# Rants: A Framework for Rank Editing and Sharing in Web Search

Byron J. Gao
Texas State University-San Marcos
601 University Drive
San Marcos, TX, USA, 78666
bgao@txstate.edu

Joey Jan
Texas State University-San Marcos
601 University Drive
San Marcos, TX, USA, 78666
jj1258@txstate.edu

## ABSTRACT

With a Wiki-like search interface, users can edit ranks of search results and share the edits with the rest of the world. This is an effective way of personalization, as well as a practice of mass collaboration that allows users to vote for ranking and improve search performance. Currently, there are several ongoing experimentation efforts from the industry, e.g., SearchWiki by Google and U Rank by Microsoft. Beyond that, there is little published research on this new search paradigm. In this paper, we make an effort to establish a framework for rank editing and sharing in the context of web search, where we identify fundamental issues and propose principled solutions. Comparing to existing systems, for rank editing, our framework allows users to specify not only relative, but also absolute preferences. For edit sharing, our framework provides enhanced flexibility, allowing users to select arbitrarily aggregated views. In addition, edits can be shared among similar queries. We present a prototype system `Rants`, that implements the framework and provides search services through the Google web search API.

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

**General Terms:** Performance, Design, Algorithms

**Keywords:** Rank editing, Edit sharing, Search interface, Personalization, Mass collaboration, Social search

## 1. INTRODUCTION

Recently, several web search giants are experimenting on a new Wiki-like search interface, where users can edit ranks of search results directly. For example, SearchWiki [3] by Google and U Rank [6] by Microsoft. This new search paradigm is an effective way of search personalization. It is also a practice of mass collaboration at a world-wide scale that allows users to vote for ranking of search results and improve search performance.

SearchWiki and U Rank are under testing, often returning inconsistent or unintuitive results. It is not revealed what exactly they aim to achieve and what the approaches are. Up to our knowledge, there is no published research under this topic. Thus this paper makes the first effort to establish a framework by identifying the fundamental issues and proposing principled solutions for rank editing and sharing.


*WWW 2010,* April 26–30, 2010, Raleigh, North Carolina, USA.
ACM 978-1-60558-799-8/10/04.

The proposed framework features extended functionalities beyond SearchWiki and U Rank. For rank editing, users can specify not only relative, but also absolute preferences. For edit sharing, the notion of aggregation is generalized and users can select arbitrarily aggregated views. Beyond sharing among users, edits can be transferred to similar queries, which can be considered sharing among queries.

These extensions generate benefits, as well as non-trivial technical complications. We deployed a prototype system `Rants` [7], that tackles the challenges and implements the framework, providing web search services through the Google API [5]. Figure 1 illustrates the interface of `Rants`.

**Rank editing**. Existing systems provide two editing operations, *promotion* and *demotion*, where a logged-in user $u$ can promote (move up) or demote (move down) a search result $r$ for a query $q$ by one or more positions. Let $up$ and $down$ denote the two operations, where $up(r, 2)$ means to move $r$ up by 2 positions if possible (it may reach the top and cannot continue).

*How to interpret a move?* The user intention behind a move is unfortunately ambiguous. Aggressively, it may mean an assertion for the ranking of all results after the move. Conservatively, it may mean several pairwise preferences for the involving results only. In our framework, we take a conservative approach and make the least inferences from a move. For example, if after $up(r, 2)$ by user $u$ for query $q$, $r$ surpassed $r'$ and $r''$, we store two pairs $(r, r')$ and $(r, r'')$, meaning that for $q$, user $u$ prefers $r$ to appear before $r'$ and $r''$.

We adopt this *least inference principle* for the following reasons. Firstly, it generates the least, if not none, ambiguity. In the above example, although by the same move different users may mean differently, all of them would mean at least those two pairwise preferences. We do not even infer on the precedence relationship between $r'$ and $r''$. Secondly, it well serves the purpose that, given the same list of unedited search results for query $q$, the same ranking as edited will be restored if the inferred pairwise preferences are respected.

Based on this interpretation, $up(r, 2)$ is equivalent to two consecutive executions of $up(r, 1)$. Thus, in `Rants`, we only allow $up(r)$ and $down(r)$, meaning $up(r, 1)$ and $down(r, 1)$, indicated by the ↑ and ↓ arrows in Figure 1. This is not a limitation, but an emphasis on primitive functionalities, instead of syntactic sugars, for conceptual clarity.

The two operations actually map to the same task, a swap of positions of two results. Let $r'$ and $r$ be two neighboring results where $r'$ is immediately before $r$. Both $up(r)$ and $down(r')$ fire a swap of $r'$ and $r$, specifying a preference of

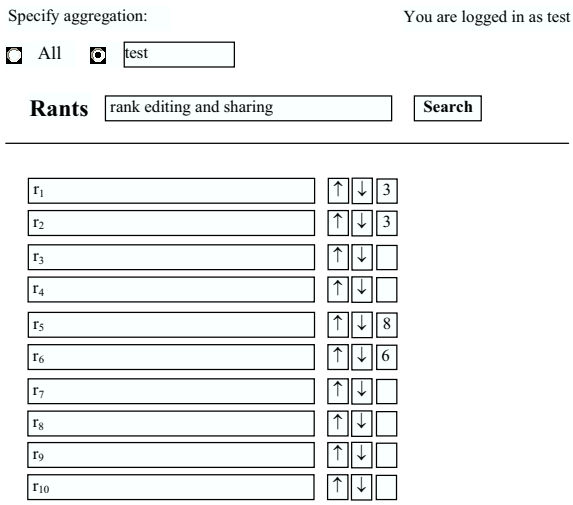


**Figure 1: Interface of `Rants`.**

$(r, r')$. In `Rants`, this task is implemented by a primitive function $swap(r', r)$.

**Extended rank editing**. Promotion and demotion specify *relative* preferences. There are often situations where users want to specify *absolute* preferences as well. For example, user $u$ may always want to see result $r$ appear among top 3 for query $q$. This cannot be achieved by relative preferences because over time, there would be new results for $q$ taking the top 3 seats whose pairwise preferences w.r.t. $r$ were not specified and stored.

Mostly, user $u$ wants to stipulate that result $r$ must appear among top $k$, instead of not. We use a pair $(r, k)$ to capture this absolute preference. As shown in Figure 1, for each $q$, $k$ can be entered into the box to the right of the $\downarrow$ arrow. In `Rants`, this operation is implemented by a primitive function $anchor(r, k)$.

**Edit sharing**. Rank edits are user preferences that can be aggregated and shared among users. `Rants` utilizes function $aggr(U)$ to generalize the notion of aggregation. No matter logged in or out, user $u$ can arbitrarily specify $U$, the set of users whose edits (relative or absolute) are to be used for aggregation. If $U = \{u\}$, the edits from $u$ herself will be used. If $U = All$, the (published) edits from all users will be used. If $U = \emptyset$, the original, unedited search results will be presented without enforcing any stored preferences.

Suppose a same edit is specified by a set $U' \subseteq U$ of users. Then the edit can be shared if $\frac{|U'|}{|U|} \geq \delta_{ag}$, where $0 \leq \delta_{ag} \leq 1$ is a tunable threshold.

**Edit transfer**. Rank editing takes user effort. It is greatly beneficial if we can properly transfer rank edits from a query to its similar ones. For example, if user $u$ edited the results for query "David Dewitt", it is very likely that she wants to reuse the edits for query "David J. Dewitt". Edit transfer can be considered as edit *sharing among queries*, in contrast to *sharing among users*.

Function $trans(q)$ returns a query $q'$ such that it is appropriate to transfer edits for query $q'$ to query $q$. In the function, two similarity measures are used. $wordSim(q, q')$ compares the keywords of $q$ and $q'$. $rankSim(q, q')$ compares the ranks of search results of $q$ and $q'$. Both need to pass their respective thresholds $\delta_{ws}$ and $\delta_{rs}$. Obviously, the bigger the thresholds, the more conservative the transfer. Setting the thresholds to 1 shuts down edit transfer.

**Constraint enforcement**. We take stored rank edits as user constraints that need to be respected and enforced in processing query $q$. In `Rants`, this is implemented by functions $enfor(R)$ and $enfor(A)$, which enforce a set of relative preferences $R$ and a set of absolute preferences $A$. $R$ and $A$ are determined by $q$ and a selected aggregation $U$.

The enforcement adopts a so-called *least modification principle*, where as little as possible modification is used in enforcing the constraints, and the degree of modification is measured by edit distance between the rankings of search results before and after the enforcement.

In `Rants`, consistency of relative preferences are maintained and they are guaranteed to be completely enforced. To do so for absolute preferences entails tremendous technical complications that significantly slow down rank editing and query processing for a minimal gain. Practically, a light-weight *best-effort* approach can achieve complete enforcement of absolute preferences in normal cases.

**Technical challenges and contributions**. In summary, in this introductory study we consider the following fundamental issues. How to interpret, capture and store user preferences? How to keep them consistent and redundancy-free? How to aggregate them for sharing among users? How to transfer them to similar queries? How to enforce them in processing queries?

In response to these questions, we construct the following functional primitives: $swap(r', r)$, $anchor(r, k)$, $aggr(U)$, $trans(q)$, $enfor(R)$ and $enfor(A)$. Their correct and efficient realization generates non-trivial technical challenges.

Thus, our contributions include the identification of fundamental issues involved in the Wiki-like web search paradigm, the formation of the corresponding functional primitives, the provision of solutions to the associated technical challenges, and the implementation of these solutions.

## 2. RELATED WORK

We have not seen published research on rank editing and sharing. However, web search giants Google and Microsoft are recently experimenting on this novel search paradigm through SearchWiki [3] and U Rank [6] respectively. While their approaches are not revealed to public, we use `Rants` to demonstrate a well-defined framework that features extended functionalities.

Rank editing can be considered as one way of search personalization. Personalized search allows fine-tuning of search results based on an individual's preferences or profile. Both Google [2] and Yahoo! [8] provide such services. Traditionally, the major source for personalization is search history, which forms a user profile and can be used to influence all queries from the user. In `Rants`, ranks of search results can be directly edited, and the edits can be used to influence limited queries, i.e., the query itself and its similar ones.

Edit sharing is a mass-collaboration way of improving search performance. It is also related to social search. In contrast to established algorithmic or machine-based approaches, social search determines the relevance of search results by considering the interactions or contributions of users. Example social search engines include Google social search [4] and "community powered" Eurekster Swiki [1].

# 3. ALGORITHM

In this section, we explain the primitive functions that are used in rank editing, edit sharing, and query processing.

## 3.1 Rank Editing

Once a user $u$ is logged in, she can edit the search results for a query $q$, specifying relative and/or absolute preferences. Since she needs to monitor her own editing process, the view must be chosen as $U = \{u\}$.

Let $RP$ and $AP$ be hashes storing all the relative and absolute preferences from $u$ respectively. Then $RP(q)$ and $AP(q)$ indicate a set of relative preferences and a set of absolute preferences for $q$ from $u$. For clarity we do not specify $u$ in the notations.

Let $L(q) = (r_1, r_2, ...)$ be the list of search results for query $q$. For a result $r \in L(q)$, we use $rank(r)$ to denote the rank of $r$ in $L(q)$. Let $L_0(q)$ be the list of unedited original search results. $L_0(q)$ may change over time. Some previous results may disappear. Some new ones maybe added. The content and ranking of results may change as well.

***swap(r', r)***. The function handles specification of a relative preference $(r, r')$, i.e., $rank(r) < rank(r')$, for query $q$. It is fired either by $up(r)$ or $down(r')$, resulting in a swap of positions of $r'$ and $r$ in the search results.

Relative preferences are transitive. E.g., with $(r_1, r_2)$ and $(r_2, r_3)$, we can infer $(r_1, r_3)$. All the preference pairs in $RP(q)$, if consistent, form a partial order. Precisely, it is a *strict partial order*, a binary relation that is irreflexive and transitive, corresponding to a directed acyclic graph (dag).

*Example 1.* Let $L_0(q) = (r_1, r_2, r_3, r_4)$. By $swap(r_1, r_2)$ and $swap(r_3, r_4)$, the user specify $(r_2, r_1)$ and $(r_4, r_3)$, which form a partial order. We do not infer the pairwise preferences for, e.g., $r_1$ and $r_4$. Thus, both $(r_2, r_1, r_4, r_3)$ and $(r_4, r_3, r_2, r_1)$ respect the specified preferences.

Due to the dynamic nature of search results and user preferences, $RP(q)$ may receive inconsistent, conflicting preferences that cannot be enforced simultaneously.

*Example 2.* At day 1, $L_0(q) = (r_1, r_2)$. $swap(r_1, r_2)$ adds $(r_2, r_1)$ to $RP(q)$, which will be enforced in query processing. At day 2, the user changed her mind, and $swap(r_2, r_1)$ would add $(r_1, r_2)$ to $RP(q)$, which contradicts with $(r_2, r_1)$.

Only consistent user preferences can be completely enforced. Thus it is essential to maintain the consistency of $RP(q)$. It is also desirable to keep $RP(q)$ redundancy-free for improved enforcement efficiency. Addition of a pair may generate new inferred preferences that are redundant to existing ones, as demonstrated in the following example.

*Example 3.* Let $RP(q) = \{(r_1, r_3), (r_2, r_3)\}$. $swap(r_2, r_1)$ adds $(r_1, r_2)$ to $RP(q)$. Then, $(r_1, r_3)$ becomes redundant because it can be inferred by $(r_1, r_2)$ and $(r_2, r_3)$.

$swap(r', r)$ maintains $RP(q)$ as a redundancy-free dag. We omit the algorithmic details due to the space limit. The enforcement of the newly added pair $(r, r')$ is trivial.

***anchor(r, k)***. The function handles specification of an absolute preference $(r, k)$, i.e., $rank(r) \leq k$, for query $q$. As consistency of $AP(q)$ is not maintained, adding $(r, k)$ is trivial. If $(r, k)$ is already in $AP(q)$, update it.

Inconsistency of $AP(q)$ arises in various cases. For example, $(r, 1) \in AP(q)$ and $(r', 1) \in AP(q)$. $AP(q)$ may not be compatible with $RP(q)$ either. For example, $(r, 1) \in AP(q)$ and $(r', r) \in RP(q)$. However, since $AP(q)$ is small, such anomalies would not arise in normal cases. Thus to avoid the tremendous technical complications, we do not maintain consistency and compatibility of $AP(q)$.

After the specification of $(r, k)$, we want to enforce it immediately. For this purpose, we call a recursive procedure $climb(r)$, which is introduced in Section 3.3.

## 3.2 Edit Sharing

User edits can be shared among users, as well as among similar queries.

***aggr(U)***. The function performs aggregation of edits for a chosen user set of $U$, returning $RP_U$ and $AP_U$, the aggregated relative and absolute preferences over $U$. Although $U$ can be arbitrarily specified, it must be pre-defined so that $RP_U$ and $AP_U$ can be pre-computed off-line, instead of during query processing, for improved response time.

We calculate $RP_U$ as follows. For each query $q$ such that $RP(q) \neq \emptyset$ for some $u \in U$, for each pair $(r, r')$ in $RP(q)$, let $U' \subseteq U$ be the set of users who specified the pair. If $\frac{|U'|}{|U|} \geq \delta_{ag}$, insert $(r, r')$ into $RP_U(q)$, where $RP_U(q)$ stores the set of aggregated relative preferences over $U$ for query $q$. Recall that $\delta_{ag}$, $0 \leq \delta_{ag} \leq 1$, is a tunable threshold.

We calculate $AP_U$ in a similar manner with slight modification. For each query $q$ such that $AP(q) \neq \emptyset$ for some $u \in U$, for each pair $(r, k)$ specifying a preference on $r$ in $AP(q)$, let $U' \subseteq U$ be the set of users who specified the pair. If $\frac{|U'|}{|U|} \geq \delta_{ag}$, insert $(r, \overline{k})$ into $AP_U(q)$, where $AP_U(q)$ stores the set of aggregated absolute preferences over $U$ for query $q$, and $(\overline{k})$ is the averaged $k$ specifications for $r$ over $U'$.

The consistency of $RP$ implies the consistency of $RP_U$. As in $AP$, the consistency of $AP_U$ is not maintained.

***trans(q)***. The function returns a query $q'$, such that the edits (w.r.t. a chosen aggregation $U$) for query $q'$ can be effectively utilized by query $q$. If none of such $q'$ can be found, the function returns -1, which means no user constraints will be enforced in processing query $q$.

A candidate query $q'$ must have an entry stored in $RP_U$ or $AP_U$. If $q$ itself is such a candidate, then $q$ will be returned. Otherwise, $trans(q)$ searches for some $q'$ that is similar enough to $q$.

As a candidate, query $q'$ must also have the properties of $wordSim(q, q') \geq \delta_{ws}$ and $rankSim(q, q') \geq \delta_{rs}$, where $\delta_{ws}$ and $\delta_{rs}$ are tunable thresholds. $wordSim(q, q')$ is a similarity measure comparing the keywords of $q$ and $q'$. In $trans(q)$, this comparison is done first to eliminate most of the unqualified candidates. $rankSim(q, q')$ is a similarity measure comparing the ranks of search results of $q$ and $q'$, in particular, $L_{10}(q)$ and $L_{10}(q')$, the top 10 unedited results for $q$ and $q'$ respectively. In the end, $trans(q)$ returns a qualified candidate $q'$ with the largest $rankSim(q, q')$.

For computing $wordSim(q, q')$, we treat $q$ and $q'$ as sets of keywords and use $J(q, q')$, the Jaccard index for $q$ and $q'$, to measure their similarity. Specifically,

$$J(q, q') = \frac{|q \cap q'|}{|q \cup q'|}.$$

For computing $rankSim(q, q')$, we have two options. The

first option is $J(L_{10}(q), L_{10}(q'))$, i.e., the Jaccard index for $L_{10}(q)$ and $L_{10}(q')$. The second option is a rank-aware similarity measure, the Kendall tau coefficient [9], a non-parametric statistic used to measure the degree of correspondence between two rankings. Specifically,

$$\tau(L_{10}(q), L_{10}(q')) = \frac{n_c - n_d}{\frac{1}{2}n(n-1)},$$

where $n_c$ is the number of concordant pairs between $L_{10}(q)$ and $L_{10}(q')$, and $n_d$ is the number of disconcordant pairs. In our case, $n = 10$, and the denominator is just the total number of pairs.

To compute $rankSim(q, q')$, $L_{10}(q')$ must be previously stored. Since it contains unedited results, potentially $L_{10}(q')$ can be shared by all users for space efficiency.

## 3.3 Query Processing

Different from existing systems, `Rants` separates editing from viewing, which means one does not need to log in to share published user edits. She only needs to select a view, i.e., a user set $U$ for aggregation.

In processing query $q$, $trans(q)$ is called first, which returns $q'$. If $q' = -1$, no stored user constraints need to be enforced and the unedited result list $L_0(q)$ will be presented intact. Otherwise, $RP_U(q')$ and $AP_U(q')$ are retrieved. They contain the relative and absolute preferences to be enforced on $L_0(q)$, the original unedited search results.

$L_0(q)$ is dynamic and changes over time. Potentially, this may cause problems for relative preference enforcement. Suppose $(r_1, r_2)$ and $(r_2, r_3)$ are in $RP_U(q')$. It is possible that $r_2$ maybe absent from $L_0(q)$. Then we need to make sure that $(r_1, r_3)$ is enforced.

A relative preference pair $(r, r') \in RP_U(q')$ is applicable if and only if both results are present, i.e., $r \in L_0(q)$ and $r' \in L_0(q)$. An absolute preference pair $(r, k) \in AP_U(q')$ is applicable if and only if $r \in L_0(q)$. We use $R$ and $A$ to denote the applicable subsets of $RP_U(q')$ and $AP_U(q')$ respectively. Then for enforcement purposes, $enfor(R)$ will be invoked first, followed by $enfor(A)$.

***enfor(R)***. The function enforces the relative preferences in $R$ on $L_0(q)$. Since $RP_U(q')$ is consistent, $R \subseteq RP_U(q')$ is also consistent and completely enforceable.

As indicated in Example 1, a partial order can be enforced in different ways, which reflects the fact that a dag can have many topological orderings.

In graph theory, a *topological ordering* of a dag is a linear ordering of its nodes where each node comes before all nodes to which it has outbound edges. It is a total order that is compatible with the partial order. Every dag has one or more topological orderings.

To comply with the least modification principle, we compute a topological ordering $T$ for $R$ that is the closest to $L_0(q)$. Then we iteratively process the edges in $T$ in order. In more detail, for each $(r, r') \in T$, if $r'$ is before $r$ in $L(q)$, move $r'$ down to the position immediately after $r$.

In this process, $(r', k) \in A$ maybe violated. But we do nothing about it until the next stage.

***enfor(A)***. The function enforces the absolute preferences in $A$ on $L_R(q)$, which is the list of search results returned by $enfor(R)$. As in $AP_U(q')$, $A$ may not be consistent. We use a best-effort approach to enforce $A$ as much as we can without violating the already enforced $R$.

To comely with the least modification principle, we sort the results in $A$ according to their orders in $L_R(q)$. Then we iteratively process each $(r, k) \in A$ in order, by invoking $climb(r)$.

$climb(r)$ is recursive. If $rank(r) > k$, it moves $r$ up by swapping $r$ and $r'$. If $r'$ blocks $r$, it recursively calls $climb(r')$. $r'$ blocks $r$ if $(r', k') \in A$ or $(r', r) \in R$ is violated by the planned swapping. $climb(r)$ stops when $rank(r) = k$, or no swapping can be conducted, in which case all results above $r$ (including $r$) are blocked.

# 4. DEMONSTRATION

`Rants` [7] is maintained at a regular desktop PC with Intel 3.0GHz Duo processor and 4GB memory. It was implemented using the Google web search API [5]. For illustration purposes, `Rants` only retrieves 40 HTML pages from the API for each query.

**Demonstration scenario**. As a user, you can visit the `Rants` URL to test the system. You can either create an account to login, or use the given testing account to login.

No matter logged in or out, you can specify the set of users whose preferences are to be used for aggregation. In "select search view", choose "All" for all users. To choose one or several users, enter a single user ID or a list of user IDs, e.g., "test" or "test1, test2, test3", in the edit box and click the radio button besides it. By leaving the edit box empty, you choose an aggregation on $\emptyset$, in which case the original unedited search results will be presented.

If logged in, you can issue web search queries and edit the results by specifying relative or absolute preferences. You can verify that these preferences are respected the next time you issue the same queries. To ease the comparison, search results are marked with their original, unedited ranks, which you can use as temporary IDs.

Edit transfer among similar queries is performed regardless of the login status. However, it is good to login because you need to create the edits to be transferred. For example, you can issue a query "David DeWitt" and edit the results. Then you can issue a similar query "David J. DeWitt" and see how those stored edits for "David DeWitt" are enforced in producing the query results for "David J. DeWitt".

# 5. REFERENCES


[1] Eurekster Swiki. *http://www.eurekster.com.*

[2] Google Personalized Search. *http://googleblog.blogspot.com/2007/02/personally-speaking.html.*

[3] Google SearchWiki. *http://googleblog.blogspot.com/2008/11/searchwiki-make-search-your-own.html.*

[4] Google Social Search. *http://googleblog.blogspot.com/2009/10/introducing-google-social-search-i.html.*

[5] Google Web Search API. *http://code.google.com/.*

[6] Microsoft U Rank. *http://research.microsoft.com/en-us/projects/urank/.*

[7] Rants. *http://dmlab.cs.txstate.edu/rants.*

[8] Yahoo! Personalized Search. *http://myweb.yahoo.com/.*

[9] W. Kruskal. Ordinal measures of association. *Journal of the American Statistical Association*, 53(284):814–861, 1958.