Large scale distributed reconstruction of retweet cascades

Bernhard Lutz

Chair of Web Science

Albert-Ludwigs-Universität Freiburg im Breisgau

Master's Thesis

Submission: 17.01.2017

Bearbeitungszeitraum
20. 06. 2016 – 17. 01. 2017

1. Gutachter
Prof. Dr. Peter Fischer
2. Gutachter
Prof. Dr. Georg Lausen
Betreuer
Prof. Dr. Peter Fischer

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum

Unterschrift

Abstract

On Twitter, retweet cascades can spread information very quickly, reaching a wide set of users. One interesting aspect about information diffusion is the influence graph that indicates which user was influenced by whom in sharing some content. The chair of Web Science has developed a distributed system for the reconstruction of retweet cascades. Since the complete social graph of Twitter is bigger than an average computer's main memory, the system allows the partitioning of the social graph among several machines. Missing social graph information is being exchanged via remote lookups.

So far, reconstruction was performed using an algorithm which needs quadratic runtime in the number of retweets of a cascade. In previous work, it turned out that an equal distribution of all users among all machines gives the highest throughput. Other distributions that go towards min cut partitioning were shown to be less efficient.

In this work, an algorithm which runs in $\mathcal{O}(n \cdot k)$ will be implemented, where n is the number of retweets and k is the average number of friends of all users in the cascade. Besides, other optimizations like incremental reconstruction, batching of remote lookups, multithreading and more efficient datastructures will be implemented. This optimizations reduced the total time for reconstructing a dataset consisting of about 3.6 million retweets from 22 minutes to less than 20 seconds, when using eight nodes for reconstruction.

Zusammenfassung

Auf Twitter können Retweetkaskaden Informationen sehr schnell verbreiten und dabei eine große Anzahl an Benutzern erreichen. Ein interessanter Aspekt ist dabei der Einflussgraph, welcher angibt, welcher Benutzer von wem beeinflusst wurde. Der Lehrstuhl für Web Science hat ein verteiltes System zur Rekonstruktion von Retweetkaskaden entwickelt. Dabei wird der soziale Graph auf mehrere Rechner verteilt, da der Hauptspeicher eines durchschnittlichen Rechners nicht ausreicht. Fehlende Informationen des sozialen Graphen werden mit entfernten Anfragen ausgetauscht.

Für die Rekonstruktion des Einflussgraphen wurde bisher ein Algorithmus mit quadratischer Laufzeit bezüglich der Anzahl Retweets einer Kaskade verwendet. In früheren Arbeiten stellte sich heraus, dass eine gleichmäßige Verteilung der Benutzer den höchsten Durchsatz liefert. Andere Verteilungen, welche in Richtung Min Cut Partitionierung gingen, waren weniger effizient.

In dieser Arbeit wird ein Rekonstruktionsalgorithmus implementiert, dessen Laufzeit $\mathcal{O}(n \cdot k)$ beträgt, wobei n der Anzahl der Retweets und k der durchschnittlichen Anzahl der Friends aller Benutzer der Kaskade entspricht. Außerdem werden weitere Optimierungen wie inkrementelle Rekonstruktion, Batchverarbeitung von entfernten Anfragen, Multithreading und effizientere Datenstrukturen vorgestellt. Diese Optimierungen konnten die Gesamtlaufzeit eines Datensatzes aus ca. 3,6 Mio Retweets von 22 Minuten auf weniger als 20 Sekunden reduzieren. Dabei wurden acht Rechner zur Rekonstruktion verwendet.

Contents

1 Introduction

On social networks, articles, posts, pictures and many other things can be shared. When a user shares a post from another user, the friends of the sharing user will get a notification of this event. The friends of this user can again share this post and then their friends can share it again and so on. This way, information can spread very quickly, reaching a large set of users.

One interesting aspect about information diffusion is the expansion or influence graph, indicating which user was influenced by whom in sharing something. Figure 1 illustrates the scenario of information diffusion. The root user created some content. This content is shared by U_1, U_2 and U_3. U_6 shares the content after seeing it on U_1*'s* profile. U_4 and U_5 have seen the content on U_3*'s* profile and U_7 has seen it on U_5*'s* profile.

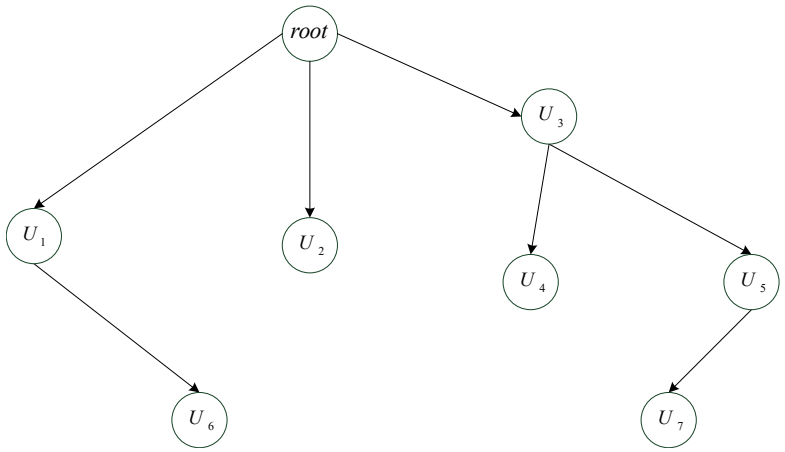

Figure 1: Information diffusion on social networks

On Twitter, users have the possibility to explicitly forward a tweet from another user to their followers. This is called retweeting a tweet. The followers of the retweeting user will see the original tweet on this user's time-line. They can again retweet the original tweet, forwarding it to their followers. The whole process is called a retweet cascade. In order to reconstruct the influence graph of a retweet cascade, we assume that influence comes only from the followers relation. Of course, might not allow us to calculate the actual influence graph. Instead, we will calculate all possible influences based on observable information. Thus, an edge $U_1 \rightarrow U_2$ of the influence graph indicates that U_2 follows U_1 and that U_2 retweeted after U_1.

The reconstruction algorithms need to access the social graph, i.e. the follower relation. The follower relation is represented in two ways: for each user, Twitter allows access to the list of followers and to the list of friends. Friends are those users that a user is following. Based on these two representations, there are two reconstruction algorithms. The algorithm based on the follower list is called *Prefix Iteration*. For every new retweet, it iterates over the *prefix*, i.e. the set of retweets that have been made before and checks whether the user who made the new retweet is contained in the follower lists of the current user of the prefix. The other algorithm

is called *User Iteration*. For every new retweet, it iterates over the set of friends of the new user and checks which of them are contained in the current prefix.

However, the algorithms alone cannot deal with an incoming stream of retweets. There must be an underlying system retrieving and grouping the retweets from Twitter, such that reconstruction is performed only for those retweets that belong to the same cascade. Besides, the algorithms are implemented for being run on a single node which does not allow scaling. Therefore, the chair of Web Science has developed a distributed system based on the stream processing framework *Apache Storm*[1]. The system is sketched in figure 2. The incoming stream of retweets is processed by the reconstruction system which emits the edges of the influence graph.

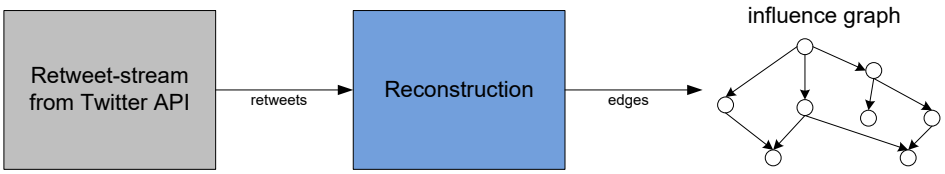

Figure 2: Conceptual sketch of the reconstruction system

For both algorithms, the respective social graph information must be held in main memory. However, the whole social graph of Twitter takes at least several hundreds of gigabytes, more space than an average computer can provide. The system allows a partitioning of the social graph among several machines. Reconstruction is performed on the node containing the follower or friends list of the *root*, i.e. the author of the original tweet. This node is also called *root node*. But what happens if the social graph information of a user is not stored on the root node? In this case, a remote lookup must be performed.

In previous work, it turned out that these remote lookups are very costly. In a later work, distributed reconstruction using Prefix Iteration was implemented. Besides, several complex partitioning strategies have been proposed that try to minimize the number of remote lookups. However, in [1], it turned out that when using Prefix Iteration, the execution of the algorithms is the main bottleneck. Prefix Iteration runs in $\mathcal{O}(n^2)$, where n is the number of retweets of a cascade. An equal distribution of all users among all nodes exploits parallelism and minimizes this time.

The system will use eight nodes for reconstruction. This is similar to a real setting. Let us make an estimation about the size of the complete social graph. On Twitter, there are 313 million monthly active users[2]. This number will be multiplied by two, assuming 626 million of total active users. The average follower count is 208 according to a study [2] made in 2012 based on 36 million profiles. Assuming that storing a single edge of the social graph takes 8 byte which is the size for a long value, we get $626\ million \cdot 208 \cdot 8\ byte \approx 1\ terabyte$. We might have to multiply this with another factor, but assuming that we can get machines with 128 to 256 gigabytes of main memory, eight nodes seem to be reasonable.

[1]http://storm.apache.org/, Homepage of Apache Storm, last lookup on 09.01.2017
[2]https://about.twitter.com/company, Twitter Statistics, last lookup on 12.02.2017

In this work, it is assumed that the social graph does not change over time. Evaluation was made with a fixed snapshot. Besides, we do not yet tackle node failures. In this work, the focus lies on minimizing the total time for reconstrucion. One of the major optimizations is the implementation of the User Iteration algorithm which runs in $\mathcal{O}(n \cdot k)$, where n is the number of retweets and k the average number of friends per user. Besides, more optimizations like more efficient datastructures, incremental reconstruction, batching of remote lookups and multithreading will be implemented.

This work is structured as follows:

- In section 2, other tools for processing large graphs are presented.
- Section 3 recapitulates previous work on the reconstruction framework at the chair of Web Science. At the end, the actual problem setting is described.
- In section 4, all improvements and optimizations are described.
- Section 5 describes how experiments were made, which metrics were used and the datasets being analyzed.
- In section 6, the results are presented. First, the baseline is shown, then optimizations are turned on step-by-step resulting in the final configuration. At the end, the results of a scaling experiment will be shown.
- Section 7 discusses the results.
- Section 8 gives possible steps for future work.

2 Related work

So far, there is not much work that tries to solve exactly the same problem. However, there are several tools that could be used to implement such a system. Two of them are presented in the following.

2.1 Turi/GraphLab

GraphLab was originally developed for executing incremental machine learning algorithms. It was started by Prof. Guestrin at Carnegie Mellon University in 2009 and acquired by Apple[3] in 2016. Under proprietary license, GrahLab Create is available as python package, offering classes for handling large graphs and time series data.

The SGraph[4] data structure allows arbitrary dictionary attributes on vertices and edges, provides flexible vertex and edge query functions. The *TimeSeries* object can be used to represent data streams with the presence of an implicitly temporal ordering. Besides, GraphLab offers packages[5] to deploy a distributed job on an Amazon EC2 or a Hadoop cluster.

2.2 Naiad

Naiad is a distributed system for executing data parallel, cyclic dataflow programs [3]. It was developed by Microsoft and aims at combining the advantages of stream processing frameworks (low latency), batch processing frameworks (high throughput) and graph processing frameworks (iterative and incremental computations). One disadvantage of a stream processor compared to a batch processor is that, since the stream operators are usually stateless and asynchronous, aggregation is rather difficult. However, the batch processor requires synchronization because every input must be available before computations can start. Besides, a stream processing framework like Apache Storm does not allow cycles in the computation (see 2.3 for a more detailed introduction to Storm).

In order to overcome these downsides, Naiad introduces a new computational model called *timely dataflow*. This is a computational model based on a directed graph in which stateful vertices send and receive logically timestamped messages along directed edges [3]. The logical timestamps allow to distinguish between several iterations of a loop. Every vertex must implement two callbacks: *onRecv(edge, message, timestamp)* and *onNotify(timestamp)*. If *onNotify(t)* is called, there may not be any more invocations of $onRecv(e, m, t')$ with $t' < t$.

Currently, some work is going on at the chair of Web Science that aims at implementing a system for the reconstruction of retweet cascades based on Naiad.

[3]https://www.bloomberg.com/news/articles/2016-08-05/apple-buys-ai-startup-turi-for-about-200-million, Bloomberg News, last lookup on 14.01.2017

[4]https://turi.com/products/create/docs/generated/graphlab.SGraph.html, Turi Documentation, last lookup on 14.01.2017

[5]https://turi.com/learn/userguide/deployment/pipeline-ec2-hadoop.html, Turi Documentation, last lookup on 14.01.2017

2.3 Apache Storm

The distributed system for reconstruction is based on the stream processing framework Apache Storm. In Storm, applications are built in shape of directed acyclic graphs, called *topologies*, consisting of *spouts* and *bolts*. Spouts are the source nodes and act as a source of data. The bolts, all inner and leaf nodes perform the processing logic. An edge $U \to V$ indicates a flow of data from U to V. Data is transferred via tuples, i.e. an ordered list of objects.

A Storm cluster consists of several machines running *worker* processes. A worker process runs at least one thread called *executor*. An executor runs at least one task of the same component, i.e. a spout or a bolt. In this work, only one task will be used per component. Figure 3 illustrates how node, worker, executor and components are related.

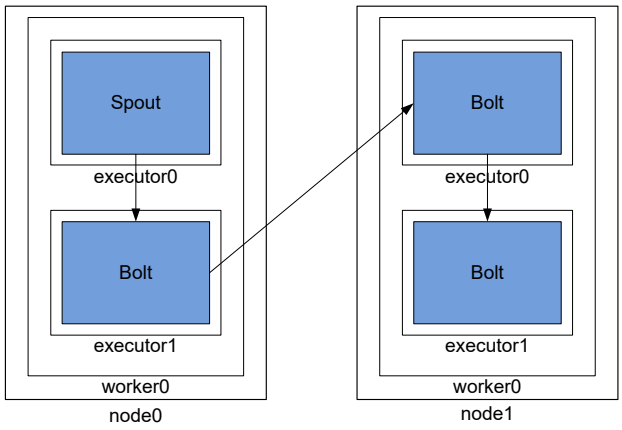

Figure 3: Apache Storm cluster with two nodes: each runs a worker, running two executors. Every executor runs exactly one task.

If several threads are used for the execution of a component, the input stream must be grouped. For this work, *fieldsGrouping* is most important. A *fieldsGrouping* ensures that tuples with the same value for a particular field name are always emitted to the same instance of a bolt [4]. There are other groupings like *shuffleGrouping* (random but equal distribution among all instances) and *allGrouping* (sends tuples to all instances). A *customGrouping* allows the programmer to define his own grouping function.

Figure 4 shows the "WordCount" topology which is often considered as an introduction to Storm: a spout is emitting tuples, where each tuple contains a line of text. A SplitWordBolt splits the line into single words and a WordCountBolt counts the occurrences of every word and emits these counts.

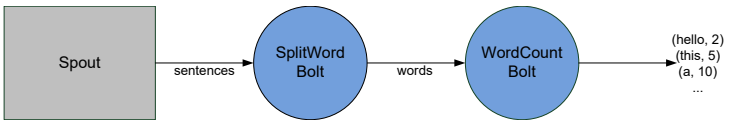

Figure 4: WordCount topology

3 Previous Work

This section covers previous work on the system that was made at the chair of Web Science.

3.1 Reconstruction algorithms

There are two basic reconstruction algorithms:

- **Prefix Iteration** iterates over the current list of retweets in this cascade. This list is called prefix. During iteration, the containment check is performed on the current user's set of followers.
- **User Iteration** iterates over the set of friends of the user who made the new retweet. Containment check is performed on the prefix.

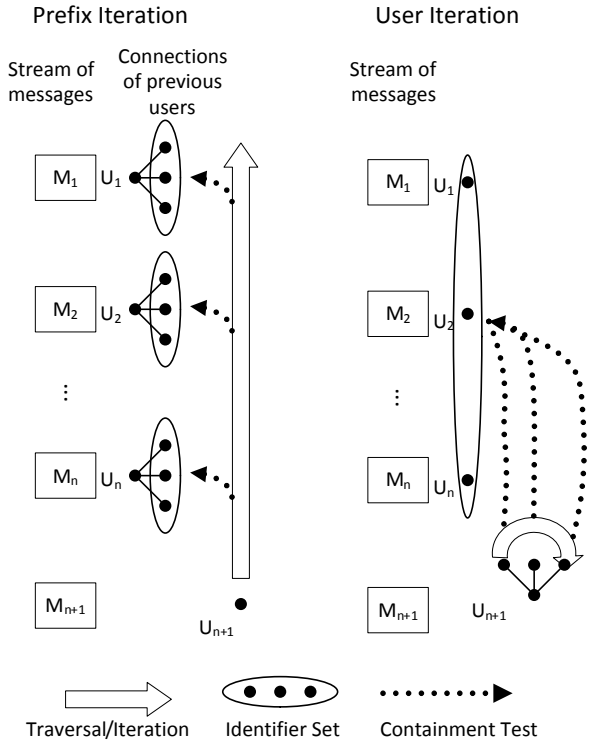

Figure 5: Prefix Iteration and User Iteration

Both algorithms are given the list of retweets and the social graph which is represented as a map $userId \rightarrow set\ of\ followers/friends$, depending on the algorithm being used. The algorithms will output an influence edge $(M_i, M_{n+1}, U_i, U_{n+1})$, if and only if, M_{n+1} was made after another retweet M_i and if U_{n+1} is following U_i.

The stream of messages $M_1, ..., M_{n+1}$ corresponds to the incoming flow of retweets. However, this stream must be grouped by some cascade identifier. Besides, the results must be stored at some point. The underlying system for reconstruction is described in the next subsection.

3.2 Centralized blocking reconstruction

In their master's project [5], Sättler and Ebner implemented the basis of the reconstruction framework: blocking reconstruction based on Apache Storm with only one reconstruction node for Prefix Iteration and User Iteration. Blocking means that before reconstruction can start, the system must wait until the cascade is finished. The topology is sketched in figure 6.

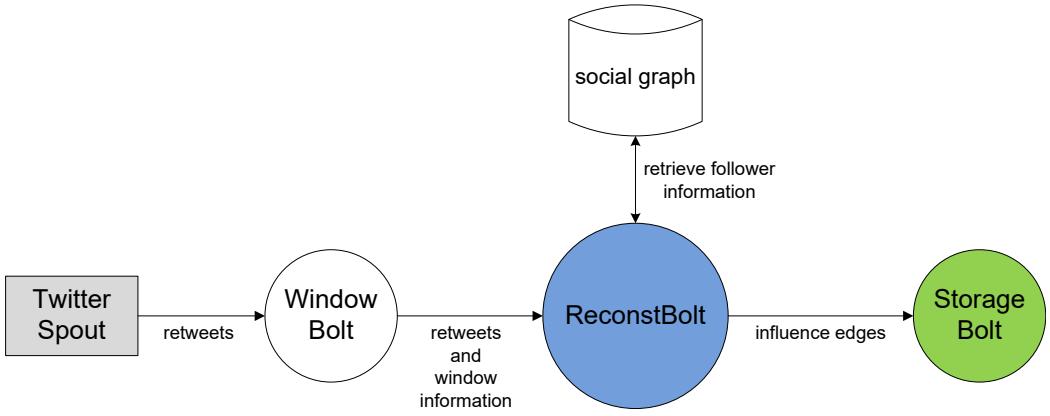

Figure 6: Topology for blocking reconstruction with a single ReconstructionBolt

A spout emits the retweet objects which can come from the Twitter API or some other source, like a JSON file. In order to simplify further processing logic, the WindowBolt separates the stream of retweets into *windows*. One window represents one cascade. A tuple of a window has the format *(windowId, windowState, payload)*. The windowId is the tweetId of the original tweet. The *payload* is a list of objects, representing the actual content. In the centralized scenario, it contains only the retweet object. The windowState has value

- **OPENING**, if this is the first retweet of this cascade
- **CONSUMING**, if this is not the first retweet of this cascade
- **CLOSING**, if the cascade is considered to be finished. In this case, the payload is empty.

The ReconstructionBolt performs the actual reconstruction. When it receives a tuple with windowState *OPENING*, it initializes some data structures and collects all upcoming retweets for this cascade. When it receives the tuple with windowState *CLOSING*, it starts reconstruction. During reconstruction, it accesses the social graph which is represented as a map: $userId \rightarrow friend/follower\ list$, depending on the algorithm being used. It is assumed that the social graph does not change over time. The ReconstructionBolt emits the influence edges to the StorageBolt which stores the edges on disk.

3.3 Distributed blocking reconstruction

Reconstruction with a single machine limits scaling. The complete social graph of Twitter consists of more than one terabyte. A usual machine cannot hold this amount in main memory. But the social graph can be partitioned among several machines. However, this creates some new problems:

- How is the graph being partitioned?
- Where to perform reconstruction?
- How is follower or friend information being exchanged?
- How much overhead does this create?

In his master's project [6], Ramos implemented a distributed social graph storage based on the database system *Cassandra*. One machine was running Cassandra and another node was issuing queries for a remote lookup. It turned out that these remote lookups are extremely costly. E.g. reconstruction of a cascade consisting of about 1,000 retweets using Prefix Iteration took about 25 seconds compared to a few milliseconds when the social graph is stored in main memory.

In his master's thesis [7], Huber implemented distributed reconstruction based on Prefix Iteration. Reconstruction of a cascade is always performed on the node containing the *root* of the cascade, i.e. the author of the tweet which is being retweeted. This node is also called *root node*. Due to the results of Ramos, Huber's intention was to perform as few remote lookups as possible. The reconstruction was blocking: before reconstruction can start, all retweets must be collected and every necessary follower information must be exchanged.

The topology for distributed reconstruction is sketched in figure 7. Spout and WindowBolt have the same functionality as described in 3.2.

3.3.1 RemoteRoutingBolt

The stream groupings of Storm usually have no knowledge about the contents of a tuple. Since reconstruction is performed on the root node, the retweet object must be accessed to determine the userId. The userId could be added to the tuple format which is not desired. In his outline, Huber was mentioning batch processing which will also be implemented in this work. A customGrouping could perform the userId lookup but it does not allow buffering of tuples. Thus, these requirements should justify introducing another bolt.

The *RemoteRoutingBolt* delivers retweets to the root node and leads the execution of the protocol for exchanging follower information. It holds several maps: $userToNodeMap:\ userId \rightarrow nodeId$ indicates the nodeId where a given user's follower information is stored. $windowToNodeMap:\ windowId \rightarrow nodeId$ indicates the root node for a given cascade. The $remoteUsersMap:\ windowId \rightarrow remoteUsers$ contains the set of *remote users* for each cascade. A remote user is a user whose follower information is not stored on the root node.

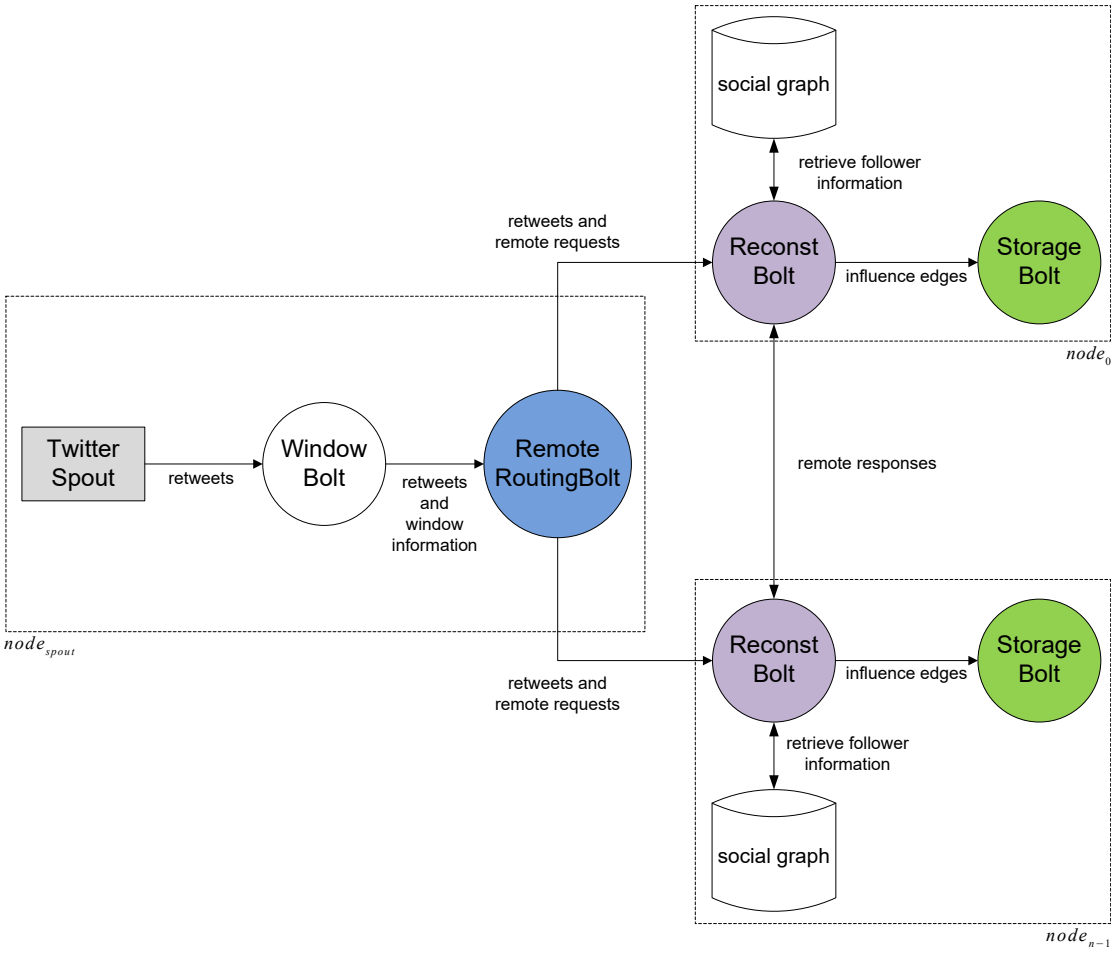

Figure 7: Topology for distributed reconstruction

For a tuple with window state *OPENING*, the RemoteRoutingBolt updates the windowToNodeMap. For tuples with window state *OPENING* or *CONSUMING*, the RemoteRoutingBolt emits the *remote requests*. A remote request has the following format: *(windowId, REMOTE_REQUEST, [rootNodeId, userId, remote userIds, remote user screen names])* A remote requests makes the remote nodes sent every connection between *userId* and *remote userIds* to the root node.

After sending all remote requests, the RemoteRoutingBolt determines the node which stores the follower information of the current retweet's user. If this user is not stored on the root node, the remoteUsersMap is updated. Finally, the actual retweet is forwarded to the root node.

For a tuple with window state *CLOSING*, the RemoteRoutingBolt emits, *remote flushes* to all remote nodes which make them send their current message count to the root node. A remote flush has the following format: *(windowId, REMOTE_FLUSH, [rootNodeId])* Then, the bolt emits a *remote close* to the root node which indicates that the cascade is finished. A remote close has the following format: *(windowId, REMOTE_CLOSE, [number of remote requests])*. Figure 8 illustrates the protocol for exchanging follower information. Table 1 summarizes the messages of the protocol.

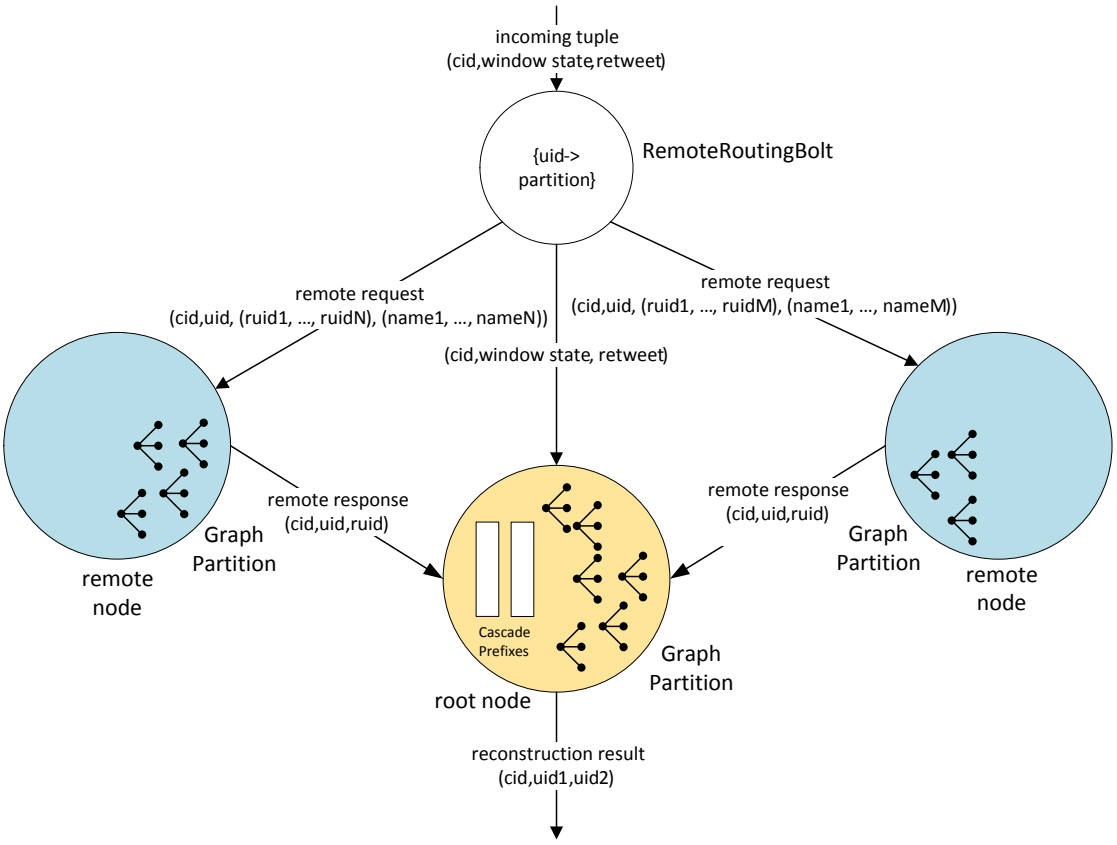

Figure 8: Stateless protocol for exchanging follower information

Message type	**Format**	**Semantic**
remote request	*(windowId, REMOTE_REQUEST, [rootNodeId, userId, remote userIds, remote screen names])*	tell root node all connections between *userId* and *remote userIds*
remote response	*(windowId, REMOTE_RESPONSE, [userId, remote userId, remote screen name, message count])*	*userId* follows *remote userId* *message count* requests were answered
remote close	*(windowId, REMOTE_CLOSE, [number of remote requests])*	no more retweets in this cascade, wait for remote responses
remote flush	*(windowId, REMOTE_FLUSH, [rootNodeId])*	no more remote requests, send your request count to root node

Table 1: Message types of the protocol for exchanging social graph information implemented by Michael Huber [7]

3.3.2 ReconstructionBolts

What is left is the behavior of the ReconstructionBolts. When receiving a remote request, the remote node iterates over the remote userIds, and for every connection *userId follows remote userId*, it emits a *remote response* to the root node. A remote response has the following format: *(windowId, REMOTE_RESPONSE, [userId, remote userId, remote user screenname, message count])* Remote responses are emitted only for positive matches. This avoids sending a lot of unnecessary messages. After emitting a remote response, the remote node resets its message count to zero. If there was no match, the message count is incremented.

When receiving a remote close, the ReconstructionBolt on the root node updates a map $windowId \rightarrow remote\ request\ count$, and waits until all remote responses have been received. When it receives a remote response, it inserts *userId follows remote userId* into its partition of the social graph. Besides, it adds *message count* to the value stored in another map: $windowId \rightarrow response\ count$. When this value is equal to the value being received from the remote close, reconstruction starts.

Since remote responses are only sent for positive matches, the ReconstructionBolt on the root node might wait forever. It is possible that the last remote request did not have any positive matches. That is where the remote flush comes into play: When the ReconstructionBolt on a remote node receives a remote flush, it sends its current message count to the root node. When reconstruction starts, the sum of the *message counts* from all responses is equal to *number of remote requests* which was contained in the remote close message.

3.4 Social graph partitioning strategies

With the results of Ramos [6] in mind, Huber assumed that remote lookups should be avoided as far as possible. In order to evaluate his protocol, he assumed that we have perfect knowledge about the retweeting activities of all users. He proposed a partitioning strategy called *community-based partitioning* that tries to allocate users that are following each other on the same partition. Huber made his experiments with the dataset of the Olympics 2012. In previous work, the whole dataset was reconstructed and from these results, the *activity graph* was created. In this graph, an edge $U_1 \rightarrow U_2$ with weight w indicates that U_2 might have been influenced w times by U_1.

In his bachelor's thesis [8], Ganz created a tool which converts the activity graph into a format that is suitable for the graph partitioning toolsuite METIS[6]. This tool creates a min cut partitioning in a very short time. In [1], two other partitioning strategies have been proposed:

- **Min cut partitioning of social graph:** Creating a min cut partitioning of the social graph seems very intuitive. In the social graph, edges are unweighted.

[6]http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, homepage of METIS, last lookup on 02.01.2017

- **Naive random partitioning:** A naive random distribution was created with the bash script provided in the appendix. This yields an equal distribution of all users. However, the number of edges is not considered: it could happen that the amount of main memory needed for two partitions differs significantly from each other.

3.5 State-based protocol, evaluation

In the master's project [1], the protocol for exchanging follower information was optimized by introducing a state at the remote nodes. The protocol proposed by Huber, made the RemoteRoutingBolt send the same list of remote userIds over and over again in every remote request. By introducing a local prefix on the remote nodes, the size of a remote request can be reduced to a single userId: *(windowId, REMOTE_REQUEST, [rootNodeId, userId])*

Besides, the screen name was removed from the processing logic, since everything is based on the userId. The new protocol is sketched in figure 9. Changes are marked in red: the remote requests are only sending a single userId and the remote nodes are holding local prefixes.

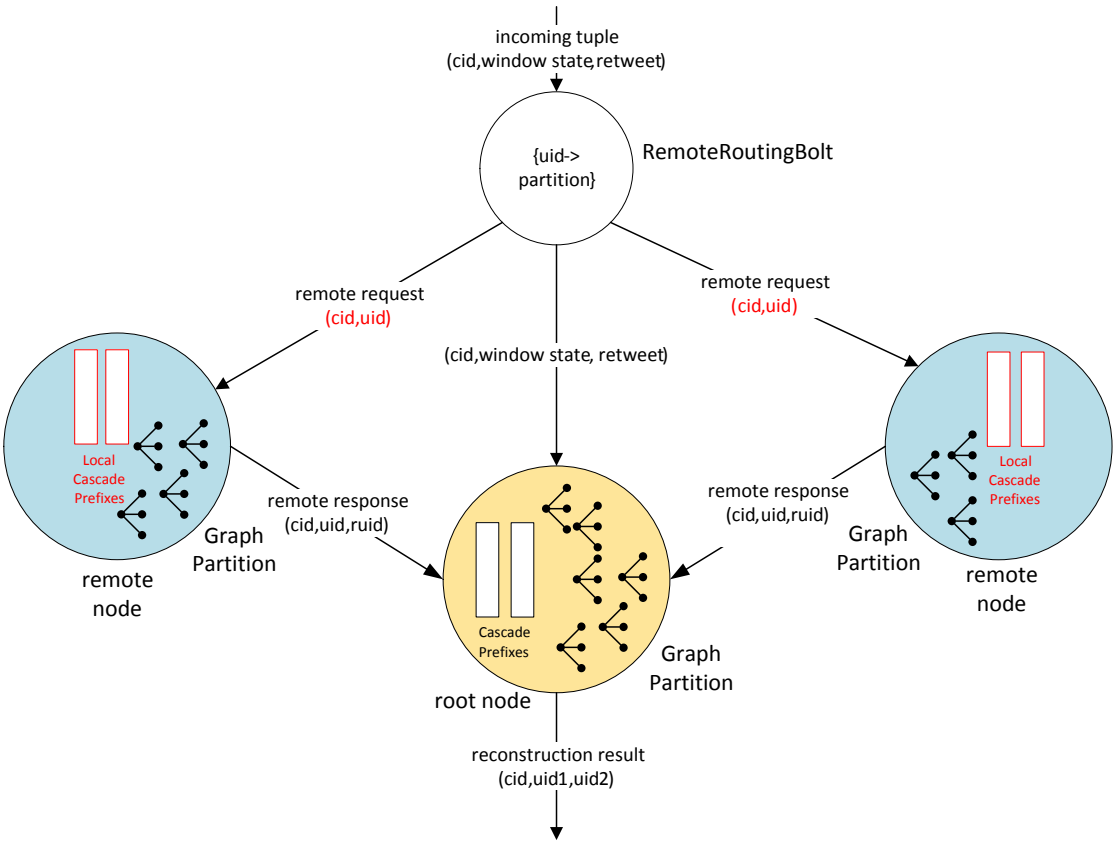

Figure 9: State-based protocol for exchanging follower information

Every ReconstructionBolt holds a map $windowId \rightarrow set\ of\ userIds$, representing the current local prefix of the cascade. For every remote request, the ReconstructionBolt iterates over this local prefix and it emits all remote responses to the root node. Then, the ReconstructionBolt determines whether the user identified by userId is contained in its partition of the social graph. If so, it adds this userId to its local prefix.

Besides, an evaluation of the system was made in [1]. Introducing a local prefix at the remote nodes was significantly reducing latency. With this improvement, all social graph partitioning strategies were compared. It turned out that an equal distribution of the users among all machines minimizes the execution time for the Prefix Iteration algorithm. The comparison between a min cut distribution on the activity graph and a random distribution in terms of reconstruction time is shown in figure 10.

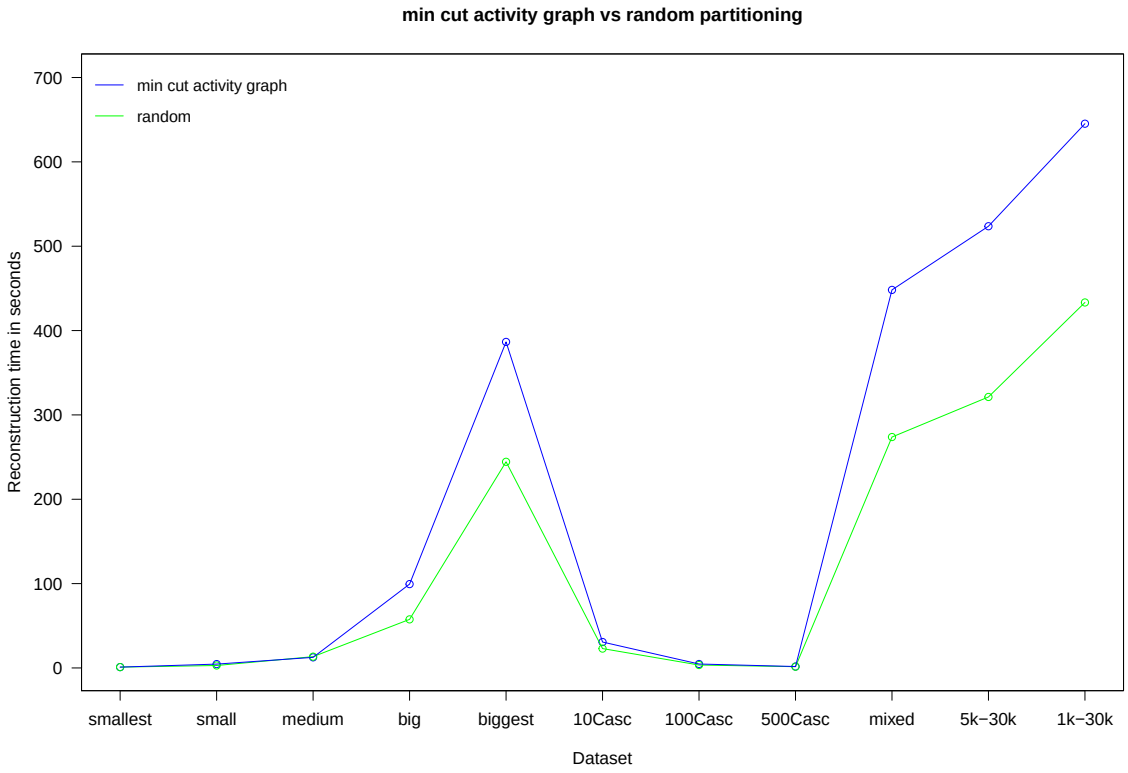

Figure 10: Results from [1]: a random distribution outperforms a min cut partitioning in terms of reconstruction time

The system is not performing a real merge of influence edges by directly emitting the remote responses to the StorageBolt. The root node re-iterates over all retweets, where it accesses local edges and remote edges. Remote edges came via remote response. By an equal distribution of all users, parallelism is maximized and the probability that a user's follower information is stored locally is just $\frac{1}{nodeCount}$. That means that a lot of set containment checks can be skipped for those users that represent a leaf node in the influence graph. Since no remote response will be sent for these users, they are also not contained in the remote edges. When using a partitioning strategy that tries to minimize the number of remote users, these checks must be performed.

3.6 Open questions

However, after the first evaluation of the system, behavior was understood only for single cascades. For datasets consisting of several cascades, there was a lot of time that could not be explained. Since only reconstruction time was measured explicitly, there was no further understanding about what is going on, when the system is not reconstructing. Total time was defined as the time span between the first node receiving a message and the last node finishing reconstruction of the last cascade. Reconstruction time was added up for all nodes. Figure 11 shows the results for four datasets. The metric *message or idle time* := *total time* - *reconstruction time* was taking the major part. From a total time of about 2,000 seconds only about a forth could be explained.

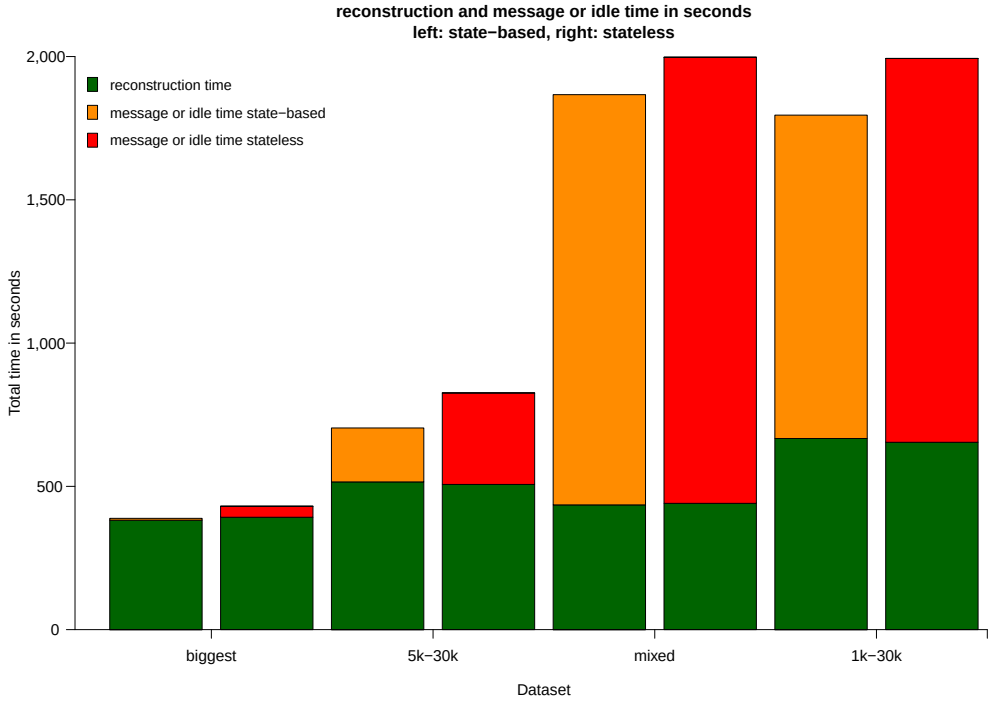

Figure 11: Results from [1]: only about a forth of total time could be explained

After the evaluation of distributed blocking reconstruction using Prefix Iteration was made, the following questions were open:

- What happens during the remaining 1,500 seconds?
- How does User Iteration perform compared to Prefix Iteration?
- How much overhead is created by incremental reconstruction?
- How does the system scale and what are the remaining bottlenecks?

In this work, most answers to these questions will be given. Besides, the system will be optimized as far as possible using more efficient datastructures and exploiting all available resources.

4 Optimizations

This section covers all optimizations that were made on the application. First, the User Iteration algorithm will be introduced. Second, incremental reconstruction and batching will be implemented. Third, a balanced partitioning strategy will be presented. Fourth, more efficient datastructures for representing the retweet tuples will be presented. Fifth, some small scale changes will be shown and multithreading will be implemented as the last optimization. At the end of this section, some points about recovery will be discussed.

4.1 User Iteration

The User Iteration algorithm is described in algorithm 1. It iterates over the set of friends of the current retweet's user. The time complexity for Prefix Iteration was $\mathcal{O}(n^2)$, where n is number of retweets of the cascade. User iteration runs in $\mathcal{O}(n \cdot k)$, where k is the average number of friends.

Algorithm 1 User Iteration

```
1:  input: retweets,
2:  localEdges : user → set of friends,
3:  remoteEdges : user → set of friends
4:  output: edges(parentId, parentTweet, childId, childTweet)
5:  // USER(retweeet): user of retweet
6:  // TIME(retweeet): timestamp of retweet
7:  // retweeets(user): retweets made by user
8:  for (i = 0; i < size(retweets); i++) do
9:      currentTweet = retweets[i]
10:     user = USER(currentTweet)
11:     if (localEdges containsKey user) then
12:         friends = localEdges(user)
13:     else
14:         friends = remoteEdges(user)
15:     foreach friends as friend do
16:         friendTweets = retweets(friend) // usually only one tweet
17:         foreach friendTweets as friendTweet do
18:             if TIME(friendTweet) < TIME(currentTweet) then
19:                 emit edge(friend, friendTweet, user, currentTweet)
20:         end foreach
21:     end foreach
```

The average number of friends in Twitter was 102 according to a study [2] made by "Beevolve" in 2012. Thus, User Iteration should perform better for longer cascades. However, for short cascades with less than 100 retweets, Prefix Iteration is expected to perform better. In 6.2.1, the results for a set of small cascades (6 to 100 retweets) are shown.

4.2 Incremental reconstruction

So far, reconstruction was blocking: the root node must wait until the cascade is finished and all necessary follower information has been collected. In this work, the end of a cascade is implicitly given by a flush tuple which is emitted by the spout after reaching the end of the dataset. During reconstruction at realtime, results should be visible after a given number of retweets or after a given number of seconds. When at least one condition is satisfied, the system should reconstruct as far as possible. For this scenario, synchronization becomes more complicated: how does the root node know, if it has received all relevant remote responses from the remote nodes?

4.2.1 Assumptions

For synchronization, it is assumed that messages on the same channel are processed in the same order in which they were sent. Besides, it is assumed that no message gets lost. Without these two assumptions, synchronization would become more costly and creating a lot of overhead. For exactly-once message processing of tuples with ordering-guarantee, Storm provides so called *trident*[7] topologies. This extension executes tuples in several batches which can be committed or aborted, similar to a database system. However, since trident builds another abstraction based on Storm primitives, it might create a large overhead. Implementing this extensions would go beyond the scope of this work.

4.2.2 Protocol

In order to invoke reconstruction after a given number of retweets or a certain period of time, there must be some bolt which has knowledge of this global state. Thus, it could be part of the RemoteRoutingBolt. However, in further optimizations, several threads of the RemoteRoutingBolt might be executed in parallel. Incremental reconstruction cannot be invoked from one of these instances, because every instance will only process parts of the data stream.

Thus, a new bolt called *WindowIncrementalBolt* was introduced. This bolt counts the number of retweets and the time that has passed since the last reconstruction was made. It has two parameters: *retweet_count* and *milliseconds*, indicating when reconstruction should be invoked. The WindowIncrementalBolt will be placed between the WindowBolt and the RemoteRoutingBolt. The new topology is sketched in figure 12. In order to invoke reconstruction, the WindowIncrementalBolt emits a tuple *(dummyId, RECONSTRUCT, empty)* to all instances of the RemoteRoutingBolt. Upon receiving such a tuple, the RemoteRoutingBolt broadcasts *(windowId, REMOTE_FLUSH, [rootNodeId])* to the remote nodes for those cascades that were part of the last block. For that, it holds a set of windowIds which is updated for every retweet. After broadcasting remote flushes for all cascades, it clears this set.

[7] http://storm.apache.org/releases/1.0.1/Trident-tutorial.html, Storm Trident Documentation, last lookup on 29.12.2016

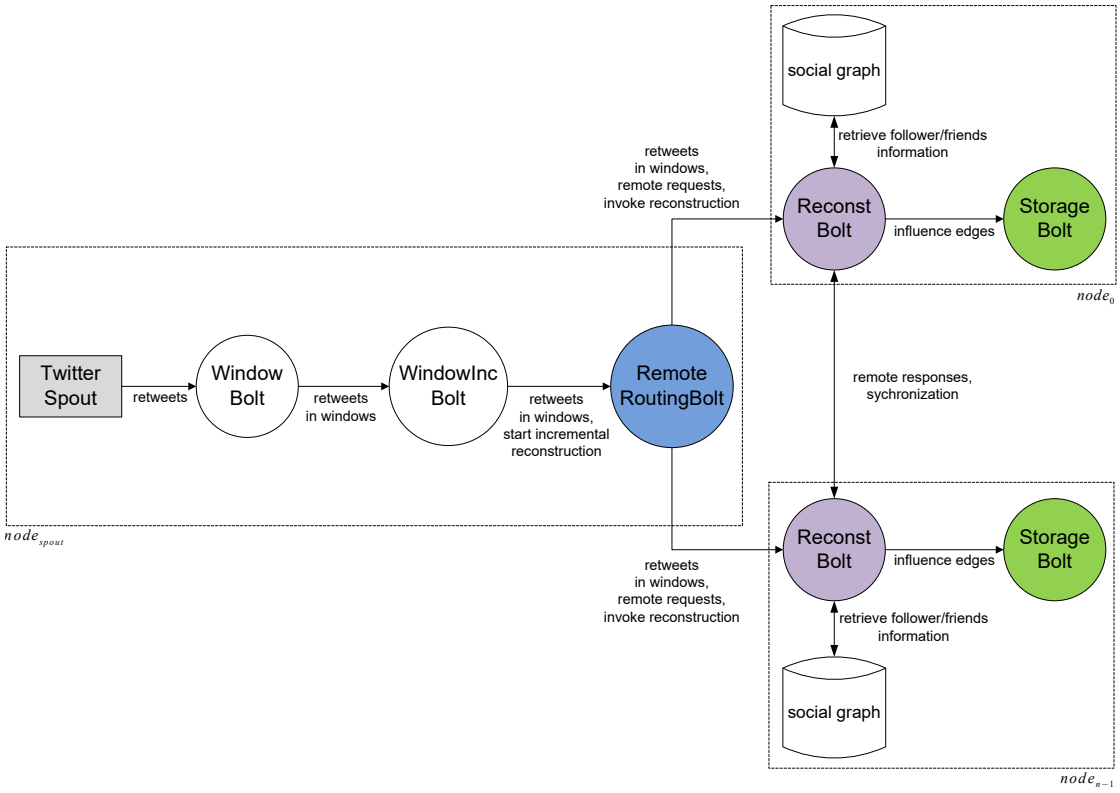

Figure 12: Topology for distributed incremental reconstruction

When a remote node receives a remote flush, it sends a tuple *(windowId, REMOTE_FLUSH_RESPONSE, tweetCount)* to the root node. A remote flush response indicates that the remote node has answered *tweetCount* remote requests. Upon receiving a remote flush response, the root node updates a nested map $windowId \rightarrow remoteNodeId \rightarrow tweetCount$ containing the current tweetCount for reach remote node. The, it updates another map $windowId \rightarrow lowestTweetCount$ by iterating over the tweetCount map. Besides, it holds a map $windowId \rightarrow currentTweetIndex$ which indicates the current reconstruction progress for a cascade. When $currentTweetIndex < lowestTweetCount$ and at least $lowestTweetCount$ retweets have already been received, reconstruction is performed until $lowestTweetCount$ and the $currentTweetIndex$ is set to this value.

Besides, the root node tries to start reconstruction, when it receives a tuple with windowState *CLOSING*, like in blocking reconstruction.

4.3 Batch remote requests and responses

In the experiment made in 6.1.5, it was shown that the RemoteRoutingBolt, more precisely sending remote requests is slowing down the retweet transfer rate by a factor of almost ten. Batching remote requests reduces the number of messages. Broadcasting is already costly when using eight nodes for reconstruction. When using 16 or 32 nodes, this will even get worse. Another advantage of batching is that less data is being transferred when batching is done per cascade.

Note: batching of remote requests and incremental reconstruction can be used independently. Incremental reconstruction just means that reconstruction is invoked after a certain condition. Batching of remote requests can also be used without enabling incremental reconstruction.

Instead of emitting remote requests directly, they are buffered until a given amount of retweets has been collected. However, it is intuitive to use incremental reconstruction together with batching. E.g. wait until 5,000 retweets have been received, emit all remote requests at once and invoke reconstruction.

4.3.1 Per cascade

Batching remote requests per cascade does not require a lot of changes to the protocol. The RemoteRoutingBolt is given a parameter B, indicating the batch size. When B remote requests have been collected for a cascade or the cascade is finished, they are emitted to the remote nodes. A remote request will have the following format:
(windowId, REMOTE_REQUEST, [rootNodeId, userIds]).
Upon receiving a remote request, the remote node iterates over the list of userIds. For every userId, it performs the same steps as when receiving an unbatched remote request, but it collects the responses in two lists called *parents* and *childs*. When iteration is finished, the remote node emits a remote response of the format:
(windowId, REMOTE_RESPONSE, [childIds, parentIds]).
When receiving a remote response, the root node just iterates over the child and parent list and inserts these edges into its map for remote edges.

Assume, we use batching per cascade with B=*3* and we have buffered the following remote requests:

```
format: (windowId, REMOTE_REQUEST, [rootNodeId, userId])
(1234567890, REMOTE_REQUEST, [0, 36464346])
(1234567890, REMOTE_REQUEST, [0, 53453561])
(1234567890, REMOTE_REQUEST, [0, 35345327])
```

When using batching per cascade, we will send:

```
(1234567890, REMOTE_REQUEST, [0, [36464346, 53453561, 35345327]])
```

This way, we avoid sending two long values for the windowId, two windowStates (windowState will be represented as an integer) and two integers for the rootNodeId.

4.3.2 Global

Global batching makes communication more complex, since a tuple is no longer associated with a single cascade. Given a parameter B, the RemoteRoutingBolt buffers the remote requests for all cascades. As soon as, B remote requests have been collected, it emits a global remote request to all nodes. A global remote request will have the following format: *(dummyId, REMOTE_REQUEST, [rootNodeIds, userIds, windowIds])*

Upon receiving a global remote request, the node iterates of the three lists which are of the same length. If the current rootNodeId equals to its own nodeId, it skips this request. If not, it buffers the responses in three lists: windowIds, parentIds and childIds. For every windowId, it updates a map $windowId \rightarrow requestCount$, to keep track of the progress for this cascade. Finally, the node emits a remote response of the following format:
(dummyId, REMOTE_RESPONSE, [childIds, parentIds, windowIds, map: windowId $\rightarrow$ *tweetCount, own nodeId]).* The lists of parents, childs and windowIds represent the buffered responses. The map and the nodeId must be added to inform the root node about the current number of answered remote requests. This batching strategy does not require separate flush tuples, since there will only be one global remote request for each batch.

Upon receiving a remote response, the root node iterates over the lists of childIds, parentIds and windowIds and inserts these edges into its map containing the remote edges. If incremental reconstruction is enabled, it will iterate over the entries of the map and perform reconstruction for these windowIds.

4.3.3 Message overview

Table 2 gives an overview about the types of messages in the new protocol.

Message type	**Format**	**Semantic**
remote flush	*(windowId, REMOTE_FLUSH, [rootNodeId])*	tell root node current requestCount
remote flush response	*(windowId, REMOTE_FLUSH _RESPONSE, [nodeId, tweetCount])*	tweetCount requests were answered. try to reconstruct
global remote request	*(windowId, REMOTE_REQUEST, [rootNodeIds, userIds, windowIds])*	iterate over lists send connections to root node
global remote response	*(dummyId, REMOTE_RESPONSE, [childIds, parentIds, windowIds, map: windowId* $\rightarrow$ *tweetCount, own nodeId])*	insert connections if incremental reconstruction enabled: try to reconstruct
cascade remote request	*(windowId, REMOTE_REQUEST, [rootNodeId, userIds])*	batched remote request, iterate over userIds, perform lookup
cascade remote response	*(windowId, REMOTE_RESPONSE, [childIds, parentIds])*	batched response insert edges in remote edges map

Table 2: Message types for incremental reconstruction and different types of remote request batching

4.4 Balanced random partitioning strategies

Since a random distribution does not consider any balance in terms of users and number of edges, a new partitioning strategy was created. It reads the list of users and greedily assigns the current user to the partition which contains the fewest edges. The number of edges is either the followers or the friends count of the current user. This creates a random, but balanced partitioning.

This partitioning was created for both, the followers and the friends count. The partitionings will be called *greedy_friends* and *greedy_followers*.

4.5 Retweet tuple datastructure

This section describes two major improvements on the datastructures that are used for representing a tuple *(windowId, windowState, payload)*.

4.5.1 Twitter API

When downloading retweets from the Twitter API, they are represented in JSON format. The default representation contains many attributes that are not needed for reconstruction, like profile image or profile background color which can be projected out. A complete retweet object is provided in the appendix.

```
{
   "created_at":1344852416000,
   "text":"RT @Louis_Tomlinson: Last night was unbelievable. I cannot
       believe we had the honour to perform at the Olympics! Has to be one
        of the be ...",
   "id":234954178848505856,
   "retweeted_status":{
      "created_at":1344852407000,
      "text":"Last night was unbelievable. I cannot believe we had the
         honour to perform at the Olympics! Has to be one of the best
         nights off my life!",
      "id":234954143758946304,
      "user":{
         "id":84279963,
         "screen_name":"Louis_Tomlinson"
      },
      "retweet_count":1
   },
   "user":{
      "id":299033003,
      "screen_name":"_BigTime1D_"
   },
   "retweet_count":1
}
```

4.5.2 Custom retweet format

Even the projected retweet format is somehow complex. Serializing a map creates some overhead, compared to serializing a flat list of values. Thus, a custom retweet format with custom serializer based on *Kryo*[8] was implemented by Prof. Fischer. Kryo gives much better performance than the default Java serializer.

```
1 public class CustomRTReconstFormat {
2   public long tweetid, timestamp, userid, originaluserid,
       originaltimestamp, cascadeid;
3   public String text;
4   public CustomRTReconstFormat(Map<String,Object> jsonMap) {
5     // construct a custom retweet object from Map representation
6   }
7 }
```

4.5.3 WindowState serialization

The windowState is currently represented as an enum:

```
1 public enum WindowState { CLOSING, OPENING, CONSUMING,
    REMOTE_REQUEST, REMOTE_REPONSE, REMOTE_FLUSH, REMOTE_CLOSE...
2 }
```

It turned out that serializing an enum is very costly. Enum is not a primitive datatype like integer or double. The following paragraph taken from the java documentation[9], describes the process of serializing an enum:

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form. To serialize an enum constant, ObjectOutputStream writes the value returned by the enum constant's name method. To deserialize an enum constant, ObjectInputStream reads the constant name from the stream; the deserialized constant is then obtained by calling the java.lang.Enum. valueOf method, passing the constant's enum type along with the received constant name as arguments.

This is not expected, serializing and parsing a string is clearly not necessary, in order to distinguish between several windowStates. Thus, a new class *WindowState* was written which contains several integer constants. A snippet of the class is provided in the following listing. The new representation maintains readability of the code and improves on the serialization performance. The effect of replacing an enum with an integer are shown in 6.1.

[8] https://github.com/EsotericSoftware/kryo, GitHub account of Kryo, last lookup on 16.01.2017

[9] http://docs.oracle.com/javase/8/docs/platform/serialization/spec/serial-arch.html#6469. Java Documentation, last lookup on 30.12.2016

```
1 public class WindowState {
2   public static final int OPENING = 0;
3   public static final int CLOSING = 1;
4   public static final int CONSUMING = 2;
5   ...
6 }
```

4.6 Small scale changes

4.6.1 AtomicIntegers as counters

Several bolts are holding maps $windowId \rightarrow count$, e.g. counting the number of reweets for a cascade. Since maps are based on objects, the following code would instantiate a new Integer object for every increment:

```
1 HashMap<Long, Integer> map;
2 ...
3 int count = map.get(windowId);
4 count++;
5 map.put(windowId, count);
```

To avoid new instantiation, an AtomicInteger was introduced which is created only once when receiving the first tuple of the cascade.

```
1 HashMap<Long, AtomicInteger> map;
2 map.put(windowId, new AtomicInteger(0));
3 ...
4 map.get(windowId).incrementAndGet();
```

4.6.2 Dealing with multiple retweets of same user

The reconstruction algorithm uses a map $userId \rightarrow retweet$ which stores the retweet per userId. It is possible that a user retweets the same tweet several times. In this case, the algorithm would only know about last retweet, since previous retweets would have been overwritten. So far, every potential influence edge is desired which corresponds to the *complete* model.

In order to deal with multiple retweets, the datastructure was changed to a map: $userId \rightarrow set\ of\ retweets$. The reconstruction algorithms were changed, so that they perform another iteration over this set.

4.6.3 Removing statistics calculation

The current implementation calculates some statistics, like: what is the percentage of influence edges coming from the root? This statistics are not part of the actual reconstruction and they can still be calculated when reconstruction is finished. Statistics creation was disabled to optimize on reconstruction time.

4.7 Multithreading

So far, every machine was running every component in only one thread leaving a lot of resources unused. The main difficulty is sharing the data structures among several threads and achieving an equal workload for each thread. The latter can be achieved by an equal distribution of the tuples. Besides, the distribution must be disjoint and always assign tuples of the same windowId to the same thread, since the bolts carry a lot of state. That is why the tuples will be grouped by the windowId. Since the maps (social graph and userToNodeMap) are only written once during initialization of the system, they can easily be shared among several threads.

4.7.1 RemoteRoutingBolt

The RemoteRoutingBolt is the heart of the system. It forwards retweets and coordinates the exchange of follower or friend information. Therefore, it must emit thousands of messages. Making this bolt running in several threads seems reasonable. For an equal distribution of tuples, the native fieldsGrouping will be used on the windowId. The userToNodeMap will be shared among all threads. All other state will be kept separate for each instance.

4.7.2 ReconstructionBolt

The ReconstructionBolts can exploit multithreading by splitting the set of cascades among several threads. Thus, the social graph must be shared. Since all accesses to the social graph are read-only, this can easily be done with a singleton. We must only make sure that only one thread is loading the social graph. But for further improvements like allocating new users for the social graph, locking must be implemented which creates some overhead. The tuple grouping is more complicated. A thread that was assigned a cascade with a given windowId must also receive the remote responses that belong to this cascade.

4.8 Recovery considerations

In this section, several optimizations have been proposed. This subsection discusses their impact on recovery. So far, it is assumed that the nodes do not crash. However, for future work, this assumption will be weakened, such that the system must be able to tolerate a given number of node failures.

In a stream processing system, state should be avoided as far as possible. Every map which is held on the bolts and which is not once initialized and never changed, like the social graph or the userToNodeMap is potential non-recoverable state. The state-based protocol which was implemented in [1], incremental reconstruction and batching of remote requests introduce a lot of non-recoverable state on RemoteRoutingBolt and ReconstructionBolts.

In order to regularly save the state, it should not take too much space. Let us go through the state of the ReconstructionBolts and determine the size that would be needed while reconstructing the biggest dataset consisting of about 3.6 million retweets and 100,000 cascades (see 5.2). We assume that the cascades are equally distributed among all nodes, i.e. every node reconstructs about 12,500 cascades consisting of about 450,000 retweets. This assumption is reasonable when using an equal distribution of users among all nodes, see 6.2.6. WindowId and userId need 8 byte each, since they are represented as a long value. A retweet object consists of 6 long values + a string of at most $140 \cdot 2$ bytes, since Java uses UTF-16 in its string representation[10]. Thus, a retweet object will need at most $6 \cdot 8 + 140 \cdot 2\ bytes = 328\ bytes$. The state does not require too much space:

- retweet objects for reconstruction: $450,000 \cdot 328\ bytes \approx 140\ megabytes$
- local prefixes (userIds of all retweets, except "own" retweets): $7 \cdot 450,000 \cdot 8\ bytes \approx 25\ megabytes$
- several maps: $windowId \rightarrow integerCount$ which need $450,000 \cdot (8 + 4)\ bytes \approx 5\ megabytes$ each

Storm guarantees *at-least once processing* by giving the spout the possibility to replay tuples that have failed. A tuple can either be explicitly failed by a bolt or it can timeout, if there was no acknowledgement within a given number of seconds (default: 30 seconds). However, since the whole system is relying on the FIFO property, all cascades that were active during the crash had to be replayed from the beginning. This is very costly. Maybe it is more efficient, when the ReconstructionBolts regularly save their state to disk, and reload it during recovery. However, some of the latest retweets must be replayed anyway.

[10] https://docs.oracle.com/javase/8/docs/api/java/lang/String.html, Java Documentation, last lookup on 12.01.2017

5 Evaluation Methodology

This section describes the experiments that were made and which metrics were used to evaluate the optimizations of the previous section. Like in previous work, the retweet cascades are stored in a JSON file. Before emitting a tuple, the spout must parse the JSON file and instantiate a retweet object. In order to avoid bottlenecks for parsing and instantiation, a PreParseSpout was written by Prof. Fischer that creates a JSON buffer and preallocates the retweet objects. This spout was used for all experiments.

5.1 Metrics

This section describes the time metrics being used and explains what is happening on the nodes of the cluster. The main focus was on reducing total time. Nevertheless, for a complete understanding of the system, some other times are important as well.

5.1.1 Reconstruction with only one ReconstructionBolt

When using only one machine for reconstruction, the whole process can be devided into two phases as sketched in figure 13:

- **tweet receiving phase:** The ReconstructionBolt is just collecting the retweets and waits for the cascades to finish, so that reconstruction can start
- **reconstruction phase:** The ReconstructionBolt has collected every retweet, and the spout has emitted a flush tuple which closes every window implicitly. The node performs reconstruction for the cascades, one after the other. After every reconstruction, some time is needed for emitting and storing the influence edges.

The time spent on reconstruction is being measured and will be called *reconstruction time*. The *tweet receiving time* contains everything that happens between the first and the last retweet receive event. The *total time* contains everything that happens between receiving the first retweet of the first cascade and having finished reconstruction of the last cascade.

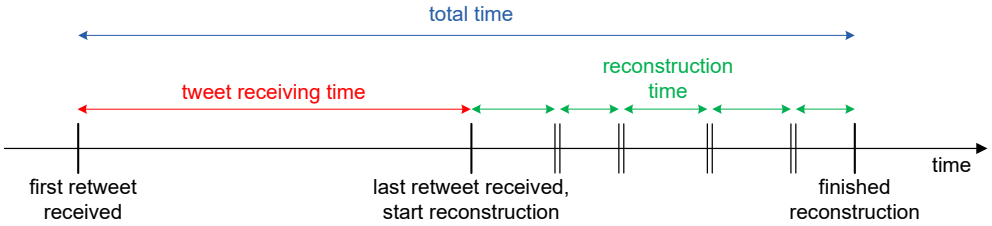

Figure 13: Centralized blocking reconstruction time metrics

5.1.2 Distributed blocking reconstruction

When several nodes are being used for reconstruction, metrics become more complex. Reconstruction time will be defined as the sum of all reconstruction times over all nodes. Since the social graph is distributed, the reconstruction nodes spent some time for answering the remote requests. This time is marked in orange and will be called *request response time*. This time is also summed up over all nodes.

The total time will contain everything between the first ReconstructionBolt receiving the first message and the last ReconstructionBolt finishing reconstruction for the last cascade. During this time span, at least one node is not idle and thus, the system can be called busy. However, by defining total time in this way, we are always measuring the outlier. Taking the point when something like 80 % of the nodes have finished would be another approach. The following figure sketches the important events at the nodes.

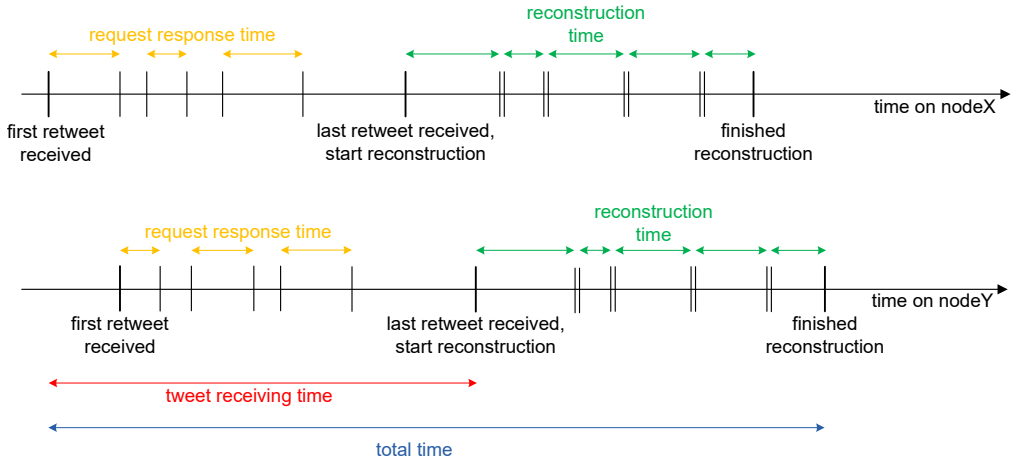

Figure 14: Distributed blocking reconstruction time metrics

5.1.3 Distributed incremental reconstruction

During distributed incremental reconstruction, the reconstruction nodes are doing everything interleaved. This is the most complex type for reconstruction. The metrics are sketched in figure 15. There are no phases that can be distinguished. Reconstruction time and request response time are summed up over all nodes. The tweet receiving time contains a lot of reconstruction time and request response time. When optimizing on this setup, the main focus should be on the total time. This is still a accurate and meaningful measure. The costs for exchanging follower information cannot be measured directly.

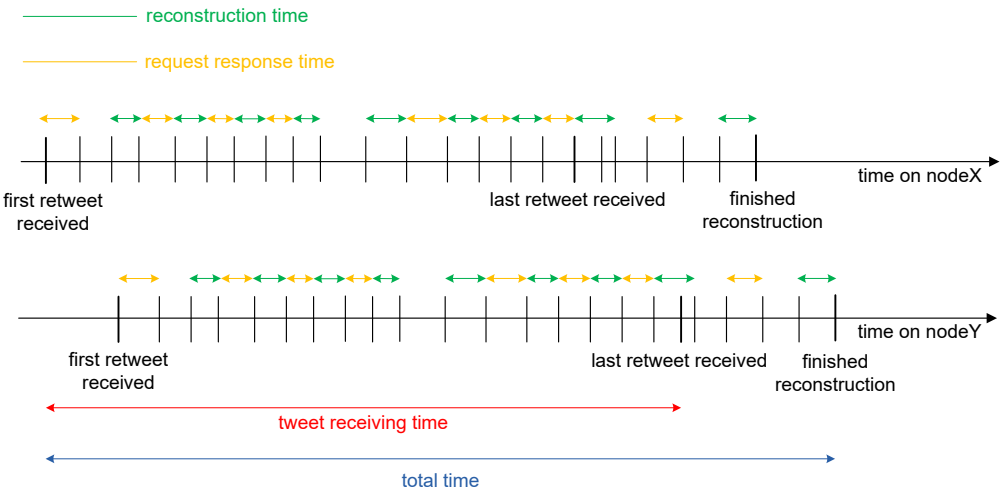

Figure 15: Distributed incremental reconstruction time metrics

Besides, two more metrics will be defined to measure the overall throughput:

- **retweet transfer rate**: $\frac{|retweets|}{tweet\ \ receiving\ \ time}$
- **reconstruction rate**: $\frac{|retweets|}{total\ \ time}$

5.2 Datasets

Like in previous work, the cascades were taken from the Olympics 2012. For every dataset, the cascades where sorted in temporal order and unnecessary attributes were projected out. Since single cascades are already well understood, larger datasets will be used. The datasets *mixed* and *1k-30k* from [1] will be used again. *Mixed* consists of three sets consisting of 10, 100 and 514 cascades with a total of about 60,000 retweets each. The biggest cascade consisting of 60,451 retweets is also contained in *mixed*. *1k-30k* consists of all cascades with at least 10,000 and at most 30,000 retweets.

Besides, two new datasets were created out of the Olympics: *6-100* consisting of all cascades with at least six and less than 100 retweets and *bigger4* consisting of every cascade with at least five retweets. This dataset contains all other datasets. The distribution of *bigger4* in logscale is plotted in figure 16. *6-100* has no common cascades with *1k-30k* and *mixed*. However, *1k-30k* and *mixed* have ten cascades in common, where each cascade consists of about 6,000 retweets. Table 3 gives more detailed information about the datasets.

Dataset	**Cascades**	**Retweets**	**Average retweets**	**Influence edges**
mixed	625	242,266	387.6	1,210,290
1k-30k	350	918,795	2,625.1	2,762,476
6-100	74,569	1,268,607	17.0	962,852
bigger4	99,561	3,683,332	37.0	7,286,620

Table 3: Statistics about the datasets being used

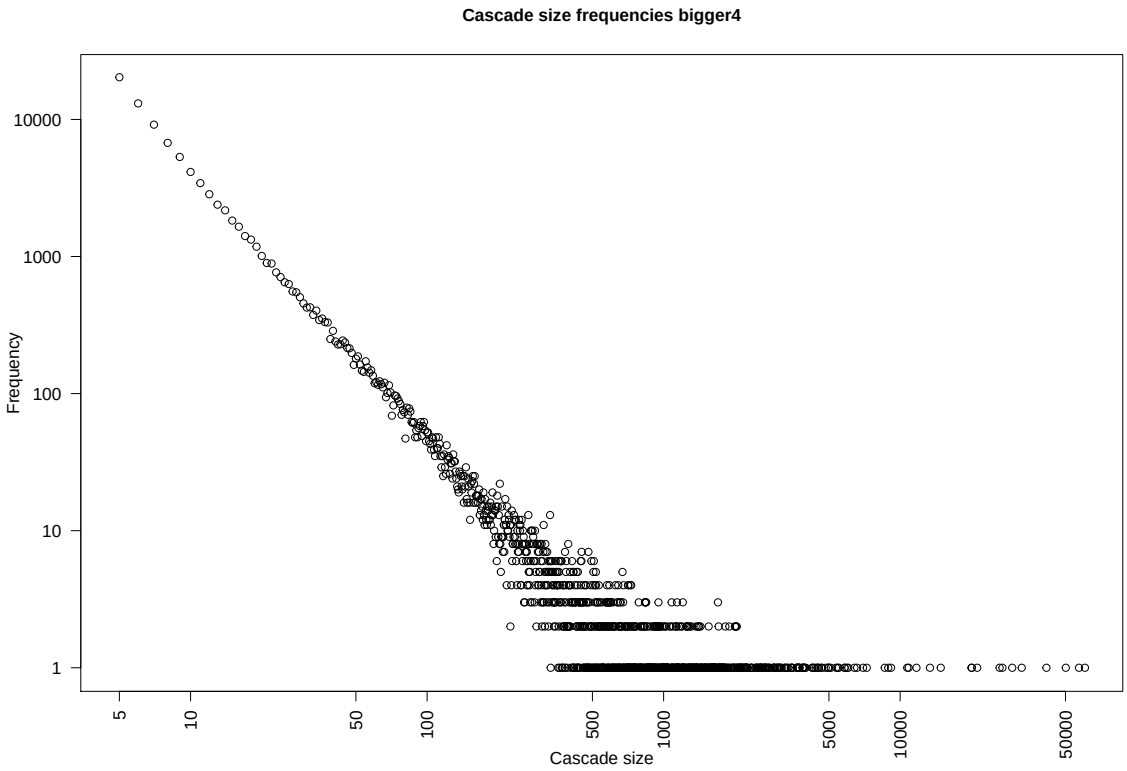

Figure 16: Frequencies of cascade sizes for bigger4

5.3 Reconstruction

5.3.1 Centralized reconstruction

For an evaluation that compares the reconstruction algorithms only, reconstruction was performed on a single machine. Since the social graph for the dataset *bigger4* needs more than 32 gigabytes of main memory, this experiment was performed on an Amazon EC2 instance *r3.8xlarge* which has 244 gigabytes of main memory.

5.3.2 Distributed reconstruction

Like in previous work, distributed reconstruction will be performed on the cluster of the chair for databases and information systems. This cluster consists of ten nodes, called dbisma01-dbisma10, each consisting of an Intel Xeon E5-2420 with twelve cores and 32 gigabytes of main memory. The nodes are connected via 1GBit links. Apache Storm was used in version 1.0.1. The configuration files of the nodes can be obtained from the appendix of [1]. The nodes were loading only those parts of the social graph that was needed for the reconstruction of the dataset. Only one topology was active at the same time. The topology was re-deployed for every experiment

5.4 Validation

For every dataset, the system should output the same influence graph in every configuration. Validation was made by reconstructing every dataset using centralized reconstruction and measuring the total number of edges. In every experiment, the number of total edges was compared to the corresponding value. Validation was satisfied for every experiment.

5.5 Expectations

- User Iteration should outperform Prefix Iteration for longer cascades. Prefix Iteration should perform better for cascades that are smaller than the average number of friends.
- Turning on incremental reconstruction should lead to earlier results. However, by decreasing the block size, throughput will suffer.
- The custom retweet format with windowStates realized as constant integers should perform best
- Batching of remote requests will decrease the total time and reduce the load on the RemoteRoutingBolt
- Multithreading on ReconstructionBolts and RemoteRoutingBolt will give a significant increase in throughput.
- A naive random partitioning performs best when using Prefix Iteration. When using User Iteration, users that have no influence cannot be eliminated, because remote edges are represented as a map $userId \rightarrow friend\ list$. The containment check must still be performed for these users. However, it must not be performed for users that were not influenced by some other user. This case is possible: imagine a user that does not follow anyone, but regularly visits other user's profiles.

6 Results

In section 4, several optimizations were mentioned. In this section, one parameter will be optimized and the best value will be reused in the following experiments. First, we will look at the transfer rate of the retweet tuples $(windowId, windowState, payload)$.

6.1 Tranfer rates of retweet tuples

In this section, the most efficient datastructure for representing the retweet tuples will be determined by measuring the throughput in tuples per second. Besides, the upper bound for transferring retweet objects will be determined by rebuilding the reconstruction topology step by step.

The first scenario is the simplest one: two machines, running one worker each. The first node runs a spout emitting retweets only. The second node runs a sink bolt which does nothing except of measuring received tuples per second. 30 values were measured and from that, average, median and standard deviation were calculated.

6.1.1 Spout and sink

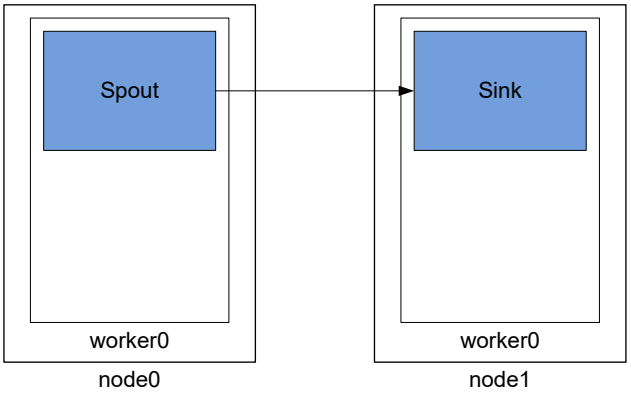

Figure 17: Two nodes: spout and sink node

Tweet format	**Average**	**Median**	**Standard deviation**
Twitter API	21,498	23,430	5,356
Twitter API projected	247,510	257,731	44,440
Custom	620,935	625,000	44,466

Table 4: Tuples per second for spout and sink

Using the unprojected representation from the Twitter API slows down by a factor of 30. Since the custom retweet format performs best, it will be used in the following experiments.

6.1.2 Spout, WindowBolt and sink

The WindowBolt was put between the spout and the sink. It wraps the retweet objects into windows by adding two values: windowId and windowState. Adding a WindowBolt decreases the transfer rate by a factor of more then seven. Let us see where this comes from.

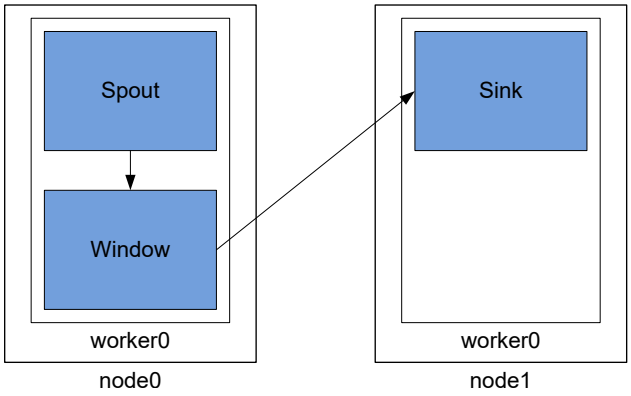

Figure 18: Two nodes: Spout, WindowBolt and Sink

Tweet format	Average	Median	Standard deviation
Custom	81,860	89,767	25,161

6.1.3 Spout, IdWrapWindowBolt and sink

Maybe the computation in the WindowBolt is too costly. It must perform a set containment check for every retweet object. To avoid this computation, an IdWrap-WindowBolt was written which just adds a default windowId and windowState.

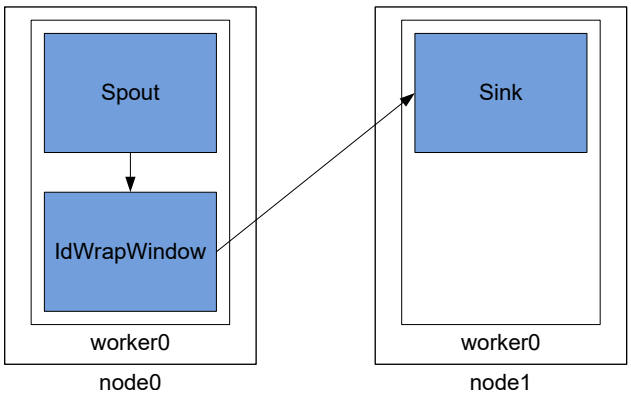

Figure 19: Two nodes: Spout, IdWrapWindowBolt and sink node

Tweet format	Average	Median	Standard deviation
Custom	83,927	94,518	27,244

A small increase could be observed, but the transfer rate is still very small. The only thing that is left is the windowState which is represented as an enum. This serialization turned out to be very costly. The following shows the usage of constant integers as described in 4.5.3.

6.1.4 Spout, WindowBolt and sink, WindowState as constant integer

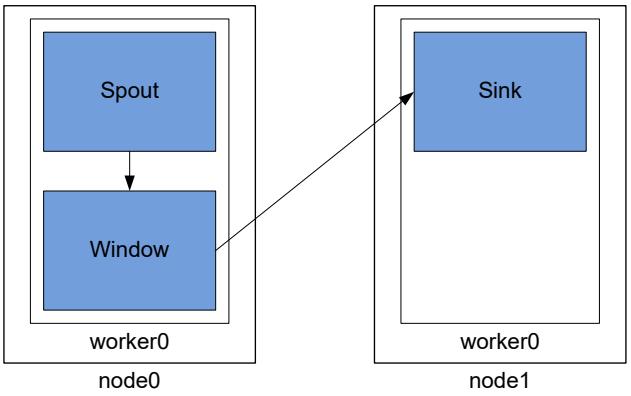

Figure 20: Two nodes: Spout, WindowBolt and sink

Tweet format	Average	Median	Standard deviation
Custom	608,065	609,756	41,442

The custom retweet format will be used together with the custom WindowState in reconstruction.

6.1.5 Dummy Reconstruction

In this experiment, the topology described in 4.2.2 was deployed with eight reconstruction nodes. However, the ReconstructionBolts were replaced by TweetCountBolts that only measure the transfer rate for retweet tuples. The usual sink bolts would also count remote requests in which we are not interested in when optimizing on retweet transfer rate. Figure 21 shows the setup for this experiment. The WindowIncrementalBolt was set to act as an IdBolt. Table 5 shows the results with RemoteRequests disabled, unbatched, batched globally with batch size 6,000 and batched per cascade with batch size 50. When using batching per cascade, multithreading can be exploited at the RemoteRoutingBolt. Nine threads were used, since the worker is already running three other components.

Per node, 30 measures were taken and from these values the total tranfer rate was calculated. Besides average, median and standard deviation were calculated among all nodes. Sending remote requests unbatched slows down the total transfer rate by a factor of almost ten, compared to sending no requests. Batching per cascade with batch size 50 is slightly slower than a global batching with batch size 6,000. However, together with multithreading, batching per cascade gives a very high throughput.

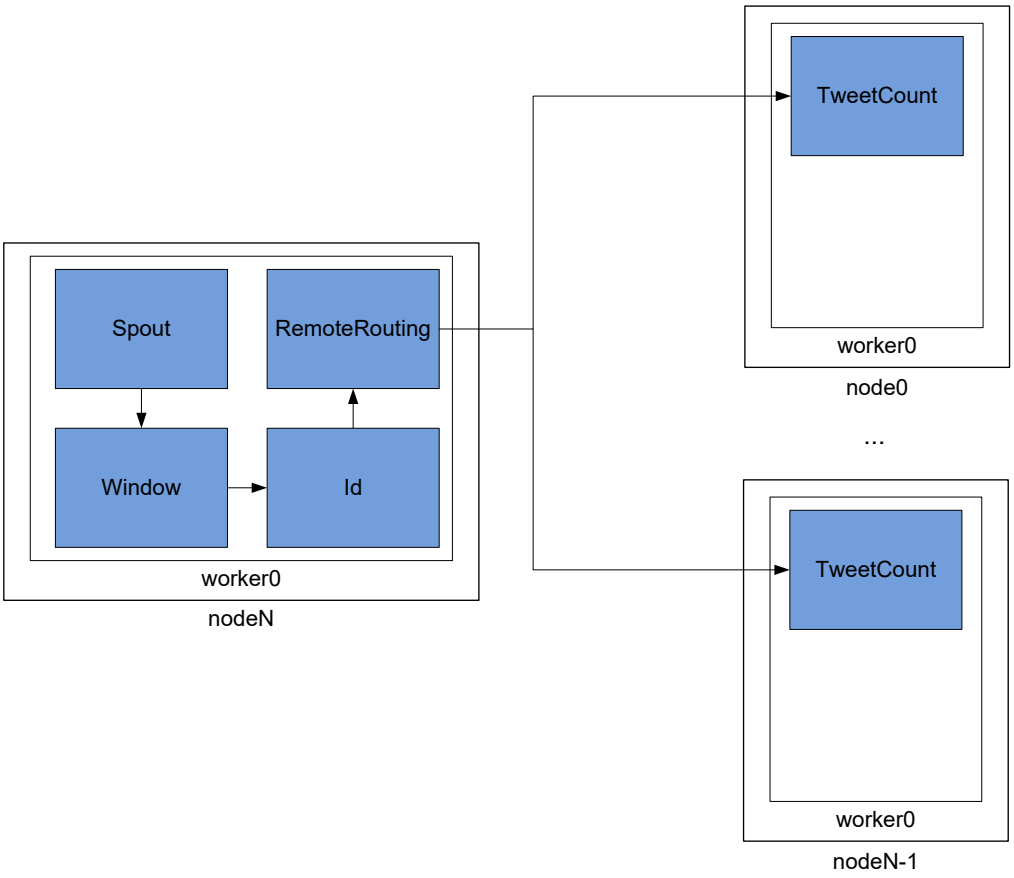

Figure 21: Setup for dummy reconstruction

Remote requests	**Total tranfer rate**	**Average nodes**	**Median nodes**	**standard deviation**
unbatched	50,697	6,337	5,460	2168
batch global 6,000	302,139	37,767	38,564	3,888
batch per cascade 50	267,383	33,422	32,544	5999
batch per cascade 50 with 9 threads	478,790	59,848	58,332	11,036
disabled	481,893	60,236	54,863	22,104

Table 5: Retweet transfer rate for different remote request settings

6.2 Reconstruction

6.2.1 User Iteration vs Prefix Iteration

For the comparison of the two reconstruction algorithms, the topology sketched in 3.3 was deployed on a single machine, so that only the performance of the reconstruction algorithms was measured. Table 6 shows the total times in seconds for the datasets. For relatively small cascades, Prefix Iteration runs a little bit faster. Still, User Iteration will be used in the following experiments. Table 7 shows the usage of ten threads for the ReconstructionBolts when using User Iteration.

Dataset	Prefix Iteration	Reconst rate	User Iteration	Reconst rate
mixed	618.4	391	5.4	44,864
1k-30k	729.9	1,258	17.8	51,617
6-100	17.7	71,672	24.6	51,569
bigger4	3,368.7	1,093	85.1	52,619

Table 6: Centralized blocking reconstruction

Dataset	User Iteration	Reconst rate	Increase
mixed	3.7	65,477	45.6 %
1k-30k	5.5	167,053	223.6 %
6-100	8.6	147,512	186.0 %
bigger4	26.1	141,123	168.2 %

Table 7: Multithreaded centralized blocking reconstruction, 10 threads

Several threads give a large increase in performance. The goal will be to achieve similar total times compared to this setup.

6.2.2 Distributed blocking reconstruction, no batching

A random distribution will be used for evaluating distributed reconstruction, since it gave the best results in [1]. The partitioning strategies will be compared in 6.2.5. Before enabling incremental reconstruction with different batching strategies, table 8 shows the time metrics for all datasets, when using blocking reconstruction with no batching. The tweet transfer rate seems to be very small.

Dataset	Total time	Reconst time	Request response time	Tweet receiving time	Tweet transfer rate	Reconst rate
mixed	8.5	2.1	5.9	6.5	37,271	28,501
1k-30k	26.2	5.9	20.1	23.4	39,264	35,068
6-100	34.3	6.6	28.4	28.2	36,985	44,986
bigger4	102.0	17.7	78.4	82.5	44,646	36,111

Table 8: Blocking reconstruction, no batching

6.2.3 Distributed incremental reconstruction

6.2.3.1 No batching of remote requests

So far, reconstruction was blocking. There is a tradeoff between getting results early and achieving a high throughput. A smaller block size is expected to reduce the throughput. Figure 22 shows the results for all datasets using a distributed incremental reconstruction for several block sizes.

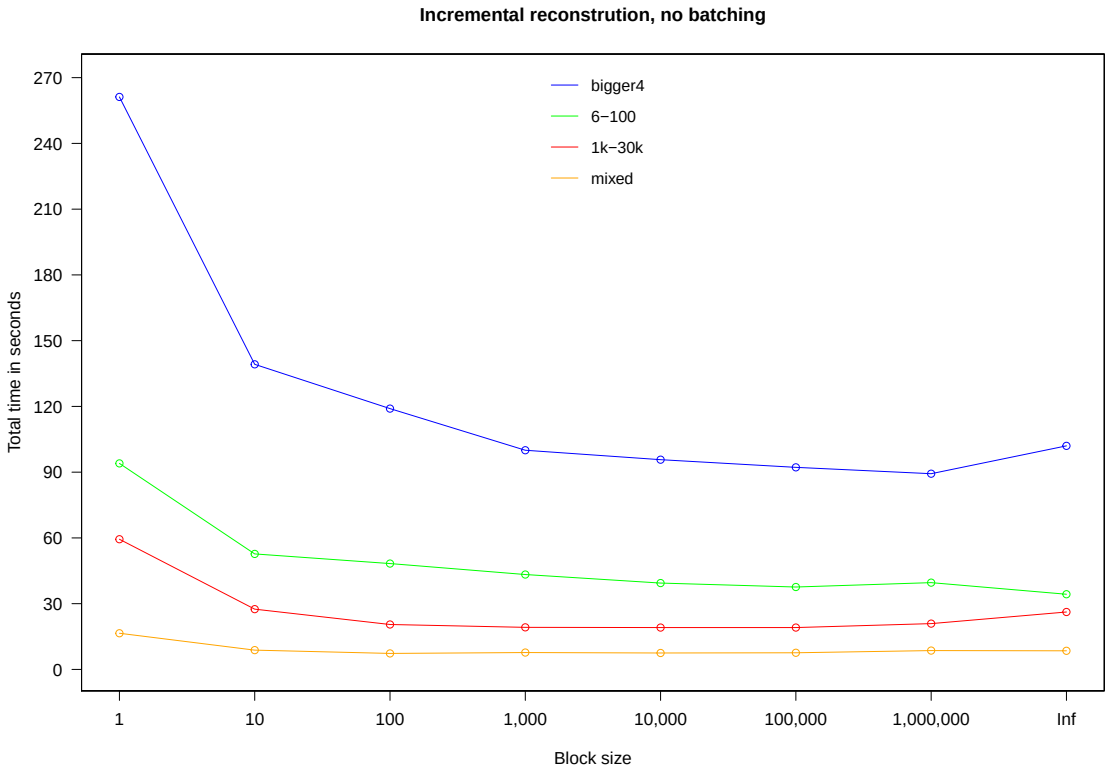

Figure 22: Distributed incremental reconstruction for several block sizes

Let us analyze these results further. The following table shows all metrics that were measured for a block size of 10,000.

Dataset	Total time	Reconst time	Request response time	Tweet receiving time	Tweet transfer rate	Reconst rate
mixed	7.5	1.8	5.2	7.4	32,738	32,302
1k-30k	19.1	4.9	20.3	19.0	48,357	48,104
6-100	39.4	6.8	28.6	29.3	43,297	32,198
bigger4	95.7	17.9	79.5	77.8	47,343	38,488

Table 9: Metrics for distributed incremental reconstruction

The tweet transfer rate is very small compared to the theoretical upper bound of 481,000 tweets per second, but the rate is close to the results of 6.1.5 when all remote

requests are emitted without batching. In this setup, the RemoteRoutingBolt emits millions of small messages: for a dataset consisting of R retweets and C cascades, it emits:

- $(nodeCount - 1) \cdot (R + C)$ remote requests (one for each retweet + one for each cascade for the root)
- $C \cdot (nodeCount - 1)$ remote flushes (one for each cascade)

For bigger4 this would be about 26.5 million messages. Additionally, every remote response is being sent as a single tuple, creating another millions of small messages. Table 10 shows statistics about the number of remote requests and reponses.

Dataset	Remote requests	Remote responses
mixed	1,700,237	1,055,906
1k-30k	6,434,015	2,263,537
6-100	9,402,232	841,566
bigger4	26,480,251	6,199,723

Table 10: Number of remote requests and responses for random distribution

6.2.3.2 Global batching of remote requests

Figure 23 shows global batching for several block sizes. Batch size and block size are the same in this setup. The highest throughput should be reached at about 10,000.

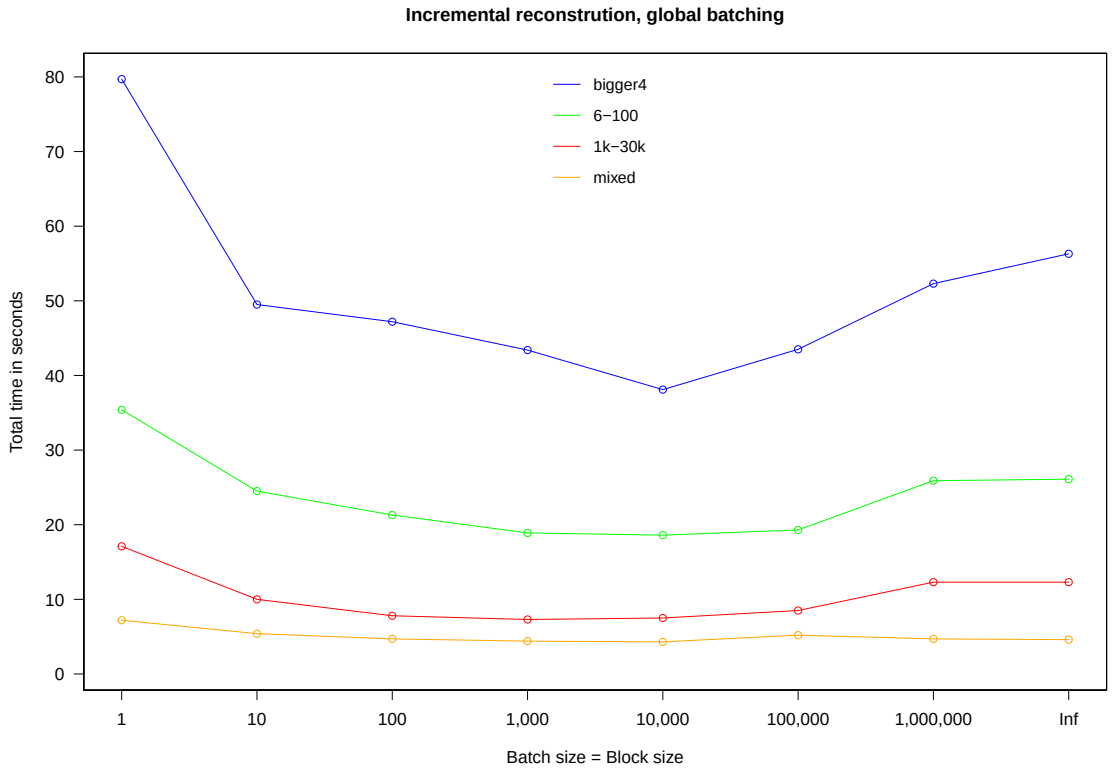

Figure 23: Distributed incremental reconstruction with global batching

6.2.3.3 Batching per cascade

Figure 24 shows batching of remote requests per cascade for several batch sizes. It performs similar to global batching with batch size around 10,000. But batching per cascade sticks to the default format of a tuple: $(windowId, windowState, payload)$. Thus, it allows an equal distribution of tuples among several threads. The effects of multithreading are shown in the next section.

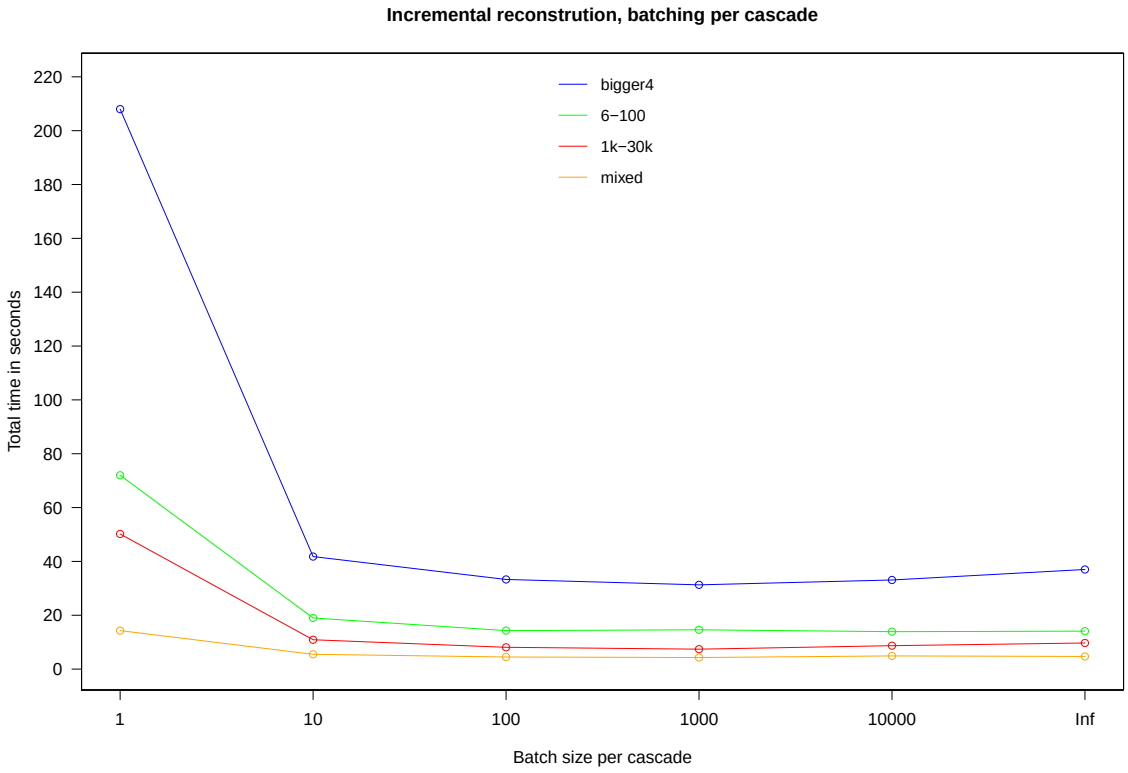

Figure 24: Distributed incremental reconstruction with batching of remote requests per cascade

For the following experiments, batching per cascade will be used with batch size 50. This seems like a reasonable tradeoff between amount of retweets and total time. Table 11 shows the reduced amount of remote requests and responses, together with the reduction in percent.

Dataset	Remote requests	Reduction	Remote responses	Reduction
mixed	36,421	97.8 %	40,171	96.2 %
1k-30k	129,920	98.0 %	803,498	64.5 %
6-100	556,808	94.0 %	651,319	22.6 %
bigger4	1,041,446	96.2 %	1,252,193	79.8 %

Table 11: Reduced number of remote requests and responses due to batching

6.2.4 Multithreading

All previous optimizations were on data structures and on the protocol. The optimizations in this section try to exploit all available ressources on the nodes of the cluster. In 6.1.5, multithreading on the RemoteRoutingBolt was shown to give an increase in the tranfer rate of retweets by 80 percent. Besides, several threads were giving a large increase in throughput when performing reconstruction on a single machine, see 6.2.1. The following shows whether this also decreases the total time when performing distributed reconstruction.

6.2.4.1 RemoteRoutingBolt

Figure 25 shows the usage of one, three, five, seven, and nine threads for the RemoteRoutingBolt. Since the CPU must execute three other threads for spout, WindowBolt and WindowIncrementalBolt, at most nine threads could be used.

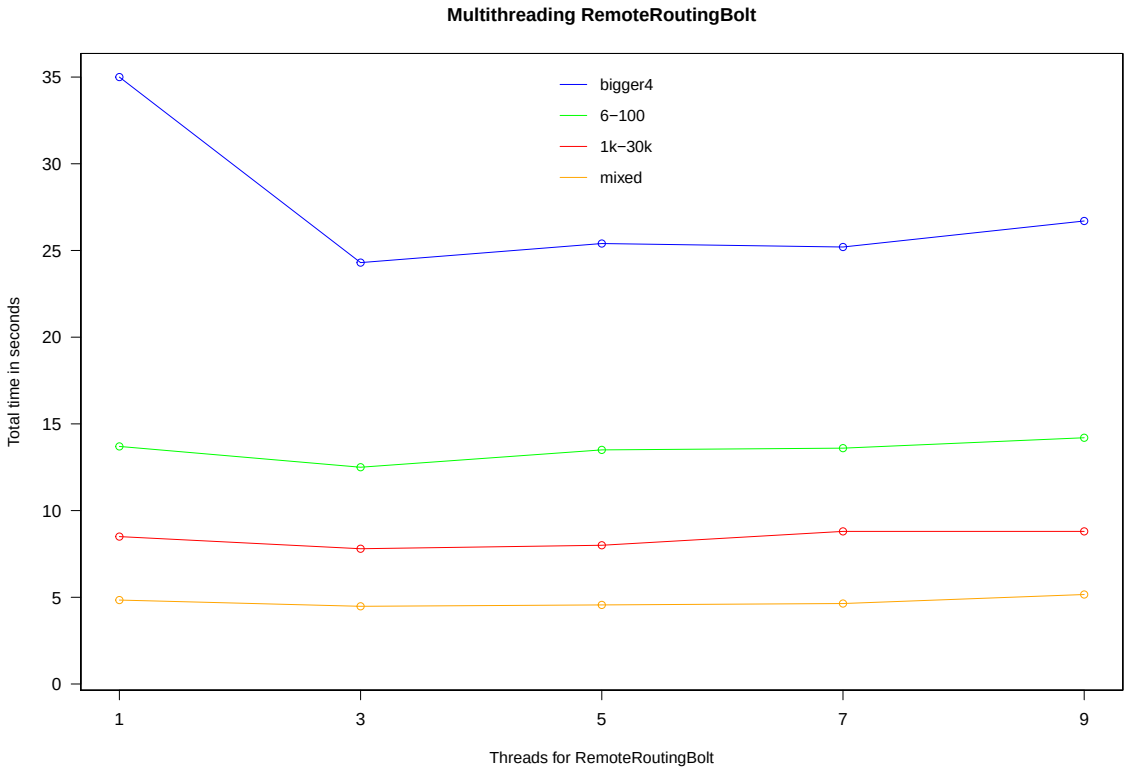

Figure 25: Multithreading RemoteRoutingBolt

Using several threads for the RemoteRoutingBolt leads to a significant decrease in total time for the biggest dataset. However, using more than three threads does not show any more improvements. Total time is even slightly increasing. Thus, three threads will be used for the RemoteRoutingBolt.

These results were not expected. Multithreading does not seem to scale very well. This behavior could not be explained. Maybe the system is upper bounded by the network link which can transfer at most 125 megabytes per second.

6.2.4.2 ReconstructionBolt

Figure 26 shows the usage of one, three, five, eight, and ten threads for the ReconstructionBolts.

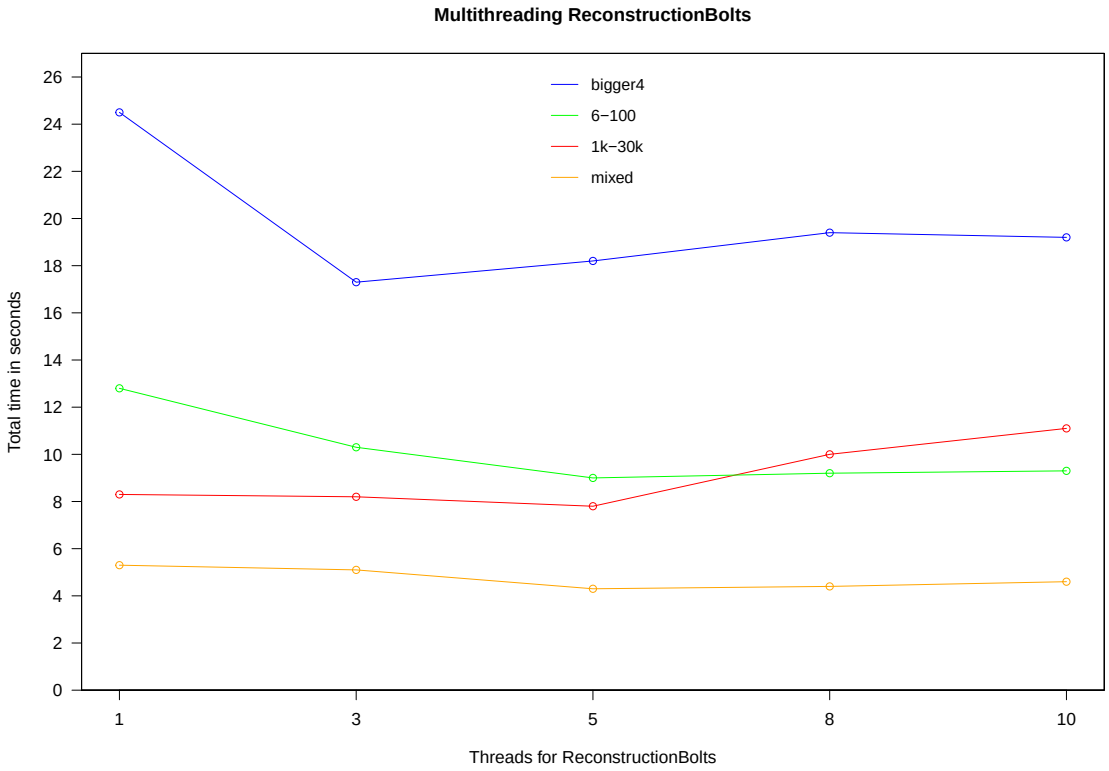

Figure 26: Multithreading ReconstructionBolts

For bigger4, using more than three threads only increases total time. 6-100 seems to show at least a slight increase. However, more than three threads do not seem to give any further improvements. For 1k-30k, the dataset consisting of large cascades only, total time significantly increases when using more than five threads. Thus, three threads will also be used for the ReconstructionBolts.

The experiments about multithreading showed that maximizing the number of threads does not lead to the highest throughput. Finding an explanation was not possible in this work. Maybe accessing the same data structure causes synchronization problems in the worker process. However, all accesses are read-only. Further work should be made to explain these results.

6.2.5 Social graph partitioning

All previous optimizations are independent from the distribution of the social graph. In figure 27, the results for all partitioning strategies are shown.

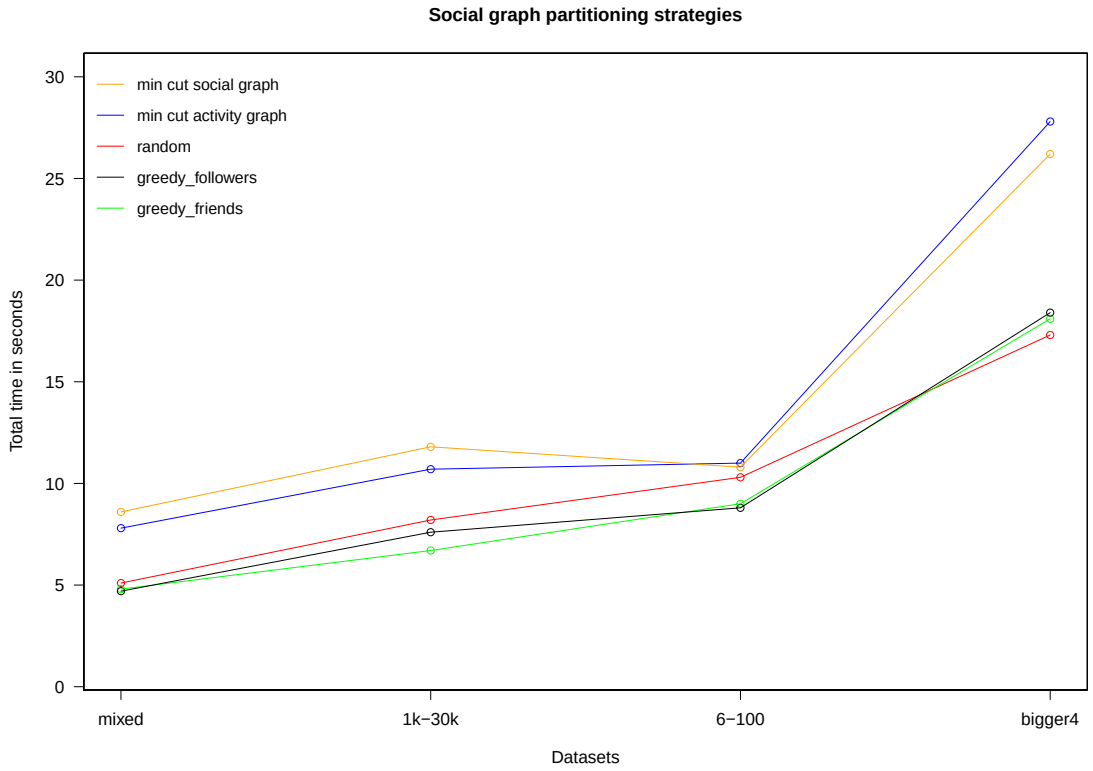

Figure 27: Partitioning strategies

The min cut strategies created with METIS perform worst. The strategies that try to achieve equal distributions of the users perform clearly better. Greedy_friends, greedy_followers and random perform about equal. However, the balanced partitionings should be preferred over random.

Table 12 shows all metrics for every dataset when using the final configuration with greedy_friends partitioning.

Dataset	**Total time**	**Reconst time**	**Request response time**	**Tweet receiving time**	**Tweet transfer rate**	**Reconst rate**
mixed	4.8	2.3	7.0	4.1	59,089	50,472
1k-30k	6.7	6.8	25.7	6.2	148,836	137,133
6-100	9.0	5.7	37.2	7.1	178,677	140,956
bigger4	18.1	22.5	92.6	14.0	263,095	203,499

Table 12: Metrics for distributed incremental reconstruction with final settings

6.2.6 Workload per reconstruction node

Let us have a look at the workload of the ReconstructionBolts. Table 13 shows the number of users together with the sum and size of edges of the friends relation and the number of cascades that were reconstructed on the node for the dataset bigger4 and greedy_friends. Table 14 shows the same information for the min cut partitioning of the social graph.

	node0	**node1**	**node2**	**node3**
users	266,952	267,835	263,860	267,415
edges	114,944,680	114,941,312	114,906,062	114,875,991
edges in MB	2,023	2,123	3,155	2,703
cascades	12,452	12,914	12,707	12,090
	node4	**node5**	**node6**	**node7**
users	265,906	263,866	262,776	265,711
edges	114,816,765	114,895,662	115,015,759	114,884,987
edges in MB	2,870	2,843	2,142	2,848
cascades	12,551	12,410	12,239	12,198

Table 13: Workload for greedy_friends

	node0	**node1**	**node2**	**node3**
users	212,892	76,386	387,009	143,136
edges	91,678,513	85,807,649	135,734,290	58,851,755
edges in MB	1,940	1,841	3,424	1,208
cascades	12,637	4,400	20,343	9,697
	node4	**node5**	**node6**	**node7**
users	468,489	209,783	275,286	351,340
edges	98,141,465	119,725,879	122,460,578	206,881,089
edges in MB	3,309	2,780	3,159	4,415
cascades	13,196	9,023	13,586	16,679

Table 14: Workload for METIS min cut social graph

The min cut partitioning on the social graph is very imbalanced. Node2 stores five times as many users compared to node1. Besides, it reconstructs almost five times as many cascades.

Greedily allocating users to the node with least friends edges leads to a very balanced distribution, with similar workloads.

6.2.7 All improvements

The following optimizations were implemented in this work:

- More efficient data structures for retweet tuples
- User Iteration reconstruction algorithm
- Incremental reconstruction with batching of remote requests
- Multithreading on RemoteRoutingBolt and ReconstructionBolts
- Balanced partitioning strategy

Figure 28 illustrates all optimizations that have been implemented in this work.

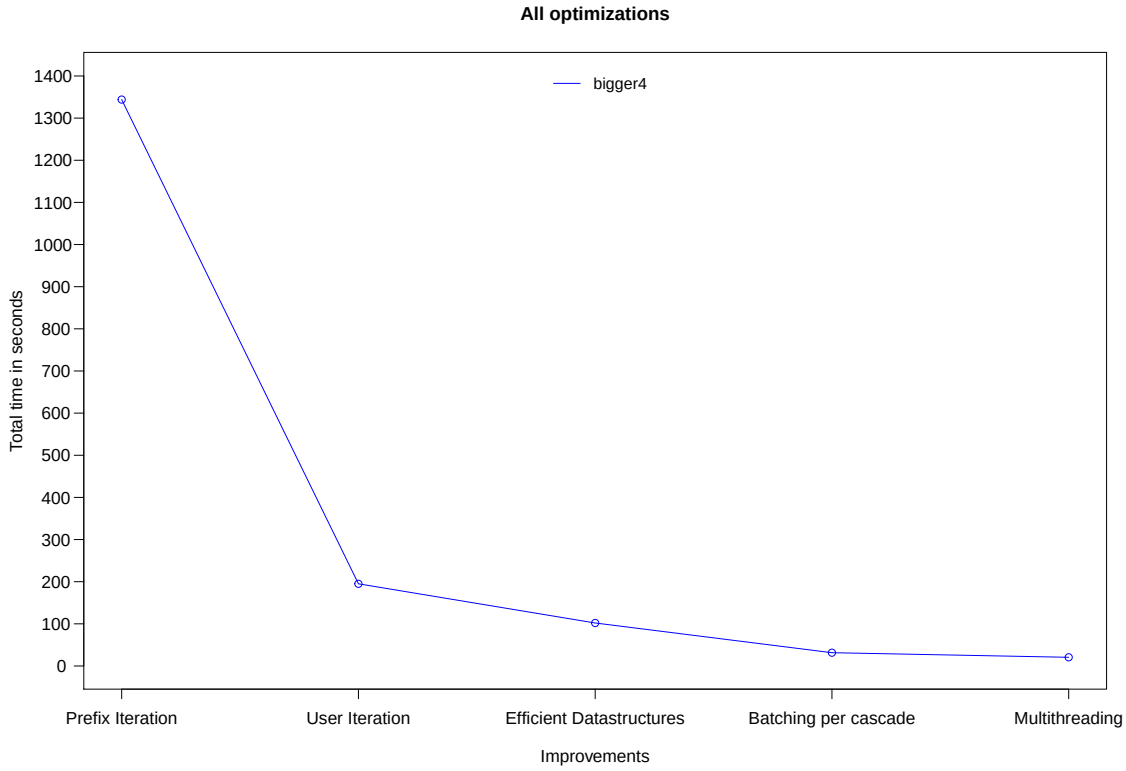

Figure 28: All optimizations of this work

6.2.8 Scaling

Two scaling experiments were made using edge_friends distribution and batching per cascade with batch size 50. In the first experiment, the total number of threads was set to eight. Four configurations with a different number of nodes were used. In the second experiment, 8 threads were used per node and the number of nodes was increased.

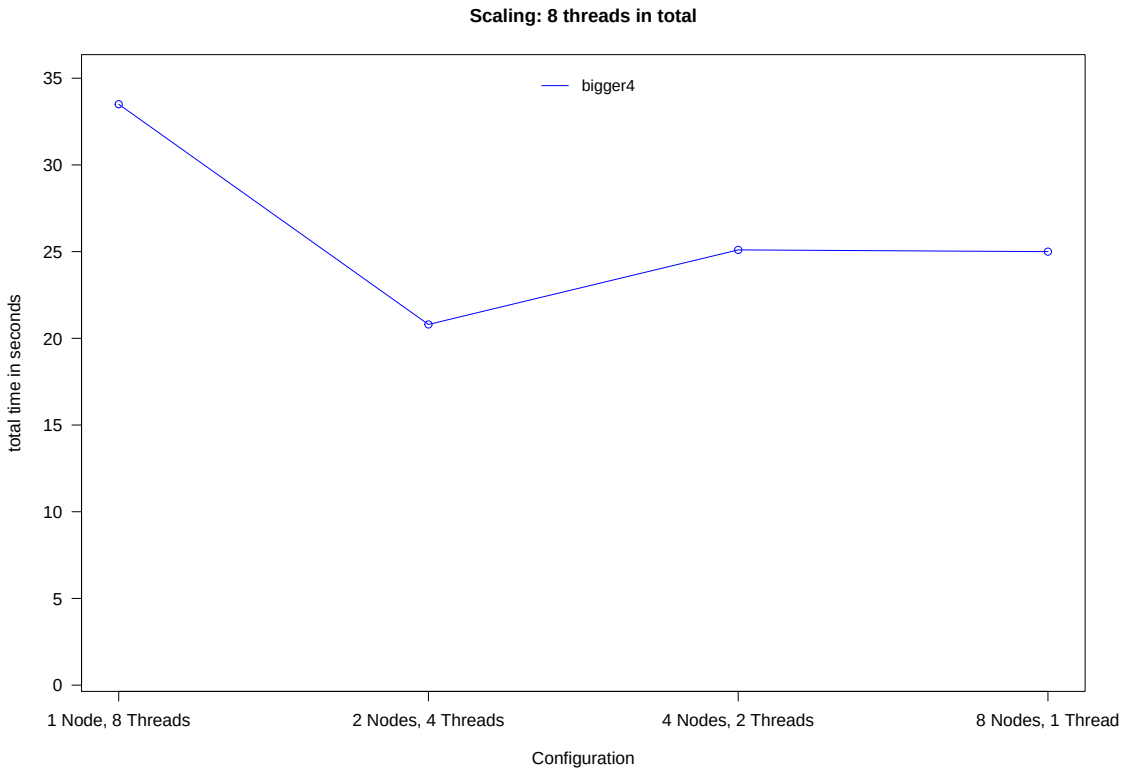

Figure 29: Scaling with fixed number of threads among several nodes

The result shows that two nodes with four threads each give the highest throughput. The expectation was that more nodes would increase the throughput. This behavior could not be explained.

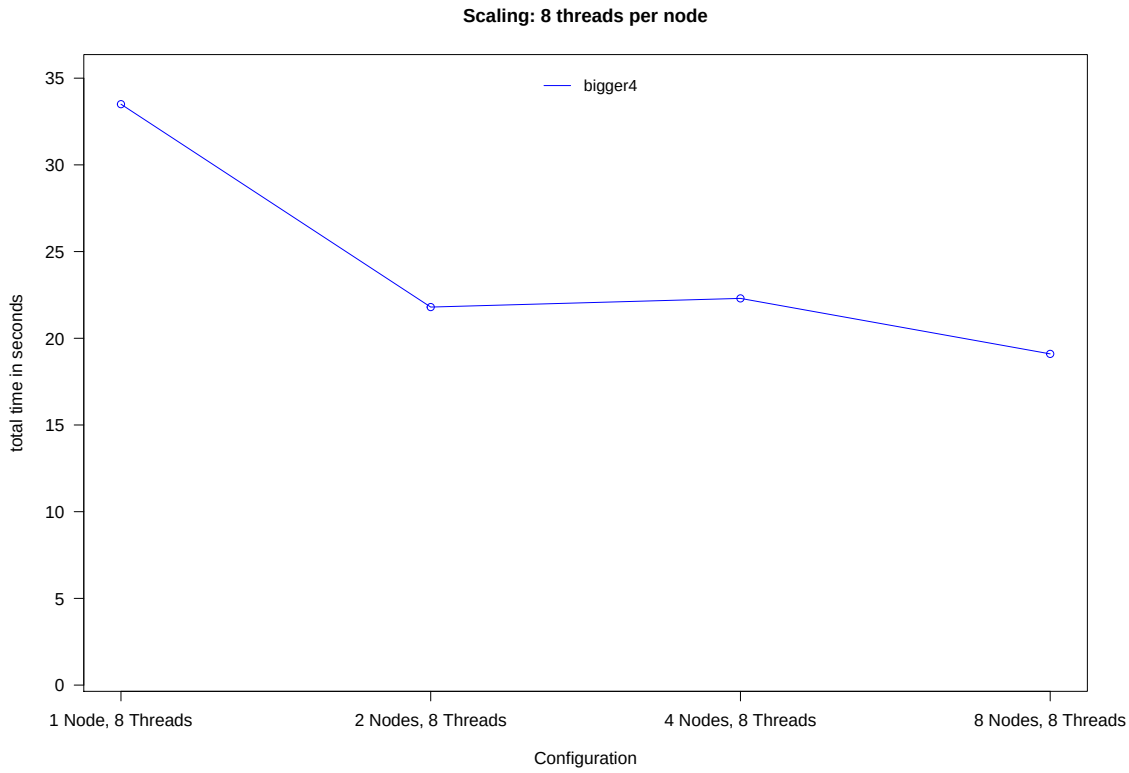

Figure 30: Scaling number of nodes

At least in this scenario, scaling works. But the increase from going from two nodes to eight nodes does not justify six additional nodes.

7 Discussion

The optimizations could reduce the total time for the biggest dataset from 22 minutes to less than 20 seconds. The system was optimized in several stages: first, more efficient data structures were implemented. Second, a faster reconstruction algorithm was implemented. Third, the protocol for exchanging social graph information was optimized towards batch processing. Instead of sending a lot of small messages, fewer but bigger messages are sent. Fourth, available resources were exploited by multithreading. However, more than three threads did not show any improvement.

User Iteration was shown to outperform Prefix Iteration for most datasets. However, there is still an open question, whether to implement an adaptive reconstruction algorithm and pay another hundreds of gigabytes of main memory for storing follower sets.

If the system should now be used in live mode, the following settings would be made:

- Choose User Iteration algorithm for reconstruction
- Turn on incremental reconstruction with batching per cascade
- If possible, use three threads for RemoteRoutingBolt and ReconstructionBolts
- Distribute users equally among all nodes while keeping the partitions balanced with respect to the number of edges with respect to the number of friends

When using the final configuration, the system can reconstruct about 180,000 retweets per second, while using eight nodes for reconstruction. This is more than the Twitter peak of 143,000 tweets per second[11]. On average, around 6,000 tweets are sent per second[12] which means that there is no problem in handling the amount of incoming retweets with eight nodes.

The optimizations of this work introduced a lot of state on the RemoteRoutingBolt and on the ReconstructionBolts. However, the state is relatively small. The increase in throughput should justify the effort for a more complex recovery. Currently, the system cannot handle node failures.

Scaling was not fully understood. Maybe the system is upper bounded by the network link or the hard disk. Further work is needed to understand these effects.

[11] https://blog.twitter.com/2013/new-tweets-per-second-record-and-how, Twitter Blog, last lookup on 06.01.2017

[12] http://www.internetlivestats.com/twitter-statistics/, Internet Live Statistics Twitter, last lookup on 14.01.2017

8 Future work

In future work, the focus should be on making the system ready to run over a longer period and being able to perform recovery.

- **Fault tolerance**
  - **Replaying failed tuples:** This is the key to achieve at-least once processing. So far, replay is not implemented. A tuple must be given an identifier, so that the spout can replay it. In this case, the id of the retweet could be used.
  - **Recovery:** The social graph and the userToNodeMap will be recovered automatically during initialization of the new worker process. However, there is more state that had to be recovered. For this purpose, Storm offers state checkpointing[13].
  - **Evaluation of Trident:** Trident guarantees exactly-once processing by adding another layer of abstraction. It might be worth to evaluate the implementation. Even if it causes a slowdown of factor ten, the system would still be at triple times the average number of tweets per second.
- **Understand scaling**: the scaling results could not be explained. Further work on other hardware might be needed to understand these effects.
- **Adaptive reconstruction algorithms:** There is still a trade-off for small cascades: use Prefix Iteration to increase throughput even further and pay another hundreds of gigabytes for storing the follower relation.
- **Updates on the social graph:** The current system works on a fixed snapshot of the social graph. However, in reality, the social graph is changing permanently. The system could regularly retrieve social graph data from the Twitter API. However, this requires locking on the shared social graph instance.
- **Online evaluation:** So far, no live evaluation has been made of the system. All retweets were coming from a JSON file that was stored on disk. All cascades were considered finished, when the end of the JSON file was reached. In reality, determining the end of a cascade is not trivial. Simple strategies like waiting for a fixed amount of time are possible. The chair of Web Science has been developing more sophisticated strategies [9] that could be combined with the system.

[13] http://storm.apache.org/releases/1.0.1/State-checkpointing.html, Apache Storm Documentation, last lookup on 11.01.2017

Bibliography

[1] B. Lutz, "Evaluation of social graph partitioning strategies and improvement of distributed reconstruction of retweet cascades", master's project, University of Freiburg, 2016.

[2] Beevolve, "An exhaustive study of twitter users across the world." `http://www.beevolve.com/twitter-statistics/#b2`, 2012.

[3] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi, "Naiad: A timely dataflow system", in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, (New York, NY, USA), pp. 439–455, ACM, 2013.

[4] S. T. Allen, M. Jankowski, and P. Pathirana, *Storm Applied: Strategies for Real-time Event Processing*. Greenwich, CT, USA: Manning Publications Co., 1st ed., 2015.

[5] L. Sättler and S. Ebner, "Social Media Analysis", team project, University of Freiburg, 2013.

[6] R. Ramos, "Distributed Graph Storage from Cascade Reconstruction", master's project, University of Freiburg, 2013.

[7] M. Huber, "Verteilte Rekonstruktion von Informationskaskaden", master's thesis, University of Freiburg, 2014.

[8] S. Ganz, "Partitionierung von sozialen Graphen zur Rekonstruktion von Informationskaskaden", bachelor's thesis, University of Freiburg, 2015.

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

Appendix

Creation of random partitioning

```
#!/bin/bash

if [ ! $# -eq 3 ] ; then
  echo "Usage: $0 <UserListFile> <outputdir> <number of nodes>"
  echo "Creates a random distribution for the UserListFile for a
  ↪  number of given nodes"
  exit 1
fi

FILE=$1
OUTDIR=$2
NODES=$3

mkdir -p $OUTDIR

while read -r line; do
        node=$(($RANDOM % $NODES))
        echo $line >>$OUTDIR/node$node.csv
done < $FILE
```

Retweet object from Twitter API

The following shows the original tweet object, with respect to the projected object shown in 4.5.1.


```
{
  "retweeted_status":{
    "contributors":null,
    "text":"Last night was unbelievable. I cannot believe we had the
       honour to perform at the Olympics! Has to be one of the best
       nights off my life!",
    "geo":null,
    "retweeted":false,
    "in_reply_to_screen_name":null,
    "truncated":false,
    "entities":{
      "hashtags":[
      ],
      "urls":[
      ],
      "user_mentions":[
      ]
    },
    "in_reply_to_status_id_str":null,
    "id":234954143758946304,
    "source":"<a href=\"http://twitter.com/download/iphone\" rel=\"
       nofollow\">Twitter for iPhone</a>",
    "in_reply_to_user_id_str":null,
    "favorited":false,
    "in_reply_to_status_id":null,
    "created_at":1344852407000,
    "retweet_count":1,
    "in_reply_to_user_id":null,
    "id_str":"234954143758946304",
    "place":null,
    "user":{
      "location":"Doncaster",
      "default_profile":true,
      "statuses_count":4239,
      "profile_background_tile":false,
      "lang":"en",
      "profile_link_color":"0084B4",
      "id":84279963,
      "following":null,
      "favourites_count":7,
      "protected":false,
      "profile_text_color":"333333",
      "verified":true,
```

```
41        "contributors_enabled":false,
42        "description":"1/5 of One Direction :) Live life for the moment
             because everything else is uncertain! We would be nowhere
             without our incredible fans, we owe it all to you.",
43        "name":"Louis Tomlinson",
44        "profile_sidebar_border_color":"C0DEED",
45        "profile_background_color":"C0DEED",
46        "created_at":1256201891000,
47        "default_profile_image":false,
48        "followers_count":5184210,
49        "profile_image_url_https":"https://si0.twimg.com/profile_images
             /1846889753/image_normal.jpg",
50        "geo_enabled":false,
51        "profile_background_image_url":"http://a0.twimg.com/images/themes
             /theme1/bg.png",
52        "profile_background_image_url_https":"https://si0.twimg.com/
             images/themes/theme1/bg.png",
53        "follow_request_sent":null,
54        "url":null,
55        "utc_offset":-18000,
56        "time_zone":"Eastern Time (US & Canada)",
57        "notifications":null,
58        "profile_use_background_image":true,
59        "friends_count":2727,
60        "profile_sidebar_fill_color":"DDEEF6",
61        "screen_name":"Louis_Tomlinson",
62        "id_str":"84279963",
63        "profile_image_url":"http://a0.twimg.com/profile_images
             /1846889753/image_normal.jpg",
64        "show_all_inline_media":false,
65        "listed_count":31569,
66        "is_translator":false
67     },
68     "coordinates":null
69  },
70  "contributors":null,
71  "text":"RT @Louis_Tomlinson: Last night was unbelievable. I cannot
        believe we had the honour to perform at the Olympics! Has to be one
         of the be ...",
72  "geo":null,
73  "retweeted":false,
74  "in_reply_to_screen_name":null,
75  "truncated":true,
76  "entities":{
77     "hashtags":[
78     ],
79     "urls":[
```

```
    ],
    "user_mentions":[
       {
          "id":84279963,
          "indices":[
             3,
             19
          ],
          "screen_name":"Louis_Tomlinson",
          "id_str":"84279963",
          "name":"Louis Tomlinson"
       }
    ]
 },
 "in_reply_to_status_id_str":null,
 "id":234954178848505856,
 "in_reply_to_user_id_str":null,
 "source":"<a href=\"http://twitter.com/download/iphone\" rel=\"nofollow
    \">Twitter for iPhone</a>",
 "favorited":false,
 "in_reply_to_status_id":null,
 "created_at":1344852416000,
 "retweet_count":1,
 "in_reply_to_user_id":null,
 "id_str":"234954178848505856",
 "place":null,
 "user":{
    "location":"Hogwarts",
    "default_profile":false,
    "statuses_count":15557,
    "profile_background_tile":true,
    "lang":"en",
    "profile_link_color":"FF0000",
    "id":299033003,
    "following":null,
    "favourites_count":972,
    "protected":false,
    "profile_text_color":"3D1957",
    "verified":false,
    "contributors_enabled":false,
    "description":"A Rusher, Directioner, Gleek, Swiftie, Sheeranator,
       and also a sixth-year at Hogwarts. From District 4, Panem and
       resides in The Shire.",
    "name":"Tom Daley. ",
    "profile_sidebar_border_color":"65B0DA",
    "profile_background_color":"642D8B",
    "created_at":1305459562000,
```

```
124       "default_profile_image":false,
125       "followers_count":351,
126       "profile_image_url_https":"https://si0.twimg.com/profile_images
             /2494789860/image_normal.jpg",
127       "geo_enabled":true,
128       "profile_background_image_url":"http://a0.twimg.com/images/themes/
             theme10/bg.gif",
129       "profile_background_image_url_https":"https://si0.twimg.com/images/
             themes/theme10/bg.gif",
130       "follow_request_sent":null,
131       "url":null,
132       "utc_offset":28800,
133       "time_zone":"Kuala Lumpur",
134       "notifications":null,
135       "profile_use_background_image":true,
136       "friends_count":792,
137       "profile_sidebar_fill_color":"7AC3EE",
138       "screen_name":"_BigTime1D_",
139       "id_str":"299033003",
140       "profile_image_url":"http://a0.twimg.com/profile_images/2494789860/
             image_normal.jpg",
141       "show_all_inline_media":false,
142       "listed_count":0,
143       "is_translator":false
144    },
145    "coordinates":null
146 }
```