ClusteringWiki: A Framework for Personalized Clustering of Search Results

Abstract

How to organize and present search results plays a critical role in the utility of search engines. Due to the unprecedented scale of the Web and diversity of search results, the common strategy of ranked lists has become increasingly inadequate, and clustering has been considered as a promising alternative. Clustering divides a long list of disparate search results into a few topic-coherent clusters, allowing the user to quickly locate relevant results by topic navigation. While many clustering algorithms have been proposed that innovate on the automatic clustering procedure, we introduce `ClusteringWiki`, the first prototype and framework for personalized clustering that allows direct user editing of the clustering results. Through a Wiki interface, the user can edit and annotate the membership, structure and labels of clusters for a personalized presentation. In addition, the edits and annotations can be shared among users as a mass-collaborative way of improving search result organization and search engine utility.

I. Introduction

The way search results are organized and presented has a direct and significant impact on the utility of search engines. The common strategy has been using a flat ranked list, which works fine for homogeneous search results.

However, queries are inherently ambiguous and search results are often diverse with multiple senses. With a list presentation, the results on different sub-topics of a query will be mixed together. The user has to sift through many irrelevant results to locate those relevant ones.

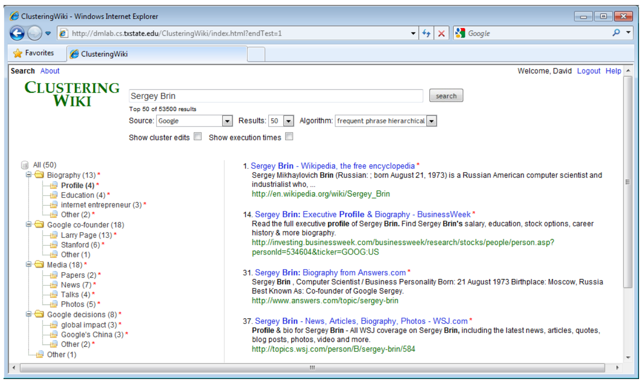
With the rapid growth in the scale of the Web, queries have become more ambiguous than ever. For example, there are more than 20 entries in Wikipedia for different renown individuals under the name of Jim Gray and 74 entries for Michael Smith as of today. Consequently, the diversity of search results has increased to the point that we must consider alternative presentations, providing additional structure to flat lists so as to effectively minimize browsing effort and alleviate information overload [11], [24], [33], [4]. Over the years clustering has been accepted as the most promising alternative.

Clustering is the process of organizing objects into groups or clusters that exhibit internal cohesion and external isolation. Based on the common observation that it is much easier to scan a few topic-coherent groups than many individual documents, clustering can be used to categorize a long list of disparate search results into a few clusters such that each cluster represents a homogeneous sub-topic of the query. Meaningfully labeled, these clusters form a topic-wise non-predefined, faceted search interface, allowing the user to quickly locate relevant and interesting results. There is good evidence that clustering improves user experience and search result quality [22].

Given the significant potential benefits, search result clustering has received increasing attention in recent years from the communities of information retrieval, Web search and data mining. Many clustering algorithms have been proposed [11], [24], [33], [34], [35], [19], [31], [20]. In the industry, well-known cluster-based commercial search engines include Clusty (www.clusty.com), iBoogie (www.iboogie.com) and CarrotSearch (carrotsearch.com).

Despite the high promise of the approach and a decade of endeavor, cluster-based search engines have not gained prominent popularity, evident by Clusty's Alexa rank [12]. This is because clustering is known to be a hard problem, and search result clustering is particularly hard due to its high dimensionality, complex semantics and unique additional requirements beyond traditional clustering.

As emphasized in [31] and [4], the primary focus of search result clustering is NOT to produce optimal clusters, an objective that has been pursued for decades for traditional clustering with many successful automatic algorithms. Search result clustering is a highly user-centric task with *two unique additional*


ClusteringWiki - Windows Internet Explorer
http://dmlab.cs.txstate.edu/ClusteringWiki/index.html?endTest=1
Google
Favorites
ClusteringWiki
Search
About
Welcome, David
Logout
Help
CLUSTERING WIKI
search

Sergey Brin

Top 50 of 53500 results

Source: Google   Results: 50   Algorithm: frequent phrase hierarchical

Show cluster edits ☐   Show execution times ☐

- All (50)
  - Biography (13) *
    - **Profile (4)** *
    - Education (4) *
    - internet entrepreneur (3) *
    - Other (2) *
  - Google co-founder (18)
    - Larry Page (13) *
    - Stanford (6) *
    - Other (1)
  - Media (18) *
    - Papers (2) *
    - News (7) *
    - Talks (4) *
    - Photos (5) *
  - Google decisions (8) *
    - global impact (3) *
    - Google's China (3) *
    - Other (2) *
  - Other (1)

1. Sergey **Brin** - Wikipedia, the free encyclopedia *
   Sergey Mikhaylovich **Brin** (Russian: ; born August 21, 1973) is a Russian American computer scientist and industrialist who, ...
   http://en.wikipedia.org/wiki/Sergey_Brin

14. Sergey **Brin:** Executive **Profile** & Biography - BusinessWeek *
   Read the full executive **profile** of Sergey **Brin.** Find Sergey **Brin's** salary, education, stock options, career history & more biography.
   http://investing.businessweek.com/businessweek/research/stocks/people/person.asp?personId=534604&ticker=GOOG:US

31. Sergey **Brin:** Biography from Answers.com *
   Sergey **Brin** , Computer Scientist / Business Personality Born: 21 August 1973 Birthplace: Moscow, Russia Best Known As: Co-founder of Google Sergey.
   http://www.answers.com/topic/sergey-brin

37. Sergey **Brin** - News, Articles, Biography, Photos - WSJ.com *
   Profile & bio for Sergey **Brin** - All WSJ coverage on Sergey **Brin,** including the latest news, articles, quotes, blog posts, photos, video and more.
   http://topics.wsj.com/person/B/sergey-brin/584

Fig. 1. Snapshot of `ClusteringWiki`.

requirements. First, clusters must form interesting sub-topics or facets from the user's perspective. Second, clusters must be assigned informative, expressive, meaningful and concise labels. Automatic algorithms often fail to fulfill the human factors in the objectives of search result clustering, generating meaningless, awkward or nonsense cluster labels [4].

In this paper, we explore a completely different direction in tackling the problem of clustering search results, utilizing the power of direct user intervention and mass-collaboration. We introduce `ClusteringWiki`, the first prototype and framework for personalized clustering that allows direct *user* editing of the *clustering results*. This is in sharp contrast with existing approaches that innovate on the *automatic* algorithmic *clustering procedure*.

In `ClusteringWiki`, the user can edit and annotate the membership, structure and labels of clusters through a Wiki interface to personalize her search result presentation. Edits and annotations can be implicitly shared among users as a mass-collaborative way of improving search result organization and search engine utility. This approach is in the same spirit of the current trends in the Web, like Web 2.0, semantic web, personalization, social tagging and mass collaboration.

Clustering algorithms fall into two categories: partitioning and hierarchical. Regarding clustering results, however, a hierarchical presentation generalizes a flat partition. Based on this observation, `ClusteringWiki` handles both clustering methods smoothly by providing editing facilities for cluster hierarchies and treating partitions as a special case. In practice, hierarchical methods are advantageous in clustering search results because they construct a topic hierarchy that allows the user to easily navigate search results at different levels of granularity.

Figure 1 shows a snapshot of `ClusteringWiki`[1]. The left-hand *label panel* presents a hierarchy of cluster labels. The right-hand *result panel* presents search results for a chosen cluster label. A logged-in user can edit the current clusters by creating, deleting, modifying, moving or copying nodes in the cluster tree. Each edit will be validated against a set of predefined consistency constraints before being stored.

Designing and implementing `ClusteringWiki` pose non-trivial technical challenges. User edits represent user preferences or constraints that should be respected and enforced next time the same query is issued. Query processing is time-critical, thus efficiency must be given high priority in maintaining and enforcing user preferences. Moreover, complications also come from the dynamic nature of search results that constantly change over time.

Cluster editing takes user effort. It is essential that such user effort can be properly reused. `ClusteringWiki` considers two kinds of reuse scenarios, *preference transfer* and *preference sharing*. The former transfers user preferences from one query to similar ones, e.g., from "David J. Dewitt" to "David Dewitt". The latter aggregates and shares clustering preferences among users. Proper aggregation allows users to collaborate at a mass scale and "vote" for the best search result clustering presentation.

[1]clusteringwiki.pilotpro.org.

In social tagging, or collaborative tagging, users annotate Web objects, and such personal annotations can be used to collectively classify and find information. `ClusteringWiki` extends conventional tagging by allowing tagging of structured objects, which are clusters of search results organized in a hierarchy.

Contributions.

- We introduce `ClusteringWiki`, the first framework for personalized clustering in the context of search result organization. Unlike existing methods that innovate on the automatic clustering procedure, it allows direct user editing of the clustering results through a Wiki interface.
- In `ClusteringWiki`, user preferences are reused among similar queries. They are also aggregated and shared among users as a mass-collaborative way of improving search result organization and search engine utility.
- We implement a prototype for `ClusteringWiki`, perform experimental evaluation and a user study, and maintain the prototype as a public Web service.

Outline. The rest of the paper is organized as follows. Section II reviews the related work. Section III overviews the `ClusteringWiki` framework. Section IV introduces the framework in detail. Section VI presents experiments and user study. Section VII concludes the paper.

II. Related Work

Clustering. Clustering is the process of organizing objects into groups or clusters so that objects in the same cluster are as similar as possible, and objects in different clusters are as dissimilar as possible. Clustering algorithms fall into two main categories, partitioning and hierarchical. Partitioning algorithms, such as k-means [21], produce a flat partition of objects without any explicit structure that relate clusters to each other. Hierarchical algorithms, on the other hand, produce a more informative hierarchy of clusters called a dendrogram. Hierarchical algorithms are either agglomerative (bottom-up) such as AGNES [17], or divisive (top-down) such as DIANA [17].

Clustering in IR. As a common data analysis technique, clustering has a wide array of applications in machine learning, data mining, pattern recognition, information retrieval, image analysis and bioinformatics [13], [7]. In information retrieval and Web search, document clustering was initially proposed to improve search performance by validating the *cluster hypothesis*, which states that documents in the same cluster behave similarly with respect to relevance to information needs [25].

In recent years, clustering has been used to organize search results, creating a cluster-based search interface as an alternative presentation to the ranked list interface. The list interface works fine for most navigational queries, but is less effective for informational queries, which account for the majority of Web queries [3], [26]. In addition, the growing scale of the Web and diversity of search results have rendered the list interface increasingly inadequate. Research has shown that the cluster interface improves user experience and search result quality [11], [34], [29], [15].

Search result clustering. One way of creating a cluster interface is to construct a static, off-line, pre-retrieval clustering of the entire document collection. However, this approach is ineffective because it is based on features that are frequent in the entire collection but irrelevant to the particular query [9], [27], [4]. It has been shown that query-specific, on-line, post-retrieval clustering, i.e., clustering search results, produces much superior results [11].

Scatter/Gather [11], [24] was an early cluster-based document browsing method that performs post-retrieval clustering on top-ranked documents returned from a traditional information retrieval system. The Grouper system [33], [34] (retired in 2000) introduced the well-known Suffix Tree Clustering (STC) algorithm that groups Web search results into clusters labeled by phrases extracted from snippets. It was also shown that using snippets is as effective as using whole documents. Carrot2 (www.carrot2.org) is an open source search result clustering engine that embeds STC as well as Lingo [23], a clustering algorithm based on singular value decomposition.

Other related work from the Web, IR and data mining communities exists. [35] explored supervised learning for extracting meaningful phrases from snippets, which are then used to group search results. [19] proposed a monothetic algorithm, where a single feature is used to assign documents to clusters and generate cluster labels. [31] investigated using past query history in order to better organize search results for future queries. [20] studied search result clustering for object-level search engines that automatically extract and integrate information on Web objects. [4] surveyed Web clustering engines and algorithms.

While all these methods focus on improvement in the automatic algorithmic procedure of clustering, `ClusteringWiki` employs a Wiki interface that allows direct user editing of the clustering results.

Clustering with user intervention. In machine learning, clustering is referred to as unsupervised learning. However, similar to `ClusteringWiki`, there are a few clustering frameworks that involve an active user role, in particular, semi-supervised clustering [1], [5] and interactive clustering [30], [14], [2] These frameworks are also motivated by the fact that clustering is too complex, and it is necessary to open the "black box" of the clustering procedure for easy understanding, steering and focusing. However, they differ from `ClusteringWiki` in that their focus is still on the clustering procedure, where they adopt a constraint clustering approach by transforming user feedback and domain knowledge into constraints (e.g., must-links and cannot-links) that are incorporated into the clustering procedure.

Search result annotation. Prototypes that allow user editing and annotation of search results exist. For example, U Rank by Microsoft (research.microsoft.com/en-us/ projects/urank) and Searchwiki by Google (googleblog.blog
spot.com/2008/11/searchwiki-make-search-your-own.html).
Rants [8] implemented a prototype with additional interesting features including the incorporation of both absolute and relative user preferences. Similar to `ClusteringWiki`, these works pursue personalization as well as a mass-collaborative way of improving search engine utility. The difference is that they use the traditional flat list, instead of cluster-based, search interface.

Tagging and social search. Social tagging, or collaborative tagging, allows users to create and associate objects with tags as a means of annotating and categorizing content. While users are primarily interested in tagging for their personal use, tags in a community collection tend to stabilize into power law distributions [10]. Collaborative tagging systems leverage this property to derive folksonomies and improve search [32]. In `ClusteringWiki` users tag clusters to organize search results, and the tags can be shared and utilized in the same way as in collaborative tagging. Since clusters are organized in a hierarchy, `ClusteringWiki` extends conventional tagging by allowing tagging of structured objects. Similar to tag suggestion in social tagging, the base clustering algorithm in `ClusteringWiki` provides suggested phrases for tagging clusters.

Social search is a mass-collaborative way of improving search performance. In contrast to established algorithmic or machine-based approaches, social search determines the relevance of search results by considering the content created or touched by users in the social graph. Example forms of user contributions include shared bookmarks or tagging of content with descriptive labels. Currently there are more than 40 such people-powered or community-powered social search engines, including Eurekster Swiki (www.eurekster.com), Mahalo (www.mahalo.com), Wikia
(answers.wikia.com/wiki/Wikianswers), and Google social search (googleblog.blogspot.com/2009/10/
introducing-google-social-search-i.html). Mass collaboration systems on the Web are categorized and discussed in [6].

III. Overview

In this section, we overview the main architecture and design principles of `ClusteringWiki`. Figure 2 shows the two key modules. The *query processing module* takes a query q and a set of stored user preferences as input to produce a cluster tree T that respects the preferences. The *cluster editing module* takes a cluster tree T and a user edit e as input to create/update a set of stored user preferences. Each user editing session usually involves a series of edits. The processing-editing cycle recurs over time.

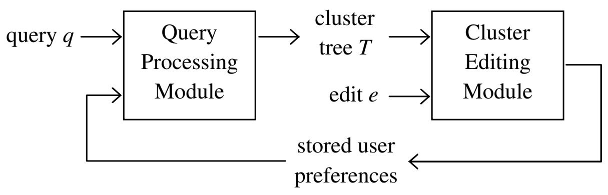


Fig. 2. Main architecture of `ClusteringWiki`.

Query processing. `ClusteringWiki` takes a query q from a user u and retrieves the search results R from a data source (e.g., Google). Then, it clusters R with a default clustering algorithm (e.g., frequent phrase hierarchical) to produce an initial cluster tree T_{init}. Then, it applies P, an applicable set of stored user preferences, to T_{init} and presents a modified cluster tree T that respects P.

Note that `ClusteringWiki` performs clustering. The modification should not alter R, the input data.

If the user u is logged-in, P will be set to $P_{q,u}$, a set of preferences for q previously specified by u. In case $P_{q,u} = \emptyset$, $P_{q',u}$ will be used on condition that q' is sufficiently close to q. If the user u is not logged-in, P will be set to $P_{q,U}$, a set of aggregated preferences for q previously specified by all users. In case $P_{q,U} = \emptyset$, $P_{q',U}$ will be used on condition that q' is sufficiently close to q.

In the cluster tree T, the internal nodes, i.e., non-leaf nodes, contain cluster labels and are presented on the left-hand *label panel*. Each label is a set of keywords. The leaf nodes contain search results, and the leaf nodes for a selected label are presented on the right-hand *result panel*. A search result can appear multiple times in T. The root of T represents the query q itself and is always labeled with *All*. When it is chosen, all search results will be presented on the result panel. Labels other than *All* represent the various, possibly overlapping, sub-topics of q. When there is no ambiguity, *internal node*, *label node*, *cluster label* and *label* are used interchangeably in the paper. Similarly, *leaf node*, *result node*, *search result* and *result* are used interchangeably.

Cluster editing. If logged-in, a user u can edit the cluster tree T for query q by creating, deleting, modifying, moving or copying nodes. User edits will be validated against a set C of consistency constraints before being written to $P_{q,u}$.

The set C contains predefined constraints that are specified on, for example, the size of clusters, the height of the tree and the length of labels. These constraints exist to maintain a favorable user interface for fast and intuitive navigation. The cluster tree T is *consistent* if it satisfies all the constraints in C.

By combining preferences in $P_{q,u}$ for all users who have edited the cluster tree T for query q, we obtain $P_{q,U}$, a set of aggregated preferences for query q. We use P_u to denote the collection of clustering preferences by user u for all queries, which is a set of sets of preferences such that $\forall q, P_{q,u} \in P_u$. We also use P_U to denote the collection of aggregated preferences by all users for all queries, which is a set of sets of aggregated preferences such that $\forall q, P_{q,U} \in P_U$. P_u and P_U are maintained over time and used by `ClusteringWiki` in processing queries for the user u.

Design principles. In a search result clustering engine, there are significant uncertainties from the data to the clustering algorithm. Wiki-facilitated personalization further adds substantial complications. Simplicity should be a key principle in designing such a complex system. `ClusteringWiki` adopts a simple yet powerful *path approach*.

With this approach, a cluster tree T is decomposed into a set of root-to-leaf *paths* that serve as independent editing components. A path always starts with *All* (root) and ends with some search result (leaf). In `ClusteringWiki`, maintenance, aggregation and enforcement of user preferences are based on simple path arithmetic. Moreover, the path approach is sufficiently powerful, being able to handle the finest user preference for a cluster tree.

In particular, each edit of T can be interpreted as operations on one or more paths. There are two primitive operations on a path p, *insertion* of p and *deletion* of p. A modification of p to p' is simply a deletion of p followed by an insertion of p'.

For each user u and each query q, ClusteringWiki maintains a set of paths $P_{q,u}$ representing the user edits from u for query q. Each path $p \in P_{q,u}$ can be either *positive* or *negative*. A positive path p represents an insertion of p, meaning that the user prefers to have p in T. A negative path $-p$ represents a deletion of p, meaning that the user prefers not to have p in T. Two *opposite* paths p and $-p$ will cancel each other out. The paths in $P_{q,u}$ may be added from multiple editing sessions at different times.

To aggregate user preferences for query q, ClusteringWiki first combines the paths in all $P_{q,u}$, $u \in U$, where U is the set of users who have edited the cluster tree of q. Then, certain statistically significant paths are selected and stored in $P_{q,U}$.

Suppose in processing query q, P is identified as the applicable set of paths to enforce. ClusteringWiki first combines the paths in P and the paths in T_{init}, where T_{init} is the initial cluster tree. Then, it presents the combined paths as a tree, which is the cluster tree T. The combination is straightforward. For each positive $p \in P$, if $p \notin T_{init}$, add p to T_{init}. For each negative $p \in P$, if $p \in T_{init}$, remove p from T_{init}.

Reproducibility. It is easy to verify that ClusteringWiki has the property of reproducing edited cluster trees. In particular, after a series of user edits on T_{init} to produce T, if T_{init} remains the same in a subsequent query, exactly the same T will be produced after enforcing the stored user preferences generated from the user edits on T_{init}.

IV. Framework

In this section, we introduce the ClusteringWiki framework in detail. In particular, we present the algorithms for the query processing and cluster editing modules and explain their main components.

A. Query Processing

Algorithm 1 presents the pseudocode for the query processing algorithm of ClusteringWiki. In the input, P_u and P_U are used instead of $P_{q,u}$ and $P_{q,U}$ for preference transfer purposes. In processing query q, it is likely that $P_{q,u} = \emptyset$ or $P_{q,U} = \emptyset$; then some applicable $P_{q',u} \in P_u$ or $P_{q',U} \in P_U$ can be used. The creation and maintenance of such user preferences will be discussed in Section IV-B. The output of the algorithm is a consistent cluster tree T.

Retrieving search results. Line 1 retrieves a set R of search results for query q from a chosen data source. The size of R is set to 50 by default and adjustable to up to 500. The available data sources include Google and Yahoo! Search APIs among others (see Section VI for details). ClusteringWiki retrieves the results via *multi-threaded parallel requests*, which are much faster than sequential requests.

The combined titles and snippets of search results retrieved from the sources are preprocessed. In order to extract phrases, we implemented our own tokenizer that identifies whether a token is a word, numeric, punctuation mark, capitalized, all caps, etc. We then remove non-textual tokens and stop words, using the stop word list from the Apache Snowball package (www.docjar.com/html/api/org/apache/ lucene/analysis/ snowball/SnowballAnalyzer.java.html). The tokens are then stemmed using the Porter (tartarus.org/ martin/PorterStemmer/) algorithm and indexed as terms. For each term, document frequency and collection frequency are computed and stored. A numeric id is also assigned to each term in the document collection in order to efficiently calculate document similarity, identify frequent phrases, etc.

Building initial tree. Line 2 builds an initial cluster tree T_{init} with a built-in clustering algorithm. ClusteringWiki provides 4 such algorithms: k-means flat, k-means hierarchical, frequent phrase flat and frequent phrase hierarchical. The hierarchical algorithms recursively apply their flat counterparts in a top-down manner to large clusters.

The k-means algorithms follow a strategy that generates clusters before labels. They use a simple approach to generate cluster labels from titles of search results that are the closest to cluster centers. In order to produce stable clusters, the typical randomness in k-means due to the random selection of initial cluster centers is removed. The parameter k is heuristically determined based on the size of the input.

Algorithm 1 *Query processing*

```
Input: $q$, $u$, $C$, $P_u$ and $P_U$: $q$ is a query. $u$ is a user. $C$ is a set of consistency constraints. $P_u$ is
    a collection of preferences by user $u$ for all queries, where $\forall q, P_{q,u} \in P_u$. $P_U$ is a collection of
    aggregated preferences for all queries, where $\forall q, P_{q,U} \in P_U$.
Output: $T$: a consistent cluster tree for the search results of query $q$.
 1: retrieve a set $R$ of search results for query $q$;
 2: cluster $R$ to obtain an initial cluster tree $T_{init}$;
 3: $P \leftarrow \emptyset$; //$P$ is the set of paths to be enforced on $T_{init}$
 4: if ($u$ is logged-in) then
 5:    $q' \leftarrow Trans(q, u)$;
 6:    if ($q' \neq NULL$) then
 7:       $P \leftarrow P_{q',u}$; //use applicable personal preferences
 8:    end if
 9: else
10:    $q' \leftarrow Trans(q, U)$;
11:    if ($q' \neq NULL$) then
12:       $P \leftarrow P_{q',U}$; //use applicable aggregated preferences
13:    end if
14: end if
15: $T \leftarrow T_{init}$; //initialize $T$, the cluster tree to present
16: clean $P$; //remove $p \in P$ if its result node is not in $R$
17: for each $p \in P$
18:    if ($p$ is positive) then
19:       $T \leftarrow T \cup \{p\}$; //add a preferred path
20:    else
21:       $T \leftarrow T - \{p\}$; //remove a non-preferred path
22:    end if
23: end for
24: $trim(T, C)$; //make $T$ consistent
25: $present(T)$; //present the set of paths in $T$ as a tree
```

The frequent phrase algorithms follow a strategy that generates labels before clusters. They first identify frequent phrases using a suffix tree built in linear time by Ukkonen's algorithm. Then they select labels from the frequent phrases using a greedy set cover heuristic, where at each step a frequent phrase covering the most uncovered search results is selected until the whole cluster is covered or no frequent phrases remain. Then they assign each search result r to a label L if r contains the keywords in L. Uncovered search results are added to a special cluster labeled *Other*. These algorithms are able to generate very meaningful cluster labels with a couple of heuristics. For example, a sublabel cannot be a subset of a superlabel, in which case the sublabel is redundant.

`ClusteringWiki` smoothly handles flat clustering by treating partitions as a special case of trees. The built-in clustering algorithms are meant to serve their basic functions. The focus of the paper is not to produce, but to modify, the initial cluster trees.

Determining applicable preferences. Lines 3 $\sim$ 14 determine P, a set of applicable paths to be enforced on T_{init}. Two cases are considered. If the user u is logged-in, P will use some set from P_u representing personal preferences of u (lines 4 $\sim$ 8). Otherwise, P will use some set from P_U representing aggregated preferences (lines 9 $\sim$ 14). The subroutine $Trans()$ determines the actual set to use if any.

The pseudocode of $Trans(q, u)$ is presented in Algorithm 2. Given a user u and a query q, it returns a query q', whose preferences stored in $P_{q',u}$ are applicable to query q. In the subroutine, two similarity

Algorithm 2 *Trans(q, u)*

```
Input: $q$, $u$ and $P_u$: $q$ is a query. $u$ is a user. $P_u$ is a collection of preferences by user $u$ for all queries,
    where $\forall q, P_{q,u} \in P_u$.
Output: $q'$: a query such that $P_{q',u}$ is applicable for $q$.
 1: if ($P_{q,u}$ exists) then
 2:     return $q$; //$u$ has edited the cluster tree of $q$
 3: else
 4:     find $q'$ s.t. $P_{q',u} \in P_u \wedge termSim(q,q')$ is the largest;
 5:     if $termSim(q,q') \geq \delta_{ts}$ then //$\delta_{ts}$ is a threshold
 6:         if $resultSim(q,q') \geq \delta_{rs}$ then //$\delta_{rs}$ is a threshold
 7:             $P_{q,u} \leftarrow P_{q',u}$; //copy preferences from $q'$ to $q$
 8:             return $q'$;
 9:         end if
10:     end if
11: end if
12: return $NULL$;
```

measures are used. *Term similarity*, $termSim(q,q')$, is the Jaccard coefficient that compares the terms of q and q'. *Result similarity*, $resultSim(q,q')$, is the Jaccard coefficient that compares the URLs of the top k (e.g., $k = 10$) results of q and q'. This calculation requires that the URLs of the top k results for q' be stored.

To validate q', both similarity values need to pass their respective thresholds δ_{ts} and δ_{rs}. Obviously, the bigger the thresholds, the more conservative the transfer. Setting the thresholds to 1 shuts down preference transfer. Instead of thresholding, another reasonable way of validation is to provide a ranked list of similar queries and ask the user for confirmation.

The subroutine in Algorithm 2 first checks if $P_{q,u}$ exists (line 1). If it does, preference transfer is not needed and q is returned (line 2). In this case, u has already edited the cluster tree for query q and stored the preferences in $P_{q,u}$.

Otherwise, the subroutine tries to find q' such that $P_{q',u}$ is applicable (lines 4 $\sim$ 11). To do so, it first finds q' such that $P_{q',u}$ exists and $termSim(q,q')$ is the largest (line 4). Then, it continues to validate the applicability of q' by checking if $termSim(q,q')$ and $resultSim(q,q')$ have passed their respective thresholds (lines 5 sim 6). If so, user preferences for q' will be copied to q (line 7), and q' will be returned (line 8). Otherwise, $NULL$ will be returned (line 11), indicating no applicable preferences exist for query q.

The preference copying (line 7) is important for the correctness of `ClusteringWiki`. Otherwise, suppose there is a preference transfer from q' to q, where $P_{q,u} = \emptyset$ and $P_{q',u}$ has been applied on T_{init} to produce T. Then, after some editing from u, T becomes T' and the corresponding edits are stored in $P_{q,u}$. Then, this $P_{q,u}$ will be used the next time the same query q is issued by u. However, $P_{q,u}$ will not be able to bring an identical T_{init} to the expected T'. It is easy to verify that line 7 fixes the problem and ensures reproducibility.

$Trans(q,U)$ works in the same way. Preference transfer is an important component of `ClusteringWiki`. Cluster editing takes user effort and there are an infinite number of queries. It is essential that such user effort can be properly reused.

Enforcing applicable preferences. Back to Algorithm 1, lines 15 $\sim$ 23 enforce the paths of P on T_{init} to produce the cluster tree T. The enforcement is straightforward. First P is cleaned by removing those paths whose result nodes are not in the search result set R (line 16). Recall that `ClusteringWiki` performs clustering. It should not alter the input data R. Then, the positive paths in P are the ones u prefers to see in T, thus they are added to T (lines 18 $\sim$ 19). The negative paths in P are the ones u

prefers not to see in T, thus they are removed from T (lines 20 $\sim$ 21). If $P = \emptyset$, there are no applicable preferences and T_{init} will not be modified.

Trimming and Presenting T. The cluster tree T must satisfy a set C of predefined constraints. Some constraints maybe violated after applying P to T_{init}. For example, adding or removing paths may result in small clusters that violate constraints on the size of clusters. In line 24, subroutine $trim(T, C)$ is responsible for making T consistent, e.g., by re-distributing the paths in the small clusters. We will discuss the constraint set C in detail in Section IV-B.

In line 25, subroutine $present(T)$ presents the set of paths in T as a cluster tree on the search interface. The labels can be expanded or collapsed. The search results for a chosen label are presented in the result panel in their original order when retrieved from the source. Relevant terms corresponding to current and ancestor labels in search results are highlighted.

Sibling cluster labels in the label panel are ordered by lexicographically comparing the lists of original ranks of their associated search results. For example, let A and D be two sibling labels as in Figure 3, where A contains P_1, P_2, P_3 and P_4 and D contains P_1 and P_5. Suppose that i in P_i indicates the original rank of P_i from the source. By comparing two lists $< 1, 2, 3, 4 >$ and $< 1, 5 >$, we put A in front of D. "Other" is a special label that is always listed at the end behind all its siblings.

Discussion. As [16] suggested, the subset of web pages visited by employees in an Enterprise is centered around the company's business objectives. Additionally, employees share a common vocabulary describing the objects and tasks encountered in day to day activities. `ClusteringWiki` can be even more effective in this environment as user preferences can be better aggregated and utilized.

B. Cluster Editing

Before explaining the algorithm handling user edits, we first introduce the essential consistency constraints for cluster trees and the primitive user edits.

Essential consistency constraints. Predefined consistency constraints exist to maintain a favorable user interface for fast and intuitive navigation. They can be specified on any structural component of the cluster tree T. In the following, we list the essential ones.

- *Path constraint*: Each path of cluster tree T must start with the root labeled All and end with a leaf node that is a search result. In case there are no search results returned, T is empty without paths.
- *Presence constraint*: Each initial search result must be present in T. It implies that deletion of paths should not result in absence of any search result in T.
- *Homogeneity constraint*: A label node in T must not have heterogeneous children that combine cluster labels with search results. This constraint is also used in other clustering engines such as Clusty and Carrot2.
- *Height constraint*: The height of T must be equal or less than a threshold, e.g., 4.
- *Label length constraint*: The length of each label in T must be equal or less than a threshold.
- *Branching constraint*: We call a label node a *bottom label node* if it directly connects to search results. Each non-bottom label node must have at least T_n children. Each non-special bottom label node must have at least T_m children. $Other$ is a special bottom label node that may have less than T_m children. All, when being a bottom label, could also have less than T_m children in case there are insufficient search results. By default both T_n and T_m are set to 2 in `ClusteringWiki` as in Clusty.

Primitive user edits. `ClusteringWiki` implements the following categories of atomic primitive edits that a logged-in user can initiate in the process of tree editing. Each edit e is associated with P_e and NP_e, the set of paths to be inserted to the tree and the set of paths to be deleted from the tree after e.

- e_1: copy a label node to another non-bottom label node as its child. Note that it is allowed to copy a parent label node to a child label node.
  Example: in Figure 3, we can copy D to A. For this edit, $P_e = \{All \rightarrow A \rightarrow D \rightarrow P_1, All \rightarrow A \rightarrow D \rightarrow P_5\}$. $NP_e = \emptyset$ for any edit of this type.

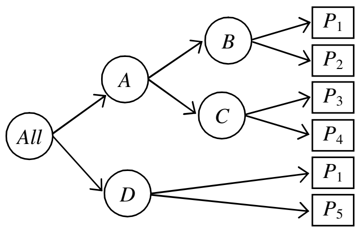


Fig. 3. Example cluster tree.

Algorithm 3 *Cluster editing*

Input: q, u, T, C, $P_{q,u}$, $P_{q,U}$ and e: q is a query. u is a user. T is a cluster tree for q. C is a set of consistency constraints for T. $P_{q,u}$ is a set of paths representing the preferences by u for q. $P_{q,U}$ is a set of paths representing the aggregated preferences for q. e is an edit by u on T.

Output: updated T, $P_{q,u}$ and $P_{q,U}$

```
1: if (pre-validation fail) then
2:     return;
3: end if
4: identify $P_e$;
5: identify $NP_e$;
6: if (validation fail) then
7:     return;
8: end if
9: update $T$;
10: add $P_e$ as positive paths to $P_{q,u}$;
11: add $NP_e$ as negative paths to $P_{q,u}$;
12: update $P_{q,U}$;
```

- e_2: copy a result node to a bottom label node.
  Example: in Figure 3, we can copy P_3 to D, but not to A, which is not a bottom label node. For this edit, $P_e = \{All \to D \to P_3\}$. $NP_e = \emptyset$ for any edit of this type.
- e_3: modify a non-root label node.
  Example: in Figure 3, we can modify D to E. For this edit, $P_e = \{All \to E \to P_1, All \to E \to P_5\}$ and $NP_e = \{All \to D \to P_1, All \to D \to P_5\}$.
- e_4: delete a non-root node, which can be either a label node or a result node.
  Example: in Figure 3, we can delete P_5. For this edit, $NP_e = \{All \to D \to P_5\}$. $P_e = \emptyset$ for any edit of this type.
- e_5: create a label node, which can be either a non-bottom or bottom label node. In particular, recursive creation of non-bottom labels is a way to add levels to cluster trees.
  Example: in Figure 3, we can add E as parent of D. For this edit, $P_e = \{All \to E \to D \to P_1, All \to E \to D \to P_5\}$ and $NP_e = \{All \to D \to P_1, All \to D \to P_5\}$.

The editing framework results in several *favorable properties*. Firstly, the primitive user edits are such that, with a series of edits, a user can produce *any* consistent cluster tree. Secondly, since e_1 only allows a label node to be placed under a non-bottom node and e_2 only allows a result node to be placed under a bottom node, the homogeneity constraint will not be violated after any edit given the consistency of T before the edit. Thirdly, the framework uses *eager* validation, where validation is performed right after each edit, compared to *lazy* validation, where validation is performed in the end of the editing process. Eager validation is more user-friendly and less error-prone in implementation.

Note that, user editing can possibly generate *empty labels*, i.e., labels that do not contain any search results and thus not on any path. Such labels will be trimmed.

To add convenience, `ClusteringWiki` also implements several other types of edits. For example, move (instead of copy as in e_1) a label node to another non-bottom label node as its child, or move (instead of copy as in e_2) a result node to a bottom label node. Such a move edit can be considered as a copy edit followed by a delete edit.

Editing algorithm. Algorithm 3 presents the pseudocode of the cluster editing algorithm in `ClusteringWiki` for a single edit e, where e can be any type of edit from e_1 to e_4.

Lines 1 $\sim$ 3 perform pre-validation of e to see if it is in violation of consistency constraints. Violations can be caught early for certain constraints on certain edits, for example, the label length constraint on e_1 type of edits. If pre-validation fails, the algorithm returns immediately.

Otherwise, the algorithm continues with lines 4 $\sim$ 5 that identify P_e and NP_e. Then, lines 6 $\sim$ 8 perform full validation of e against C, the set of consistency constraints. If the validation fails, the algorithm returns immediately.

Otherwise, e is a valid edit and T is updated (line 9). Then, the personal user preferences are stored by adding P_e and NP_e to $P_{q,u}$ as positive paths and negative paths respectively (lines 10 $\sim$ 11). In adding these paths, the opposite paths in $P_{q,u}$ cancel each other out. In line 12, the aggregated preferences stored in $P_{q,U}$ are updated. We further discuss preference aggregation in the following.

Preference sharing. Preference sharing in `ClusteringWiki` is in line with the many social-powered search engines as a mass-collaborative way of improving search utility. In `ClusteringWiki`, U is considered as a special user and $P_{q,U}$ stores the aggregated user preferences.

In particular, we use $P^0_{q,U}$ to record the paths specified for query q by all users. Each path $p \in P^0_{q,U}$ has a *count* attribute, recording the total number of times that p appears in any $P_{q,u}$. All paths in $P^0_{q,U}$ are grouped by leaf nodes. In other words, all paths that end with the same search result are in the same group. For each group, we keep track of two *best* paths: a positive one with the most count and a negative one with the most count. We mark a best path if its count passes a predefined threshold. All the marked paths constitute $P_{q,U}$, the set of aggregated paths that are used in query processing. Note that, here `ClusteringWiki` adopts a conservative approach, making use of at most one positive path and one negative path for each search result.

Editing interface. Cluster editing in `ClusteringWiki` is primarily available through context menus attached to label and result nodes. Context menus are context aware, displaying only those operations that are valid for the selected node. For example, the *paste result* operation will not be displayed unless the selected node is a bottom label node and a result node was previously copied or cut. This effectively implements pre-validation of cluster edit operations by not allowing the user to choose invalid tasks.

Users can drag and drop a result node or cluster label in addition to cutting/copying and pasting to perform a move/copy operation. A label node will be tagged with an icon if the item being dragged can be pasted within that node. An item that is dropped outside a label node in which it could be pasted simply returns to its original location.

V. Implementation

`ClusteringWiki` was implemented as an AJAX-enabled web application running in a Java Enterprise Edition 1.5 container. In this section we detail the choices made in implementing the system.

A. Query processing

`ClusteringWiki` search requests are sent to the server via AJAX and expect in return a JSON structure including both the search result set and cluster tree data. The received data is interpreted to display an in-page cluster tree, attach appropriate tree functionality, and display results contained in the root cluster node.

Retrieving query results. In order to easily test `ClusteringWiki` with multiple search engines and data sources, we created a web service, named *AbstractSearch*, responsible for hiding query execution details. *AbstractSearch* runs as a separate Java Enterprise Edition application and interprets received query parameters into parameters specific to the requested search source. For example, Google AJAX Search API[2] expects a zero-based first requested result parameter, while Yahoo! Search API[3] expects a one-based equivalent parameter. The Google API can retrieve a maximum of 8 results per request and a total of 64 results per query, while the Yahoo! API can retrieve 100 results per request and a total of 1000 results per query. *AbstractSearch* retrieves the results via one or more parallel requests to the search source and returns the entire requested result set at once as either XML or JSON data. The *multi-threaded parallel execution* of requests allows 500 Yahoo! results (executed using 5 Yahoo! requests) to be returned in less than 2 seconds instead of the 8 seconds it would take if the requests were executed sequentially.

During our testing we found that the Yahoo! Search API sometimes returns duplicate results among multi-page requests for the same query (ex: last result from the first page of results is repeated as the first result of the second page). *AbstractSearch* corrects this issue by removing identified duplicates from the returned result set. The returned result set in these cases will contain less than the requested number of results.

Preprocessing. `ClusteringWiki` analyzes the result set retrieved from *AbstractSearch* and builds a collection context data structure used in later processing. The combined title and snippet fields of a search result are used to textually represent a search result document. Each result document is first broken into a bag of lowercase words. After removing removing non-textual characters and stop words, the remaining words are stemmed using the Porter[4] algorithm creating a list of document terms. The stop word list used is that from the Apache Snowball[5] package. The terms are then added to an index of collection *terms* spanning all retrieved search result documents, and each term is associated with a numeric index id. Document frequency and collection frequency are also computed for each term.

Similar to the Apache Lucene tokenizers, our tokenization process identifies additional information about each token which it stores as bitwise flags in a short value. We identify whether a token is a word, numeric, or punctuation mark, or whether a word token is capitalized, all caps, starting or ending with a punctuation mark. The token attributes are used to identify the start and end of phrases within the document text, which are stored in an array as pairs of document text index integers.

Further textual processing is done using the assigned numeric term and document ids to increase efficiency. A *ClusterDocument* data structure is used to encompass all necessary information for a result document being clustered, including term ids for terms in the given document, term counts, normalized term frequencies, and term and word phrase boundaries.

Clustering results. `ClusteringWiki` clusters documents using one of four pre-defined clustering algorithms: modified k-means, modified hierarchical k-means, frequent phrase flat, or frequent phrase hierarchical.

Unlike the standard version of k-means, `ClusteringWiki` creates consistent clusters over the same data set by choosing the same initial cluster centers rather than random ones. Cluster centers are chosen as a function of the number of retrieved results, after first pre-ordering the results using a result-specific parameter (ex: url). The modified hierarchical k-means uses stable functions, based on the cluster level, parent cluster size, and tree height constraint, to decide whether a parent cluster should be sub-clustered and how many initial cluster centers should be chosen for the sub-cluster. The k-means based clusters are assigned the title of the document closest to the cluster centroid as the cluster label.

Frequent phrase flat and frequent phrase hierarchical algorithms are based on identifying frequent phrases within the document text. We use a suffix tree data structure built using Ukkonen's linear time online construction algorithm to identify term frequent phrases within the combined text of all documents being clustered. We then assign to that cluster all documents from the collection being clustered that

[2]http://code.google.com/apis/ajaxsearch/

[3]http://developer.yahoo.com/search/web/V1/webSearch.html

[4]http://tartarus.org/ martin/PorterStemmer/

[5]http://www.docjar.com/html/api/org/apache/lucene/analysis/snowball/SnowballAnalyzer.java.html

contain any of the label terms. The details of the frequent phrase algorithm, as applied for each level of clustering, are as follows:

1) We build an integer sequence from all the initial term phrases identified in each document being clustered, noting phrase boundaries with unique negative integers, which are characters outside of the document alphabet. We also note document boundaries in a separate integer stack.
2) We apply Ukkonen's linear time online suffix tree construction algorithm to create a suffix tree for the integer sequence. We also keep track of document ids for each of the phrase suffixes entered in the suffix tree.
3) We walk the suffix tree, identifying and retrieving frequent phrases with a given minimum and maximum length and minimum support. For our current implementation we experimentally chose to use minimum term phrase length 2, maximum term phrase length 5, and minimum document support 2. For each phrase we also retrieve from the suffix tree a bit set representing all the documents that contain the given phrase.
4) We greedily retrieve the phrase with the highest coverage of uncovered documents within the collection being clustered, ignoring phrases that are comprised of a subset of terms of any parent label. We consider the root cluster label to be the executed query. The bit set representation of covered documents for each phrase allows for fast set based operations when computing phrase coverage among the set of uncovered documents.
5) For each label identified, we build a cluster and assign it all documents containing any of the terms in the label.
6) We greedily choose a word phrase label for each cluster by choosing the word phrase associated with the cluster label term phrase that has the highest support in the cluster documents.
7) When all qualified phrases are processed, any remaining uncovered documents are added to an additional cluster labeled *Other*.
8) We heuristically choose to subcluster a given cluster if it contains more than 5 documents and the path of the child cluster does not violate our given tree *height constraint*. However, we choose not to subcluster if subclustering produces less than two clusters.

Our process for retrieving frequent phrases from the text collection is similar with that used in Carrot2[6], except they retrieve frequent *word* phrases and then apply certain heuristic scoring methods to prioritize phrases retrieved from the suffix tree. The remainder of the clustering algorithm is also quite different in Carrot2.

`ClusteringWiki` uses a *Cluster* data structure to encompass cluster related information such as documents contained in the cluster, cluster label and term label, and references to the cluster's parent and possible child clusters. In the case of flat clustering all identified clusters are made children of a root cluster labeled *All*.

Retrieving and merging preferences. `ClusteringWiki` cluster preferences are stored in a MySQL database described by the schema below. Fields and tables not related to cluster editing have been omitted.

users (*id*: number, *email*: string)

queries (*id*: number, *user_id*: number, *executed_on*: date, *service*: string, *query_text*: string, *parsed_query_text*: string)

query_responses (*id*: number, *query_id*: number, *url*: string)

cluster_edits (*id*: number, *query_id*: number, *clustering_algo*: number, *path1*: string, *path2*: string, *path3*: string, *path4*: string, *path5*: string, *cardinality*: number, *executed_on*: date)

cluster_edits_all (*id*: number, *query_id*: number, *clustering_algo*: number, *path1*: string, *path2*: string, *path3*: string, *path4*: string, *path5*: string, *cardinality*: number, *executed_on*: date)

[6]http://project.carrot2.org/

Users are assigned unique numeric ids upon account creation. Executed queries are stored in the *queries* table and associated with a given user id. The query text is added to a full-text search index and additionally a list of stemmed terms contained in the query text is kept. A referential integrity constraint exists between the *queries* and *users* tables via the *user_id* field.

The top k (e.g., $k = 20$) results of an executed query are stored in the *query_responses* table and associated with the executed query id. If a query already existed in the database, its set of responses are updated if the query was last executed more than a day before. A referential integrity constraint exists between the *query_responses* and *queries* tables via the *query_id* field.

In order to test `ClusteringWiki` with multiple clustering algorithms we have associated a user preference $P_{q,u}$ with the chosen clustering algorithm in addition to the executed query q and the logged in user u. A preference is stored as a *cluster_edits* tuple containing the path of the preference and an associated cardinality (ex: $+1$ or -1), signifying a positive ($p \in P_u$) or negative ($p \in NP_u$) preference. A path is represented by the set of cluster labels along the tree path from the $root$ node to the bottom label node containing the result node (leaf node) in the path, along with the label of the result node. The $root$ node label is assumed and ignored when storing a path since all preferences would contain this label. Given a relatively low maximum tree height h constraint, we have chosen to keep the path in $h-1$ $path_x$ fields within the *cluster_edits* tuple. Alternative approaches for storing a path without a height constraint include adjacency list and nested set storage models[7]. A referential integrity constraint exists between the *cluster_edits* and *queries* tables and between the *cluster_edits_all* and *queries* tables via the *query_id* field.

Query processing is time-critical. Aiming to minimize query response time, `ClusteringWiki` defines a special user *all* and stores aggregated preferences from all users in $P_{q,all}$. To make the preferences in $P_{q,all}$ ready to use for query processing, the aggregation is done during cluster editing. The cluster editing process is interactive. Aggregation is processed in the background, while the user is performing cluster edits on the interface, causing none or negligible waiting time for the user.

Preferences for the user *all* are incrementally stored in the *cluster_edits_all* table whenever a preference $P_{q,u}$ is stored for any application user u. The *cluster_edits_all* table structure is identical to that of the *cluster_edits* table. Retrieving a set of preferences for the current search query becomes trivial: execute a database query for the set of preferences associated with the given query q, the chosen clustering algorithm, and the logged in user u, or the user *all* if the user is not logged in. Changes to *cluster_edits_all* tuples are efficiently executed via database triggers attached to the *cluster_edits* table.

In the event that the initial database query does not return any results, `ClusteringWiki` searches for a similar query q', first via a MySQL full-text search for the queried text, and then by searching for the conjuncted stemmed query text terms. If any matching similar queries related to the chosen clustering algorithm and appropriate user are found, they are then checked against the δ_{ts} term similarity and δ_{rs} result similarity thresholds. `ClusteringWiki` retrieves and merges the preferences of the first similar query q' that passes these tests into the initial cluster.

The retrieved set of preferences is applied to the cluster hierarchy sequentially, first adding all positive cardinality preferences and then removing any negative preferences as long as they do not violate any pre-defined cluster constraints. A reverse-lookup index of result labels to cluster node paths is used to speed up negative preference validations.

Ordering cluster labels. After clustering has finished, the clusters are ordered by ascending minimum document ids contained in each cluster. The final cluster T, along with the original search result set are then added to a JSON structure and returned to the browser.

Displaying the cluster. The JSON data received from the server is passed on to a JavaScript object, named *EditableClusterTree*, which encapsulates all client-side functionality of the editable search result cluster. Using a JavaScript object allows creating an efficient stateful representation of the cluster tree and its operations. *EditableClusterTree* first builds an internal cluster node object hierarchy from the data

[7]http://dev.mysql.com/tech-resources/articles/hierarchical-data.html.

received. The internal cluster representation computes and stores additional node information such as references to parent and children nodes, current node path, level, and maximum depth, which are used to efficiently validate and execute cluster operations. Paths are stored internally as sets of numeric cluster index ids to optimize node retrieval. Furthermore, the internal cluster tree stores result nodes as arrays of result index ids within T bottom label nodes, shortening the internal tree height by one level and improving efficiency of subtree operations.

EditableClusterTree then creates an HTML unordered list representation of the cluster tree and a set of HTML result node structures, which it appends to the page within specified $< div >$ elements. Cluster node labels and result node titles along paths that were added due to a previous cluster edit are tagged with a red asterisk. CSS styles are used to make the cluster structure appear as a tree. JavaScript events are added to individual tree and result nodes to enable *EditableClusterTree* functionality.

Highlighting. Relevant terms corresponding to currently selected and ancestor labels in search results are highlighted. The highlighting function stems all label words to create label terms using the same stemming algorithm used during clustering (the Porter stemming algorithm) and then uses JavaScript regular expressions to match and highlight each term found within the document text.

B. Cluster editing

Cluster editing in `ClusteringWiki` involves editing the in-page cluster using the *EditableClusterTree* object and storing any cluster path changes resulting from executed operations. Editing is primarily available through context menus attached to cluster and result nodes displayed in the page.

Context menus. `ClusteringWiki` context menus display only those operations that are valid for the cluster or result node that was right-clicked. For example, the *Paste result* operation will not be displayed unless the node right-clicked is a bottom label node and a result node was previously copied or cut. This effectively implements pre-validation of cluster edit operations by not allowing the user to choose invalid tasks. Edit operations are only displayed when the application user is logged in.

The *EditableClusterTree* object displays a different context menu depending on the current cluster tree edit mode: an *edit disabled* menu, an *edit enabled* menu, or a *browse only* menu. Menus contain all possible operations for the given mode. Each operation is pre-validated via internal *EditableClusterTree* methods and only displayed if the operation passes validation. *EditableClusterTree* takes advantage of its internal cluster tree representation to efficiently pre-validate cluster operations.

Operation validation. Additional operation validation methods are executed after an operation has been invoked but before effectively executing the operation. These include validation methods that cannot be executed during pre-validation (ex: checking a modified node label is valid), or ones that can be slow and would delay the context menu from being displayed (ex: checking a node being copied does not already exist in the node being copied to). When a validation method fails, a message is displayed above the cluster tree alerting the user to the cause of the failure.

Operation execution. Once an operation has been validated, *EditableClusterTree* computes the set of positive paths and negative paths caused by the given operation. Positive paths are assigned cardinality $+1$ and negative paths are assigned cardinality -1. If the set of path changes is not empty it is encoded as a binary upload and sent to the server for processing via AJAX. For each path received, `ClusteringWiki` updates a preference for both the logged in user and the *all* user. For either application user, if a tuple for the preference does not already exist it is inserted and given the received cardinality. If the path already existed in the database, the received cardinality is added to the existing preference cardinality. The preference associated with the logged in user's query ($P_{q,u}$) is restricted to a cardinality within the set $\{-1, 1\}$. Paths with a cardinality of 0 after an update are deleted in order to improve database efficiency.

The server returns a confirmation message to the browser when all paths have been successfully stored. *EditableClusterTree* then modifies the in-page HTML tree and selected result set to display the effects of the executed operation.

Convenience features. We have implemented several `ClusteringWiki` convenience operations that increase the usability of the application. In addition to copying a cluster or result node, executed via a copy followed by a paste operation, we allow users to also move a node via cut and paste. Additionally, double-clicking on any label node expands/collapses the clicked tree node and all its children.

Users can see what the cluster tree would look like without edits by selecting *Show tree w/o edits* from the *root* label context menu while being logged in. Once selected, the cluster tree is re-built without adding or subtracting any user preferences. The re-built tree is displayed in *browse only* mode, without access to any cluster editing operations.

Users can drag and drop a result node or cluster label in addition to cutting/copying and pasting to perform a move/copy operation. A label node will be tagged with an icon if the item being dragged can be pasted within that node. An item that is dropped outside a label node in which it could be pasted simply returns to its original location.

VI. Evaluation

`ClusteringWiki` was implemented as an AJAX-enabled Java Enterprise Edition 1.5 application. The prototype is maintained on an average PC with Intel Pentium 4 3.4 GHz CPU and 4Gb RAM running Apache Tomcat 6. We have conducted a comprehensive experimental evaluation detailed below.

A. Methodology and Metrics

We performed two series of experiments: system evaluation and utility evaluation. The former focused on the correctness and efficiency of our implemented prototype. The latter, our main experiments, focused on the effectiveness of `ClusteringWiki` in improving search performance.

Data sources. Multiple data sources were used in our empirical evaluation, including Google AJAX Search API (code.google.com/apis/ajaxsearch), Yahoo! Search API (developer.yahoo.com/search/web/webSearch.html), and local Lucene indexes built on top of the New York Times Annotated Corpus [28] and several datasets from the TIPSTER (disks 1-3) and TREC (disks 4-5) collections (www.nist.gov/ tac/data/data_desc.html). The Google API can retrieve a maximum of 8 results per request and a total of 64 results per query. The Yahoo! API can retrieve a maximum of 100 results per request and a total of 1000 results per query. Due to user licence agreements, the New York Times, TIPSTER and TREC datasets are not available publicly.

System evaluation methodology. For system evaluation of `ClusteringWiki`, we focused on *correctness* and *efficiency*. We tested the correctness by manually executing a number of functional and system tests designed to test every aspect of application functionality. These tests included cluster reproducibility, edit operation pre-validations, cluster editing operations, convenience features, applying preferences, preference transfer, preference aggregation, etc. `ClusteringWiki` is a multi-tiered system with interactive components written in multiple programming languages. As such, standard unit tests are not as helpful in determining the proper functionality.

In order to have repeatable search results for same queries, we used the stable New York Times data source. We chose queries that returned at least 200 results.

We evaluated system efficiency by monitoring query processing time in various settings. In particular, we considered:

- 2 data sources: Yahoo! and New York Times
- 5 different numbers of retrieved search results: 100, 200, 300, 400, 500
- 2 types of clusterings: flat (F) and hierarchical (H)

For each of the combinations, we executed 5 queries, each twice. The queries were chosen such that at least 500 search results would be returned. For each query, we monitored 6 portions of execution that constitute the total query response time:

- Retrieving search results
- Preprocessing retrieved search results

- Initial clustering by a built-in algorithm
- Applying preferences to the initial cluster tree
- Presenting the final cluster tree
- Other (e.g., data transfer time between server and browser)

For the New York Times data source, the index was loaded into memory to simulate the server side search engine behavior. The time spent on applying preferences depends on the number of applicable stored paths. For each query, we made sure that at least half the number of retrieved results existed in a modified path, which is a practical upper-bound on the number of user edits on the clusters of a query.

Utility evaluation methodology. For utility evaluation, we focused on the *effectiveness* of `ClusteringWiki` in improving search performance, in particular, the time users spent to locate a certain number of relevant results. The experiments were conducted through a user study with 22 *paid* participants. Study participation was advertised within the Computer Science department at our University and users were chosen on a first-come first-serve basis. The participants were primarily undergraduate, with a few graduate, college students.

We compared 4 different search result presentations:

- Ranked list (RL): search results were not clustered and presented as a traditional ranked list.
- Initial clustering (IC): search results were clustered by a default built-in algorithm (frequent phrase hierarchical).
- Personalized clustering (PC): search result clustering was personalized by a logged-in user after a series of edits, taking on average 1 and no more than 2 minutes per query.
- Aggregated clustering (AC): search result clustering was based on aggregated edits from on average 10 users.

Navigational queries seek the website or home page of a single entity that the user has in mind. The more common [3], [26] informational queries seek general information on a broad topic. The ranked list interface works fine for the former in general but is less effective for the latter, which is where clustering can be helpful [22]. In practice, a user may explore a varied number (e.g., 5 or 10) of relevant results for an informational query. Thus, we considered 2 types of informational queries. In addition, we argue that for some *deep* navigational queries where the desired page "hides" deep in a ranked list, clustering can still be helpful by skipping irrelevant results. Thus, we also considered such queries:

- R_{10}: Informational. To locate any 10 relevant results.
- R_5: Informational. To locate any 5 relevant results.
- R_1: Navigational. To locate 1 pre-specified result.

For each query type, 10 queries were executed, 5 on Google results and 5 on the AP Newswire dataset from disk 1 of the TIPSTER corpus. The AP Newswire queries were chosen from TREC topics 50-150, ensuring that they returned at least 15 relevant results within the first 50 results. For R_1 queries, the topic descriptions were modified to direct the user to a single result that is relatively low-ranked to make the queries "deep". Google queries were chosen from topics that participants were familiar with. All queries returned at least 50 results.

Below we present the chosen queries for each data set. Each query was presented with a description of the task to be performed. Informational queries were also followed by a narrative further explaining the information need for the current task.

AP Newswire, R_{10} and R_5:

- *Query*: Rail Strikes
  Description: Find relevant pages that predict or anticipate a rail strike or report an ongoing rail strike.
  Narrative: A relevant document will either report an impending rail strike, describing the conditions which may lead to a strike, or will provide an update on an ongoing strike. To be relevant, the document will identify the location of the strike or potential strike. For an impending strike, the document will report the status of negotiations, contract talks, etc. to enable an assessment of the probability of a strike. For an ongoing strike, the document will report the length of the strike to the current date and the status of negotiations or mediation.

- *Query*: Surrogate Motherhood
  Description: Find relevant pages that report judicial proceedings and opinions on contracts for surrogate motherhood. After tagging relevant results, please edit the result clusters so that you can find those relevant results easier in the future.
  Narrative: A relevant document will report legal opinions, judgments, and decisions regarding surrogate motherhood and the custody of any children which result from surrogate motherhood. To be relevant, a document must identify the case, state the issues which are or were being decided and report at least one ethical or legal question which arises from the case.
- *Query*: Nuclear Proliferation
  Description: Find relevant pages that discuss efforts by the United Nations or those nations currently possessing nuclear weapons to control the proliferation of nuclear weapons capabilities to the non-nuclear weapons states.
  Narrative: A relevant document will report on efforts by the UN's International Atomic Energy Agency to monitor compliance with the Nuclear Non-proliferation Treaty, or, report on efforts by the United States, Britain, France, USSR, India, or China to control the transfer of technology, equipment, materials, or delivery systems to nations suspected of nuclear weapons development programs, or, a relevant document will report any nuclear activities by Argentina, Brazil, Iraq, Israel, North Korea, Pakistan, or South Africa (all suspected proliferators).
- *Query*: Non-commercial Satellite Launches
  Description: Find relevant pages that provide data on launches worldwide of non-commercial space satellites.
  Narrative: A relevant document will provide information which helps identify the purpose and capabilities of satellites launched anywhere in the world which might have non-commercial applications. Accordingly, periodic launches of INTELSAT communications satellites are clearly NOT relevant. On the other hand, data on launches of military communication or intelligence sensor satellites are definitely relevant. Potentially useful data on any given launch would be press comment as to whether the satellite has a military, scientific, or intelligence purpose, type of launch vehicle employed, reported satellite configuration, launch trajectory and speed, projected orbit, etc.
- *Query*: U.S. Political Campaign Financing
  Description: Find relevant pages that report how U.S. politicians finance their election campaigns and/or moves to ”reform” campaign finance practices.
  Narrative: A relevant document will show how U.S. politicians (federal, state, or local – individually or as a group) pay for their election campaigns, the role played by ”special interests” and contributors in the electoral process, allegations or evidence of campaign contributions buying political favors, and/or proposals to limit the cost of campaigns or ”reform” electoral finance practices.

AP Newswire, R_1:

- *Query*: South African Sanctions
  Description: Find document AP900622-0043, discussing sanctions against South Africa not being the focus of Nelson Mandela's visit to New York.
- *Query*: Insider Trading
  Description: Find document AP900729-0090, discussing the U.S. cracking down on international insider traiding.
- *Query*: MCI
  Description: Find document AP901017-0153, discussing a $168 million MCI quarterly net loss partially due to the acquisition of Telecom USA.
- *Query*: Greenpeace
  Description: Find document AP901112-0111, reporting Greenpeace attempts to stop UK nuclear tests in Nevada.
- *Query*: Iran-Contra Affair
  Description: Find document AP900301-0130, which talks about a list of 140 people that could testify in the case against the former national security adviser, including the former president Reagan.

Google, R_{10} and R_5:

- *Query*: Kanye West albums
  Description: Find relevant pages about Kanye's newest album that was released this month.
  Narrative: A relevant document will talk about Kanye West's newest album being released in Oct 2010 or refer to it by name: My Beautiful Dark Twisted Fantasy.
- *Query*: Iron Man
  Description: Find relevant pages that talk about Iron Man, the movie.
  Narrative: A relevant result may contain information about either Iron Man or its sequel, or about any of the main actors within the movie.
- *Query*: Longhorns
  Description: Find relevant pages about the University of Texas football team, the Longhorns.
  Narrative: A relevant result may contain information about the team, its players, or games played recently or upcoming.
- *Query*: The office

Description: Find relevant pages about the show the Office and its cast members.
Narrative: A relevant page should describe the show or offer information about its cast. A page simply allowing one to see episodes of the show should not be considered relevant.

- *Query*: Justin Bieber albums
  Description: Find relevant pages about any of Justin Bieber's albums.
  Narrative: Relevant pages will name and describe a Justin Bieber album.

Google, R_1:

- *Query*: marathon
  Description: Find the Runner's World home page, which has information about running shoes, marathon training, and racing.
- *Query*: $X\ University$
  Description: Find the page for the graduate college at $X\ University$.
- *Query*: Longhorns
  Description: Find the page for the Texas Longhorn Breeders Association of America.
- *Query*: $Y\ Professor$
  Description: Find the KDD 2007 Conference program information page, Dr $Y\ Professor$ had a paper published in that conference with Dr $Z\ Professor$.
- *Query*: global warming
  Description: Find the NRDC global warming site which discusses causes and effects of global climate change.

Each user was given 15 queries, 5 for each query type. Each query was executed 4 times for the 4 presentations being compared. Thus, in total each user executed $15 \times 4 = 60$ queries. For each execution, the user exploration effort was computed.

User effort was the metric we used to measure the search result exploration effort exerted by a user in fulfilling her information need. [18] used a similar metric under a probabilistic model instead of user study. Assuming both search results and cluster labels are scanned and examined in a top-down manner, user effort Ω can be computed as follows:

- Add 1 point to Ω for each examined search result.
- Add 0.25 point to Ω for each examined cluster label. This is because labels are much shorter than snippets.
- Add 0.25 point to Ω for each *uncertain result*. Based on our assumption, all results before a tagged relevant result are examined. However, results after the last tagged result remain uncertain. For linked list presentation, there is no uncertainty because the exploration ends at a tagged result due to the way the queries are chosen (more relevant results than needed).
  Uncertainty could occur for results within a chosen cluster C. As an effective way of utilizing cluster labels, most users would partially examine a few results in C to evaluate the relevance of C itself. If they think C is relevant, they must have found and tagged some relevant results in C. If they think C is irrelevant, they would ignore the cluster and quickly move to the next label. Thus, each uncertain result has a probability of being examined. Based on our observation for this particular user study, we empirically used 0.25 for this probability.

B. System Evaluation Results

`ClusteringWiki` operation is independent of parameters such as number of results or chosen clustering algorithm. We chose the following defaults when executing correctness evaluation:

- Results: 200
- Algorithm: hierarchical k-means
- Similarity calculator: Jaccard
- Term similarity threshold: 0.5
- Result similarity threshold: 0.05

A system test was also executed which verified the application functionality with other chosen values for the above parameters.

Functional tests. `ClusteringWiki` is a multi-tiered system with interactive components written in multiple programming languages. As such, standard unit tests are not as helpful in determining the proper functionality of our system. We used manually executed function tests to verify that `ClusteringWiki`works as indented. All tests described below were executed successfully. In general, a *label node* refers to a *non-root non-bottom label node* in the following section, unless otherwise noted.

1) Cluster reproducibility: Test that `ClusteringWiki` builds the same cluster tree after subsequent executions with the same query input and parameters.

Steps:

1. Execute a `ClusteringWiki` search using any query text that will produce a non-empty search result set. Note the set of paths within the created cluster.
   Expected result: `ClusteringWiki` creates a cluster of search results. Cluster paths noted.
2. Repeat the search from the previous step and check the set of paths recorded previously against the set of paths of the newly created cluster.
   Expected result: The two sets of paths are identical.
3. Copy or move one or more result nodes and one or more label nodes. Note the set of paths within the created cluster.
   Expected result: Cluster paths noted.
4. Repeat the search from the previous step and check the set of paths recorded previously against the set of paths of the newly created cluster.
   Expected result: The two sets of paths are identical.

2) Context menu operation pre-validation: Test that context menus display appropriate menu items for different contexts.

Steps:

1. While logged out of `ClusteringWiki`, execute a search and right-click on the root node.
   Expected result: Menu items: *Please log in to enable editing*, *Expand all*, *Collapse all.*
2. Right-click on each of the following: a non-root label node, a bottom label node, and a result node.
   Expected result: Menu items: *Please log in to enable editing.*
3. Log into the application and execute a search. Right-click on the root node.
   Expected result: Menu items: *Create internal label*, *Create bottom label*, *Copy label*, *Expand all*, *Collapse all*, *Clear all edits*, *Show tree w/o edits.*
4. Right-click on a non-root label node.
   Expected result: Menu items: *Create internal label*, *Create bottom label*, *Copy label*, *Cut label*, *Rename label.*
5. Right-click on a bottom label node.
   Expected result: Menu items: *Copy label*, *Cut label*, *Rename label.*
6. Right-click on a result node.
   Expected result: Menu items: *Copy result*, *Cut result.*
7. Copy or cut a result node. Right-click on a bottom label node.
   Expected result: An additional *Paste result* menu item is displayed.
8. Right-click on a label node.
   Expected result: No additional *Paste result* menu item is displayed.
9. Copy a result node into another bottom label node, then select the bottom label node you pasted the result node into. Right-click on the result node that you copied.
   Expected result: An additional *Delete result* menu item is displayed.
10. Select a label node that is a parent of both bottom label nodes containing the result node copied in the previous step (ex: *All*). Right-click on the result node that you copied.
    Expected result: No additional *Delete result* menu item is displayed.
11. Copy a label node that has at least one child label node and right-click on a label node outside the path of the copied node.
    Expected result: An additional *Paste label* menu item is displayed.
12. Right-click a label node that is a child of the copied label node.
    Expected result: An additional *Paste label* menu item is displayed.
13. Right-click on a bottom label node.
    Expected result: No additional *Paste label* menu item is displayed.
14. Right-click on the root node and select *Show tree w/o edits*. In the re-built tree, right-click on the root node.
    Expected result: Menu items: *Re-enable editing*, *Expand all*, *Collapse all.*

15. Right-click on each of the following: a non-root label node, a bottom label node, and a result node.
    Expected result: Menu items: *Re-enable editing*

3) Result node operations: Test that a result node can be copied, moved, and deleted.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Select a bottom label node. Copy any given result node to another bottom label node. Select the new bottom label node, then the original bottom label node the result was copied from. Note the paths of each of the bottom label nodes containing the copied result node.
   Expected result: The result node is listed in the list of results for both of the bottom label nodes noted in the step.
2. Re-execute the search and select each of the bottom label nodes noted in the previous step.
   Expected result: The result node is listed in the list of of results for each of the bottom label nodes noted in the step.
3. Copy the result node previously copied again and paste it in one of the bottom label nodes that already contain the result node.
   Expected result: Error: Result node already exists in destination node.
4. With one of the bottom label nodes from the previous step selected, cut the result node previously copied and paste it into a new bottom label node. Note the paths of the two bottom label nodes containing the result node as well as the path of the bottom label node the result node was cut from.
   Expected result: Result node moved successfully.
5. Re-execute the search and select each of the bottom label nodes noted in the previous step.
   Expected result: The result node is not listed in the bottom label node the result node was cut from and is listed in the other two bottom label nodes.
6. Select a label node that is a parent of both bottom label nodes containing the result node from the previous step (ex: *All*). Cut the result node and paste it in a new bottom label node. Note the path of the node pasting into.
   Expected result: Result node moved successfully.
7. Re-execute the search and select each of the bottom nodes noted in the previous three steps.
   Expected result: Result node is only listed in the bottom label node the result node was moved to in the last step.
8. Copy any result node to an alternate bottom label node. Note the paths of the two bottom nodes containing the result node. Select one of the bottom label nodes and delete the copied result node.
   Expected result: Result node removed successfully.
9. Re-select each of the bottom label nodes noted in the previous step. Then re-execute the search and re-select each of the bottom label nodes noted in the previous step.
   Expected result: The result node is listed only in the bottom label node the result was not deleted from.
10. Copy a bottom label node (preferably containing few result nodes) to another label node. Select either of the copies of the bottom label node and delete all the result nodes it contains. Note the path of the selected bottom label node.
    Expected result: Result nodes removed successfully. A warning message is displayed in the results panel noting that the cluster label will not be included in the cluster when the search is re-executed.
11. Re-execute the search from the previous step. Check whether the bottom label node whose results were removed is listed in the cluster tree.
    Expected result: The bottom label node whose results were removed is not listed in the cluster tree.
12. Drag any result and drop it away from the cluster tree. Then drag the same result and drop it on the cluster tree root node. Then drag a result node and drop it on an internal label node.
    Expected result: In all three cases the result node returns to its original location in the page.
13. Drag any result node to a bottom label node that does not already contain the result.
    Expected result: The result node was moved successfully.

4) Bottom label node operations: Test that a bottom label node can be copied, moved, modified, added, and deleted.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Right-click on a bottom label node and copy it. Note the path of the copied node and the set of result nodes it includes. Right-click on any label node and paste the copied node. Note the path of the newly created copy of the bottom label node.
   Expected result: Label node copied successfully.
2. Select the copied node and check that the results it contains are the same as in the bottom label node that was copied.
   Expected result: The copy bottom label node contains the same result nodes as its original.
3. Try to paste the copied bottom label node in the same label node in which you previously pasted the bottom label node.
   Expected result: Error: Node to be copied already exists in destination node.

4. Re-execute the search and check the node paths noted in the first step. Ensure that both bottom label nodes exist and contain the same set of result nodes as before the copy.
   Expected result: Both copies of the bottom label node exist and contain the same set of result nodes as before the copy.
5. Modify the name of the copy of the bottom label node to an empty string.
   Expected result: Error: The node label cannot be empty.
6. Modify the name of the copy of the bottom label node to an alternate valid name. Make note of the new path of the bottom label node that was renamed.
   Expected result: Label renamed successfully.
7. Re-execute the search and check the path of the bottom label node copied previously as well as the renamed bottom label node.
   Expected result: Both bottom label nodes exist and contain the same set of result nodes.
8. Copy the original bottom label node copied in the first step and paste it in the same node pasted in before. Check the result nodes contained in all three copies of the bottom label nodes.
   Expected result: Label node copied successfully. All three copies of the bottom label node contain the same set of result nodes.
9. Cut one of the copies of the bottom label node and paste it into an alternate label node. Select the moved node and make note of its path.
   Expected result: Label node moved successfully. The moved node contains the same set of result nodes as before being moved.
10. Re-execute the search and re-select the moved node noted in the previous step.
    Expected result: The moved node exists and contains the same set of result nodes as before being moved.
11. Right-click on the moved bottom label node and delete it.
    Expected result: Node deleted successfully.
12. Re-execute the search and check for the path of the moved node that was deleted in the previous step.
    Expected result: The node does not exist.
13. Select a label node that contains few bottom label nodes and copy it into another label node. Then delete all bottom label nodes it contains.
    Expected result: All bottom label nodes are deleted. After the last bottom label node is deleted, the parent label node is transformed into a bottom label node.
14. Right-click on any label node and create a bottom label node. Give it any name you wish. Note the path of the newly created node. Copy at least one result node into the new bottom label node. Select the new bottom label node.
    Expected result: The bottom label node is created successfully and contains the copied result nodes.
15. Re-execute the search and select the bottom label node created in the previous step.
    Expected result: The bottom label node created in the previous step exists and contains the same result nodes that were copied into it in the previous step.
16. Drag any bottom label node and drop it on a result node. Then drag a bottom label node and drop it away from the cluster tree. Then drag a bottom label node and drop it on another bottom label node.
    Expected result: In all cases the bottom label node returns to its original location in the cluster tree.
17. Drag a bottom label node and drop it on a label node.
    Expected result: The bottom label node is moved successfully.

5) Non-bottom label node operations: Test that a non-bottom label node can be copied, moved, modified, added, and deleted.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Right-click on an internal label node and copy it. Paste it into an alternate label node. Make note of the paths of the copied label node and the set of its children. Expand the label node copy.
   Expected result: The label node was copied successfully and the copy node contains the same children as the original label node.
2. Re-execute the search and select the label node copy again.
   Expected result: The label node exists and contains the same children as the original label node.
3. Copy the original label node again and try to paste it into the same label node as previously pasted into.
   Expected result: Error: Node to be copied already exists in destination node.
4. Modify the name of the copy label node to a blank string.
   Expected result: Error: The node label cannot be empty.
5. Modify the name of the copy label node to an alternate valid name. Make note of the new path for the renamed node.
   Expected result: Label node renamed successfully.

6. Re-execute the search and select the previously renamed label node.
   Expected result: The renamed label node exists and contains the same child nodes as before being renamed.
7. Copy the original label node and paste it into the same label node as before, then select it.
   Expected result: Label node copied successfully. The new copy label node contains the same children as the original.
8. Delete one of the copies of the label node previously copied. Note its path.
   Expected result: Label node deleted successfully.
9. Re-execute the search and check for the label node that was deleted.
   Expected result: The deleted label node and all its children no longer exist.
10. Right-click on any label node and create an internal label. Copy at least one bottom label node containing one or more result nodes into the newly created internal label node. Make note of the path for the newly created node and its set of child nodes.
    Expected result: Label node created successfully.
11. Re-execute the search. Select the newly created label node.
    Expected result: The new label node exists and contains the same set of children that was copied in the previous step.
12. Drag any label node and drop it on a result node. Then drag a label node and drop it away from the cluster tree. Then drag a label node and drop it on a bottom label node.
    Expected result: In all cases the label node returns to its original location in the cluster tree.
13. Drag a label node and drop it on an alternate label node.
    Expected result: The label node is moved successfully.
14. Select a label node with at least one child internal label node. Ensure the child label node is collapsed and then collapse the selected label node. Double-click on the selected label node.
    Expected result: The node double-clicked is expanded and all its child label nodes are also expanded.
15. Right-click on the root node and select *Expand all*.
    Expected result: All cluster tree internal label nodes are expanded.
16. Right-click on the root node and select *Collapse all*.
    Expected result: All cluster tree internal label nodes other than the root node are collapsed.

6) Preference transfer: Test that preferences are transferable between similar queries.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Copy and paste any label or result node and make note of the paths that were added. Re-execute the query and look for the paths that were added.
   Expected result: The added paths are displayed in the cluster tree.
2. Log out of `ClusteringWiki` and re-execute the query. Look for the paths that were previously added.
   Expected result: The query executed while logged out of the system creates a cluster tree that contains the added paths.
3. Log into `ClusteringWiki`using an alternate account than the one used in the first step. Re-execute the same query as before. Look for the paths added in the first step.
   Expected result: The paths added in the first step are not displayed in the tree.
4. Copy and paste any label or result node other than the ones copied in the first step and make note of the paths that were added. Re-execute the query and look for the paths that were added.
   Expected result: The added paths are displayed in the cluster tree.
5. Log out of `ClusteringWiki` and re-execute the query. Look for the paths that were added in the previous step and those that were added in the first step.
   Expected result: Both the paths that were added in the previous step and those that were added in the first step are displayed in the tree.
6. Execute a very similar query to the previous query (for example add the word ”a” in front of the previous query text). Look for the paths that were added in the previous steps.
   Expected result: The cluster tree contains the paths that were added previously associated with the similar query.

7) Convenience features: Test that users can see the cluster tree without applied preferences as well as clear all their preferences associated with a given query.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Copy and paste at least one label or result node and make note of the set of paths added. Right-click on the root node and select *Show tree w/o edits*. Look for the paths that were previously added.
   Expected result: The cluster tree is rebuilt and the added paths are not displayed.
2. Right-click on any node and select *Re-enable editing*. Look for the paths added in the previous step.
   Expected result: The tree is rebuilt and the added paths are once again displayed in the cluster tree.

3. Re-execute the search and look for the paths added in the first step.
   Expected result: The added paths are displayed in the cluster tree.
4. Right-click on the root node and select *Clear tree edits*. Look for the paths added in the first step.
   Expected result: The cluster tree is rebuilt and the added paths are not displayed.
5. Re-execute the search and look for the paths added in the first step.
   Expected result: The added paths are no longer displayed in the cluster tree.

System test. The test steps below were executed using application settings other than the defaults used in the previous tests to verify that application settings do not affect application functionality.

Steps:

1. Log in to a new `ClusteringWiki` session and execute any query that will produce results. Copy and paste to alternate appropriate locations one of each of the following: a result node, a bottom label node, an internal label node. Make note of the added paths.
   Expected result: Nodes copied successfully.
2. Re-execute the query and check for the paths added in the previous step.
   Expected result: The added paths are still displayed in the cluster tree.
3. Move each of the nodes copied in the first step to alternate locations. Make note of the new paths.
   Expected result: Nodes moved successfully.
4. Re-execute the query and check for the moved nodes.
   Expected result: The moved nodes are displayed in the cluster tree.
5. Log out of `ClusteringWiki` and re-execute the query. Check for the nodes previously moved.
   Expected result: The moved nodes are still displayed in the cluster tree.
6. Log back into the application and re-execute the query. Delete all the nodes previously moved.
   Expected result: Nodes deleted successfully.
7. Re-execute the query and look for the previously moved nodes.
   Expected result: The moved nodes are no longer displayed in the cluster tree.

System Efficiency

We measured 6 sections of the `ClusteringWiki` total response time, as follows:

- *Retrieving results*. The retrieving results section of the total response time includes processing query parameters, passing the search request to *AbstractSearch*, retrieving its JSON response, and processing the JSON data into a collection of Java search response document objects used in the remainder of the algorithm execution.
- *Preprocessing*. The document preprocessing section of the total response time includes analyzing the text of the search response document titles and snippets, creating document bags of words, the collection term index, and various other reverse lookup indexes used by different sections of `ClusteringWiki` execution.
- *Initial clustering*. In the clustering section the initial document cluster tree T_{init} is created using the chosen clustering algorithm.
- *Applying preferences*. This section includes identifying the set of preferences to be applied to T_{init} as well as merging those preferences into T_{init} to create the final cluster tree.
- *Presenting final tree*. Presenting the final tree includes the browser side time needed to process the data received from the server into a new *EditableClusterTree* object, embedding the HTML representation of the cluster tree and result nodes into the browser page, and attaching necessary JavaScript events to enable tree functionality.
- *Other*. The remaining time, which is out of `ClusteringWiki` control, includes transferring requests and data between the browser and server and some negligible time for program control.

Table I shows the averaged (over 10 queries) runtime in seconds for all 6 portions of total response time. In addition, we computed and list the average *total execution time*, which includes preprocessing, initial clustering, applying preferences and presenting the final tree. This is the time that our prototype is responsible for. The remaining time is irrelevant to the way our prototype is designed and implemented. From the table we can see that:

TABLE I

EFFICIENCY EVALUATION

Data source	**Yahoo!**										**New York Times**									
Number of results	**100**		**200**		**300**		**400**		**500**		**100**		**200**		**300**		**400**		**500**	
Type of clustering	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**
Retrieving results	0.979	1.018	1.309	1.222	1.615	1.391	1.847	1.579	1.679	1.661	0.102	0.113	0.131	0.130	0.285	0.259	0.326	0.338	0.419	0.387
Preprocessing	0.009	0.011	0.052	0.052	0.037	0.037	0.049	0.112	0.152	0.150	0.019	0.019	0.067	0.066	0.068	0.069	0.096	0.168	0.189	0.187
Initial clustering	0.004	0.005	0.051	0.040	0.033	0.042	0.104	0.063	0.118	0.144	0.005	0.006	0.022	0.025	0.035	0.041	0.053	0.062	0.147	0.196
Applying preferences	0.006	0.007	0.049	0.012	0.015	0.011	0.021	0.015	0.08	0.013	0.012	0.011	0.013	0.024	0.018	0.011	0.013	0.017	0.016	0.014
Presenting final tree	0.143	0.172	0.249	0.278	0.341	0.421	0.451	0.66	0.723	0.752	0.282	0.279	0.372	0.449	0.591	0.691	0.751	0.872	0.846	0.941
Other	0.396	0.416	0.469	0.524	0.624	0.684	0.558	0.593	0.853	0.912	0.338	0.497	0.478	0.678	0.589	0.672	0.625	0.937	0.736	0.868
Total execution time	0.160	0.194	0.401	0.381	0.426	0.511	0.624	0.850	1.073	1.059	0.317	0.315	0.473	0.565	0.713	0.813	0.913	1.118	1.197	1.338
Total response time	1.535	1.628	2.179	2.127	2.665	2.585	3.029	3.022	3.604	3.632	0.758	0.925	1.083	1.373	1.587	1.744	1.863	2.393	2.352	2.593

- The majority of the total response time is taken up by retrieving search results, which would be negligible if we were a search company.
- Applying preferences takes less than $1/10$ second in all test cases, which certifies the efficiency of our "path approach" for managing preferences.
- Presenting the final tree takes the majority (roughly 80%) of the total execution time, which can be improved by using alternate user interface technologies.

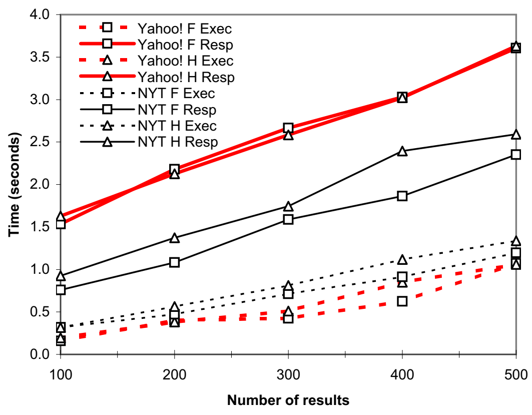


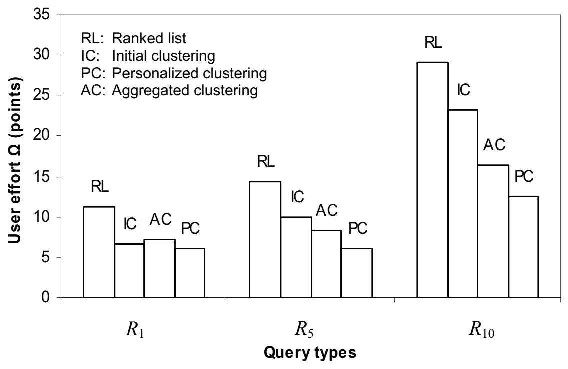
Fig. 4. Efficiency evaluation.

Figure 4 shows the trends of the average total execution time (Exec in the figure) and response time (Resp) for both flat (F) and hierarchical (H) presentations over 2 sources of Yahoo! (Yahoo!) and New York Times (NYT). From the figure we can see that:

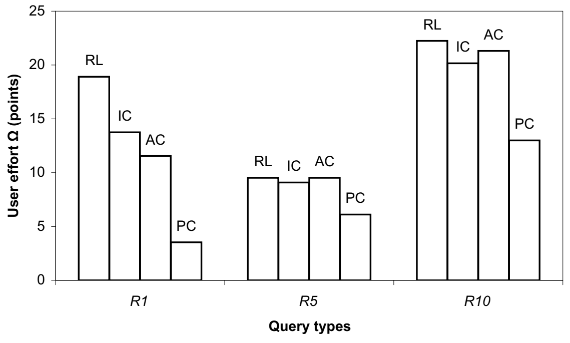
- Response and execution time trends are linear, testifying to the scalability of our prototype. In particular, for both flat and hierarchical clustering, the total execution time is about 1 second for 500 results and 0.4 second for 200 results from either source. Note that most existing clustering search engines, e.g., iBoogie (www.iboogie.com) and CarrotSearch (carrotsearch.com), cluster 100 results by default and 200 at maximum. Clusty (www.clusty.com) clusters 200 results by default and 500 at maximum.
- Hierarchical presentation (H) takes comparable times to flat presentation (F), showing that recursive generation of hierarchies does not add significant cost to efficiency.
- There is a bigger discrepancy between response and execution times for the Yahoo! data source compared to New York Times, suggesting a significant efficiency improvement by integrating our prototype with the data sources.
- Execution times for Yahoo! are shorter than New York Times due to the shorter titles and snippets.

C. Utility Evaluation Results

System utility.



(a) Google data source



(b) New York Times data source

Fig. 5. Utility evaluation on Google and New York Times data sources

Figure 5 shows the averaged user effort (over $22 \times 5 = 110$ queries) for each of the 4 presentations (RL, IL, PC, AC) and each of the query types (R_1, R_5, R_{10}) on the Google and New York Times data sources. From the figure we can see that:

- Clustering saves user effort in informational and deep navigational queries, with personalized clustering the most effective, saving up to 50% of user effort.
- Aggregated clustering also significantly benefits, although it is not as effective as personalized clustering. However, it is "free" in the sense that it does not take user editing effort, and it does not require user login.

  In evaluating aggregated clustering, we made sure that the users using the aggregated clusters were not the ones who edited them.
- The effectiveness of clustering is related to how "deep" the relevant results are. The lower they are ranked, the more effective clustering is because more irrelevant results can be skipped.

The hierarchy of cluster labels plays a central role in the effectiveness of clustering search engines. From the data we have collected as well as the user feedback, we observe that:

- Cluster labels should be short and in the range of 1 to 4 terms, with 2 and 3 the best. The total levels of the hierarchy should be limited to 3 or 4.
- There are two types of cluster edits, (1) assigning search results to labels and (2) editing the hierarchy of labels. Both types are effective for personalized clustering. However, they respond differently for aggregated clustering. For type 1 edits, there is a ground truth (in a loose sense) for each assignment that users tend to agree on. Such edits are easy to aggregate and be collaboratively utilized. For type 2 edits, it can be challenging (and a legitimate research topic) to aggregate hierarchies because many edited hierarchies can be good but in diverse ways. A good initial clustering (e.g., frequent phrase hierarchical) can alleviate the problem by reducing the diversity.

System usability.

At the end of the utility study we asked the study participants to complete a survey about `ClusteringWiki`. Users answered the following four questions, assigning a rating between 1 and 10, where 1 meant *strongly disagree* and 10 meant *strongly agree*:

1) In your opinion, is clustering of search results a helpful technique for finding relevant web search results easier?
2) Does the ability to edit the search result cluster increase your chances to find relevant results in the future?
3) If thousands of people were contributing on editing search result clusters on many topics, would you likely take advantage of the mass collaboration by using a system like `ClusteringWiki`?
4) Was the `ClusteringWiki` interface easy to use?

User ratings ranged between 3 and 10 and mostly showed the users were satisfied with the application and willing to use `ClusteringWiki` or a similar system in the future. The average user ratings for the four questions are listed below:

Q1	8.23
Q2	8.18
Q3	8.50
Q4	7.91

Additionally, users were asked to provide optional comments for improving `ClusteringWiki`. The comments we received show that, while most users found the application interface easy to use, they desired additional convenience features. The user study participant comments are listed below:

- Being able to move several documents at once into a label; being able to search the contents of each label.
- The divisions of the "cluster" sometimes were misleading or cryptic - If so many people can edit the search result clusters, can't that be just as cumbersome? Won't we then have to sort through results files as well?
- Providing keywords that can be tagged to a web page for easier categorization.
- Web 2.0. Pretty straight forward. Mainly editing was finicky at times and could be polished a little more.
- A temp folder based on additional keywords.
- The concept idea is great to me and as far as I know completely user friendly to the max but if being made an actual search engine then I would suggest making the interface more graphic oriented and dynamic. I like being able to move around the entire cluster result kind of like a minimized window on a maximized window.
- Interface could benefit from some more polishing. Maybe not allow more than one instance of a web site in the cluster on the left.
- Trolling will be a problem if an open community is allowed edit privileges.
- I would just suggest the "quick links" on the left read more relevantly to the actual topic - to read more easily. Otherwise it's a great use. :-)
- The editing needs to streamline. Folders with the same name need to aggregate. Icons need to be clickable. Make editing drag and drop only. Basically needs to be fast and idiot proof. Increase size of icons/cluster text.
- Online examples of how to edit (with a graphic display) would help. Too much open to interpretation (how to align to the folder to move a search item, etc). Also, if a long list would occur, the drag and drop feature did not work.
- I think making the drag and drop be copy instead of cut would be more helpful. Or maybe provide keyboard shortcuts for cutting, copying, pasting. Overall takes a while to get the hang of it, but after about ten minutes everything makes sense.
- Maybe only allow results to show up under 1 folder.
- I encountered an error which prevented me from creating a bottom label in a top label. I believe it mentioned something about the tree being too long. A few times I tried dragging a top label into a top label that contained a bottom label by the same name, and I could not perform the action. I think merging of the labels' contents might be useful.
- You might consider having the cluster in a scrollable macro, so it follows the main page's scrolling when going much further down the page into results.
- After create[ing a] new folder and nam[ing] it, if we move the cursor somewhere else, it would be better if the folder [could] be created without the need [to press] "Enter".

VII. Conclusion

Search engine utility has been significantly hampered due to the ever-increasing information overload. Clustering has been considered a promising alternative to ranked lists in improving search result organization. Given the unique human factor in search result clustering, traditional automatic algorithms often fail to generate clusters and labels that are interesting and meaningful from the user's perspective. In this paper, we introduced `ClusteringWiki`, the first prototype and framework for personalized clustering, utilizing the power of direct user intervention and mass-collaboration. Through a Wiki interface, the user can edit the membership, structure and labels of clusters. Such edits can be aggregated and shared among users to improve search result organization and search engine utility.

There are many interesting directions for future work, from fundamental semantics and functionalities of the framework to convenience features, user interface and scalability. For example, in line with social browsing, social network can be utilized in preference aggregation. Another interesting direction is to seamlessly integrate personalization of search result ranking [8] with that of search result clustering, providing a more complete solution for personalized and collaborative information retrieval and Web search.

REFERENCES

[1] S. Basu, M. Bilenko, and R. J. Mooney. A probabilistic framework for semi-supervised clustering. In *KDD*, 2004.

[2] R. Bekkerman, H. Raghavan, J. Allan, and K. Eguchi. Interactive clustering of text collections according to a user-specified criterion. In *IJCAI*, 2007.

[3] A. Broder. A taxonomy of web search. In *SIGIR Forum*, 2002.

[4] C. Carpineto, S. Osiński, G. Romano, and D. Weiss. A survey of web clustering engines. *ACM Comput. Surv.*, 41(3):1–38, 2009.

[5] D. Cohn, A. K. McCallum, and T. Hertz. *Constrained Clustering: Advances in Algorithms, Theory, and Applications*, chapter Semi-Supervised Clustering with User Feedback. Chapman and Hall/CRC, 2009.

[6] A. Doan, R. Ramakrishnan, and A. Halevy. Mass collaboration systems on the world-wide web. *Communications of the ACM*, to appear.

[7] B. S. Everitt, S. Landau, and M. Leese. *Cluster analysis*. Oxford University Press, 2001.

[8] B. J. Gao and J. Jan. Rants: a framework for rank editing and sharing in web search. In *WWW*, 2010.

[9] A. Griffiths, H. C. Luckhurst, and P. Willett. Using interdocument similarity information in document retrieval systems. *Journal of the American Society for Information Sciences*, 37(1):3–11, 1986.

[10] R. V. S. H. Halpin H. The complex dynamics of collaborative tagging. *WWW*, 2007.

[11] M. A. Hearst and J. O. Pedersen. Reexamining the cluster hypothesis: scatter/gather on retrieval results. In *SIGIR*, 1996.

[12] A. Iskold. Overview of clustering and clusty search engine. *www.readwriteweb.com/archives/overview_of_clu.php*, 2007.

[13] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, 1988.

[14] X. Ji and W. Xu. Document clustering with prior knowledge. In *SIGIR*, 2006.

[15] M. Käki. Findex: search result categories help users when document ranking fails. In *CHI*, 2005.

[16] A. Kale, T. Burris, B. Shah, T. L. P. Venkatesan, L. Velusamy, M. Gupta, and M. Degerattu. icollaborate: harvesting value from enterprise web usage. In *SIGIR*, 2010.

[17] L. Kaufman and P. Rousseeuw. *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 1990.

[18] J. Koren, Y. Zhang, and X. Liu. Personalized interactive faceted search. In *WWW*, 2008.

[19] K. Kummamuru, R. Lotlikar, S. Roy, K. Singal, and R. Krishnapuram. A hierarchical monothetic document clustering algorithm for summarization and browsing search results. In *WWW*, 2004.

[20] J. Lee, S.-w. Hwang, Z. Nie, and J.-R. Wen. Query result clustering for object-level search. In *KDD*, 2009.

[21] J. MacQueen. Some methods for classification and analysis of multivariate observations. In *5th Berkeley Symposium on mathematics, Statistics and Probability*, pages 281–297, 1967.

[22] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[23] S. Osinski and D. Weiss. A concept-driven algorithm for clustering search results. *IEEE Intelligent Systems*, 20(3):48–54, 2005.

[24] P. Pirolli, P. Schank, M. Hearst, and C. Diehl. Scatter/gather browsing communicates the topic structure of a very large text collection. In *CHI*, 1996.

[25] C. J. V. Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 1979.

[26] D. E. Rose and D. Levinson. Understanding user goals in web search. In *WWW*, 2004.

[27] G. Salton. *The SMART Retrieval System*. Prentice-Hall, 1971.

[28] E. Sandhaus. The New York Times Annotated Corpus. *Linguistic Data Consortium, Philadelphia*, 2008.

[29] A. Tombros, R. Villa, and C. J. Van Rijsbergen. The effectiveness of query-specific hierarchic clustering in information retrieval. *Inf. Process. Manage.*, 38(4):559–582, 2002.

[30] K. Wagstaff, C. Cardie, S. Rogers, and S. Schrödl. Constrained k-means clustering with background knowledge. In *ICML*, 2001.

[31] X. Wang and C. Zhai. Learn from web search logs to organize search results. In *SIGIR*, 2007.

[32] S. Xu, S. Bao, B. Fei, Z. Su, and Y. Yu. Exploring folksonomy for personalized search. *SIGIR*, pages 155–162, 2008.

[33] O. Zamir and O. Etzioni. Web document clustering: a feasibility demonstration. In *SIGIR*, 1998.

[34] O. Zamir and O. Etzioni. Grouper: a dynamic clustering interface to web search results. In *WWW*, 1999.

[35] H.-J. Zeng, Q.-C. He, Z. Chen, W.-Y. Ma, and J. Ma. Learning to cluster web search results. In *SIGIR*, 2004.