University of Freiburg

**Faculty of Engineering**

**Research Group on Web Science**

# Combining Graph-Iterative and Streaming Computations for Influence Graph Derivations

Master's Thesis

**Author:** Bastian Meyer

**Advisor:** Prof. Dr. Peter Fischer
Io Taxidou

**Examiners:** Prof. Dr. Peter Fischer
Prof. Dr. Gerhard Schneider

**Submission:** July 31, 2017

# Acknowledgements

Special thanks go out to all the people who made this thesis possible:

Frank McSherry, for helping me understand Differential Dataflow.

Professor Fischer, for pushing me towards understanding everything.

Sabine, for proof-reading and finding all these nasty typos.

My family, for always supporting me.

Felicitas, for everything.

## Abstract

Social networks have become a natural part of our daily lives, and the content we see there affects what we think. It is therefore important to understand how users are influenced to share some content.

These influences are not provided by the social networks, but have to be reconstructed from the available data. Since this data is too large for a single computer's main memory, the reconstruction system has to be distributed among multiple computers.

In previous work, such a system has been implemented for *Twitter*. It reconstructs the influences within a *Retweet cascade*, that is, a chain of Retweets of some Tweet. This system can compute the influences of a data set of 3.6 million Retweets in under twenty seconds.

For a full evaluation of this system, this thesis implements another approach, called `CRGP`, that uses a computational model called Differential Dataflow. This model is a data-parallel framework with the unique feature of combining incremental input updates with iterative stream processing. Two reconstruction algorithms are introduced as part of this new system: `LEAF` and `GALE`.

For both algorithms, the social graph is distributed among multiple nodes called *workers*. `LEAF` sends each Retweet to the worker storing the friendships of the Retweet's author where possible influence edges are created for all of the author's friends. These are then filtered at the workers storing the respective influencers. `GALE` broadcasts the Retweets to all workers, and only the worker storing the author's friendships creates the influence edge.

The evaluation of `CRGP` shows that `GALE`, when run on eight machines, can reconstruct the data set with 3.6 million Retweets in just over six seconds.

## Zusammenfassung

Soziale Netzwerke sind ein selbstverständlicher Anteil unseres Alltags geworden, und die dort geteilten Inhalte beeinflussen, was wir denken. Deshalb ist es wichtig zu verstehen, wie Nutzer beeinflusst werden, dort bestimmte Inhalte zu verbreiten.

Diese Einflüsse werden von den sozialen Netzwerken nicht bereitgestellt, sondern müssen aus den verfügbaren Daten rekonstruiert werden. Da diese Daten für den Hauptspeicher eines einzelnen Rechners zu groß sind, muss das Rekonstruktionssystem auf mehrere Rechner verteilt werden.

In früheren Arbeiten wurde ein solches System für *Twitter* implementiert, das die Einflüsse innerhalb von *Retweet-Kaskaden*, Ketten von Retweets eines bestimmten Tweets, rekonstruiert. Dieses System kann den Einflussgraphen für einen Datensatz aus 3,6 Millionen Retweets in unter zwanzig Sekunden berechnen.

Für eine vollständige Evaluation des Systems wird im Rahmen dieser Arbeit ein weiterer Ansatz namens `CRGP` implementiert, der ein Rechenmodell nutzt, welches inkrementelle Input-Updates mit iterativer Stream-Bearbeitung vereint. Darauf aufbauend werden zwei Algorithmen zur Rekonstruktion vorgestellt: `LEAF` und `GALE`.

Durch den Einsatz von `GALE` auf acht Rechnern kann `CRGP` den Datensatz mit 3,6 Millionen Retweets in knapp sechs Sekunden verarbeiten.

# Contents

# 1. Introduction

Social networks have become omnipresent in our daily lives. As of June, 2017, *Facebook*[1] alone has 2.01 billion monthly active users [7], more than a quarter of the entire world population [6]. With 328 million monthly active users [30], *Twitter*[2] is smaller, but still one of the most influential networks. Other important social networks include *YouTube*[3] with 1 billion total users [31], *Instagram*[4] with 700 million users [12], and *Reddit*[5] with 250 million users [23].

These online portals typically allow users to connect to other people all around the world, subscribe to news sites, celebrities, corporations, or any other group they want to stay informed about, and most importantly, post and reshare anything they deem interesting with other users, be it their own opinion on a specific matter, everyday experiences, or important news.

While this idea promises many advantages, such as staying in touch with friends from the other side of the world, or promoting free speech, it also leads to a serious problem. Users tend to prefer news sites and other users who share their own opinions [8, 9], and the algorithms deciding where to display which content further influence this behavior by placing contents the user is more likely to care about at more prominent places on their websites [4], resulting in a sociological phenomenon called the *filter bubble* [22]: users mostly perceive their own viewpoints, but are isolated from contradicting ones, encouraging them to believe their own opinion is shared more widely than it actually is, and thus enforcing it even further. In these confined groups, it is easier to spread false information and fake news supporting that group's standpoint, as rectifying contents are unlikely to be introduced into this group due to their contrasting the group's opinions. The filter bubble can therefore threaten democracy [22] which depends on a multitude of opinions, commonly accepted facts, and a civic discourse on them.

A key point in counteracting filter bubbles is understanding how they form and function. If their behavior is known it will be easier to implement measures breaking them up or preventing them in the first place. This, again, requires to study how people interact with each other in social networks: are there certain patterns in the connections between users? What and who influences a user to share some content?

## 1.1. Retweet Cascades

The research group on Web Science at the University of Freiburg is investigating how such influences can be obtained from the data provided by social networks. Their social network of interest is Twitter, which is one of the most influential social networks and has often been studied before; examples include detecting real-time events such as earthquakes [24], predicting election outcomes [28], and tracking the spread of diseases [26].

[1] `https://www.facebook.com/`.
[2] `https://twitter.com`.
[3] `https://www.youtube.com/`.
[4] `https://www.instagram.com/`.
[5] `https://www.reddit.com/`.

All these studies yielded positive results and found Twitter to be a good indicator for important issues.

On Twitter, users can post short public messages called *Tweets*. If user Alice wants to see all of user Bob's Tweets, she can *follow* him; Bob's Tweets will now appear at the time of their writing in Alice's *timeline* along with all other Tweets of users she is following. Bob cannot deny Alice to follow him[6], but he also does not have to follow her back. This concept of following is visualized in figure 1, showing a small graph of four users and their following each other. In this scenario, Alice will see all the Tweets posted by Bob and Carol, but not the ones posted by Dave whom she is not following.

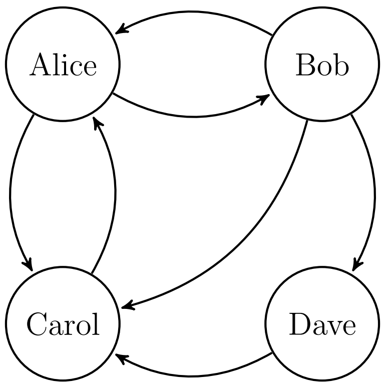


Figure 1: A sample social graph with the four users Alice, Bob, Carol, and Dave. Alice follows Bob and Carol, Bob follows all other users, Carol follows Alice, and Dave follows Carol.

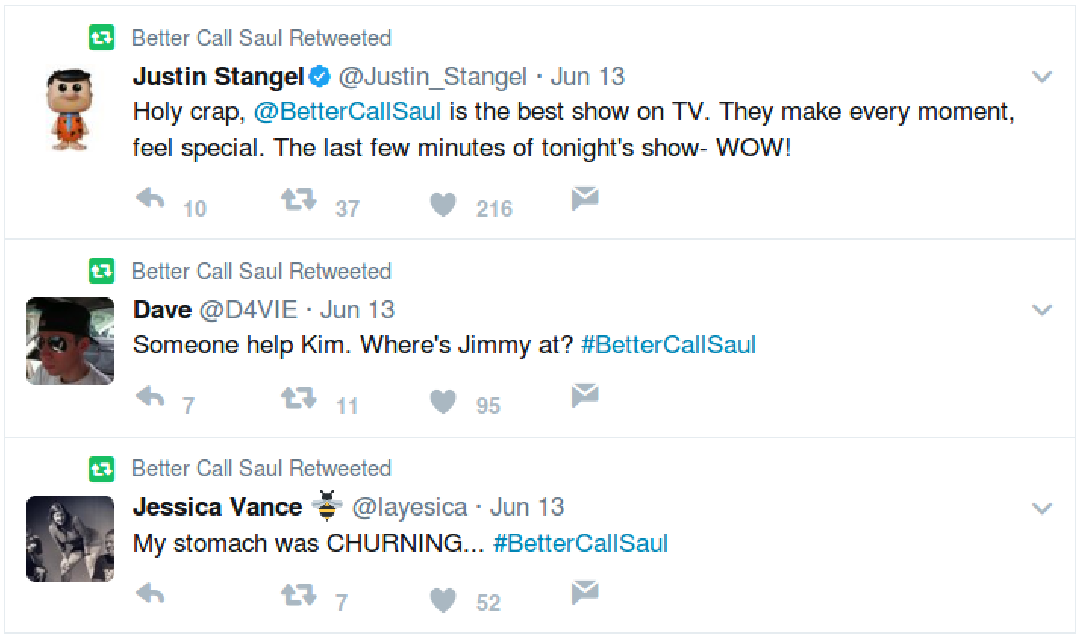
Twitter's central resharing mechanism is called *retweeting*: if Alice wants to share Bob's Tweet with her own followers, she can *retweet* Bob's Tweet. This will make it appear in the timelines of Alice's followers at the time of Alice's retweeting, with a remark that it was retweeted by Alice. Figure 2 shows an example of three such *Retweets*. In the case of the social graph in figure 1, Carol will therefore see Alice's Retweet of Bob's Tweet, although she is not following Bob.

Retweets can also be chained: if Bob retweeted one of Dave's Tweets, and Alice then retweeted Bob's Retweet, Carol would see Alice's Retweet in her timeline, but it would appear to her as if Alice had directly retweeted Dave's Tweet. Such a chain of Retweets is called a *Retweet cascade*, for which an example is shown in figure 3a.

These Retweet cascades can be a good measure for the aforementioned influences in social graphs: more influential users are more likely to be retweeted by other users, be it their own Tweet or a Retweet of another user. The problem is, though, that Twitter does not store the intermediate Retweets in a cascade; all Retweets only reference the original Tweet: "(...) Retweets of Retweets do not show representations of the intermediary Retweet, but only the original Tweet." [29]

Figure 3a shows a sample cascade of Retweets how it could have actually occurred. An outsider would only be able see that Bob, Carol, and Dave retweeted Alice, while in

[6] At least by default; he could entirely block Alice, or make his complete user account private.

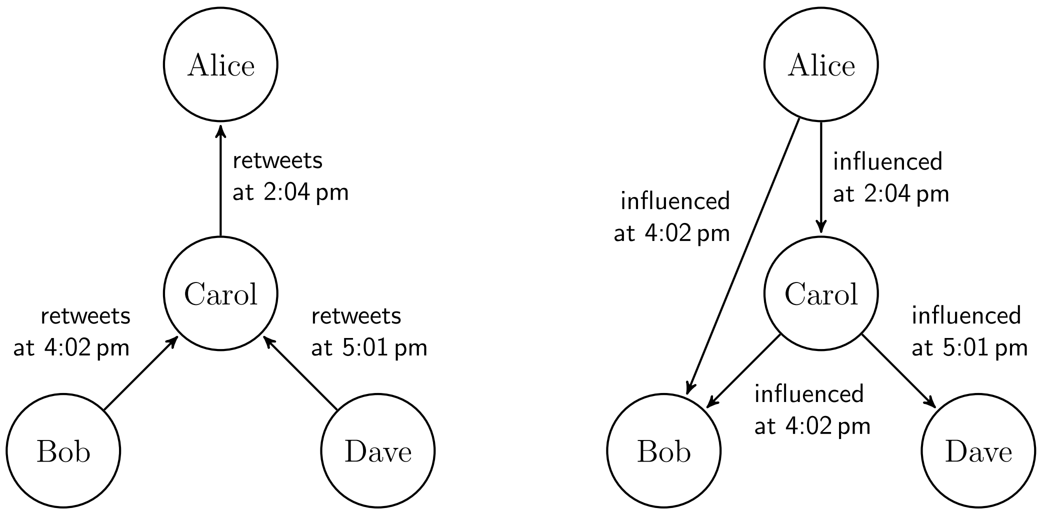


Better Call Saul Retweeted
Justin Stangel @Justin_Stangel · Jun 13
Holy crap, @BetterCallSaul is the best show on TV. They make every moment, feel special. The last few minutes of tonight's show- WOW!
10 37 216

Better Call Saul Retweeted
Dave @D4VIE · Jun 13
Someone help Kim. Where's Jimmy at? #BetterCallSaul
7 11 95

Better Call Saul Retweeted
Jessica Vance @layesica · Jun 13
My stomach was CHURNING... #BetterCallSaul
7 52

Figure 2: Three Retweets as they appear in the timelines of users following the retweeting user. In front of each original Tweet, a small remark highlights that the Tweets have been retweeted.



(a) A small Retweet cascade: Alice posted the original Tweet which was then retweeted by Carol. Bob and Dave retweeted Carol's Retweet.

(b) The reconstructed influences. If Dave had retweeted Carol's Retweet before Bob (for example, at 3:03 pm), Dave could have influenced Bob as well.

Figure 3: A small Retweet cascade on the social graph of figure 1, and its reconstructed influences.

fact, Bob and Dave retweeted Carol's Retweet of Alice's Tweet. Assuming, users only retweet Tweets of users they are following, the outsider could deduce the influences of each user as shown in figure 3b if they had knowledge of the social graph in figure 1, and of the times when each Retweet in the cascade was posted.

The research group on Web Science have developed algorithms and systems for reconstructing such Retweet cascades, that is, deducing the intermediate Retweets from the data provided by Twitter. The idea of these systems is sketched in figure 4: a stream of Retweets is processed by the reconstruction system which computes the influences. These influences are edges between users of the *influence graph*, stating whom a user influenced to share a Retweet.

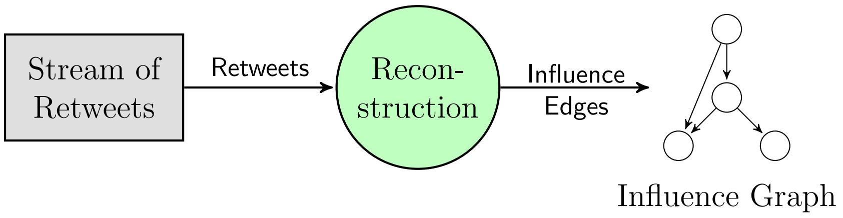


Figure 4: A conceptual sketch of a Retweet cascade reconstruction system, from [15]. A stream of Retweets is passed into the reconstruction system which computes the edges of the influence graph.

The algorithms performing the actual cascade reconstruction are rather simple: for each Retweet they check for (direct) relationships between the retweeting user and the cascades's *prefix*, that is, all users who have retweeted the original Tweet before. If this is the case, these respective users become the retweeting user's *influencers*.

The complexity of the problem arises from the amount of data that needs to be processed, and the time in which it is expected to be processed. For real-world reconstructions of arbitrary Retweet cascades, Twitter's entire social graph must be available to the computation. With 328 million monthly active users [30] and on average 208 followers per user [3], Lutz [15] estimated a total size of a little more than 1 $TB$ to store the social graph.

Typically, about 500 million Tweets are posted per day, that is, 5,700 Tweets per second; the highest observed rate was 143,199 Tweets per second [13]. The cascade reconstruction must be able to hold up to this rate to produce real-time results. Thus, it is insufficient to store the social graph on slow secondary storage devices such as hard disk (HDD) or solid-state drives (SSD); instead it must be kept in main memory for the entire reconstruction. However, the social graph's size of 1 $TB$ is larger than an average computer's main memory. Therefore, the entire computation must be distributed among several machines.

The research group's system builds on top of *Apache Storm*, a framework for the distributed processing of streams. Their most recent results have been very promising—Lutz [15] was able to reconstruct a data set of more than 3.6 million Retweets in less than twenty seconds, which is a reconstruction rate of 180,000 Tweets per second—but for a more complete evaluation of this approach's performance, it was the research group's intention to re-implement a similar approach using another framework than Apache Storm.

The choice fell on a computational model called *Differential Dataflow* [19], a stream-processing framework supporting both incremental input updates and iterative data handling. Differential Dataflow was introduced and implemented as a `C#` framework called *Naiad* [21] by a *Microsoft Research* group. Although the research group has been closed, and therefore the development of Naiad has been discontinued, Differential Dataflow has been re-implemented in `Rust` by McSherry [16], co-author of the original Differential Dataflow papers, and, in this new version, is further maintained. The idea behind this computational model is that work is only done on differences (in comparison to previous computations) in the input data; parts that have not changed will not be re-evaluated. This approach has the potential to save on computation time, especially for small changes in large data sets. It will, however, lead to an increased memory usage, since previous data has to be stored for changes in the data to be detectable.

`Rust`[7] is a new systems programming language developed by *Mozilla Research*, aiming for memory safety without penalizing performance. Its strong type system statically guarantees that compiled programs do not produce segmentation faults, buffer overflows, or even data races in concurrent programms, yet it still achieves performance similar to `C++` [27], making it a good candidate for such distributed frameworks and computations:

> "When we first started working on `Rust` as a research project in 2009, we (...) were particularly interested in two things: how to build more ambitious parallel architectures, and how to implement high-performance software without many of the pitfalls and vulnerabilities of `C++`."
>
> — *Dave Hermann*, Director of Strategy at Mozilla Research [10]

This thesis will introduce and evaluate a system for reconstructing Retweet cascades using the Differential Dataflow framework written in `Rust`, and compare it to the existing solution using Apache Storm.

## 1.2. Definitions

In the previous section, a few terms have been informally introduced. For a more profound understanding, this section lists these and other terms used in this thesis with their definitions.

#### User

A person or an entity (e. g. a political party, a company, or a band) registered on a social network. For this thesis, a user simply consists of a unique identifier.

[7] `https://www.rust-lang.org`.

**Social Network**

An online portal typically allowing its users to connect to other people, share content of some form with other users, and subscribe to the content updates by other users or groups.

**Social Graph**

A graph $G = (U, E)$ modelling a social network, where the nodes $U$ are the social network's users, and the edges $E$ depict the relations between the users. Unless otherwise noted, this thesis assumes a graph with directed edges $(u_1, u_2) \in E, u_1, u_2 \in U$. The social graph could be regarded as a temporal graph with changes over time, but for simplicity, this thesis considers it to be static.

**Twitter**

A social network where users can share short public messages called Tweets with other users. Twitter's social graph $G = (U, E)$ consists of directed edges.

**Tweet**

A short message on Twitter of up to 140 characters in length, and some meta-data, of which the timestamp of the Tweet's creation, its internal unique identifier, and the Tweet's author are the most important for this thesis.

**Retweet**

A reshared Tweet, consisting of the same meta-data fields as a Tweet, and additionally a meta-data field for the original Tweet.

**Author**

A user who created a specific Tweet or Retweet.

**Retweet Cascade**

A chain of Retweets, containing the original Tweet and all its Retweets. For this thesis, the retweet cascade is identified by the original Tweet's unique ID.

**Prefix**

For a Retweet cascade, a set mapping each user in the cascade to the time of their Tweet or Retweet. For a specific Retweet, its (logical) prefix is the subset of the cascade's prefix containing those users whose associated timestamp is smaller than the Retweet's timestamp. Thus, the prefix of the first Retweet in a cascade contains only the original Tweet's author; the prefix of the $n^{\text{th}}$ Retweet is the prefix of the $(n-1)^{\text{th}}$ Retweet appended with that Retweet's author.

**Follower**

A Twitter user who subscribed to another user's Tweets. This is a unidirectional relationship, i. e. the other user does not have to follow them back. In terms of the social graph $G = (U, E)$, a follower is the user $u_1 \in U$ of a directed edge $(u_1, u_2) \in E$.

**Friend**

A Twitter user another user is following. In terms of the social graph $G = (U, E)$, a friend is the user $u_2 \in U$ of a directed edge $(u_1, u_2) \in E$.

**Timeline**

A user's stream of their friends' Tweets and Retweets. The timeline is updated in real-time and displayed in descending chronological order.

**Influencee**

A user who was inspired by another user to share some content.

**Influencer**

A user who inspired another user to share some content. For this thesis, an influencer in a Retweet cascade is a user's friend who shared the original Tweet (either by posting it themselves or by retweeting it) before the influenced user.

**Influence Graph**

For a Retweet cascade and a social graph $G = (U, E)$, a temporal directed graph $G' = (U', E')$ of users $U' \subseteq U$, containing exactly those users whose Tweets are part of the Retweet cascade, and influence edges $E'$, where an edge $(u_1, u_2, t_i) \in E'$, $u_1, u_2 \in U'$ indicates that $u_1$ is an influencer of $u_2$ at the time $t_i$ of $u_2$'s Retweet.

**Dataflow**

A computational model also referred to as *stream processing* that can be represented as a graph $G = (O, E)$ of operators $O$ and directed edges $E$. Data flows along the graph's edges through the computation. Operators process the data on their incoming edges by some inner logic and emit the output data to their outgoing edges. Operators without incoming edges are *data sources*, operators without outgoing edges *data sinks*, introducing data into the computation and returning the computation's result, respectively.

## 1.3. Problem Statement

Given a social graph $G = (U, E)$, with a set of users $U$ and a set of directed edges $E$, where an edge $(u_1, u_2) \in E$, $u_1, u_2 \in U$ indicates that $u_1$ is following $u_2$, and given a stream of Tweets $\mathcal{T}$. This stream $\mathcal{T}$ consists of an initial Tweet $T^*$, and multiple Retweets $T_i$ of $T^*$. The original Tweet $T^*$ consists of the user $u_1 \in U$ who posted the Tweet, a unique identifier, and a timestamp $t^*$ when the Tweet was posted; the Retweets $T_i$ consist of the original Tweet $T^*$, the retweeting user $u_2 \in U$, a unique identifier, and a timestamp $t_i$ when the Retweet was created.

Identify the influence graph, that is, a temporal directed graph $G' = (U', E')$ of exactly those users $U' \subseteq U$ whose Tweets are part of the stream $\mathcal{T}$, and influence edges $E'$, where an edge $(u_1, u_2, t_i) \in E',\ u_1, u_2 \in U'$ indicates that $u_1$ is a possible source of $u_2$'s Retweet $T_i$ at the Retweet's time $t_i$. User $u_1$ is the *influencer* of the *influencee* $u_2$.

For simplicity, assume that the social graph $G$ is static, that is, there are no changes to $G$ during the identification of the Retweet cascade. The computation of the Retweet cascade should happen in real-time. In a simple model looking only at 1-hop neighborhoods, each edge $(u_1, u_2, t_i) \in E'$ implies that $(u_2, u_1) \in E$, that is, save for the inversion of the edges' directions, $E' \subseteq E$, and thus $G'$ can be seen as a subgraph of $G$, $G' \subseteq G$.[8]

[8] In real life, users might come across Tweets they want to retweet outside of their timeline, e. g. when browsing user profiles of users they are not following and do not want to follow, when reading news articles embedding a Tweet, or when being mentioned in a Tweet by a user they are not following.

# 2. Related and Previous Work

Computing influences on Twitter has been studied before by a number of researchers. For example, in 2010, Cha et. al. [5] measured three different types of influences: the number of followers a user has, the number of times a user's Tweets are retweeted, and the number of times a user is mentioned in other users' Tweets. These three measures were found to be not directly dependent on each other, for example, a user with many followers is not necessarily retweeted often. Influence has also to be earned by users, and is not gained spontaneously, but once a user is influential, they remain so over some time. In contrast, Bakshy et. al. [2] concluded in 2011 that a larger number of followers leads to larger cascades. As Twitter's Retweet feature did not exist at the time of their study, cascades were not defined as Retweets of the same original Tweet, but as all Tweets containing the same URL. Both these papers calculated individual influence measures instead of influence graphs.

The first reconstruction system for influence graphs based on Retweet cascades was implemented at the research group on Web Science. The naïve approach is a cubic algorithm: for each new Retweet, check if for each previous Retweet or each previous user, the target of each relation is the current user.

Sättler and Ebner implemented the first reconstruction system on top of Apache Storm in their master's project [25]. Their approach was only centralized and not yet distributed, and could only process a single cascade at a time, but it provided the basic infrastructure and improved the naïve algorithm to a quadratic running time by regarding the relations as a set with a constant look-up time:

**Prefix Iteration** The prefix is the set of users who have retweeted the same Tweet before a given time, including the user of the original Tweet. For every Retweet $T$ in a cascade, posted by user $u$ at time $t$, the algorithm iterates over the current prefix. For each user $u_i$ in the prefix, the algorithm checks if $u$ is a follower of $u_i$ by doing a containment test on the set of $u$'s followers. If this the case, the algorithm produces an influence edge $(u_i, u, t)$. After the iteration, the Retweeting user $u$ is appended to the prefix. The algorithm's running time is in $\mathcal{O}(n^2)$ for $n$ Retweets.

The real social graph of Twitter is too large to be processed by a single computer in main memory. In his master's thesis [11], Huber therefore further developed the centralized system and distributed the social graph and the computation among multiple nodes, using a blocking implementation of the `Prefix Iteration` algorithm, that is, reconstruction could only start once all Retweets of the cascade were collected. The reconstruction was then executed on the node storing the followers of the original Tweet's author, the so-called *root node*. In order to reduce the expensive communication between nodes for exchanging follower information, Huber partitioned the social graph among the nodes using a community-based strategy: users following each other were identified and stored on the same node, assuming that retweeting mostly occurred within such communities.

Lutz evaluated this partitioning strategy and Huber's distributed implementation in his master's project [14]. He introduced two important changes: firstly, he replaced the formerly stateless protocol for exchanging user information between nodes with a stateful implementation, reducing the size of the messages shared between the nodes, and thus significantly the reconstruction time. Secondly, he proposed a more simple partitioning algorithm called *Naïve Random Partitioning* where a user and their followers are stored on a randomly chosen node, resulting in an equal distribution of the social graph[9]. This strategy turned out to outperform Huber's community-based partitioning, and Lutz could identify the `Prefix Iteration` to be the bottleneck in the distributed reconstruction instead of the graph partitioning, which was Huber's assumption. Therefore, he proposed the `User Iteration` reconstruction algorithm:

**User Iteration** For large cascades, iterating over the prefix will quickly become very expensive, as the prefix will grow with every Retweet in a cascade. Instead, it is cheaper to iterate over the retweeting user's friends—on average, users have only 102 friends [3]. Therefore, for every Retweet $T$ in a cascade, posted by user $u$ at time $t$, the `User Iteration` algorithm iterates over $u$'s friends. For each friend $u_i$, the algorithm checks if $u_i$ has been active in this cascade by doing a containment test on the prefix. If this is the case, the algorithm produces an influence edge $(u_i, u, t)$. After the iteration, the Retweeting user $u$ is appended to the prefix. The algorithm's running time s in $\mathcal{O}(n \cdot k)$ for $n$ Retweets and on average $k$ friends per user.

Lutz then implemented the `User Iteration` algorithm in his master's thesis [15], and several other optimizations, which enabled him to reconstruct a cascade of about 3.6 million Retweets in less than twenty seconds—previously, this cascade took twenty-two minutes to complete.

An Apache Storm application can be distributed among multiple *nodes* of machines. Each node runs (possibly several) *worker* processes, which in turn consist of at least a single *executor* thread. These executors each run one of two types of components, called *spouts* and *bolts*. Spouts are data sources, emitting the data that will be processed by the bolts, performing the processing logic. All spouts and bolts are connected in a directed acyclic graph (DAG), called *topology*, where an edge $(U, V)$ indicates that data flows from $U$ to $V$.

The topology of Lutz's Apache Storm reconstruction system is illustrated in figure 5. It consists of $n$ nodes, where one node called the *Spout* collects the Retweets and distributes them to their respective root nodes, and the other $n-1$ nodes store their assigned partitions of the social graph and perform the actual reconstruction. The Spout node is made up of a single Apache Storm spout and three bolts:

- The `Twitter Spout` gathers the Retweets from a specific source[10], and passes them on to the `Window Bolt`.

[9] At least on the number of nodes in the social graph; depending on the number of edges, partitions might still require different amounts of main memory.

[10] So far, only JSON files have been used, but other sources are possible as well, e. g., the Twitter API.

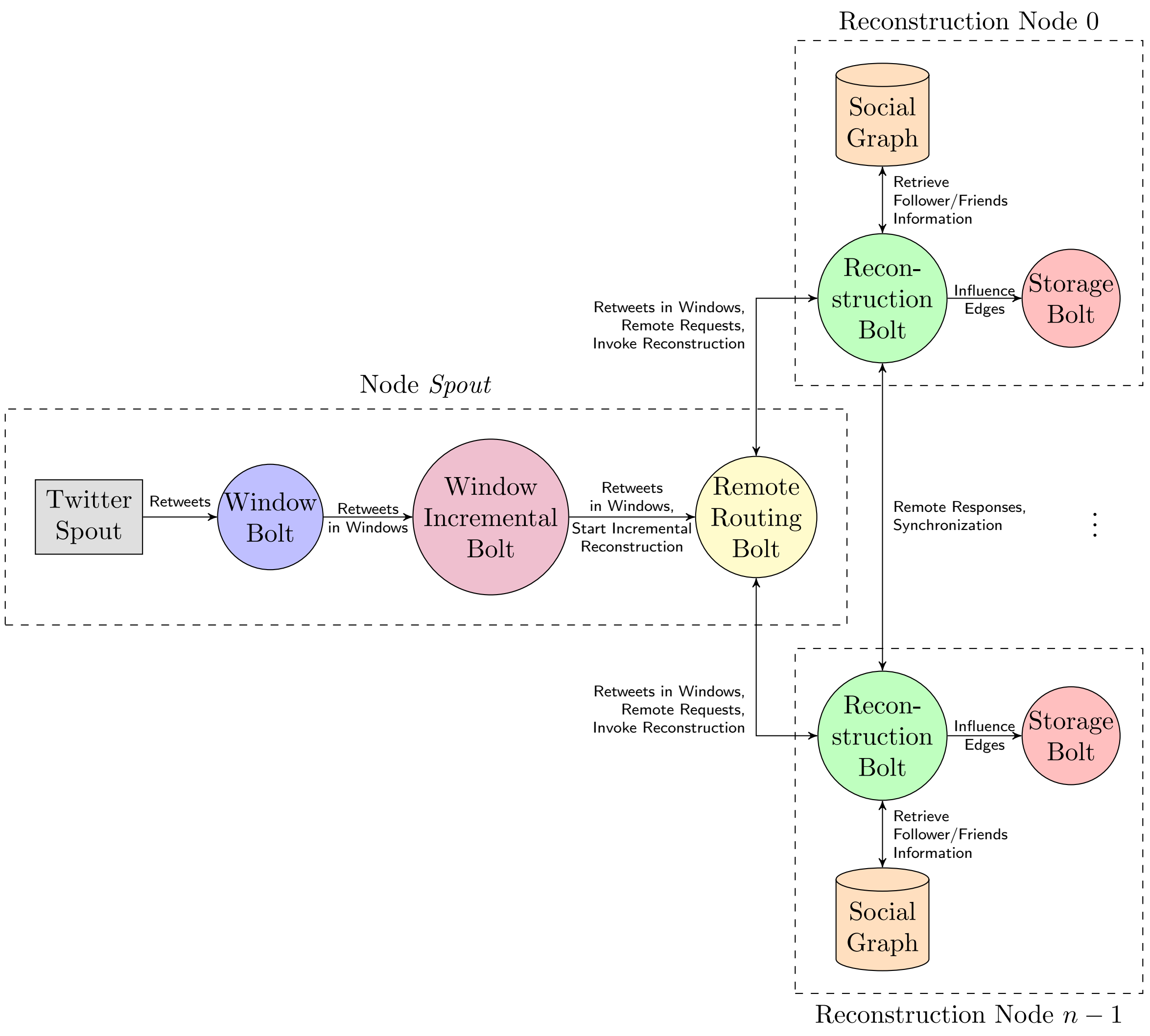

Reconstruction Node 0
Social Graph
Retrieve Follower/Friends Information
Recon-struction Bolt
Influence Edges
Storage Bolt
Retweets in Windows, Remote Requests, Invoke Reconstruction
Node *Spout*
Twitter Spout
Retweets
Window Bolt
Retweets in Windows
Window Incremental Bolt
Retweets in Windows, Start Incremental Reconstruction
Remote Routing Bolt
Remote Responses, Synchronization
Retweets in Windows, Remote Requests, Invoke Reconstruction
Recon-struction Bolt
Influence Edges
Storage Bolt
Retrieve Follower/Friends Information
Social Graph
Reconstruction Node $n-1$

Figure 5: The Apache Storm topology implemented by Lutz [15]. Bolts are depicted as circles, spouts as rectangles.

- The `Window Bolt` separates the possibly intermingled Retweet cascades into *windows*, with one window per cascade, to simplify the processing. These windows are then sent to the `Window Incremental Bolt`.
- The `Window Incremental Bolt` keeps track on the number of Retweets and the elapsed time since the last reconstruction. If these values surpass given thresholds, it instructs the `Remote Routing Bolt` to initiate reconstruction. Retweets are passed on to the `Remote Routing Bolt`.
- The `Remote Routing Bolt` knows which reconstruction node stores which user. It sends each Retweet to that Retweet cascade's root node[11]. If the user of the

[11] That is, the reconstruction node storing the original Tweet's author.

Retweet is not stored on the root node, but on a different node, it also instructs this *remote node* to send that user's information[12] to the root node. After the last Retweet in a cascade, or when told by the `Window Incremental Bolt`, the `Remote Routing Bolt` orders the reconstruction nodes to reconstruct the cascade.

The reconstruction nodes consist of two bolts per node:

- The `Reconstruction Bolt` has access to its node's partition of the social graph and executes the reconstruction algorithm (`Prefix/User Iteration`). Its behavior for a given cascade depends on whether the node it is a part of is a remote node for this cascade or the cascade's root node:
  - As part of a remote node, the `Remote Routing Bolt` can instruct the `Reconstruction Bolt` to send a *remote response* to the root node, containing a given user's follower or friend list (depending on the reconstruction algorithm).
  - As part of the root node, the `Reconstruction Bolt` receives the Retweets from the `Remote Routing Bolt` and the remote responses with the needed user information from the remote nodes. Both the Retweets and the remote responses are buffered until the `Remote Routing Bolt` instructs the Reconstruction Bolt to perform the actual reconstruction. The computed influence edges are then sent to the `Storage Bolt`.
- The `Storage Bolt` receives the influence edges emitted by the `Reconstruction` and saves them to disk.

This system assumes a static social graph, that is, there are no changes to it while the system is running. Furthermore, messages sent between nodes are not only assumed not to get lost, but also to be received in the same order they were sent. Node failure is not considered.

[12] The user's followers for the `Prefix Iteration` algorithm, the user's friends for the `User Iteration` algorithm.

# 3. Differential Dataflow

*Differential Dataflow* is a data-parallel computational model introduced by *Microsoft Research Silicon Valley* in 2012 [18, 19, 21][13]. In contrast to other data-parallel models such as *Apache Hadoop* and *MapReduce*, it supports both incremental input updates and iterative stream processing. This combination of features makes Differential Dataflow unique in the field of data-parallel computational models, and thus has been chosen as the framework for this thesis.

In traditional dataflow systems, operators receive and transform sets of data called *collections*, for each new input recomputing the entire collection even if it has not changed with respect to the previous round of input. Differential Dataflow considers only these differences: only the changes in the input will be recomputed, possibly resulting in an empty output because no new information could be retrieved from the input[14]. This requires *every* operator to maintain a state containing information on all previous inputs and outputs.

For example, in traditional systems, an operator computing the distinct elements in its input does not need any further information other than the input itself to compute the output, and is therefore stateless. Differential Dataflow operators, however, consider the previous rounds of computation and thus are stateful. This example is illustrated in figure 6.

At time $t = 0$, the operators receive no input, and therefore produce no output. At time $t = 1$, the input consists of two $A$'s, one $B$, and one $C$. Since the Differential Dataflow operator receives differences with regard to its previous input, it sees that it has gotten two more $A$'s, one more $B$, and one more $C$ than before. It also remembers that it has not produced any output before; and thus its output now consists of one more $A$, $B$, and $C$ each than before. The next input, at time $t = 2$, is one $A$, $B$, and $C$ each; for the Differential Dataflow operator, this is one $A$ less than its previous input. In contrast to the traditional stateless operator which has to recompute its entire output, the stateful operator notices that this input does not change its overall output, and therefore its output difference (with regard to the previous output) is empty. At the final time $t = 3$, the input consists of a $B$ and a $C$, which—again—is an $A$ less than before for the Differential Dataflow operator. This, however, does change the overall output, since there is no $A$ anymore in the total collection (i. e. the sum of all differences up to this time). Therefore, the Differential Dataflow operator propagates the removal of an $A$.

The key concept enabling Differential Dataflow is its time model, *Timely Dataflow.* It differs from other models in the order applied to the timestamps: instead of a total order, Timely Dataflow uses a partial order over times, allowing for multi-dimensional time grids. The first dimension is the collection's *epoch*, that is, a new batch of input data. All higher dimensions are (nested) fixed-point iterations on the data, that is, some

[13] Unless otherwise noted, this chapter and its subsections are summarizations of these three papers, especially [18]. With the exception of figure 10, the figures are taken from [18], as well.

[14] An empty output does not necessarily mean that the result of a computation is empty; it simply states that the output is the same as in the previous round.

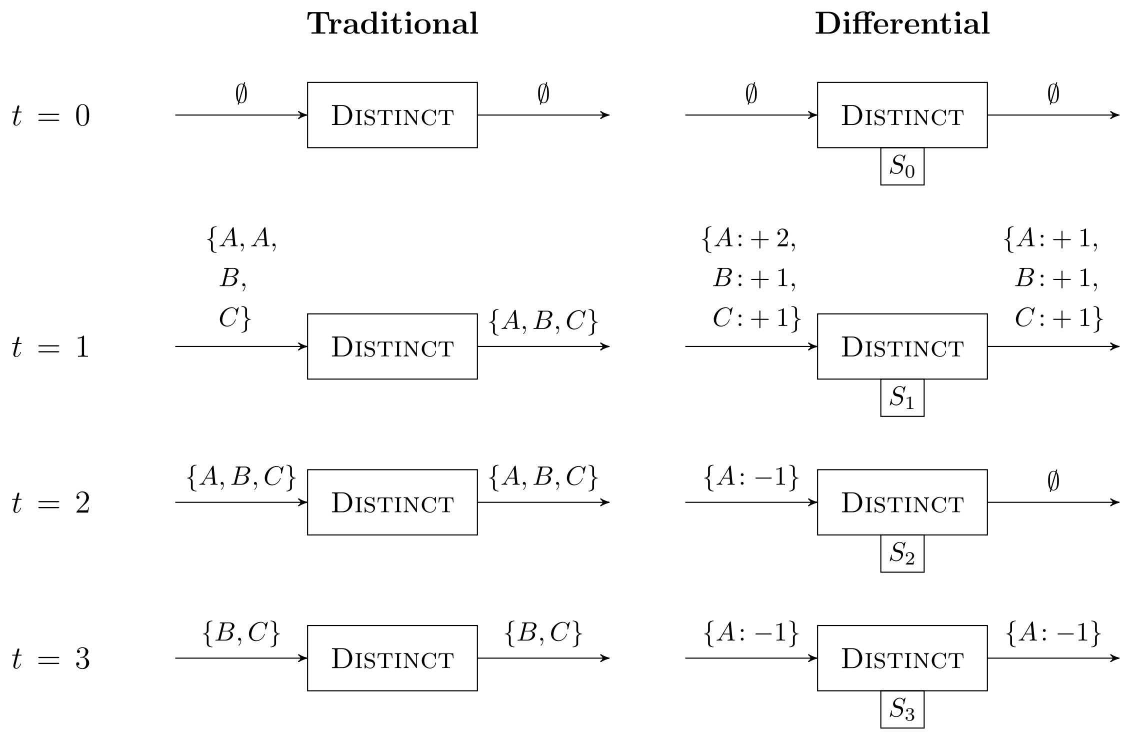


Figure 6: An example visualizing a stateless `DISTINCT` operator in a traditional dataflow system and a stateful `DISTINCT` operator in Differential Dataflow with state $S_t$ for time $t$.

function $f(x)$ applied $n$ times to the data $x$, $f^n(x)$, until $f^n(x) = f^{n-1}(x)$. Figure 7 illustrates the partial order in a two-dimensional time grid.

The collection at time $t$ is the sum of all previous differences. Since the timestamps of these differences have a partial order, there are multiple ways to derive the entire collection. Figure 8 visualizes this concept: In the first batch of input data, an $A$ and a $B$ are introduced into the computation. During the fixed-point iteration, a $C$ is added, then subtracted, and finally a $B$ is subtracted, resulting in a collection only containing an $A$ in the end. In the second batch, a $B$ is subtracted. Since the collection contained an $A$ and a $B$ at the same point in the first batch's fixed-point iteration, the collection contains only the $A$ in the second batch. Although the fixed-point iteration does not introduce any changes in the next two steps, the total collection changes, since the first batch's iterations have to be reflected. The next iteration is of particular interest: the collection does not contain a $B$ at this point, but the first batch's fixed-point iteration removes a $B$. Since collections cannot contain negative amounts of an entry, the fixed-point iteration in the second batch has to be a $B$ to keep the state consistent.

## 3.1. Formal Model

A collection $A$ is a function for some record type $R$ mapping to integers, $\mathbb{Z}$, $A : R \to \mathbb{Z}$, where $A(r), r \in R$ is the frequency of the record $x$ in the collection $A$. Addition and

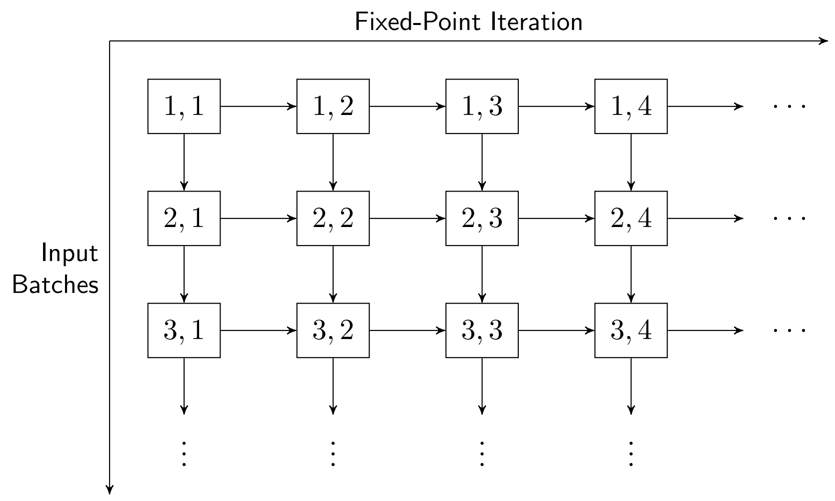


Figure 7: An example of a two-dimensional time lattice in Timely Dataflow for multiple rounds of inputs, each processed by a fixed-point iterator. For two times $t_1, t_2$, $t_1 < t_2$ if there is a path along the directed edges from $t_1$ to $t_2$.

subtraction are applied to two collections $A : R \to \mathbb{Z}$ and $B : R \to \mathbb{Z}$ by applying them to their respective frequencies:

$$(A + B)(r) = A(r) + B(r) \tag{1}$$
$$(A - B)(r) = A(r) - B(r) \tag{2}$$

An operator $f$ is a function from collections $A_i : R_i \to \mathbb{Z}, i \geq 1$ to another collection $B : R_B \to \mathbb{Z}$, $f : A_1 \times A_2 \times \cdots \times A_n \to B$. A key function $k_i : A_i \to K$ can be applied to an operators input collections $A_i$, mapping the input records to a key of type $K$ that is common to all the operator's input collections. Because of equations (1) and (2), an operator $f$ can therefore be rewritten as:

$$f(A_1, A_2, \ldots, A_n) = \sum_{k \in K} f(A_{1_k}, A_{2_k}, \ldots, A_{n_k}) \tag{3}$$

The collections $A_{i_k}$ in equation (3) are partitions of their respective base collection $A_i$ based on their associated key function $k_i$, containing those records $r \in R$ which match the key $k$:

$$A_{i_k}(r) = \begin{cases} A_i(r), & \text{if } k_i(r) = k \\ 0, & \text{else} \end{cases} \tag{4}$$

This application of key functions enables data-parallel computations by distributing an operator among multiple threads, where each thread is responsible for one or more keys.

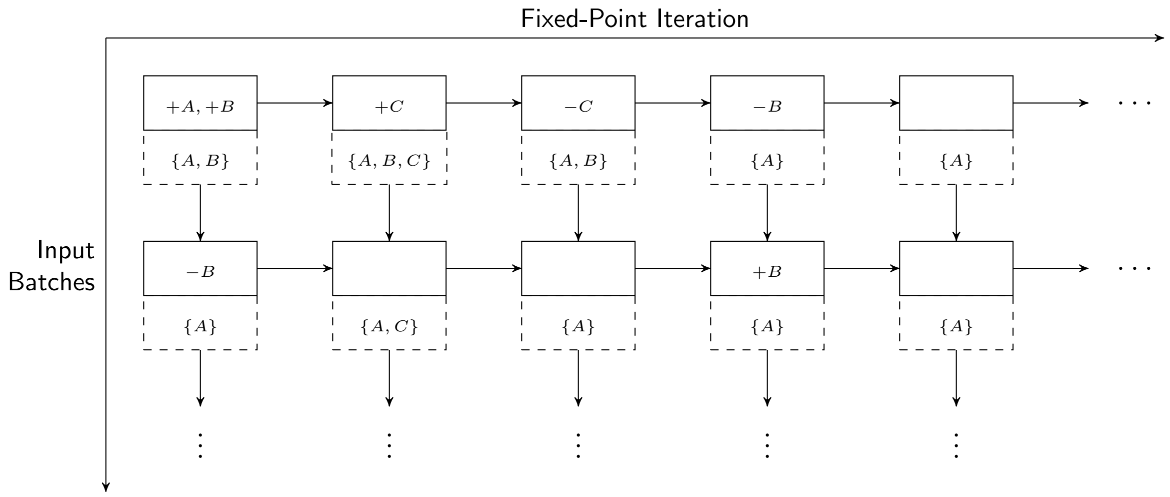


Figure 8: An example of deriving collections from differences in a two-dimensional time grid. Differences are given in solid rectangles, the corresponding derived collection in the dashed rectangle.

Collections vary over time. This *collection trace* is given as a function over a time lattice $T$: $\mathscr{A} : T \to (R \to \mathbb{Z})$ with $\mathscr{A}(t), t \in T$ being the collection $A : R \to \mathbb{Z}$ at time $t$. Instead of storing complete collections $\mathscr{A}(t)$ for each time $t \in T$ for a collection trace $\mathscr{A}$, the differences between collections can be stored. These *difference traces*, $\delta\mathscr{A} : T \to (A \to \mathbb{Z})$, reflect the frequency changes of a collection over time. The entire collection $\mathscr{A}(t)$ at a time $t \in T$ then is simply the sum of all differences up to $t$:

$$\mathscr{A}(t) = \sum_{s \leq t} \delta\mathscr{A}(s) \tag{5}$$

$$\delta\mathscr{A}(t) = \mathscr{A}(t) - \sum_{s < t} \delta\mathscr{A}(s) \tag{6}$$

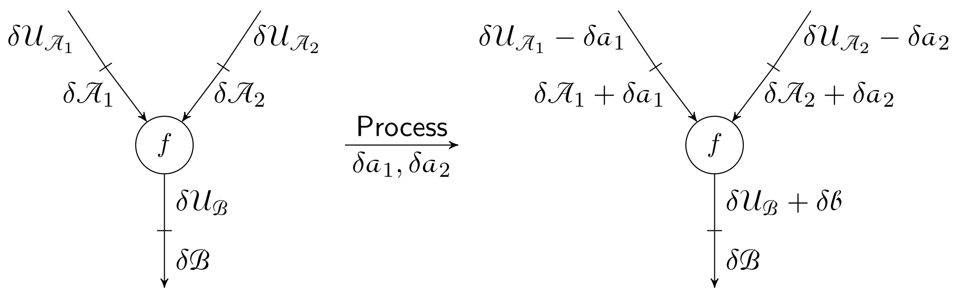
Equation (6) is a simple rearrangement of equation (5) for a more explicit definition of a difference trace. Since these two equations show it is trivially possible to convert between collection and difference traces, the term *trace* will be used below to refer to both.

Differential Dataflow's execution model is a directed, possibly cyclic dataflow graph, where the edges correspond to traces, and the nodes correspond to inputs and outputs of the computation, and operators. The trace on an operator node's outgoing edge is the result of applying the operator to the traces on its node's incoming edges.

Each edge in the dataflow graph is assigned two traces: the first one is a trace of records that have been processed by the edge's target node, the second trace contains all records that the edge's source node has emitted but that have not been processed by the target node. In the beginning, all traces are empty. The computation makes progress in one of two ways:

1. Either, an input node adds a trace to the `unprocessed` trace on its output edge,
2. or, an operator node moves a record on an input edge from its `unprocessed` trace to the corresponding `processed` trace, applies the operator's logic to that record, and adds the result to the `unprocessed` trace on its output edge.

Once all `unprocessed` traces in the dataflow graph are empty, the computation is finished until an input node introduces new traces into the computation.



$\delta\mathcal{U}_{\mathcal{A}_1}$
$\delta\mathcal{U}_{\mathcal{A}_2}$
$\delta\mathcal{A}_1$
$\delta\mathcal{A}_2$
$f$
$\delta\mathcal{U}_{\mathcal{B}}$
$\delta\mathcal{B}$
Process
$\delta a_1, \delta a_2$
$\delta\mathcal{U}_{\mathcal{A}_1} - \delta a_1$
$\delta\mathcal{U}_{\mathcal{A}_2} - \delta a_2$
$\delta\mathcal{A}_1 + \delta a_1$
$\delta\mathcal{A}_2 + \delta a_2$
$f$
$\delta\mathcal{U}_{\mathcal{B}} + \delta b$
$\delta\mathcal{B}$

Figure 9: Visualization of the advancement of a general Differential Dataflow operator $f$ with two inputs $\mathcal{A}_1$ and $\mathcal{A}_2$ with output $\mathcal{B} = f(\mathcal{A}_1, \mathcal{A}_2)$. The `processed` traces are given as $\delta\mathcal{T}$, the corresponding `unprocessed` ones as $\delta\mathcal{U}_{\mathcal{T}}$. When advancing, the operator processes $\delta a_1$ and $\delta a_2$ from its unprocessed inputs, resulting in $\delta b$.

Figure 9 visualizes the second possibility for advancement in a computation for an arbitrary operator $f(\mathcal{A}_1, \mathcal{A}_2)$ with output $\mathcal{B}$: it subtracts the records it will update from the `unprocessed` input traces and adds them to the corresponding `processed` input traces. The result of applying $f$ to these newly processed records is then added to the `unprocessed` output trace.

## 3.2. Implementations

Microsoft Research Silicon Valley have developed a reference implementation of Differential Dataflow called *Naiad*. It is written in `C#` and available as open-source software. [20] The development, though, has been ceased, since Microsoft closed the research group in 2014.

However, McSherry, co-author of Differential Dataflow and co-developer of Naiad, has ported Naiad to `Rust` and improved the model. This implementation is actively maintained and freely available as two libraries [17, 16].

By default, a Differential Dataflow computation runs as a single process with two threads: one main thread for initialization and clean-up, and one thread performing the computation, called a *worker*. The number of workers can arbitrarily be increased to parallelize the computation. A computation using $w$ workers thus runs $w + 1$ threads in total.

The computation can further be distributed across multiple processes; these can be run on a single or on multiple machines. Inter-process communication is always handled on the system's network via `TCP/IP`. Each process must run the same amount of workers; the total number of workers in a distributed computation with $n$ processes and $w$ workers per process therefore is $n \cdot w$.[15]

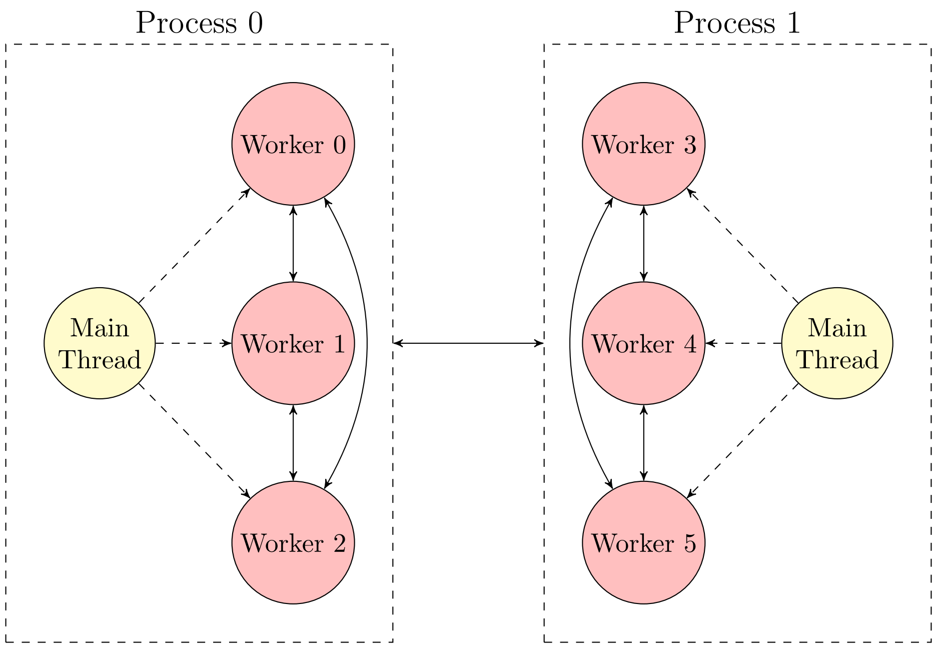


Figure 10: A distributed Differential Dataflow computation with two processes and three workers per process.

The data being processed in a Differential Dataflow computation has an attached timestamp indicating its round of input and loop iteration. Data with the same timestamp belongs to a *batch*. Since later batches can depend on previous data, the Differential Dataflow operators exchange information on their progress on each batch to synchronize the computation. Therefore, the smaller a batch is, the more frequent the operators will have to synchronize and exchange progress information. For multiple workers, these progress notifications have to be communicated to the operators on all workers. Furthermore, data belonging to a batch can be collected and sent at once, resulting in larger, but fewer messages for larger batches.

Within a batch, each operator preserves the order of the data in which it arrives. A single worker will thus return the output data in the order corresponding to the input data's order. If the computation is distributed across multiple workers, each batch will be partitioned among the workers according to the specified strategy. Within each partition of a batch, the order is preserved, and thus its output is ordered correspondingly. The overall output, however, can be interleaved arbitrarily, and an operator accepting data from multiple workers is not guaranteed to receive it in any particular order.

[15] The total number of threads is $n(w + 1)$.

# 4. Implementation

As part of this thesis, an influence graph reconstruction system called `CRGP`[16] was implemented using the Timely Dataflow[17] framework [17] in `Rust`. Details of the implementation will be explained in this chapter, using the conventions listed in table 1.

| **Expression** | **Explanation** |
|---|---|
| $\mathcal{R}$ | Set of all Retweets. |
| $\mathcal{T}$ | Set containing the original Tweets of all Retweets $r \in \mathcal{R}$. |
| $\mathcal{T}[r],\ r \in \mathcal{R}$ | The original Tweet of the Retweet $r$. |
| $R_{\mathtt{t}} = \{r \in \mathcal{R} \mid \mathcal{T}[r] = \mathtt{t}\}$ | Retweet cascade of the original Tweet $\mathtt{t}$. |
| $t[r],\ r \in \mathcal{T} \cup \mathcal{R}$ | Timestamp of the Tweet or Retweet $r$. |
| $G = (U, E),\ U \subset \mathbb{Z}$ | Social graph with users $U$ and directed edges $E$. |
| $F(u) = \{f \mid (u, f) \in E\},\ u \in U$ | Friends of user $u$. |
| $U[r],\ r \in \mathcal{T} \cup \mathcal{R}$ | Author of the Tweet or Retweet $r$. |
| $P[R_{\mathtt{t}}] = \{U[r] \mapsto t[r] \mid r \in R_{\mathtt{t}} \cup \{\mathtt{t}\}\}$ | Prefix for the cascade $R_{\mathtt{t}}$, mapping each user in the cascade to the time of their Tweet or Retweet. |
| $(f, U[r], t[r]),\ r \in \mathcal{R}, f \in F(U[r])$ | An influence edge where $f$ influenced $U[r]$. |
| $(f, U[r], t[r], U[\mathcal{T}[r]]),\ r \in \mathcal{R}, f \in F(U[r])$ | A possible influence edge, including the author of the original Tweet. |
| $W$ | Total number of workers. |
| $w(u),\ u \in U$ | The worker storing user $u$'s friendships $(u, f) \in E$. |

Table 1: Conventions used throughout this chapter.

Each worker has a unique identifier $0 \leq i < W$. The worker with ID $i = 0$ is the main worker in the sense that it handles input and output, that is, it reads the social graph and Retweet cascade data sets from their input files and distributes them among all workers, and reports the statistics collected during reconstruction. Apart from that, the workers behave the same.

[16] Its source code is available at `https://github.com/BMeu/CRGP`; usage documentation can be found in appendix A.

[17] For the current implementation, Differential Dataflow is not required, although there are ideas where it can be used. These will be discussed in chapter 7.

The social graph data set can reduce its disk usage by only containing those friends for each user that are also part of the Retweet cascades. Since this falsifies the measurements, the social graph data set can specify how many friends each user actually has. `CRGP` can use this information and create the required amount of dummy friends for each user to fill the social graph. These dummy users receive user IDs $u < 0$; real users have IDs $u \geq 0$ as assigned by Twitter.

## 4.1. Partitioning

In [14], Lutz found a naïve random partition strategy to perform better than strategies involving intrinsic information of the graph, for example, community-based clustering. `CRGP` uses a similar, yet deterministic, partitioning strategy based on the user's ID and the number of workers:

$$w(u) := u \bmod W,\ u \in U \tag{7}$$

This partitioning strategy has the advantage of being a cheap and fast computation of $w(u)$. While loading the social graph data set, each edge can immediately be sent to its respective worker, without performing any graph analysis first. Since $W$ is known to all workers, $w(u)$ can be computed by any worker when needed, and therefore no additional data structure for associating users with their workers has to be maintained.

## 4.2. Algorithms

Based on the `User Iteration` algorithm, two reconstruction algorithms were implemented for `CRGP`, differing in the way the prefix is maintained. These are described below.

The overall computation setup loading the data, initializing Timely Dataflow, and managing the computation is given in listing 1. The outermost function `run()` is executed by each process's main thread, and immediately initializes the Timely Dataflow workers with the given configuration. Each worker then sets up the dataflow graph for the chosen algorithm—this does not yet execute the reconstruction itself. The algorithms are described in full detail in the following subsections. Afterwards, worker 0 loads and instantly distributes the social graph; this is handled in the `tar::load()` function. Once the entire social graph is processed on all workers, worker 0 loads and parses the Retweets, before inserting them into the computation where they are processed by the reconstruction algorithms. After every `batch_size` Retweets, worker 0 starts a new batch. After the last Retweet has been introduced into the dataflow graph, worker 0 waits until the entire computation is finished.

```
pub fn run(mut configuration: Configuration) {
    let timely_configuration: TimelyConfiguration =
        configuration.get_timely_configuration();

    timely_execute(timely_configuration, move |computation| {
        let index = computation.index();
        let algorithm = configuration.algorithm;
        let output_target: OutputTarget =
            configuration.output_target.clone();

        // Set up the dataflow graph.
        let (mut graph_input, mut retweet_input, probe) =
            computation.dataflow::<u64, _, _>(move |scope|
        {
            match algorithm {
                Algorithm::GALE => gale(scope, output_target),
                Algorithm::LEAF => leaf(scope, output_target)
            }
        });

        // Worker 0: Load the social graph into the computation.
        if index == 0 {
            let input: InputSource =
                configuration.social_graph.clone();
            tar::load(input, configuration.pad_with_dummy_users,
                      &mut graph_input);
        }
        computation.sync(&probe, &mut graph_input,
                         &mut retweet_input);

        // Worker 0: Preload the Retweets.
        let retweets: Vec<Retweet> = if index == 0 {
            twitter::get::from_source(configuration.retweets.clone())
        } else {
            Vec::new()
        };

        // Process the retweets.
        let batch_size: usize = configuration.batch_size;
        for (round, retweet) in retweets.iter().enumerate() {
            retweet_input.send(retweet.clone());

            // Sync the computation after each batch.
            let is_batch_complete: bool =
                round % batch_size == (batch_size - 1);
            if is_batch_complete {
                computation.sync(&probe, &mut retweet_input,
                                 &mut graph_input);
            }
        }
        computation.sync(&probe, &mut retweet_input,
                            &mut graph_input);
    });
}
```

Listing 1: The overall Timely Dataflow computation setup.

### 4.2.1. LEAF

`LEAF` (**L**ocal **E**dges, **A**ctivations, and **F**iltering) partitions the prefix across all workers: each worker adds those users to the local prefix whose friendship edges are also stored on this worker. The Retweets are therefore sent only to the author's worker which then produces possible influence edges for each of the author's friends. These edges are sent to the workers storing the respective friends which then filter the edges based on their local prefix. This idea is sketched in figure 11. In detail, `LEAF` performs the following steps for each Retweet $r \in \mathcal{R}$, where $u := U[r]$ and $\mathsf{t} := \mathcal{T}[r]$:

1. Send $r$ to worker $w(u)$.
2. On worker $w(u)$:
   a) Add $u \mapsto t[r]$ to $P[R_\mathsf{t}]$.
   b) For each friend $f \in F(u)$: produce a possible influence edge $(f, u, t[r], U[\mathsf{t}])$ and send it to worker $w(f)$.
3. On workers $w(f)$: output the influence edge $(f, u, t[r])$ if
   - $f = U[\mathsf{t}]$, that is, the friend $f$ is the author of the original Tweet, or if
   - $f \in P[R_\mathsf{t}]$, and in that case,
   - $P[R_\mathsf{t}](f) < t[r]$.

The author $U[\mathsf{t}]$ of the original Tweet has to be in the prefix on worker $w(U[\mathsf{t}])$. However, the first possibility to add $U[\mathsf{t}]$ to this prefix is when the first Retweet $r$ with $w(U[r]) = w(U[\mathsf{t}])$ occurs in $R_\mathsf{t}$—for all Retweets occurring before $r$, the algorithm cannot add $U[\mathsf{t}]$ to the prefix on the correct worker, and therefore will miss the influence edges where the influencer is $U[\mathsf{t}]$.

By including $U[\mathsf{t}]$ in the possible influence edges $(f, U[r], t[r], U[\mathcal{T}[r]]),\ f \in F(U[r])$ for a Retweet $r$, this problem can be avoided: if the worker $w(f)$ also stores the friendships of $U[\mathsf{t}]$, it checks if $f$ is the author of the original Tweet, $U[\mathsf{t}]$, and in that case, produce the actual influence edge $(f, U[r], t[r])$.

Since each Retweet is sent to one worker, and this worker produces a possible influence edge for each of the author's friends, the number of messages is $|\mathcal{R}| \cdot (1 + |F|)$, where $F = \{f \mid (U[r], f) \in E,\ r \in \mathcal{T} \cup \mathcal{R}\}$. That is, $F$ is the set of all users whose followers are authors of Retweets in the data set $\mathcal{R}$ or are the original Tweet's authors of these

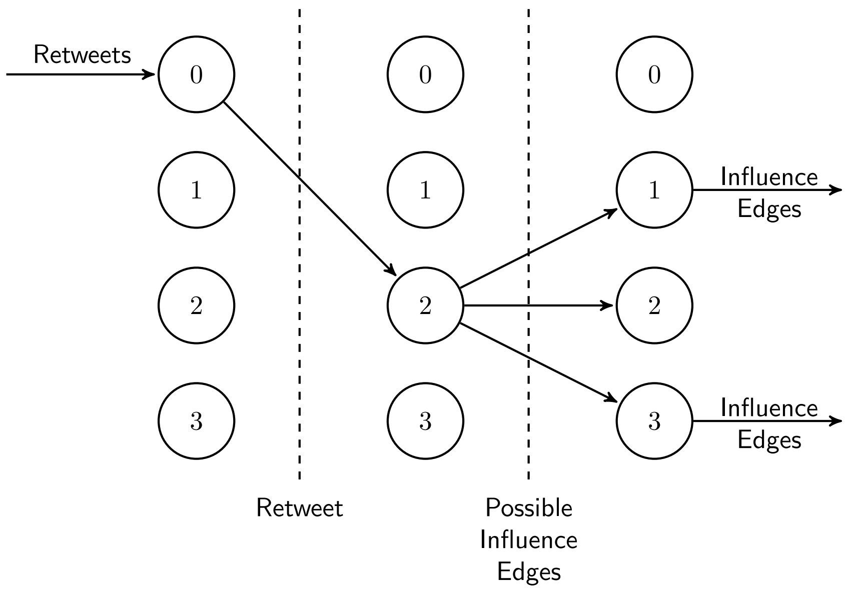


Figure 11: A conceptual sketch of the Retweet processing phase in the `LEAF` algorithm with four workers: worker 0 sends a Retweet to the worker storing the author's friendships (in this case, worker 2) which adds the Retweet's author to the prefix and sends possible influence edges for all the author's friends to the respective workers (in this case, workers 1, 2, and 3). If a friend is in the prefix, the respective influence edge is output (in this case, worker 2 does not receive such possible influence edges).

Retweets. On average, the number of messages is $|\mathcal{R}| \cdot (1 + \frac{|E|}{|U|})$. For each processed Retweet, it takes two hops across the workers to produce the resulting influence edges.

The top-level implementation using Timely Dataflow is shown in listing 2. The function `leaf()` sets up the dataflow graph and returns handles for interacting with it. First, the input handles for the social graph and the Retweets, and the corresponding streams for the dataflow graph are created. In the next step, the data structure for the prefix is initialized.[18] The dataflow graph is initialized by passing the streams and a reference to the prefix to the custom `find_possible_influences` operator. Using Timely's `exchange` operator, its output is then sent to the worker storing the influencer. There, the possible influences are filtered as described above. The custom `write` operator afterwards writes the result to the specified output target. In the end, Timely Dataflow's `probe` operator returns a handle to the computation that indicates the computation's progress; this handle is used to control the batch sizes.

```rust
pub fn leaf<'a>(scope: &mut Scope<'a>, output: OutputTarget)
    -> (GraphHandle, RetweetHandle, ProbeHandle)
{
    let (graph_input, graph_stream) = scope.new_input();
    let (retweet_input, retweet_stream) = scope.new_input();

    let prefix: Rc<RefCell<HashMap<u64, HashMap<User, u64>>>> =
        Rc::new(RefCell::new(HashMap::new()));

    let probe = graph_stream
        .find_possible_influences(retweet_stream, prefix.clone())
        .exchange(|influence: &InfluenceEdge<User>| {
            influence.influencer.id as u64
        })
        .filter(move |influence: &InfluenceEdge<User>| {
            let is_influencer_activated: bool = match prefix.borrow()
                .get(&influence.cascade_id)
            {
                Some(users) => match users.get(&influence.influencer) {
                    Some(activation_timestamp) => {
                        &influence.timestamp > activation_timestamp
                    },
                    None => false
                },
                None => false
            };
```

[18] Since the prefix is used within two closures, `Rust` cannot guarantee at compile time that at most one of these closures has mutable access to the prefix. Therefore, the prefix must be wrapped to allow dynamic borrow checks. For more details on `Rust`'s borrow checker, refer to its documentation, for example, at `https://doc.rust-lang.org/stable/book/second-edition/ch04-00-understanding-ownership.html`.

```
            let is_influencer_original_user: bool =
                influence.influencer == influence.original_user;

            is_influencer_activated || is_influencer_original_user
        })
        .write(output)
        .probe();

    (graph_input, retweet_input, probe)
}
```

Listing 2: The top-level implementation of `LEAF`.

The custom `find_possible_influences` operator takes the social graph and the Retweet streams as inputs and distributes their elements correspondingly. The actual logic on each worker receives the users with their friends who will be stored on that particular worker, and inserts them into its social graph data structure. The Retweets are processed as described above.

### 4.2.2. GALE

`GALE` (**G**lobal **A**ctivations, **L**ocal **E**dges) maintains a complete prefix on all workers by broadcasting each Retweet. The worker storing the friendships of the current Retweet's author can then produce that user's influence edges without having to communicate with any other worker. This idea is sketched in figure 12. In detail, `GALE` performs the following steps for each Retweet $r \in \mathcal{R}$, where $u := U[r]$ and $\mathsf{t} := \mathcal{T}[r]$:

1. Broadcast $r$ to all workers.
2. Each worker:
   a) If $r$ is the first Retweet in its cascade $R_\mathsf{t}$, initialize $P[R_\mathsf{t}]$ with $U[\mathsf{t}] \mapsto t[\mathsf{t}]$.
   b) Add $u \mapsto t[r]$ to $P[R_\mathsf{t}]$.
3. On worker $w(u)$:
   a) If $|F(u)| \leq |P[R_\mathsf{t}]|$, iterate over $F(u)$: for each friend $f \in F(u)$, produce an influence edge $(f, u, t[r])$ if
      - $f \in P[R_\mathsf{t}]$, and in that case,
      - $P[R_\mathsf{t}](f) < t[r]$.

   b) Otherwise, iterate over $P[R_\mathsf{t}]$: for each user and their timestamp $f \mapsto t[r']) \in P[R_\mathsf{t}]$, produce an influence edge $(f, u, t[r])$ if
      - $f \in F(u)$, and
      - $t[r'] < t[r]$.

The production of the influence edges is in fact an adaptive combination of the `Prefix Iteration` and the `User Iteration` algorithms: if the prefix is shorter than the user's list of friends, `Prefix Iteration` is performed, otherwise `User Iteration`.

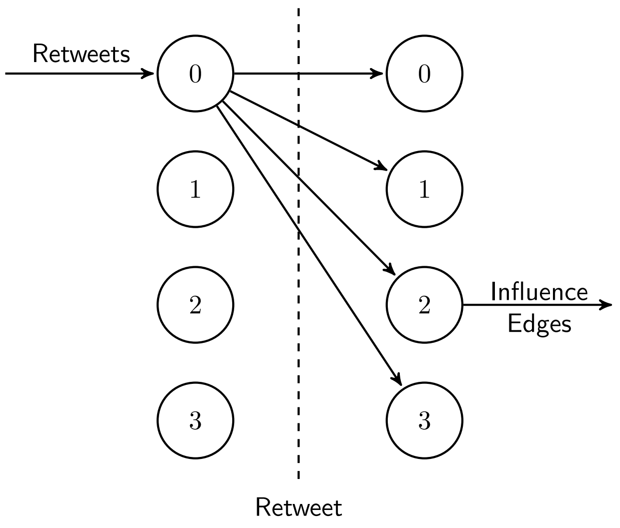


Figure 12: A conceptual sketch of the Retweet processing phase in the `GALE` algorithm with four workers: worker 0 broadcasts a Retweet to all workers which add the Retweet's author to the prefix. Only one worker (in this case, worker 2) stores the author's friendships and produces the influence edges.

The number of messages transmitted by `GALE` is $|\mathscr{R}| \cdot W$, since each Retweet is sent to each worker. For each processed Retweet, it takes one hop across the workers to produce the resulting influence edges.

The top-level implementation using Timely operators is shown in listing 3. The function `gale()` creates two inputs and streams, one for the social graph and one for the Retweets. The Timely Dataflow computation is built on top of the streams: The retweets are broadcast to all workers feeding directly into the custom `Reconstruct` operator which also takes the social graph as an input. The result of this operator is passed on to the—again, custom—`Write` operator which saves its input stream to the given output. As with the `LEAF` implementation, a probe handle is returned to monitor the computation's progress.

```
pub fn gale<'a>(scope: &mut Scope<'a>, output: OutputTarget)
    -> (GraphHandle, RetweetHandle, ProbeHandle)
{
    // Create the inputs.
    let (graph_input, graph_stream) = scope.new_input();
    let (retweet_input, retweet_stream) = scope.new_input();

    // The actual algorithm.
    let probe = retweet_stream
        .broadcast()
        .reconstruct(graph_stream)
        .write(output)
        .probe();

    (graph_input, retweet_input, probe)
}
```

Listing 3: The top-level implementation of `GALE`.

As does the `find_possible_influences` operator, the `reconstruct` operator distributes the social graph among all workers. The Retweets, however, are not further distributed, since they already are at all workers due to the `broadcast` operator. These are processed as described above.

# 5. Evaluation Methodology

For the analysis of the reconstruction system implemented in chapter 4, several experiments were conducted and evaluated. These experiments can be divided into three parts: analyzing `Rust`, analyzing Timely Dataflow, and analyzing the overall reconstruction system and its algorithms. After a general overview of the data sets and the hardware used for these measurements, their setup is described in more detail for the aforementioned groups of experiments. Unless specified otherwise, each setup has been run ten times to compensate measuring inaccuracies, and their arithmetic mean will be presented, with error bars indicating the standard deviation of the measurements.

The influence graphs for each data set reconstructed by `CRGP` must be equal to each other, independent of the configuration in the experiments. The results were therefore compared using the script given in listing 10 in appendix B. Differences did not occur, the results are therefore valid.

**Hardware Specifications** The machines used to run the measurements are listed in table 2. All instances are virtual machines. When running distributed measurements, Amazon Web Services (AWS) EC2 instances of the same type are used; these are connected via a 10 GBit/s network.

| Name | Cores | Clock Speed | Type | RAM |
|---|---|---|---|---|
| `Corse` | 10 | 2.5 GHz | Intel Sandy Bridge EP | 16 GiB |
| `Guatteri` | 16 | 2.5 GHz | Intel Sandy Bridge EP | 16 GiB |
| `r3.xlarge` | 4 | 2.5 GHz | Intel Xeon E5-2670 | 30.5 GiB |
| `r4.2xlarge` | 8 | 2.3 GHz | Intel Broadwell E5-2686 | 61 GiB |
| `r4.4xlarge` | 16 | 2.3 GHz | Intel Broadwell E5-2686 | 122 GiB |
| `r4.16xlarge` | 64 | 2.3 GHz | Intel Broadwell E5-2686 | 488 GiB |

Table 2: Statistics of the machines used for evaluation: for each machine, the number of CPU cores, the CPU clock speed, the CPU type, and the size of the main memory. All machines are virtual machines. The first two machines are maintained by the research group, the others are rented on Amazon Web Services EC2 (`https://aws.amazon.com/ec2/`).

The CPU cores on the AWS EC2 instances are virtual hyperthreads of the underlying physical CPU [1]. The physical machine running `Corse` and `Guatteri` has 12 physical CPU cores with enabled hyperthreading.

**Software Specifications and Default Configurations** Unless explicitly stated otherwise, the experiments were run on *Ubuntu 16.04* machines using `Rust` in version 1.18.

The social graph includes the implicitly given dummy users, and the computed influence graphs are written to files.

**Data Sets** In order to achieve results comparable to [15], the same data sets were used in this thesis for evaluation. These data sets were collected by the research group on Web Science and contain data from the 2012 Summer Olympics.

**Cascades** The Retweet data sets are select cascades from the 2012 Olympics. All Retweets in the data sets are sorted in temporal order, with unnecessary attributes projected out. The data sets are listed in table 3.

| Data Set | Cascades | Retweets | Average Retweets | Influence Edges |
|---|---|---|---|---|
| `smaller` | 1 | 3,500 | 3,500.0 | 2,664 |
| `small` | 1 | 7,226 | 7,226.0 | 16,210 |
| `medium` | 1 | 14,847 | 14,847.0 | 70,916 |
| `big` | 1 | 29,783 | 29,783.0 | 329,226 |
| `biggest` | 1 | 60,451 | 60,451.0 | 964,315 |
| `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 of the cascades used for evaluation: for each data sets, the number of cascades in that data set, the number of Retweets in the data set, the average number of Retweets per cascade, and the number of influence edges for the data set.

**Social Graph** The social graph consists of the users who participated in the cascades listed above. For each user, only the friends who were also active in the cascades are explicitly given; the total number of friends for each user is given as meta data. In total, the social graph consists of 3,177,361 users, with 341,698,525 explicitly given edges. Including the implicit edges from the meta data, the graph contains 1,397,187,866 edges.

## 5.1. Rust

Since the size of the social graph is the main reason to distribute the reconstruction, the `Rust` experiments analyze data structures for an efficient storage of the social graph, both in terms of memory requirements and temporal performance.

Four data structures using only collection types of `Rust`'s standard library[19] have been evaluated; these are listed in listing 4. The first two data structures use hash maps to associate users with their friends; in the first structure a hash set is used for the friends, in the second a vector[20]. The third and fourth data structures both are a tuple of two vectors: the first vector is a list of all users, the second a list of all hash sets or vectors of friends, respectively. Both these lists are of the same length; the friends of the user at index $i$ in the first list are stored at index $i$ in the second list.

```
HashMap<User, HashSet<User>>

HashMap<User, Vec<User>>

(Vec<User>, Vec<HashSet<User>>)

(Vec<User>, Vec<Vec<User>>)
```

Listing 4: The potential social graph data structures.

The memory usage of these data structures was measured on an `r3.xlarge` instance using the script given in listing 9 in appendix B, reporting the peak memory usage of the observed process in $KiB$. For these experiments, the social graph (without dummy users) was inserted into the data structures. The time to insert the social graph was measured as part of these experiments.

In addition, common operations performed on the `HashSet` and `Vec` collections were analyzed. Firstly, the performance of iterating over various sizes of these containers was measured; both with sorted and unsorted contents in case of the experiments on the `Vec` container. Secondly, containment tests in two scenarios were analyzed: containment tests for elements not in the collections, and containment tests for elements that are in the collections. The `Vec` container was again analyzed both with sorted and unsorted contents.

These experiments were measured using `Rust`'s built-in benchmark utility. This tool automatically determines the number of runs per benchmark, depending on the time for an initial run, but it collects at least fifty samples per benchmark. The points in the graphs are the median of these results and the error bars the deviation, that is, the difference between the measured maximum and the minimum.[21]

[19] Other collection types, including third-party libraries, have been tested as well, but since they were either multiple orders of magnitude slower than the structures evaluated in the end, or crashed the computers due to too high memory requirements, have been immediately discarded.

[20] Vectors (type `Vec<T>`) are `Rust`'s dynamically allocated arrays and thus comparable to e. g. `Java`'s `ArrayList`.

[21] The information presented in this paragraph is not officially documented anywhere, but was determined from the tool's source code (available at `https://github.com/rust-lang/rust/blob/cb92ab93a224a7728103f837ec761b1dafd5fbb1/src/libtest/lib.rs#L1606-L1670`).

## 5.2. Timely Dataflow

Message sharing between workers is handled by the Timely Dataflow framework. Since this is a critical part of distributed computations, its capacity was analyzed to understand how it saturates the network links. These measures also give a maximum performance the actual reconstruction algorithms can reach; these results can therefore be used to evaluate the algorithms excluding the effects of the network utilization.

For the Timely Dataflow experiments, worker 0 creates a list of 100 million Retweets and broadcasts these to all workers which then send them back to worker 0 without any further processing. The time from broadcasting the first Retweet until receiving the last Retweet back at worker 0 is measured; the performance is then given in *Retweets per Second* ($RT/s$). The experiments are performed on `r4.2xlarge` instances.

## 5.3. CRGP

The evaluation of the entire reconstruction system `CRGP` consists of two main parts: analyzing the partitioning strategy and thus the load distribution among all workers, and analyzing the performance when reconstructing the data sets presented at the beginning of this chapter.

### 5.3.1. Load Distribution

The distribution of the social graph is measured by logging the number of users and the number of edges stored on each worker, for distributions with one up to sixteen workers. Furthermore, the number of Retweets processed by each worker is logged for the `6-100` cascade to examine the amount of work each worker has to do.

### 5.3.2. Reconstruction Performance

Independent of the setup and the reconstruction algorithm, the reconstruction of influence graphs consists of three phases, as illustrated in figure 13:

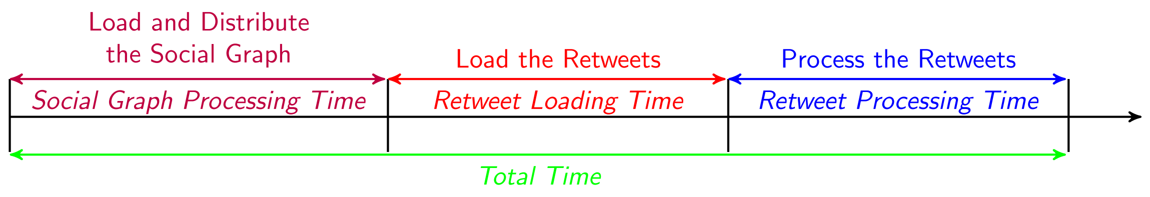


Figure 13: The three phases of reconstructing influence graphs: loading and distributing the social graph, loading the Retweets, and processing the Retweets. The names of the times spent in each phase are given in italics.

1. **Loading and Distributing the Social Graph:** worker 0 loads the social graph from its source file and distributes it to all workers according to the partitioning strategy. The time spent in this phase is called *Social Graph Processing Time*.

2. **Loading the Retweets:** worker 0 loads and buffers the Retweet data set. The time spent in this phase is called *Retweet Loading Time*.

3. **Processing the Retweets:** worker 0 emits the buffered Retweets. All workers execute the reconstruction algorithm; the reconstructed influence edges are written to files. This phase ends once all influence edges have been written. The time spent in this phase is called *Retweet Processing Time*.

The times spent in each phase are measured by worker 0. The performances of the first and second phase mostly depend on comparatively slow system file operations and parsing the input. Therefore, only the third phase is suitable to measure the performance of `CRGP`. Based on the Retweet Processing Time and in accordance with [15], the *Reconstruction Rate* (*RR*) is used to evaluate the system's throughput. Its definition is given in equation (8).

$$RR := \frac{Number\ of\ Retweets}{Retweet\ Processing\ Time} \tag{8}$$

The reconstruction experiments analyzed three critical configuration parameters: the algorithm (`GALE` and `LEAF`), the size of the Retweet batches, and the distribution of the reconstruction, consisting in turn of the two independent properties *number of processes* and *number of workers per process*. For a better understanding of their interplay, a multitude of variations on these parameters were evaluated.

The data sent around between workers is serialized by Timely Dataflow. Since this serializer takes the raw bytes from memory, its performance should be optimal and not have any measurable effects on the overall performance. Therefore, it is not analyzed.

# 6. Results

For the evaluation of `CRGP`, certain aspects of its implementation and its parameters will be analyzed in this chapter. Each experiment will measure the effects of changing a single variable to find its optimum setting. Finally, Lutz's [15] final scaling experiment will be reproduced[22] to compare his reconstruction system with the one presented in this thesis.

## 6.1. The Social Graph Data Structure

In this section, the four potential data structures given in listing 4 (chapter 5) for storing the social graph will be evaluated to find the most efficient one in terms of both memory requirements and performance.

Table 4 lists the expected size of the social graph (without dummy users) and its observed peak memory usage. The expected size has been calculated manually: both `HashMap` and `HashSet` have a fixed overhead of 40 B, additionally require 8 B per entry for the pointer to that entry, and allocate 11 % more memory than needed to store their contents in order to maintain a load factor of 90 % for optimal performance;[23] `Vec` has a fixed overhead of 24 B but does not allocate more memory than actually needed to store its contents; and the `User` struct requires 8 B.

| **Data Structure** | **Expected Size** | **Peak Usage** |
|---|---|---|
| `HashMap<User, HashSet<User>>` | 8.5 GiB | 8.9 GiB |
| `HashMap<User, Vec<User>>` | 2.8 GiB | 4.1 GiB |
| `(Vec<User>, Vec<HashSet<User>>)` | 8.4 GiB | 9.0 GiB |
| `(Vec<User>, Vec<Vec<User>>)` | 2.7 GiB | 4.2 GiB |

Table 4: The memory requirements of the potential social graph data structures.

These measurements show that using hash sets to store the friends for each user causes a massive amount of overhead on the memory requirements, since each of the 3,177,361 hash sets[24] maintains the aforementioned load factor of 90 %.

The times to insert the social graph into the data structures are given in table 5. The structures using hash maps are multiple orders of magnitude faster than the structures using two vectors in a tuple, the reason being firstly that the users are sorted for efficient look-ups during reconstruction, and secondly that the friends of a user are inserted at the same index as the user in their respective lists to maintain the association between both, and therefore contents of both lists have to be shifted when inserting new data.

[22] Albeit on different (and faster) machines, since the machines used by Lutz were occupied during the work on this thesis.

[23] This load factor is hard-coded into `Rust` at the time of this writing.

[24] One for each user in the social graph.

| Data Structure | Loading Time |
|---|---|
| `HashMap<User, HashSet<User>>` | 66.9 s |
| `HashMap<User, Vec<User>>` | 61.4 s |
| `(Vec<User>, Vec<HashSet<User>>)` | 8,698.4 s |
| `(Vec<User>, Vec<Vec<User>>)` | 13,622.1 s |

Table 5: The times to load the social graph into the respective data structures.

Since the data structures using two vectors in a tuple are hard to maintain, take an unfeasible amount of time to initialize with the entire social graph, and do not offer any advantages in terms of memory requirements, they are not further evaluated; only the performance of the `HashMap<User, HashSet<User>>` and the `HashMap<User, Vec<User>>` data structures will be analyzed.

These analyses evaluate the performance of the most important operations on the inner containers used for storing the friendships: iteration and containment tests. The `Vec<User>` is measured with both sorted and unsorted contents.

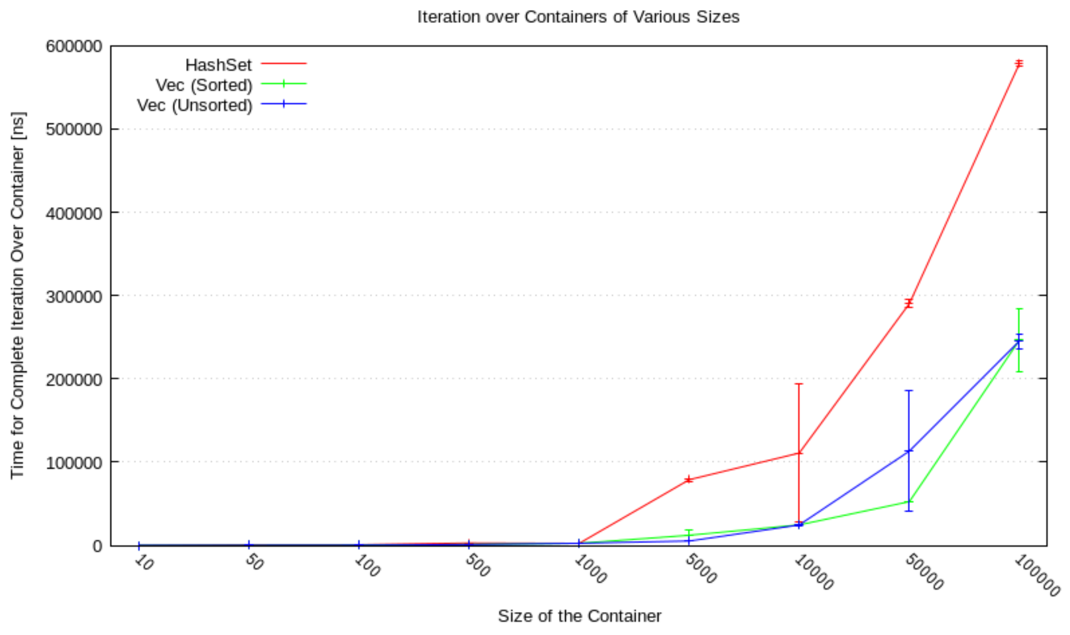


Figure 14: The performance of iterating over hash sets and vectors of various sizes in comparison.

The graph in figure 14 displays the performance of iterating over the hash sets and vectors for various sizes of these data structures. Up to 1,000 elements, the differences are negligible, but afterwards, the hash set's performance is significantly worse than the vectors' performances. The performance difference between the sorted and unsorted vectors at 50,000 elements cannot be explained, but is within the measurement inaccuracies.

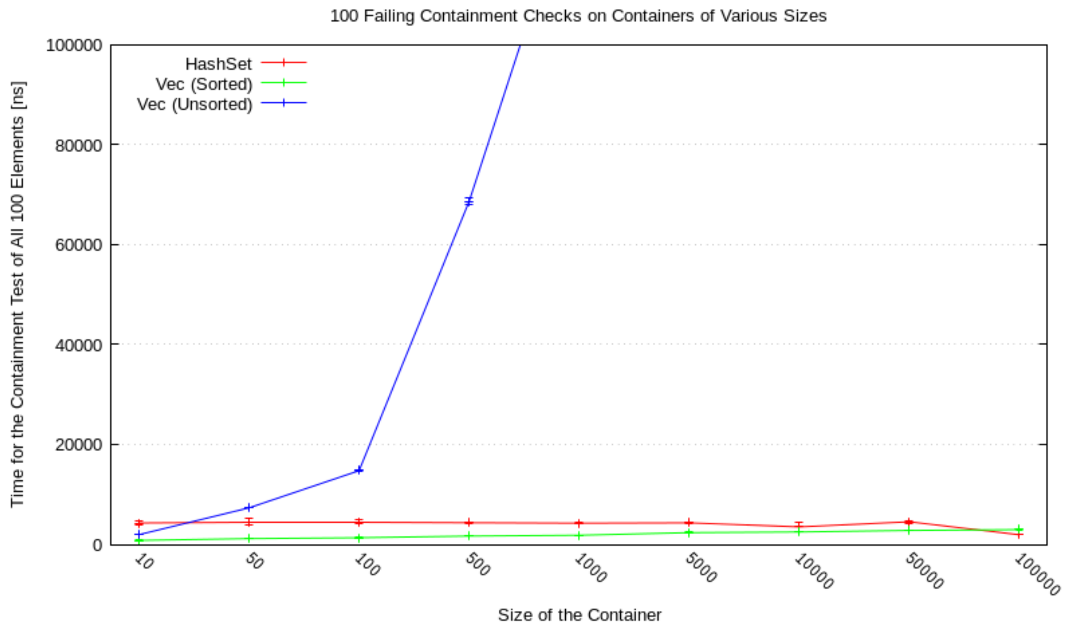


Figure 15: The performance of 100 failing containment tests on hash sets and vectors of various sizes in comparison. The graph has been cut on the $y$-axis at 100,000 ns to distinguish between the hash set's and the sorted vector's performances.

The performance of executing one hundred containment tests on the containers is displayed in figures 15 and 16; the former presents the results of one hundred failing containment tests, the latter the results of one hundred successful containment tests. For the sorted vector, the containment test used a binary search to find the requested element. In both graphs, the $y$-axis has been cut at 100,000 ns to be able to distinguish between the hash set's and the sorted vector's performances; the unsorted vector's results keep growing.

The performance of the containment tests on the hash set and the sorted vector are comparable, while the unsorted vector performs the worst in these experiments. The vectors yield better results for larger sizes during iteration, and their memory requirements are significantly lower than the hash set's requirements. Therefore, the `HashSet<User, Vec<User>>` data structure was chosen for the social graph.

The entire social graph, including dummy users, requires 14.8 GiB of main memory. As a conservative estimate, an `r4.4xlarge` instance on AWS EC2, which offers 122 GiB of main memory, could thus handle a social graph seven times larger than the one used in this thesis. With an average of 440 friends per user, as in this data set, this were 22,241,527 users and a total of 9,786,271,880 friendship edges.

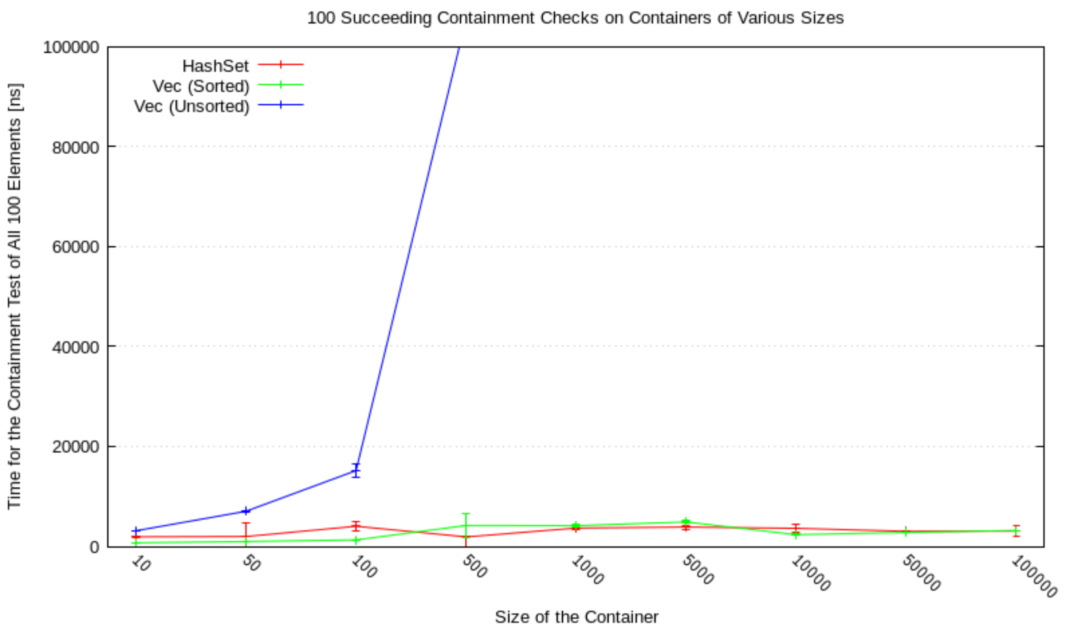


Figure 16: The performance of 100 succeeding containment tests on hash sets and vectors of various sizes in comparison. The graph has been cut on the $y$-axis at 100,000 ns to distinguish between the hash set's and the sorted vector's performances.

## 6.2. Distribution of the Social Graph

The social graph is distributed among all workers using the partitioning function (7). This strategy will be evaluated in this section in terms of the number of users and the number of friendships stored at each worker, for one up to sixteen workers.

The results are listed in the tables 6 and 7 for the users and the friendships, respectively. Ideally, for $W$ workers, each worker would store $u/W$ users and $f/W$ friendships, where $u$ is the number of users in the social graph and $f$ the number of edges.

Table 8 exemplarily lists the distribution of the `6-100` cascade across multiple workers with the social graph distributed as above, that is, for how many Retweets in the cascade, a worker stores the Retweeting user. This gives a measure of how the activity load is distributed across all workers. Note that while the cascade consists of 1,268,607 Retweets, only 1,250,880 Retweets are processed; this is due to the fact that the remaining Retweets were posted by users who are not in the social graph data set, and are therefore not stored on any worker.

Deviations from the optimum were to be expected, and are in tolerable ranges. The results thus show that the chosen partitioning strategy yields a fair distribution of the social graph across all workers. Furthermore, as an important side result, these experiments have shown that the sum of the users across all workers is the actual number of users in the social graph, that is, no users are lost or added when distributing the social graph; the same holds for the friendships and the Retweets.

| **Workers** | **Optimum** | **Minimum** | **Maximum** |
|---|---|---|---|
| 1 | 3,177,361 | 3,177,361 | |
| 2 | 1,588,681 | 1,565,479 | 1,611,882 |
| 3 | 1,059,120 | 1,058,232 | 1,060,087 |
| 4 | 794,340 | 782,452 | 806,410 |
| 5 | 635,472 | 631,912 | 646,296 |
| 6 | 529,560 | 509,002 | 550,040 |
| 7 | 453,909 | 452,972 | 455,285 |
| 8 | 397,170 | 390,898 | 403,555 |
| 9 | 353,040 | 351,915 | 354,009 |
| 10 | 317,736 | 312,110 | 333,856 |
| 11 | 288,851 | 287,971 | 289,515 |
| 12 | 264,780 | 254,352 | 275,755 |
| 13 | 244,412 | 243,538 | 245,199 |
| 14 | 226,954 | 222,941 | 231,351 |
| 15 | 211,824 | 210,021 | 215,837 |
| 16 | 198,585 | 195,240 | 201,990 |

Table 6: Distribution statistics of the users in the social graph. The optimum is the average number of users per worker, that is, 3,177,361 divided by the number of workers.

| **Workers** | **Optimum** | **Minimum** | **Maximum** |
|---|---|---|---|
| 1 | 1,397,187,866 | 1,397,187,866 | |
| 2 | 698,593,933 | 689,122,185 | 708,065,681 |
| 3 | 465,729,289 | 463,231,248 | 468,064,708 |
| 4 | 349,296,967 | 343,936,596 | 355,713,443 |
| 5 | 279,437,573 | 273,200,218 | 295,476,726 |
| 6 | 232,864,644 | 227,617,105 | 239,790,341 |
| 7 | 199,598,267 | 198,702,451 | 201,026,497 |
| 8 | 174,648,483 | 170,798,534 | 178,498,045 |
| 9 | 155,243,096 | 153,951,219 | 156,883,439 |
| 10 | 139,718,787 | 135,783,748 | 157,429,809 |
| 11 | 127,017,079 | 124,012,739 | 129,798,732 |
| 12 | 116,432,322 | 112,807,818 | 121,018,201 |
| 13 | 107,475,990 | 104,569,442 | 109,479,393 |
| 14 | 99,799,133 | 97,503,657 | 102,018,541 |
| 15 | 93,145,858 | 90,032,164 | 101,094,072 |
| 16 | 87,324,242 | 85,124,968 | 90,792,602 |

Table 7: Distribution statistics of the friendships in the social graph. The optimum is the average number of friendships per worker, that is, 1,397,187,866 divided by the number of workers.

| **Workers** | **Optimum** | **Minimum** | **Maximum** |
|---|---|---|---|
| 1 | 1,250,880 | 1,250,880 | |
| 2 | 625,440 | 615,306 | 635,574 |
| 3 | 416,960 | 415,338 | 417,919 |
| 4 | 312,720 | 306,812 | 318,268 |
| 5 | 250,176 | 246,720 | 254,300 |
| 6 | 208,480 | 200,097 | 216,961 |
| 7 | 178,697 | 177,802 | 179,830 |
| 8 | 156,360 | 152,685 | 159,401 |
| 9 | 138,987 | 137,903 | 139,725 |
| 10 | 125,088 | 121,778 | 132,522 |
| 11 | 113,716 | 112,586 | 114,391 |
| 12 | 104,240 | 99,251 | 108,742 |
| 13 | 96,222 | 95,348 | 97,168 |
| 14 | 89,349 | 87,206 | 91,837 |
| 15 | 83,392 | 81,268 | 85,320 |
| 16 | 78,180 | 75,614 | 79,978 |

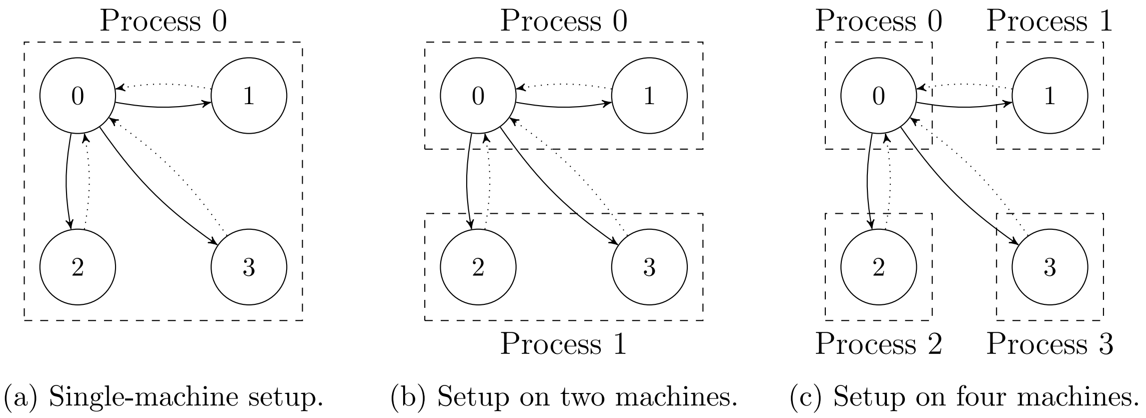
Table 8: Distribution of the `6-100` cascade across the workers. The optimum is the average number of Retweets per worker, that is, 1,250,880 divided by the number of workers.

## 6.3. Message Exchange Rate and Network Requirements

The goal of this experiment is to understand how fast Timely Dataflow can exchange messages between workers and processes, and which requirements this imposes on the network when distributing the computation across multiple machines.

The setup of these experiments consists in total of four workers. Worker 0 broadcasts 100 million dummy Retweets to the other workers which then send them back; no further processing occurs. This exchange protocol is similar to the `GALE` algorithm.

The experiment consists of three sub-experiments. In the first one, the four workers are all hosted on a single machine running a single process. The second experiment distributes the four workers across two machines, each running a process with two workers. In the last experiment, the four workers are distributed among four machines with a single process each. These setups are illustrated in figure 17. The machines are connected via a 10 GBit/s network. These experiments are conducted for batch sizes of 50, 500, 5,000, 50,000, and 500,000 to understand their effects on the performance.



(a) Single-machine setup. (b) Setup on two machines. (c) Setup on four machines.

Figure 17: The setup for measuring the message exchange rate of Timely Dataflow. Solid arrows represent a broadcast of a message, dotted ones a unicast.

The message exchange rate, given in Retweets per Second (*RT/s*), is graphed in figure 18. The setup running on a single machine performs better than the multi-machine setups. This result was to be expected, since the message exchange between multiple workers in a single process uses shared memory whereas multiple processes use the slower network. The performances of the multi-machine setups are similar; why the performance of the setup with two machines drops at a batch size of 50,000 cannot be explained, though. In general, larger batch sizes perform better, especially when distributing the computation. A concrete value that clearly outperforms the other batch sizes does, however, not stand out. Therefore, the reconstruction experiments will be conducted for multiple batch sizes.

During these experiments, the network utilization was measured to estimate the network requirements and understand when it could become a bottleneck. The results are

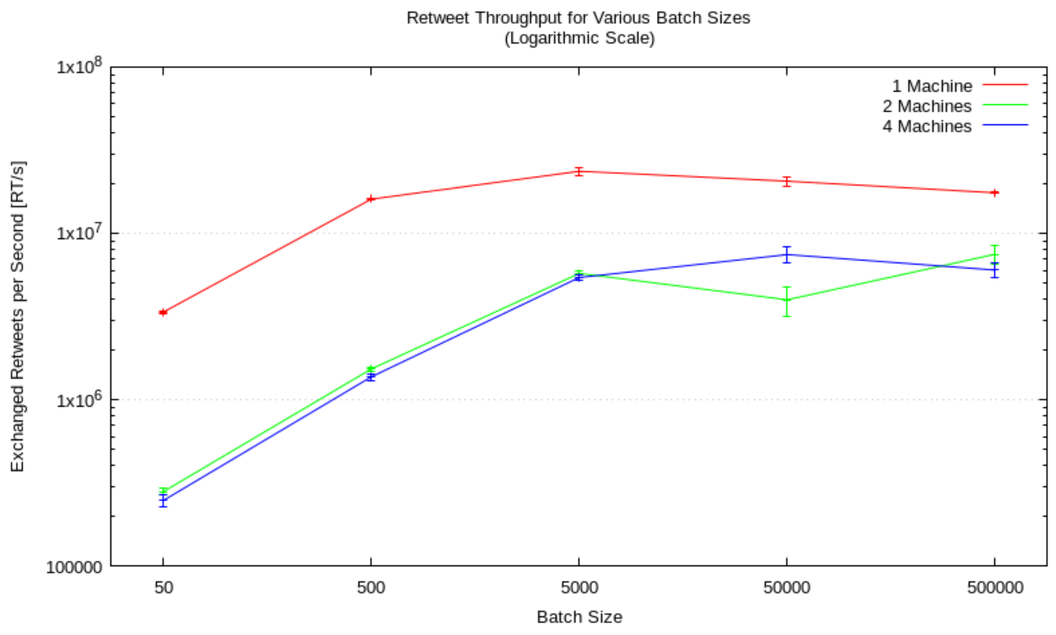


Figure 18: Comparison of the message exchange rate over various batch sizes for the different setups.

visualized in figure 19. In all cases, the 10 GBit/s network suffices and does not limit the performance; a 1 GBit/s connection (128 MiB/s) would, however, be insufficient. The setup running on four machines transfers more data than the two-machine setup, due to the broadcast to three machines instead of one. This suggests that for even more machines, the 10 GBit/s network could become a bottleneck.

Distributing the workers across multiple processes should not change the amount of data transferred between the workers, that is, the amount of data sent by worker 0 to each worker should be independent of the number of processes, as long as the total number of workers does not change. For workers within the same processes, this data cannot be measured, but for the setups with two and four machines, the corresponding data is graphed in figure 20. It visualizes for the evaluated batch sizes the average amount of data sent from worker 0 to workers 2 and 3, and for four machines, worker 1 as well. With the exception of the batch size of 50, the results are almost constant when going from two machines to four. The amount of data increases, however, by 200 MiB for batch size 50, supposedly due to the increased amount of synchronization messages between the four machines.

## 6.4. Reconstruction

While the previous experiments evaluated the underlying mechanisms, the following measurements will analyze the actual reconstruction performance, comparing both algorithms.

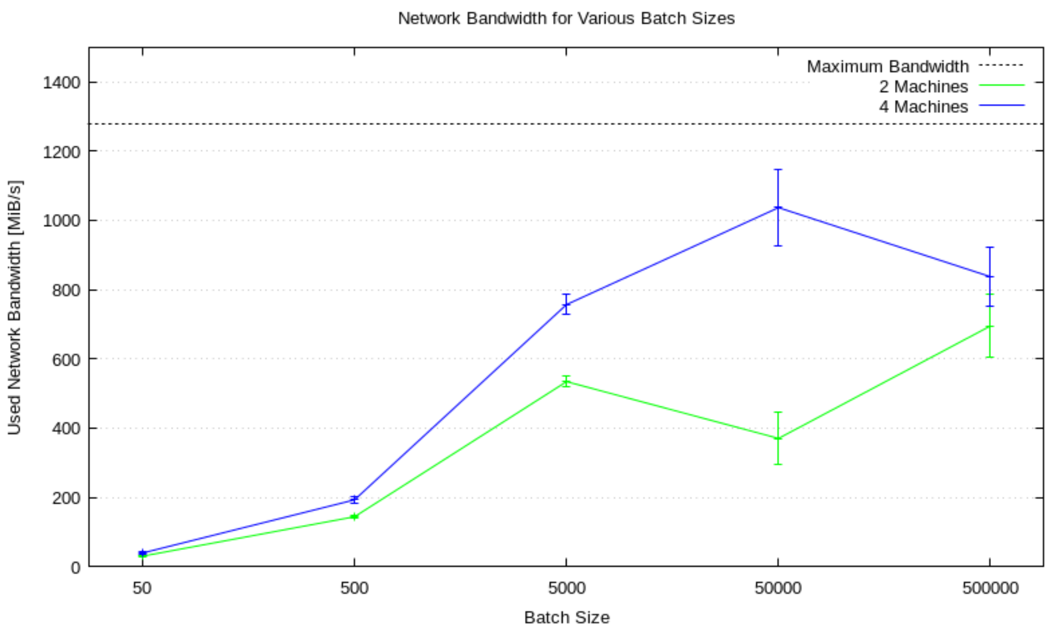


Figure 19: Comparison of the used network bandwidth over various batch sizes for different setups. The maximum bandwidth available on the network is 10 GBit/s = 1,280 MiB/s.

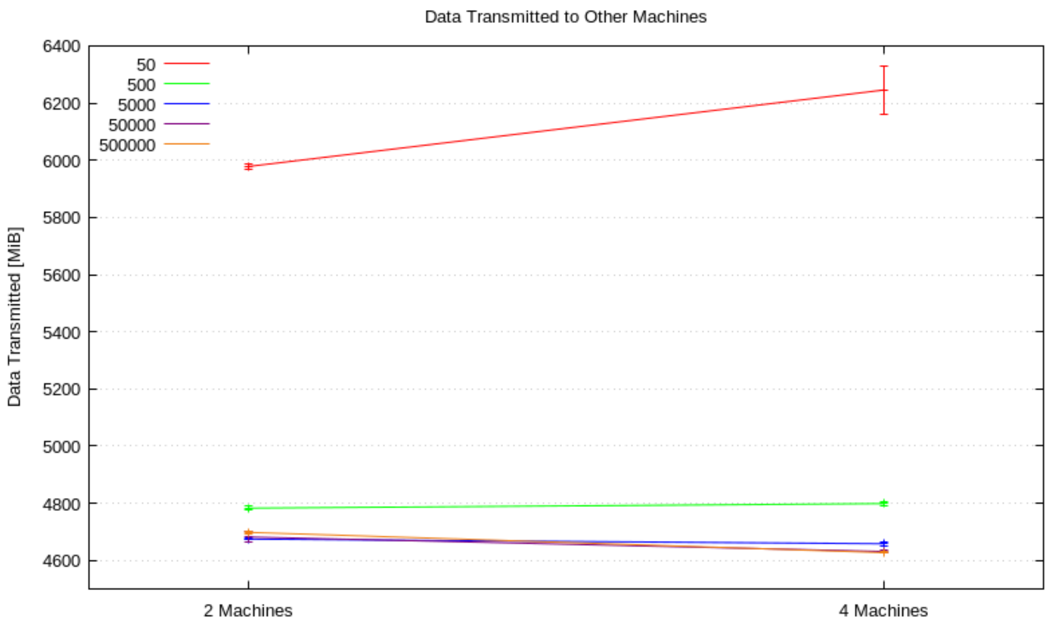


Figure 20: Per batch size, the amount of data sent by worker 0 to each worker not on the same machine.

### 6.4.1. GALE's Adaptive Iteration

The `GALE` algorithm uses an adaptive combination of the `Prefix Iteration` and the `User Iteration` algorithms: if the prefix is shorter than the user's list of friends, `Prefix Iteration` is performed, otherwise `User Iteration`. In theory, `User Iteration` will perform better for large cascades with more Retweets than the average number of friends, and `Prefix Iteration` correspondingly for small cascades.

The following measurements will evaluate if the adaptive approach can successfully combine the advantages of both algorithms, that is, if its performance is similar to the performance of the `User Iteration` for a large cascade, and similar to the performance of the `Prefix Iteration` for small cascades.

For these experiments, the `GALE` algorithm is run in three variations: using the adaptive approach as described before, iterating solely over the retweeting user's friends, and iterating solely over the prefix. The performance is measured for one up to four workers in a single process on `Corse`. The `medium` data set is used for the large cascade measurements, the `6-00` data set for the small cascade measurements. The results are graphed in figures 21 and 22, respectively.

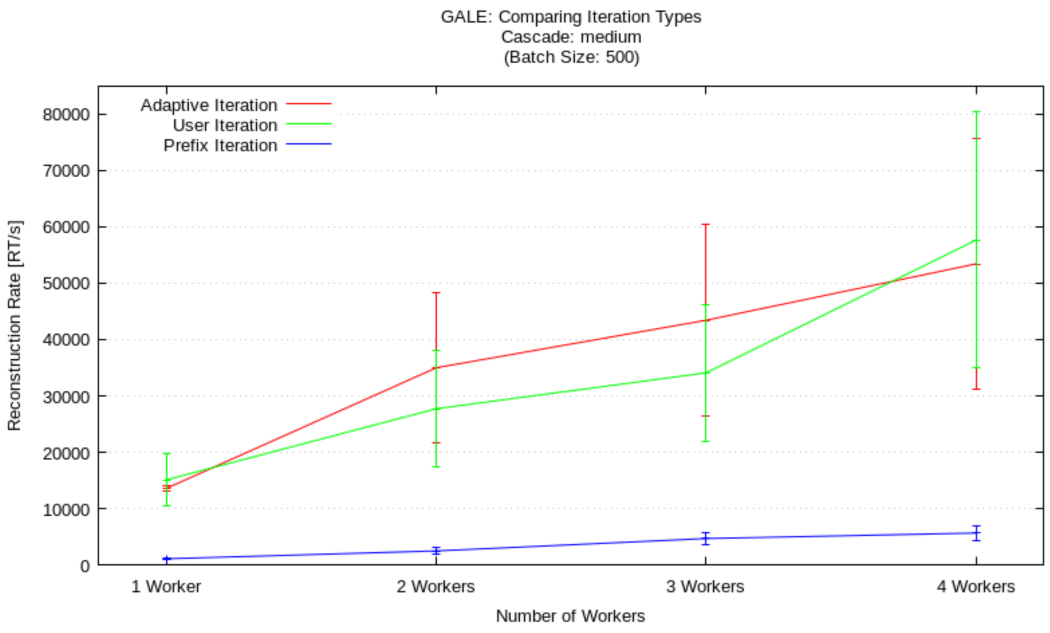


Figure 21: Comparison of `GALE`'s iteration types on the `medium` cascade.

As expected, `User Iteration` performs better than `Prefix Iteration` on the large cascade, but worse on the small cascades. In both cases, the adaptive approach reaches a similar performance to the respective naïve iteration; it performs even slightly better, although this is within the measurement inaccuracies. Therefore, the adaptive iteration combines the advantages of the `Prefix Iteration` and the `User Iteration` to perform well on both small and large cascades, and will be used in `GALE`.

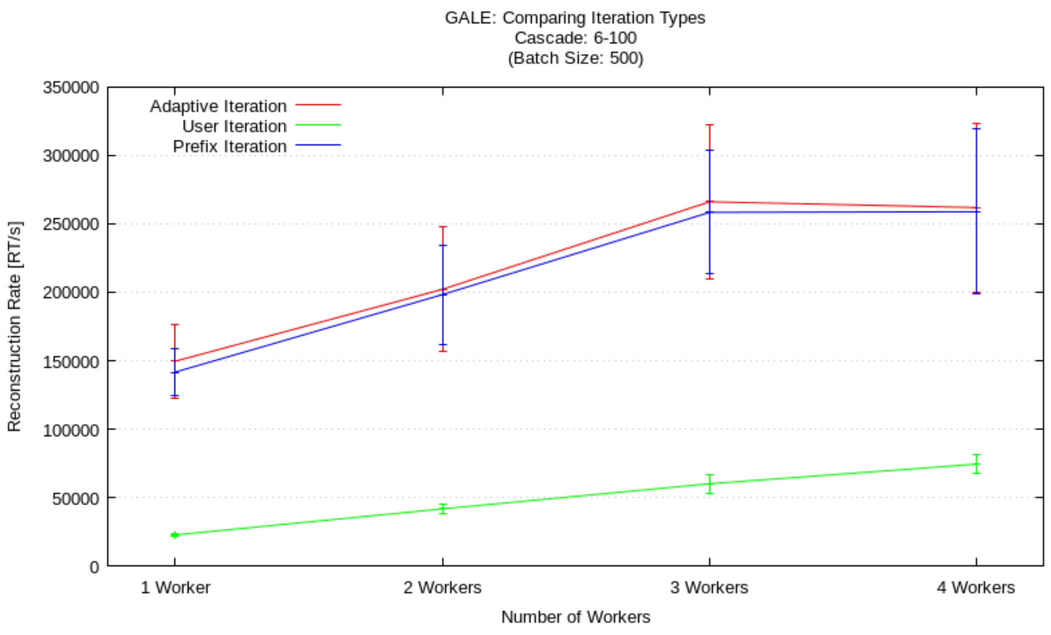


Figure 22: Comparison of `GALE`'s iteration types on the `6-100` cascade.

### 6.4.2. Twitter Timestamps in Timely Dataflow

Retweets have an attached timestamp, stating their time of creation. Since Timely Dataflow associates each data element in its computation with a timestamp as well, it seems obvious to use the Retweets' already existing timestamps for the Timely Dataflow computation.

However, as stated in chapter 3.2, the batches within the computation are defined by the timestamps of the data, that is, each batch contains only data with the same timestamp. Therefore, using the Retweets' timestamps would result in variably-sized and usually small batches, as shown in the following examples:

- The cascade `smaller` consists of 3,500 Retweets of which 2,510 have a unique timestamp. Four timestamps occur a maximum amount of seven times in the data set.
- The cascade `medium` consists of 14,847 Retweets. Of these, 9,709 Retweets have a unique timestamp, and another 1,090 timestamps exist only twice. The maximum amount a timestamp occurs within this data set is sixteen.
- The cascade `bigger4` consists of more than 3.6 million Retweets. 578,093 of these have a unique timestamp, and 245,737 timestamps exist only twice. Two timestamps occur fifty-eight times.

The effects of small batch sizes on the reconstruction rate will by studied in the following experiments. Instead of using the timestamps of the Retweets for Timely Dataflow, fixed-sized batches of sizes 1, 10, 50, 100, and 500 will be evaluated. The cascade `smaller` will be reconstructed both by the `GALE` and `LEAF` algorithms on `Guatteri`, using a single worker, two workers in one process, and two processes with each one worker. The graph in figure 23 visualizes the results for the `LEAF` algorithm, the one in figure 24 the measurements for `GALE`.

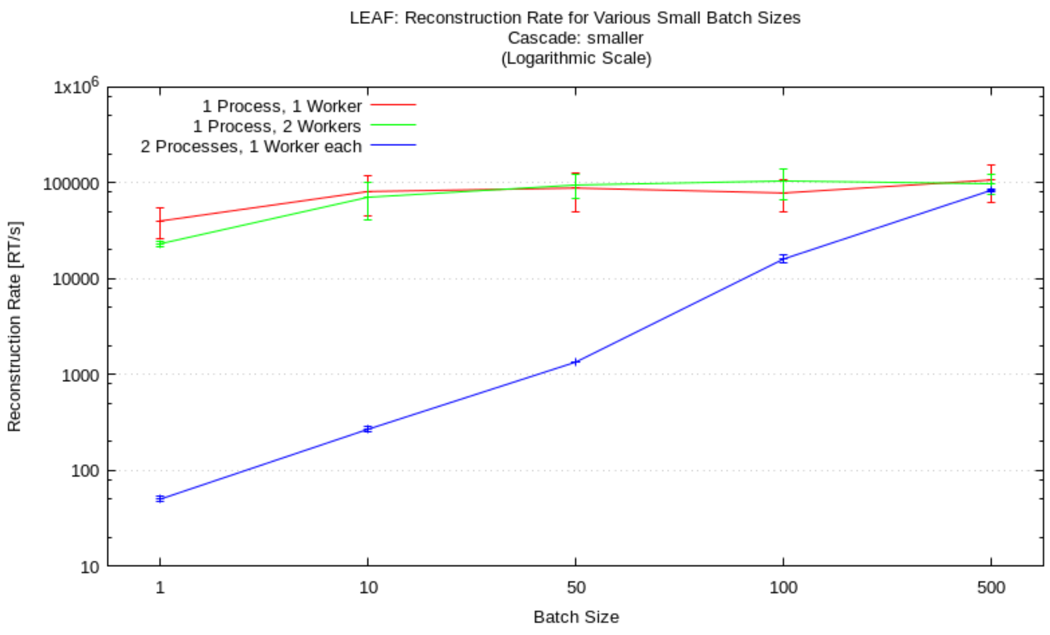


Figure 23: The effect of small batch sizes on the `LEAF` algorithm reconstructing the `smaller` cascade.

Both algorithms show the same behavior: the single-process reconstructions perform comparably for batch sizes greater than one; for batch size one, the two-worker reconstructions are only half as fast as the respective single-worker reconstructions. The reconstruction on two processes performs terribly for batch sizes smaller than five hundred: with only fifty Retweets per second, `LEAF` takes seventy seconds to reconstruct the influence graph for the 3,500 Retweets; `GALE` is only slightly faster with 64.5 RT/s, or fifty-four seconds. Only at a batch size of five hundred, the dual-process reconstruction catches up to the single-process results.

The reason for this behavior is the need to synchronize the operators after every batch: the smaller the batch size, the more often the operators will synchronize and send data packages. In a single process, the synchronization messages and the actual data are passed around in shared (and fast) memory, but for multiple processes, they are sent over the network, requiring them to be serialized and packed into the respective network packages. The synchronization pauses the entire computation—smaller batch sizes thus result in more frequent interruptions.

For this reason, the Retweets' timestamps are not used for the Timely Dataflow computation. Instead, arbitrary timestamps will be introduced to increase the size of the batches for better performance.

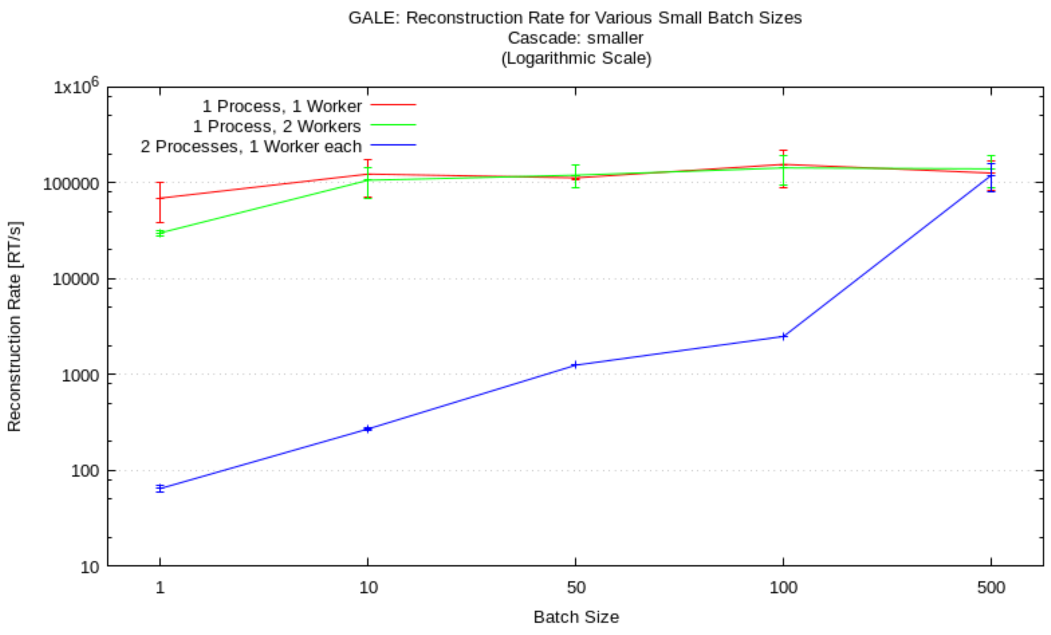


Figure 24: The effect of small batch sizes on the `GALE` algorithm reconstructing the `smaller` cascade.

### 6.4.3. Large Batch Sizes and the Correctness of Results

The last experiment has shown that small batches negatively impact the performance, but it has not yet determined how large the batches should be for optimal efficiency. The following experiments will thus measure the reconstruction rate for batch sizes from 50 through 500,000. Furthermore, in section 3.2, it was stated that multiple workers affect the order of the output data within each batch; with larger batches, this effect should grow stronger. The second question answered in these experiments therefore is if different batch sizes change the reconstructed influence graphs.

These measurements have been run on an `r4.4xlarge` instance, reconstructing the `bigger4` data set; the evaluated batch sizes are 50, 500, 5,000, 50,000, and 500,000. The reconstructed influence graphs have been compared using the script in listing 10, appendix B.

Figures 25 and 26 show the performance of `GALE` and `LEAF`, respectively, when run in a single process with varying numbers $W$ of workers. For `GALE`, the results are rather precise: increasing the size of the batches up to 50,000 increases the performance, especially when also adding more workers to the computation. Batch sizes of 50,000 and 500,000 result in the same performance; an—due to the lack of time—open question is if this is due to some kind of upper bound or if batch sizes in between these two values behave differently. The results for `LEAF` are not as distinct: the performance with a batch size of 500,000 is worse than with a batch size of 50, and now the batch sizes 500 and 5,000 perform equally and show the best results. These findings cannot be explained.

The reconstructed influence graphs are the same for all batch sizes and independent

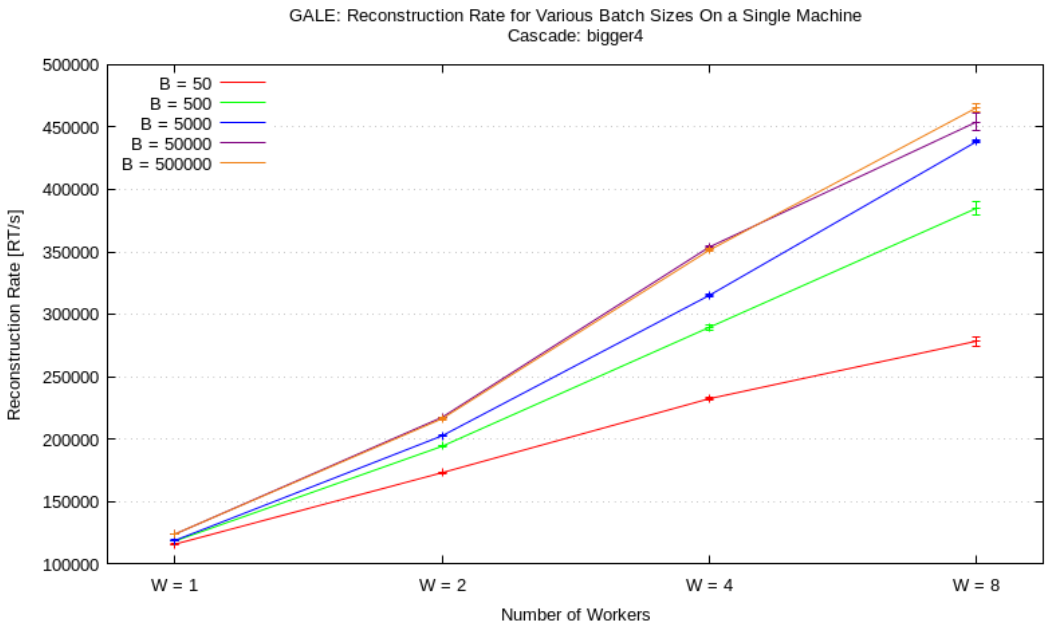


Figure 25: The reconstruction rate of `GALE` on a single machine for multiple batch sizes.

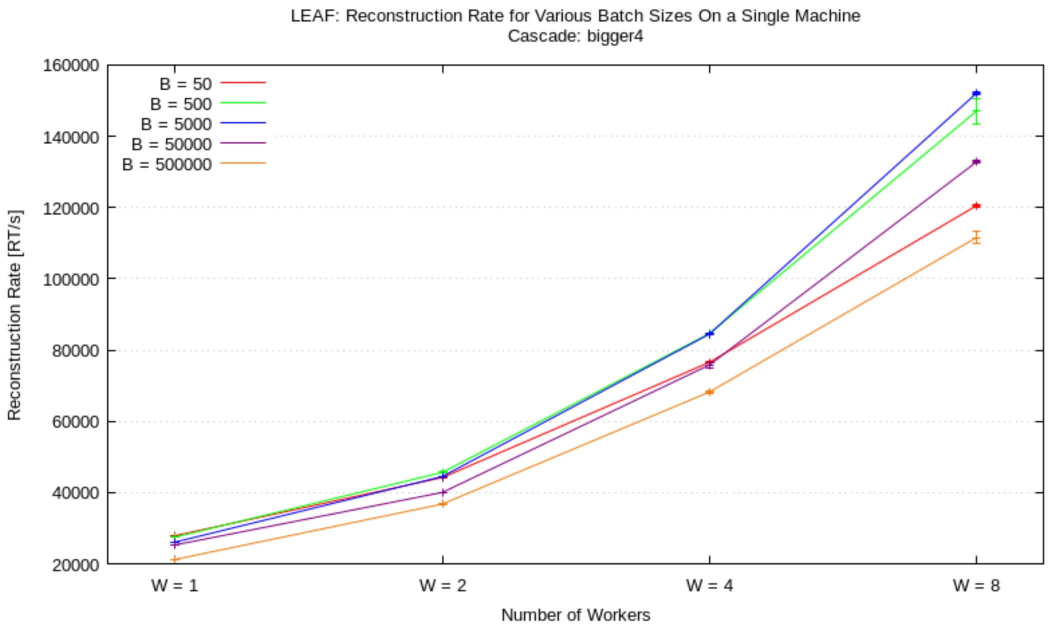


Figure 26: The reconstruction rate of `LEAF` on a single machine for multiple batch sizes.

of the number of workers. The part of the reconstruction where the order of the data is most crucial is the prefix: if a Retweet $r$ is processed before its actual predecessors, this can lead to two problems. On the one hand, the prefix will not contain any information on these predecessors and thus, influence edges for the Retweet $r$ will be missed. On the other hand, the author of $r$ will be added to the prefix too early, and once its predecessors are processed, these can produce false positives if they find $r$'s author to be an influencer.

The second case is eliminated by the fact that not only the retweeting user is added to the prefix, but also the timestamp when they posted their Retweet. This timestamp is then checked by all Retweets, and only if it is smaller than the Retweet's timestamp, the influence edge will be produced. This is the same for both algorithms.

For `GALE`, the first case cannot occur due to the fact that the order of the Retweets is preserved until the influence edges have been produced: all Retweets are broadcast to all workers at the beginning of their batch's computation. This means that the partition of a batch that each worker receives is the entire batch, and, as explained in section 4.1, within each partition of a batch the order is preserved. Each worker thus processes the entire batch in the correct order, first adding a Retweet's author to the prefix and then—if the worker is storing the author's friendships—producing their influence edges.

`LEAF` creates the local prefixes in the correct order, as well, since the partitions of a batch of Retweets at each worker still have the same order as the entire batch. This also means that each worker produces the possible influence edges in the correct order. However, the possible influence edges are then exchanged between workers to filter them. If such a possible influence edge is processed by the `filter` operator on a worker before the influencer's Retweet has been processed and the influencer has been added to the prefix, this possible influence edge will be discarded. This behavior has not been observed, though, but if it occurs in the future, it can be solved by letting the `find_possible_influences` operator hold back the possible edges until the entire batch of Retweets has been distributed. As this is presumed to decrease the performance, this has not been done yet.

### 6.4.4. Effects of the Social Graph's Size on the Performance

The social graph consists of 341,698,525 explicitly given friendship edges; including the implicitly given friendships in the meta data of each user, its size grows by a factor of four to 1,397,187,866 edges. This provides a good opportunity to measure the effects of the social graph's on the performance of the reconstruction algorithms.

For these experiments, the `bigger4` cascade has been reconstructed on an `r4.2xlarge` instance using the *unpadded* social graph (that is, excluding the implicitly given friendships), and on an `r4.4xlarge` instance using the *padded* graph. These instance types offer the same underlying hardware and differ only in the number of CPU cores and the amount of main memory, but since the limits of neither were reached during the reconstructions, the results are comparable. The reconstructions were run with batch sizes of 50, 500, 5,000, 50,000, and 500,000, but since in these experiments, the results for three larger batch sizes turned out to be equal to each other, the results for batch sizes 50,000 and 500,000 will be omitted from the graphs to aid legibility.

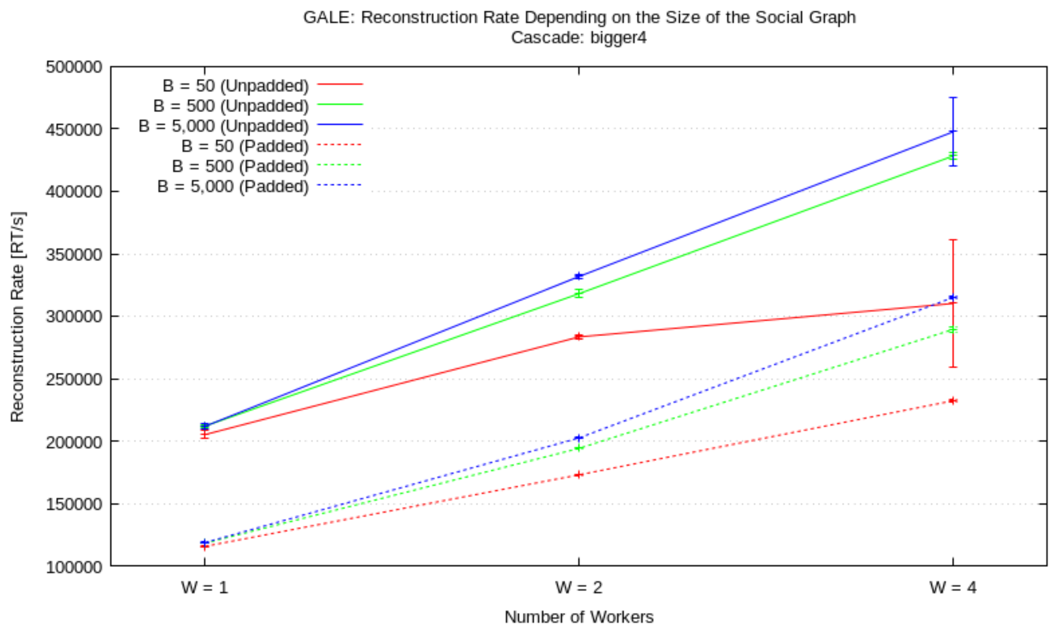


Figure 27: The reconstruction rate of `GALE` depending on the social graph's size.

The results for `GALE` and `LEAF` are graphed in figures 27 and 28, respectively. In all cases, the performance on the padded social graph is significantly slower than on the unpadded social graph. The ratios of the unpadded performance to the padded performance is visualized in figure 29: although the social graph has grown by a factor of four, the performance on a single worker has decreased by a factor of only 2.8 for `LEAF`, and 1.8 for `GALE`, almost independent of the batch size. Adding more workers, further decreases these ratios. It is, however, doubtful that the ratios will reach 1 with enough workers.

`LEAF`'s ratio is significantly larger than `GALE`'s. This stems from the fact that `LEAF` always iterates over the full list of friends for each retweeting user, while `GALE` in most cases iterates over the prefix, since the `bigger4` data set consists of cascades that are on average smaller (thirty-seven Retweets per cascade) than the number of friends (108 per user in the unpadded social graph and 440 per user in the padded graph). `GALE`'s lookup in the users' friend lists outweighs `LEAF`'s sending a possible influence edge for each friend.

## 6.5. Scaling

The final experiments will evaluate how `CRGP` scales when distributing the computation. In section 6.4.3, the measurements already showed that increasing the number of workers in a single process resulted in a better performance (figures 25 and 26). Therefore, the following experiments analyze the performance of reconstructing the `bigger4` data

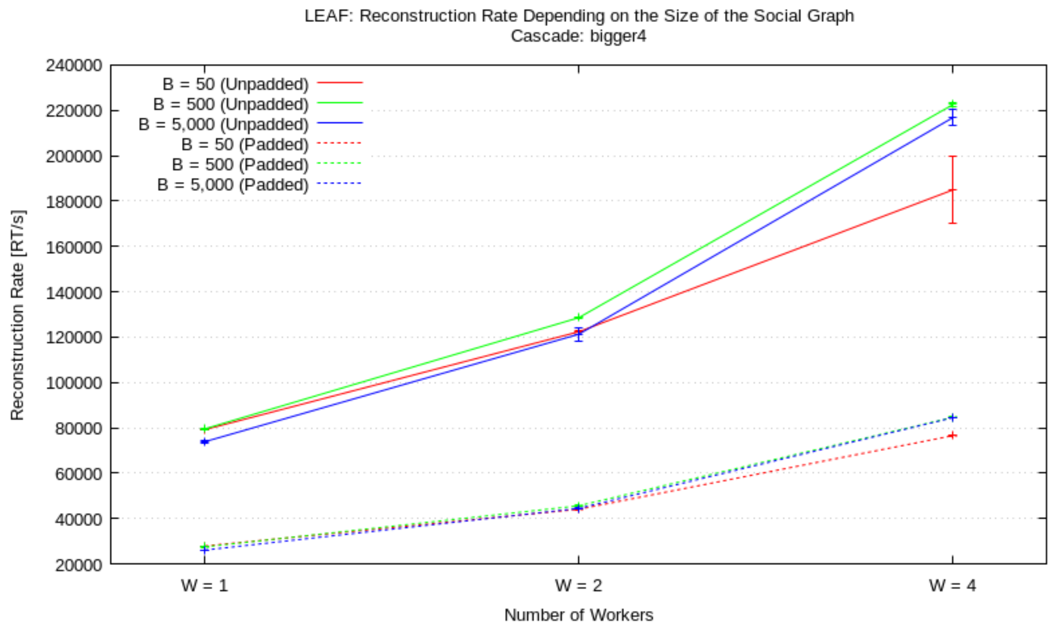


Figure 28: The reconstruction rate of `LEAF` depending on the social graph's size.

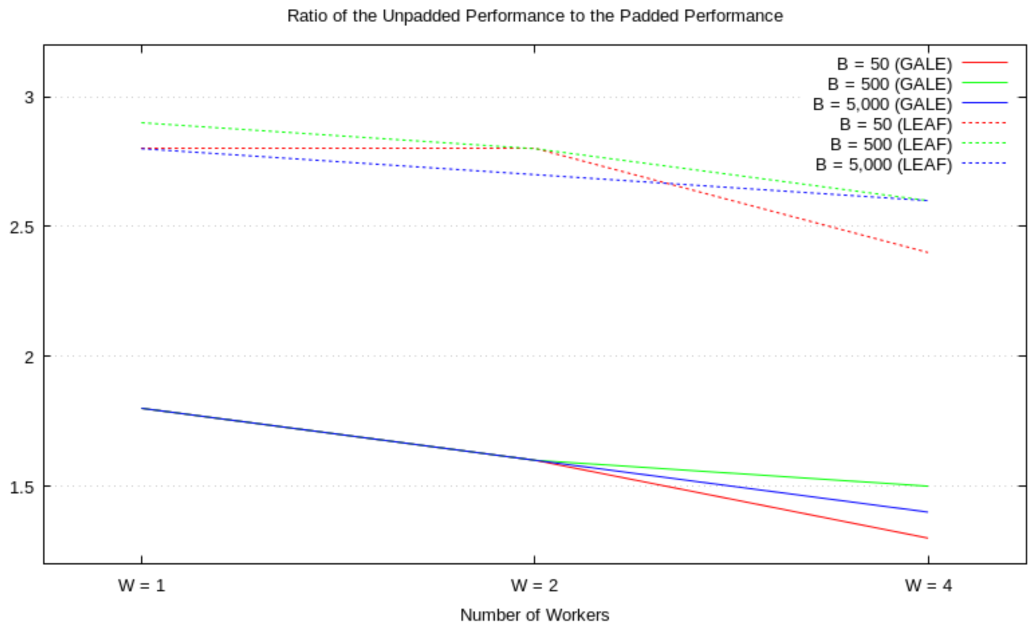


Figure 29: Ratios of the unpadded performance to the padded performance.

set when increasing the number of processes involved in the computation. Since the measurements in section 6.4.3 did not yield an obvious choice for the batch size, the three most promising batch sizes of 5,000, 50,000, and 500,000 will be evaluated.

### 6.5.1. Constant Number of Total Workers

The first experiments measure the performance when keeping the total number of workers constant and changing the number of processes correspondingly. These experiments were conducted on `r4.4xlarge` instances, with a total of eight workers.

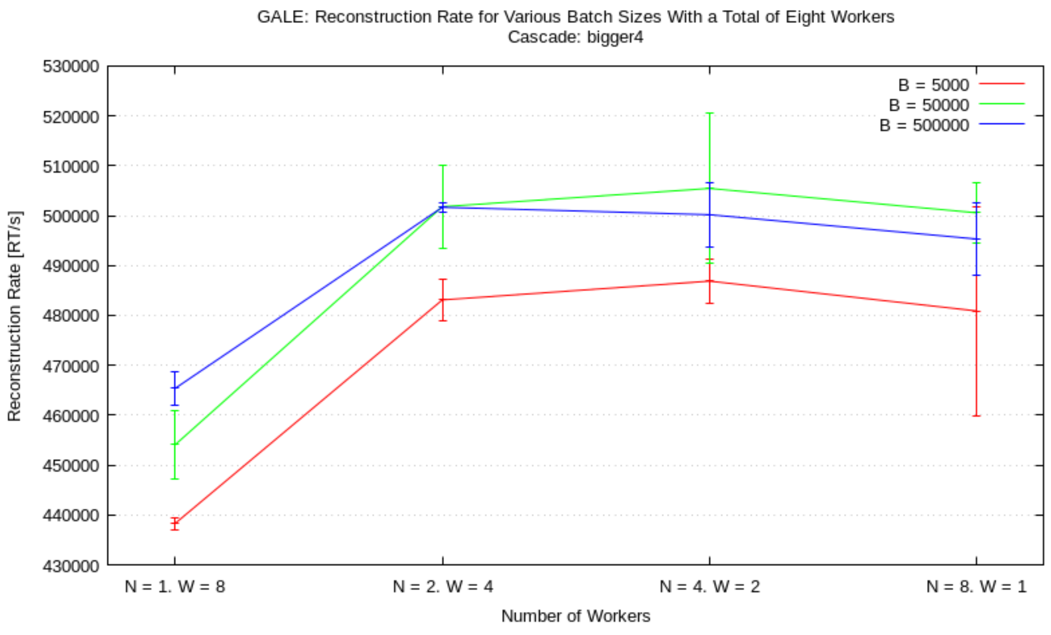


Figure 30: The reconstruction rate of `GALE` for multiple batch sizes $B$ and distributions with a total of eight workers. $N$ is the number of processes, $W$ the number of workers per process.

Scaling `GALE` from one to two processes results in a steep increase in performance, but scaling to four processes only marginally increases the performance. With eight processes, the performance begins to decrease again, presumably due to network saturation. The batch size of 5,000 performs significantly worse than the sizes of 50,000 and 500,000. These—as in section 6.4.3—perform almost equally within the measurement accuracies, but with a tendency towards 50,000 for multiple processes.

`LEAF`, again, shows more peculiar results. For all batch sizes, the performance drops when going from a single process to two processes, but then increases again for more processes, although it does not reach the single-process results again. The reason for this behavior presumably is the large amount of messages shared between workers. As before for `LEAF`, the batch size of 5,000 shows the best results.

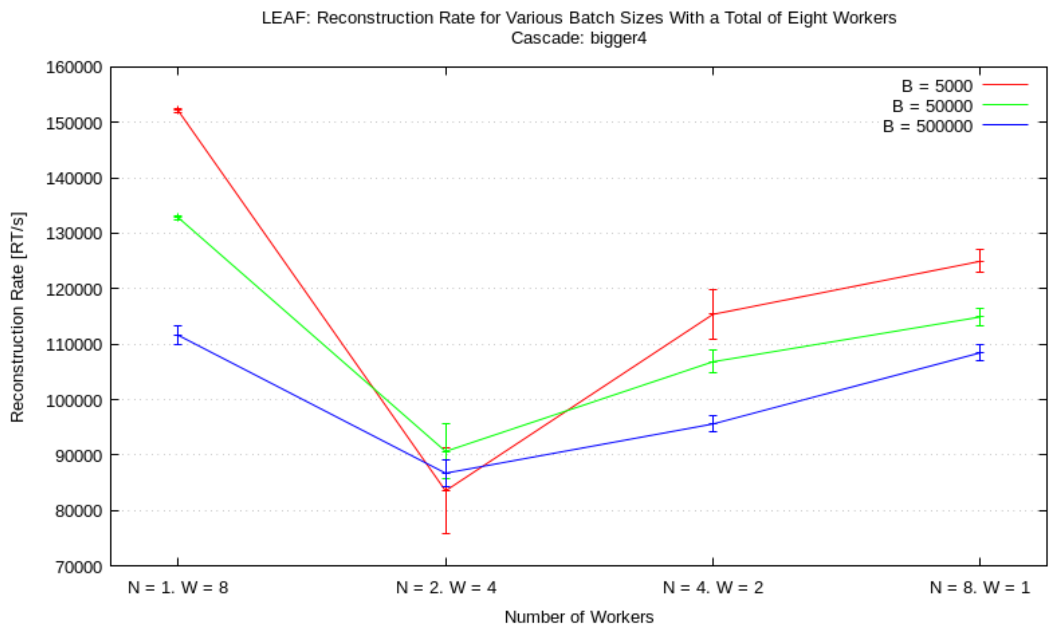


Figure 31: The reconstruction rate of `LEAF` for multiple batch sizes $B$ and distributions with a total of eight workers. $N$ is the number of processes, $W$ the number of workers per process.

### 6.5.2. Constant Number of Processes on a Single Machine

The next experiments measure the performance of running a fixed number of processes on a single machine and increasing the number of workers. These experiments are conducted on an `r4.16xlarge` instance with sixty-four CPU cores. The number of processes was fixed at eight, and the number of workers per process increased from one to eight, and thus in total sixty-four. The results are graphed in figures 32 and 33.

In accordance with the measurements from section 6.5.1, the batch size of 5,000 performs worse than the other two sizes when using `GALE`. This time, however, the batch size of 500,000 performs better than the size of 50,000. The best performance is reached when using two and four workers, with almost identical reconstruction rates of 562,066.5 RT/s and 565,110.0 RT/s, respectively. From there on, the performance drops, presumably because of too many synchronizing workers. For `LEAF`, the results very similar, although the curves are not as steep, and—again—batches of size 5,000 perform better than larger batches.

### 6.5.3. Constant Number of Processes on Multiple Machines

The last experiments repeat the setup of the previous experiment but distribute each process to a single `r4.4xlarge` instance. Since each instance provides sixteen CPU cores, the number of workers per process will be increased up to that amount. The results are graphed in figures 34 and 35.

For `GALE`, the results are similar to the single-machine measurements, although now,

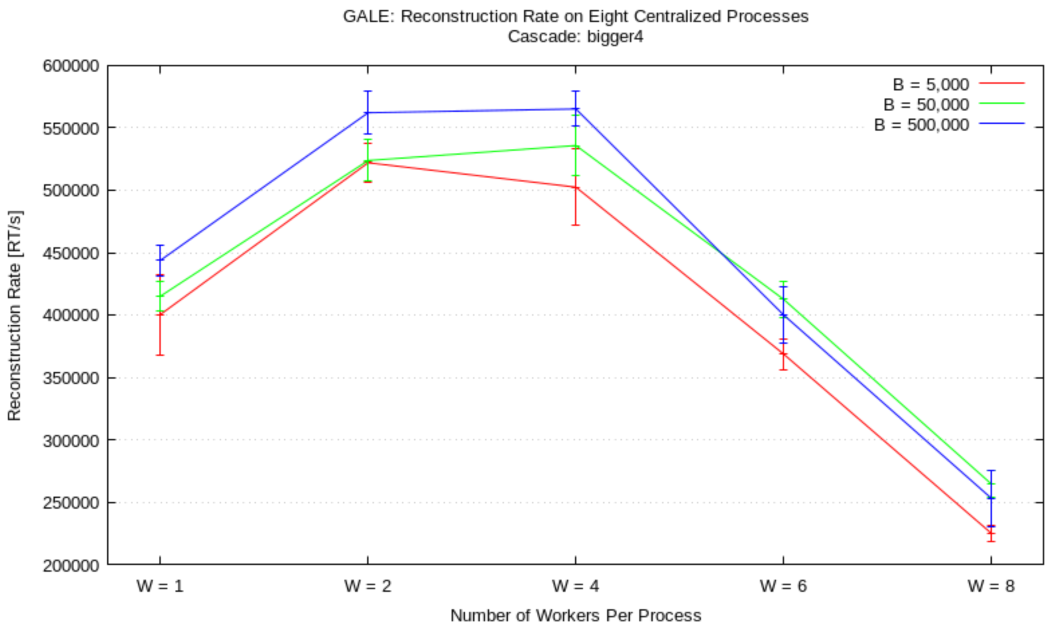


Figure 32: Scaling `GALE` over eight processes on a single machine for multiple batch sizes $B$ and number of workers per process $W$.

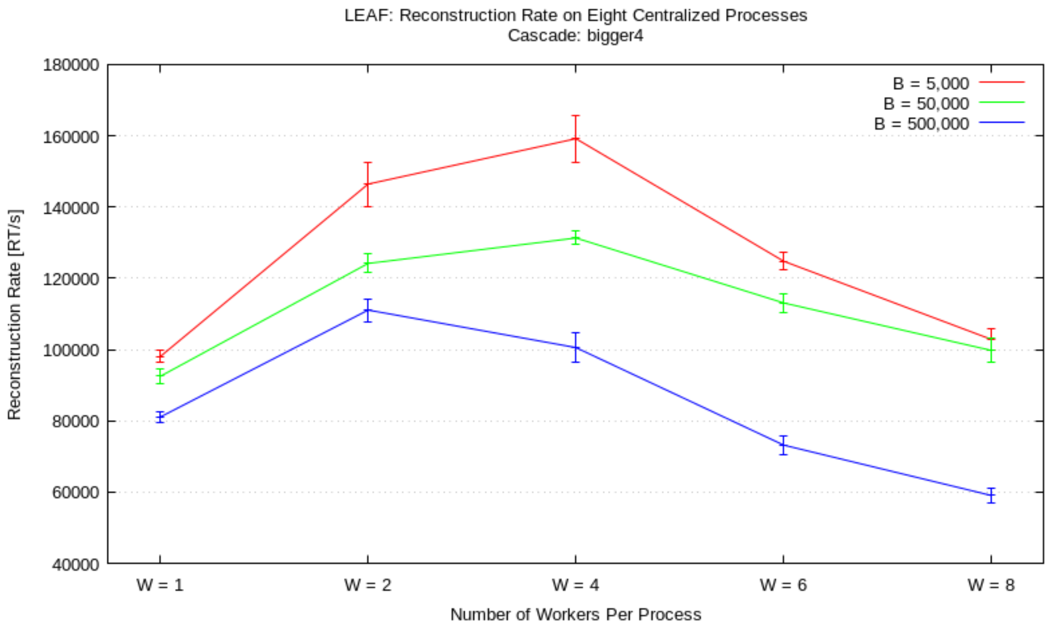


Figure 33: Scaling `LEAF` over eight processes on a single machine for multiple batch sizes $B$ and number of workers per process $W$.

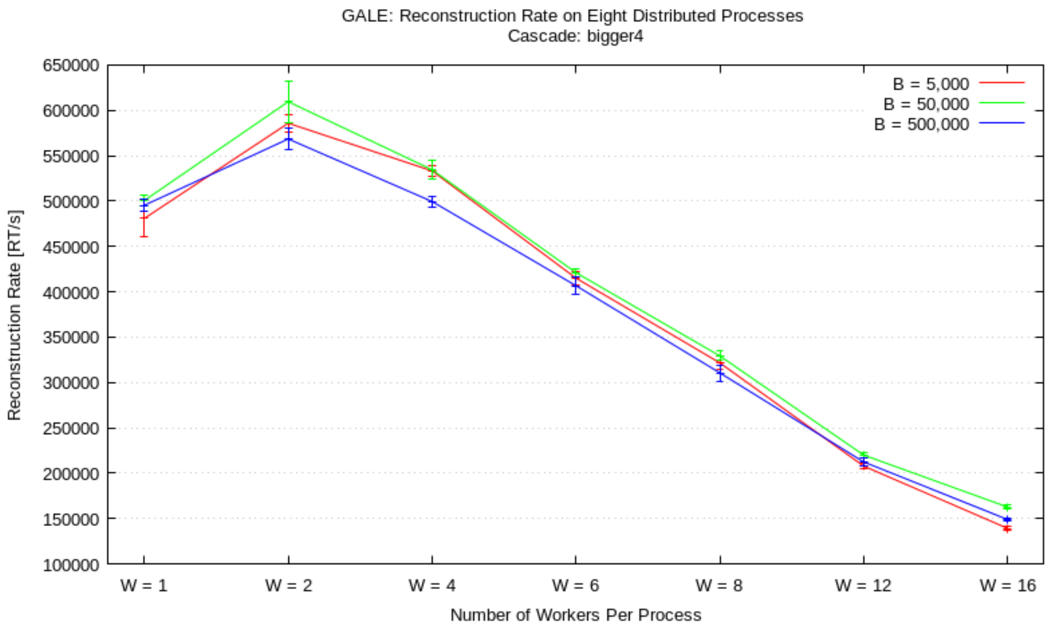


Figure 34: Scaling GALE over eight processes on eight machines for multiple batch sizes $B$ and number of workers per process $W$.

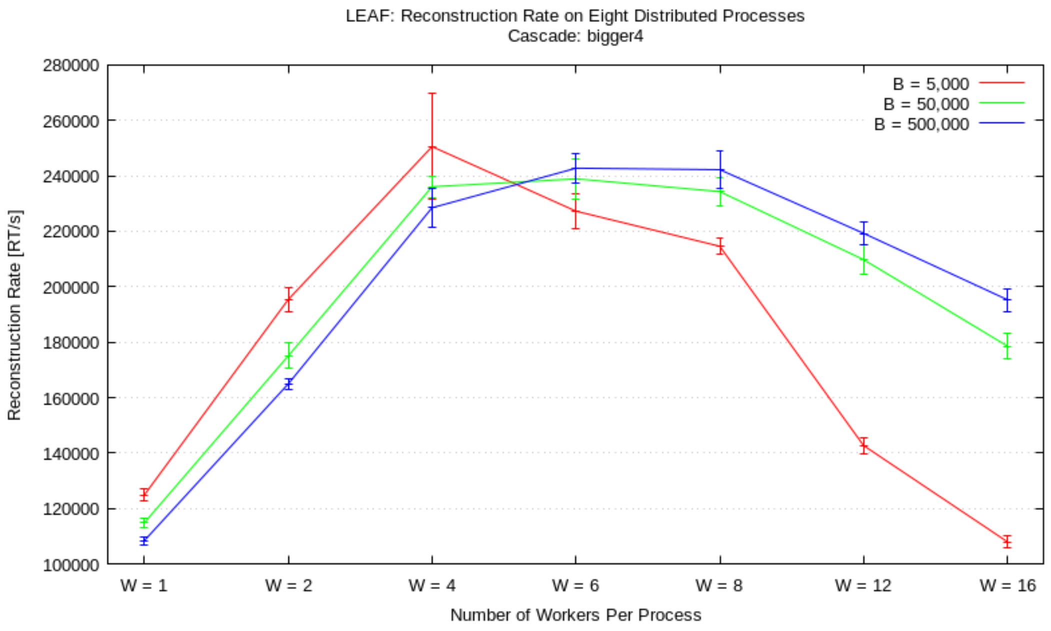


Figure 35: Scaling LEAF over eight processes on eight machines for multiple batch sizes $B$ and number of workers per process $W$.

the batch size of 50,000 performs better than batches of size 500,000, and four workers per process do not reach the same performance as two workers: these reach a top reconstruction rate of 609,474.0 RT/s.

`LEAF`'s results are not as similar to the previous ones: the batch size of 50,000 performs better than the other two batch sizes up to workers where it reaches the overall top performance of 250,542.0 RT/s, but with more workers, its performance falls behind the performances of the other two batch sizes. With six workers, the batch size of 500,000 also surpasses the batch size of 50,000, and reaches the second-best performances 242,748.9 RT/s and 242,228.1 RT/s for six and eight workers, respectively.

Overall, both algorithms perform better when the processes are distributed over multiple machines as opposed to being run on a single machine. The reason for this behavior is unknown.

# 7. Conclusion

Due to the amount of data, the reconstruction of influence graphs in social networks is a non-trivial problem that can only be successfully solved by distributing the computation across multiple computers. The Research Group on Web Science have implemented such a system that can process a data set of 3.6 million Retweets in under twenty seconds.

For a full evaluation of the existing system, this thesis implemented a different approach, using a computational model called Differential Dataflow. This model is a data-parallel framework with the unique feature of combining incremental input updates with iterative stream processing. As part of this new implementation, the two reconstruction algorithms `LEAF` and `GALE` were introduced. The results show that `GALE` is able to reconstruct the aforementioned data set within just over six seconds.

## 7.1. Discussion

The experiments have shown that `GALE` reaches significantly higher reconstruction rates than `LEAF`, which exchanges more data between workers. Furthermore, `GALE` is guaranteed to reconstruct the correct influence graph, while this is not the case for `LEAF`. Therefore, `GALE` is the preferred algorithm.

The effect of the batch sizes on the performance is not yet fully understood; batches of size 50,000 and 500,000 perform almost identical for `GALE`, although batches of size 50,000 tend to be a bit better, especially when distributing the computation over multiple machines.

Scaling the computation can occur in two directions: the number of workers per process, and the number of processes. When distributing the computation across multiple processes, the performance is better when running each process on its own machine. With more than a single worker per process, this yields the best results. However, scaling too far in either direction results in a decreased performance.

With a reconstruction rate of 609,474.0 RT/s, `CRGP` could reconstruct the data set consisting of 3.6 million Retweets in just over six seconds. Although this result was achieved on better hardware, it is still significantly better than Lutz's [15] result of under twenty seconds.

## 7.2. Outlook

A comparison of Lutz's work and `CRGP` on the same hardware could not be conducted due to time constraints and the unavailability of the cluster used by Lutz. Such experiments should be executed in the future to understand how much the difference in the reconstruction rates of both systems depends on the hardware.

Currently, only a single worker loads and distributes the social graph. For larger data sets, this could prove to be a bottleneck. `CRGP` could therefore be extended to use multiple workers for this task. Similarly, loading the Retweets could be distributed across several workers, as well.

The reconstruction system considers the direct friends of a user as influencers, that is, the one-hop neighborhood in the social graph. The Research Group on Web Science wants to extend this model to an $n$-hop graph search. For this, the Differential Dataflow framework could prove useful: its ability of arbitrarily nested iterations would allow to implement this functionality on top of `CRGP`.

Social networks have become all-encompassing in our daily lives, influencing what we see and believe, and how we behave outside the Internet. With the threat of filter bubbles, it is therefore important to understand how users interact in the social networks and why they do so. Analyzing their influences can contribute to this understanding. With `CRGP`, such an analysis instrument has been presented in this thesis, that can reconstruct the influences within Retweet cascades on Twitter in real-time.

# A. Usage Documentation

This appendix documents how CRGP, the cascade reconstruction system implemented in this thesis, can be installed and run.

## System Requirements

CRGP has only been tested on *Ubuntu*-based *Linux* systems, but should run on any operating system that is able to run the following software:

- Rust, version 1.17 or newer.[25]
- OpenSSL, version 1.0.1 or newer.[26]

There are no specific hardware requirements. The required main memory depends on the size of the social graph and the distribution of the computation. For distribution across multiple machines, a network link of at least 1 *Gbit/s* is recommended.

## Installation

The source code of CRGP can be downloaded from its repository at https://github.com/BMeu/CRGP. Once downloaded, CRGP can be compiled using Rust's package manager Cargo:

```
$ cd crgp
$ cargo build --release
```

Listing 5: Installation instructions for CRGP.

## Execution

After installation, the compiled binary is available under ./target/release/crgp. Alternatively, CRGP can be compiled and directly run with `cargo run --release`. The basic usage for running CRGP is: `crgp <FRIENDS> <RETWEETS>`.[27] Table 9 explains the required arguments, table 10 the optional flags, and table 11 the options. All usage information can also be obtained by running `crgp --help`.

[25] https://www.rust-lang.org/.

[26] On Linux, the developer packages openssl-devel or libssl-dev (depending on the used distribution), and the regular development utilities pkg-config must be installed. More information on how to install OpenSSL for Rust can be found at: https://github.com/sfackler/rust-openssl/blob/0f02a8b61d9d8878eece3264bdc3271e2bd9b86b/README.md#building.

[27] If run directly via Cargo, a double dash must be inserted to separate Cargo's parameters from the ones of CRGP: `cargo run --release -- <FRIENDS> <RETWEETS>`.

| Argument | Explanation |
|---|---|
| `<FRIENDS>` | Path to the data set containing the social graph's friendships. |
| `<RETWEETS>` | Path to the `JSON` file containing the Retweets. |

Table 9: `CRGP`'s required arguments.

| Short | Long | Explanation |
|---|---|---|
| `-h` | `--help` | Print the full usage information and exit. |
| | `--no-output` | Do not write any results. Overwrites `--output-directory`. |
| | `--pad-users` | Fills the social graph with dummy friendships if there are users with fewer friends than specified in their meta data. |
| | `--connection-progress` | Print connection progress to `STDOUT` when using multiple processes. |
| `-V` | `--version` | Print version information and exit. |
| `-v` | | Enable logging to `STDERR`. This flag can be specified multiple times to set the log level. |

Table 10: `CRGP`'s optional flags.

### Distributed Execution

`CRGP` can be distributed across several machines, as long as each machine can run `CRGP`, and all machines can reach each other over the network via `TCP/IP`.

Assuming $N$ machines will be used, each running a single `CRGP` process with $w$ workers[28]. For each process, a host configuration file must be created with the same contents across all machines: per line, it contains the address (either the hostname or the `IPv4` address) and the port for each process, separated by a colon. An example is given in listing 6. Each process is assigned a unique ID $n \in [0, N-1]$. This ID corresponds to the position of the process's host in the host configuration file; process 0 runs on the host given in line 1, process 1 on the host in line 2, and so on.

Assuming the host configuration file is called `hosts.txt` and located in the same directory as `CRGP`, the computation can be started using the command given in listing 7. Only the process with ID 0 accesses the data sets given with `<FRIENDS>` and `<RETWEETS>`; for all other processes, any other path (even an invalid one) can be given.

[28] Each process *must* use the same number of workers.

```
<HOST 1>:<PORT>
<HOST 2>:<PORT>
...
<HOST N>:<PORT>
```

Listing 6: A host configuration file for CRGP.

```
$ crgp
    --process n
    --processes N
    --workers w
    --hostfile hosts.txt
    <FRIENDS>
    <RETWEETS>
```

Listing 7: Distributed configuration of CRGP. Linebreaks are added for legibility and must be ommitted when running the command.

In the beginning, processes might print the error message `error connecting to worker 0: Connection refused (os error 111); retrying`. This will stop once process 0 is ready. If this error persists, more diagnostic information can be printed by passing the flag `--connection-progress` to the processes.

Machines can also run multiple CRGP processes. This can be achieved by either specifying the same host multiple times but with different ports in the host configuration file, or by using $N > 1$ processes but not specifying a host configuration file[29]. In this case, CRGP will automatically use `localhost` on the ports 2101 through $2101 + (N - 1)$.

### AWS S3 Access

Both the social graph and the Retweet cascade file can be loaded from *Amazon Web Services S3*. This requires programmatic access to *AWS S3* via access and secret keys. These must be given using the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`, respectively, prior to executing CRGP. If an access token is required, it can be specified using the environment variable `AWS_TOKEN`.

The bucket for the social graph is given using the `--s3-sg-bucket` option, the one for the Retweet cascade file via `--s3-tweets-bucket`. The *S3* regions are given with `--s3-sg-region` and `--s3-tweets-region`, respectively. A list of valid regions can be found in table 12. The paths within each bucket are specified via the usual arguments.

[29] This does not allow more than one machine, though.

## Input and Output

This section will explain CRGP's expectations for input file formats and the format of its output files.

### Input

CRGP requires two input data sets: a social graph and the Retweets. Example data sets can be found in the `data` folder in the repository.

**Social Graph** CRGP expects the friends for each user in a CSV file, each user in a defined directory structure within a TAR archive, and each TAR archive in a defined directory structure. Each CSV file contains all the friends (one per line) of a single user. The first line of a file may contain meta data about the user in the format `[Name];[ID];[#Followers];[#Friends];[#Statuses]`, that is, a semicolon-separated list of the user's screen name, their user ID, how many followers they have, how many friends they have, and how many Tweets they wrote. The number of friends in the meta data is allowed to differ from the amount of friends actually specified in the file; this can be used to reduce the size of social graph if its subset is known for a cascade.

The user ID (`[ID]`, must be parsable to `u64`) of a user is encoded into the filename and the directory path:

- Filename: `friends[ID].csv`
- Directory path (without TAR archive): `[ID]` is padded with leading zeroes to twelve digits, then broken into a path with chunks of size three.

Within each top-level folder, the sub-directories are grouped by their first two digits and packed inside a TAR archive with those two digits as file name, for example:

- User 42: file `/000/000/friends42.csv` within archive `000/00.tar`.
- User 1,337: file `/000/001/friends1337.csv` within archive `000/00.tar`.
- User 420,001,000,024: file `/001/000/friends420001000024.csv` within archive `420/00.tar`.

**Retweets** The retweet file is a list of JSON-encoded Retweets, each Retweet on a new line. It may contain Retweets from multiple cascades. Each JSON object must contain the fields listed in the example in listing 8.

### Output

CRPG can create multiple output files: execution statistics, cascade reconstructions, and logs. The path where statistics and reconstructions will be written to can be changed by passing the `--output-directory` parameter to CRGP; it defaults to the current directory.

```
{
    "created_at": 987654321,
    "id": 2,
    "retweeted_status": {
        "created_at": 123456789,
        "id": 1,
        "user": {
            "id": 42
        },
    },
    "user": {
        "id": 1337
    }
}
```

Listing 8: The expected input format of Retweets. Linebreaks are only added for legibility and must be ommitted.

**Execution Statistics** The statistics file collects information on the entire computation in `TOML` format[30]. It contains information on the sizes of the data set, the used configuration, and execution times. All times are given in nanoseconds ($ns$). The Retweet processing rate specifies how many Retweets per second ($RT/s$) were processed on average.

If any error occurred during file creation, the most important statistics will be printed to `STDOUD`. Furthermore, if logging is enabled, the statistics will always be written to the log.

**Cascade Reconstructions** For each processed cascade, `CRGP` will write a result file named `cascs-[ID].csv` where `[ID]` is the ID of the original Tweet in the cascade. Each line is in the format `[ID];[RETWEET];[USER];[INFLUENCER];[TIME];-1`, specifying a single reconstructed influence, with the following replacements:

- `[ID]` is the ID of the original Tweet in the cascade.
- `[RETWEET]` is the ID of the Retweet for which the influence was computed.
- `[USER]` is the user ID being influenced, i. e. the one who posted the Retweet.
- `[INFLUENCER]` is the ID of the user who influenced `[USER]` to post this Retweet.
- `[TIME]` is the timestamp when the Retweet was posted.

[30] `https://github.com/toml-lang/toml`.

- `-1` is static and out-dated; it is only included to easily compare `CRGP`'s results with previous work.

When distributing the computation, each cascade is assigned to one of the machines where the result will be saved to reduce load on the machines. If no results files are needed, `CRGP` can be run with the `--no-output` flag.

**Logging** `CRGP` supports four levels of logging which can be enabled and set via the `-v` flag. In ascending order of severity, the log levels are *Trace* (`-vvvv`), *Info* (`-vvv`), *Warn* (`-vv`), and *Error* (`-v`); specifying a less severe log level always includes all more severe levels. The following events will be logged on each level:

- **Error:**
  - Failures during I/O operations.
- **Warn:**
  - Parse failures: user IDs, Retweets.
  - Encountering users with more friends than the meta data claims.
  - Encountering users without any friends when loading the social graph.
  - Encountering input (e. g. in files) containing invalid `UTF-8`.
- **Info:**
  - Algorithm parameters (e. g. batch size, data sets, number of processes and workers, . . . ).
  - Starting and finishing to load the social graph.
  - Starting and finishing to load the Retweet file into memory.
  - Starting and finishing to process the Retweets.
  - Overall information on the actual number of friends in the social graph, the given number of friends, and the possibly created dummy friends.
  - Execution statistics.
- **Trace:**
  - Invalid directory and file names within the social graph directory and `TAR` files.
  - Creation of result and statistics files.
  - Per-user information on the user's actual number of friends, the given number of friends, and the possibly created fake friends.
  - In-progress information for each processed batch of Retweets.

Unless a target directory is explicitly given (using the `--log-directory` option), log messages are written to `STDERR`. Logging will deteriorate the performance; depending on the level and the input data, log messages with a volume of several hundreds of megabytes will produced.

| **Short** | **Long** | **Argument** | **Explanation** |
|---|---|---|---|
| `-a` | `--algorithm` | `GALE`, `LEAF` | Use the specified algorithm. |
| `-b` | `--batch-size` | $\mathbb{N}^+$; 50,000 | Size of the Retweet batches processed at once. |
| `-f` | `--hostfile` | File path | Path to a text file specifying all hosts in the computation. |
| `-l` | `--log-directory` | Directory path | Create log files in the specified directory. Only effective if logging is enabled with `-v`. |
| `-o` | `--output-directory` | Directory path; . | Create result and statistics files in the specified directory. |
| `-p` | `--process` | $0, \ldots, n-1$ | Identity of this process. $n$ is the number of processes. |
| `-n` | `--processes` | $\mathbb{N}^+$; 1 | Number of processes involved in the computation. |
| | `--s3-sg-bucket` | String | The *AWS S3* bucket for the social graph. |
| | `--s3-sg-region` | Region string (table 12) | The *AWS S3* region for the social graph. |
| | `--s3-tweets-bucket` | String | The *AWS S3* bucket for the Retweet cascade file. |
| | `--s3-region-bucket` | Region string (table 12) | The *AWS S3* region for the Retweet cascade file. |
| `-w` | `--workers` | $\mathbb{N}^+$; 1 | Number of worker threads per process. |

Table 11: `CRGP`'s options. The *Argument* column contains a description of valid values; default values are underlined.

| **AWS Region** | **Region String** |
|---|---|
| US East (North Virginia) | `us-east-1` |
| US East (Ohio) | `us-east-2` |
| US West (North California) | `us-west-1` |
| US West (Oregon) | `us-west-2` |
| Canada (Central) | `ca-central-1` |
| Asia Pacific (Mumbai) | `ap-south-1` |
| Asia Pacific (Tokyo) | `ap-northeast-1` |
| Asia Pacific (Seoul) | `ap-northeast-2` |
| Asia Pacific (Singapore) | `ap-southeast-1` |
| Asia Pacific (Sydney) | `ap-southeast-2` |
| EU (Frankfurt) | `eu-central-1` |
| EU (Ireland) | `eu-west-1` |
| EU (London) | `eu-west-2` |
| South America (São Paulo) | `sa-east-1` |

Table 12: Valid *AWS S3* regions and the respective region string for `CRGP`'s command-line interface.

# B. Additional Scripts

This appendix contains some additional scripts that were used when evaluating the work presented in this thesis. Not all scripts were written by the author of this thesis; acknowledgements are given where necessary.

## Main Memory Usage

The `bash` script given in listing 9 and written by Jaeho Shin reports the peak memory usage of a process in kibibytes (*kiB*). It runs the given command and watches the commands current memory usage in 0.1 *s* intervals.

```
#!/usr/bin/env bash
# memusg -- Measure memory usage of processes
# Usage: memusg COMMAND [ARGS]...
#
# Author: Jaeho Shin <netj@sparcs.org>
# Created: 2010-08-16
#
# Copyright 2010 Jaeho Shin.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied. See the License for the specific language governing
# permissions and limitations under the License.

set -um

# check input
[[ $# -gt 0 ]] || { sed -n '2,/^#$/ s/^# //p' <"$0"; exit 1; }

pgid=$(ps -o pgid= $$)
# make sure we're in a separate process group
if [[ "$pgid" == "$(ps -o pgid= $(ps -o ppid= $$))" ]]; then
    cmd=
    set -- "$0" "$@"
    for a; do cmd+="'${a//"'"/"'\\''"}' "; done
    exec bash -i -c "$cmd"
```

```
fi

# detect operating system and prepare measurement
case $(uname) in
    Darwin|*BSD) sizes() { /bin/ps -o rss= -g $1; } ;;
    Linux) sizes() { /bin/ps -o rss= -$1; } ;;
    *) echo "$(uname): unsupported operating system" >&2; exit 2 ;;
esac

# monitor the memory usage in the background.
(
peak=0
while sizes=$(sizes $pgid)
do
    set -- $sizes
    sample=$((${@/#/+}))
    let peak="sample > peak ? sample : peak"
    sleep 0.1
done
echo "memusg: peak=$peak" >&2
) &
monpid=$!


# run the given command
exec "$@"
```

Listing 9: Script by Jaeho Shin to measure the peak memory usage of a process, available at `https://gist.github.com/netj/526585`.

## Result Verification

The `bash` script given in listing 10 takes two input files, sorts them, and compares them line by line. The result is written to an output file. If this output file is empty, the input files are identical (except possibly the ordering). If the input files differ, the lines that can only be found in the first file will be written to the output file's first column, lines only in the second file to the second column.

```
#!/bin/bash

# Copyright 2017 Bastian Meyer
#
# Compares the two input files line by line and writes the
# differences to the output file.

if [ $# -ne 3 ]
   then
       echo "USAGE: $0 [INPUT FILE 1] [INPUT FILE 2] [OUTPUT FILE]"
       echo ""
       echo "Compares the two input files line by line and writes \
            the differences to the output file."
       exit 1
fi

comm -3 <(sort $1) <(sort $2) > $3
```

Listing 10: Verification script for reconstructed results.

# List of Code Listings

# List of Figures

# List of Tables

# Bibliography

All websites have last been accessed July 27, 2017.

[1] Amazon Web Services, Inc. *Amazon EC2 Instance Types* (2017). `https://aws.amazon.com/ec2/instance-types/`.

[2] Bakshy, Eytan; Hofman, Jake M.; Mason, Winter A.; et al. *Everyone's an Influencer: Quantifying Influence on Twitter.* In: *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining*, WSDM '11, pp. 65–74. ACM, New York, NY, USA (2011). ISBN 978-1-4503-0493-1. `http://doi.acm.org/10.1145/1935826.1935845`.

[3] Beevolve, Inc. *An Exhaustive Study of Twitter Users Across the World* (October 2012). `http://www.beevolve.com/twitter-statistics/`.

[4] Bozdag, Engin. *Bias in Algorithmic Filtering and Personalization.* Ethics and Information Technology, 15 (3), pp. 209–227 (September 2013). ISSN 1388-1957, 1572-8439. `https://link.springer.com/article/10.1007/s10676-013-9321-6`.

[5] Cha, Meeyoung; Haddadi, Hamed; Benevenuto, Fabricio; et al. *Measuring User Influence in Twitter: The Million Follower Fallacy.* Icwsm, 10 (10-17), p. 30 (2010).

[6] Department of Economic and Social Affairs. *World Population Prospects. Key Findings & Advance Tables.* United Nations, New York, NY, USA (2017). `https://esa.un.org/unpd/wpp/Publications/Files/WPP2017_KeyFindings.pdf`.

[7] Facebook, Inc. *Company Info* (July 2017). `https://newsroom.fb.com/company-info/`.

[8] Garrett, R. Kelly. *Echo Chambers Online?: Politically Motivated Selective Exposure among Internet News Users1.* Journal of Computer-Mediated Communication, 14 (2), pp. 265–285 (January 2009). ISSN 1083-6101. `http://onlinelibrary.wiley.com/doi/10.1111/j.1083-6101.2009.01440.x/abstract`.

[9] Garrett, R. Kelly; Carnahan, Dustin; Lynch, Emily K. *A Turn Toward Avoidance? Selective Exposure to Online Political Information, 2004–2008.* Political Behavior, 35 (1), pp. 113–134 (March 2013). ISSN 0190-9320, 1573-6687. `https://link.springer.com/article/10.1007/s11109-011-9185-6`.

[10] Hermann, Dave. *Why Rust Is the Most Loved Language by Developers* (April 2017). `https://medium.com/mozilla-tech/why-rust-is-the-most-loved-language-by-developers-666add782563`.

[11] Huber, Michael. *Verteilte Rekonstruktion von Informationskaskaden.* Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany (November 2014).

[12] Instagram, Inc. *700 Million* (April 2017). `http://blog.instagram.com/post/160011713372/170426-700million`.

[13] Krikorian, Raffi. *New Tweets per Second Record, and How!* (August 2013). `https://blog.twitter.com/engineering/en_us/a/2013/new-tweets-per-second-record-and-how.html`.

[14] Lutz, Bernhard. *Evaluation of Social Graph Partitioning Strategies and Improvement of Distributed Reconstruction of Retweet Cascades.* Master's project, Albert-Ludwigs-Universität, Freiburg, Germany (May 2016).

[15] Lutz, Bernhard. *Large Scale Distributed Reconstruction of Retweet Cascades.* Master's thesis, Albert-Ludwigs-Universität, Freiburg, Germany (January 2017).

[16] McSherry, Frank. *Differential-Dataflow: An Implementation of Differential Dataflow Using Timely Dataflow on Rust* (June 2017). `https://github.com/frankmcsherry/differential-dataflow`.

[17] McSherry, Frank. *Timely-Dataflow: A Modular Implementation of Timely Dataflow in Rust* (June 2017). `https://github.com/frankmcsherry/timely-dataflow`.

[18] McSherry, Frank; Isaacs, Rebecca; Isard, Michael; et al. *Composable Incremental and Iterative Data-Parallel Computation with Naiad.* Microsoft Research (2012). `https://www.microsoft.com/en-us/research/wp-content/uploads/2012/10/naiad.pdf`.

[19] McSherry, Frank; Murray, Derek G; Isaacs, Rebecca; et al. *Differential Dataflow.* Microsoft Research (2013). `https://www.microsoft.com/en-us/research/wp-content/uploads/2013/01/differentialdataflow.pdf`.

[20] Microsoft Research. *Naiad* (March 2017). `https://github.com/TimelyDataflow/Naiad`.

[21] Murray, Derek G.; McSherry, Frank; Isaacs, Rebecca; et al. *Naiad: A Timely Dataflow System.* In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pp. 439–455. ACM, New York, NY, USA (2013). ISBN 978-1-4503-2388-8. `http://doi.acm.org/10.1145/2517349.2522738`.

[22] Pariser, Eli. *The Filter Bubble: How the New Personalized Web Is Changing What We Read and How We Think.* Penguin, New York, NY, USA (2011). ISBN 978-0-14-196992-3.

[23] reddit, Inc. *Advertise* (July 2017). `https://about.reddit.com/advertise/`.

[24] Sakaki, Takeshi; Okazaki, Makoto; Matsuo, Yutaka. *Earthquake Shakes Twitter Users: Real-Time Event Detection by Social Sensors.* In: *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pp. 851–860. ACM, New York, NY, USA (2010). ISBN 978-1-60558-799-8. `http://doi.acm.org/10.1145/1772690.1772777`.

[25] Sättler, Lukas; Ebner, Simon. *Social Media Analysis.* Team project, Albert-Ludwigs-Universität, Freiburg, Germany (November 2013).

[26] Signorini, Alessio; Segre, Alberto Maria; Polgreen, Philip M. *The Use of Twitter to Track Levels of Disease Activity and Public Concern in the U.S. during the Influenza A H1N1 Pandemic.* PLOS ONE, 6 (5), p. e19 467 (May 2011). ISSN 1932-6203. `http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0019467`.

[27] The Rust Project Developers. *How Fast Is Rust?* (2017). `https://www.rust-lang.org/en-US/faq.html#how-fast-is-rust`.

[28] Tumasjan, Andranik; Sprenger, Timm Oliver; Sandner, Philipp G; et al. *Predicting Elections with Twitter: What 140 Characters Reveal about Political Sentiment.* Icwsm, 10 (1), pp. 178–185 (2010).

[29] Twitter, Inc. *Twitter Developer Documentation: Tweets* (June 2017). `https://dev.twitter.com/overview/api/tweets`.

[30] Twitter, Inc. *Twitter Usage* (2017). `https://about.twitter.com/company`.

[31] YouTube LLC. *Press* (July 2017). `https://www.youtube.com/yt/about/press/`.

# Declaration

I hereby declare, that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

| | |
|---|---|
| Freiburg, July 31, 2017 | |
| (Place, Date) | (Signature) |