2012 ASE/IEEE International Conference on Social Computing and 2012 ASE/IEEE International Conference on Privacy, Security, Risk and Trust

Efficient Graph Models for Retrieving Top-k News Feeds from Ego Networks

Rene Pickhardt, Thomas Gottron, Ansgar Scherp, Steffen Staab
Institute for Web Science and Technology
Universtity of Koblenz-Landau
Koblenz, Germany
Email: {rpickhardt,gottron,scherp,staab}@uni-koblenz.de

Jonas Kunze
Metalcon.de
Johannes Gutenberg University
Mainz, Germany
Email: jonas@metalcon.de

***Abstract*—A key challenge of web platforms like social networking sites and services for news feed aggregation is the efficient and targeted distribution of new content items to users. This can be formulated as the problem of retrieving the top-k news items out of the d-degree ego network of each given user, where the set of all users producing feeds is of size n, with $n \gg d \gg k$ and typically $k < 20$. Existing approaches employ either expensive join operations on global indices or suffer from high redundancy through denormalization. This makes retrieval of different top-k news feeds for thousands of users per second very inefficient in a large social network. In this paper, we propose two graph models GRAPHITY and STOU to address this problem. GRAPHITY is optimized for fast retrieval of news feeds and has a runtime of $O(k \log(k))$. The GRAPHITY index does not involve data redundancy. An update of the index upon insertion of a new item to the feed is possible in a runtime linear to the nodes' indegree d_{in}. New content can be stored in STOU in $O(1)$ at the cost of slower retrieval speed of $O(d \log(d))$. We verify the theoretical runtime complexity of GRAPHITY and STOU on two data sets of different characteristics and size. We show that on a single machine GRAPHITY is able to retrieve more than 10 000 unique news feeds per second in a network with more than one million users. Our evaluation confirms that retrieval of news feeds with GRAPHITY is independent of the node degree d of a user's ego network and network size n and does scale to networks of arbitrary size.**

***Index Terms*—social network; graph data base; social news feed; graphity; graph index; scalability; top k join**

I. INTRODUCTION

Providing streams of recent news items is a typical feature of today's social networking platforms like Facebook[1] and news feed aggregators such as Google Reader[2]. These streams of news items depend heavily on a user's friendship network or a user's individual choice of news sites to follow. Furthermore, users typically are interested only in the most recent news items, i.e. the top-k news items. This leads to a scenario, where *content items* have to be assembled based on the local ego network of an *aggregator node* and sorted by a global relevance criterion (e.g. by date of creation). Typically, the aggregator is a user and his ego network is formed by his friends or his personal news interests.

Depending on the concrete setting, it is necessary to compute top-k news item feeds for thousands of users per second and to flexibly reflect changes in the ego networks or newly created content items. Thus, the task requires an algorithmic solution which is flexible, efficient, and scalable. In this context efficiency means that the most recent news items can be computed fast for an individual user. Scalability, instead, requires that the performance is independent of the network size, e.g. the number of users, friendship relations, or content items.

[1]http://www.facebook.com/
[2]http://www.google.com/reader/

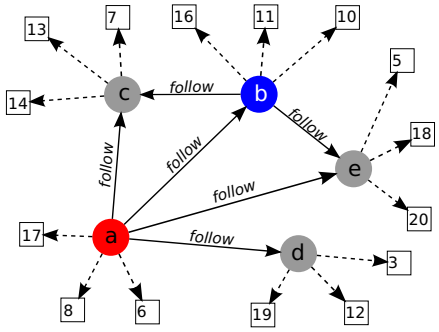


Fig. 1. Social network of users (round nodes) and the news items they produce (squared nodes).

A. Social Network Example

Let us illustrate the task of retrieving the top-k news items feed for a user with a concrete example in the scenario of a social network. In Figure 1, the circular nodes a, b, c, d and e, represent users in a social network. The *follow* relation between two circular nodes represents a directed friendship relation of two users in the social network. Over time, the users create content items, e.g. they post status updates, upload photos, share links, etc. These content items are represented as squared nodes in the graph in Figure 1 and are linked to the user who created them by a dashed *createdNews* relation. The numbers in these squared nodes represent the item's creation time, where a higher value indicates a more recently generated content item.

To retrieve the top-k news items feed of a user, we have to consider the user's ego network as well as the content items created in this network. For instance, the ego network of node a is the set $\{b, c, d, e\}$ as indicated by the *follow* edges in the

978-0-7695-4848-7/12 $26.00 © 2012 IEEE
DOI 10.1109/SocialCom-PASSAT.2012.73

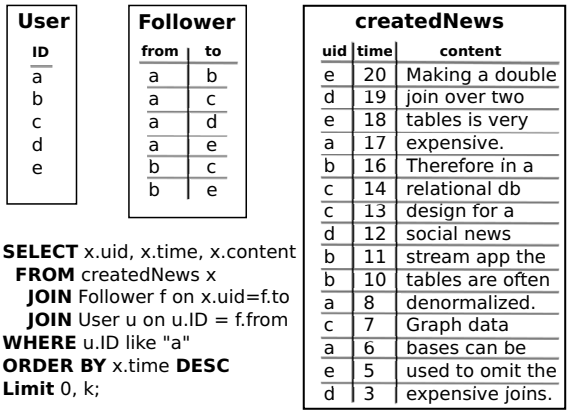


Fig. 2. Relational data base scheme for a social news stream application and query with 2 joins to retrieve the most recent content items for aggregation node a from its ego network

graph. In order to retrieve the top-k news items of node a, all nodes in the ego network are visited for collecting the content items they have created. These content items are ordered by time and aggregated by cutting off all items after position $k+1$.

B. Solutions to Top-k News Feed Aggregation

The procedure we just described corresponds to the task of top-k join queries in relational data bases [1] and is depicted in Figure 2. In a relational model, our scenario above translates to a join operation over the users based on the *follow* relation and a second join with the news items based on the *createdNews* relation. The disadvantage of this approach is exactly the need to perform two join operations and a consecutive sorting, which renders retrieval a very expensive operation (see also the examples in [1]).

An alternative naive approach is to give up normalization in the data model and store the history of content items redundantly for each individual user. These redundant content lists entail a much higher need for storage capacity and can cause anomalies when changing the topology of the network graph, e.g. when new *follow* relations between users are established or existing ones break up.

In this paper, we develop two new graph-based models for fast and flexible retrieval of the top-k news items feeds in such scenarios as described above: STOU and GRAPHITY. The STOU approach is an extension of top-k join query processing on relational data bases. It involves modeling the problem in a graph data base and taking advantage of the specific strengths of graph data bases, which allow a more flexible possibility to model chaining relations of items. In this way it is possible to easily maintain pre-sorted lists of content items which allow a more efficient sorting while retrieving items. In our case, sorted lists are more efficient than a b-tree. This is due to the fact that the sorting is done by the timestamp of the creation time. Thus, new items are always inserted at the head of our sorted list. The retrieval operation of STOU depends on the size of the ego network, i.e. the number of *follow* relations. Index maintenance operations, e.g. as they are required for the creation of new content items or when adding new friendship relations, have constant runtime in STOU. As a second data structure, we introduce GRAPHITY which yields a still better retrieval performance. The runtime of GRAPHITY depends only on k, the number of retrieved content items. In particular the retrieval operation is independent of the size of the ego network, the overall number of news items, and other characteristics of the social network graph. However, index maintenance operations are more expensive than in STOU but as in denormalized relational approaches they are at most linear in the size of the ego network. Finally, neither for STOU nor for GRAPHITY it is necessary to store content items redundantly. Thus, we present two approaches with varying strengths and weaknesses, which can be chosen for the top-k news feed aggregation depending on the application scenario.

Further contributions of our work are the following:

- We show analytically the runtime of STOU and GRAPHITY as well as of several baseline methods.
- We conduct an empiric evaluation showing that our graph models perform very well in realistic scenarios.
- We provide evidence which of the two indices to choose in which setting, depending if update or read operations are more frequent.

The remainder of the paper is organized as follows: In Section II, we formalize the problem of retrieving the top-k news items feeds for ego networks. Afterwards in III, we introduce our novel index models STOU and GRAPHITY for increasing efficiency of retrieving the news feeds for ego networks. In Section IV, we describe different baselines and conduct a theoretical analysis and comparison of the baselines with STOU and GRAPHITY. Besides the theoretical analysis and comparison of our model with the baselines, we have also implemented the algorithms and evaluated them on two datasets of different size and characteristics. The data sets are described in more detail in Section V, while the actual evaluation is presented in Section VI. In our evaluation, we have conducted different experiments to show among others GRAPHITY's independence of the node degree d and network size for retrieving the news feeds and have compared the performance of STOU in updating and maintaining the index when the network changes. As our theoretical analysis predicted, GRAPHITY outperforms the baselines in retrieving the news feeds, while STOU is more efficient in maintenance operations. In Section VII, we discuss related work and compare it to our approaches before discussing the lessons learned and concluding the paper in VIII.

Please note that the source code of our evaluation framework, the implementation of GRAPHITY and STOU, and the preprocessed wikipedia data files can be downloaded under an open source license at:
http://www.rene-pickhardt.de/graphity-source-code/

II. FORMALIZATION OF TOP-k NEWS ITEM FEED AGGREGATION FROM EGO NETWORKS

In order to formalize the problem of retrieving the top-k most recent content items from a user's ego network and to

Term	Notation
Nodes	$V = A \cup C$
Aggregating nodes	A
Content nodes	C
Labels	$L = \{\textit{follows}, \textit{createdNews}\}$
Edges	$E \subset V \times L \times V$
$Ego(a)$	$\{b \in A \mid (a, \textit{follows}, b) \in E\}$
$C(a)$	$\{c \in C \mid (a, \textit{createdNews}, c) \in A \times L \times C\}$
$News(a)$	$\{c \in C \mid \exists b \in Ego(a) : c \in C(b)\}$
$News_k(a)$	$Top_k(Sort(News(a)))$
Size of the network	$n = \lvert V \rvert + \lvert E \rvert$
Node degree	d
In / Out degree	d_{in}, d_{out}
News feed length	k
Avg. # content nodes per aggregating node	$u = 1/\lvert A \rvert \sum_{a \in A} \lvert C(a) \rvert$

TABLE I
NOTATION TO REPRESENT SOCIAL NETWORKS AND NEWS FEED AGGREGATION NETWORKS

discuss several baselines we define some notation. For quick reference, the same notation is also summarized in Table I.

- Let $G(V, L, E)$ be a graph with a finite set of vertices V, a finite set of labels L and a set of edges $E \subset V \times L \times V$.
- The set of vertices V is a complete partitioning i.e. $V = A \cup C$, $A \cap C = \emptyset$ into aggregating nodes A (e.g. users in a social network) and content nodes C (e.g. news items created by users)
- L consists of the set $\{\textit{follow}, \textit{createdNews}\}$. These two labels reflect the partitioning of $V = A \cup C$ and the relations between the different elements in the graph. Edges between nodes in A are labeled with *follow* and, for instance, correspond to the social network. Edges from nodes in A to nodes in C are labeled with *createdNews*.
- The elements $c \in C$ are pairs $c = (t, s)$, where t is a timestamp and s is the content which can be a text, a photo, a video, etc. We write $t(c)$ to denote the timestamp of a content node c.
- Each content node is created by exactly one aggregating node $a \in A$.
- We define the set $C(a) = \{c \in C \mid (a, \text{createdNews}, c) \in E\}$ to be the set of all content nodes that belong to a (e.g. news items created by a specific user).
- We call the set $Ego(a) := \{b \in A \mid (a, \textit{follows}, b) \in E\}$ the *ego network* of an arbitrary aggregating node $a \in A$.
- $In(a)$ is defined to be the set of aggregating nodes following a, i.e. $In(a) = \{b \in A \mid (b, \textit{follow}, a) \in E\}$.

The problem of retrieving the top-k news items feed of a user a can now be formalized as computing $News_k(a)$, which is defined by $News_k(a) = Top_k(Sort(\{c \in C \mid \exists b \in Ego(a) : c \in C(b)\}, t))$ where *Sort* is a generic sorting function using the timestamp extraction function t to extract a value from each element that is sorted and Top_k returns the first k elements of the list returned by *Sort*.

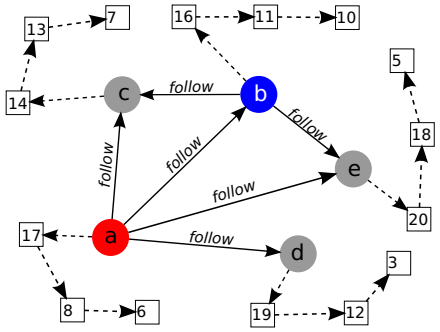


Fig. 3. Social network graph. Content items are now stored as a temporally ordered linked list improving the runtime of retrieving news streams.

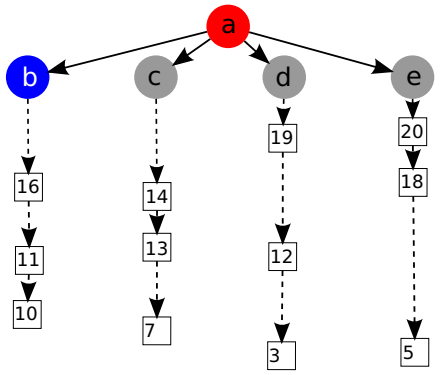


Fig. 4. Ego network of aggregating node a. Remark that for retrieving the most recent content item from $Ego(a)$ all aggregating nodes $b \in Ego(a)$ have to be visited

III. TWO GRAPH MODELS FOR RETRIEVING SOCIAL NEWS STREAMS

As described above, we apply methods from graph data bases to the task of top-k news items feed aggregation. This allows for two key changes in the model. For the STOU approach, we convert the *createdNews* relation to a list structure in the graph as shown in Figure 3. In this way, we can store the content items for each aggregating node sorted by the time of their creation, which in turn allows a quicker retrieval and sorting of the top-k news items by means of a merge algorithm. A second step introduced for GRAPHITY is to additionally maintain a pre-sorted list of the nodes in a ego network based on where the most recent content items were created. This allows for even quicker compilation of the top-k items from the sorted lists of content items. We now explain first the STOU algorithm and subsequently extend this approach to GRAPHITY.

A. Ego Network Representation and Data Structures for STOU

In Figure 3, we have organized the content nodes generated by each aggregating node in a linked list, the entries of which are ordered according to their timestamps. The ego network of node a is shown in Figure 4. Extending our notation from Section II, we introduce this list by simply connecting the

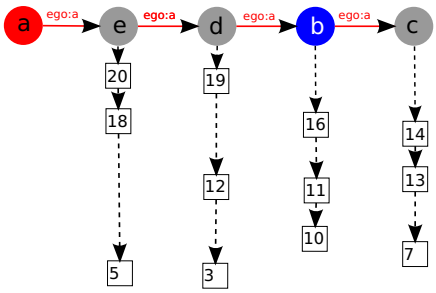


Fig. 5. Ego network of aggregating node a with it's GRAPHITY index. Please note that the linked list $ego{:}a$ is implemented as doubly linked list.

content nodes with directed edges with a new label ***nextItem***. This means, we connect two content nodes $c_i, c_j \in C(a)$ generated by the same aggregating node a with an edge $(c_i, \textit{nextItem}, c_j)$ if they are subsequently published content items. Formally, we connect $t(c_i)$ and $t(c_j)$ if $t(c_i) > t(c_j)$ and if there is no $c_k \in C(a)$ such that $t(c_i) > t(c_k) > t(c_j)$. In order to retrieve the news feed for an aggregating node the lists of content nodes in its ego network have to be merged. Given that we operate on sorted lists, this step can be achieved efficiently. The remaining drawback of this method is that even for obtaining only the first most recent content item (i.e. $k = 1$) in $\mathit{News}_1(a)$ still all ***follows*** edges from a to all nodes in $\mathit{Ego}(a)$ have to be traversed.

B. Ego Network Representation and Data Structures in GRAPHITY

The key idea of GRAPHITY is to execute this expensive step of sorting a node's ego network while retrieving a stream to a step that happens less frequently: the creation of a new content item. Thereby, we benefit from the fact that we have to perform the sorting less often. Furthermore, sorting becomes easier and computationally less expensive as we do not have to sort all the nodes in an ego network, but simply update an already sorted list by moving one of its elements to the head of list – the one that has just created a new content item.

In order to achieve this, we define an individual GRAPHITY index for each aggregating node. The individual index consists of a list of length d_{out} of aggregating nodes ordered according to the time stamps of the content nodes they contribute. The lists are doubly linked and each list is distinguished from all others by using its own edge type (implying as many edge types as there are aggregating nodes in the network, e.g. users).

Extending our example from Figure 3, we show a GRAPHITY index for user a in Figure 5. For the aggregating node a, we introduced edges with the label ego:a. These edges build a linked list of all the aggregating nodes in a's ego network. In our example, the first edge connects a to the aggregating node e, as e has created the most recent content node (timestamp 20) in a's ego network. The GRAPHITY index for a further lists node d (most recent content node has timestamp 19) b (timestamp 16) and c (timestamp 14).

C. Retrieving News Items Feeds in STOU and GRAPHITY

Recall that for every aggregating node a the linked list of content items $C(a)$ stores the content items that are created by a. When retrieving a news feed for a, we have to look at $C(b)$ for every $b \in \mathit{Ego}(a)$ and extract the top-k most recent content items into a joint sorted list. The runtime for STOU is $O(d \log(d))$ since for all d nodes $b \in \mathit{Ego}(a)$ the list $C(b)$ has to be considered when merging the items. In the case of GRAPHITY retrieving a news feed can be achieved by the application of a top-k n-way merge algorithm which is attached in Appendix A. Therefore, GRAPHITY's runtime is only $O(k \log(k))$ due to the fact that at most the k lists with the k most recent updates have to be included in top-k n-way merge. This means, we do not have to consider the content items $C(b)$ of users where already their most recent item would not qualify to get into the top-k list. In Figure 5 one can see that for $k = 3$ only two lists have to be included for retrieving $\mathit{News}_3(a)$ with the help of GRAPHITY. We remark that an additional feature of GRAPHITY is that the number of k can be set individually and flexibly for every aggregation node during runtime. Furthermore, it is possible to extend a top-k list by fetching l additional items at relatively small computational cost (for details see Appendix A).

D. Incrementally Maintaining the Data Structures

There are several types of events which generate the need to update the data structures. These events are the creation or deletion of a new content node, the creation of an aggregating node and changes in the ***follow*** network structure, i.e. the addition or deletion of ***follow*** edges between aggregating nodes. We will now look at the theoretic effort involved in updating the index for these events.

1) Inserting a Content Node: When an aggregating node a creates a new content node, both methods STOU and GRAPHITY need to add it as the top element of the list representing $C(a)$. This operation has a runtime complexity of $O(1)$. For GRAPHITY additional effort has to be done. The index of each node following a has to be updated, i.e. a has to be moved to the beginning of every list representing the GRAPHITY index of its followers. Since the GRAPHITY index is a doubly linked list and the ***follow*** edges remain in the graph this operation can be done in $O(1)$ for every follower and, thus, in a time linear for all of the followers i.e. $O(d)$.[3]

2) Inserting a new Aggregating Node: This operation is rather simple for both STOU and GRAPHITY. A new aggregation node does not have any ***follow*** relations, neither has it created any content items yet. Accordingly it is sufficient to create the node and, thus, this operation is of $O(1)$.

[3]An alternative to updating the GRAPHITY index of all aggregating nodes following an aggregating node that has created a content node is to simply mark the index as outdated and perform bulk updates at pre-defined intervals or on demand when an affected news feed is retrieved. However, as marking the indices of the aggregating nodes following a as outdated is of $O(d_{in})$ there is no real gain over incrementally updating the GRAPHITY indices at the same cost.

3) Deleting a Content Node: Deleting a content node is $O(1)$ for STOU and for the most cases in GRAPHITY, as it is sufficient to remove an item from the sorted list representing $C(a)$. A particular case for GRAPHITY is, when the most recent content node of an aggregating node is deleted, as this can affect the sorted list of users in an ego network. For example, in Figure 5 node d wants to delete the content node with timestamp 19. In this case, the GRAPHITY index of a and the one of all other nodes following d has to be updated. Each update includes shifting the node d to the correct position of the GRAPHITY index. This means that this rare operation depends on the node degree d and is $O(d^2)$ for GRAPHITY.

4) Changing the Friendship Graph: In the case of STOU these operations are obviously $O(1)$ as it is sufficient to add or remove a ***follow*** edge. Again in the case of GRAPHITY we have to pay more attention.

If a new edge $(a, \textit{follow}, b)$ is added from a to b, the node b will have to be inserted into a's GRAPHITY index. Since the list representing the index needs to be in temporal order, the insertion point of b depends on the timestamp of the last content node b created as well as on the timestamps of the nodes in the linked list. This requires a linear traversal through the linked list to identify the correct insertion point. Thus, the algorithm for adding a ***follow*** edge between aggregating nodes is linear in the node degree of a.

Removing an existing edge $(a, \textit{follow}, b)$ entails the deletion of b from the GRAPHITY index of a. Using the edge that is to be removed, we can directly access b in $O(1)$. Since the GRAPHITY index consists of a doubly linked list, we can remove the element b by interlinking its predecessor and successor in $O(1)$.

IV. BASELINES AND THEORETICAL COMPARISON

After having analyzed our two models GRAPHITY and STOU in detail, we now introduce alternative approaches for computing $\textit{News}_k(a)$ news items feeds. These approaches are taken from related work in the field and correspond to a relational model (ST) as well as denormalization by introducing redundancy in the data model (RCL). The ST approach is in principle comparable to conducting top-k join queries [1] in relational databases as introduced in Section I. The RCL approach of storing content items redundantly is followed by Li et. al [2] in their system for personalized recommendations in social networks[4] (see also Section VII on related work).

A list of redundant content items is used in settings similar to social networks. The UNIX emailing platform sendmail [3] provides a distinct file in the /var/spool/mail directory containing a list of the emails of each individual user. Thus, any email send to more than one user within the same sendmail installation is stored redundantly on the hard disk.

We will briefly describe the ideas of the approaches and analyze their complexity. In this context, we will frequently depend on the average number of content nodes attached to an aggregating node. We will refer to this value as u.

[4]From the paper it is unclear whether the authors actually use lists or employ a different data structure. However, this detail does not affect the performance in our setting.

A. Joining Along the Star Topology of Nodes (ST)

The star topology baseline is motivated by the naive modeling of a social network graph of creating an edge between two vertices whenever a connection exists in the network. This approach can be seen as the direct implementation of the social network model of Figure 1. This approach enables fast insertion since all kinds of operations require only a single write operation. However, retrieval of news feeds is highly inefficient. This is due to the fact that it involves a breadth first search of depth 2 to reach out first to the aggregating nodes in the ego network over the ***follow*** edges and then to obtain their content nodes over the ***createdNews*** edges. Afterwards all content nodes have to be sorted by time which is in $O((d \times u)\log(d \times u))$.

B. Redundant Content Lists (RCL)

The RCL approach is motivated by the idea of heavy denormalization of data and a redundant way to store them in order to make retrieval as fast as possible. Every aggregating node a in the graph maintains its own sorted list of content items from the ego network of the entity $\textit{Ego}(a)$. Furthermore RCL can easily be distributed. Whenever an aggregating node a creates a content item all d lists for every node b with $a \in \textit{Ego}(b)$ have to be updated. This is the same amount of updates as is required in GRAPHITY. In order to achieve comparability, we implemented the RCL in a graph data base which limited our ways in scaling. For obvious reasons the storage used by this approach is $O(d \times (|V| + |R|))$

C. Theoretical Analysis and Comparison

Table II summarizes the theoretic analysis of runtime and space complexity for the baseline approaches ST and RCL as well as our approaches STOU and GRAPHITY. From this table, it becomes obvious, that STOU clearly dominates the ST approach. It can be seen that STOU is faster in retrieving, without drawbacks in storage or adding and removing ***follow*** edges as well as inserting new content nodes.

Also when comparing GRAPHITY to RCL the advantages of GRAPHITY become obvious. GRAPHITY has a far lower space complexity and is capable of dynamically adding and removing edges. Due to the denormalization, it depends on the concrete implementation if and at which cost RCL supports and reflects changes in the social network when retrieving the top-k news items. The complexity for adding a content node is the same for both approaches. Only for computing a news items feed, GRAPHITY has a complexity of $O(k\log(k))$ compared to $O(k)$ for RCL. But, as we can consider k to be rather small parameter, both fall back to constant runtime in practice.

V. USED DATA SETS

In order to verify the theoretical runtime complexity of the different approaches in practice, we needed to apply them to

Task	ST	RCL	STOU	GRAPHITY
Storage	$\|V\| + \|E\|$	$d \times (\|V\| + \|E\|)$	$\|V\| + \|E\|$	$\|V\| + 2 \times \|E\|$
Retrieve news feed	$O((d \times u)\log(d \times u))$	$O(k)$	$O(d \times \log(d))$	$O(k \log(k))$
Create Content node	$O(1)$	$O(d)$	$O(1)$	$O(d)$
Add follow edge	$O(1)$	unclear	$O(1)$	$O(d)$
Remove follow edge	$O(1)$	unclear	$O(1)$	$O(1)$

TABLE II
THEORETICAL AGGREGATED AVERAGE RUNTIME OF THE DIFFERENT APPROACHES

Year	$\|A\|$	$\|A_{d>10}\|$	$\|E\|_A$	$\|C\|$
Metalcon	0.06	0.05	0.36	0.3
2004	0.17	0.03	0.9	0.2
2005	0.51	0.11	4.1	1.9
2006	0.81	0.23	8.4	5.8
2007	1.16	0.36	12.9	12.3
2008	1.43	0.48	16.5	19.1
2009	1.65	0.60	20.4	25.8
2010	1.86	0.72	24.5	31.9
2011	2.06	0.84	28.9	38.5

TABLE III
NETWORK CHARACTERISTICS OF THE DATA SETS (IN MILLIONS). $A_{d>10}$ IS THE SET OF AGGREGATION NODES WITH DEGREE BIGGER THAN 10

real world data. To our best knowledge there is no established dataset to evaluate the runtime behavior of index structures on dynamic networks. Therefore, we introduce two new datasets which are described in detail in the following sections. The Metalcon dataset provides the scenario of a real social network with different types of aggregating and content nodes. The Wikipedia dataset serves for the analysis of scalability issues.

A. Usecase 1: Social Networks - Metalcon

For our first use case, we took data from the social networking site Metalcon. It is a German special purpose social network for fans of heavy metal music and musicians. The data set consists of 60,158 aggregating nodes, 356,978 follow edges, and 303,581 content nodes.

It is worthwhile mentioning that only 8 thousand of these aggregating nodes are users. The other 52 thousand nodes represent bands, records, and geographical places. The content items consist of uploaded pictures, reviews for the records and discussions of the users which are linked to the bands or some general topics of interest. Even though so many different types of aggregating nodes and content nodes are involved the data follows a typical power law.

Furthermore, for Metalcon we do also have statistics about deletion of content items. Only 0.34% of all content items have been deleted while running the platform. Given this very small ratio of data being deleted, we did not consider this operation in our evaluations.

B. Usecase 2: Beyond Social Networks - Wikipedia

The retrieval of news feeds from ego networks may be of high interest even beyond social networks. One example is the reporting of updates in Wikipedia pages. This will also test the scalability of GRAPHITY to a larger extent than Metalcon.

Editors in Wikipedia may be interested in observing changes of the content in pages that reference "their" Wikipedia pages, as such changes may affect the proper meaning and context of the corresponding page. We can represent this use case by modeling each Wikipedia page as an aggregating node and each update action to a page as a content node. Thus, $News_k(a)$ models the k most recent changes applied to pages referencing page a. A change in the link structure is furthermore modeled as a change of the ego networks. We use a data dump of the German Wikipedia containing all revisions of all articles since its emergence in 2004 until the August, 13th 2011. The evolution of the data set over time is represented in Table III by showing some characteristics for snapshots at given points in time.

VI. EVALUATION

As proof of concept and to verify the theoretically predicted runtime in a real implementation of a graph database, we have conducted an experimental evaluation. For this purpose, we implemented the baselines ST and RCL as well as our models STOU and GRAPHITY on top of the Neo4j[5] graph data base. We point out that ST baseline is corresponding to the relational data base design. Implementing it also on a graph data base (using the same technology Neo4j) makes the results more comparable. For that reason, RCL was also implemented on a graph data base. Using two data scenarios from above, we have evaluated the performance regarding:

- **Retrieving news feeds** We retrieved the news feeds for every aggregating node and normalized the time to obtain the number of news feeds an approach can generate on average per second. We also binned aggregating nodes together with their node degree. In this way, we give empirical proof of the fact that the retrieval rate of GRAPHITY is independent of the aggregating node's degree and the size of the network.
- **Index maintenance** We measure the time needed to add content nodes to the graph and to update all relevant index structures. This was also combined with adding or removing friendship edges. With this measurement, we want to give evidence that STOU will indeed outperform GRAPHITY on these operations.
- **Index build time** Starting from an existing network with ST, we measured how long it takes to build the STOU and the GRAPHITY index.

[5]http://neo4j.org/

Metric	RCL	STOU	GRAPHITY
Retrieval rate	77k	12k	12k
Index built time	5 057s	3.2s	15s
Storage	5.1 GB	0.057 GB	0.058 GB

TABLE IV
EVALUATION OF RCL, STOU AND GRAPHITY ON THE METALCON DATASET.

- **Storage space** To understand space complexity in practice, we looked at the size of the graph database.

All our experiment were run on a 24 core machine with 48GB of RAM. To overcome latency of harddisk and OS caching mechanisms, all data was processed on a RAM memory disk.

A. Storage Complexity of RCL

From the theoretical analysis, we can clearly expect the RCL approach to perform best with respect to the retrieval rate. At the same time, due to the high redundancy in the data, RCL has a horrendous space complexity. In fact, during our experiments we found out that it is not feasible to compute RCL on larger datasets.

To underline these findings, we have evaluated the retrieval rate, index built time, and storage space requirements of RCL in comparison to STOU and GRAPHITY on our smallest data set: Metalcon[6]. The results of this evaluation are shown in Table IV. Regarding the retrieval rate, RCL clearly outperforms STOU and GRAPHITY by a factor of 6. However, already the time necessary to compute the RCL index was two to three orders of magnitude slower. Also storage space explodes by a factor of 100. Note, that this space requirement involves only the content nodes of the graph structure and not the actual content substance; texts, pictures, audio or video data was not stored in the graph database.

B. Runtime of ST

In a preliminary experiment, we measured the time in order to calculate the rate of retrieved news feeds per second for one snapshot of Wikipedia for every year as shown in Figure 6. The ST baseline, which is equivalent to the relational data base approach with two joins, drops in speed very fast. On the 2011 Wikipedia data set only 132 newsfeeds per second could be retrieved with ST. STOU is already 64 times faster than ST and GRAPHITY performs even 142 times faster.

C. Retrieving News Feeds

1) Independence of the Node Degree: The node degree is the most influential parameter for the runtime of ST and STOU. We used equal sized bins to group articles of similar node degree $[0-10], [10-20], [20-30], \ldots$ together. Consecutively, we have randomly selected articles from each bin and we retrieved the news feeds for those articles to empirically evaluate the runtime performance.

[6]As the Metalcon dataset represents a snapshot of an online community, it does not provide any information on the dynamics of the network structure. Hence, we could not perform a realistic analysis of the index maintenance and updating performance in this case.

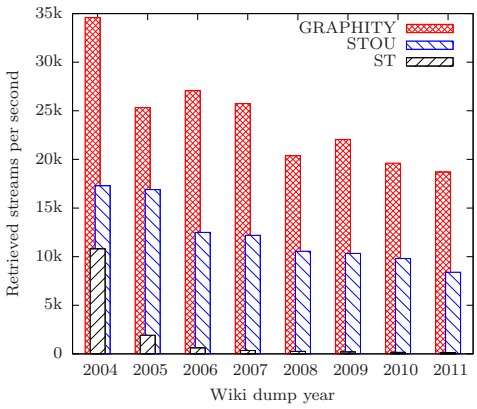


Fig. 6. Retrieved news feeds per second for data sets Wiki'04 - Wiki'11

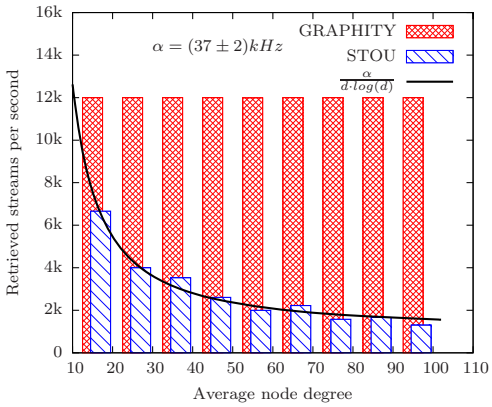


Fig. 7. Number of retrieved news feeds per seconds against node degree on the Wikipedia 2009 snapshot

From Figure 7, we can see that the rate of retrieved news feeds stays constant with GRAPHITY. This is the experimental evidence that GRAPHITY is indeed independent of the node degree and does therefore scale. The news feed rate of STOU on the other side drops as expected with a behavior proportional to $1/(d \log(d))$

2) Dependency on k for News Items Feed Retrieval: For our tests, we choose $k = 15$ for retrieving the news items feeds. In this section, we argue for the choice of this value for k and show the influence of selecting k with respect to the performance of retrieving the news feeds per second. On the Wikipedia 2009 snapshot, we retrieved the news item feeds for all aggregating nodes with a node degree $d > 10$ and varied k.

As can be seen in Figure 8, GRAPHITY's retrieval rate clearly depends on the choice of k. For a small k, STOU's retrieval rate is almost constant and sorting of ego networks (which is independent of k) is the dominant factor. With larger k, STOU's speed drops as both merging $O(k \log(k))$ and sorting $O(d \log(d))$ need to be conducted. The dashed line shows the interpolation of the measured frequency of retrieving the news feeds given the function $1/k \log(k)$ while

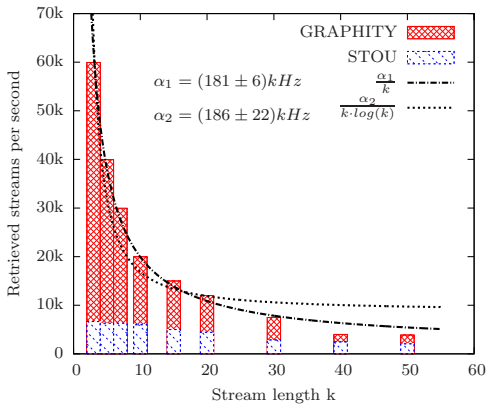


Fig. 8. Number of retrieved news feeds per seconds for different k's on the Wikipedia 2009 snapshot

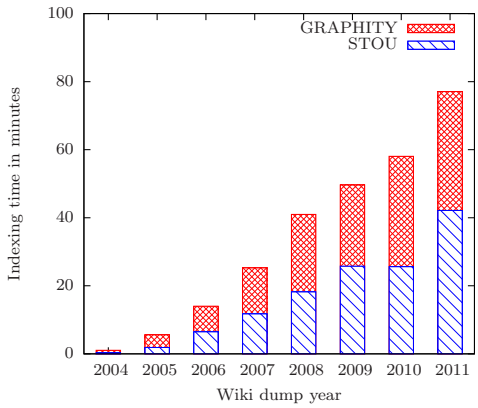


Fig. 10. Time to build the index with respect to network size

the dotted line is the interpolation based on the function $1/k$. As we can see, the dotted line is a better approximation to the actually measured values. This indicates that our theoretical estimation for the retrieval complexity of $k \log(k)$ is quite high compared to the empirically measured value which is close to k.

D. Index Maintenance

Here, we investigate the runtime of STOU and GRAPHITY in maintaining changes of the network as *follow* edges are added and removed as well as content nodes are created. We have evaluated this for the snapshots of Wikipedia from 2004 to 2008. For Metalcon, this data on social network evolution was not available. We simulated the events in the same order as they actually occurred in the Wikipedia history.

We see in Figure 9(a) that the number of updates the algorithms are able to handle drops as the data set grows. However, their relative speed up of STOU over GRAPHITY stays almost constant at a factor between 10 and 20. As the retrieval rate of GRAPHITY for big data sets stays with $12k$ retrieved news feeds per second, the update rate of the biggest data set is only about 170 updated GRAPHITY indices per second.

Figure 9(b) shows the rates for adding *follow* edges. This relative performance between STOU and GRAPHITY is about the same as for adding new content nodes. This makes perfect sense since both operations are linear in the node degree $O(d)$. Overall, STOU was expected to outperform GRAPHITY in this case since the complexity class of STOU for these tasks is $O(1)$.

As we can see from Figure 9(c) the rates for removing friendships are comparable, meaning that this task is in GRAPHITY as fast as in STOU. This is also as expected since the complexity class of this task is $O(1)$ for both algorithms.

E. Index Build Time

We have analyzed how long it takes to build the GRAPHITY and STOU index for a given, entire network. Both indices have been computed on a graph with existing *follow* relations. To compute the GRAPHITY and STOU indices, for every aggregating node a all content nodes are inserted to the linked list representation of $C(a)$. Subsequently, only for the GRAPHITY index for every aggregating node a the ego network is sorted by time in descending order. For both indices, we have measured the rates of processing the aggregating nodes per second as shown in Figure 10.

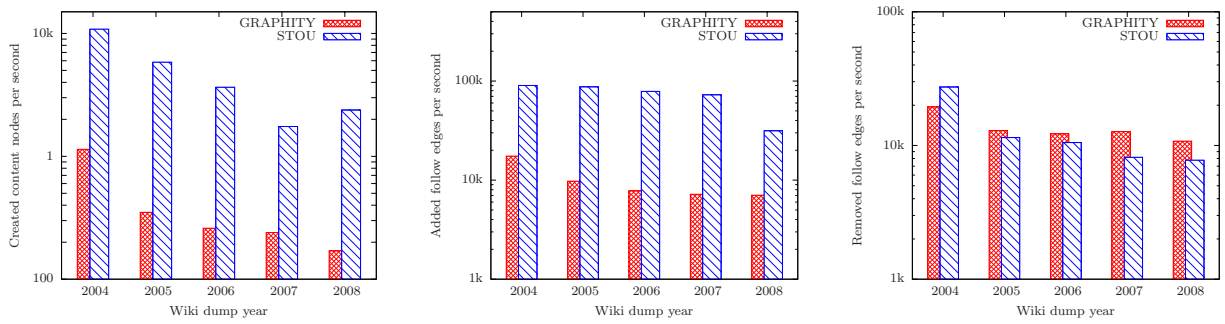
As can be seen from the figure, the time needed for computing the indices increases over time. This can be explained by the two steps of creating the indices: For the first step, the time needed for inserting content nodes increases as the average amount of content nodes per aggregating node grows over time. For the second step, the time for sorting increases as the size of the ego networks grows. Overall, we can say that for the largest Wikipedia data set from 2011, still a rate of indexing 433 nodes per second with GRAPHITY is possible. Creating the GRAPHITY index for the entire Wikipedia 2011 data set can be conducted in 77 minutes. The computing of the STOU index required 42 minutes.

YEAR	ST	RCL	STOU	GRAPHITY
Metalcon	0.057	5.1	0.057	0.058
2004	0.09	0.6	0.09	0.11
2005	0.35	10.7	0.35	0.51
2006	0.8	59*	0.8	1.2
2007	1.5	177*	1.5	1.9
2008	2.1	326*	2.1	2.8
2009	2.7	519*	2.7	3.6
2010	3.2	721*	3.2	4.4
2011	3.8	998*	3.8	5.3

TABLE V
STORAGE SPACE OF THE ADDITIONAL INDEXES FOR THE WIKIPEDIA DATA SET IN GB (VALUES INDICATED WITH * ARE EXTRAPOLATED).

F. Storage Space

Except for the update speed, operators of web platforms are also interested in operating costs such as resulting from data base sizes. Table V shows the additional storage space required for the indexes.



(a) Creating new content nodes (b) Adding *follow* edges (c) Removing *follow* edges

Fig. 9. Index update rates for various changes of the network.

We see that the data base for ST and STOU is the same size. This makes sense since nothing but the topology for the content nodes changes. The data bases for the RCL approach explode as expected. Later numbers are just extrapolated from network statistics. GRAPHITY is bigger that STOU but does not grow as fast as RCL.

VII. Related Work

The retrieval of top-k items is considered, e.g. in temporal relational databases where a single global index is created over all content items [4], [5]. While this index serves for general purpose, it makes the retrieval of top-k items from an ego network very expensive. Basically, all items of the entire index need to be retrieved and joined with the user's friendship relations to select the relevant news items. An overview of approaches for top-k query processing in relational databases as well as XML-based databases is provided by the extensive survey of Ilyas et al. [1]. The approaches are classified into a taxonomy of top-k query processing techniques along different dimensions like the underlying query model. One example is the query model of top-k join processing and is applied, e.g. by Ladwig et al. [6] when conducting keyword search in graph databases. In principle, this approach can be applied to find the top-k content items in social networks. However, in top-k join processing one is typically not interested in the temporal order in which the content items appeared. In addition, the search is conducted over the entire graph and not on a specific subgraph like one's ego network. Thus, top-k join processing for keyword search can be considered different to our problem of retrieving top-k content items from one's ego network and in the temporal order of the content items.

FeedEx [7] is a distributed archive system for news feeds based on a peer-to-peer infrastructure. It mainly addresses network issues such as a proper communication protocol and dynamic clustering of clients based on the similarity of subscribed news feeds. Goal is to save communication costs for the providers of news feeds by receiving content items from one's neighbors in FeedEx's peer-to-peer network rather from constantly polling the original news feed providers. Similar to FeedEx, Magnet [8] is a peer-to-peer network in the field of distributed event-based systems [9] that clusters the aggregating nodes based on similar subscriptions using distributed hash tables. Like FeedEx, the goal is to increase efficiency of the dissemination of content items and establishing the peer-to-peer network and maintaining it. The problem of clustering clients based on overlapping news feed subscriptions is similar to clustering users of similar ego networks. This might be a future extension of our graph models for top-k retrieval.

A user in a social network might be interested in incorporating feeds from other users on specific topics like music. These users, however, may or may not be members of one's ego network. This makes the problem of incorporating messages in one's news feed one of reachability and distance between nodes in the graph, where the nodes represent the users. Cohen et al. [10] present a data structure based on 2-hop covers that allows for an efficient processing of queries on graphs for determining the reachability and distance between two nodes. Broder et al. [11] present an approach considering reachability constraints in graphs in the context of publish/subscribe systems. The targeted application areas are web advertisement and news feeds in social networks represented as graphs. The news are filtered and ranked along different criteria like topical preference, social network distance, and popular users [2], novelty [12], or diversity [13], [14]. In contrast, we concentrate with STOU and GRAPHITY on the processing of content items from one's ego network, i.e. the users that are just one hop away. This problem is insofar different, as we are not concerned with questions of reachability and distance in graphs [10], [15], [16]. This information is already given by the ego network, i.e. the fact that all users are connected with oneself.

Finally, Hexastore [17] and Neo4j[7] aim at providing scalable general-purpose query processing on very large graphs. They do not explicitly consider the specific problem of efficiently retrieving the top-k content nodes from a social network graph as it is the aim here. However, the existing graph databases

[7]http://neo4j.org/

can be used as basis for implementing our graph models. In fact, Neo4j has been used to implement STOU and GRAPHITY and the RCL baseline (see Section VI).

VIII. CONCLUSION

Even though the RCL approach is the fastest in retrieving news items feeds, the high redundancy of data makes it impractical. In addition to the large storage overhead, it is unclear how to handle update anomalies that can occur due to the denormalization of the data model. Finally creating new content items requires the update of d indices. Our novel GRAPHITY approach has the same complexity class for its runtime without the need for redundancy in the data. Thus, GRAPHITY is preferable to RCL. In further analysis, we showed that GRAPHITY clearly outperforms STOU in retrieving news items feeds on our data sets by a factor of 3. However, our second contribution STOU performs better when adding content items to the network or the network changes its structure. The performance loss of GRAPHITY in this case depends on the average node degree of the network and has been shown empirically to be of an order of magnitude. From these observations we conclude that GRAPHITY is favourable in settings where significantly more retrieval tasks are expected compared to the write operations in order to have a improvement in performance. In our experiments, we observed a break even in performance between GRAPHITY and STOU when the retrieval of news streams happens approximately 3 times as often as the creation of new content items. If the creation of content items is more frequent or, alternatively, if node degrees are really small the STOU approach yields a better performance.

As for Metalcon, we are in a setting where users retrieve significantly more content than they produce. After an ad-hoc comparison to the currently used relational database technology[8], Metalcon has decided to migrate its data model towards a neo4j implementation of GRAPHITY. The code is open source and can be found at http://www.rene-pickhardt.de/graphity-source-code/. A demo of GRAPHITY on a small data set can be found at http://gwt.metalcon.de/GWT-Modelling/. While discussing the decision of migrating Metalcon to GRAPHITY with developers of other social networking sites, we came across specific scenarios where the developers stated that they would rather use STOU since in their network application the node degrees stay rather small and they could accept the slower retrieval rate for a faster writing process[9].

IX. FUTURE WORK AND EXTENSIONS

So far, we only worked on index structures for news items feeds based on a temporal ranking. Under this aspect, two natural extensions are the incorporation of a content based global relevance ranking or a personalized filtering of content items in a news feed. Conceptually there is little difference in using global relevance weights over the time of creation for sorting the content items. The management of the GRAPHITY or STOU index can still be achieved efficiently. However, a thorough empiric evaluation still needs to confirm the theoretic bounds. Filtering a news feed based on the personal preference of a user, instead, can be implemented on top of our indices. This implies a far higher value for k in order to retrieve more content items as basis for a filtering approach, which also yet needs to be evaluated in a real world scenario.

A second direction of future work is the scalability beyond a single machine. Given that the underlying graph data base technology Neo4j currently does not scale horizontally, this calls for modified and distributed versions of GRAPHITY and STOU. The naive approach of distributing the nodes equally over several instances and maintaining the relatively small index structures for each aggregating node locally seems promising. The framework for empirical proof has to be developed and tests have to be conducted.

[8]http://www.rene-pickhardt.de/time-lines-and-news-streams-neo4j-is-377-times-faster-than-mysql/

[9]http://neo4j.org/nabble/#nabble-td3477669 and http://www.rene-pickhardt.de/graphity

X. ACKNOWLEDGMENTS

The research leading to these results has received partial funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 257859, ROBUST and grant agreement no. 287975, SocialSensor. Thanks to Metalcon. Special thanks to Mattias Persson and Peter Neubauer from neotechnology.com and to the community on the Neo4j mailinglist for helpful advices on their technology and for providing a Neo4j fork that was able to store that many different relationship types. Thanks to Knut Schumach for coming up with the name GRAPHITY as well as Matthias Thimm and Leon Kastler for helpful discussions.

REFERENCES

[1] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Comput. Surv.*, vol. 40, no. 4, pp. 11:1–11:58, Oct. 2008.

[2] H. Li, Y. Tian, W.-C. Lee, C. L. Giles, and M.-C. Chen, "Personalized feed recommendation service for social networks," in *SocialCom/PASSAT*, 2010, pp. 96–103.

[3] B. Costales, *Sendmail, 3rd Edition*. O'Reilly Media Formats, 2009.

[4] D. Gao, S. Jensen, T. Snodgrass, and D. Soo, "Join operations in temporal databases," *VLDB*, vol. 14, pp. 2–29, March 2005.

[5] S. Wang and E. A. Rundensteiner, "Scalable stream join processing with expensive predicates: workload distribution and adaptation by time-slicing," in *EDBT*. ACM, 2009, pp. 299–310.

[6] G. Ladwig and T. Tran, "Index structures and top-k join algorithms for native keyword search databases," in *Proceedings of the 20th ACM international conference on Information and knowledge management*, ser. CIKM '11. New York, NY, USA: ACM, 2011, pp. 1505–1514.

[7] S. Jun and M. Ahamad, "Feedex: collaborative exchange of news feeds," in *Proceedings of the 15th international conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 113–122.

[8] S. Girdzijauskas, G. Chockler, Y. Vigfusson, Y. Tock, and R. Melamed, "Magnet: practical subscription clustering for internet-scale publish/subscribe," in *DEBS*. ACM, 2010, pp. 172–183.

[9] A. M. Hinze and A. Buchmann, *Principles and Applications of Distributed Event-Based Systems*. Idea Group Reference, 2010.

[10] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick, "Reachability and distance queries via 2-hop labels," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA '02. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2002, pp. 937–946.

[11] A. Broder, S. Das, M. Fontoura, and et al.a, "Efficiently evaluating graph constraints in content-based publish/subscribe," in *WWW*. ACM, 2011, pp. 497–506.
[12] E. Gabrilovich, S. T. Dumais, and E. Horvitz, "Newsjunkie: providing personalized newsfeeds via analysis of information novelty," in *WWW*. ACM, 2004, pp. 482–490.
[13] S. A. Munson, D. X. Zhou, and P. Resnick, "Sidelines: An algorithm for increasing diversity in news and opinion aggregators," in *ICWSM*. The AAAI Press, 2009.
[14] M. De Choudhury, S. Counts, and M. Czerwinski, "Identifying relevant social media content: leveraging information diversity and user cognition," in *Proceedings of the 22nd ACM conference on Hypertext and hypermedia*, ser. HT '11. New York, NY, USA: ACM, 2011, pp. 161–170.
[15] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry, "3-hop: a high-compression indexing scheme for reachability query," in *Proceedings of the 35th SIGMOD international conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 813–826.
[16] R. Schenkel, A. Theobald, and G. Weikum, "Efficient creation and incremental maintenance of the hopi index for complex xml document collections," in *Proceedings of the 21st International Conference on Data Engineering*, ser. ICDE '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 360–371.
[17] C. Weiss, P. Karras, and A. Bernstein, "Hexastore: sextuple indexing for semantic web data management," *PVLDB*, vol. 1, no. 1, pp. 1008–1019, 2008.

APPENDIX

In Section III-C, we relied on the ability to efficiently merge the lists of content items of several users connected by the GRAPHITY index structure. In this appendix, we provide the algorithm and proof its runtime complexity. Algorithm 1 shows the top-k n-way merge as pseudo code.

Algorithm 1 RetrieveFeed(a,k)

```
1:  Q ← ∅                                          // priority queue
2:  R ← ∅                                          // result list
3:  b_first ← SUCCESSOR(a, ego:a)
4:  c_first ← SUCCESSOR(b_first, update)
5:  R.ADD(c_first)
6:  Q.ADD(SUCCESSOR(c_first, update))
7:  b_latest ← SUCCESSOR(b_first, ego:a)
8:  c_latest ← SUCCESSOR(b_latest, update)
9:  Q.ADD(c_latest)
10: while ! Q.EMPTY() && SIZE(R) < k do
11:   c_mostRecent ← Q.POPFIRST()
12:   R.ADD(c_mostRecent)
13:   Q.ADD(SUCCESSOR(c_mostRecent, update)
14:   if c_mostRecent = c_latest then
15:     b_latest ← SUCCESSOR(b_latest, ego:a)
16:     c_latest ← SUCCESSOR(b_latest, update)
17:     Q.ADD(c_latest)
18:   end if
19: end while
20: return R
```

The algorithm starts in line 1 with initializing an empty priority queue Q to manage content nodes which still have to be considered. The content nodes in this queue are sorted in temporal order from the most recent to the oldest node. At any time, Q contains at most one content node from each aggregating node $b \in Ego(a_i)$. The list R in line 2 is used to store the result list $News_k(a)$ and is initially empty, too. By using the GRAPHITY index (i.e. the *ego*:a edges), we obtain the first aggregating node b_{first} of a in line 3. As the aggregating nodes in the GRAPHITY index are sorted in temporal order of their most recent content node, the first content node c_{first} of b_{first} (line 4) is also the first entry for our result list R (line 5).

For the further processing, we now add in line 6 the second most recent content node of b_{first} to the priority queue. In line 7 and 8 we also determine the next aggregating node from the GRAPHITY index and add its first content node to Q. Throughout the loop, we denote with b_{latest} the latest already considered aggregating node from the GRAPHITY index and its first content node with c_{latest}. Inside the loop, we pop the next content node $c_{mostRecent}$ from the priority queue, append it to the result list and add the next most recent content item produced by the same aggregating node to Q (lines 11 to 13). As c_{latest} is in the queue itself, we can be sure to consider only newer content nodes, until c_{latest} itself is retrieved from the queue. Likewise, we can be sure there is no need to look at further aggregating nodes from the GRAPHITY index, as their content nodes are at least not newer than the one of b_{latest}. In case $c_{mostRecent}$ coincides with c_{latest} (line 14), this guarantee does not hold any longer. Hence, we have to look in the GRAPHITY index at the next aggregating node and its first content node which then take the role of b_{latest} and c_{latest} (line 15 to 17). The loop ends as soon as we have considered all content nodes of all aggregating nodes in a's ego network (i.e. the queue is empty), or when $News_k(a)$ is complete, i.e. the result list contains k content nodes.

For a runtime analysis, we point out that for every iteration of the while loop a content node is added to the result list. Therefore, the while loop is iterated at most k times. Next we note that the queue will at most consist of $k+1$ elements. The most expensive operations within the loop are adding and removing elements from the queue. Both can be done in at most logarithmic time $O(\log(k))$ (e.g. using a heap data structure for the priority queue). In conclusion, the overall complexity is $O(k \log(k))$. Please note in particular that these operations neither depend on the network size n nor on the node degree d of a.

Further, by caching the state of the priority queue Q after retrieval it is even possible to retrieve additional l elements for incorporation into the news feed. Thereby, we can extend an existing news feed $News_k(a)$ with top-k content items to a $News_{k+l}(a)$ news items feed with the top-$(k+l)$ items at an additional cost of $O(l \times \log(k+l))$ instead of restarting from an empty result list with a runtime of $O((k+l)\log(k+l))$.