# Workload-Aware Indexing for Keyword Search in Social Networks

Truls A. Bjørklund
Norwegian University of Science and Technology
trulsamu@idi.ntnu.no

Michaela Götz and Johannes Gehrke
Cornell University
{goetz,johannes}@cs.cornell.edu

Nils Grimsmo
Norwegian University of Science and Technology
nilsgri@idi.ntnu.no

## ABSTRACT

More and more data is accumulated inside social networks. Keyword search provides a simple interface for exploring this content. However, a lot of the content is private, and a search system must enforce the privacy settings of the social network. In this paper, we present a workload-aware keyword search system with access control based on a social network. We make two technical contributions: (1) HeapUnion, a novel union operator that improves processing of search queries with access control by up to a factor of two compared to the best previous solution; and (2) highly accurate cost models that vary in sophistication and accuracy; these cost models provide input to an optimization algorithm that selects the most efficient organization of access control meta-data for a given workload. Our experimental results with real and synthetic data show that our approach outperforms previous work by up to a factor of three.

## General Terms

Performance, Security

## 1. INTRODUCTION

More and more data is accumulated inside social networks where users tweet, update their status, chat, post photos, comment on each other's lives, get updates through news feeds, and search for information. Examples of such social interaction platforms include Facebook, Twitter, LinkedIn, YouTube and Flickr. From a user's perspective, some of her content may be private and should only be accessible to a selected set of users in the network. To limit arbitrary information flow, social networks enable users to adjust their privacy settings at a fine granularity; e.g., to ensure that only friends can see the content they have posted. Thus any component that enables access to data in the social network *must* adhere to the privacy settings in place.

Search over collections of documents is a well-studied problem [37]. However, when supporting search over content in a social network, new opportunities and challenges arise. A document in a social network may be considered as consisting of two types of properties: *document-centric properties* and *network-centric properties*. The document-centric properties consist of the document and its metadata, for example the time when it was posted, the terms in the document, or properties of the user who posted the document. The network-centric properties consist of additional information added by other users, such as comments, tags, or ratings. The ranking of a document for a given search query could thus be based on both the document-centric and the network-centric part, where properties such as the relationship between the user who tagged a document and the user who submitted the query can be taken into account [1, 29].

When searching in social networks we need to enforce the access restrictions determined by the network structure, and we focus on the case where a user only has access to her friends' documents. This is a hard problem since as a result nearly every user has access to a unique subset of the documents and the resulting solution needs to scale not only with the number of documents but also the number of users.

In this paper, we propose to materialize special-purpose metadata called *author lists*, which are lists of identifiers for all documents authored by a set of users. When a user asks a search query, we use the author lists to filter the results of the query. This approach brings about two challenges. First, there is a large space of possible author lists, and we need to select which author lists to materialize to process a workload efficiently. Accurate cost models are required to identify a good set of lists. However, accurate cost models for query costs have received very limited attention in the literature as compared, for example, to cost models in database systems, probably because the space of possible query plans for a keyword search query is usually small compared to the space of possible query plans for an SQL query. To the best of our knowledge, this is the first paper that introduces highly accurate cost models for the operators in a search system. The second challenge is that a query may now need to access a very large number of author lists in order to determine the set of documents that a user has access to. Efficiently processing search queries with many author lists is essential for an efficient solution.

We focus on ranking functions based on simple document-centric properties in this paper, such as the time when the document was posted or the user who posted the document. This provides an important step towards full support for search in social networks with access control for two reasons: First, there are situations where it is natural to rank documents only on simple properties and thus the techniques proposed in this paper solve the whole problem. An example is the search for tweets on Twitter where recency is a well-understood and reasonable ranking function. Second, the technical contributions of this paper can be used as building blocks in a solution that uses more general ranking functions. All solutions need to enforce access control, and query processing operators and data structures that efficiently enforce access control are therefore important building blocks.

We make the following technical contributions:

- We introduce a new operator called HeapUnion which efficiently allows us to process a large number of author lists while skipping over irrelevant documents. Compared to previous approaches, HeapUnion improves query processing time of queries with access control by up to a factor of two. (Section 3)
- We provide highly accurate cost models of the query operators in a keyword search system. We also describe how our cost models interact with the solution space of the problem of selecting a good author list design for a particular workload. (Section 4)
- In a thorough experimental analysis, we evaluate the efficiency of our novel HeapUnion operator, validate the accuracy of our cost models, and test the optimized access designs. (Section 5)

We discuss related work in Section 6 and conclude in Section 7. We now continue with a more formal description of the problem we address in Section 2.

## 2. PROBLEM DEFINITION

In this section, we will introduce some notation which is used to define the problem addressed in this paper.

### 2.1 Data and Query Model

We view a social network as a directed graph $\langle V, E \rangle$, where each node $u \in V$ represents a user. There is an edge $\langle u, v \rangle \in E$ if user $v$ is a friend of user $u$, denoted $v \in F_u$, or alternatively that $u$ is one of $v$'s followers, denoted $u \in O_v$. We always have $u \in F_u$ and $u \in O_u$.

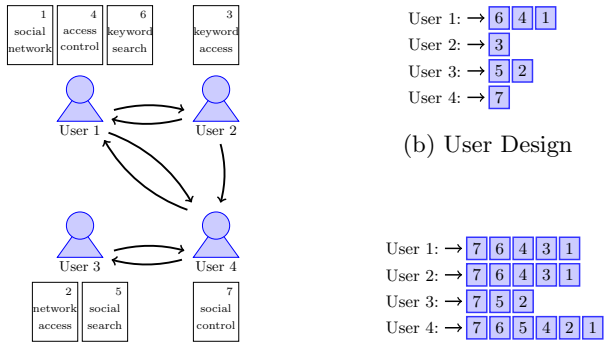
We consider workloads that consist of two different operations: posting new documents and issuing queries. A new document, which we also refer to as an *update*, consists of a set of terms. We will call the user who posted document $d$ the *author* of $d$, and we will also say that the user *authored* $d$. The new document gets assigned a unique document ID; more recently posted documents have higher IDs. Let $n_u$ denote the number of documents authored by user $u$, and let $N = \sum_u n_u$ denote the total number of documents.

A query submitted by a user $u$ consists of a set of keywords. As mentioned in the previous section, we assume that only documents authored by users in $F_u$ are accessible to $u$. For the remainder of this paper, we will also assume that the ranking is based on recency, with newer documents ranked higher than older documents. Thus the results of a keyword query are the $k$ documents that (1) contain the query keywords, (2) are authored by a user in $F_u$, and (3) have the $k$ highest document IDs among the documents that satisfy (1) and (2). Facebook currently supports these queries (with an artificial limit to posts of the last 30 days).[1] However, the technical details are proprietary.

Figure 1(a) shows an example of a social network with four users. User 4 is friends with Users 1 and 3, and User 2 is friends with Users 1 and 4. The users' posted documents are shown as well with their ID in the top right corner. User 3 has posted Documents 2 and 5, and in our model she can search through Documents 2, 5, and 7.

### 2.2 User and Friends Designs

In previous work we developed a conceptual framework of solutions to this problem based on two axes; the index axis and the access axis [12]. The *index axis* captures the idea that instead of creating one single inverted index over all the content in the social network, it is possible to create several inverted indexes, each containing a subset of the content. A set of inverted indexes and their content is called an *index design*. The *access axis* mirrors the index axis and describes the meta-data used to filter out inaccessible results; the meta-data is organized into *author lists*. As described in the introduction, an author list contains the IDs of all documents authored by a set of users. An *access design* describes a set of author lists.




(a) Example social network with posted documents

(b) User Design

(c) Friends Design

**Figure 1: Social Network and Basic Designs**

Our experiments with some basic points in the solution space showed that two of the most promising solutions both use an index design with a single index containing all users' documents, while the access design in the two approaches differ. The first approach is called *user design* and has one author list per user that contains the document IDs posted by that particular user. The second approach is called *friends design*; it also has one author list per user, but this author list contains the documents posted by the user and all of her friends. The author lists for the user and friends designs for our example from Figure 1(a) are shown in Figures 1(b) and 1(c), respectively. In both of these designs, a keyword query from a user is processed in the single inverted index. To enforce access control, the results from the index are intersected with a set of author lists containing all friends of the user. In the friends design, all friends of the user are represented in the author list for the user, whereas in the user design, we need to calculate the union of the author lists for all friends.

Note that in the user design, updates are efficient because only $u$'s author list is updated when $u$ posts a document; queries, however, need to access the author lists of all users in $F_u$. In the friends design, queries are more efficient because only the author list of $u$ is accessed. Updates, however, need to change the author lists of all users in $O_u$.

### 2.3 Beyond User and Friends Designs

The relative merits of the user and friends designs motivate the work in this paper. In our new hybrid approach, we start out with the user design. In addition, we add one additional author list $l_u$ for each user $u$; $l_u$ contains the IDs of all documents authored by a selected set of users $L_u \subseteq F_u$. When user $u$ submits a query, there is no need to access specific author lists for users in $L_u$, and queries therefore

[1] http://blog.facebook.com/blog.php?post=115469877130

become more efficient as more users are represented in $L_u$. On the other hand, representing more users in $L_u$ also leads to higher update costs. We therefore determine the contents of $L_u$ (and thus $l_u$) based on the workload characteristics.

## 2.4 System Model

A system implementing the strategy we propose in this paper contains both posting lists and author lists. Both of these types of lists are lists of document IDs, and the resulting index is thus a standard inverted index including author lists that are identical to posting lists in structure.

An updatable keyword search system is usually implemented with a hierarchy of indexes [24, 14, 21, 26]. New data is accumulated in a small updatable structure that also supports concurrent queries, while the main part of the hierarchy consists of a set of read-only indexes. The read-only indexes are formed by merging a set of smaller read-only indexes. During a merge, queries are still processed in the old indexes, and an atomic switch to the new indexes is performed once the merge is finished. Then, after all searches in the old indexes are finished, the old indexes can be deleted. Thus documents will be merged into larger and larger indexes over time, and the largest read-only indexes will contain the least recent documents. Such an index hierarchy is well suited for search in social networks, especially when used in combination with an access design that adapts to the workload. The time at which indexes are merged represents an opportunity to modify the access design and adapt it to the current workload, so that different indexes in the hierarchy potentially have different access designs. When ranking documents based on recency as we do in this paper, the largest indexes in the hierarchy (with the oldest documents) will probably be accessed less frequently than smaller indexes, and using different access designs among the indexes in a hierarchy might be beneficial in such settings.

All individual indexes in the hierarchy except the small updatable part process *stratified* workloads because the index is constructed before it is used to answer queries. Because the stratified workloads therefore dominate in the index system, we focus on such workloads in this paper. We have implemented a system that constructs an index for a set of documents, and then processes search queries with the index. Our system is main-memory based and accumulates an index for a batch of documents at a time in a structure where the lists are compressed using VByte [30]. The batches are combined in the end to form the complete index, where the lists are compressed using PForDelta [38, 36]. Next, we describe how queries are processed.

# 3. THE HEAPUNION OPERATOR

In this section, we describe how our solution supports efficient query processing that scales to a large number of author lists. Without loss of generality, our explanation assumes single-term queries to simplify the presentation; extensions to multi-term queries are straightforward and our implementation supports the general case.

## 3.1 Query Processing

Our search system answers queries by computing the intersection of a posting list $p_t$ for a term $t$ with a union of author lists $a_1 \cup \cdots \cup a_m$.[2] A template for the query plan

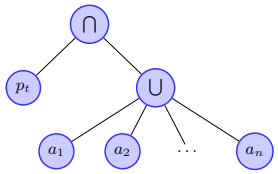



**Figure 2: Query Template**

is shown in Figure 2. It uses three operators (intersection, union, and list iterator) that all support the following interface:

- *Init*(): Initialize the operator;
- *Current*(): Retrieve the current result;
- *Next*(): Forward to the next result and retrieve it;
- *SkipTo*(*val*): Forward to the next result $\leq$ *val*.

All results are returned in sorted order based on descending document IDs to facilitate efficient ranking on recency, and the query plan follows a document-at-a-time processing strategy [34]. The top-$k$ ranked results are therefore found by calling *Next*() on the intersection operator $k$ times. We use a standard intersection operator that alternates between the inputs and skips forward to find a value that occurs in both [20]. With this solution, the *SkipTo*(*val*) operation in the union operator will be called repeatedly, and thus its implementation is essential to the overall processing efficiency. We will therefore focus on the union operator in the remainder of this section. We use standard techniques to implement the other operators [19, 38, 9, 20].

There exist two basic algorithms for the union operator, Eager Merge and No Merge [28]. Eager Merge merges all inputs to the union first, and then supports skipping in the intermediate merged list. The initial merge is the dominating cost when using Eager Merge. Assuming a standard multi-way merge strategy and $R$ entries in total in all $m$ input lists, the worst-case total merge cost of Eager Merge is $\Theta(R \log m)$. No Merge, on the other hand, processes a skip operation on the union by performing a *SkipTo*(*val*) in each input and returning the maximum result. Eager Merge and No Merge are thus preferable in different situations: If there are many skip operations compared to the total number of entries in the inputs, Eager Merge is preferable. On the other hand, No Merge is preferable when the number of skip operations is small compared to the input sizes.

Raman et al. have introduced a union operator called Lazy Merge, which is based on the idea that if the number of skip operations in the union is large compared to the length of an input, it would have been ideal to pre-merge this input into a set of intermediate results. Lazy Merge adaptively merges an input into the intermediate results when the number of skip operations exceeds the length of the input times a constant $\alpha$. Raman et al. show that Lazy Merge never uses more than twice the processing time of a solution that pre-merges the optimal set of inputs [28]. However, the approach has three drawbacks. First, although the analysis by Raman et al. argues that Lazy Merge adapts gracefully to all kinds of inputs, the analysis is, as the authors note, based on a simplifying approximation of the cost of merges. This approximation becomes increasingly inaccurate as the number of inputs and their sizes grow, and consequently, Lazy Merge does not scale well in practice. Second, the value of $\alpha$ has

[2] Similar types of queries occur in several scenarios, e.g., in star joins in data warehouses [28].

**Algorithm 1** HeapUnion Operator

```
1: function Init():
2:     Allocate heap

3: function Next():
4:     heap[0].Next()
5:     heapify(0)
6:     return Current()

7: function SkipTo(int val):
8:     Perform a breadth-first search in the heap from the root
9:     while BFS queue is not empty:
10:        if Current iterator is forwarded in SkipTo(val):
11:            Add to LIFO-list of entries for heap reorganization
12:            Add children to BFS queue
13:    Call heapify() for forwarded inputs in LIFO-list
14:    return Current()

15: function Current():
16:    if size(heap) == 0:
17:        return EOF
18:    else:
19:        return heap[0].Current()
```

significant impact on performance, but it is non-obvious how to select the "right" value for this parameter. Third, we process top-$k$ queries and therefore only need the first $k$ results from the intersection during query processing. Thus any method that pre-merges complete inputs, such as Lazy Merge, will have poor performance in our setting.

## 3.2 The HeapUnion Operator

HeapUnion, our novel union operator, is designed (1) to scale gracefully to a very large number of inputs regardless of the characteristics of the skip operations; (2) not to have any parameters whose values may be difficult to determine, and (3) to be efficient regardless of whether all or only a fraction of the results are actually needed. HeapUnion is based on a binary heap which contains one entry for each input operator and is ordered based on the value obtained from calling *Current*() on each input operator (referred to as the current value for the input operator), just as in a standard multi-way merge strategy.

We support the standard operator interface by always having the input with the highest current value at the top of the heap, so that this value is also the current value for HeapUnion. The heap is initialized the first time *Next*() or *SkipTo*(*val*) is called. When the first call is a *Next*() (*SkipTo*(*val*) resp.), HeapUnion calls *Next*() (*SkipTo*(*val*)) on all inputs, and the heap is constructed using Floyd's Algorithm [18]. Floyd's Algorithm calls a recursive subprocedure called *heapify*() which constructs a legal heap from an entry with two legal sub-heaps as children. The *heapify*() operation has logarithmic worst-case complexity in the size of the heap, and Floyd's Algorithm runs in linear time [18]. We will also use *heapify*() for heap maintenance.

After initialization, HeapUnion works as shown in the pseudo code in Algorithm 1. The *Current*() operation either returns the current value from the input operator at the top of the heap, or it indicates that there are no more results if the heap is empty. The *Next*() operation forwards the input with the current value, and calls *heapify*() to ensure that the input with the new highest current value is at the top of the heap. The worst-case complexity of this operation is thus logarithmic in the number of input operators.

The most novel aspect of HeapUnion, however, is the very efficient *SkipTo*(*val*) operation that is based on a breadth-first search (BFS) in the heap. When forwarding to a value *val*, only the inputs with a current value $>$ *val* actually need to be forwarded. Because the heap is organized according to the current value for all inputs, we know that if a given input has a current value $\leq$ *val*, the same is true for all of its descendants. If we determine that no skip is necessary in a given input, we thus also know that no skip is required in any of its children in the heap, and there is no need to process the children in the BFS. Furthermore, if an input is not forwarded, we know that its position in the heap relative to its children will not change. We also take advantage of this observation by calling *heapify*() only for the inputs where an actual skip occurred, and use a complete run of Floyd's algorithm only in the worst-case.

To illustrate how skipping in HeapUnion performs compared to both Lazy Merge and the basic union implementations described above, we will now present an example where HeapUnion outperforms all of them. Throughout the example, we assume that when merging a set of inputs, a standard multi-way merge is used. Furthermore, both the inputs and the pre-merged values are assumed to be represented as lists that support skipping forward with a cost logarithmic in the number of skipped entries. These assumptions agree with the implementations used in Section 5.

EXAMPLE 1. *Consider a union operator with $k+1$ inputs. The highest value in any input list is $(k+\alpha)\cdot l$, where $\alpha$ denotes the same constant that is used in Lazy Merge and $l$ is a chosen constant. The first input contains the first $\alpha \cdot l$ values starting from the highest value. The rest of the values are partitioned equally among the rest of the inputs such that input $i$ has the values $\{i-1+k\cdot j | 0 \leq j < l\}$.*

*We will now evaluate the cost of a set of skip operations that can be partitioned into two phases: In the first phase, we skip to each of the $\alpha l$ highest values in descending order. All these values are found in the first input. In the next phase, we perform a single skip to the value* 1. *We will evaluate the cost of both phases for both Eager Merge, No Merge, Lazy Merge and HeapUnion.*

***Eager Merge**: For Eager Merge, we will analyze the cost of the merges for the two phases separately. The merge of Phase* 1 *will only involve accessing one input list, and thus have a constant cost per entry. In Phase* 2*, each input will move to a leaf in the heap once a value has been extracted in the merge, and the cost is therefore $\Theta(k\cdot l\cdot \log(k))$. The costs of the actual skip operations in this example are $\Theta(\alpha\cdot l)$ for the $\alpha\cdot l$ skips of length* 1 *in Phase* 1*, and $\Theta(\log(k\cdot l))$ for the single skip in Phase* 2*.*

***No Merge**: In Phase* 1*, each skip operation with No Merge will result in a skip of length* 1 *in input* 1*, and an attempt at skipping in all other input lists, with total cost $\Theta(k\alpha l)$. Phase* 2 *has cost in $\Theta(k\log l)$ due to $k$ skips of length $l$.*

***Lazy Merge**: Because the $\alpha$ in the example is the one used to configure Lazy Merge, this method will behave as No Merge in Phase* 1*. The remaining $k$ inputs are merged before Phase* 2*, resulting in the same cost as Eager Merge in this phase.*

***HeapUnion**: HeapUnion will behave like a standard multi-way merge in Phase* 1*, and thus have the same cost as Eager Merge. In Phase* 2 *on the other hand, its costs are similar to with No Merge because the same skips are performed in each list, and the heap maintenance costs are in $O(k)$.*

*In this example, Eager Merge is clearly preferable in Phase* 1

*as compared to No Merge with a speed-up factor of $k$. In Phase 2 on the other hand, No Merge is clearly preferable with a speed-up factor $\frac{l \log(k)}{\log(l)}$ compared to Eager Merge. Lazy Merge actually achieves the worst of both worlds in the two phases. HeapUnion, however, outperforms Lazy and No Merge by a factor $k$ in Phase 1, and it outperforms Lazy and Eaper Merge by a factor $\frac{l \log(k)}{\log(l)}$ in Phase 2. HeapUnion thus scales much better than the other methods with increasing $l$ and $k$.*

We will now compare the worst-case performance of HeapUnion with the worst-case of Eager Merge and No Merge. First, we provide a bound on the combined complexity of all skip operations in one HeapUnion:

LEMMA 1. *Assuming that the cost of skipping forward $s$ entries in an input is in $O(s)$ and that each skip operation on HeapUnion forwards at least one input, the total cost of all skip operations for a HeapUnion is in $O(R \log m)$.*

We omit the proof due to space constrains. As a consequence of Lemma 1, the worst-case cost for all skip operations in HeapUnion is no worse than the worst-case cost of all skips in Eager Merge because the initial merge in Eager Merge has worst-case complexity $\Theta(R \log m)$.

Furthermore, the following lemma follows immediately from the fact that the heap maintenance reduces to Floyd's algorithm in the worst-case and that Floyd's algorithm is $O(m)$ for $m$ heap entries.

LEMMA 2. *The heap maintenance cost in one skip operation in HeapUnion is $O(m)$.*

As a consequence of Lemma 2, the cost of a specific skip operation with HeapUnion is comparable to the cost of the same skip operation with No Merge: In both methods, the same skips will occur on the inputs, and No Merge may potentially try to skip on more inputs that are not forwarded compared to HeapUnion (due to the BFS). Furthermore, the cost of finding the largest return value with No Merge is $\Theta(m)$, and the cost of heap maintenance in HeapUnion is $O(m)$. The worst-case cost of one skip operation in HeapUnion is therefore not worse than with No Merge.

HeapUnion will thus achieve the best of both worlds: When there are only a few skip operations compared to the lengths of the inputs, it is an advantage that its worst-case performance for each operation is as good as with No Merge. And, when there are many skip operations, it is an advantage that its worst-case performance is as good as with Eager Merge. In addition, HeapUnion does not pre-merge any inputs, and retrieving only a fraction of the results is therefore supported efficiently. Compared to Lazy Merge it also has the advantage that no configuration parameters are required. An experimental validation of the efficiency of HeapUnion is presented in Section 5.2.

# 4. COST MODELS AND OPTIMIZATION

The efficiency of our system for a particular workload depends on the contents of the additional author lists, and selecting a good set of lists is therefore essential. For each user $u$, any subset of $F_u$ can be included in $L_u$, which leads to $2^{\sum_{u \in V} |F_u|} = 2^{|E|}$ possible designs. We use cost models to explore this large space. In information retrieval, cost models have traditionally been used to explain and compare the relative merits of different algorithms [35, 34, 14, 16]. In this paper, however, the cost models are used in optimization algorithms to select between different access designs for a stratified workload of updates and queries.

| | Update Cost ($\|l_u\| = n$) |
|---|---|
| Monotonic | $c_{update} n$ |
| Non-monotonic | $\begin{cases} c_{1b} * n & \text{if } \frac{N}{n} < b_1 \\ c_{2b} * n & \text{if } b_1 \leq \frac{N}{n} < b_2 \\ c_{3b} * n & \text{otherwise} \end{cases}$ |

**Figure 3: Overview of Update Cost Estimates**

It turns out that accurate cost models are required in order to find close-to-optimal access designs, and as we will show in Section 5, adaptations of cost models used in related work are not accurate enough for our purposes. However, accuracy is not the only factor when choosing a cost model, because an advantage of simpler models is that they can allow analytically reducing the number of potentially optimal solutions, i.e., the search space of an optimization algorithm.

In the following subsections, we will introduce two accurate cost models. With the first model, *Monotonic*, we try to achieve the best possible accuracy while ensuring that the search space of the optimization algorithm is limited. *Non-monotonic* increases the accuracy further at the expense of an increase in the size of the search space. Both *Monotonic* and *Non-monotonic* model the processing time of the individual query operators, and they can therefore be employed in other applications that attempt to estimate the processing time of different query plans.

## 4.1 Monotonic

Monotonic is designed to be an accurate yet tractable cost model, where only a small number of access designs must be checked in the optimization algorithm to find a globally optimal solution. The model for updates in Monotonic simply assumes that the cost of constructing a list is linear in the number of document IDs in the list. For modelling query costs, Monotonic has one cost model for each operator and the total query costs are estimated by combining the models for all operators in the query. The cost model for an operator describes the cost of each method supported by the operator. For operators that have other operators as inputs, like HeapUnion and Intersection, the cost is calculated by combining the cost of operations within the operator with the cost of method calls on the inputs. To find the cost of the queries we use in this paper, we combine the operators according to the template in Figure 2, and calculate the cost of $k$ *Next*() calls for the Intersection to retrieve the top-$k$ results (assuming there are at least $k$).

Monotonic is described in Figures 3 and 4. *Skip*($s$) is a model for *SkipTo*(*val*). If *SkipTo*(*val*) forwards the current value of an operator with $\Delta v$ document IDs, we model the cost of the operation by *Skip*($\frac{r \Delta v}{N}$), where $r$ is the number of results of the operator. The number of results for an operator is the number of times we can call *Next*() and retrieve a new result. Monotonic and Non-monotonic use the same models for List Iterator and Intersection, but they have different models for HeapUnion. We will now explain Monotonic's model for HeapUnion; models for the other operators are explained in the full version of this paper along with the microbenchmarks used to determine the constants in the model [11].

**HeapUnion.** Let us assume that HeapUnion has $m$ inputs $k_1, \ldots, k_m$. We use $r_i$ to denote the number of results from input $i$, and define $R = \sum_{i=1}^{m} r_i$. We assume that the cost of initialization within the HeapUnion operator itself is neg-

| | *Init*() | *Next*() | *Skip*(*s*) |
|---|---|---|---|
| List Iterator | $\begin{cases} c_{einit} & \text{empty list} \\ c_{init} & \text{otherwise} \end{cases}$ | $c_{next}$ | $\begin{cases} c_c + \frac{s}{b}c_d + scan(s)c_{sc} & \text{if } s \leq b \\ Skip(b) + c_g \log\left(\frac{s}{b}\right) & \text{otherwise} \end{cases}$ |
| Intersection | $k_1.Init() + k_2.Init()$ | $k_1.Next() + (t-1)k_1.Skip(1 + \frac{r_1}{r_2})$ $+t \cdot k_2.Skip(\frac{t-1}{t}(1 + \frac{r_2}{r_1}) + \frac{r_2}{r_1 t})$ | not used |
| Monotonic HeapUnion | $\sum_{i=1}^{m} k_i.Init()$ | $\begin{cases} \sum_{i=1}^{m} k_i.Next() + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \frac{r_i}{R} k_i.Next() + (\gamma + 1) \cdot c_h & \text{otherwise} \end{cases}$ | $\begin{cases} \sum_{i=1}^{m} k_i.Skip(\frac{r_i s}{R}) + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \min(1, \frac{r_i s}{R}) k_i.Skip(\max(1, \frac{r_i s}{R})) & \\ \quad +(\gamma + 1) \cdot m_s \cdot c_h & \text{otherwise} \end{cases}$ |
| Non-monotonic HeapUnion | $\sum_{i=1}^{m} k_i.Init()$ | $\begin{cases} \sum_{i=1}^{m} k_i.Next() + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \frac{r_i}{R}(k_i.Next() + \log(1 + p_i) \cdot c_h) & \text{otherwise} \end{cases}$ | $\begin{cases} \sum_{i=1}^{m} k_i.Skip(\frac{r_i s}{R}) + (\gamma + \frac{1}{2}) \cdot m \cdot c_h & \text{first call} \\ \sum_{i=1}^{m} \min(1, \frac{r_i s}{R}) k_i.Skip(\max(1, \frac{r_i s}{R})) & \\ \quad + \max(m_s \gamma + \min(m_s, \frac{m}{2}), & \\ \quad m_s(\sum_{j=1}^{m} \frac{r_j}{R} \log(1 + p_j)) - h(\lfloor m_s \rfloor)) \cdot c_h & \text{otherwise} \end{cases}$ |

**Figure 4: Overview of Query Cost Estimates**

ligible, and therefore model the cost of initialization as the sum of the initialization costs for all inputs.

Recall from Section 3.2 that the first call to either *Next*() or *SkipTo*(*val*) will involve construction of the heap; we therefore have two different cases in the model for *Next*() and *SkipTo*(*val*), depending on whether it is the first or a subsequent call. The cost of the first call to *Next*() includes the cost of calling *Next*() on all $m$ inputs, and the cost of the heap construction using Floyd's Algorithm. For heap construction, we model the cost of each call to *heapify*() as being constant, $c_h$. With Floyd's algorithm, *heapify*() is called for half the heap entries, and then recursively every time there is a reorganization. The average case complexity of Floyd's algorithm is well known, and the number of relocations in the heap is approximately $\gamma m = 0.744m$ [33]. Thus we model the cost of heap construction as $(\gamma + \frac{1}{2}) \cdot m \cdot c_h$.

A first call to *SkipTo*(*val*) involves skipping in all inputs, in addition to heap construction. Given that $s$ results are skipped in this operation, we simply assume that the number of entries skipped in input $i$ is proportional to its length, i.e. it is $\frac{r_i s}{R}$, resulting in a cost of $\sum_{i=1}^{m} k_i.Skip(\frac{r_i s}{R})$. The cost of heap construction is modeled as explained for *Next*() above.

Subsequent calls to *Next*() involve a call to *Next*() for the input at the top of the heap and heap reorganization. We estimate the cost of the call to *Next*() for the input at the top as a weighted average over the inputs. The model for the cost of heap maintenance is simple. We assume that there will be $\gamma$ relocations when a single operator is forwarded, leading to $\gamma + 1$ calls to *heapify*().

Subsequent calls to *SkipTo*(*val*) will potentially forward all inputs, and then reorganize the heap according to the new current values of the inputs. On average, each input will be forwarded past $\frac{r_i s}{R}$ entries. However, HeapUnion will ensure not to call *SkipTo*(*val*) for inputs that will not be forwarded. Therefore, when the average skip length for an input is less than 1, we model the cost as $\frac{r_i s}{R}$ calls that skip 1 entry. To find the cost of heap maintenance we estimate the number of forwarded inputs as: $m_s = \sum_{i=1}^{m} \min(\frac{s r_i}{R}, 1)$. Assuming that there will be as many relocations in the heap as when constructing a heap with $m_s$ entries, the cost of heap maintenance is modeled as $(\gamma + 1) \cdot m_s \cdot c_h$.

**Optimization Algorithm.** We will now show in two steps that an optimization algorithm that uses Monotonic as the cost model only has to test $|E|$ different access designs in order to find the optimal solution. First, we show in the following lemma that we can find a globally optimal solution by choosing the contents of the additional author list for each user individually; it follows directly from the definitions in Figures 3 and 4.

LEMMA 3. *When using Monotonic as a cost model, the cost of including a user $v_1$ in $L_{u_1}$ is independent of the cost of including a user $v_2$ in $L_{u_2}$ for $u_1 \neq u_2$.*

Lemma 3 reduces the number of access designs to test in the optimization algorithm from $2^{\sum_{u \in V} |F_u|}$ to $\sum_{u \in V} 2^{|F_u|}$. The following theorem shows that we can reduce the size of the search space further under certain conditions.

THEOREM 4. *If Monotonic estimates that for a user $u$ and a given workload, the performance is improved if $v \in F_u$ is included in $L_u$, then Monotonic will predict that it leads to a performance improvement to also include user $w$ in $L_u$ if $w \in F_u$, $0 < n_w < n_v$, and $c_d - \frac{c_{sc}}{2b} \geq \frac{c_g}{\ln 2}$.*

The proof of Theorem 4 is found in the full version of this paper [11]. We notice that in our case, $c_d - \frac{c_{sc}}{2b} \geq \frac{c_g}{\ln 2}$ translates to $0.331 \geq 0.114$, which holds with a significant margin. Theorem 4 implies that if we sort all friends of $u$ based on the number of documents they post, the optimal contents of $L_u$ is a prefix of this sorted list. We thus need to check only $|E|$ designs in total in order to find the optimal solution. Furthermore, notice that there is no cost associated with including users who do not post documents in the additional author lists, and it is therefore always beneficial to do so.

## 4.2 Non-monotonic

Non-monotonic is designed to be more accurate than Monotonic; however, it sacrifices the nice property that only a small number of designs must be checked to find the globally optimal access design. To achieve better accuracy, we improve the model for heap maintenance costs, and we use a slightly more advanced update model. The formal description of Non-monotonic is found in Figures 3 and 4.

The update model from Monotonic is extended by taking list compression into account. During accumulation, the lists are compressed using VByte, which implies that lists with few entries result in long deltas that use more space. The model assumes that the cost of updating a list depends on the number of bytes used by VByte to represent the average delta length.

The models for heap maintenance in subsequent calls to *Next*() and *SkipTo*(*val*) reflect that the cost of heap maintenance often depends on the total number of inputs as well as on the number of forwarded inputs. Given that input $i$

is at the top of the heap when *Next*() is called, let $p_i$ denote the number of inputs that will have *Current*() values larger than input $i$ after the call to *Next*(). We estimate $p_i$ as $\sum_{k=1}^{m} \min(\frac{r_k}{r_i}, 1)$. [3] We replace the rough heap maintenance cost estimate of $(\gamma + 1)c_h$ with $\log(1 + p_i)c_h$ when input $i$ is at the top of the heap. By calculating a weighted average over all inputs, we end up with an average cost of heap maintenance for a *Next*() as shown in Figure 4.

The model for heap maintenance in *SkipTo*(*val*) is slightly more complex, and the maximum of two different estimates is used: (1) The first alternative is similar to the estimate in Monotonic, but incorporates that Floyd's algorithm will never call *heapify*() for more than half the inputs, which yields the following estimate: $(\gamma m_s + \min(m_s, \frac{m}{2}))c_h$. (2) The other alternative reflects that the cost can be logarithmic in the number of inputs when the number of forwarded inputs is low. We have already estimated the average number of calls to *heapify*() when only one input is forwarded in the model for *Next*(), denoted $h_{next}$ in the following. We now assume that all forwarded inputs will lead to $h_{next}$ *heapify*() operations, but compensate for the fact that many of the inputs are not at the top of the heap when *heapify*() is called. The compensation is achieved with the function $h(m_s)$ in Figure 4, which returns the minimum possible total distance from the root to $m_s$ entries in the heap.

**Optimization Algorithm.** Lemma 3 also holds for Non-monotonic. However, the proof of Theorem 4 does not hold due to the extensions in Non-monotonic. As a result, an optimization algorithm that uses Non-monotonic must test $\sum_{u \in V} 2^{|F_u|}$ access designs to find the optimal approach. In social networks, users have hundreds of friends, and it is therefore not feasible to test such a large space of access designs. We thus choose to limit the space to the same space that the optimization algorithm explores for Monotonic. If Non-monotonic is actually more accurate than Monotonic, the resulting design should be at least as efficient.

# 5. EXPERIMENTS

We will now present experiments evaluating the efficiency of our novel HeapUnion operator, validate the accuracy of our cost models, and test the optimized access designs.

## 5.1 Experimental Setup

**Workloads.** In our experiments we are using a set of workloads that are based on a crawl of a subset of Twitter from February 2010. The first workload is based on the actual crawled Twitter network and is denoted Workload Real. The workload consists of 417,156 users with 74,326,889 unique friendships, and their 2,500,000 most recently posted documents. Due to a restriction in the Twitter API, none of the users have more than 200 posted documents in this workload. We have also generated two workloads with synthetic networks, Workload 1 and Workload 2, to enable to test our solutions with varying social network characteristics. Workload 1 has 10,000 users and Workload 2 has 100,000. In both networks, users have 100 friends each and the friendships are generated with the widely used preferential attachment model [6]. Documents in both workloads were obtained from the Twitter crawl. In both workloads, each user is assigned a posting frequency from a Twitter user, and the documents are assigned to users according to the resulting distribution. In Workload 1, we pick the posting frequency for a user dependent on the user's number of followers, which creates a strong correlation between the number of followers and the number of posted documents. We assign 1,500,000 documents accordingly. In Workload 2 we pick a frequency for a user independent of the number of followers and assign 2,500,000 documents accordingly.

The workloads also involve search queries. As we do not have access to actual search logs for the crawled data, we generated the search queries based on the actual document collection. The query terms were selected by removing all the stop words in the collection, and then choose a random remaining term. We thus select query terms based on term frequency except for stop words. We use two different strategies for selecting the user who submits the search query. We either use a uniform distribution such that each user is selected with equal probability, or a Zipfian distribution with exponent 1.5 such that a few users are selected much more frequently than others. In our plots, the workloads using the Zipfian distribution have "Z" as a suffix. Different complete workloads are generated by combining the 3 basic workloads above with 100,000 subsequent queries (with the exception of the last experiment). We return the top-100 results for all queries unless explicitly stated otherwise.

**Hardware.** All experiments were run on a computer with an Intel Xeon 3.2 GHz CPU with 6 MB cache and 16 GB main memory running RedHat Enterprise 5.3; our system is implemented in Java, and we ran Java 1.6.

## 5.2 Performance of HeapUnion Operator

We compare the relative efficiency of HeapUnion and Lazy Merge [28] by exchanging the HeapUnion operator in our query template with Lazy Merge. Our implementation of Lazy Merge stores the intermediate results in an uncompressed list where skips are implemented with a galloping search [9]. The parameter $\alpha$ describes how eager Lazy Merge is at merging inputs into the intermediate results. When setting $\alpha$ to 0, Lazy Merge behaves as Eager Merge, and with $\alpha = \infty$, Lazy Merge behaves as No Merge.

In the experiments with HeapUnion, we limit the access design to the user design because it provides a real test of any solutions' ability to process queries with many author lists efficiently. We report the query processing time for each of the workloads, and vary $\alpha$ between 0 and $\infty$. We have argued that one of the reasons why Lazy Merge is not ideal for our workloads is that we typically process top-$k$ queries. To isolate this effect, we experiment both with only returning the top-100 results and with returning all results.

The results from the experiments are shown in Figure 5. Notice that HeapUnion does not depend on $\alpha$, and its cost is therefore constant. When using Lazy Merge, the difference between the cost of top-100 queries and retrieving all results increases with the size of $\alpha$. This reflects the inadequacy of approaches that pre-merge when processing top-$k$ queries. The processing time of HeapUnion is clearly dependent on the number of retrieved results, and HeapUnion is therefore an attractive solution for top-$k$ queries as expected.

We observe that Workload 1 incurs the slowest performances across workloads. In Workload 1 users have access to more posts on average. This is particularly challenging to LazyMerge as it merges some of these longer lists. Lazy

[3] $\frac{r_k}{r_i}$ is the average number of *Next*() calls in input $k$ for each *Next*() call in input $i$. If both inputs each had values at equal distances this estimation would be exact.

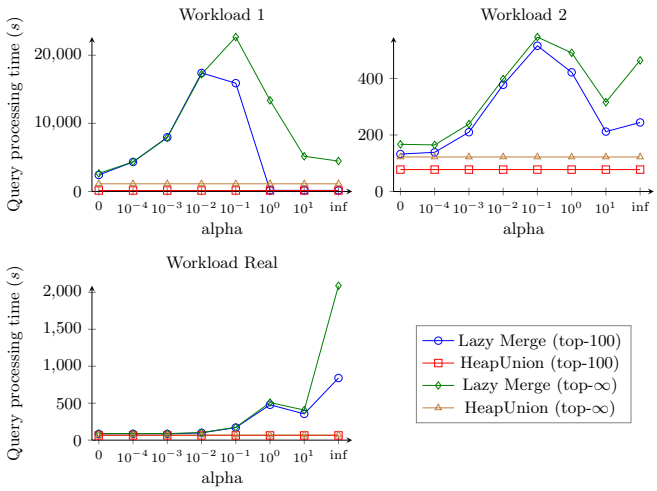



**Figure 5: HeapUnion vs. Lazy Merge varying $\alpha$.**

Merge performs best with $\alpha$ set to extreme values. Poor performance otherwise is often caused by the large number of merges resulting from many inputs with different lengths. In what end of the scale the best $\alpha$ value is found for a particular workload depends on the average length of the author lists compared to the posting list. HeapUnion outperforms the best configurations of Lazy Merge in all workloads with a speed-up between 1.12 and 2.36, reflecting that HeapUnion is efficient regardless of workload characteristics.

## 5.3 Validation of Cost Models

To test the accuracy of our cost models, we compare their predictions to the actual running times in our system for Workloads 1 and 2. We also need to define a set of access designs to test so that the results will be indicative of to which extent using the different cost models in an optimization algorithm will lead to efficient access designs. Here we make the observation that the optimization algorithms associated with all our cost models will select a limit for each user, $u$, such that all friends of $u$ who post fewer documents than this limit will be included in $L_u$. For testing purposes, we choose a single *limit* per access design, such that all users will include a friend $v$ in their $L_u$ if the number of documents $n_v$ posted by user $v$ satisfies $n_v < limit$. By choosing a set of different values for *limit*, we obtain a set of designs ranging from empty $L_u$s to $L_u$s that include all friends.

As a basis for comparison in our experiments, we use a straight-forward cost model based on the one introduced by Silberstein et al. for the problem of finding the most recent posts to compose user event feeds in social networks [31]. We refer to this model as *Simple*; it assumes that the cost of processing a query is linear in the number of accessed author lists, and that the cost of constructing a list is linear in the number of document IDs in the list.

Figure 6 compares the actual running times of our system to predictions from both Monotonic and Non-monotonic as well as for Simple. As we can see, the query cost estimates with Monotonic are much more accurate than the estimates with Simple, but there is still room for improvement when *limit* is close to 0 in Workload 2. Inaccurate modeling of the cost of heap maintenance is one factor that contributes to this error, and the accuracy of Non-monotonic is therefore higher. The extended update model in Non-monotonic also leads to a slight improvement in terms of accuracy.

## 5.4 Workload-Aware Designs

We have tested the accuracy of the different cost models above, but the key success factor for a cost model is whether using it in an optimization algorithm leads to efficient designs. We therefore conducted a set of experiments to compare the access designs suggested by the different cost models and associated optimization algorithms. To do so, we use the basic workloads in Section 5.1, and combine them with different numbers of queries to vary the ratio of queries to updates in the workload. We compare the performance of the designs based on Simple, Monotonic and Non-Monotonic to the user design and the friends design.

The results of our experiments are shown in Figure 7, where the first three columns in the first line show the results for workloads where the queries are uniformly distributed among the users. To get a better view of the relative differences between the methods, the second line shows the performance of the approaches relative to the best of the user design and the friends design.

For the workloads with uniformly distributed queries, Simple often leads to designs that are slower than choosing the best of the user design and the friends design due to the prediction inaccuracies. Compared to Monotonic and Non-monotonic, the designs from Simple are up to 67% slower, a difference that occurs in Workload Real. Monotonic and Non-monotonic generally lead to reasonable designs that are comparable to or faster than the basic approaches. However, for Workload 2, both approaches lead to sub-optimal designs when queries are frequent relative to updates. This reflects the inaccuracies in the estimates for low limits for Workload 2 in Figure 6. However, Non-monotonic clearly results in better designs than Monotonic in this case; the designs from Monotonic are up to 12% slower, so the additional complexity pays off.

The results from Workload 2 with Zipfian queries are shown in the last column in Figure 7, denoted Workload 2Z. We have omitted the results for other workloads with Zipfian queries because the same patterns are observed. The results show that our overall solution performs much better compared to the basic designs when there is skew, with a speed-up of up to 3.4. With skew, the optimization problem is simple because a few users submit almost all queries, and all cost models are able to reflect that these users should have additional author lists with nearly all their friends represented. Non-monotonic is still slightly better than the others, but the difference is small.

In summary, these experiments confirm the benefits of using our more accurate cost models and associated optimization algorithms; we are able to find access designs that result in significantly better performance than the basic approaches from previous work.

# 6. RELATED WORK

Due to its clear commercial value there has recently been significant interest in search in social networks, and Bing for example supports real-time search over public Twitter posts. Unlike Google+, Facebook allows users to search their friends' posts. However, the details of these commercial solutions have not been published.

Several recent papers also address search in social net-

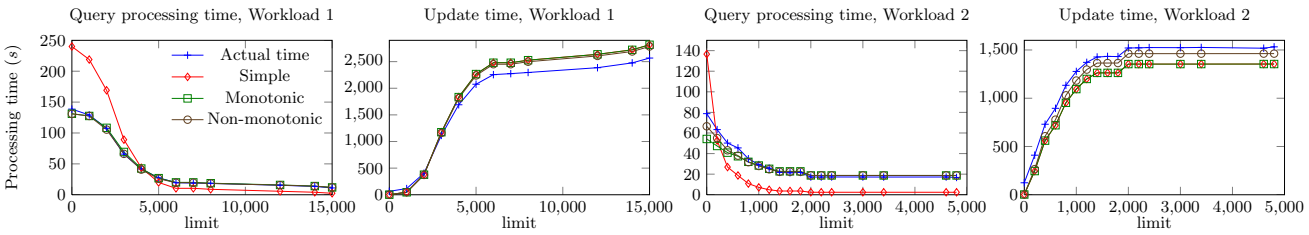



**Figure 6: Accuracy of cost models for queries and updates in Workload 1 and Workload 2**

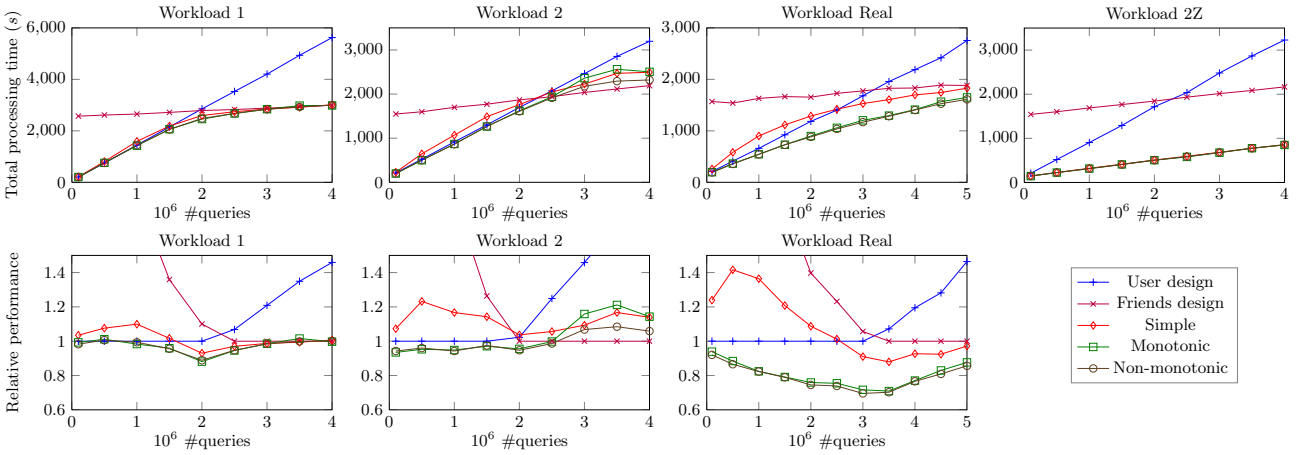



**Figure 7: Performance for workloads with different fractions of documents vs. queries**

works [29, 2, 12, 22, 5, 1]. Some of these papers address the design and evaluation of new ranking functions that incorporate different network-centric properties of social networks (see for example [22, 5]). Others are concerned with efficient processing techniques for top-$k$ queries with a focus on social tagging sites [2, 29]. However, none of these methods enforce access control because they may return any document tagged by users connected to the user who submitted the query, and this may include inaccessible documents.

Another related problem studied by Silberstein et al. [31] is that of computing event feeds containing the most recent posts of friends in a social network. Our work and the work of [31, 2] took a workload-aware approach to find a design in a large design space. Despite the fact that each work seeks to compute a different function (event feeds, vs. query results based on tags by friends vs. query results based on post content and access rights) there is a nice parallel between our design space and that of Silberstein et al. However, as we saw in the experimental section, adapting the cost model of Silberstein et al. produces poor results in our setting and we therefore developed our own cost models that perform much better.

The problem we address is related to access control in both information retrieval [13, 32] and for structured data [10].

Cost models have been used to estimate the efficiency of processing strategies in both information retrieval and databases [35, 14, 16, 25]. There exist advanced cost models to evaluate different index construction and update strategies [14, 35]. For search queries, however, simple models are most commonly used, sometimes without verification on an actual system [16]. Unlike previous models that estimate an abstract notion of cost (such as memory accesses [25]), we developed models to estimate the exact running time of a workload. This is more challenging and advanced models are required to predict query processing costs accurately as shown in our experimental evaluation.

The problems of calculating unions and particularly intersections of lists have attracted a lot of attention, both through the introduction of new algorithms with theoretical analyses [23, 20, 3, 7], and experimental investigations [8, 4]. The algorithms for single operators are also combined into full query plans [28, 17]. Our implementation computes intersections and unions over compressed lists with synchronization points for skipping following [19, 27]. Our intersection operator is based on the ideas from Demaine et al. [20]. For the union operators we also could have employed the practical implementations of Raman et al. [28], but we instead developed HeapUnion and demonstrated its superiority experimentally. Other implementations seem less suitable because [20] does not support skipping and would thus be inefficient in our setting, and [17] has a theoretical focus. HeapUnion saves work in the heap maintenance compared to a multi-way merge by carefully analyzing what parts of the heap need to be examined and rebuilt. The idea of optimized heap maintenance has been used in the context of tf-idf-based ranking [15]. Our techniques are different and can be used to improve previous work in case many documents contain only a subset of the queried terms.

## 7. CONCLUSIONS

In this paper we have presented an efficient system for keyword search in social networks with access control. Through

the introduction of accurate cost models and associated workload-aware optimization algorithms, we are able to find designs of access control meta-data that speed up performance by a factor of up to 3.4 over previous work. We also introduced HeapUnion, a novel query processing operator that efficiently supports skipping over unions of sorted inputs. HeapUnion improves query processing efficiency with a factor between 1.12 and 2.36 in our system.

With this foundation in place, we have the basis for extensions to more advanced ranking functions such as ranking based on network-centric properties in social networks [29, 2] while enforcing access control. We may also be able to apply the techniques in this paper in other areas, such as in star joins in data warehouses [28], where HeapUnion might be an interesting query processing operator.

# 8. REFERENCES


[1] S. Amer-Yahia, M. Benedikt, and P. Bohannon. Challenges in searching online communities. *IEEE Data Eng. Bull.*, 30(2):23–31, 2007.

[2] S. Amer-Yahia, M. Benedikt, L. V. S. Lakshmanan, and J. Stoyanovich. Efficient network aware search in collaborative tagging sites. *Proc. VLDB Endow.*, 1(1):710–721, 2008.

[3] R. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In *Proc. CPM*, 2004.

[4] R. Baeza-Yates and A. Salinger. Fast intersection algorithms for sorted sequences. *Algorithms and Applications*, 2010.

[5] S. Bao, G. Xue, X. Wu, Y. Yu, B. Fei, and Z. Su. Optimizing web search using social annotations. In *Proc. WWW*, 2007.

[6] A. L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[7] J. Barbay and C. Kenyon. Alternation and redundancy analysis of the intersection problem. *ACM Trans. Algorithms*, 4(1), 2008.

[8] J. Barbay, A. López-Ortiz, T. Lu, and A. Salinger. An experimental investigation of set intersection algorithms for text searching. *J. Exp. Algorithmics*, 14, 2009.

[9] J. L. Bentley and A. C.-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3), 1976.

[10] E. Bertino, S. Jajodia, and P. Samarati. Database security: research and practice. *Inf. Syst.*, 20(7), 1995.

[11] T. A. Bjørklund. *Column Stores versus Search Engines and Applications to Search in Social Networks*. PhD thesis, NTNU, 2011.

[12] T. A. Bjørklund, M. Götz, and J. Gehrke. Search in social networks with access control. In *Proc. KEYS*, 2010.

[13] S. Büttcher and C. L. A. Clarke. A security model for full-text file system search in multi-user environments. In *FAST*, 2005.

[14] S. Büttcher, C. L. A. Clarke, and B. Lushman. Hybrid index maintenance for growing text collections. In *Proc. SIGIR*, 2006.

[15] B. B. Cambazoglu and C. Aykanat. Performance of query processing implementations in ranking-based text retrieval systems using inverted indices. *Inf. Process. Manage.*, 42, 2006.

[16] B. B. Cambazoglu, V. Plachouras, and R. Baeza-Yates. Quantifying performance and quality gains in distributed web search engines. In *Proc. SIGIR*, 2009.

[17] E. Chiniforooshan, A. Farzan, and M. Mirzazadeh. Worst case optimal union-intersection expression evaluation. In *ICALP*, 2005.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[19] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. SPIRE*, 2007.

[20] E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proc. SODA*, 2000.

[21] S. Gurajada and S. K. P. On-line index maintenance using horizontal partitioning. In *Proc. CIKM*, 2009.

[22] A. Hotho, R. Jäschke, C. Schmitz, and G. Stumme. Information retrieval in folksonomies: Search and ranking. In *The Semantic Web: Research and Applications*, 2006.

[23] F. K. Hwang and S. Lin. Optimal merging of 2 elements with n elements. *Acta Informatica*, 1(2), 1971.

[24] N. Lester, A. Moffat, and J. Zobel. Efficient online index construction for text databases. *ACM Trans. Database Syst.*, 33(3), 2008.

[25] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *Proc. VLDB*, 2002.

[26] G. Margaritis and S. V. Anastasiadis. Low-cost management of inverted files for online full-text search. In *Proc. CIKM*, 2009.

[27] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Trans. Inf. Syst.*, 14(4), 1996.

[28] V. Raman, L. Qiao, W. Han, I. Narang, Y.-L. Chen, K.-H. Yang, and F.-L. Ling. Lazy, adaptive rid-list intersection, and its application to index anding. In *Proc. SIGMOD*, 2007.

[29] R. Schenkel, T. Crecelius, M. Kacimi, S. Michel, T. Neumann, J. X. Parreira, and G. Weikum. Efficient top-k querying over social-tagging networks. In *Proc. SIGIR*, 2008.

[30] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. SIGIR*, 2002.

[31] A. Silberstein, J. Terrace, B. F. Cooper, and R. Ramakrishnan. Feeding frenzy: Selectively materializing users. event feeds. In *Proc. SIGMOD*, 2010.

[32] A. Singh, M. Srivatsa, and L. Liu. Efficient and secure search of enterprise file systems. In *Proc. ICWS*, 2007.

[33] R. Sprugnoli. Recurrence relations on heaps. *Algorithmica*, 15(5), 1996.

[34] H. Turtle and J. Flood. Query evaluation: strategies and optimizations. *Inf. Process. Manage.*, 31(6), 1995.

[35] I. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes*. Academic Press, 1999.

[36] J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proc. WWW*, 2008.

[37] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 2006.

[38] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Proc. ICDE*, 2006.