Evaluation of social graph partitioning strategies and improvement of distributed reconstruction of retweet cascades

Bernhard Lutz

Chair of Web Science

Albert-Ludwigs-Universität Freiburg im Breisgau

Master's Project

Submission: 30.05.2016

Abstract

On the social network twitter, retweet cascades can spread information very quickly to a wide set of users. From an economical and sociological point of view, it it is interesting to know, which user was influenced by whom. To perform reconstruction of these influences in realtime, the social graph must be stored in main memory. However, the size of the complete social graph quickly exceeds an average computer's main memory. This problem can be solved by distributing the social graph among several machines. Then, reconstruction is performed on the so called root node, which contains the root of the cascade, i.e. the author of the original tweet.

In case a user's follower list is not stored on the root node, the missing follower information must be transferred, which is considered to cause a lot of latency. The amount of users being stored on remote nodes strongly depends on the quality of the partitioning strategy. In previous work, several strategies have been proposed, but no live evaluation has been made so far. This work catches up on this analysis by defining a cost model for distributed reconstruction and performing reconstruction of given sample datasets.

Besides, the implementation of distributed reconstruction bears some space for improvements. Small-scale improvements will be directly implemented.

Contents

1 Introduction

Twitter is a social network where users can post messages with a maximum length of 140 characters, called *tweets*. If a user wants to get a notification whenever another user has written a new tweet, he can *follow* this user. The set of users following a given user U are the *followers* of U. The set of users that U is following are called *friends* of U.

In case a user U wants to forward an exciting tweet of another user to his followers, he can *retweet* (i.e. explicitly repeat) the original tweet. The followers of U will receive a notification and see this tweet on $U's$ profile. The followers of U can again retweet this tweet, forwarding it to their followers, which can retweet it again and so on. In this way, information can spread quickly, reaching a wide set of users. This scenario is called a *retweet cascade*, but unlike forwarding emails, the users at the end of a retweet cascade cannot determine all the stations in between. They only see the author of the original tweet and their friend(s) that retweeted it.
From an economical and sociological point of view, the *influence graph* indicating which user was influenced by whom is an interesting aspect.

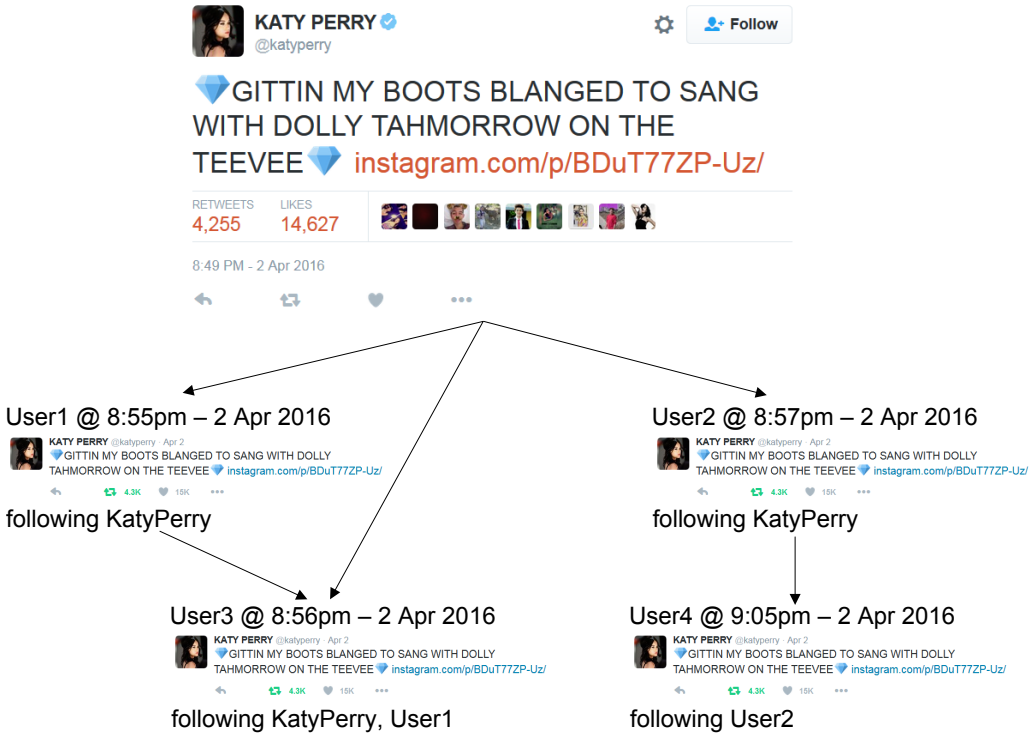


Figure 1: Small retweet cascade with influence edges

Figure 1 shows a small retweet cascade and possible influences. In this scenario, it is not clear whether User3 was mainly influenced by Katy Perry or by User1. Therefore, both edges should be created. Since User4 is only following User2, it is very likely that he saw Katy Perry's tweet on User2's profile. But he could also have randomly

visited Katy Perry's (or even another) profile and retweeted it from there. In this work, it is assumed that influence only comes from the followers relation i.e. a user can only be influenced by his friends.

The chair of web science at the university of Freiburg has developed a framework for retweet reconstruction based on *Apache Storm*. Apache Storm is a distributed stream processing framework, where applications are built in shape of directed acyclic graphs, called *topologies*. A small topology is shown in figure 2. The nodes represent processing logic and an edge (u, v) represents a flow of information from node u to node v.

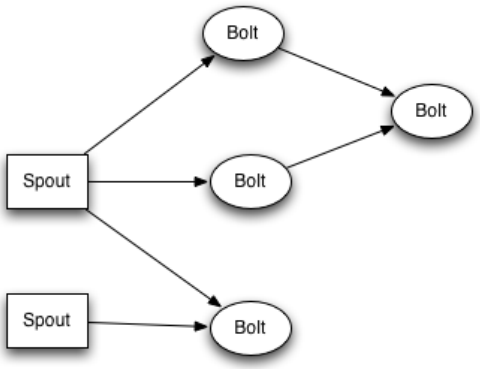


Figure 2: A small Storm topology

The source nodes are called *spouts* and act as a source of data. Inner and leaf nodes are called *bolts* and contain the actual processing logic.

In our scenario, the spout delivers the stream of retweets and the bolts perform reconstruction of the cascade. The chair of web science aims at performing reconstruction in realtime. Therefore, the *social graph* representing all follower information, must be stored in main memory. However, the complete social graph of twitter consists of several hundreds of gigabytes. An average computer cannot hold this amount of data in main memory. Thus, the social graph is partitioned and distributed among several machines.

In this work, the implementation of distributed reconstruction will be further analyzed and bottlenecks will be identified. Based on this information, a cost model will be defined, which allows comparison of different partitioning strategies. Obvious but small improvements will be implemented and observed bugs will be fixed.

This work is structured as follows: in section 2, previous work and the basic algorithms used for reconstruction are explained. Besides, the process of going from centralized reconstruction to distributed reconstruction is described. The actual problem setting is stated in section 3. The major changes on the implementation are described in section 4. In section 5, the metrics are defined and explained. In Section 6 the results are presented and section 7 concludes this work.

2 Previous work

2.1 Centralized reconstruction algorithms

Centralized reconstruction on a single node was implemented by Sättler and Ebner in their master's project [1]. They implemented the basic infrastructure to perform reconstruction using Apache Storm. A single cascade could be reconstructed by exactly one node.

The basic algorithm for reconstruction is called *Prefix Iteration* and works as follows: whenever a new retweet of a particular tweet is made by user U, iterate over the *prefix* i.e. the set of users that retweeted this tweet before and check which of them U is following. After the iteration, append U to the prefix.

This algorithm performs well for small retweet cascades, but as soon as the cascade consists of more than ten thousand retweets, Prefix Iteration gets very expensive. Therefore, another algorithm called *User Iteration* was implemented. This algorithm considers the friend relation for reconstruction. Usually, the set of friends of a user is very small compared to the set of his followers. For big cascades it is much faster to check whether the friend list of the next user contains users of the prefix. Figure 3 illustrates both algorithms.

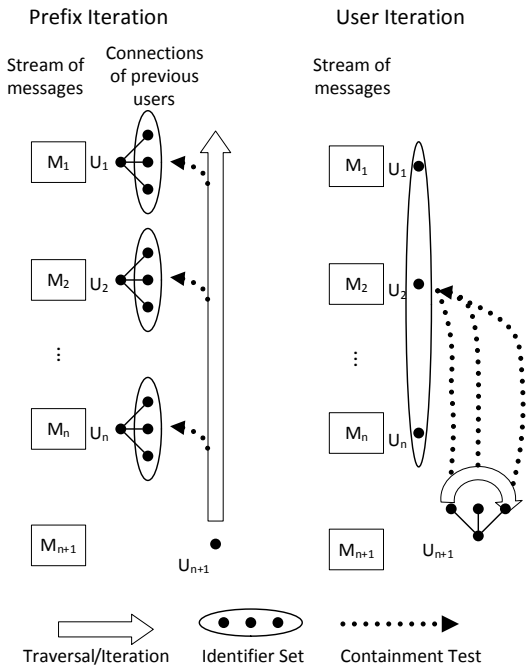


Figure 3: Prefix and User Iteration algorithms for central reconstruction

For further optimization, an adaptive reconstruction algorithm was implemented. It chooses between prefix and friend iteration depending on the length of the prefix and the size of the friend list of the current user.

The first metric of the cost model will be called *reconstruction time*, i.e. the time which is spent for reconstruction only.

2.2 Distributed reconstruction

When performing reconstruction on a single node, this node must store the whole social graph in its main memory. However, the size of the complete social graph of twitter quickly exceeds what an average computer can provide. In order to achieve better scalability, Michael Huber implemented distributed blocking reconstruction using Prefix Iteration in his master's thesis [2]. The social graph is partitioned among several nodes, but actual reconstruction is only performed on the *root node*. The root node is the node storing the follower information of the *root*, i.e. the author of the original tweet. All other nodes containing at least one user of the cascade, are called *remote nodes* and they are storing the follower information of the *remote users*, i.e. those users that are not stored on the root node.

Out of this scenario the following question arises: what happens, if the follower information of a retweeting user is not stored on the root node? Huber discussed several protocols for exchanging follower information. He ended up with a blocking implementation: before reconstruction can start, all necessary information must be transferred from the remote nodes to the root node.

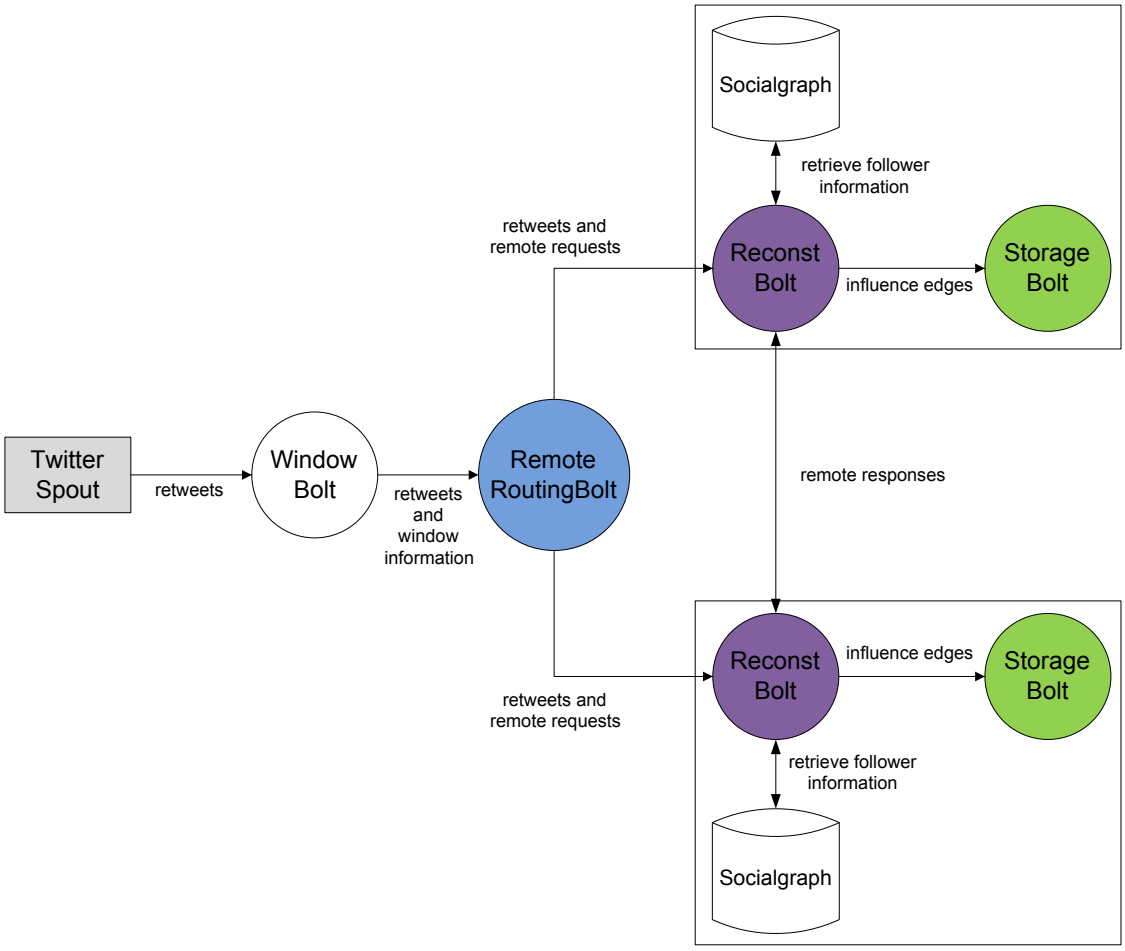


Figure 4: Storm topology for distributed reconstruction [2]

The application was implemented in shape of the Apache Storm topology shown in figure 4. The topology contains a spout emitting the retweets, which are processed by the WindowBolt. The WindowBolt embeds the retweets in *windows* by adding two values of meta information: *windowId* and *window state*. The windowId relates to the retweet cascade and simplifies further processing logic. The first tweet of a retweet cascade has window state *OPENING*, all other tweets have window state *CONSUMING*. After the last retweet of a cascade has been emitted, the WindowBolt emits a tuple *(windowId, CLOSING, empty)*, indicating that there are no more retweets in this cascade.

So far, everything works the same as in the centralized implementation. But now tuples emitted by the WindowBolt are first processed by the *RemoteRoutingBolt* and not directly by the ReconstructionBolt. The RemoteRoutingBolt holds a map $userId \rightarrow node$, indicating which users's follower information is stored on which node. When the RemoteRoutingBolt receives a tuple with window state *OPENING*, it determines the root node of this cascade and updates a map $windowId \rightarrow node$, indicating where the retweet cascade with the given windowId will be reconstructed. Since the implementation uses the Prefix Iteration algorithm for reconstruction, follower information of remote users must be transferred to the root node. The amount of messages for exchanging this information should be minimized.

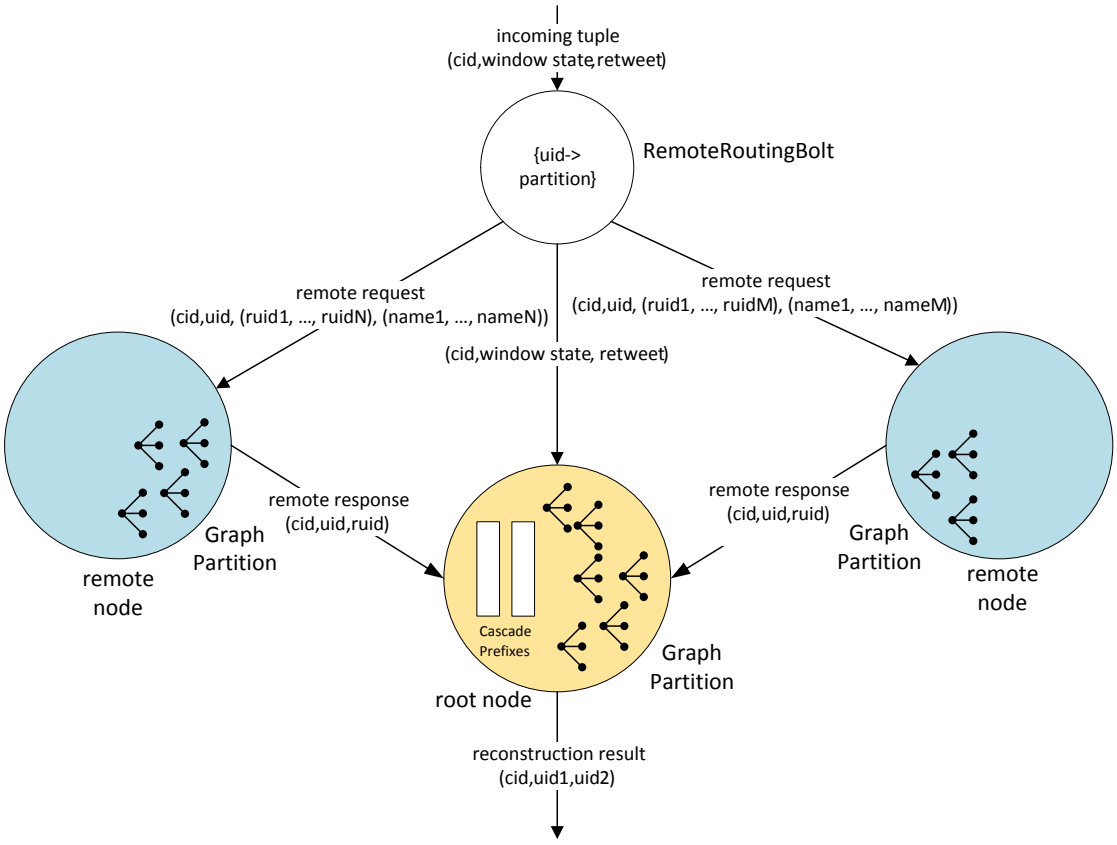


Figure 5: Stateless protocol for exchanging follower information

Therefore the cost model must contain a metric which depends on the number of messages. The corresponding protocol is sketched in figure 5. For every retweet, the RemoteRoutingBolt sends *remote requests* to the set of remote nodes, that contain users of this cascade. At the beginning, this set is empty. But as soon as every remote node as at least one user, the root node emits $\#nodes - 1$ remote requests for every upcoming retweet. A remote request has the following format:
(windowId, REMOTE_REQUEST, (root node, userId, list of remote userIds, list of remote user screennames, timestamp))
The lists only contain userIds and screennames of users that are stored on the particular remote node. This avoids redundant lookups.

Upon receiving a remote request, the remote node iterates over the list of remote userIds, and by accessing its part of the social graph, it determines which of these users the user identified by userId is following. For every positive match, the remote node emits a *remote response* to the root node:
(windowId, REMOTE_RESPONSE, (userId, remote userId, remote user screenname, message count, TRUE, timestamp))
When the root node receives a remote response, it inserts *userId follows remote userId* into its part of the social graph. *Message count* represents the number of remote requests received until this time. After sending a remote response, the remote node resets its *message count* to zero. Below, we will see why this information is needed.

After having sent all remote requests, the RemoteRoutingBolt determines whether the follower information of the the current tweet's author is stored on the root node. If this is not the case, the user is added to the list of remote users and the node containing the follower information of this user is added to the set of remote nodes. Finally, the RemoteRoutingBolt forwards the retweet to the root node, which stores the retweet for later reconstruction.

When the RemoteRoutingBolt receives a tuple with window state *CLOSING*, it emits a *remote close* to the root node. A remote close has the following format:
(windowId, REMOTE_CLOSE, number of remote requests)
It indicates that there are no more retweets in this cascade and that the node can wait for remote responses. As soon as all remote responses are have been received, the root node can start reconstruction, but it must somehow be able to determine this state. Since remote responses are only sent in case of positive matches, the root node might wait forever, if the last remote requests do not cause any remote responses. To avoid this, the RemoteRoutingBolt sends a *remote flush* to all remote nodes. A remote flush has the following format:*(windowId, REMOTE_FLUSH, root node)*
It makes the remote nodes send their current message count to the root node. For every received remote response, the root node adds up the message counts. As soon as it has reached the value of pending remote requests, received from the remote close, it starts reconstruction. The influence edges are emitted to the corresponding StorageBolt, which writes the influence graph to disk.

2.3 Partitioning strategies

In previous work, two partitioning strategies based on the activity graph of the Olympics in 2012 have been developed. In the activity graph, the nodes represent the users that were acting during the Olympics and an edge (u, v) with weight w indicates, that user v may have been influenced w times by user u. Since edges with weight zero are dropped, the activity graph is a subset of the usual social graph. The activity graph was created by reconstructing every retweet cascade on a single node and merging all influence edges.

It was assumed, that the amount of messages needed for transferring follower information, i.e. remote requests and remote responses should be minimized. Therefore, both strategies try to allocate users that are following and retweeting each other on the same partition. The activity graph gives perfect knowledge about these relations.

Micheal Huber proposed a partitioning strategy called *community-based* partitioning. The idea behind this strategy is to identify groups of users that have retweeted each other. The members of a group should be stored on the same node.

In his bachelor's thesis [3], Simon Ganz created a tool for converting the activity graph into a format that can be processed by the graph partitioning tool suite $METIS$, which creates a balanced min cut partitioning.
So far, only disjoint partitions are considered: every user is stored on exactly one node.

3 Task of this work

In previous work, several partitioning strategies have been proposed, but no real evaluation has been made so far. Simon Ganz only performed a static analysis of the partitioning. Michael Huber performed a live analysis on a cluster, but only the community-based partitioning was evaluated. In order to perform evaluation of different partitioning strategies, the implementation of distributed reconstruction by Michael Huber must be further understood. Bottlenecks and problems should be identified and analyzed.

To compare the proposed partitioning strategies, a cost model must be defined. So far, there are two metrics: reconstruction time and number of messages needed for exchanging follower information.

The task of this work further contains fixing bugs and implementing small-scale improvements. Ideas that that require a lot of refactoring should only be noted.

4 Changes on the implementation

4.1 Fixing bugs

4.1.1 Nodes were loading complete social graph

In Michael Huber's implementation of distributed reconstruction, every ReconstBolt was passed the same *SocialGraphManager* object, making the ReconstBolts load every user of the social graph. This makes every exchange of follower information redundant. Since this bug does not affect the reconstruction results, it was not discovered for some time. The bug was fixed by saving every parameter needed for instantiation of a *SocialGraphManager* and creating a new instance for every ReconstBolt.

4.1.2 Remote edges were missing for mixed cascades

Upon receiving a remote response *(windowId, REMOTE_RESPONSE, (userId, remote userId, remote user screenname, message count, TRUE, timestamp))*, the root node inserts an edge *userId* $\rightarrow$ *remote userId* into its partition of the social graph. After having reconstructed the cascade, edges that arrived via a remote response were deleted to prevent shortage of main memory. A problem occurs, when these remote edges are needed in several retweet cascades. After having reconstructed the first cascade, the root node deletes these edges from its partition of the social graph but during reconstruction of the next cascade, these edges are missing.

To overcome this problem, the root node stores remote edges in a separate data structure, which is created for every windowId. This is the first step towards merging of influence edges, since in the current implementation, Prefix Iteration is performed twice: once on the remote nodes and once again during reconstruction on the root node. The second iteration could be avoided by merging those edges that were transferred via remote response with those that were computed locally. However, the merge step is not trivial, since the temporal order of influence edges should not change compared to a reconstruction on a single node. Besides, the tweetIds would have to be added to remote requests and responses, since an influence edge also indicates the actual tweets that lead to this influence.

4.2 Improvements on the protocol for exchanging follower information

So far, two lists, one containing userIds and one containing screennames, are sent in every remote request. After the remote node has answered the request, these lists are dropped. Instead of sending the list of remote users over and over again, the remote nodes could as well remember the list of remote userIds. This changes the procotol from stateless into state-based. In the state-based protocol, every node stores an additional map $windowId \rightarrow list\ of\ userIds$ representing its *local prefix* of the given cascade. In Figure 6, the changes are marked in red. A remote request is reduced to: $(windowId, REMOTE\_REQUEST, (rootNodeId, userId, timestamp))$

Upon receiving a remote request, the remote node iterates over its local prefix of the given cascade and checks which of these users the user identified by userId is following. For every positive match, the remote node emits a remote response like in the stateless protocol. After that, the remote node checks whether this user's follower list is stored in its part of the social graph. If so, the remote node adds this user to its local prefix. Screennames are not needed anymore, since newer implementations of the *SocialGraphManager* are based on the userIds.

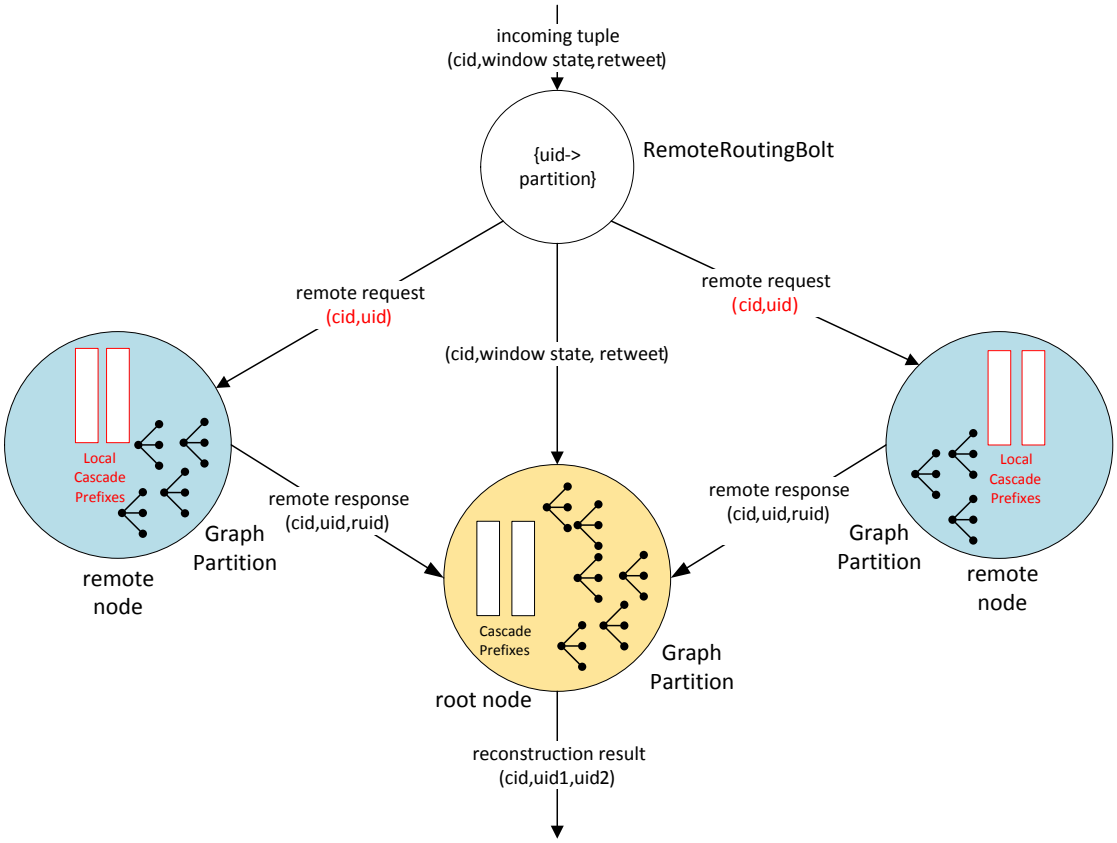


Figure 6: State-based protocol for exchanging follower information. Changes are marked in red.

This improvement reduces the amount of userIds being sent around in remote requests to the number of remote requests. A remote request now contains just one userId. The number of remote requests is upper bounded by $(\#nodes-1)\cdot\#retweets$. Experiments showed that the number is approximately this bound.

4.3 Sequential spout

The (parallel) spout of the previous version was emitting all retweets of all cascades at once. Thus, there were several cascades active at the same time. This might lead to conflicts for the cpus during reconstruction. Besides, results were showing strange values for message latency (see section 5.2.4 for an explanation of time metrics).
In order to have only one pending cascade, a sequential spout was implemented by Prof. Fischer. This version of the spout waits until the previous cascade has been completely reconstructed before emitting tuples of the next cascade. This behavior can be realized by waiting for the acknowledgement of the last tuple of the previous cascade.

To achieve guaranteed message processing[1], Apache Storm uses a method called *tree acknowledgement*. The bolts are allowed to *ack* or *fail* a tuple. If a tuple has failed, it can be replayed by Storm. A tuple is acknowledged when itself and all tuples that were created out of it have been acknowledged. Besides, a tuple is considered failed if it has not been acknowledged within a particular amount of time (default: 30 seconds). To tell Storm that a tuple was created out of another tuple, the new tuple must be *anchored*: *collector.emit(stream, tuple, newtuple)*.

Thus, simply acknowledging the retweet tuples sent by the RemoteRoutingBolt is not sufficient, since these acknowledgements are made before all remote responses have been received and even before any reconstruction has been performed. To achieve real sequential behavior, remote requests and remote responses must be anchored. Since reconstruction of a big cascade can take longer than 30 seconds, the timeout value was increased to several hours.

Note: implementing a sequential spout would also have fixed the bug, described in section 4.1.2.

[1] http://storm.apache.org/releases/0.9.6/Guaranteeing-message-processing.html, Documentation of Apache Storm, last lookup on 09.05.2016

5 Methodology

Like in Huber's master's thesis, experiments were executed on the researchers' cluster of the chair of databases and information systems. This cluster consists of ten nodes, each consisting of an Intel Xeon E5-2420 and 32 gigabytes of main memory. Eight nodes were running a ReconstBolt and one additional node was running the spout, WindowBolt and RemoteRoutingBolt. The version of Apache Storm was 0.9.5. Usually, Storm does not allow assigning a specific bolt to a particular machine, but this problem can be solved by using a scheduler. In this work, the scheduler written by Michael Huber was used.
The detailed configuration of Apache Storm is provided in the appendix.

5.1 Partitioning strategies in detail

5.1.1 Community-based partitioning

Besides implementing distributed reconstruction, Huber developed a partitioning strategy called *community-based partitioning*. The idea behind this strategy is to identify groups of users that have retweeted each other. The members of a group should be stored on the same node.

The creation of the partition works as follows: first the *labeling propagation algorithm* is performed on the activity graph. Every node v_i initializes its label to its id: $\ell_i = i$. In each round, node v_i determines its most popular neighbor: $n_i^* = \operatorname{argmax}_{v_j \in Neighbors(v_i)} weight(v_i, v_j)$ (In case of a tie, the neighbor with the highest id wins.) Node v_i takes the label of n_i^*: $\ell_i = \ell_{n_i^*}$. The labeling propagation algorithm creates a mapping $user \rightarrow groupId$ with about 80,000 groups. These groups have to be allocated among the partitions. Therefore, a random partitioning is created and stepwise improved by a hill climbing algorithm that searches for a local minimum on the number of crossing edges. The number of hill climbing iterations was reduced to 5,000, which was shown to be a reasonable trade-off between time and the amount of crossing edges [4].

The whole process took about 45 minutes on a machine with an Intel Xeon E5-2640 and 16 gigabytes of main memory. Another downside of this strategy is the fact that it does not consider the balance of the partitions.

5.1.2 Min cut partitioning of activity graph

The idea behind this strategy is similar to community-based partitioning: users that were retweeting each other should be allocated on the same partition. This implies that the number of intra partition edge weights is maximized. Thus, the number of crossing edges between the partitions must be minimized, yielding a min cut partitioning. In his bachelor's thesis [3], Simon Ganz created a tool for converting the activity graph into a format that can be processed by the graph partitioning tool

suite METIS[2], which creates a weight balanced min cut partitioning in a very short time. On the machine mentioned above, the creation of the partitioning took only a few seconds. This strategy has two major advantages: much shorter creation time and balance of the partitions. (Note: The simplest min cut partitioning would be a single partition containing all users, leaving the other partitions empty.)

5.1.3 Min cut partitioning of social graph

During realtime reconstruction, the information given by the activity graph will not be available. Using the usual social graph presents a more realistic approach. The usual social graph will also be partitioned by METIS.

5.1.4 Random partitioning

As a baseline, a random partitioning of the activity graph was created with the following bash script:

```
#!/bin/bash
if [ ! $# -eq 3 ]; then
  echo "Usage: $0 <activity graph> <number of partitions> <outputdir>"
  exit 1
fi
INPUT=$1
RANGE=$2
OUT=$3
while read line; do
  NODE=$RANDOM
  let "NODE %= $RANGE"
  echo $line >> $OUT/node${NODE}.csv
done < $INPUT
```

Obviously, this partitioning strategy does not explicitly consider any balancing.

To recap, the following partitioning strategies will be analysed:

- min cut partitioning of activity graph (metis ag)
- min cut partitioning of usual social graph (metis sg)
- community-based partitioning of activity graph (community)
- random partitioning of activity graph (random ag)

Besides, there are two versions of the protocol used for exchanging follower information:

- Stateless protocol: growing sizes of remote requests, since remote nodes do not record any state
- State-based protocol: remote nodes remember their state

This yields eight possible configurations in total.

[2]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, homepage of METIS, last lookup on 02.05.2016

5.2 Metrics

5.2.1 Remote users

A remote user is a user whose follower information is not stored on the root node. For every retweet that was made after a remote user's retweet, a remote request must be sent to the corresponding remote node to determine the missing follower information. A good partitioning strategy should minimize the number of remote users.

5.2.2 Remote requests

The number of remote requests is roughly the same for every partitioning strategy. As soon as a user from each remote node has acted in the cascade, a remote request must be sent to every remote node for every remaining retweet in the cascade. The number of remote requests depends only on the temporal appearance of remote users in the cascade. The difference in the amount of remote requests between the partitioning strategies was shown to be negligible.

5.2.3 Remote responses

Unlike remote requests, the number of remote responses strongly depends on the quality of the partitioning strategy and the selectivity of lookups (which may be approximated by the activity graph). For datasets consisting of several retweet cascades, the number of remote responses was summed up.

5.2.4 Time

To get a better understanding of the impact of a large number of remote responses, the times shown in figure 7 were measured. The *total time* comprises everything that happens between receiving the first tweet of a cascade and finishing reconstruction. The reconstruction time should be independent from the partitioning strategy.

In his master's thesis, Michael Huber declared *message or idle time* as the difference between total time and reconstruction time. Since this time can be measured very accurately, it will also be used in this work.

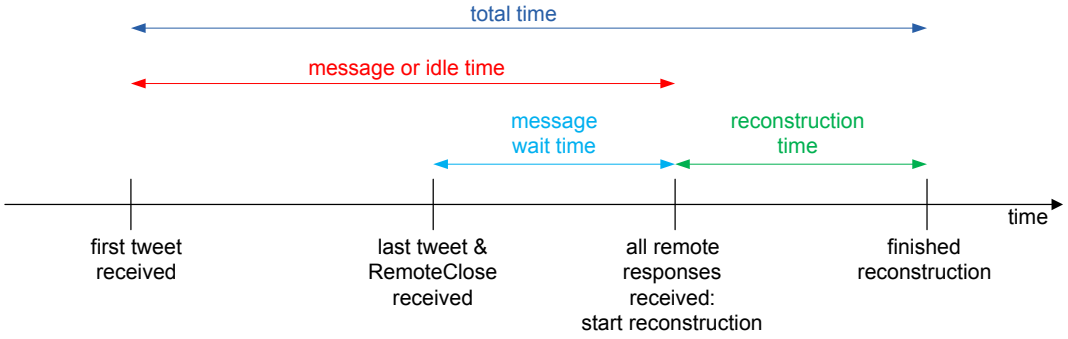


Figure 7: Timeline with events happening at a root node

However, the actual latency caused by exchanging follower information would be the time marked in blue: *message wait time*. It spans the time difference between having received a remote close and having received all remote responses.

Another interesting time metric is the amount of time which is spent by the remote nodes for answering the remote requests. This time will be called *request response time* and the sum of reconstruction time and request response time will be called *total reconstruction time*.

For datasets consisting of several cascades, reconstruction time, message wait time and request response time were summed up. The total time comprises the time between receiving the first retweet of the first cascade and having finished reconstruction of the last cascade.

The cost model will contain the following metrics: remote users, remote responses, message or idle time, reconstruction time, total time. Every metric should be minimized.

5.3 Datasets

The datasets being reconstructed were the same as in Huber's thesis. The dataset *500Casc* contained some *random cascades*, i.e. cascades with unknown root user. For such cascades, the RemoteRoutingBolt selects a root node at random, yielding non-deterministic results for the amount of remote users and remote responses. Thus, random cascades were deleted from the datasets. The machines were loading only those parts of the social graph that was necessary for correct reconstruction of the particular dataset and only one topology was running at the same time.

Additionally, two datasets were added: *5k-30k* and *1k-30k* containing all retweet cascades with at least 5,000 or 1,000 retweets and at most 30,000 retweets. Longer cascades were omitted, since for these cascades, reconstruction time takes most of total time.

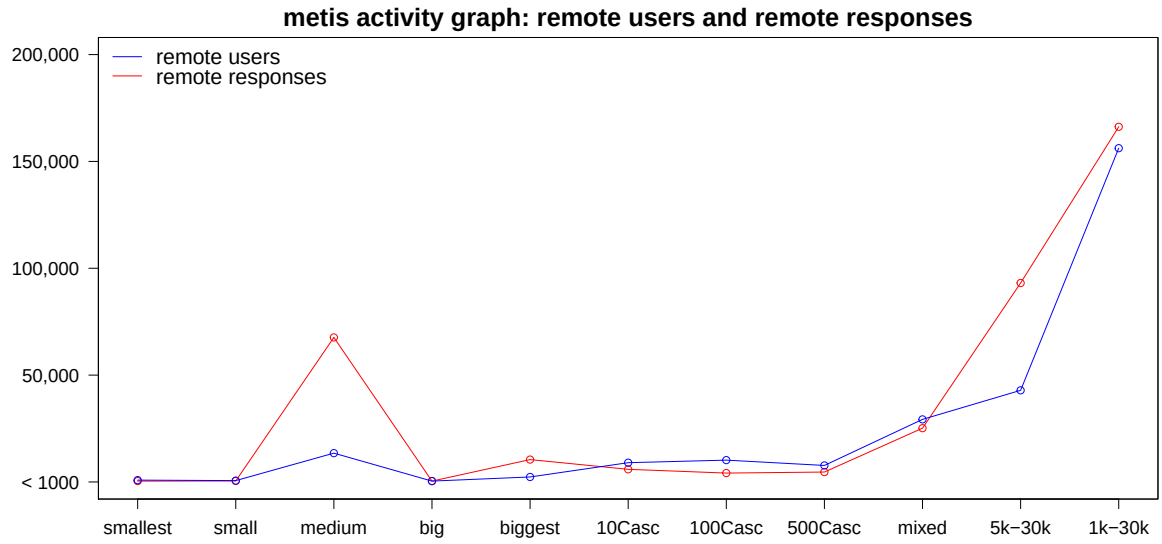
5.4 Expectations

1. **Remote users and remote responses:** It is expected, that there is some correlation between the number of remote users and the number of remote responses. Metis ag and community should perform best. Metis sg should perform significantly worse and random ag should perform worst, since $\frac{7}{8}$ of all users are expected to be remote.
2. **Message or idle time:** The state-based protocol should show a significant decrease in message or idle time compared to the stateless version.
3. **Total time:**, The partitioning strategies should perform in the same order has described in 1.
4. **Reconstruction time:** There should not be any significant difference, since reconstruction is always performed on the root node using the Prefix Iteration algorithm.

6 Results

6.1 Remote users and remote responses

Remote users and remote responses should somehow be correlated. Metis ag should be significantly better than metis sg. Community should be similar to metis ag.



As expected, there is some correlation between remote users and remote responses. The cascade *medium* seems to be a big outlier. This has also been discovered by Ganz and Huber. But no deeper analysis has been made so far. Section 6.2 catches up on this analysis.
Table 1 shows the relative amount of remote responses, where metis ag acts as a baseline. Metis ag performs best in minimizing remote responses. Community is slightly worse, metis sg significantly worse and random ag performs very bad.

Dataset	**metis ag**	**community**	**metis sg**	**random ag**
smallest	403	107.0 %	41.2 %	176.4 %
small	473	126.2 %	681.8 %	1788.4 %
medium	67,632	100.0 %	100.1 %	90.3 %
big	400	71.5 %	744.0 %	65,787.8 %
biggest	10,464	104.6 %	77.03 %	6,510.8 %
10Casc	5,912	162.0 %	442.5 %	1,309.8 %
100Casc	4,157	147.6 %	159.0 %	950.2 %
500Casc	4,633	137.2 %	131.1 %	390.2 %
mixed	25,166	131.2 %	186.4 %	3,243.8 %
5k-30k	93,101	102.4 %	292.6 %	1,257.9 %
1k-30k	166,208	111.0 %	249.5 %	1,175.5 %

Table 1: Remote responses for every partitioning strategy and dataset. Metis represents 100%

6.2 Analysis of medium

A sketch of the cascade *medium* is shown in figure 8. During the closing ceremony, the official twitter account of the Olympics tweeted that British boy group "Onedirection" is currently performing a song. Four minutes and 321 retweets later, the band's twitter account retweeted this tweet, forwarding it to their millions of followers.

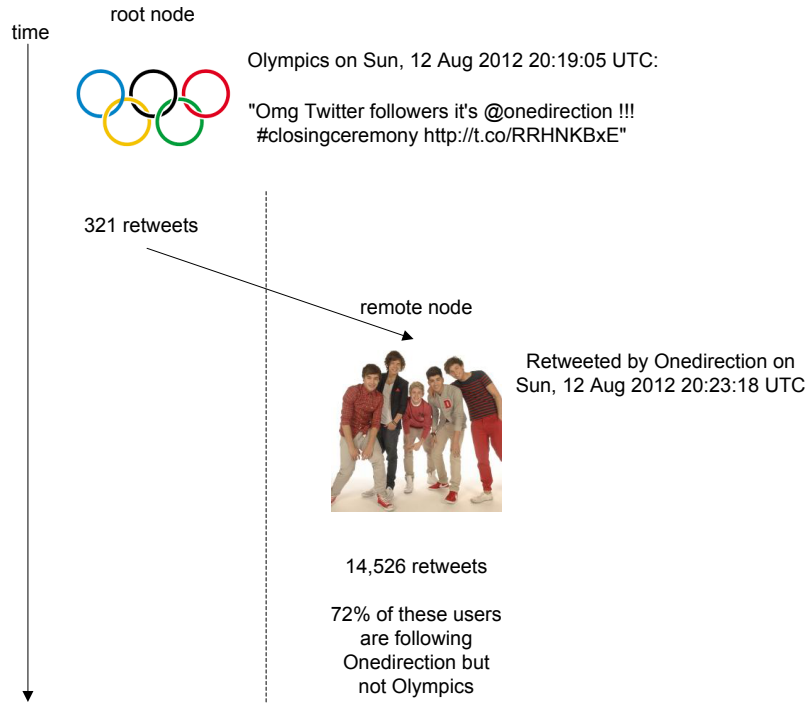


Figure 8: Sketch of cascade *medium*

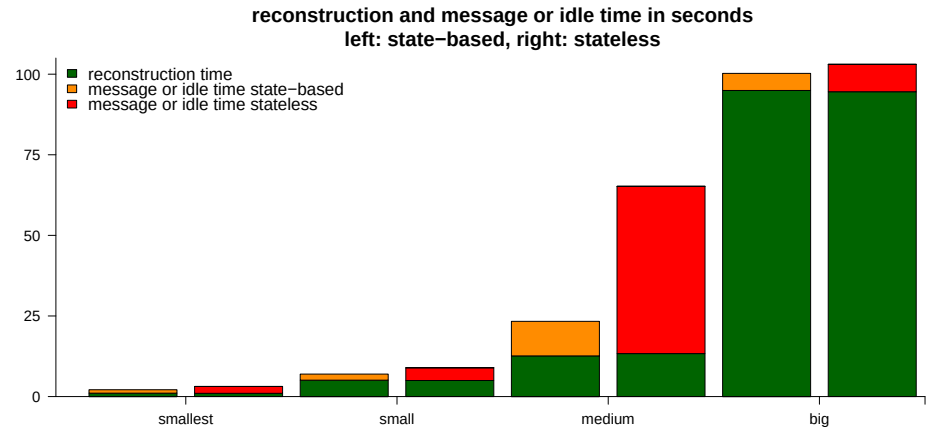
About three quarters of the remaining 14,526 retweeters are following Onedirection but not Olympics. Table 2 shows the number of users stored on the root node containing Olympics and the number of users on the remote node containing Onedirection. The "smart" partitioning algorithms allocated the followers of Onedirection acting in this cascade on the partition containing the band. As expected, the random partitioning allocates $\frac{1}{8}$ of all users on both nodes.

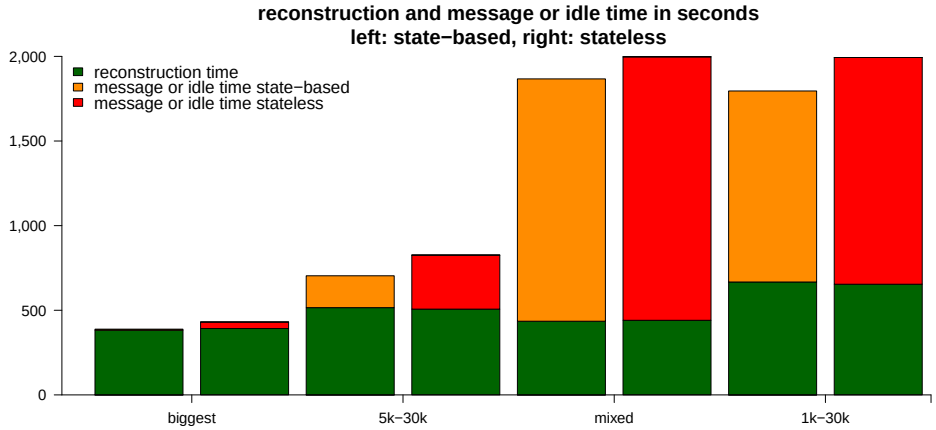
Distribution	**Users on root node containing Olympics**	**Users on remote node containing Onedirection**
metis ag	285	13,163
metis sg	76	13,408
community	307	13,111
random ag	1732	1785

Table 2: Distribution of users for cascade medium

6.3 Stateless vs state-based protocol

The stateless version of the protocol for exchanging follower information should perform significantly worse compared to the state-based version. Reconstruction was performed using the partitioning strategy community.

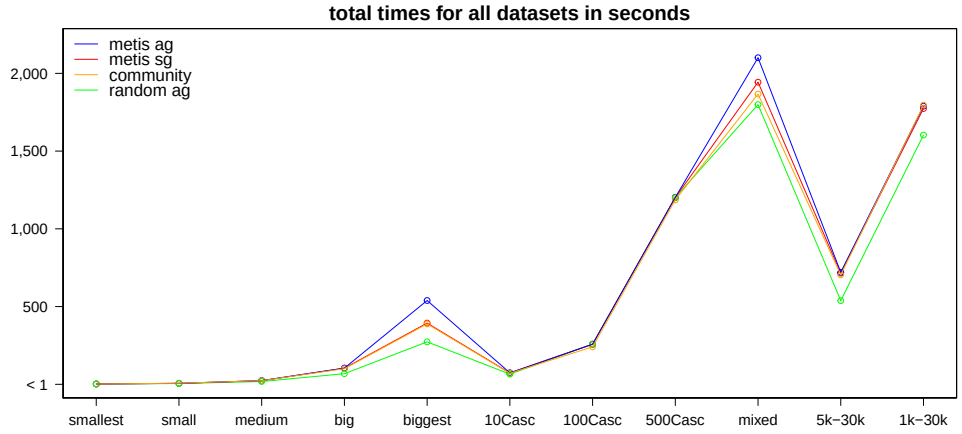




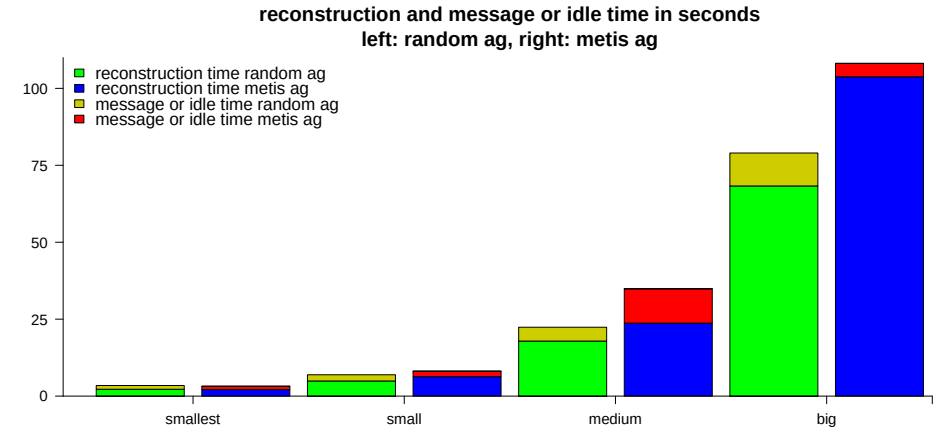
For single cascades, the largest impact appears for the cascade *medium*. For *biggest*, exchanging follower information is not critical. For this cascade, total time is almost completely used for reconstruction.
Changing the protocol from stateless to state-based creates a significant gain in message or idle time. In the following, only the improved protocol will be used.

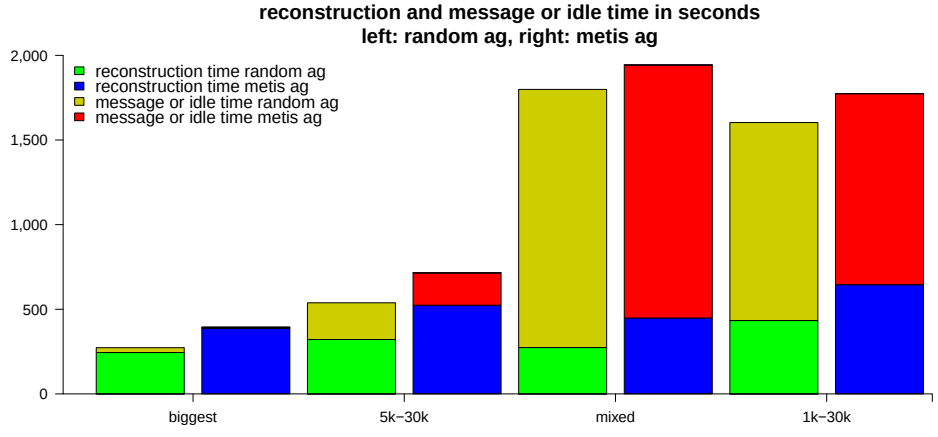
6.4 Totaltimes all partitionings

Metis ag and community are expected to perform best. Metis sg should perform slightly worse and random ag should perform worst.

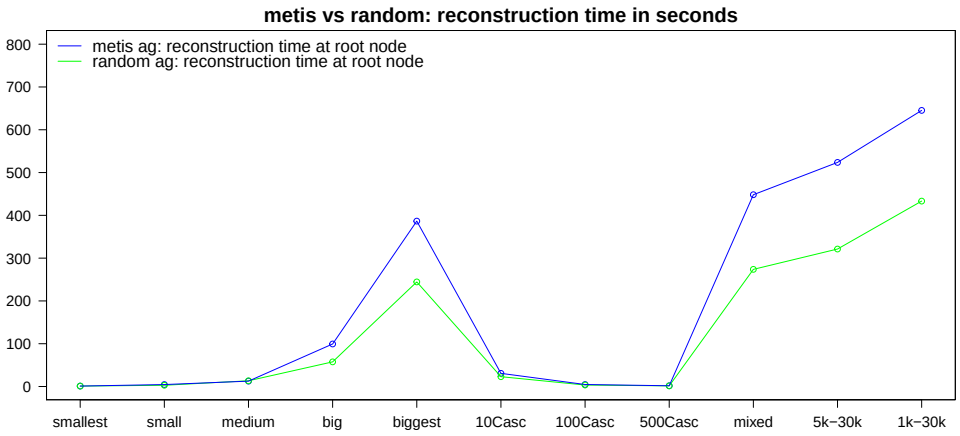


The results are surprising: random ag shows an outperformance compared to the other partitioning strategies. The following figure breaks down the costs for message or idle time and reconstruction time for metis ag and random ag. Random partitioning performs significantly better in reconstruction time.





6.5 Metis vs random: reconstruction time



Using a random partitioning makes reconstruction of large cascades much faster. Such a difference in reconstruction time is clearly not expected. The following section explains where this difference comes from.

6.6 Analysis of reconstruction time for random

Algorithm 1 describes the Prefix Iteration algorithm, which is performed on the root node, when all remote responses have been received.

Algorithm 1 Prefix Iteration

```
1:  input: retweets, localEdges, remoteEdges
2:  // USER(retweeet): user of retweet
3:  for (i = 0; i < size(retweets); i++) do
4:      follower = USER(retweets[i])
5:      for (j = 0; j < i; j++) do
6:          user = USER(retweets[j])
7:          if (localEdges containsKey user) then
8:              if localEdges[user] contains follower then
9:                  emit influence edge (user, follower)
10:                 continue
11:         if (remoteEdges containsKey user) then
12:             if remoteEdges[user] contains follower then
13:                 emit influence edge (user, follower)
```

Prefix Iteration takes $\Theta(n^2)$ time, where n is the number of retweets. The calculation of an influence edge $(parent,\ child)$ contains two steps:

1. check whether $parent's$ follower information is contained in the datastructure
2. If so, check whether $child$ is contained in $parent's$ follower list

The containment check must only be executed, if the existance check was positive. For random, *localEdges* consists of only $\frac{1}{8}$ of all users and *remoteEdges* contains only users that were influencing at least one other user. (Note: *remoteEdges* is built from remote responses.) For metis, most users are stored on the root node. Consider a user U acting in the cascade, but with no influence on anybody. For random it is very likely ($\frac{7}{8} = 87.5\%$), that this user will be remote. Since he does not have any influence, he does not appear in *remoteEdges*. Thus, depending on the size of the cascade, several thousand containment checks can be skipped. For metis it is much more likely, that U is stored on the root node. Thus, these containment checks must be performed. That is were the difference in reconstruction time comes from.

7 Conclusion and future work

The implementation of distributed centralized reconstruction could be improved by introducing a state-based protocol for exchanging follower information. The time in which the nodes are idle or exchanging messages could be significantly reduced.

Several "smart" partitioning strategies have been analyzed. Community partitioning did not show any outperformance compared to partitions created by METIS, that would justify the amount of time spent on hill climbing.

The activity graph yields much better partitionings than the usual social graph. However, a naive random partitioning strategy performs better than any "smart" partitioning strategy based on perfect information. The assumption that exchanging follower information should be minimized was shown to be wrong. The real bottleneck is the cost of Prefix Iteration. Thus, reconstruction should be parallelized, yielding an equal distribution of users among all nodes.
In future work, the following ideas should be implemented and evaluated:

- **Merging of remote responses:** actually, every remote response represents an influence edge. Instead of performing Prefix Iteration twice, these edges should be merged with those edges that were created by Prefix Iteration on the root node. However, the merge step is not trivial, since the temporal order of influence edges should be maintained. To preserve temporal order, the *tweetIds* had to be added to remote requests and remote responses,

- **User Iteration:** so far, only Prefix Iteration was implemented in distributed reconstruction. For cascades with several thousand retweets, Prefix Iteration gets very expensive. The protocol had to be changed, so that every user in the prefix is broadcasted to every remote node.

- **Migration to Storm 1.0.1:** version 0.9.5 of Storm was used during the experiments. The current version includes some performance improvements. For example, the maximum rate at which spouts can emit data has nearly doubled[3].

[3]http://storm.apache.org/2016/05/06/storm101-released.html, Storm Release overview, last lookup on 24.05.16

- **Incremental reconstruction:** blocking reconstruction is not feasible for realtime reconstruction. Actually, reconstruction can start as soon as the first retweet has been made. Incremental reconstruction could be implemented by waiting a fixed amount of time and performing reconstructing on the data collected until this point. The wait time could even be adapted to the frequency of retweets made in the past. The nodes still have to terminate reconstruction when they consider the cascade as finished. However, the chair of web science has already developed approaches for dealing with this problem [5].
- **Changing the root node:** if the major part of the retweeters is stored on a specific remote node (see cascade *medium*), this node could be made the new root node. This requires sending every information including all retweets and received remote responses from the previous root node to the new one.

Bibliography

[1] L. Sättler and S. Ebner, “Social Media Analysis”, team project, University of Freiburg, 2013.

[2] M. Huber, “Verteilte Rekonstruktion von Informationskaskaden”, master’s thesis, University of Freiburg, 2014.

[3] S. Ganz, “Partitionierung von sozialen Graphen zur Rekonstruktion von Informationskaskaden”, bachelor’s thesis, University of Freiburg, 2015.

[4] S. Embgen, “Evaluating a Community-Based Partitioning of a Social Graph over Evolving Interactions”, master’s project report, University of Freiburg, 2015.

[5] I. Taxidou, A. Alzoghbi, P. M. Fischer, and C. Schöller, “Towards Real-time Lifetime Prediction of Information diffusion”, 2015.

Appendix

Creation of partitionings

The activity graph consists of two files: *retweeters.orig.csv* containing the nodes and *olympics.activationPerUser* containing the (weighted) edges.
In retweeters.orig.csv, the line *null;null;null;null;null* was deleted.
For the creation of the metis partitionings, the program written by Simon Ganz had to be changed as shown in the following diff. The twitterId of the user is needed in order to create the nodefiles.

```
--- old/MetisInputGeneratorMain.cpp 2015-07-30 17:12:49.000000000 +0200
+++ new/MetisInputGeneratorMain.cpp 2016-05-07 17:11:37.753399828 +0200
 -105,7 +105,7
   // Write all nodes in the graph and their weights.
   newfile.open("Node-Weights");
   for(unordered_map<long,int>::iterator it=ids.begin(); it!=ids.end(); ++
       it) {
- newfile << it->second << " " << nodes[it->second] << endl;
+ newfile << it->first << " " << it->second << " " << nodes[it->second]
    << endl;
     nodeWeight += nodes[it->second];
   }
```

metis on activity graph

```
./ MetisInputGenerator retweeters.orig.csv olympics.activationPerUser
gpmetis MetisInputFile 8
java -jar MetisDistribution.jar Node-Weights MetisInputFile.part.8
    retweeters.orig.csv metisDist
```

Nodefiles can be found in ”metisDist”.

community on activity graph

```
bin/clusterAndAlloc --filetype=adjlist --refile=retweeters.orig.csv
 --file=olympics.activationPerUser --iterHC=5000 --numNodes=8
```

Nodefiles are created in the current folder.

Storm configuration

Storm configuration files were created as described in Huber's master's thesis. The nodes of the cluster are named "dbisma01-dbisma10". The nodes dbisma02-dbisma10 were used for running Storm.

Basic configuration

The following settings were used for every node:

```
storm.zookeeper.servers:
  - "sydney"
storm.zookeeper.port: 2181
supervisor.slots.ports:
  - 6700
  - 6701
  - 6702
  - 6703
nimbus.host: "sydney"
```

In the following, the lines shown have to be added to the basic configuration.

Sydney

The node "sydney" was running the nimbus and the webinterface of Apache Storm. Jobs for running topologies were submitted at this node, too.

```
ui.port: 13337
storm.scheduler: "de.unifreiburg.informatik.websci.scheduler.DistributedFixedScheduler"
supervisor.scheduler.meta:
 name: "nimbus"
```

dbisma02-dbisma09

These nodes were running the ReconstructionBolts. Therefore, the scheduler meta data must be filled with the corresponding nodeId X $\in \{0, ..., 7\}$

```
supervisor.scheduler.meta:
 name: "nodeX"
```

dbisma10

This node was running zookeeper (with standard configuration), the spout, WindowBolt and RemoteRoutingBolt. Therefore, the scheduler meta data must be set to "spout":

```
supervisor.scheduler.meta:
 name: "spout"
```

Script for running topologies

```
#!/bin/bash
# usage informations
if [ ! $# -eq 3 ]; then
  echo "Reconstruct a cascade"
  echo "Usage: $0 <cascade name eg: cascades-michael/kleiner>
   <metis metissg random truerandom michael> <jar file>"
  exit 1
fi

# jar file
DIST_JAR="$3"
echo "Cascade: $1"
echo "Distribution: $2"

VOL2="/vol2/storm-distrib"
CASCADE_VOL2=$VOL2/$1
C=`basename $1`
OUTPUT_DIR_PROJECTION="$VOL2/tempdir-$C"
SOCIAL_GRAPH_HIRARCHY_DIR="$VOL2/sg2013/"
USERS_IN_CASCADE="$OUTPUT_DIR_PROJECTION/retweeters.csv"
SG_USER_LIST_FILE="$OUTPUT_DIR_PROJECTION/sgUserListFile.csv"

NODE_DIR="$OUTPUT_DIR_PROJECTION/nodedistribution_8_$2"

C=`basename $1`
TOPOLOGY="dist-reconstruct-$C"
# clear logs on all nodes
clearLogs
storm jar $DIST_JAR de.unifreiburg.informatik.websci.topology.
    CentralCascWindowReconstTopology -maxHeapNode 30G -f "$CASCADE_VOL2" -
    sgLoadLogging False -nodeCount 8 -nodeDirectory $NODE_DIR -deployName
    "$TOPOLOGY" -sgmDirectory "$SOCIAL_GRAPH_HIRARCHY_DIR"
```