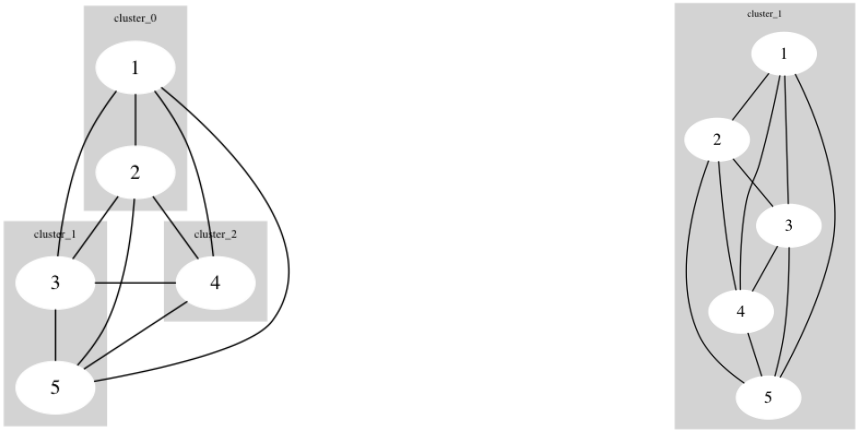
(a) Before (b) After

Figure 2: Example 0



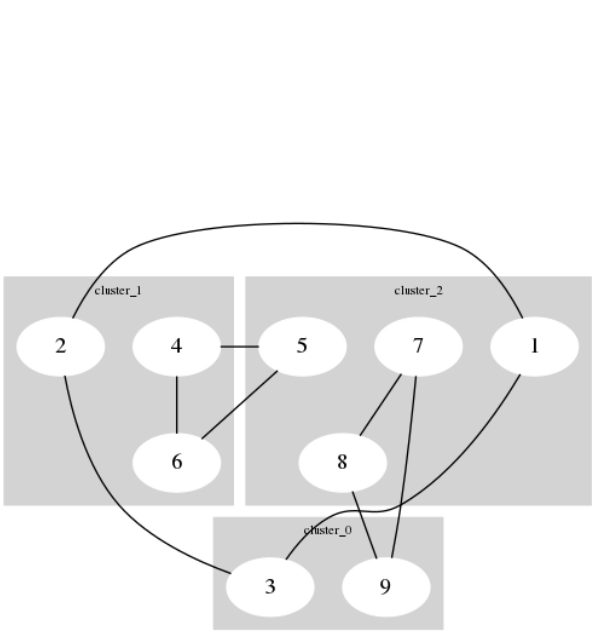
(a) Before (b) After

Figure 3: Example 1

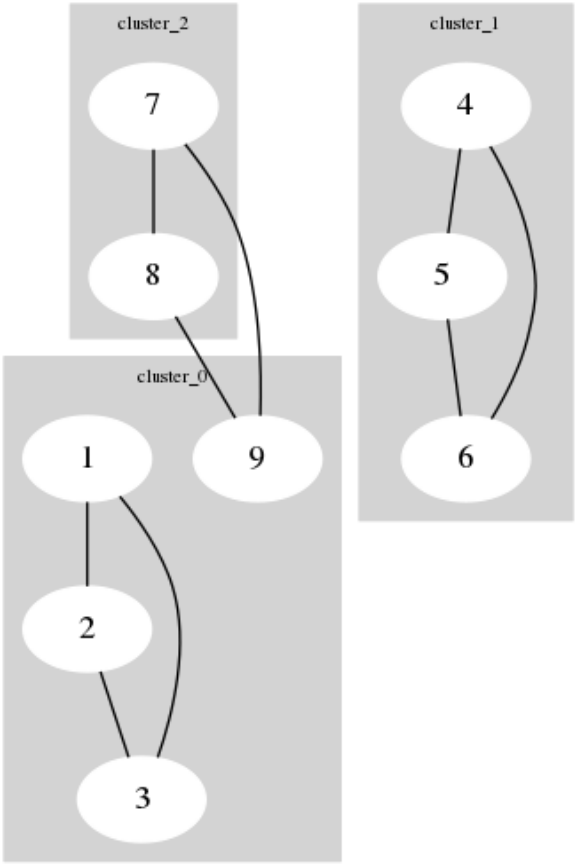


(a) Before (b) After

Figure 4: Example 2

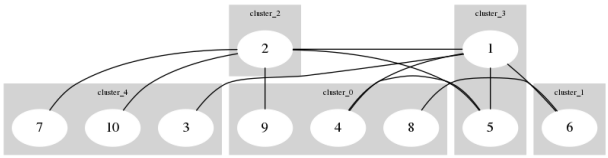


(a) Before

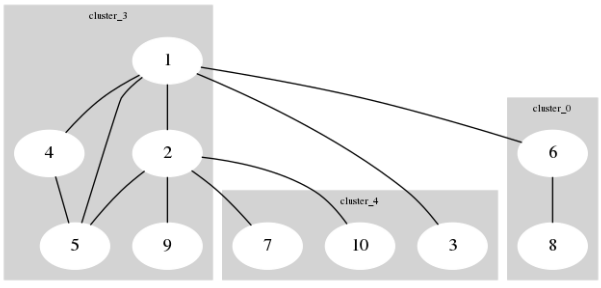


(b) After

Figure 5: Example 3

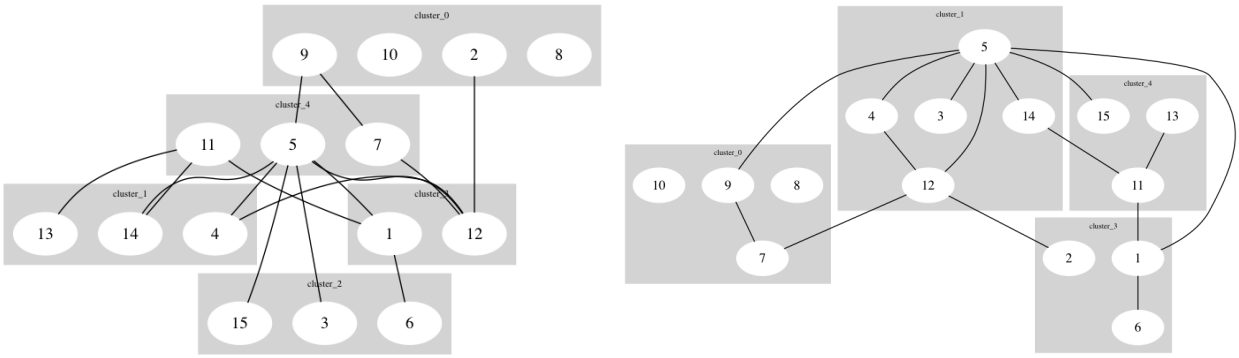


(a) Before



(b) After

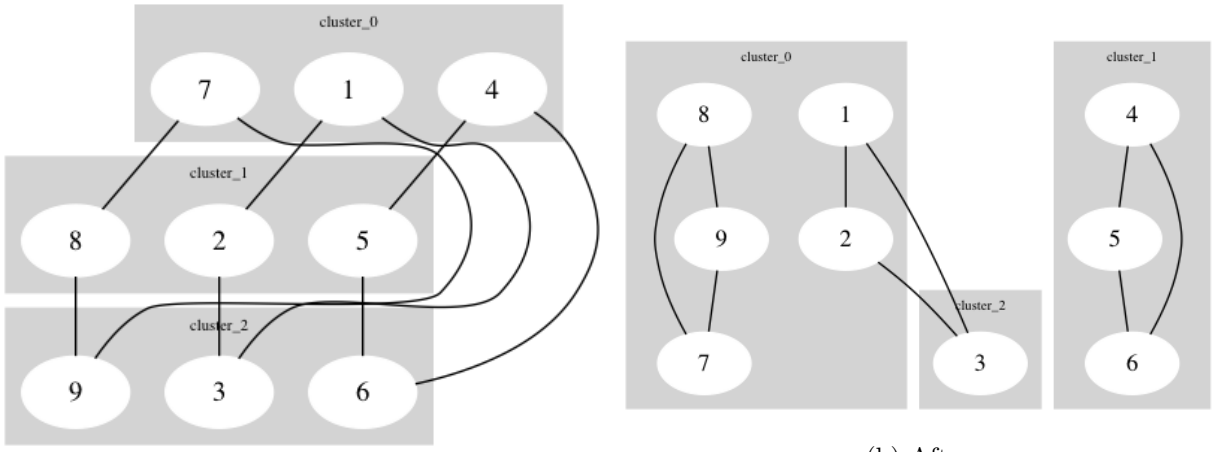
Figure 6: Example 4



(a) Before

(b) After

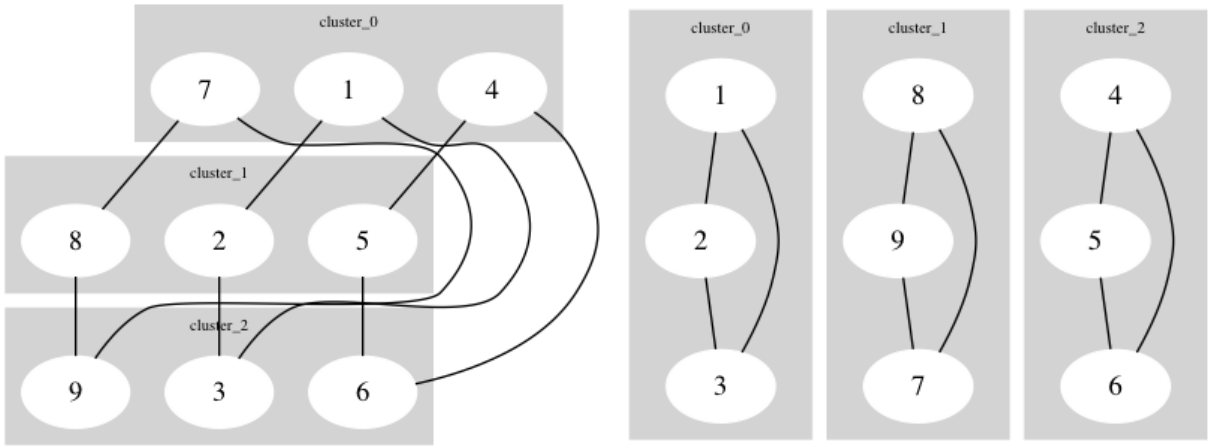
Figure 7: Example 5



(a) Before

(b) After

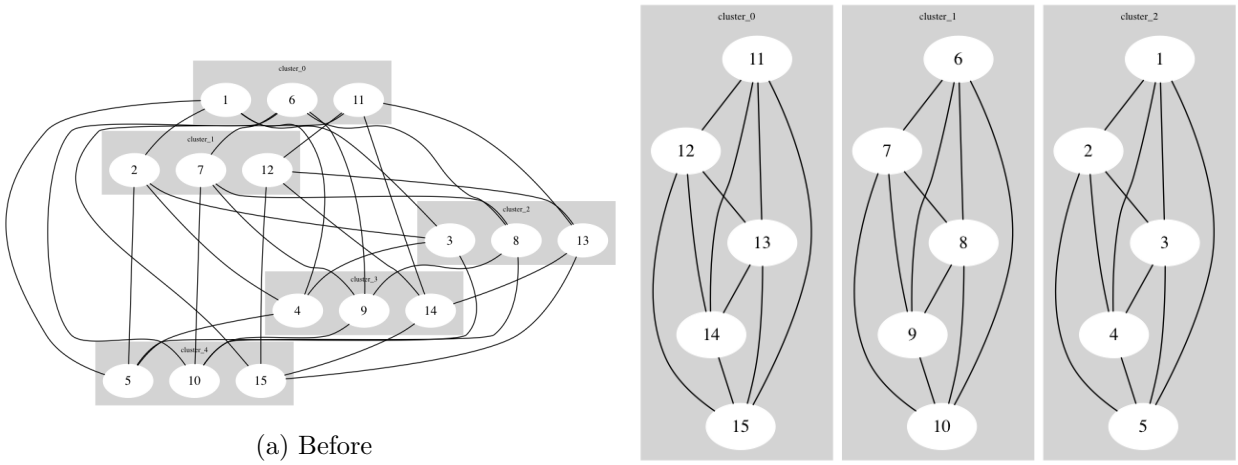
Figure 8: Example 6



(a) Before

(b) After

Figure 9: Example 7



(a) Before

(b) After

Figure 10: Example 8

4 Development Process and Future Work

As part of this project, we did a variety of experimenting and prototyping. These include work on:

1. **Simulation**: A simulation to test our partitioning algorithm. Here, nodes swapped vertices all within a single process. See 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. We also looked at the Pregel paper (graph processing system). One of our original algorithms, close to our final version, is in the appendix section.
3. **rpyc**: Distributed model using rpyc remote procedure calls. This was abandoned as it did not seem to scale to what we wanted to do. See rpyc/ folder.
4. **Direct Connection**: Prototyping model in which nodes connect to other nodes directly (rather than broadcast transaction details to the entire cluster). This was abandoned as we decided on that all nodes should have consensus on shared data structures (node capacities and vertex to node mappings). See original/ branch.

Various extensions to this project can be made as follows:

1. **Load balancing**: Because we know the capacity of a node at any given time, we can move around vertices to better load balance the system. If each node keeps a track of read/write requests to each vertex it can move vertices around to better balance this.
2. **Group Messaging**: While nodes all have a 'local' view of the graph, they message each other to determine what other edges exist. Using this and the information gained from Paxos messages (as those including complete proposed transactions), they create a fuller view of the graph - filling in missing edges probabilistically based on average degree counts of the current network. Then, they propose transactions that span multiple nodes (for example: move cluster to node 2, or move this triad to node 3), choosing clusters with higher cluster coefficients, etc.

We will be open sourcing our system on github.com/anshulsamar.

Acknowledgements and Notes

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

We also looked into some general resources while studying this problem (i.e. explanations of Karger's algorithm or lecture videos on max flow min cut) that we did not include in the bibliography to save space. We also used some boilerplate networking python/threading code from tutorials that we have cited. Graphviz was used for creating graph images. We used similar variable names as the cited Paxos lecture [4] for ease of debugging.

While we do not have a dataset to release since this was a significant implementation project, our code has been publicly made available on github.com/anshulsamar/partition.

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] Leslie Lamport. *Paxos Made Simple* 2001.

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

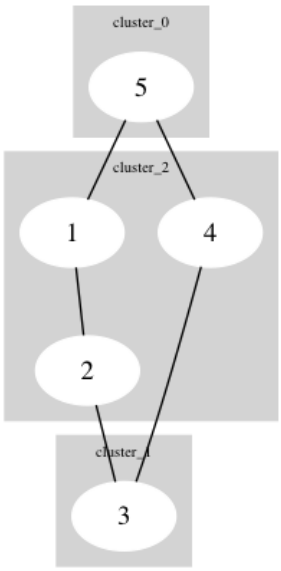
[5] Leskovec, Jure. *Lecture 11: Network Formation Processes: Power-law degrees and Preferential Attachment* Stanford University CS224W.

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

Appendix

This algorithm was not used, but we are including it as it was part of our development process.

Figure 11: Example of Small Graph with Lost Vertices



1. Let $E(v, n)$ be the number of edges some vertex v has with node n (i.e. how many of vs neighbors are on n).
2. If v is currently on node c, we define $L(n, v) = E(v, n) - E(v, c)$. This represents how 'lost' a vertex v is with respect to node n. For vertex $v5$, $L(n2, v5) = 2$ because it has 2 edges to node $n2$ but none inside its own node. A vertex is 'lost' if it has more edges to some node n than in its current node.

Assume each node n is single threaded and maintains four data structures.

1. v2v: vertex-to-vertex map describing the vertices currently on n and the endpoints of their edges (either within the n or crossing over into some other node).
2. v2n: an incomplete vertex-to-node mapping. It is important to note that this is not a global mapping, but just a mapping of the vertices that n knows about. For example, node 3's vertex-to-node map would include the following three (vertex,node) pairs: $(v5, n3), (v1, n2), (v4, n2)$. Its vertex-to-vertex map would include: $(v5, v1), (v5, v4)$.
3. n2c: a node-to-capacity mapping which holds the remaining capacity of a node. Assume that each node's maximum capacity is C. In Figure 1, if the maximum capacity was 4 then the node-capacity map would be: $(1, 3), (3, 1), (1, 3)$.
4. L: a sorted list of (node,vertex) tuples, sorted by $L(n, v)$.

Assume, for now, that all vector-node maps and node-capacity maps are updated together across all nodes and every node has the latest copy.

See Algorithm 1 below for details about this algorithm for some node x.

Algorithm 1 Repartition

```
1: v2v ← vector to vectors mapping
2: v2n ← vector to node mapping
3: n2c ← node to capacity mapping
4: procedure PUT
5:     for i ← 1, |L| do
6:         n, v ← L[i]                                  ▷ get the best v and n from L
7:         if n2c(n) < C then
8:             send(PROPOSAL,n,v)                       ▷ send n proposal to accept v
9:             msg ← listen()                           ▷ if timeout, continue
10:            if msg == ACCEPT then
11:                flush TRANSFER(n,v) to LOG
12:                send(DATA,n,v,v2v(v))                ▷ send v and v2v(v) to n
13:                msg ← listen()
14:                if msg == COMMITED then
15:                    flush DELETE(n,v) to LOG
16:                    delete v and v2v(v) from DISK.
17:                    v2n(v) ← n                       ▷ update mapping
18: procedure GET
19:     clear message buffer
20:     x, v ← get_proposal()                            ▷ get latest proposal
21:     if capacity ≥ C then send(ABORT,x)
22:     else
23:         send(ACCEPT, x)                              ▷ accept proposal from x
24:         data ← listen()                              ▷ get data, if timeout goto GET
25:         store data to DISK
26:         v2n(v) ← n                                   ▷ update mapping
27:         flush COMMIT(x,v) to LOG
28:         send(COMMIT, x)                              ▷ sends commit to x
```

Failures

If node x fails before it has sent a proposal, it can resume normal operation when it is back online. If failure occurs after having sent a proposal but before data has been transferred, no cleanup is needed. Node x upon returning can start the PUT procedure again and node n will timeout. If failure at x while data as being sent up till the point of deleting v, x upon return will find a TRANSFER log and will begin to retransmit the data. Here, x will need to ask n if data finished transferring. If n responds that it already committed, x can continue. If not, we resume transfer again. If x fails after it receives a commit message, it can replay its log and delete the now stale nodes and update its map. If n fails after having sent the ACCEPT message, we proceed similarly.