Abstract

In this paper, we address the issue of reusing the top-$k$ query results for information retrieval applications. A query comprises of a number of search terms in which multiple queries use the same search terms. The re-occurring search terms and associated results are indexed and easily accessible for retrieval. The individual results of the search terms are merged during the query processing stage and return the best combined results. Hence, the queries are observed as independent events but should be viewed in tandem. We present a lattice framework that groups frequently requested search terms and corresponding results in a cache. The stored results provide a reduction in database accesses while minimally affecting the performance. We compare the aggregate ranking (from the actual rankings) to the lattice-based aggregate ranking (using at least one stored result). We perform an empirical study of our proposed framework in which we analyze its benefits and drawbacks to the conventional method.

1 Introduction

In recent years, researchers have studied how to best represent several input rankings through the rank aggregation problem, which supposes that a set of objects are ordered by several judges or input rankings. The objective is to combine the input rankings and produce a consensus or aggregate ordering of the objects. The top few objects are typically of interest to the average user so the complete order of all objects in the aggregate ranking is unnecessary. The input rankings are the results of ordering objects from a set of sources through scoring functions, which determines the grade of match or closeness to the targeted data. The request for the first $k$ objects requires a simple output of the objects of the ranking from that particular source. Many aggregation algorithms [5, 7, 9] produce the aggregate ranking by retrieving partial information, say the first $k$ objects, from each ranking. The aggregation method can be any mathematical form such as a linear combination or average of the input rankings.

With a large amount of data, the need to simplify retrieval of requested information is in higher demand, particularly over the Web. Internet searching is a special case of information retrieval. In the earlier years of the Web, Jansen et al. [10] analyzes queries posed to Excite. They investigate a multitude of factors including the queries, sessions and search terms. Their experiments reveal the mean number of search terms (2.21) is much lower than other information retrieval systems (7–15). In addition, nearly 77% of users only viewed the top-20 results. Later research sets this percentage higher. In general, queries are not independent of each other but should be viewed in tandem. The popular search terms and associated results are indexed and easily accessible for retrieval. However, there remains the requirement to perform query processing by merging the individual results of the search terms before returning the best combined results. We can optimize the query processing step by storing some results in memory. In a naive approach, we can store a number of “canned” query results in a cache. The cache is optimal in the case where the queries are repeatedly requested and the bottled results can be returned. The drawback of this model is that only the stored queries and results will be hit and become the source of saving in query processing. A related problem is addressed in Wu et al. [15] that considered forming a highly scalable and effective metasearch engine. The queries are rewritten into several subqueries and answered by various local search engines. The subqueries results are returned, reordered and displayed for the user. Their framework is primarily concerned with the database selection problem, which is how to select the appropriate local sources to produce a solution, and does not consider caching or stored results.

We present a lattice framework that groups frequently requested search terms and corresponding results in a cache. We assume aggregate ranking is produced by taking the first $k$ objects from each ranking. Given $n$ search terms, the grouped search terms ($gst$), $gst = 2, 3, \ldots, n$, serves as the index term and the aggregation of the $gst$ rankings that can be stored in memory. The request can first check the cache
for stored results and combine them with the remaining uncached ranking(s). The stored results provide a reduction in query processing. The cache stores the most valuable $g_{st}$ rankings where eviction is based on lack of reuse. We consider the following interesting open research issues in this framework:

1. How will stored results affect the aggregate ranking?
2. How does the number of stored results affect the aggregate ranking?
3. How do we conduct storage maintenance for effective reuse?

We address the effect of more uncertainty incurred by using stored results as compared to using the original input rankings. The uncertainty may be tolerable where less accurate results are produced in lieu of the additional query processing required to produce the actual results. We vary the number of grouped search terms stored in the cache. We assess the influence of the stored results through computing the precision and Kendall-tau $[6]$ measures. More formally, the Kendall-tau counts the number of pairwise inversions in two lists, namely suppose we have two lists or rankings, $r_1, r_2$, and for each pair of objects $i$ and $j$, $K(r_1,r_2) = |(i,j)\text{ s.t. } i < j, r_1(i) < r_1(j) \text{ but } r_2(i) > r_2(j)|$. We also determine which grouped search terms to maintain in the cache for effective reuse. The query can be rewritten into a series of $g_{st}$ terms and the whole query can be answered solely with the indexed stored results. We compare the aggregate ranking created directly from the original rankings to the lattice-based aggregate ranking in which at least one stored result is applied.

2 Related Work

Informational retrieval in the context of the web has many facets. We now review a small subset of the work. Researching the Web includes: search engine performance, caching schemes, top-$k$ algorithms and rank aggregation methods. Search engine performance deals with understanding how the rankings from various search engines coincide with the perspective of the average user. The work of Vaughan $[14]$ evaluates search engine performance by testing the quality of rankings from three popular search engines to the rankings using human subjects to order the documents. The same set of queries was searched over a number of weeks, which investigated the stability of the results. The precision and recall measures serve as the comparison and evaluation tools. Precision records the number of overlapping documents between the rankings but does not evaluate the order or positional similarly (or dissimilarly) that exists between rankings. Recall is a measure that can have several interpretations since it evaluates the number of relevant documents to the set of relevant documents retrieved.

But it is impossible to know the complete set of relevant documents associated with a query. Google was found to best coincide with the rankings devised by the human subjects. Bar-Ilan $[1]$ performs a case study of search engine evaluation over time. The experiments show information loss (dropped results) and information recovery (reappearance of results) which provides evidence of search engine instability. Later in Bar-Ilan $[2]$, search engine performance is investigated comparing the result rankings of six search engines including Google.

The caching schemes of $[13]$ and $[11]$ are examined within a search engine. Saraiva et al. $[13]$ combines two types of caches: cache of query results and cache of inverted lists. These schemes store lists of documents associated with a given query, in the case of the cache of query results, or a given query term, in the case of the cache of inverted lists. Once again, caching is not on a per-item basis. The eviction from these caches removes all related documents so the interaction among documents is not maximized. This grouping effect leads to removal of documents that are not necessary. The cache of inverted lists consists of equal byte-sized pages which are stored portions of the query term’s inverted list but due to the equal pages, more documents are stored than requested (case of indirect prefetching). Lempel et al. $[11]$ proposes a probabilistic model approach to caching that distinguishes search engine users. These authors define a result as a set of documents such that documents 1-10 are the first result, documents 11-20 are the second result and so forth. The grouping of 10 objects on a result page neglects these objects’ independence with respect to different but similar queries. For example, the 10 objects for a query may contain 5 objects that solve another query. Result pages are prioritized based on the query topic and entry time frame.

Our objective is to design a cache architecture that would include the benefits of a cache of query results while avoiding some drawbacks of the cache of inverted lists and be usable independent of the choice of top-$k$ algorithm or aggregation method. Algorithms have been proposed to process rankings $[8, 9, 12]$ or schedule the evaluation of costly fuzzy rankings $[4]$ for producing the aggregate ranking with the least cost. These algorithms differ in the assumptions they make about the cost of different types of access for rankings, such as sorted access and random access. The objective of these algorithms is to minimize one type of access.

The improved versions of Fagin’s Algorithm $FA$ can be found in $[12]$. The threshold algorithm (TA) is a direct improvement of $FA$ which uses both random and sorted accesses to determine the aggregate ordering. No Random Access algorithm (NRA) only considers sorted access maintains a data structure to remember the objects seen thus far. The medrank $[9]$ algorithm approximates the median
rank of each object in which the median can be retrieved once at least 50% of the rankings have seen that particular object. The minimal probing (MPro) [4] algorithm distinguishes between search rankings, which are readily accessible in the database table, and probe rankings, which are computed via user-defined functions or externally accessed. Their algorithm orders the objects with respect to the search ranking and schedules the order of execution for the probe rankings to reduce the number of random accesses.

Dwork et al. [7] considers several rank aggregation methods for the purpose of spam reduction. Their local Kemenization process tries to optimize the aggregation of the predicates through local positional swaps of objects. The authors developed four variants of Markov chain model to be used in rank aggregation and compare them to other well-known methods. In an extension to Dwork et al. [7], Chin et al. [5] propose a new aggregation method that imposes weights to determine final placement and minimize the Kendall-tau. Bruno et.al. [3] incorporate query processing and optimization methods with rank aggregation algorithms. They develop methods to estimate an approximate range for nearest neighbor queries using histograms. For our purposes, we simplify our model such that all objects are available for retrieval and there exists a fixed retrieval cost. In future work, the cache scheme can incorporate the object’s information as factors for admission and eviction in the lattice framework.

3 Lattice Methodology

In this section, we present our lattice framework which stores a collection of predicates. These predicates are the aggregate ranking of at least two search terms. First, we discuss some basics and terminology associated with our framework. We then explain the implementation details of the structure of the cache including the entry and removal criteria. Lastly, we provide a small example.

3.1 Preliminaries

We suppose there is a set of objects $S$. We have a collection of scoring functions $P$ that assign ranks between 1 and $S$. We refer to each implementation of a scoring function as a predicate. Now we can construct a $S \times P$ matrix $(T)$ which includes each predicate. Given any position or rank $pos$, predicate $j$ and object $i$, the matrix entry $T(pos, j) = i$. We extract the first $k$ entries from each predicate for query processing. It is important to note that the same set of objects is not seen in each predicate. In this situation, we assume that the unobserved objects for a predicate will appear later therefore having a lower rank.

We perform rank aggregation on small subqueries consisting of 2 or 3 predicates. We assume that a query comprises of at least 4 search terms. We also suppose that the first 2 or 3 search terms are the most relevant to the query and more frequently requested for a general user. The combination of search terms, we term them grouped search terms (gst), index the results produced by the aggregate order of $k$ objects. The concatenation of the search terms serves as the index within $G$. If the $gst$ label would include only 1 search term, this would reduce to a cache of inverted lists. If the $gst$ label would consist of all search terms, we would have a cache of query results. Our proposed framework would require a lookup operation to locate the $gst$ label that matches the first search terms (which is also necessary for a cache of inverted lists). The presence of the $gst$ label is less costly and incurs a small amount of computational overhead. Therefore, the performance is not significantly affected. We store only $k$ objects in the cache. Given a set of $gst$ labels ($G$) each comprising a list of top-$k$ objects, we construct a matrix $L$ that is $G \times k$ where $L \subset T$. Subsequent queries must match their search terms with those located in $L$.

3.2 Lattice Details

The cache contains the results of subqueries in the hopes that the search terms are requested again. By only conducting subqueries, the search terms can appear in many requests yielding less database accesses for the cached predicate combinations. We can select which subqueries we would like to store in the cache for every query. If we chose all combinations of search terms to construct a $gst$ label, we fill the cache quickly. However, this may be a good or bad decision depending on the frequency of certain queries. The cache may be unable to discover the valuable $gst$ rankings with a higher rate of removal. In the event that the initial queries are frequently requested, then the valuable $gst$ rankings fill the cache early and more savings can be observed. If we chose to store only one subquery in the $gst$ label, namely the first 2 or 3 search terms, the cache may take longer to fill but the valuable rankings are well-established and candidates for eviction are most likely $gst$ rankings from infrequent queries. As displayed in Figure 1, we have three search terms and we assume that two search terms construct a $gst$ label. Depending on the order of the search terms, there is a possibility of three combinations that can be stored in the cache. The frequency of requests for these search terms and the space availability of the cache dictated the existence of all three combinations. Also, the query $ABC$ is different from query $BCA$. For instance, $ABC$ will form $gst$ label $AB$ and store the aggregate top-$k$ order. The execution of $BCA$ will produce $gst$ label $BC$ but use the lattice results from $AB$ as well as predicate associated with search term $C$. The request for $ABC$ (again) would use the lattice results from $AB$ and $BC$. For more accurate top-$k$ results, the lattice results from $AC$ should be included. In the case of no previously computed results, only the original predicates are used in rank aggregation and production of the top-$k$ results. Since we have computed...
results in the cache, there are several issues that need to be addressed: (1) how do you measure the size of the cache, (2) which lattice result(s) should we use and (3) when it is appropriate to replace the index entry.

As discussed briefly above, we chose to structure the cache based on number of predicates. This offers easier implementation where a positive match of the $gst$ label returns the stored aggregate ordering. We can fix the cache size to be determined by the $k$ value. For a smaller value of $k$, the cache size is low allowing more predicates to be entered into the cache while larger values of $k$ do not have this flexibility. An alternative to storing the predicates would be to store on a per-object basis such as those needed in [4].

New challenges and issues arise under these conditions. We must then consider how deep in the predicate to store. If we are looking at top-$k$ objects, then only store $k$ objects of the aggregate ranking or should we retrieve top-$l$ objects where $l > k$ such that $k$ objects are not badly positioned. The cache size becomes more important. We would need to balance how to store adequate information about a particular $gst$ label while maintaining good entry candidates in the cache. Our primary focus is the effective and valuable use of previously computed results using a lattice formation.

As the cache becomes full, the queries posed by the user can be answered using one or more results from $L$. In a conservative approach, we can use only the $gst$ label that matches the first terms. The savings occurs by not retrieving one predicate from the database. We impose a more lenient structure that the query can be decomposed into a set of subqueries, which are all possible combinations of search terms. We assess the value of the $gst$ entry to be a combination of the hit count and elapsed time of retrievals. Each label maintains the last five timestamps indicating the age of the cached ranking. We also record hit count (up to 5) to fairly compare rankings with similar elapsed times. The hit count and elapsed time addresses the frequency of accesses associated with each $gst$ entry. A large ratio implies that the entry is useful to query processing while a small ratio refers to the aging of the pair and candidate for eviction.

We measure each $gst$ entry with label $j$ as follows:

$$M(j) = \frac{\delta_j}{h_j}$$

$$h_j = \text{hit count of } gst \text{ label } j$$

$$\delta_j = \text{elapsed time in seconds}$$

The cache entries are sorted with respect to their $M(j)$ values in ascending order. The highest value indicating its advanced age in the cache and a removal candidate. In the presence of a tie, the eviction $gst$ label is selected randomly from the candidates. For new cache entries, the elapsed time becomes 0 while the hit count is assigned to 1 ensuring that the new entries are not evicted in the next iteration (thus, avoiding the division by 0).

The order of the search terms in the $gst$ label does not matter under our implementation of the lattice. Since we store the aggregate ranking based on at least two search terms, a query $q_a = (q_1, q_2, \ldots, q_m)$ and $q_b = (q_2, q_1, \ldots, q_m)$ will access the same stored $gst$ term. The aggregate ranking of the search terms will be identical for any combination and the likelihood of reuse increases. Queries using some of the same search terms can now achieve savings through using partial stored information to assist in the retrieval of the top few results. The cache of query results can not provide savings for similar queries unless the results are already stored while the cache of inverted lists must perform full query processing by accessing the index entries and merging the results for the users.

### 4 Experimental Results

We conduct a series of tests to evaluate our lattice framework for top-$k$ retrieval. The popularity of certain queries is of particular interest for these experiments. Hence, we design our queries such that some search terms appear more frequently. Our experiments generate 50 search terms allowing 6 search terms to be frequently requested. We assume each query contains 6 search terms or input rankings $(q = q_1, q_2, \ldots, q_6)$ where each predicate orders the same set of 100 objects. We chose an aggregation method that
creates the aggregate ordering of objects by using their average rank. We chose to construct the aggregate ordering using the average rank aggregation method based on its simplicity. We then vary the size of grouped search terms (gst) label to consist of 2 and 3 input rankings. We select only one value of \( k \) to be 10 since only the length of the predicate stored would change and not the number of predicates. In the case where the number of objects in the cache needs to be restricted, then different values of \( k \) would be significant. In the first series of tests, the cache size is unlimited therefore we can assess how to properly (effectively) select the combination of gst labels to best answer the request. In the second series of tests, we restrict the cache to only store a restricted number of gst labels.

We compare the aggregate ordering without the stored results to the lattice-based aggregate ordering. We assess the performance of our lattice framework through the precision and Kendall-tau measures. We define the precision to be the percentage of overlapping objects in two predicates (\( A \) and \( B \)) to the value of \( k \) (e.g. \( pr = 1 - \frac{k(A \cap B)}{k} \)). The Kendall-tau measure is the percentage of the number of inversions between the two predicates to the number of possible inversions

\[
kd = \frac{K(A, B)}{2(k - z)} \quad \text{where } z \text{ represents } A \cap B.
\]

The percentage assigned to both measures allows for easier comparison since they are on the same scale \([0, 1]\); however, it may minimize the differences inherent to the measures. The precision has an upper bound of \( k \). Kendall-tau has an upper bound of \( K_{\text{total}} = \frac{((2k - z)(2k - z - 1))}{2} \) [8]. We seek to have low numeric values for each measure. For instance, the improvement of precision has impact \( \frac{1}{k} \) whereas the benefit of an agreement for Kendall-tau is \( \frac{1}{K_{\text{total}}} \). The precision improves only when the number of overlapping objects increases. The Kendall-tau can improve when the order of the objects are changed where swapping two objects produces more objects that are in order.

We can evaluate the saving achieved with our model through counting the number of accesses needed during top-\( k \) query processing. We count the number of accesses for no-lattice model as \( k^* \text{(number of input predicates)} \), which is fixed to be 60 accesses. For the lattice-based aggregate ordering, we count the number of accesses to be \( k^* \text{(number of uncached predicates)} \) and the cached predicates would have a zero-cost.

Given the number of possible combinations of the first 2 or 3 search terms, we restricted the memory allotted to the cache as a percentage of the combinations. Once the cache is full, we execute an additional 500 queries in order to evaluate the impact of our model. The cost ratio depicted in the tables refers to the average cost of our lattice framework over the traditional cache of inverted lists. As shown in Table 1, the precision and Kendall-tau values are generally unaffected maintaining a consistent set of at least 5-6 objects in the top-10. The order of the top-10 objects (Kendall-tau) are still reasonably positioned indicated that at least half the objects are in a good order in relation to each other. We see that we are able to achieve some improvement over the cache of inverted lists approach. The cost ratio sees little improvement for the smaller cache (33% memory) but make reasonable gains for the largest cache size. The smaller cache requires more data to be removed decreasing the likelihood repetitive hits.

Table 2 shows the results where the gst label consists of 3 search terms. In general, the measurements of precision and Kendall-tau remain at about 56% and 47% respectively. The slight degradation in performance of the precision and Kendall-tau as memory size increases occurs due to the greater likelihood of having stored results and using the gst labels to compute the final ranking. The cost ratio, however, produces more performance gains. The smaller cache is comparable to those displayed in Table 1 with our proposed model obtaining 15% improvement. The larger cache sees more than 30% improvement. We believe this is due to the coverage attained under the gst label. In this case, a query can be answered solely by the cache if at a minimum of 2 gst labels are stored. The partial aggregate orderings are used to compute the final ranking and incur no cost in query processing. We show evidence that using at least one stored result can achieve savings without causing significant degradation of the information.

### Table 1. gst label consisting of 2 search terms

<table>
<thead>
<tr>
<th>Memory (%)</th>
<th>Precision</th>
<th>Kendall-tau</th>
<th>Cost Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>33%</td>
<td>0.5455</td>
<td>0.4534</td>
<td>0.8800</td>
</tr>
<tr>
<td>50%</td>
<td>0.5515</td>
<td>0.4496</td>
<td>0.8553</td>
</tr>
<tr>
<td>85%</td>
<td>0.5724</td>
<td>0.4633</td>
<td>0.7320</td>
</tr>
</tbody>
</table>

### Table 2. gst label consisting of 3 search terms

<table>
<thead>
<tr>
<th>Memory (%)</th>
<th>Precision</th>
<th>Kendall-tau</th>
<th>Cost Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>40%</td>
<td>0.5538</td>
<td>0.4690</td>
<td>0.8500</td>
</tr>
<tr>
<td>50%</td>
<td>0.5672</td>
<td>0.4803</td>
<td>0.7700</td>
</tr>
<tr>
<td>65%</td>
<td>0.5764</td>
<td>0.4857</td>
<td>0.6780</td>
</tr>
</tbody>
</table>

### 5 Conclusions

We have presented a framework for reusing top-\( k \) query results. We combine the benefits of two traditional approaches, cache of query results and cache of inverted lists. The cache of query results suffers from inflexibility in answering queries that use a portion of the results while the cache of inverted lists must perform query processing using
the indexed terms. Our proposed technique allows for a subset of the results to be stored in the cache therefore reducing the query processing required. We evaluate the usefulness of such a framework with our definitions of precision and Kendall-tau measures. Through an experimental evaluation, we observe our lattice framework produce some performance improvements as compared to the traditional approaches. A more extensive performance evaluation would provide a broader testbed for balancing the stored ranked lists.

There still remain several open issues. We can vary the number of search terms in a query to simulate specific information retrieval systems while examining the effects of adjusting the number of search terms that are assigned to a gst label. We can then examine what number of search terms for the gst label are best and under what constraints. A modification of this technique would store the data on a per-object basis. It requires a set of storage maintenance criteria but would offer benefits for top-k algorithms that have externally accessible data. Since the hit count and the elapsed time are only two factors that assess the significance of the data to the cache, we can investigate the impact of an object’s rank and access cost associated with each object and predicate.

References


