SNFS: The design and implementation of a Social Network File System

Charalabos Kaidos
University of Patras
kaidos@ceid.upatras.gr

Andreas Pasiopoulos
University of Patras
pasiopou@ceid.upatras.gr

Nikos Ntarmos
University of Ioannina
ntarmos@cs.uoi.gr

Peter Triantafillou
University of Patras
peter@ceid.upatras.gr

Abstract

Social network systems and services have become amazingly popular in recent years. This has resulted in huge amounts of data being published by users. At the same time, a great number of relationships between users, user groups, and (collections of) data items are constantly being established based on highly dynamic tagging activities by users.

With this work we present the design and implementation of a special-purpose user-level file system, coined SNFS, designed to manage social-network entities (data items, users and their profiles, and tags) and their relationships. At the core of our approach lie tagging, indexing, and ranked retrieval (top-k) algorithms, allowing the key functionality to be provided in a timely manner. We discuss the core design and implementation features of SNFS and present a performance evaluation, exposing the key performance costs, and present alternative designs and implementations to overcome them. Finally, we provide a brief comparison with a well-known desktop search application, Beagle, and show, using real datasets, that for our envisaged queries SNFS provides significant performance gains.

Categories and Subject Descriptors H [*3*]

General Terms Design, Performance

Keywords Social networks, file systems, indexing, top-k queries

SNS '11 April 10–13, 2011, Salzburg, Austria.

1. Introduction

Social network sites are defined as "*web-based services that allow individuals to (1) construct a public or semi-public profile within a bounded system, (2) articulate a list of other users with whom they share a connection, and (3) view and traverse their list of connections and those made by others within the system*"[Boyd 2007]. Part of these services is to allow users to communicate with others through publishing, viewing, or interacting with *objects* that express their *interests*. These objects may be text, photos, videos, audio, user profiles or other digital media. The amount of data published by users is so large that even traversing some general categories to find interesting objects is not reasonable or practical. Thus, social service providers need mechanisms to present to their users with objects that match their explicitly or implicitly published interests and also allow them to search for such entities.

To allow such a mechanism, each object is associated with a set of terms and keywords, or *tags* as they are known in on-line social networks terminology. Essentially tags are basic information morsels, characterizing other more complex information entities. This makes tags appropriate for use in search engines and matching systems. Tags may be provided by the publisher of the object, the users of the service, or even an authority or metadata extraction algorithm, though the first two are the most common cases in contemporary social network services. YouTube, Flickr annotations, and Facebook tags are examples of the first case, while Google Book keywords represent the latter.

In this paper we present the thesis that, the relation between entities in a social network system – be it pictures or videos, users, user groups, annotations/tags on objects, and so on – along with the searching and matching mechanisms, can be presented through the well-known abstraction of a file system, yielding a Social Network File System (or SNFS).

SNFS, a file system specifically designed for social networks, pays particular attention to accommodating content items and user profiles, as well as their tags. The current design of SNFS is geared towards allowing users to identify relevant content and/or other users using a set of keywords/tags; through this discovery step, users can then befriend other users, create expert groups, participate in social discussions, etc. At the core of our query processing engine lie implementations of Fagin's Threshold Algorithms, facilitating top-K query execution. In this way, we achieve high performance as we avoid the need to scan complete (typically huge) index lists.

We contribute a detailed performance evaluation of SNFS. We identify the major sources of performance costs – such as the number of disk accesses, the number of system calls, and the impact of different versions of the TA algorithms – and we perform experiments which measure their impact on the total cost. In the same vein, we offer alternative designs/implementations that can avoid/reduce each major cost contributor, such as RAM-disk SNFS and FUSE-less SNFS.

Moreover, in an effort to compare SNFS against other keyword-based search engines, we pitched SNFS's performance against a well-known and acknowledged desktop search application: Beagle. Although Beagle was designed with slightly different purposes in mind, we performed head-to-head comparisons for queries from the same datasets. SNFS's response time is shown to be a fraction of that of Beagle.

Finally, we discuss how to extend SNFS to operate in a distributed environment (be it for cloud infrastructures or for other overlay-based distributed networked infrastructures) and how our basic design accommodates this.

2. Related work

The idea of using a file system mechanism to provide associative attribute-based access to the content of an information storage system was first presented in semantic file systems[Gifford 1991] as a more effective storage abstraction than traditional tree structured file systems. The querying system is represented by virtual directories; when the user asks for the contents of a directory, a virtual directory is created and populated with files whose attributes match the requested path.

The years following the semantic file system, other designs and implementations to extend this idea were presented. With such file systems as the Be File System (or BeFS) [Giampaolo 1999] and its descendants, the idea of attaching keywords to files and indexing them to allow for content searching were brought to the wide public. Unfortunately, practical shortcomings (the Be Operating Systems, featuring BeFS, never really took off) and performance issues of such file systems with regard to the actual content indexing, did not let them make a big splash.

On the other hand, the functionality of desktop search was made commercially available by Microsoft in 1998, bundled in the "Windows NT 4.0 Option Pack" product. It provided filters to extract information from plain text files, Microsoft Office files, HTML, and MIME data. Although quite efficient, the early-stage, low penetration of the Internet, and the relatively small amount of data stored in end-user computers, were manageable by the user through traditional means and the need of a desktop search engine was not enough to warrant a paradigm shift.

The idea was later popularized with the release of Windows Desktop Search (in 2004), Mac OS X Spotlight, and Google Desktop (2005). These applications monitor the file systems in use and support extraction of metadata, indexing and retrieval of many common files, emails, applications and visited web pages. These application were quite successful as the timing was right; multimedia was prevalent, large storage devices were cheaply available, and the internet was already gigantic and wide-spread. These products are still in continuous development, supporting more platforms (like Linux and iOS), while many other implementations have spurred (e.g., Linux's Beagle and Strigi) with varying levels of popular acceptance.

Albeit useful and popular, these programs are subtly flawed. Their primary shortcoming is that they separate the data the user has stored in her terminal from the on-line available data in other devices and/or on the Internet. Thus, the results that these applications produce are only part of the information available to the user.

Also these programs are more useful to advanced users that know what they are looking for and how to use these tools to find it by making specialized queries with many search terms and even boolean or SQL-like operators. However, for the average user who not only does not know how to perform these queries, but also in many cases does not know beforehand what exactly she is looking for, the usefulness of these desktop search applications is quite limited.

The approach of ranked retrieval in search results in contrast to the "belonging to a result set" retrieval of the boolean model was discussed, implemented, and evaluated with the SMART system[Shalton 1968]. This system ranks each document by its relevance to a free text query using a weighting scheme to assign a value to each document-term pair and to each term of the query. It creates a vector[Shalton 1975] for each document and for the query, then uses the cosine or overlap correlations to determine the similarity between each document and the query. Other than these correlations, Shalton and Buckley also presented and tested several other functions to compute scores of documents and queries[Shalton 1988].

Top-k query processing stands as an important building block for ranked retrieval systems. Usually these are implemented using variants of the seminal Threshold Algorithm[Fagin 2003]. These algorithms return the top k doc-

uments with the highest similarity to the query, where the query is a set of attributes. The algorithms expect lists of documents sorted by descending relevance to each attribute. They traverse the lists in parallel while computing the similarity of documents they come by using a scoring system. When no further improvement on the top-k results can be made, the algorithms stop without traversing the rest of the lists.

3. The Design of SNFS

The first design decision was to choose a scoring system for out algorithms. We opted for the classic approach of a tf-idf scheme, but our system can accommodate other scoring schemes as well. Moreover, the algorithms we describe in section 3.2 need the scoring function to be monotonic, but impose no other limitation on them. In brief, the architecture of the system, consists of two processes, the SNFS process and the searching application.

3.1 SNFS Process

The Social Network File System (SNFS) process is responsible for the manipulation and indexing of objects and their corresponding tags. Each object is associated with a unique id and the object path is stored in a Red-Black Tree. This tree resides in main memory for quick access. The tags, or terms, are extracted and stemmed with the Porter2 algorithm [Porter 1981]. Then, weights for each term in each object are calculated, and the scoring is done using a tf-idf approach. When computing term weights, we are using a term frequency variation; specifically, the term frequency normalization function[Shalton 1988]:

$$ntf_{t,d} = a + (1-a)\frac{tf_{t,d}}{tf_{max}(d)}$$

where $tf_{max}(d)$ is the maximum term frequency of all terms in object d, and a is a smoothing factor to avoid large swings of $ntf_{t,d}$ caused by possibly modest changes in $tf_{t,d}$. As proposed by [Manning 2008] we chose $a = 0.4$. The term frequency of a tag is defined as the number of users that have assigned this tag to an object. That also allows us to minimize the importance of tag spamming. Thus, the normalized frequency does not favor popular objects (equally to a classic retrieval system not favoring longer documents). On the other hand, as the objects are dynamically tagged by users, a change in $tf_{max}(d)$ requires that all term weights of object d are recalculated. This cost may become significant in very volatile systems, and delayed/lazy update techniques or other relevant solutions may be considered.

This information is stored in an inverted index. Each term is associated with a posting list, containing its document frequency and pairs of document ids and term weights. To allow efficient insertions into posting lists while maintaining them in descending order of weights, we have chosen to use *B+ Trees* due to their efficiency in storing information in secondary memory while allowing for minimum disk accesses to retrieve any node. The B+ Tree is a variation of the B-Tree in which all records are stored in the leaves and all leaves are linked sequentially. So each term is associated with the root node of the B+ Tree which stores its posting list. The key used in B+ Trees is the weight, thus keeping the documents in the posting lists stored in decreasing order. These indices can then be transparently distributed across several network nodes (e.g., in a cloud-based scenario) by using distributed B+Trees[Wu 2010] or other similar constructs.

The terms are associated with their posting list using a second in-memory Red-Black Tree. This tree stores the position of the root node in the inverted index file with the key being the term. We use this tree to efficiently check for the existence of a term in our collection and to retrieve the root node when searching the collection.

3.2 Searching Process

The searching process is responsible for receiving queries, searching the inverted index, computing the scores of the objects for the given query, and returning the top-k results. The queries supported consist of identifying content items and/or user profiles based on the aforementioned tags. So given a set of tags the search client can find relevant objects and given an object it should extract tag information and return other objects with the same characteristics. Thus the queries SNFS may answer are useful to social network systems ranging from "objects with certain tags" to "profiles of users that may be interested in certain objects" or "profiles related to other profiles with a defined form of relationship (friendship, common interests, etc.)"

To this extent, the terms given by the user or extracted by an object are stemmed with the same algorithm as the tags during the extraction phase. For all terms t of a query, the corresponding document frequencies stored in the posting list of each term are retrieved, and the total inverse document frequency is computed using the formula:

$$idf_t = \log \frac{N}{df_t}$$

where N is the number of documents (objects) in the collection. To compute the score of each document d for the given query q we use the common tf-idf multiplication function[Shalton 1988]:

$$Score(q,d) = \sum_{t \in q} ntf_{t,d} \times idf_t$$

In order to find the top-k documents, a threshold algorithm is used. We inspected two of the variants proposed by [Fagin 2003], specifically the Threshold Algorithm with Random Access (TA) and the No Random Access (NRA) algorithm, briefly outlined below.

3.2.1 The Random Access Threshold Algorithm

Let t be the terms in the query, and L_t be the corresponding posting lists (ordered in descending order of weight). TA

does sorted access in parallel to each list. For each object d that is seen in some list, it fetches its weights x_t in each of the other lists by performing random accesses. It then computes the score of the object d using the scoring function $f(x_1, x_2, \ldots, x_t)$. If this score is one of the k highest seen thus far, the object is added to the top-k set, else it is discarded, so that only k objects need to be maintained at any time. For each L_t, let $\underline{x}_t$ be the weight of the last object seen under sorted access, and let τ (the *threshold value*) be equal to $f(\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_t)$. As soon as k objects have been seen whose score is at equal to or larger than τ, the algorithm halts and returns the top-k result set.

3.2.2 The No Random Access Algorithm

Let t be the terms in the query, and L_t be the corresponding posting lists ordered in descending order of weight. NRA accesses these lists in parallel, scanning through each list in a sequential manner in descending weight order. At each depth m (that is, after having scanned m entries in each list), it maintains a list of the last (and thus least) weight $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_t$ encountered in each list. For each object d encountered in some list for which not all t weights have been seen yet, it computes a lower $W^m(d)$ and upper $B^m(d)$ limit for its score, substituting either 0 or the bottom values $\underline{x}_1, \underline{x}_2, \ldots, \underline{x}_t$ respectively for the yet unseen weights. Let T_k^m be the current top-k list containing the k objects with the largest W^m values seen so far and their scores, and M_k be the k^{th} largest W^m value in T_k^m. An object D is called *viable* if $B^m(D) > M_k$. The algorithm halts when at least k distinct objects have been found and there are no more viable objects – that is, $B^m(D) \leq M_k$ – at which point the objects in T_k are returned.

3.2.3 Qualitative Performance Comparison

We will now attempt to qualitatively compare the expected performance of these algorithms. The time cost of each algorithm is dominated by three main factors.

First, the number of disc accesses. This are dependent on the number of objects needed to be seen before the algorithm halts. The NRA algorithm will typically go much deeper in the posting lists than TA, but as we are using a B+ tree we are reading a node at a time. If wasting space on rare terms is not a concern, we may increase the node size up to the optimal (i.e., the block size of our device) to minimize disc accesses. That said, we expect TA to demand much more disk accesses, as it performs random accesses on the index. The estimated number of disk accesses required by NRA are $DA_{NRA} = (h + \frac{d}{M}) \times t$, whereas for TA they are $DA_{TA} = (h + \frac{d}{M}) \times t + (t \times (t-1) \times h \times d)$, where h is the height of the B+Tree, d is the depth in which each algorithm halts, M is the lower bound of the number of keys in the B+Tree, and t is the number of terms in the query.

Second, the number of system calls. Each system call is expected to take time as it halts the execution of the program and makes a request to the operating system kernel. The system calls most frequently used by these algorithms are *seek* and *read*. For each disc access, both system calls are expected to be used at least once. So we expect TA to make more system calls than NRA.

Finally, the amount state and book keeping chores. As we described earlier, TA requires minimal bookkeeping. On the other hand, NRA maintains a much larger state and performs more computations in each iteration, thus being more expensive in this field.

4. Implementation

We have built a real-world implementation of our Social Network File System so that we can support our thesis of its use in retrieving top-k objects from social network services. The system was written in the C programming language, intended to be deployed in a Linux/Unix environment. To implement the file system as a user space application we used the FUSE infrastructure, available in all Linux distributions and BSD-derived operating systems (i.e., FreeBSD, NetBSD[1], OpenBSD, DragonflyBSD, etc.). FUSE allows us to implement our own file system functions (like *write* or *read*) while also giving us access to underlying reliable storage file systems, like ext4, ReiserFS, FFS, and UFS. The main shortcoming of FUSE is its overhead with regard to the number of system calls required to perform its operations. As shown in Figure 1, a system call from a client (like our searching application) will be routed through VFS to the FUSE kernel module and back to user level and SNFS. From there the SNFS will decide whether to issue another system call to access a storage file system (e.g., ext), or if it will retrieve the necessary data from a distributed system (like HDFS) or data structure. Our small scale testing implementation uses the former, being layered on top of an ext4 file system through VFS.

Since our objects in this case are real files containing data and not a network or other abstraction, we need physical disc space to store them. So the SNFS in this case acts as an overlay to an existing file system (i.e., *ext4*). Note that in general the objects are meant to be arbitrary resources, stored either locally or on some remote server, or available through some on-line social networking service. In our implementation the extraction function reads N tags from the file with name *path*. Then the tags are stemmed using the Porter2 algorithm. This results in K ($K \leq N$) terms each with tf_K frequency. For the Porter2 stemming algorithm we used the C API provided by Dr. Martin Porter[2]. Pairs of *document-ids* and *file paths* are stored in-memory using a red-black tree. The weight of each term is then computed as described in the design section. Each object is also assigned a document id (i.e., a unique large integer). The list of terms and weights along with the document id is then stored in an inverted index.

[1] A native PUFFS port is also under consideration.

[2] `http://snowball.tartarus.org/index.php`

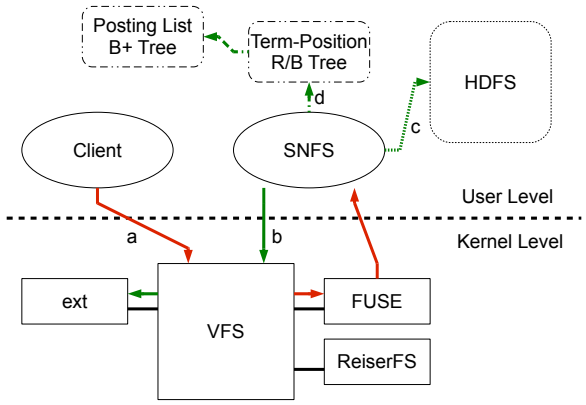

Figure 1: System calls through SNFS. a) Clients issue a system call to VFS. The call is routed through FUSE to SNFS. b) SNFS issues a system call to VFS to access the underlying file system (ext4 in our implementation). c) It is possible that SNFS is an HDFS (or other distributed FS) client, retrieving data from there, or d) SNFS may be distributed itself by using a distributed index and/or distributed B+Trees

The latter consists of a pair of B+ Tree structures for each posting list of the collection: one indexed on the object weights to allow for sequential access in decreasing weight order, and one indexed on the document IDs to allow for random id accesses. These trees are stored in two files in the root directory of our file system. The size of the nodes can be adjusted to match the needs of various specific systems. Larger nodes reduce the height of the tree resulting in a reduction of the number of disk accesses, while smaller nodes save up disc space as tags usually have very low document frequencies. The document frequency of each term is stored in the root node. Note that the index does not contain the actual terms. To this extent, we use a second in-memory red-black tree, mapping each term to the position of the root nodes of its posting lists in the inverted index file. Both red-black trees are also stored as files in the root of the file system, and updated during the unmounting phase of the file system. When SNFS is mounted again, the files are opened and the data are loaded back into main memory.

As far as the searching process is concerned, we built a simple application allowing the user to enter free-text queries and ask for top-k results. K results are returned along with the time needed to perform the search.

Currently we have not created any interface for SNFS; instead, we opted for the default (hierarchical) presentation of the underlying file system. An abstraction akin to the virtual directories of the semantic file system is left for future work.

5. Performance Results

As outlined in the qualitative performance comparison section, we expect the number of disk accesses to represent a large percentage of the overall time cost, especially for TA. To test this assumption we provide two different implementations of SNFS: (i) one using a standard mechanical hard disk to store the index, and (ii) an implementation using RAM-disks, minimizing the importance of disk accesses and concentrating on the other factors that contribute to the time overhead of our algorithms.

The datasets we used are from real on-line social networks and other content providers which use a tagging system, in the form described earlier. More specifically, we used the following four datasets: YouTube (8208 obj.), Flickr (18246 obj.), Google (4338 obj.) and IMDB (8099 obj.).

Time consumption during the pre-processing phase increased linearly to the number of objects being indexed, reaching 19.000 insertions in the B+Tree per second. We then created queries that consist of tags with high document frequencies. These queries engage terms that have long posting lists and thus represent a worst-case scenario for any ranked retrieval system. We tested those queries on the above two implementations. In all cases before executing each query, the VFS buffer cache memory was cleared to allow for more objective results, not affected by previous executions.

5.1 Disk-based SNFS

Our tests on disk based indices confirmed our hypothesis that the NRA algorithm is faster for this application. NRA goes much deeper into the sorted lists than TA, often reaching the end of the lists before halting, resulting in the paradox of what seems like a constant time overhead for larger values of k. This was not the case with TA, whose time overhead increased with k. As expected time cost rises for more terms in the query due to the increase in relevant objects. As is shown in the first column of Figure 2, NRA manages to return results quite faster than TA in all cases.

5.2 RAM-based SNFS

The RAM-disk implementations provides much lower access times and a very large sustained bandwidth compared to the HDD-based variant. As shown in the second column of Figure 2, the time needed to process the queries is reduced considerably for both algorithms: TA needs less than $\frac{1}{2}$ of each time overhead with the HDD-based implementation, while almost NRA drops to $\frac{1}{5}$ of each previous time, achieving sub-second performance in all cases. Moreover, NRA is still faster than TA as the access time may have been essentially nullified but the time needed to make each system call was not, and TA still uses many more system calls than NRA.

5.3 Avoiding FUSE

Using FUSE hinders our program with system calls overhead. As of this, we decided to take a detour and run our queries directly on the indices stored on ext4. The results showed that the time overhead imposed by FUSE is about

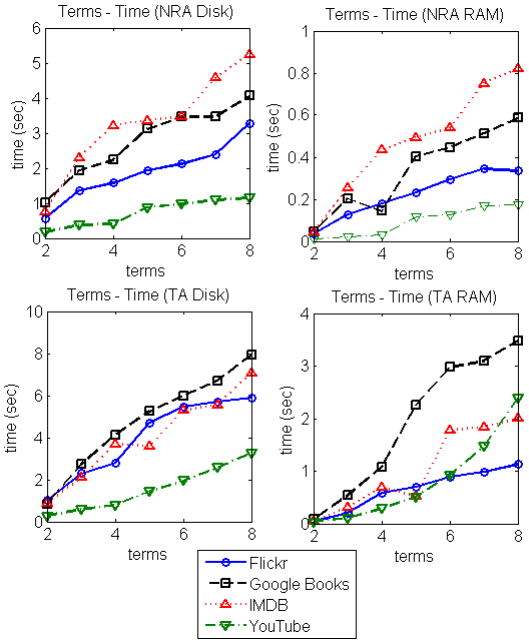

Figure 2: Time needed to compute the top-10 result set for varying length queries

40% of the total processing time, being even more evident with TA which makes the most system calls.

5.4 Comparing SNFS (NRA) with Beagle

We needed to see how our indexing system performed against competition. We opted to test it against a well known desktop search application; Beagle. We would like to note that Beagle's Lucene engine is not optimized for a single application but to index several different file types and answer more complex queries consisting of boolean operators. Nonetheless, it is interesting that SNFS managed to perform quite well. We tested Beagle on the populous Flickr data set. The default filtering operator of Beagle is AND, so we used the OR operator between terms to achieve the partial matching we do with free text queries on NRA. We varied both k (demanded number of objects) and the number of terms as shown on Table 1. Much like NRA, Beagle appears to be less and less affected by further increasing k.

Table 1: Comparison of Beagle and NRA algorithm

Number of terms	2	3	4	5	6	7	8
NRA (Disk based) (sec)	0.570	1.365	1.578	1.940	2.120	2.389	3.268
Beagle (sec)	4.917	5.163	5.158	5.286	5.112	5.345	5.834

k (4 terms query)	1	2	5	10	50	100
NRA (Disk based) (sec)	1.010	1.451	1.572	1.578	1.564	1.562
Beagle (sec)	5.017	5.052	5.126	5.565	5.915	5.988

6. Conclusions and future work

With this work we present the Social Network File System; a novel means of indexing and accessing social network data, through the familiar abstraction of files in a virtual file system. Our purpose was twofold: (i) to provide social network sites with a new way of managing user data and activity, and (ii) to offer end-users a way to tag, index, and search for objects, lying both in their computers and devices as well as over the internet. We have discussed the design details of SNFS and its basic building blocks. We have also presented an experimental evaluation of its performance for various on-line datasets. Our results showcase the viability of our approach. We also proposed two ways of expansion towards distributed and cloud systems: (a) first, to use an underlying distributed storage file system, such us HDFS, which will take care of scalability and reliability chores; (b) second, to distribute SNFS's data structures across multi-node systems by utilizing distributed indexes and/or distributed B+Trees, layered over commodity storage file systems in each node.

References

[Boyd 2007] Danah M. Boyd and Nicole B. Ellison. Social network sites: Definition, history, and scholarship. *Journal of Computer-Mediated Communication*, 13(1):210–230, 2007.

[Fagin 2003] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4), 2003.

[Giampaolo 1999] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann, 1999. ISBN 1-55860-497-9.

[Gifford 1991] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole Jr. Semantic file systems. In *SOSP '91 Proceedings of the thirteenth ACM symposium on Operating systems principles*, New York, NY, USA, 1991. ACM.

[Manning 2008] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008. ISBN 0521865719.

[Porter 1981] M. Porter. Snowball: A language for stemming algorithms, 1981. `http://snowball.tartarus.org/texts/introduction.html`.

[Shalton 1988] Gerard M. Shalton and Christopher Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal*, 24(5), 1988.

[Shalton 1968] Gerard M. Shalton and Michael E. Lesk. Computer evaluation of indexong and text processing. *Journal of the ACM*, 15(1), 1968.

[Shalton 1975] Gerard M. Shalton, Andrew K C Wong, and C S Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11), 1975.

[Wu 2010] Sai Wu, Dawei Jiang, Beng Chin Ooi, and Kun-Lung Wu. Efficient b-tree based indexing for cloud data processing. *Proceedings of the VLDB Endowment*, 15(1), 2010.