# A Framework for Personalized and Collaborative Clustering of Search Results

David C. Anastasiu
Texas State University-San Marcos
San Marcos, TX, USA
da1143@txstate.edu

Byron J. Gao
Texas State University-San Marcos
San Marcos, TX, USA
bgao@txstate.edu

David Buttler
Lawrence Livermore National Laboratory
Livermore, CA, USA
buttler1@llnl.gov

## 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.

**Categories and Subject Descriptors:** H.3.3 [Information Storage and Retrieval]: Information Search and Retrieval – *Clustering*

**General Terms:** Algorithms, Design, Human Factors

**Keywords:** Personalized clustering, Search result clustering, Search result organization, Document clustering, Wiki, Mass collaboration, Social tagging, Information retrieval

## 1. 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 [12, 25, 34, 5]. 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 [23].

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 [12, 25, 34, 35, 36, 20, 32, 21]. 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 [13]. 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 [32] and [5], 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 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 [5].

In this paper, we explore a completely different direc-


*CIKM'11,* October 24–28, 2011, Glasgow, Scotland, UK.

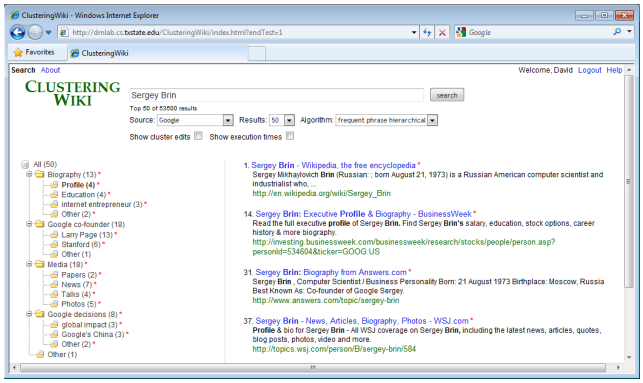


**Figure 1: Snapshot of `ClusteringWiki`.**

tion 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.

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 2 reviews the related work. Section 3 overviews the `ClusteringWiki` framework. Section 4 introduces the framework in detail. Section 5 presents experiments and user study. Section 6 concludes the paper.

## 2. 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 [22], 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 [18], or divisive (top-down) such as DIANA [18].

**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 [14, 8]. 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 [26].

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 [4, 27]. In addition, the growing scale of

[1] dmlab.cs.txstate.edu/ClusteringWiki/index.html.

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 [12, 35, 30, 16].

**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 [10, 28, 5]. It has been shown that query-specific, on-line, post-retrieval clustering, i.e., clustering search results, produces much superior results [12].

Scatter/Gather [12, 25] 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 [34, 35] (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 [24], a clustering algorithm based on singular value decomposition.

Other related work from the Web, IR and data mining communities exists. [36] explored supervised learning for extracting meaningful phrases from snippets, which are then used to group search results. [20] proposed a monothetic algorithm, where a single feature is used to assign documents to clusters and generate cluster labels. [32] investigated using past query history in order to better organize search results for future queries. [21] studied search result clustering for object-level search engines that automatically extract and integrate information on Web objects. [5] 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 [2, 6] and interactive clustering [31, 15, 3] 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.blogspot.com/2008/11/searchwiki-make-search-your-own.html). Rants [9] 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.

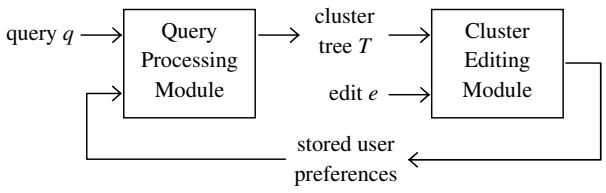


**Figure 2: Main architecture of `ClusteringWiki`.**

**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 [11]. Collaborative tagging systems leverage this property to derive folksonomies and improve search [33]. 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 [7].

# 3. 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.

**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 $T_{init}$.

# 4. 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.

## 4.1 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 4.2. 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 5 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

**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 ← ∅; //P is the set of paths to be enforced on T_init
4: if (u is logged-in) then
5:    q' ← Trans(q, u);
6:    if (q' ≠ NULL) then
7:       P ← P_{q',u}; //use applicable personal preferences
8:    end if
9: else
10:   q' ← Trans(q, U);
11:   if (q' ≠ NULL) then
12:      P ← P_{q',U}; //use applicable aggregated preferences
13:   end if
14: end if
15: T ← T_init; //initialize T, the cluster tree to present
16: clean P; //remove p ∈ P if its result node is not in R
17: for each p ∈ P
18:   if (p is positive) then
19:      T ← T ∪ {p}; //add a preferred path
20:   else
21:      T ← 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
```

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.

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.

**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} ∈ P_u ∧ termSim(q, q') is the largest;
5:    if termSim(q, q') ≥ δ_ts then //δ_ts is a threshold
6:       if resultSim(q, q') ≥ δ_rs then //δ_rs is a threshold
7:          P_{q,u} ← P_{q',u}; //copy preferences from q' to q
8:          return q';
9:       end if
10:   end if
11: end if
12: return NULL;
```

**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 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 $\delta_{ts}$ and $\delta_{rs}$. 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 4.2.

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 [17] 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.

## 4.2 Cluster Editing

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

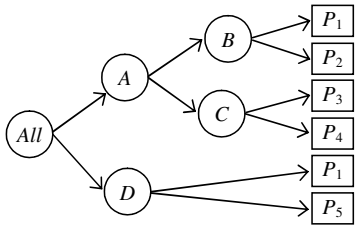


**Figure 3: Example cluster tree.**

**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.

**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, $P_e = \{All \rightarrow D \rightarrow P_3\}$. $NP_e = \emptyset$ 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, $P_e = \{All \rightarrow E \rightarrow P_1, All \rightarrow E \rightarrow P_5\}$ and $NP_e = \{All \rightarrow D \rightarrow P_1, All \rightarrow D \rightarrow P_5\}$.

- $e_4$: 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, $NP_e = \{All \rightarrow D \rightarrow P_5\}$. $P_e = \emptyset$ 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, $P_e = \{All \rightarrow E \rightarrow D \rightarrow P_1, All \rightarrow E \rightarrow D \rightarrow P_5\}$ and $NP_e = \{All \rightarrow D \rightarrow P_1, All \rightarrow D \rightarrow P_5\}$.

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.

# 5. 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.

## 5.1 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 [29] 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.

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. The participants were primarily undergraduate, with a few graduate, colleage 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 [4, 27] 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 [23]. 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. These queries and their descriptions and narratives can be found at [1].

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. [19] 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 $\Omega$ can be computed as follows:

- Add 1 point to $\Omega$ for each examined search result.
- Add 0.25 point to $\Omega$ for each examined cluster label. This is because labels are much shorter than snippets.
- Add 0.25 point to $\Omega$ 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).

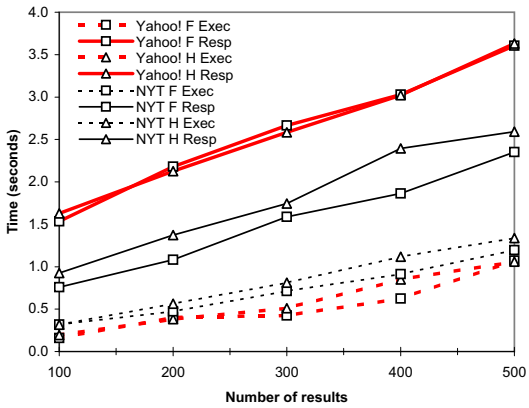


**Figure 4: Efficiency evaluation.**

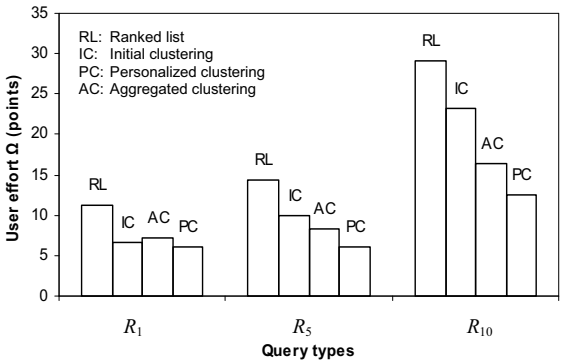


**Figure 5: Utility evaluation on Google data source.**

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.

## 5.2 System Evaluation Results

For correctness, all functional and system tests were executed successfully. A detailed description of these tests can be found at [1]. In the following, we focus on the efficiency evaluation results.

We recorded and averaged (over 10 queries) the runtime in seconds for all 6 portions of total response time. In addition, we also computed 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. While the details are reported in [1], 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.

In addition, we observe (and report in [1] with supporting data) that applying preferences takes less than 1/10 second in all test cases, which certifies the efficiency of our "path approach" for managing preferences. Moreover, presenting the final tree takes the majority (roughly 80%) of the total execution time, which can be improved by using alternate user interface technologies.

## 5.3 Utility Evaluation Results

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 data source. Similar trends can be observed from the AP Newswire data source (see [1] for details). 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.

As part of the user study, we also surveyed on the effectiveness of general, personalized and aggregated clustering in helping with search result exploration. On a scale of 1 to 10 with 10 as the best, users responded with an average rating of 8.21. Most users found `ClusteringWiki` efficient and useful in reducing their search effort.

# 6. 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.

# 7. ACKNOWLEDGMENTS

This work (LLNL-CONF-461652) was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. It was partially supported by the Texas Norman Hackerman Advanced Research Program under Grant No.003656-0035-2009.

# 8. REFERENCES

[1] D. C. Anastasiu, D. Buttler, and B. J. Gao. Clusteringwiki technical report. *dmlab.cs.txstate.edu/ClusteringWiki/pdf/cw.pdf*.

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

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

[4] A. Broder. A taxonomy of web search. *SIGIR Forum*, 36(2):3–10, 2002.

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

[6] 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.

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

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

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

[10] 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.

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

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

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

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

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

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

[17] 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.

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

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

[20] 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.

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

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

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

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

[25] 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.

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

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

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

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

[30] 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.

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

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

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

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

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

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