PERSONALIZED AND COLLABORATIVE CLUSTERING OF SEARCH RESULTS

THESIS

Presented to the Graduate Council
of Texas State University-San Marcos
in Partial Fulfillment
of the Requirements

for the Degree

Master of SCIENCE

by

Dragos Anastasiu, B.A.

San Marcos, Texas
August 2011

FAIR USE AND AUTHOR'S PERMISSION STATEMENT

Fair Use

Duplication Permission

ACKNOWLEDGEMENTS

Many thanks go to Dr. Byron J. Gao, without whose guidance and inspiration I would not be involved in research. I would like to thank Dr. Anne H.H. Ngu and Dr. Yijuan Lu for their constant support and encouragement during the thesis process. I would also like to thank Dr. David Buttler from Lawrence Livermore National Laboratory, who gave me invaluable conceptual and implementation insight and feedback. Finally, I wish to thank my family for their patience and understanding during my many late nights at the office the past year or two.

This manuscript was submitted on June 22, 2011.

TABLE OF CONTENTS

LIST OF TABLES

Table **Page**

LIST OF FIGURES

Figure **Page**

ABSTRACT

PERSONALIZED AND COLLABORATIVE CLUSTERING OF SEARCH RESULTS

by

Dragos Anastasiu, B.A.

Texas State University-San Marcos

August 2011

SUPERVISING PROFESSOR: BYRON J. GAO

Organizing and presenting 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, I 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.

CHAPTER I

INTRODUCTION

We live in the information age. Billions of documents on all topics imaginable are connected and accessible on the Web through the simple concept of the hyperlink. Yet the sheer size of the Web makes browsing to locate desired information a daunting task. Web search attempts to alleviate this problem by connecting short phrase queries to relevant documents on the Web, which are generally displayed in a flat ranked list.

Every day millions of people search the Web, unaware of the complexity involved in matching their query with the information they seek. They hope that the exact search results they are looking for will be displayed as soon as they execute their query. 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, including a computer

scientist, a diplomat, a linguist, a poet, a turbine design engineer, a filmmaker, and so on. Suppose we intend to find information about `Jim Gray`, the Turing Award winner, we can issue a query of "Jim Gray" in Yahoo![1]. For this extremely famous name in computer science, only 2 are relevant in the top 10 results.

The way search results are organized and presented has a direct and significant impact on the utility of search engines. While the flat ranked list presentation is acceptable for homogeneous search results, the diversity of search results for most queries has increased to the point that we must consider alternative presentations by providing additional structure to flat lists so as to effectively minimize browsing effort and alleviate information overload [Carpineto et al., 2009; Hearst and Pedersen, 1996; Pirolli et al., 1996; Zamir and Etzioni, 1998]. 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,

[1]Other choices of search engine in this example would not change the validity of the observations. Also note that search results and their ranks may change over time.

faceted search interface, allowing the user to quickly locate relevant and interesting results. Evidence shows that clustering improves user experience and search result quality [Manning et al., 2008].

Given the significant potential benefits, search result clustering has received increasing attention in recent years from the communities of information retrieval (IR), Web search, and data mining. Many clustering algorithms have been proposed [Hearst and Pedersen, 1996; Kummamuru et al., 2004; Lee et al., 2009; Pirolli et al., 1996; Wang and Zhai, 2007; Zamir and Etzioni, 1998, 1999; Zeng et al., 2004]. In the industry, well-known cluster-based commercial search engines include Clusty (www.clusty.com), iBoogie (www.iboogie.com) and CarrotSearch (carrotsearch.com). Carrot2 (www.carrot2.org) is an open source clustering engine distributed under the BSD license.

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 [Iskold, 2007]. 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 [Wang and Zhai, 2007] and [Carpineto et al., 2009], 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 [Carpineto et al., 2009].

In this thesis, I explore a completely different direction in tackling the problem of clustering search results, utilizing the power of direct user intervention and mass-collaboration. I 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 [Anastasiu et al., 2011], the user can edit and annotate the membership, structure and labels of clusters through a Wiki interface to personalize their search result presentation. Personalization provides direct and immediate benefit to the user by reducing user effort spent locating desired results. 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 as other current trends in the Web, like Web 2.0, semantic web,

personalization, social tagging and mass collaboration.

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.

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 1.1 shows a snapshot of ClusteringWiki[2]. 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

[2]dmlab.cs.txstate.edu/ClusteringWiki/.

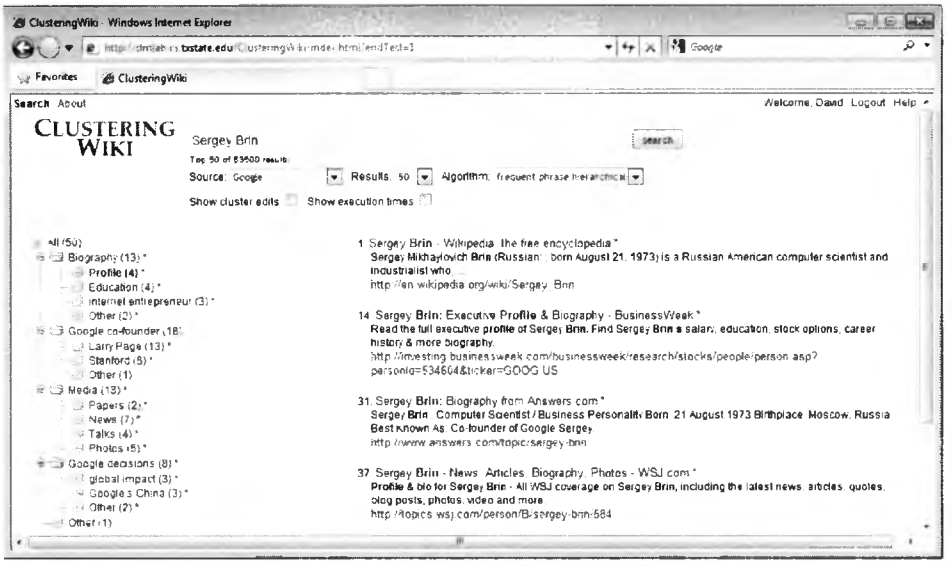


Figure 1.1: Snapshot of ClusteringWiki.

challenges. User edits represent user preferences or constraints that should be respected and enforced when the same query is next 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 clustering presentation.

CHAPTER II

BACKGROUND

The World Wide Web was created in 1990 as a result on Sir Tim Barners-Lee's vision for a decentralized system for information dissemination [Zimmerman, 2000]. Since then it has grown exponentially both in terms of number of users and linked documents. Today there are over 17.47 billion estimated pages[1] on the Web, not including documents hidden behind web forms or ftp servers (hidden web documents). This explosion in both the size and depth of the Web makes "browsing" as the main means of finding Web information obsolete.

The research community has been active over the past several decades, investigating new methods of analyzing, organizing, and presenting Web documents, with the goal of minimizing the time spent between executing the user query and filling the information need. Below I present some of the related research which either influences or enables the work in this thesis.

[1]Retrieved from www.worldwidewebsize.com on Tuesday, 14 June, 2011

2.1 Web Search

Information retrieval (IR) aims to retrieve, from a large collection, those materials (usually documents) that satisfy an information need [Manning et al., 2008]. When applied to the Web, IR focuses on free-text documents and multimedia files, and is better known as Web search.

Vector Space Model. The vector space model (VSM), an algebraic model for text document representation coined by Gerard M Salton [Salton et al., 1975], was first used in the SMART [Salton, 1971] information retrieval system. In the vector space model the query and each indexed document are represented by vectors of term weights,

$$D_i = (w_{i1}, w_{i2}, ..., w_{ij}),$$

where w_{ij} represents the weight of the jth term in document i.

Initially, document term frequency (tf) was recommended as a good term weighting scheme. In 1972, Karen Spärck Jones introduced a weighting scheme based on collection term specificity, the inverse document frequency (idf). [Salton et al., 1975] used a combination of the two schemes, defining the term frequency-inverse document frequency (tf-idf) weighting scheme, in which

$$w_{i,j} = tf_{i,j} \times \log \frac{|D|}{|\{d' \in D \mid j \in d'\}|},$$

where $tf_{i,j}$ is the number of times term j appears in document i, $|D|$ is the number

of documents in the collection, and $|\{d' \in D \mid j \in d'\}|$ represents the number of collection documents containing the term j.

Various functions have been developed for computing the similarity of two documents defined by their document vectors d_1 and d_2, including the *Jaccard index* and the *Tanimoto coefficient.* [Salton et al., 1975] suggested using an inverse function of the angle between the two document vectors, after normalizing all vector lengths to one. The *cosine similarity* he defined, which is also often used in document clustering algorithms [Tan et al., 2005], can be derived from the Euclidean dot product formula,

$$d_1 \cdot d_2 \;=\; ||d_1||\;||d_2||\;\cos(\theta),$$

where θ is the angle between the normalized term vectors of documents d_1 and d_2. Therefore,

$$similarity \;=\; \cos(\theta) \;=\; \frac{d_1 \cdot d_2}{||d_1||\;||d_2||} \;=\; \frac{\sum_{j=1}^{n} d_{1,j} \times d_{2,j}}{\sqrt{\sum_{j=1}^{n}(d_{1,j})^2} \times \sqrt{\sum_{j=1}^{n}(d_{2,j})^2}}.$$

ClusteringWiki relies on external search engines and on Apache Lucene[2] to retrieve and rank appropriate results for user queries. The process followed by Google or Yahoo! is unknown. Lucene, however, indexes documents locally and uses, among others, the *vector space model* and *cosine similarity* to rank a retrieved set of results. In addition to their use in retrieving initial search results,

[2]http://lucene.apache.org/

ClusteringWiki uses the *vector space model* and *cosine similarity* to compute cluster cohesion in two of its implemented *k-means*-based clustering algorithms.

2.2 Improving Search Through Personalization

User queries are short [Jansen et al., 2000] and generally ambiguous [Krovetz and Croft, 1992], causing many of the retrieved results to be irrelevant for a given search intent. [Lawrence, 2000] suggested that searcher and query context could be used to better direct search, producing more relevant results or raking those relevant results higher in the returned list.

Personalized search. Personalized search algorithms use additional searcher information to return a personalized list of results in response to a query. For example, if an entomologist searches for "fly southwest", he would rather find species of diptera found in the south-west, whereas someone else may wish to find the web site for Southwest Airlines[3].

Algorithms that focus on the searcher context generally build a long-term or short-term user profile for the searcher. The user profile is built either explicitly, by asking users to provide preferences [Chirita et al., 2005], or implicitly [Pretschner and Gauch, 1999]. Since most users do not provide explicit preference information [Carroll and Rosson, 1987], most research has been focused on implicit user profile

[3]www.southwest.com

generation.

Some personalized search algorithms build user profiles as ontology-based concept hierarchies [Chaffee and Gauch, 2000; Chirita et al., 2005; Gauch et al., 2003]. Others use lists of terms extracted from previous search contexts [Sugiyama et al., 2004; Tan et al., 2006]. Once built, user profiles can be used to refine the executed query (e.g. using relevance feedback) [Chirita et al., 2007; Joachims et al., 2005; Kelly and Teevan, 2003; Ruthven and Lalmas, 2003; Teevan et al., 2005], to guide the result gathering process via a personalized version of the PageRank algorithm [Haveliwala, 2002; Jeh and Widom, 2003; Qiu and Cho, 2006; Sarlós et al., 2006], or to re-rank non-personalized retrieved results [Speretta and Gauch, 2005]. [Dou et al., 2009] pose that personalized search is only effective for some queries and propose an algorithm for identifying those personalization-prone queries. [Wen et al., 2009] provides an extensive survey of personalized Web search.

Similar to personalized search algorithms that re-rank results, ClusteringWiki personalizes the view to the results list. However, the personalization is applied to the clustering of search results, not altering the order or make-up of the result set. Through selecting a personalized cluster label, the searcher is able to quickly review only those results of interest to them.

In general, personalization in ClusteringWiki is explicit. The user edits the labels and membership of the clustering of search results formed in response to a

query. The user "profile" is made up of cluster edits performed by the user in response to one or more queries. When a similar enough query whose search result cluster has been personalized can be found, the present query is implicitly personalized by applying the similar query's edits (*preference transfer*).

Collaborative search. Personalized Web search algorithms cannot always be implemented efficiently for large number of users, due to the space requirements for off-line storage of user profiles or Personalized PageRank Vectors (PPV). Some algorithms, which fall under the umbrella of (implicit) Collaborative Search, personalize Web search using representative user profiles for groups of like-minded users. This approach also alleviates the "cold-start" problem of new users without well-defined profiles. Collaborative Filtering (CF) algorithms, made popular by their use in recommendation systems, are applied in [tao Sun et al., 2005; Xue et al., 2009] to match an individual user with a group profile. [Dalal, 2007] extends community context based personalization with explicit social search and cooperative search methods.

Research has shown that people value information provided by family, friends and other collaborators [Wilson, 2006]. A survey conducted in [Morris, 2008] highlights the fact that users often engage in cooperative search behaviors. This type of (explicit) collaborative search can either be distributed [Morris and Horvitz, 2007], where users interact through electronic means, or co-located [Amershi and

Morris, 2008]. [Twidale et al., 1997] describes a remote asynchronous collaborative search model in which both the search process and product are captured and communicated.

Collaboration in ClusteringWiki is a result of implicit aggregation of community preferences for a given search result cluster tree. Unlike collaborative search approaches that apply personalization of a community profile, ClusteringWiki uses preferences from any user that has edited the cluster tree for the executed query, irrespective of that user profile's similarity to the searcher. In fact, community preferences are applied for logged-out searchers, whose profiles are not known. However, the mass-collaboration editing effort of logged-in users is utilized "free of charge" by logged-out users.

Social search. Search is often a social collaborative experience. Social search extends personalized and collaborative search by giving special consideration to content created or touched by users in the searcher's social graph. Example forms of user contributions include shared bookmarks, tagging of content with descriptive labels, and even explicit assistance through chat or email. Currently there are more than 40 such people-powered or community-powered social search engines, including

Eurekster Swiki[4], Mahalo[5], Wikia[6], and Google social search[7]. [Evans and Chi, 2008] model social search behavior through a survey of 150 users on Amazon's Mechanical Turk.

In ClusteringWiki statistically significant paths are chosen from community preferences without special consideration for a searcher's social network. Therefore, the ClusteringWiki collaboration strategy is distinct from that employed in social search.

Tagging / Social Annotations. Tagging allows users to associate objects with tags, generally keywords or short phrases, as a means of annotating and categorizing them. While users are primarily interested in tagging for their personal use, tags in a community collection tend to stabilize into power law distributions [Halpin et al., 2007]. Collaborative tagging systems leverage this property to derive folksonomies and improve search [Xu et al., 2008]. [Sigurbjörnsson and van Zwol, 2008] and [Song et al., 2008] have studied tag suggestion as a means to minimize user tagging effort. [Zollers, 2007] has shown that most user tags are phrases rather than single words.

In ClusteringWiki, users tag clusters to organize search results, and the tags can be shared and utilized in a similar way as in collaborative tagging. Since

[4]www.eurekster.com

[5]www.mahalo.com

[6]answers.wikia.com/wiki/Wikianswers

[7]googleblog.blogspot.com/2009/10/introducing-google-social-search-i.html

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. While social tagging is leveraged to build a concept hierarchy (folksonomy) from the bottom up, ClusteringWiki automatically generates top-down concept hierarchy sections editable by the searcher.

Prototypes that allow user editing and annotation of search results exist, e.g. U Rank by Microsoft[8] and Searchwiki by Google[9]. Rants [Gao and Jan, 2010] 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.

Semantic Web. Another popular use of annotations is to assign machine readable meaning to words and phrases in free-text. Semantic annotations, as they are called, are metadata attached to parts of text which assign them formal semantics (knowledge), often though ontology references [Horrocks, 2008]. As first envisioned by Sir Tim Barners-Lee in 2001 [Berners-Lee et al., 2001], semantic annotations

[8]research.microsoft.com/en-us/ projects/urank

[9]googleblog.blogspot.com/2008/11/searchwiki-make-search-your-own.html

would allow the creation of intelligent Web search agents capable of automatically interacting with data and other agents to provide complex and specific answers to our queries. This futuristic concept of the Web has been dubbed the Semantic Web.

The task of on-demand integration of data, without participation by humans, cannot be completed without semantic annotation of the current Web content. As a result, mass-collaborative projects have been started, such as Web Ontology Language (OWL), to create ontologies which span the Web and to semantically annotate Web content.

The scope of ClusteringWiki and the Semantic Web are the same: to improve search performance. However, ClusteringWiki provides direct answers to the query given, without intelligently deducing and retrieving answers to subsequent follow-up questions, as a Semantic Web agent would do. The mass-collaborative effort to build semantic Web ontologies is similar to the ClusteringWiki collaborative effort which builds annotation hierarchies.

2.3 Why Personalized and Collaborative Clustering

The initial goal of the World Wide Web was to make information readily available to whomever wished and was authorized to retrieve it [Zimmerman, 2000]. Two decades later, the Web is so much more: a place for social gathering, self-expression, entertainment, business, etc.

Web 2.0. Web 2.0 ushered in a new era in Internet publishing. It changed the web from a medium where information was made available for users to consume to one where users work together to create and share information. Two themes stand out at the core of Web 2.0: the *Web as Platform*, and *users control data.* Web as platform means applications are built with Web community authorship in mind. Powerful platforms like Amazon, Wikipedia, eBay, YouTube, Twitter, and Facebook harness the collective intelligence of the masses to build services [O'Reilly and Battelle, 2009]. While their actions benefit all in the community, individual authors are in charge of and responsible for their own content.

ClusteringWiki is built on the same Web 2.0 principles. It works to improve search for the Web community through the efforts of many individual authors who manage their own search result cluster edits.

Mass collaboration. One of the effects of the Web 2.0 movement has been mass collaboration. A mass collaboration system uses a large number of people to help solve a problem. User collaboration can be either implicit, e.g. playing an online game which collaboratively is used for optical character recognition [von Ahn and Dabbish, 2004], or explicit. As defined by [Doan et al., pear], users of explicit mass collaboration systems expressly provide data or services by:

- *Evaluating.* Users evaluate products, services, or other users. (Netflix, Amazon)

- *Sharing.* Users share knowledge, products, services, etc. (Wikinews, Technorati, [Bernardo, 2007])
- *Networking.* Users form connections with other users, building a graph which is exploited to provide services. (Twitter, Facebook)
- *Building artifacts.* Users coordinate efforts to build a product. (Linux, Hadoop)
- *Executing tasks.* Users execute sub-tasks, providing a community solution for a larger task. (Mechanical Turk)

Mass collaboration has been applied to many aspects of the Web search problem. Wikia Search[10] Alpha, launched January 7, 2008, used the power of mass-collaboration to develop and popularize open-source search engine software. The site has been replaced by WikiAnswers[11], a Wikia site allowing users to ask questions, in lieu of traditional Web search, which are then answered by the community. The Eurekster swicki[12] is a customized, community-driven social search portal. Swiki owners can customize sites the search engine should crawl (whitelist), sites it should ignore (blocklist), and can even manually add new search results or comment on existing ones. Google SearchWiki[13] allows users to customize their

[10]http://search.wikia.com

[11]http://wiki.answers.com/

[12]http://www.eurekster.com

[13]http://googleblog.blogspot.com/2008/11/searchwiki-make-search-your-own.html

Google search results by ranking, removing, or adding comments to them. Comments are public and thus shared with the community. [Gao and Jan, 2010] describes the first academic system for search result rank editing and provides capabilities for sharing edits within the searcher's social network. Freebase[14] is an open, searchable, community-driven repository of structured data. Mass collaboration roles in the Freebase project include data contributors and curators, schema builders, and application developers.

Search is a problem that cannot be perfectly solved by machines. ClusteringWiki enables users to author new creative content by editing cluster hierarchies, and the efforts of individual users are shared with the community as a collaborative effort to improve search performance. It can thus be categorized as a sharing explicit mass collaboration system. A challenging task when designing a mass collaboration system is identifying portions of the final task that can be performed by a crowd and finding ways to combine the individual user results. ClusteringWiki treats cluster edits independently and uses a novel root-to-leaf node path approach to aggregate significant edits from multiple users.

[14]http://www.freebase.com/

2.4 Improving Search Through Presentation

Given its exponential growth, the Web likely contains documents on many sub-topics within any given topic. Additionally, user's queries are often short and ambiguous, vulnerable to the problem of polysemy. Given multiple senses associated with the query, how can we know which one the searcher has in mind? While some have attempted to answer this question through analyzing the searcher's context (e.g. Personalized Search), diversification and search result clustering employ alternate presentations of the search results to let the user quickly choose thier intended meaning.

Diversification. Diversification works under the premise that the user's search intent cannot be fully known and instead spreads results with different characteristics throughout the search result list. This approach aids users exploring different themes within a topic. Results can be diversified based on their similarity to each other [Zhai et al., 2003; Zhang and Hurley, 2008], by following the maximal marginal relevance (MMR) paradigm [Carbonell and Goldstein, 1998], based on novelty [Clarke et al., 2008], or based on topic coverage [Agrawal et al., 2009; Carterette and Chandar, 2009]. Personalized Web search approaches are also applied to the diversification problem in [Radlinski and Dumais, 2006; Rafiei et al., 2010]. While most diversification methods focus on re-ranking results, [Santos et al.,

2010] explores query reformulations to retrieve diverse results for a given topic. [Drosou and Pitoura, 2010] surveys the field of search result diversification.

While diversification tries to provide users with results from many query senses in the same page, ClusteringWiki takes a different approach, providing label hierarchies for these senses and allowing the user to filter results pertaining to a chosen sense. Diversification can be detrimental for informational queries, as users may have to chase sub-topic results through many result pages before finding what they are looking for. ClusteringWiki combats this problem by "gathering" sub-topic results and allowing the user to find them easily.

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 [MacQueen, 1967], 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 agglomerative (bottom-up) such as AGNES [Kaufman and Rousseeuw, 1990], divisive (top-down) such as DIANA [Kaufman and Rousseeuw, 1990], or use hybrid clustering approaches [Zhao and Karypis, 2002].

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 [Everitt et al., 2001; Jain and Dubes, 1988]. 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 [Rijsbergen, 1979].

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 [Broder, 2002; Rose and Levinson, 2004]. 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 [Hearst and Pedersen, 1996; Käki, 2005; Tombros et al., 2002; Zamir and Etzioni, 1999].

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 [Carpineto et al., 2009; Griffiths et al., 1986; Salton, 1971]. It has been shown that query-specific, on-line,

post-retrieval clustering, i.e., clustering search results, produces much superior results [Hearst and Pedersen, 1996].

Scatter/Gather [Hearst and Pedersen, 1996; Pirolli et al., 1996] 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 [Zamir and Etzioni, 1998, 1999] (retired in 2000) introduced the well-known Suffix Tree Clustering (STC) algorithm that groups Web search results into clusters which are 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 utilizes STC as well as Lingo [Osinski and Weiss, 2005], a clustering algorithm based on singular value decomposition.

Other related work from the Web, IR and data mining communities exists. [Zeng et al., 2004] explored supervised learning for extracting meaningful phrases from snippets, which are then used to group search results. [Kummamuru et al., 2004] proposed a monothetic algorithm, where a single feature is used to assign documents to clusters and generate cluster labels. [Wang and Zhai, 2007] investigated the use of past query history in order to better organize search results for future queries. [Lee et al., 2009] studied search result clustering for object-level search engines that automatically extract and integrate information on Web objects.

[Carpineto et al., 2009] surveyed Web clustering engines and algorithms.

Methods of organizing search results based on text categorization are studied in [Chen and Dumais, 2000; Dumais et al., 2001]. In this work, a text classifier is trained using a Web directory, and search results are then classified into the predefined categories. The authors designed and studied different category interfaces and they found that category interfaces are more effective than list interfaces. However predefined categories are often too general to reflect the finer granularity aspects of a query.

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 [Basu et al., 2004; Cohn et al., 2009] and interactive clustering [Balcan and Blum, 2008; Bekkerman et al., 2007; Ji and Xu, 2006; Raghavan et al., 2005; Wagstaff et al., 2001] 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.

Clustering aggregation. The problem of clustering aggregation tries to find, among a set of clusterings, one that agrees the most with the entire set of clusterings. Often, when the clusterings are produced by different clustering algorithms, the problem is known as ensemble clustering, and has been studied extensively [Fern and Lin, 2008; Gionis et al., 2007; Li et al., 2010; Singh et al., 2008].

Similar to clustering aggregation, ClusteringWiki aggregates multiple user clusterings to form a final community-edited cluster tree. However, unlike ensemble clustering, it does so without regard to the membership of each cluster or its agreement with the set of initial clusterings. Instead, significant edited root-to-leaf paths are chosen and applied to the community clustering.

CHAPTER III

CLUSTERINGWIKI

In this chapter I will present ClusteringWiki, including its main architecture and the design principles of the clustering and editing frameworks.

3.1 Overview

Hierarchical clustering forms a tree structure. A root cluster exists and each cluster can have 0 or more sub-clusters or results. There are different ways to represent a cluster hierarchy in a Web application. One way is to use the concept of "file folders" to represent clusters. As extension of this concept, executing cluster edits through *copy/paste* and *drag and drop* type functionality is instantly familiar to most users. Additionally, it allows users complete control, with few restrictions, to reshape the clusters as they see fit.

Architecture. Figure 3.1 shows the two ClusteringWiki 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

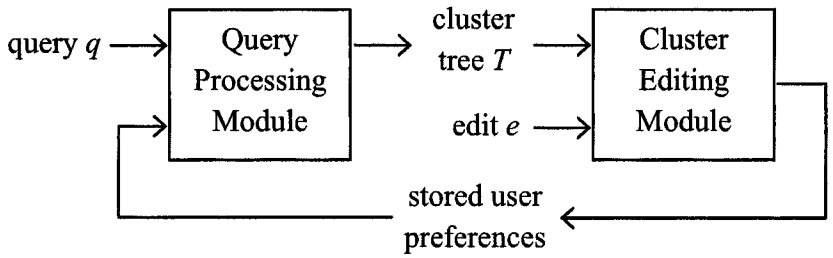


Figure 3.1: Main architecture of ClusteringWiki.

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 thesis. 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, I obtain $P_{q,U}$, a set of aggregated preferences for query q. I 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 $\forall q, P_{q,u} \in P_u$. I 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 $P_{q,u}$ that represents 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}.

3.2 Framework

This section introduces the ClusteringWiki framework in detail. In particular, I present the algorithms for the query processing and cluster editing modules and explain their main components.

3.2.1 Query Processing

Algorithm 1 presents the pseudocode for the query processing algorithm in 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 3.2.2. 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, I implement a custom tokenizer that identifies whether a token is a word, numeric, punctuation mark, capitalized, all

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.

retrieve a set R of search results for query q;
cluster R to obtain an initial cluster tree T_{init};
$P \leftarrow \emptyset$; //P is the set of paths to be enforced on T_{init}
if (u is logged-in) **then**
  $q' \leftarrow Trans(q, u)$;
  if ($q' \neq NULL$) **then**
    $P \leftarrow P_{q',u}$; //use applicable personal preferences
  end if
else
  $q' \leftarrow Trans(q, U)$;
  if ($q' \neq NULL$) **then**
    $P \leftarrow P_{q',U}$; //use applicable aggregated preferences
  end if
end if
$T \leftarrow T_{init}$; //initialize T, the cluster tree to present
clean P; //remove $p \in P$ if its result node is not in R
for each $p \in P$
  if (p is positive) **then**
    $T \leftarrow T \cup \{p\}$; //add a preferred path
  else
    $T \leftarrow T - \{p\}$; //remove a non-preferred path
  end if
end for
trim(T, C); //make T consistent
present(T); //present the set of paths in T as a tree

caps, etc. I 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.

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 [Ukkonen, 1995]. 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 thesis is not the production, but rather the modification, of 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 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.

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;
```

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 may be 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. I will discuss the constraint set C in detail in Section 3.2.2.

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 3.2, 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 $< 1, 5 >$, A goes in front of D. "Other" is a special label that is always listed at the end behind all its siblings.

Discussion. As [Kale et al., 2010] 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.

3.2.2 Cluster Editing

Before explaining the algorithm handling user edits, I will 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, I 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.

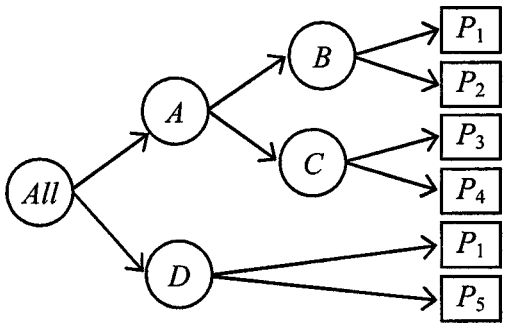


Figure 3.2: Example cluster tree.

- *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.

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 3.2, we can copy D to A. For this edit,

  $P_e = \{All \to A \to D \to P_1, All \to A \to D \to P_5\}$. $NP_e = \emptyset$ for any edit of this type.

- e_2: copy a result node to a bottom label node.

  Example: in Figure 3.2, 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 3.2, 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 3.2, 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 a bottom label node. In particular, recursive creation of non-bottom labels is a way to add levels to

cluster trees.

Example: in Figure 3.2, 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

$NP_e = \{All \rightarrow D \rightarrow P_1, All \rightarrow D \rightarrow P_5\}$.

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};
```

The editing framework results in several *favorable properties.* First, 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 are 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. Preference aggregation is described further 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, let $P^0_{q,U}$ signify 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, the system keeps track of two *best* paths: a positive one with the most count and a negative one with the most count. A best path is marked 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.

CHAPTER IV

IMPLEMENTATION

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

4.1 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, to attach appropriate tree functionality, and to display results contained in the root cluster node.

Retrieving query results. In order to easily test ClusteringWiki with multiple search engines and data sources, I 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[1] expects a zero-based first requested result parameter, while Yahoo!

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

Search API[2] 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, I 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 that is 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 non-textual characters and stop

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

words, the remaining words are stemmed using the Porter[3] algorithm creating a list of document terms. The stop word list used is that from the Apache Snowball[4] 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, my tokenization process identifies additional information about each token which it stores as bitwise flags in a *short* value. I identify whether a token is a word, numeric, or punctuation mark, or whether a word token is capitalized, all caps, and 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

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

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

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. I 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. I then assign to that cluster all documents from the collection being clustered that contain any of the label terms. The details of the frequent phrase algorithm, as applied for each level of clustering, are as follows:

1. I 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. I also note document boundaries in a separate integer stack.

2. I apply Ukkonen's linear time online suffix tree construction algorithm to create a suffix tree for the integer sequence. I also keep track of document ids for each of the phrase suffixes entered in the suffix tree.

3. I walk the suffix tree, identifying and retrieving frequent phrases with a given minimum and maximum length and minimum support. For our current implementation I experimentally chose to use minimum term phrase length 2, maximum term phrase length 5, and minimum document support 2. For each phrase I also retrieve from the suffix tree a bit set representing all the documents that contain the given phrase.

4. I 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. I 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, I build a cluster and assign it all documents containing any of the terms in the label.

6. I 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. I heuristically choose to subcluster a given cluster if it contains more than 5 documents and the path of the child cluster does not violate the given tree *height constraint*. However, I 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[5], 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

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

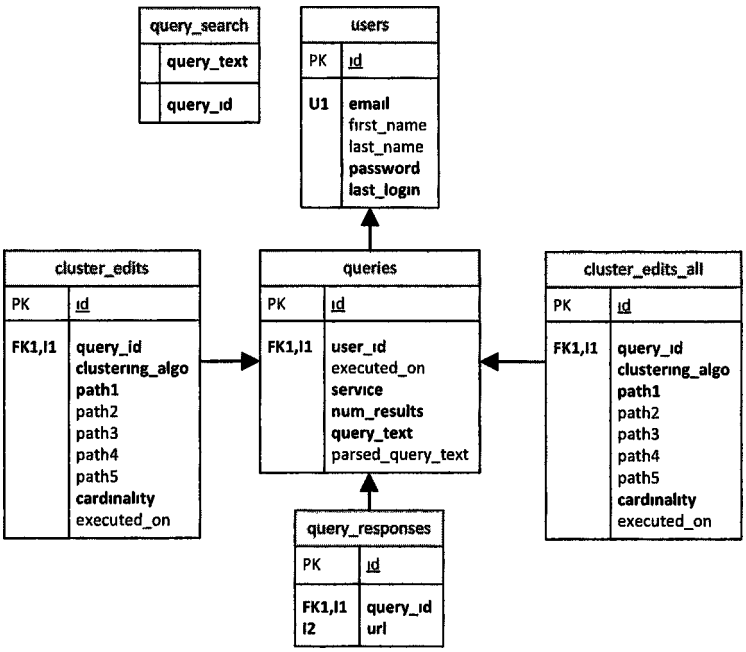


Figure 4.1: ClusteringWiki database schema.

stored in a MySQL database. Figure 4.1 shows the ClusteringWiki database schema. Fields and tables not related to cluster editing have been omitted.

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. 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, I 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, I have chosen to keep the path in $h-1$ *path* fields within the *cluster_edits* tuple. Alternative approaches for storing a path without a height constraint include adjacency list and nested set storage models[6]. 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,

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

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 no 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 either 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 tree. 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 tree. Using a JavaScript object allows the creation of an efficient stateful representation of the cluster tree and its operations. *EditableClusterTree* first builds an internal cluster node object hierarchy from the data 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.

4.2 Cluster Editing

Cluster editing in ClusteringWiki involves editing the in-page cluster tree 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 preference cardinality. If the path already existed in the database, the preference cardinality is added to the existing 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. I 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, I 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 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.

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.

CHAPTER V

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. I have conducted a comprehensive experimental evaluation detailed below.

5.1 Methodology and Metrics

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

Data sources. Multiple data sources were used in the 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 [Sandhaus, 2008] 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, I focused on *correctness* and *efficiency*. I 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 the same query, I used the stable New York Times data source when evaluating ClusteringWiki correctness. I chose queries that returned at least 200 results.

I evaluated system efficiency by monitoring query processing time in various settings. In particular, the following were 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, I executed 5 queries, each twice. The queries were chosen such that at least 500 search results would be returned. For each query, I 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 more closely simulate the server side search engine behavior. The time spent on applying preferences depends on the number of applicable stored paths. For each query, I 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 a query's cluster of search results.

Utility evaluation methodology. For utility evaluation, I 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 Texas State University-San Marcos and users were chosen on a first-come first-serve basis. The participants were primarily undergraduate, while a few were graduate, college students.

I 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 [Broder, 2002; Rose and Levinson, 2004] 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 [Manning et al., 2008]. In practice, a user may explore a varied number (e.g., 5 or 10) of relevant results for an informational query. Thus, I considered 2 types of informational queries. In addition, I 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, I 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.

A subset of the chosen queries for each data set are presented below. The entire set can be found in [Anastasiu et al., 2010]. 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 data source, R_{10} and R_5 queries:

- *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.

Google data source, R_1 queries:

- *Query*: Texas State University-San Marcos

  Description: Find the page for the graduate college at Texas State University-San Marcos.

- *Query*: Longhorns

  Description: Find the page for the Texas Longhorn Breeders Association of America.

- *Query*: Byron J. Gao

  Description: Find the KDD 2007 Conference program information page, Dr. Byron J. Gao had a paper published in that conference with Dr. Martin Ester.

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 I used to measure the search result exploration effort exerted by a user in fulfilling thier information need. [Koren et al., 2008] used

a similar metric under a probabilistic model instead of a 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*. I assume that 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 my observation for this particular user study, I empirically used 0.25 for this probability.

5.2 System Evaluation Results

ClusteringWiki operation is independent of parameters such as number of results or chosen clustering algorithm. I 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 this system. I used manually executed function tests to verify that ClusteringWiki works as indented. A description of each test can be found in [Anastasiu et al., 2010]. All tests were executed successfully and no anomalies were encountered.

System efficiency. I 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 5.1: Efficiency evaluation using Yahoo! data source

Number of results	**100**		**200**		**300**		**400**		**500**	
Type of clustering	**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
Preprocessing	0 009	0 011	0.052	0 052	0 037	0 037	0 049	0 112	0 152	0 150
Initial clustering	0 004	0 005	0 051	0 040	0 033	0 042	0 104	0 063	0 118	0 144
Applying preferences	0 006	0 007	0 049	0 012	0 015	0 011	0 021	0 015	0 08	0 013
Presenting final tree	0 143	0 172	0 249	0 278	0 341	0 421	0 451	0 66	0 723	0 752
Other	0 396	0 416	0 469	0 524	0 624	0 684	0 558	0 593	0 853	0 912
Total execution time	0 160	0 194	0 401	0 381	0 426	0 511	0.624	0 850	1 073	1 059
Total response time	1 535	1 628	2 179	2 127	2 665	2 585	3 029	3 022	3 604	3 632

Tables 5.1 and 5.2 show the averaged (over 10 queries) runtime in seconds for all 6 portions of the total response time for the two tested data sources. In addition, I 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 the prototype is responsible for. The remaining time is irrelevant to the way the prototype is designed and implemented. From the table we can see that:

Table 5.2: Efficiency evaluation using New York Times data source

Number of results	**100**		**200**		**300**		**400**		**500**	
Type of clustering	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**	**F**	**H**
Retrieving results	0 102	0 113	0 131	0 130	0 285	0 259	0 326	0 338	0 419	0 387
Preprocessing	0 019	0 019	0 067	0 066	0 068	0 069	0 096	0 168	0 189	0 187
Initial clustering	0 005	0 006	0 022	0 025	0 035	0 041	0 053	0 062	0 147	0 196
Applying preferences	0 012	0 011	0 013	0 024	0 018	0 011	0 013	0 017	0 016	0 014
Presenting final tree	0 282	0 279	0 372	0 449	0 591	0 691	0 751	0 872	0 846	0 941
Other	0 338	0 497	0 478	0 678	0 589	0 672	0 625	0 937	0 736	0 868
Total execution time	0 317	0 315	0 473	0 565	0 713	0 813	0 913	1 118	1 197	1 338
Total response time	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 ClusteringWiki was implemented by a search company.

- Applying preferences takes less than 1/10 second in all test cases, which certifies the efficiency of my "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.

Figure 5.1 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:

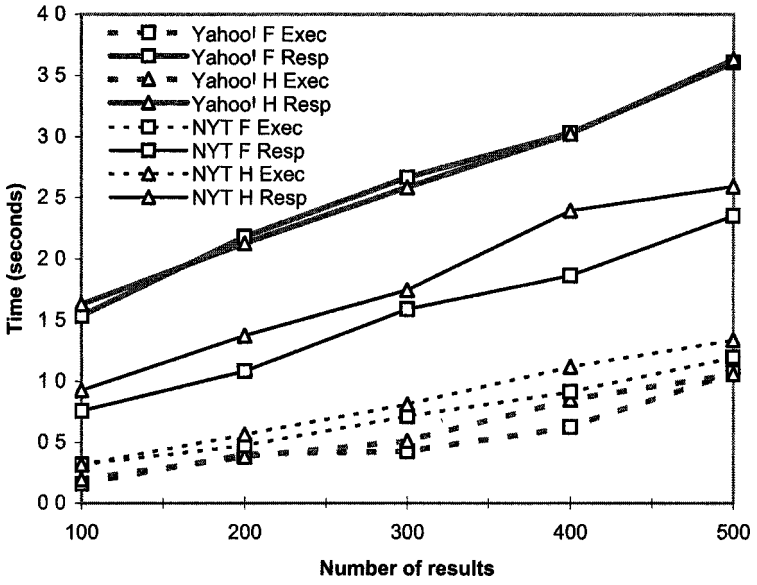


Figure 5.1: Efficiency evaluation.

- Response and execution time trends are linear, testifying to the scalability of the 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 the prototype with the data sources.

- Execution times for Yahoo! are shorter than New York Times due to the shorter titles and snippets.

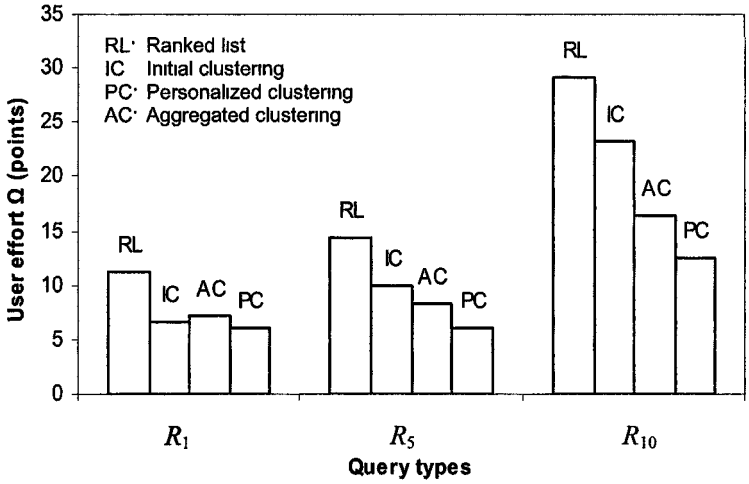
5.3 Utility Evaluation Results

System utility. Figure 5.2 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:

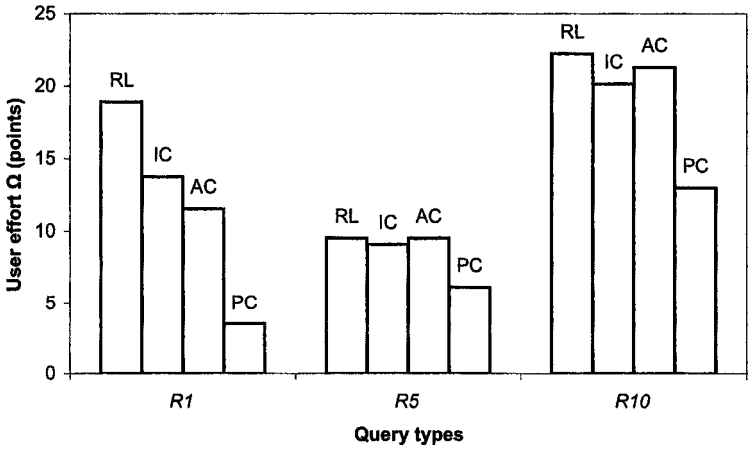
- Clustering saves user effort in informational and deep navigational queries, with personalized clustering being the most effective, saving up to 50% of user effort.

- Aggregated clustering also provides significant 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 or require user login.

  In evaluating aggregated clustering, I made sure that the users browsing the aggregated clusters were not the same ones who edited them.



(a) Google data source



(b) New York Times data source

Figure 5.2: Utility evaluation on Google and New York Times data sources

- 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 I have collected as well as the user feedback, I can make the following observations:

- 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, I 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 I received show that, while most users found the

application interface easy to use, they desired additional convenience features. The user study participant comments can be found in [Anastasiu et al., 2010].

CHAPTER VI

CLUSTERINGWIKI2

This thesis introduced the concept of cluster editing as a way of personalizing a search result clustering presentation, and ClusteringWiki, the first prototype for personalized and collaborative clustering of search results. ClusteringWiki uses the concept of "file folders" to represent cluster hierarchies and, by extension, includes file and folder *copy/paste* and *drag-and-drop* functionality to execute cluster editing (personalization). I conducted a comprehensive experimental evaluation of ClusteringWiki, including a user study, which proved the potential of search result cluster personalization to improve search utility. The feedback received from users highlighted some advantages and limitations of the current implementation, presented below.

Advantages:

- The ClusteringWiki interface was user-friendly and did not require previous training.
- Users instantly associated clusters/sub-clusters with folders/sub-folders and search results with files.

- Users took advantage of all provided editing operations to personalize their clusters.

Limitations:

- Editing was not always transparent to the community. At times, some users did not understand the rationale behind previous user edits.
- When working on result cluster placement, users wanted to edit multiple results at once.
- Some users wanted the ability to customize results by tagging them with personally identifiable keywords.

As mentioned in Chapter 3, there is more than one way to represent a cluster hierarchy and perform cluster edits. Keeping in mind the original system usability goal, I designed a new system which addresses the ClusteringWiki limitations identified in the user study. This chapter introduces the new system, ClusteringWiki2, and highlights the similarities and differences between the clustering and editing frameworks of the two systems.

6.1 Overview

While ClusteringWiki uses typical file and folder specific operations for editing clusters, ClusteringWiki2 takes a different approach, allowing users to annotate

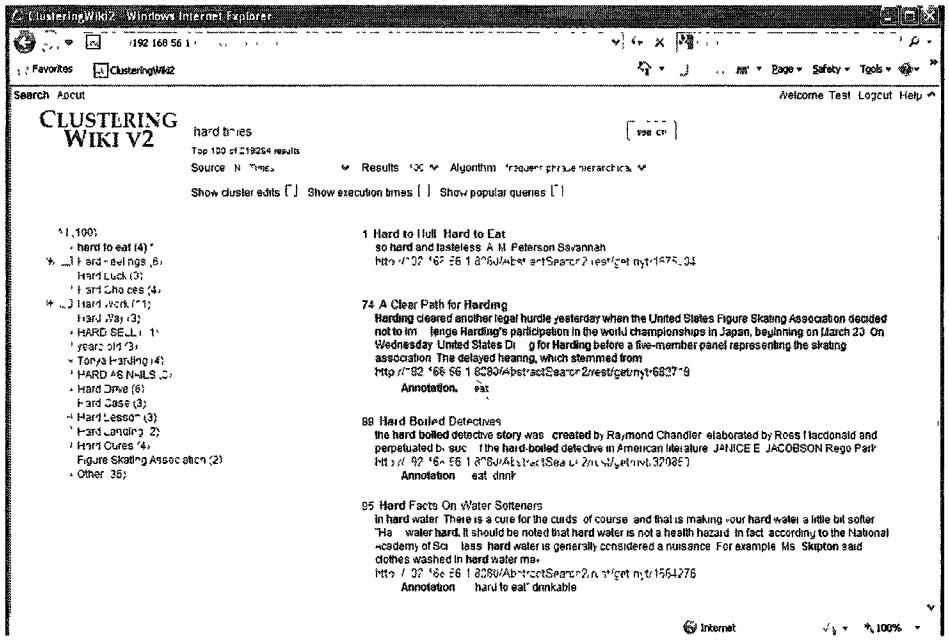


Figure 6.1: Snapshot of ClusteringWiki2.

search results both directly and through their cluster participation. Cluster labels are themselves annotations and can be composed of *positive terms*, *negative terms*, *positive phrases* and *negative phrases*. ClusteringWiki2 establishes a communication channel between editors by enforcing a reasonable, straightforward "membership condition": to be included in a cluster, a search result or its annotation must contain all the positive terms and positive phrases and not contain any of the negative terms or negative phrases in the cluster's *label path*[1]. This introduces transparency among editors and allows better collaboration.

[1]Considering the executed query as the root node label, a cluster's *label path* is the set of labels along the path from the root node to the given cluster node.

Figure 6.1 presents a snapshot of ClusteringWiki2[2]. The architecture of ClusteringWiki2 is identical to that of ClusteringWiki, containing two modules, *query processing* and *cluster editing*, which are used cyclically over time. The *path* approach to storing and incorporating user preferences in clusters was very efficient in ClusteringWiki, and it validated the system architecture design.

Query processing. ClusteringWiki2 query processing is similar to that of ClusteringWiki. The set of search results R_{init} is retrieved from a chosen data source in response to a query q from a user u. R_{init} is then clustered using a chosen clustering algorithm to produce an initial cluster tree T_{init}. Then, ClusteringWiki2 applies P, an applicable set of stored user preferences, to T_{init} and R_{init} and presents a modified cluster tree T and an annotated set of results R that respect P.

In ClusteringWiki, preferences represent root-to-leaf paths within the cluster, where leaf nodes are the results contained in *bottom level nodes.* ClusteringWiki2 continues to use the root-to-leaf path approach. However, leaf nodes are the bottom level nodes rather than the search results contained in those nodes. This allows search result preferences and cluster preferences, even though conceptually linked, to be treated separately. Thus P is the union of P_c, the set of stored cluster preferences, and P_r, the set of stored search result preferences.

P_c is applied to T_{init} and P_r is applied to R_{init} to create T and R respectively.

[2]dmlab.cs.txstate.edu/ClusteringWiki2/.

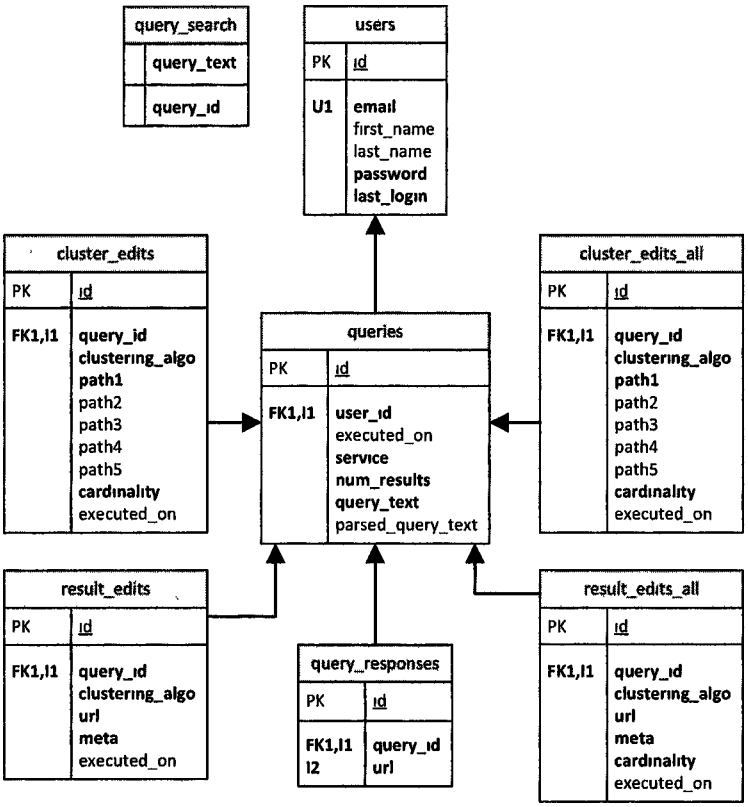


Figure 6.2: ClusteringWiki2 database schema.

P_r does not alter R_{init}, the input data. Rather, each annotation preference $p_r \in P_r$ associated with a result r is attached as metadata to r *iff* $r \in R$. Note that the membership condition allows result annotations to be independent of cluster membership. Therefore, unlike in ClusteringWiki, result preferences are only associated with the result r and query q and not connected with the cluster path a result may be found in. This leads to a significant reduction in the number of stored user preferences.

Figure 6.2 shows the database schema for ClusteringWiki2. Fields and tables not related to cluster editing have been omitted. The schema is identical to that for ClusteringWiki, adding only two tables, *result_edits* and *result_edits_all*, for storing result annotation preferences.

The procedure for identifying which preferences should be applied to query q, utilizing *preference transfer* and *query transfer* as appropriate, is identical to that in ClusteringWiki. Similarly, the procedure for storing and applying *positive* and *negative* paths from P_c to T_{init} is the same in the two systems. Please see Chapter 3 for details.

ClusteringWiki assigns search results to clusters according to the chosen clustering method. ClusteringWiki2 ignores the initial assignment of results to clusters by the base clustering algorithm. After both cluster preferences and result preferences are applied, ClusteringWiki2 traverses T and assigns to each node those results from R that respect the *membership condition.*

Cluster and result editing. If logged-in, a user u can edit the cluster tree T for query q by creating, deleting, or modifying nodes, and they can edit a result in R by modifying its attached annotation. Similar to ClusteringWiki, user edits will be validated against a set C of consistency constraints before being written to $P_{q,u}$, the set of preferences for q previously specified by u.

By combining preferences in $P_{q,u}$ for all users who have edited the cluster tree

T for query q, I obtain $P_{q,U}$, a set of aggregated preferences for query q. I use P_u to denote the collection of clustering preferences by user u for all queries, and P_U to denote the collection of aggregated preferences for all queries from all users. P_u and P_U are maintained over time and used by ClusteringWiki2 in processing queries for the user u.

Reproducibility. It is easy to verify that ClusteringWiki has the property of reproducing edited cluster trees. In particular, if T_{init} remains the same in a subsequent query, after a series of user edits on T_{init} to produce T, the same T will be produced after enforcing the stored user preferences generated from the user edits on T_{init}. Similarly, if R_{init} remains the same in a subsequent query, after a series of user edits on R_{init} to produce R, the same R will be produced after enforcing the stored user preferences generated from the user edits on R_{init}. Finally, the membership condition leads to a stable algorithm for assigning results to cluster nodes, described further in the following.

6.2 Framework

This section introduces the ClusteringWiki2 framework in detail. In particular, I present the algorithms for the query processing and cluster editing modules, and then explain their main components.

6.2.1 Query Processing

The query processing framework is the same in ClusteringWiki2 as in ClusteringWiki, with the exception that P contains both P_c, the cluster preferences, and P_r, the result preferences. P_c is applied to T_{init} in the same way as P was applied to T_{init} in ClusteringWiki. If the user u is logged in, preferences in the set P_r are applied to R_{init} by attaching each annotation p_r associated with a result r to the result if $r \in R$. Otherwise, p_r will be formed by aggregating the set of statistically significant preferences for the result r from all users that have edited r.

Annotation aggregation. A user preference for a result r is an annotation. An annotation $\mathcal{A}$ is the union of the set of positive terms $\mathcal{T}$, negative terms $\mathcal{NT}$, positive phrases $\mathcal{P}$, and negative phrases $\mathcal{NP}$ in that annotation. Algorithm 4 describes the simple algorithm used to aggregate a set of annotations. Lines $1 \sim 5$ initialize the aggregated annotation $\mathcal{A}$ which will be returned. Line 6 initiates the loop traversing each annotation in the input set. For each annotation in the input set, the four sets of loops at lines $8 \sim 14$, $15 \sim 21$, $22 \sim 28$, and $29 \sim 35$ add a term t or a phrase p to the aggregated annotation $\mathcal{A}$ if the opposite of t or p is not present in $\mathcal{A}$. Alternatively the opposite of t or p is removed from $\mathcal{A}$.

Cluster result assignment. Once T and R have been devised, ClusteringWiki2 assigns results to each cluster node according to the *membership condition*. In

Algorithm 4 *aggregate($\mathcal{S}$)*

Input: $\mathcal{S}$: $\mathcal{S}$ is the set of annotations to be aggregated.
Output: $\mathcal{A}$, the aggregated annotation from all annotations in S

1: $\mathcal{A} \leftarrow \emptyset$; //initialize aggregated annotation
2: $\mathcal{T}_\mathcal{A} \leftarrow \emptyset$; //initialize set of positive terms
3: $\mathcal{NT}_\mathcal{A} \leftarrow \emptyset$; //initialize set of negative terms
4: $\mathcal{P}_\mathcal{A} \leftarrow \emptyset$; //initialize set of positive phrases
5: $\mathcal{NP}_\mathcal{A} \leftarrow \emptyset$; //initialize set of negative phrases
6: **for each** $A \in \mathcal{S}$
7: $A := \mathcal{T} \cup \mathcal{NT} \cup \mathcal{P} \cup \mathcal{NP}$; //$A$ is set of positive and negative terms and phrases
8: **for each** $t \in \mathcal{T}$
9: **if** $(t \in \mathcal{NT}_\mathcal{A})$ **then**
10: $\mathcal{NT}_\mathcal{A} \leftarrow \mathcal{NT}_\mathcal{A} \setminus \{t\}$;
11: **else**
12: $\mathcal{T}_\mathcal{A} \leftarrow \mathcal{T}_\mathcal{A} \cup \{t\}$;
13: **end if**
14: **end for**
15: **for each** $t \in \mathcal{NT}$
16: **if** $(t \in \mathcal{T}_\mathcal{A})$ **then**
17: $\mathcal{T}_\mathcal{A} \leftarrow \mathcal{T}_\mathcal{A} \setminus \{t\}$;
18: **else**
19: $\mathcal{NT}_\mathcal{A} \leftarrow \mathcal{NT}_\mathcal{A} \cup \{t\}$;
20: **end if**
21: **end for**
22: **for each** $p \in \mathcal{P}$
23: **if** $(p \in \mathcal{NP}_\mathcal{A})$ **then**
24: $\mathcal{NP}_\mathcal{A} \leftarrow \mathcal{NP}_\mathcal{A} \setminus \{p\}$;
25: **else**
26: $\mathcal{P}_\mathcal{A} \leftarrow \mathcal{P}_\mathcal{A} \cup \{p\}$;
27: **end if**
28: **end for**
29: **for each** $p \in \mathcal{NP}$
30. **if** $(p \in \mathcal{P}_\mathcal{A})$ **then**
31: $\mathcal{P}_\mathcal{A} \leftarrow \mathcal{P}_\mathcal{A} \setminus \{p\}$;
32: **else**
33: $\mathcal{NP}_\mathcal{A} \leftarrow \mathcal{NP}_\mathcal{A} \cup \{p\}$;
34: **end if**
35: **end for**
36: **end for**
37: $\mathcal{A} \leftarrow \mathcal{T}_\mathcal{A} \cup \mathcal{NT}_\mathcal{A} \cup \mathcal{P}_\mathcal{A} \cup \mathcal{NP}_\mathcal{A}$;
38: return $\mathcal{A}$;

ClusteringWiki2 cluster node labels are annotations. Similar to result annotations, a cluster annotation is composed of positive and negative terms and phrases.

Considering the executed query as the root node label, a cluster's *label path* is the set of labels along the path from the root node to the given cluster node. A straight forward approach to the result assignment problem would be to compare terms and phrases between the cluster's label path and each result. A result belongs to a cluster if it (including its annotation) contains all the positive terms and phrases while not containing any negative terms or phrases in the cluster label path annotation. Note that a negative term or phrase in a result annotation indicates the result *does not contain* that term or phrase, even if it is present in the result title or snippet.

ClusteringWiki2 takes advantage of the hierarchical nature of the cluster nodes and implements a recursive method for assigning results to clusters. In doing so, it reduces both the number of results and annotation terms and phrases considered at each subsequent recursion level. The pseudocode of $assignResults(n, D', M)$ is presented in Algorithm 5. Given a cluster node n, a set of potentially covered results D', and a dictionary M correlating terms to results that contain them, the subroutine finds the subset of D' which are results covered by the label path annotation of n, and assigns it to node n if different than the currently assigned result set.

Algorithm 5 *assignResults(n, D', M)*

Input: n, D', M: n is a node. D' is a set of potentially covered results. M is a dictionary correlating terms with results containing them.

Output: n': the node n after results have been assigned to it.

1. $A \leftarrow getAnnotation(n)$; //$A$ is the annotation of node n
2. $A := \mathcal{T} \cup \mathcal{NT} \cup \mathcal{P} \cup \mathcal{NP}$; //$A$ is set of positive and negative terms and phrases
3. $D \leftarrow getDocs(n)$;
4. **for each** $t \in \mathcal{NT}$
5. $D_t \leftarrow termDocs(M, t)$; //retrieve set of results t is in
6. $D' \leftarrow D' \setminus D_t$;
7. **end for**
8. **for each** $t \in \mathcal{T}$
9. $D_t \leftarrow termDocs(M, t)$;
10. $D' \leftarrow D' \cap D_t$;
11. **end for**
12. **for each** $d \in D'$
13. $A_d \leftarrow getAnnotation(d)$; //$A_d$ is the annotation of result d
14. $A_d := \mathcal{T}_d \cup \mathcal{NT}_d \cup \mathcal{P}_d \cup \mathcal{NP}_d$;
15. **for each** $p \in \mathcal{NP}$
16. **if** $p \ni \mathcal{NP}_d$ **then**
17. **if** $contains(d, p)$ OR $p \in \mathcal{P}_d$ **then**
18. $D' \leftarrow D' \setminus \{d\}$;
19. **end if**
20. **end if**
21. **end for**
22. **for each** $p \in \mathcal{P}$
23. **if** $p \ni \mathcal{P}_d$ **then**
24. **if** $\neg$ $contains(d, p)$ AND $p \ni \mathcal{NP}_d$ **then**
25. $D' \leftarrow D' \setminus \{d\}$;
26. **end if**
27. **end if**
28. **end for**
29. **end for**
30. **if** $D' \neq D$ **then**
31. $n' \leftarrow setDocs(n, D')$;
32. $N \leftarrow children(n')$;
33. **for each** $n_c \in N$
34. $n_c \leftarrow assignResults(n_c, D', M)$; //recursive call for children nodes
35. **end for**
36. **end if**
37. return n';

Line 1 of the algorithm retrieves the node's annotation A, which is composed of sets of positive and negative terms and phrases. Note that the algorithm is not retrieving the aggregated annotation of the node's *label path.* This is because we can assume, as a result of the recursive call of *assignResults*, that results in D are already covered by n's parent's label path. *assignResults* is initially called using $n \leftarrow root$ and $D \leftarrow R$.

The subroutine *getAnnotation* simply retrieves an annotation stored either in a cluster node (representing the node's label) or in a search result. Line 3 uses the *getDocs* subroutine to retrieve the current set of results D contained in node n.

Lines $4 \sim 7$ and $8 \sim 11$ present the term-based reduction of D'. First, results containing any of the negative terms in A are removed from D'. Then, only those results containing the positive terms in A are kept in D'. The subroutine *termDocs(M, t)* retrieves the set D_t of results containing the term t from a dictionary M, which is constructed during the preprocessing phase of ClusteringWiki2.

Once D' has been reduced to those results covered by the positive and negative terms in A, phrase coverage is considered sequentially in the remaining results, in lines $12 \sim 29$. The result annotation A_d is first retrieved. Lines $15 \sim 21$ consider negative phrases p in the cluster annotation A. If p is not a negative phrase in the result annotation and it is present in the result title or snippet, the result is

not covered by the cluster annotation and is removed from D'. Similarly, lines $15 \sim 21$ are concerned with positive phrases p in the cluster annotation A. If p is not a positive phrase in the result annotation and it is not present in the result title or snippet, the result is not covered and is removed from D'. The subroutine *contains(d, p)* simply checks whether the phrase p is present in the result d's title or snippet.

The resulting items in D' are covered by the cluster annotation. If D' is different than the already assigned set of contained results D, the algorithm assigns D' to the node n (line 31), and then recursively calls *assignResults* on all children nodes of n (lines $32 \sim 35$).

Ordering. ClusteringWiki2 orders cluster labels within each level in the same way as ClusteringWiki — by lexicographically comparing the lists of original ranks of cluster's associated search results. "Other" is a special label that is always listed at the end, behind all its siblings, and contains parent cluster results not contained in any other sibling cluster.

6.2.2 Cluster and Result Editing

Editing in ClusteringWiki2 is executed through changing annotations, whether they be cluster labels or result metadata. Each annotation change must re-consider result assignments for clusters.

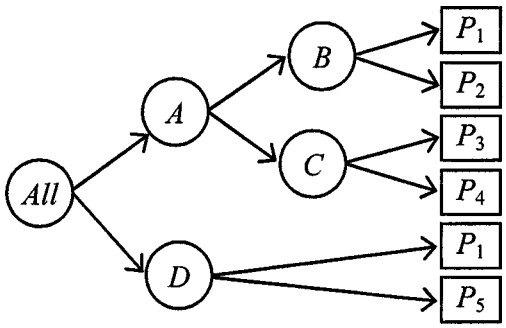


Figure 6.3: Example cluster tree.

Primitive user edits. ClusteringWiki2 implements the following categories of atomic primitive edits that a logged-in user can initiate in the process of tree editing or result annotation. Like in ClusteringWiki, each cluster edit e is associated with P_e and NP_e, which are respectively the set of paths to be inserted to the tree and the set of paths to be deleted from the tree after e. However, leaf nodes in a ClusteringWiki2 cluster tree are cluster nodes rather than results. Note that e_1, e_2, and e_3 in ClusteringWiki2 are equivalent with e_3, e_4, and e_5 in ClusteringWiki, though they differ on the added and deleted paths.

- e_1: modify a non-root label (cluster) node.

  Example: in Figure 6.3, we can modify A to E. For this edit,

  $P_e = \{All \rightarrow E \rightarrow B, All \rightarrow E \rightarrow C, All \rightarrow E\}$ and

  $NP_e = \{All \rightarrow A \rightarrow B, All \rightarrow A \rightarrow C, All \rightarrow A\}$.

- e_2: delete a non-root label node.

Example: in Figure 6.3, we can delete A. For this edit,

$NP_e = \{All \rightarrow A \rightarrow B, All \rightarrow A \rightarrow C, All \rightarrow A\}$. $P_e = \emptyset$ for any edit of this type.

- e_3: create a label node. Recursive creation of labels is a way to add levels to cluster trees.

  Example: in Figure 6.3, we can add E as child of D. For this edit,

  $P_e = \{All \rightarrow D \rightarrow E\}$ and $NP_e = \emptyset$.

- e_4: modify a result annotation.

  Example: modifying a result annotation does not change the structure of the cluster tree. However, it can change the set of results that 0 or more nodes contain, and, by extension, the level ordering of cluster nodes.

Notice that modification and deletion of cluster nodes requires subtree path changes in ClusteringWiki2. ClusteringWiki2 deletes and adds both a node and all its subtree nodes, while ClusteringWiki only deletes and adds leaf (result) node paths. This is due to the different result assignment methods used in the two systems. In ClusteringWiki results are assigned by the clustering algorithm and users have uninhibited control over node edits. Additionally, non-bottom level nodes contain the cumulative set of results from their subtree leaf nodes, and nodes without results are trimmed from presentation. In ClusteringWiki2, results are assigned based on the *membership condition.* Thus, removing all children nodes of a

cluster node does not affect the set of results it contains. The example below further clarifies the difference between the two systems.

Example: in Figure 6.3, a user first modifies the node A to E and then deletes nodes B and C. Finally, they re-execute their query. If ClusteringWiki2 only recorded leaf node path additions and subtractions, as ClusteringWiki does, the following would be the stored edits for the two systems:

ClusteringWiki: $\{ P_e = \{All \rightarrow E \rightarrow B \rightarrow P_1, All \rightarrow E \rightarrow B \rightarrow P_2, All \rightarrow E \rightarrow C \rightarrow P_2, All \rightarrow E \rightarrow C \rightarrow P_3\}$, and $NP_e = \{All \rightarrow A \rightarrow B \rightarrow P_1, All \rightarrow A \rightarrow B \rightarrow P_2, All \rightarrow A \rightarrow C \rightarrow P_2, All \rightarrow A \rightarrow C \rightarrow P_3, All \rightarrow E \rightarrow B \rightarrow P_1, All \rightarrow E \rightarrow B \rightarrow P_2, All \rightarrow E \rightarrow C \rightarrow P_2, All \rightarrow E \rightarrow C \rightarrow P_3\} \}$.

ClusteringWiki2: $\{ P_e = \{All \rightarrow E \rightarrow B, All \rightarrow E \rightarrow C\}$, and $NP_e = \{All \rightarrow A \rightarrow B, All \rightarrow A \rightarrow C, All \rightarrow E \rightarrow B, All \rightarrow E \rightarrow C\} \}$.

After applying all edits following the search re-execution, both clusters contain the node at path $\{All \rightarrow A\}$. However, since they changed the node A to E, the user would expect the path $\{All \rightarrow E\}$ instead. In ClusteringWiki, the node A is trimmed, as it no longer contains any results, thus solving the naming problem. In ClusteringWiki2, the node A will be displayed with results assigned according to the *membership condition*, causing the unexpected result.

When applying e_1 and e_2 correctly, as defined above, the ClusteringWiki2 stored edits will accurately re-create the user changes to the cluster tree. The

corrected stored edits for the example are displayed below:

ClusteringWiki2: { $P_e = \{All \rightarrow E \rightarrow B, All \rightarrow E \rightarrow C, All \rightarrow E\}$, and $NP_e = \{All \rightarrow A \rightarrow B, All \rightarrow A \rightarrow C, All \rightarrow A, All \rightarrow E \rightarrow B, All \rightarrow E \rightarrow C\}$ }.

Once P_e and NP_e are computed, ClusteringWiki2 applies the same editing algorithm as ClusteringWiki for validating and storing edits. For details please see Algorithm 3. *Preference sharing* is implemented the same way in both systems. ClusteringWiki2 implements most of the consistency constraints that ClusteringWikidoes, with some modifications:

- *Presence constraint*: Each initial search result must be present in T. Results not present in a custom node of T will exist in the first level node *Other*, which must be maintained.

- *Path constraint*: Each path of the cluster tree T must start with the root node labeled *All* and end with a cluster leaf node. In case there are no search results returned, T is empty without paths.

- *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.

- *Result annotation constraint*: The length of each result annotation in R must be equal or less than a threshold.

Node result set update following edit. ClusteringWiki2 stores a copy of T and R in memory during an editing session. An editing session starts upon the execution of a query and ends when either another query is executed or the browser session expires. Result node assignment is only maintained in memory in ClusteringWiki2 and not persisted to disk. An edit e updates the in-memory representations of T and R and then stores any path changes P_e and NP_e and any result annotation changes to disk. Executing an edit e may require re-computing search result assignments for some or all of the cluster nodes. ClusteringWiki2 implements efficient methods for each, described below.

Algorithm 6 *changeLabel(n, A', M)*

Input: n, A', M: n is a node. A' is the updated annotation for node n. M is a dictionary correlating terms with results containing them.

Output: n': the node n after results have been re-assigned to it.

1: $A \leftarrow getAnnotation(n)$; //$A$ is the annotation of node n
2: $D \leftarrow getDocs(n)$;
3: $n_p \leftarrow getParent(n)$;
4: $D_p \leftarrow getDocs(n_p)$;
5: $\mathcal{A} \leftarrow getDifference(A, A')$;
6: $\mathcal{T} \leftarrow getTerms(\mathcal{A})$;
7: $D' \leftarrow termsDocs(M, \mathcal{T})$;
8: $D' \leftarrow D' \cup D_p$;
9: $n' \leftarrow assignResults(n, D', M)$;
10: $n' \leftarrow setAnnotation(n', A')$;
11: $reorderLevels()$;
12: $updateOtherNode()$;
13: return n;

e_1: modify a non-root label node. A straight forward approach for this kind of edit would be to consider each $r \in R$ against all nodes in the paths in P_e.

Algorithm 6 describes the process used to update node result sets after a cluster label is modified. Lines $1 \sim 4$ retrieve the node's annotation and contained result set as well as the node parent's contained result set. Line 5 retrieves $\mathcal{A}$, the difference between the current and new annotations for the node n. Line 6 retrieves $\mathcal{T}$, the set of terms, present in either individual changed terms or phrases in $\mathcal{A}$. Line 7 retrieves the set of results that contain any of the terms in $\mathcal{T}$. In line 9, the union of the parent node's results and the results containing the changed terms is passed to *assignResults*, which re-assigns results to the node n and its children. *assignResults* is defined in Algorithm 5. Finally lines $11 \sim 12$ maintain consistency constraints for T by, if necessary, updating the membership of the *Other* node and re-ordering cluster levels. *Other*'s membership is updated by simply removing from the set of all parent results those results covered by the siblings of *Other*.

Algorithm 7 *deleteNode(n, M)*

Input: n, M: n is a node. M is a dictionary correlating terms with results containing them.

Output: n_p: the parent of node n after n has been deleted.

1: $n_p \leftarrow getParent(n)$;
2: $n_p \leftarrow deleteChild(n_p, n)$;
3: $updateOtherNode()$;
4: return n_p;

e_2: delete a non-root label node. This type of edit does not require re-assignment of results. Since the parent node's annotation does not change, it will contain the same set of results as before the edit. No other cluster or result

annotations are changed. Algorithm 7 describes the process used to delete a cluster node. Lines 1 $\sim$ 2 retrieve the parent node and remove n from its list of children. Line 3 maintains consistency constraints for T by, if necessary, updating the membership of the *Other* node.

Algorithm 8 *addNode(n_p, A', M)*

Input: n_p, A', M: n_p is the parent node to which the new node is added. A' is the annotation for the new node n. M is a dictionary correlating terms with results containing them.

Output: n: the added node n.

1: $D_p \leftarrow getDocs(n_p)$;
2: $n \leftarrow addChildNode(n_p)$;
3: $n \leftarrow setAnnotation(n, A')$;
4: $n \leftarrow assignResults(n, D_p, M)$;
5: $updateOtherNode()$;
6: $reorderLevel(n)$;
7: return n;

e_3: create a label node. Algorithm 8 describes the process used to add a node n with label L' as child to parent node n_p. Line 1 retrieves the set of results contained in the parent node. It will be used to assign results to the newly created node. Lines 2 $\sim$ 4 add the node n to the parent node n_p. The subroutine *assignResults* reduces the set of results contained in the parent node to those covered by the annotation A' assigned to n. Lines 5 $\sim$ 6 maintain consistency constraints for T by, if necessary, updating the membership of the *Other* node and re-ordering n's tree level, which places n in the correct place within the level.

e_4: modify a result annotation. Modifying a result annotation can affect all

nodes in the cluster tree. A straightforward solution for this operation would be to re-consider assigning the result to each node in the cluster tree. Checking the *membership condition* would require comparing each label term and phrase of each node with the result title and snippet. ClusteringWiki2 reduces both the number of nodes it will consider for assignment as well as the terms and phrases it will check against the result. First, the terms and phrases considered are reduced by only considering terms and phrases in the annotation difference between the old and new annotations ($getDifference(A, A')$), and vice-versa ($getDifference(A', A)$). The operation also updates the dictionary M, which correlates terms with results containing them. The following describes the sets of terms and phrases that are considered when modifying a result r, and the update operations on M for each of the members in those sets.

- *added positive terms*: for each term t in the set, add r to the result set at index t.
- *removed positive terms*: for each term t in the set, remove r from the result set at index t only if result title or snippet does not contain t.
- *added negative terms*: for each term t in the set, remove r from the result set at index t.
- *removed negative terms*: for each term t in the set, add r to the result set at

index t only if result title or snippet contains t.

- *added positive phrases*: for each term t in each phrase p in the set, add r to the result set at index t.
- *removed positive phrases*: for each term t in each phrase p in the set, remove r from the result set at index t only if result title or snippet does not contain t.
- *added negative phrases*: no update.
- *removed negative phrases*: no update.

No update to the dictionary M is necessary for *added negative phrases* and *removed negative phrases* as the terms of each phrase, if already in the result, continue to exist independently of the phrase within the result. After updating M, the four sets of terms and four sets of phrases are respectively combined into a set of changed terms $\mathcal{T}$ and a set of changed phrases $\mathcal{P}$.

Algorithm 9 describes *assignResult(n, $\mathcal{T}$, $\mathcal{P}$, r)*, the method used to re-assign result r to the cluster tree. After updating a result annotation in R, *assignResult* is called given $n \leftarrow root$ and $assign \leftarrow FALSE$. The algorithm skips checking terms and phrases and eagerly re-assigns the result to children nodes if a parent node has been re-assigned the result (lines $2 \sim 3$). The *assignResult* subroutine is similar to the *assignResults* subroutine described in Algorithm 5, with two exceptions: it only assigns one result and it does not recursively assign the result to children nodes.

Algorithm 9 *annotateResult(n, $\mathcal{T}$, $\mathcal{P}$, r)*

Input: n, $\mathcal{T}$, $\mathcal{P}$, r, M, *assign*: n is a node. $\mathcal{T}$ is the set of changed terms. $\mathcal{P}$ is the set of changed phrases. r is the result to be assigned. M is a dictionary correlating terms with results containing them. *assign* is a *boolean* noting whether the parent node re-assigned results.

Output: n': the node n after result r has been re-assigned to it and its children.

1: $n' \leftarrow n$;
2: **if** $assign = TRUE$ **then**
3: $n' \leftarrow assignResult(n, r, M)$;
4: **else**
5: $A \leftarrow getAnnotation(n)$;
6: **for each** $t \in \mathcal{T}$
7: **if** $contains(A, t)$ **then**
8: $assign \leftarrow TRUE$;
9: **end if**
10: **end for**
11: **for each** $p \in \mathcal{P}$
12: **if** $contains(A, p)$ **then**
13: $assign \leftarrow TRUE$;
14: **end if**
15: **end for**
16: $N \leftarrow children(n')$;
17. **if** $assign = TRUE$ **then**
18: $n' \leftarrow assignResult(n, r, M)$;
19: **end if**
20: **for each** $n_c \in N$
21: $n_c \leftarrow annotateResult(n_c, \mathcal{T}, \mathcal{P}, r, M, assign)$;
22: **end for**
23: **end if**
24: return n';

After checking the *membership condition*, the subroutine may either add or remove the given result from the cluster node in consideration.

If the parent node has not been re-assigned the result, ClusteringWiki2 checks whether the node's annotation contains any of the terms in $\mathcal{T}$ (lines $6 \sim 10$) or phrases in $\mathcal{P}$ (lines $11 \sim 15$). If it does, the result is assigned to the current node (line 18). Finally, the *assignResult* call is propagated to children nodes in lines $20 \sim 22$.

Editing interface. Cluster editing in ClusteringWiki2, like in ClusteringWiki, is primarily available through context menus attached to label and result nodes. However, given its alternative editing methodology, ClusteringWiki2 offers fewer operations than ClusteringWiki. Cluster nodes will show a menu containing *Create child label*, *Change label*, and *Delete label*. Results will only show *Edit result annotation*. Additionally, ClusteringWiki2 does not include *drag and drop* or *copy/paste* functionality.

6.3 Cluster Aggregation Discussion

Currently ClusteringWiki and ClusteringWiki2 aggregate personalized clusters to form a collaborative (community) cluster tree by applying to T_{init} those significant personalized *paths*. Significant paths are simply defined by paths that have been personalized by a minimum threshold of users. Two ways this model could be

improved in future work are described below:

Mass collaboration optimization. As this point applies to both ClusteringWiki and ClusteringWiki2, I will use ClusteringWiki to reference both systems in the following. The goal of ClusteringWiki is to present a search result cluster that will aid the user in reaching their results faster. When the user is logged into the system, this goal is achieved by allowing the user to personalize the cluster tree. However, users will often personalize portions of the cluster tree based on their current search intent interest. Additionally, users will not always agree on what clusters should exist or what results they should contain. Thus, aggregating personalized clusters to create a collaborative clustering is a hard problem.

The goal of ClusteringWiki when a user is not logged in to the system does not change. Therefore, assuming a personalized clustering would best achieve the goal, we wish to present a non-logged in user with a clustering that is as close as possible to their *imaginary personalized clustering*, the cluster tree they would create if they did log in and edit the search result cluster tree. Borrowing some techniques from *collaborative search*, this problem can be solved by formulating it as an optimization problem. We wish to find a set $U_s \subseteq U$ of other known users *similar* enough to the current user and who have edited the cluster tree for the current query. Once we do, applying their edits would be more meaningful to the current user than applying edits from all users.

Since the current user is unknown (not logged in) and may not have edited any cluster tree, we cannot rely on cluster edits as a means of finding similar users. One way this problem could be solved is to store user profiles created from a user's search history and click-through information. We can then create an objective function that minimizes a user profile similarity score to identify U_s from those user profiles of known users who have edited the cluster tree in question. Note that $u \ni U_s$ and $|U_s| > 1$, where u is the current user. Allowing $|U_s| = 1$ would make our *similar user* selection open to noise.

Path to result annotation reference. In both ClusteringWiki and ClusteringWiki2 there is an inherent relation between the search result clusters and the results themselves. ClusteringWiki strongly correlates the two. Path edits in ClusteringWiki affect both a cluster node and its membership. ClusteringWiki2 employs a looser strategy which includes two types of edits: cluster (path) edits, and search result annotation edits. Currently, ClusteringWiki2 treats aggregation of the two types of edits independently. However, users will often make both cluster and annotation edits to achieve a personalized cluster goal. Ideally, we wish to choose those annotation edits which are strongly associated with those chosen cluster edits. One way this could be achieved is by associating cluster and annotation edits made in an editing session with the same session id. Then, significant annotation edits (made by the same minimum threshold of users as the cluster edits) could be chosen

from the subset of annotations made in one of the chosen editing sessions.

There are many other 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, a user's social network can be utilized in preference aggregation. Another interesting direction is to seamlessly integrate personalization of search result ranking [Gao and Jan, 2010] with that of search result clustering, providing a more complete solution for personalized and collaborative information retrieval and Web search.

CHAPTER 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 thesis I 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.

Both personalized and collaborative clustering of search results aid users in locating those search results they seek. Personalized clustering saves user effort by allowing the user to place results in familiar clusters. Aggregated clustering also provides significant benefits and is "free," in the sense that it does not take user editing effort.

As an alternate method of personalized and collaborative clustering of search

results, I presented ClusteringWiki2, a cluster editing system based on annotations. With complete control over both positive and negative terms and phrases in annotations, users can have the same editing freedom as in ClusteringWiki, while maintaining collaborative transparency.

BIBLIOGRAPHY

Agrawal, R., Gollapudi, S., Halverson, A., and Ieong, S. (2009). Diversifying search results. In *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, WSDM '09, pages 5–14, New York, NY, USA. ACM.

Amershi, S. and Morris, M. R. (2008). Cosearch: a system for co-located collaborative web search. In *Proceedings of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1647–1656, New York, NY, USA. ACM.

Anastasiu, D. C., Buttler, D., and Gao, B. J. (2010). Clusteringwiki technical report. *dmlab.cs.txstate.edu/ClusteringWiki/pdf/cw.pdf*.

Anastasiu, D. C., Buttler, D., and Gao, B. J. (2011). Clusteringwiki: Personalized and collaborative clustering of search results. In *Proceeding of the 34th international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '11, New York, NY, USA. ACM.

Balcan, M.-F. and Blum, A. (2008). Clustering with interactive feedback. In *Proceedings of the 19th international conference on Algorithmic Learning Theory*, ALT '08, pages 316–328, Berlin, Heidelberg. Springer-Verlag.

Basu, S., Bilenko, M., and Mooney, R. J. (2004). A probabilistic framework for semi-supervised clustering. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 59–68, New York, NY, USA. ACM.

Bekkerman, R., Raghavan, H., Allan, J., and Eguchi, K. (2007). Interactive clustering of text collections according to a user-specified criterion. In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 684–689, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Bernardo, T. (2007). Employing mass collaboration information technologies to protect human lives and to reduce mass destruction of animals. *Veterinaria Italiana*, 2(43):273–284.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Broder, A. (2002). A taxonomy of web search. In *SIGIR Forum*.

Carbonell, J. and Goldstein, J. (1998). The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 335–336, New York, NY, USA. ACM.

Carpineto, C., Osiński, S., Romano, G., and Weiss, D. (2009). A survey of web clustering engines. *ACM Computing Surveys (CSUR)*, 41(3):1–38.

Carroll, J. M. and Rosson, M. B. (1987). *Paradox of the active user*, pages 80–111. MIT Press, Cambridge, MA, USA.

Carterette, B. and Chandar, P. (2009). Probabilistic models of ranking novel documents for faceted topic retrieval. In *Proceeding of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 1287–1296, New York, NY, USA. ACM.

Chaffee, J. and Gauch, S. (2000). Personal ontologies for web navigation. In *Proceedings of the ninth international conference on Information and knowledge management*, CIKM '00, pages 227–234, New York, NY, USA. ACM.

Chen, H. and Dumais, S. (2000). Bringing order to the web: automatically categorizing search results. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '00, pages 145–152, New York, NY, USA. ACM.

Chirita, P. A., Firan, C. S., and Nejdl, W. (2007). Personalized query expansion for the web. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 7–14, New York, NY, USA. ACM.

Chirita, P. A., Nejdl, W., Paiu, R., and Kohlschütter, C. (2005). Using odp metadata to personalize search. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 178–185, New York, NY, USA. ACM.

Clarke, C. L., Kolla, M., Cormack, G. V., Vechtomova, O., Ashkan, A., Büttcher, S., and MacKinnon, I. (2008). Novelty and diversity in information retrieval evaluation. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 659–666, New York, NY, USA. ACM.

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

Dalal, M. (2007). Personalized social & real-time collaborative search. *Proceedings of the 16th international conference on World Wide Web WWW 07*, page 1285.

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

Dou, Z., Song, R., Wen, J.-R., and Yuan, X. (2009). Evaluating the effectiveness of personalized web search. *Knowledge and Data Engineering, IEEE Transactions on*, 21(8):1178 –1190.

Drosou, M. and Pitoura, E. (2010). Search result diversification. *ACM SIGMOD Record*, 39:41–47.

Dumais, S., Cutrell, E., and Chen, H. (2001). Optimizing search by showing results in context. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '01, pages 277–284, New York, NY, USA. ACM.

Evans, B. M. and Chi, E. H. (2008). Towards a model of understanding social search. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, CSCW '08, pages 485–494, New York, NY, USA. ACM.

Everitt, B. S., Landau, S., and Leese, M. (2001). *Cluster analysis.* Oxford University Press.

Fern, X. Z. and Lin, W. (2008). Cluster ensemble selection. *Statistical Analysis and Data Mining*, 1(3):128–141.

Gao, B. J. and Jan, J. (2010). Rants: a framework for rank editing and sharing in web search. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 1245–1248, New York, NY, USA. ACM.

Gauch, S., Chaffee, J., and Pretschner, A. (2003). Ontology-based personalized search and browsing. *Web Intelligence and Agent Systems*, 1:219–234.

Gionis, A., Mannila, H., and Tsaparas, P. (2007). Clustering aggregation. *ACM Transactions on Knowledge Discovery from Data*, 1(1):4–es.

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

Halpin, H., Robu, V., and Shepherd, H. (2007). The complex dynamics of collaborative tagging. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 211–220, New York, NY, USA. ACM.

Haveliwala, T. H. (2002). Topic-sensitive pagerank. In *Proceedings of the 11th international conference on World Wide Web*, WWW '02, pages 517–526, New York, NY, USA. ACM.

Hearst, M. A. and Pedersen, J. O. (1996). Reexamining the cluster hypothesis: scatter/gather on retrieval results. In *Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '96, pages 76–84, New York, NY, USA. ACM.

Horrocks, I. (2008). Ontologies and the semantic web. *Communications of the ACM*, 51(12):58.

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

Jain, A. K. and Dubes, R. C. (1988). *Algorithms for clustering data.* Prentice-Hall.

Jansen, B. J., Spink, A., and Saracevic, T. (2000). Real life, real users, and real needs: a study and analysis of user queries on the web. *Information Processing and Management: an International Journal*, 36:207–227.

Jeh, G. and Widom, J. (2003). Scaling personalized web search. In *Proceedings of the 12th international conference on World Wide Web*, WWW '03, pages 271–279, New York, NY, USA. ACM.

Ji, X. and Xu, W. (2006). Document clustering with prior knowledge. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 405–412, New York, NY, USA. ACM.

Joachims, T., Granka, L., Pan, B., Hembrooke, H., and Gay, G. (2005). Accurately interpreting clickthrough data as implicit feedback. In *Proceeding of the 28th international ACM SIGIR conference on Research and development in information retrieval (SIGIR).*

Käki, M. (2005). Findex: search result categories help users when document ranking fails. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '05, pages 131–140, New York, NY, USA. ACM.

Kale, A., Burris, T., Shah, B., Venkatesan, T. L. P., Velusamy, L., Gupta, M., and Degerattu, M. (2010). icollaborate: harvesting value from enterprise web usage. In *Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '10, pages 699–699, New York, NY, USA. ACM.

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

Kelly, D. and Teevan, J. (2003). Implicit feedback for inferring user preference: a bibliography. *ACM SIGIR Forum*, 37(2):18–28.

Koren, J., Zhang, Y., and Liu, X. (2008). Personalized interactive faceted search. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 477–486, New York, NY, USA. ACM.

Krovetz, R. and Croft, W. B. (1992). Lexical ambiguity and information retrieval. *ACM Transactions on Information Systems (TOIS)*, 10:115–141.

Kummamuru, K., Lotlikar, R., Roy, S., Singal, K., and Krishnapuram, R. (2004). A hierarchical monothetic document clustering algorithm for summarization and browsing search results. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 658–665, New York, NY, USA. ACM.

Lawrence, S. (2000). Context in web search. *IEEE Data Engineering Bulletin*, 23(3):25–32.

Lee, J., Hwang, S.-w., Nie, Z., and Wen, J.-R. (2009). Query result clustering for object-level search. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '09, pages 1205–1214, New York, NY, USA. ACM.

Li, Z., Li, T., and Ding, C. (2010). Hierarchical ensemble clustering. In *Proceeding of the IEEE 10th International Conference on Data Mining (ICDM)*, volume 1, pages 1–6. IEEE.

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

Manning, C. D., Raghavan, P., and Schtze, H. (2008). *Introduction to Information Retrieval.* Cambridge University Press.

Morris, M. R. (2008). A survey of collaborative web search practices. In *Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, CHI '08, pages 1657–1660, New York, NY, USA. ACM.

Morris, M. R. and Horvitz, E. (2007). Searchtogether: an interface for collaborative web search. In *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, pages 3–12, New York, NY, USA. ACM.

O'Reilly, T. and Battelle, J. (2009). Web squared: Web 2.0 five years on. *assets.en.oreilly.com/1/event/28/web2009_websquared-whitepaper.pdf*.

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

Pirolli, P., Schank, P., Hearst, M., and Diehl, C. (1996). Scatter/gather browsing communicates the topic structure of a very large text collection. In *Proceedings of the SIGCHI conference on Human factors in computing systems: common ground*, CHI '96, pages 213–220, New York, NY, USA. ACM.

Pretschner, A. and Gauch, S. (1999). Ontology based personalized search. In *Proceedings of the 11th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '99, pages 391–, Washington, DC, USA. IEEE Computer Society.

Qiu, F. and Cho, J. (2006). Automatic identification of user interest for personalized search. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 727–736, New York, NY, USA. ACM.

Radlinski, F. and Dumais, S. (2006). Improving personalized web search using result diversification. In *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '06, pages 691–692, New York, NY, USA. ACM.

Rafiei, D., Bharat, K., and Shukla, A. (2010). Diversifying web search results. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 781–790, New York, NY, USA. ACM.

Raghavan, H., Madani, O., and Jones, R. (2005). Interactive feature selection. In *Proceedings of the 19th international joint conference on Artificial intelligence*, pages 841–846, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Rijsbergen, C. J. V. (1979). *Information Retrieval.* Butterworth-Heinemann, Newton, MA, USA.

Rose, D. E. and Levinson, D. (2004). Understanding user goals in web search. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 13–19, New York, NY, USA. ACM.

Ruthven, I. and Lalmas, M. (2003). A survey on the use of relevance feedback for information access systems. *Knowledge Engineering Review*, 18(1).

Salton, G. (1971). *The SMART Retrieval System.* Prentice-Hall.

Salton, G., Wong, A., and Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18:613–620.

Sandhaus, E. (2008). The New York Times Annotated Corpus. *Linguistic Data Consortium, Philadelphia.*

Santos, R. L., Macdonald, C., and Ounis, I. (2010). Exploiting query reformulations for web search result diversification. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 881–890, New York, NY, USA. ACM.

Sarlós, T., Benczúr, A. A., Csalogány, K., Fogaras, D., and Rácz, B. (2006). To randomize or not to randomize: space optimal summaries for hyperlink analysis. In *Proceedings of the 15th international conference on World Wide Web*, WWW '06, pages 297–306, New York, NY, USA. ACM.

Sigurbjörnsson, B. and van Zwol, R. (2008). Flickr tag recommendation based on collective knowledge. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 327–336, New York, NY, USA. ACM.

Singh, V., Mukherjee, L., Peng, J., and Xu, J. (2008). Ensemble clustering using semidefinite programming. *Advances in Neural Information Processing Systems 20*, 79(1-2):177–200.

Song, Y., Zhuang, Z., Li, H., Zhao, Q., Li, J., Lee, W.-C., and Giles, C. L. (2008). Real-time automatic tag recommendation. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 515–522, New York, NY, USA. ACM.

Speretta, M. and Gauch, S. (2005). Personalized search based on user search histories. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*, pages 622 – 628.

Sugiyama, K., Hatano, K., and Yoshikawa, M. (2004). Adaptive web search based on user profile constructed without any effort from users. In *Proceedings of the 13th international conference on World Wide Web*, WWW '04, pages 675–684, New York, NY, USA. ACM.

Tan, B., Shen, X., and Zhai, C. (2006). Mining long-term search history to improve search accuracy. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '06, pages 718–723, New York, NY, USA. ACM.

Tan, P.-N., Steinbach, M., and Kumar, V. (2005). *Introduction to Data Mining.* Addison-Wesley.

tao Sun, J., Zeng, H.-J., Liu, H., and Lu, Y. (2005). Cubesvd: A novel approach to personalized web search. In *Proceedings of the 14th International World Wide Web Conference (WWW)*, pages 382–390. Press.

Teevan, J., Dumais, S. T., and Horvitz, E. (2005). Personalizing search via automated analysis of interests and activities. In *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '05, pages 449–456, New York, NY, USA. ACM.

Tombros, A., Villa, R., and Van Rijsbergen, C. J. (2002). The effectiveness of query-specific hierarchic clustering in information retrieval. *Information Processing and Management: an International Journal*, 38(4):559–582.

Twidale, M. B., Nichols, D. M., and Paice, C. D. (1997). Browsing is a collaborative process. *Information Processing and Management: an International Journal*, 33:761–783.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14:249–260.

von Ahn, L. and Dabbish, L. (2004). Labeling images with a computer game. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, CHI '04, pages 319–326, New York, NY, USA. ACM.

Wagstaff, K., Cardie, C., Rogers, S., and Schrödl, S. (2001). Constrained k-means clustering with background knowledge. In *Proceedings of the Eighteenth International Conference on Machine Learning*, ICML '01, pages 577–584, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Wang, X. and Zhai, C. (2007). Learn from web search logs to organize search results. In *Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '07, pages 87–94, New York, NY, USA. ACM.

Wen, J.-R., Duo, Z., and Song, R. (2009). Personalized web search. *Encyclopedia of Database Systems.*

Wilson, T. D. (2006). On user studies and information needs. *Journal of Documentation*, 62(6):658–670.

Xu, S., Bao, S., Fei, B., Su, Z., and Yu, Y. (2008). Exploring folksonomy for personalized search. In *Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '08, pages 155–162, New York, NY, USA. ACM.

Xue, G.-R., Han, J., Yu, Y., and Yang, Q. (2009). User language model for collaborative personalized search. *ACM Transactions on Information Systems (TOIS)*, 27:11:1–11:28.

Zamir, O. and Etzioni, O. (1998). Web document clustering: a feasibility demonstration. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '98, pages 46–54, New York, NY, USA. ACM.

Zamir, O. and Etzioni, O. (1999). Grouper: a dynamic clustering interface to web search results. In *Proceedings of the eighth international conference on World Wide Web*, WWW '99, pages 1361–1374, New York, NY, USA. Elsevier North-Holland, Inc.

Zeng, H.-J., He, Q.-C., Chen, Z., Ma, W.-Y., and Ma, J. (2004). Learning to cluster web search results. In *Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval*, SIGIR '04, pages 210–217, New York, NY, USA. ACM.

Zhai, C. X., Cohen, W. W., and Lafferty, J. (2003). Beyond independent relevance: methods and evaluation metrics for subtopic retrieval. In *Proceedings of the 26th annual international ACM SIGIR conference on Research and development in informaion retrieval*, SIGIR '03, pages 10–17, New York, NY, USA. ACM.

Zhang, M. and Hurley, N. (2008). Avoiding monotony: improving the diversity of recommendation lists. In *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, pages 123–130, New York, NY, USA. ACM.

Zhao, Y. and Karypis, G. (2002). Evaluation of hierarchical clustering algorithms for document datasets. In *Proceedings of the eleventh international conference on Information and knowledge management*, CIKM '02, pages 515–524, New York, NY, USA. ACM.

Zimmerman, M. (2000). Weaving the web: the original design and ultimate destiny of the world wide web by its inventor [book review]. *IEEE Transactions on Professional Communication*, 43(2):217 –218.

Zollers, A. (2007). Emerging motivations for tagging: Expression, performance, and activism. In *Tagging and Metadata for Social Information Organization Workshop*, WWW '07, New York, NY, USA. ACM.

VITA

Dragos Anastasiu was born in Bucharest, Romania, on February 13, 1979, the son of Mariana and Miron Anastasiu. He came to the United States to pursue a Bachelor Degree in Theology. After completing his work at Moody Bible Institute in Chicago, Illinois, he worked in the Information Technology industry for a number of years. In Summer 2008 he entered Texas State University-San Marcos. In the Spring of 2009, he received a post-graduate Certificate in Computer Science from Texas State University-San Marcos. He continued on at Texas State University-San Marcos, pursuing a Masters Degree in Computer Science.

Permanent Address: 500 Keystone Loop
Kyle, Texas 78640

This thesis was typed by Dragos Anastasiu.