Rank Quantization

Ravi Kumar*
Google
Mountain View, CA, USA
ravi.k53@gmail.com

Ronny Lempel
Yahoo! Labs
Matam, Haifa 31905, Israel
rlempel@yahoo-inc.com

Roy Schwartz†
Technion, Israel Institute of Technology
Haifa 32000, Israel
schwartz@cs.technion.ac.il

Sergei Vassilvitskii‡
Google
New York, NY, USA
sergeiv@google.com

ABSTRACT

We study the problem of aggregating and summarizing partial orders, on a large scale. Our motivation is two-fold: to discover elements at similar preference levels and to reduce the number of bits needed to store an element's position in a full ranking. We proceed in two steps: first, we find a total order by linearizing the rankings induced by the multiple partial orders and removing potentially inconsistent pairwise preferences. Next, given a total order, we introduce and formalize the *rank quantization* problem, which intuitively aims to bucketize the total order in a manner that mostly preserves the relations appearing in the partial orders. We show an exact quadratic-time quantization algorithm, as well as a greedy $2/3$-approximation algorithm whose running is substantially faster on sparse instances. As a running example, we aggregate rankings of top-10 search results over millions of search engine queries, approximately reproducing and then efficiently encoding the underlying static ranks used by the engine. We evaluate the performance of our algorithms on a web dataset of 12 million ($2^{23.5}$) unique pages and show that we can quantize the pages' static ranks using as few as eight bits, with only a minor degradation in search quality.

1. INTRODUCTION

With the proliferation of comparison data available, whether implicitly derived from movie or restaurant ratings or explicit as in the case of "top-10" lists, there is a need for aggregating and summarizing the ranking derived from the large-scale data. While the rank aggregation problem has been well studied in the past [7], the problem of accurately quantizing the ranking into a given number of buckets in order to concisely summarize the ordering has not been sufficiently addressed.

Quantizing ranks is related to the problem of discovering partial orders [18, 22] and bucket orders [9, 12]: given information on the preference relation between pairs (or higher-order subsets) of elements, the goal is to assign elements to any number of ordered buckets in a way that is most compatible with the given preferences. In addition to practical uses of this problem [10, 16], it has also been studied from a theoretical perspective [12, 23]. While the positive results known for this problem imply almost linear-time algorithms, they all assume that the input is a tournament, i.e., *every* pair of elements has a preference relation [23]. Our setting of interest, however, is fundamentally different for two main reasons. First, the number of buckets is prespecified in advance which is not the case in the bucket ordering problem. Second, and more importantly, our setting is extremely sparse as the vast majority of pairs of elements have no specified preference ranking, though through syllogism we can compare elements indirectly. These reasons make our problem also different from the rank aggregation problem [1, 17].

A running example. A concrete example that we will use throughout this work is that of quantizing static rank scores in the context of web search. It is well known that search engines make use of query independent *static scores* to quantify the inherent quality of Web pages. Such scores may be based on link-based attributes of the page (e.g., the widely used PageRank score [4]), URL-based attributes, various design and content signals, click counts, and more [6, 25]. At runtime, upon submission of a query, static scores are mixed with query-dependent page scores to rank query results. Search engines also expose static scores to users (e.g., via browser toolbars) to give them indications on the quality of pages being browsed.

As the size of the searchable Web grows to tens of billions of pages, storing fine-grained static scores requires a

*Work done while at Yahoo! Research, Sunnyvale, CA.
†Work done while interning at Yahoo! Labs, Haifa, Israel.
‡Work done while at Yahoo! Research, New York, NY.

WSDM'13, February 4–8, 2013, Rome, Italy.

significant number of bits. Even storing the rank of each page, as derived from the scores, requires over 32 bits per page. Representing static scores with many bits is problematic on several counts. First, exposing fine-grained page scores via toolbars is an overkill for users and may reveal engine-internal scoring logic to competitors. Second, fine-grained scores cause the size of the index to increase. Expanding on the latter issue, we note that in order to keep query latencies low, search engines — by means of massive distribution [2] — keep their indexes (mostly) in main memory. This means fitting the lexicon, postings lists, and per-document ranking-related metadata in RAM. The fit is tight, and conserving memory for various optimizations is crucial. While the vast body of index compression literature (see [19, 24] and references therein) focuses on reducing the footprint of the postings lists, reducing the size of per-page ranking metadata is also beneficial, especially with access patterns to such data being more random and less sequential than the access patterns to postings lists.

Our contributions. In this work we describe how to quantize the ranks, encoding them with a given number of bits while preserving as much as possible the relative rankings reflected in a given set of partial orders. This allows the quantized ranks to be utilized in the original task with only minor degradation of quality. Specifically, in the Rank Quantization problem, we are given an order (static rank) over items (Web pages), a set of relative ranks $\mathcal{Q}$ (for example the relative rankings of top-t results for a set of queries), and a bit budget b. The goal is to *quantize* the total order into $k = 2^b$ buckets in a way that minimizes the number of ties in $\mathcal{Q}$ induced by the new, quantized ranking.

We formally define the problem and the objective function in Section 3, and show an optimal dynamic programming algorithm for this problem. However, as its running time is quadratic in the number of pages, it does not scale to Web sized inputs. We therefore develop a greedy approximation algorithm whose running time is substantially faster on sparse instances, and prove its near-optimality by carefully exploiting the structure of the Rank Quantization problem. We note that a typical input in the above setting is sparse.

For the formalization of the problem, we assume that the input to Rank Quantization is consistent: for every pair (i, j) of elements either i is preferred to j or vice-versa and for every triple (i, j, k) of elements, if i is preferred to j and j is preferred to k, then i is preferred to k. In practice, however, these assumptions rarely hold, and thus we begin with a pre-processing step, aggregating all of the (potentially inconsistent) rankings into a total order, which we call Rank Linearization. As this problem is NP-complete (it is equivalent to the Feedback-Arc-Set problem), we describe a number of heuristic approaches aimed at removing a minimal number of data points so that the remaining rankings are consistent.

A summary of our contributions is the following:

- We define the Rank Quantization problem, and give an exact quadratic-time algorithm and a greedy approximation algorithm whose running time is substantially faster on sparse instances. In particular, the structure of the problem allows us to prove a better approximation result than that obtained for general submodular problems.
- We show experimentally that our approximation algorithm far exceeds its guaranteed approximation ratio, and is superior to several baselines.
- We show evidence that static scores play a significant role in ranking the top results of search engine queries.
- We evaluate the approaches for Rank Linearization and Rank Quantization on a dataset collected by posing millions of queries to Bing and show that we can quantize the (approximate) static scores to just eight bits while still distinguishing the relative rankings of the vast majority, $\approx 98\%$, of pairs of pages that appear together in the search results.
- Finally, we empirically show that a very simple baseline, which is inferior to our approximately optimal quantization, can close the performance gap if it is allowed to use twice the number of rank buckets.

Organization. The rest of this work is organized as follows. Section 2 surveys the related work. Section 3 formally defines the problem we solve. Section 4 describes the datasets used throughout our experiments. Section 5 details the algorithms we use and the results they achieve for the Rank Linearization problem. Section 6 details the same for the Rank Quantization problem. Concluding remarks are in Section 7.

2. RELATED WORK

The specific application of quantizing static scores was first addressed by Haveliwala [13]. He proposed two metrics to measure how quantized scores distort fine-grained scores. The first metric, *TDist*, is similar in spirit to the metric we use: it is a function of the number of ties induced by the quantized scores, among the results of a particular query. He proposed several instance-independent heuristic quantization schemes, and measured their empirical performance on a small set of queries. He also considered a model whereby query results are drawn from a uniform sample over the space of Web pages, and are ranked solely by the static score. For this model he proved that the best quantization partitions the space of pages uniformly so that each bucket will hold the same number of pages. In contrast, we optimize the quantization for an actual sample of query results, solving an instance-dependent problem.

Botev et al. [3] bucket static scores in an incremental indexing context, where postings lists are maintained in decreasing static score order. They relaxed the requirement of strict document ordering by static score, instead only demanding that document ordering respects a bucketized static score. In addition to experimenting with the quantization schemes proposed by [13], they experimented with buckets where the difference between the largest and smallest static score did not exceed a certain value. They show that the enterprise search engine's rank function was hardly affected by the move from fine grained scores (over 500K

documents) to bucketized scores (with 64 buckets). Their work is purely experimental in the sense that they do not pose rank (or score) quantization an optimization problem but rather evaluate the performance of various heuristics.

Moffat et al. [20, 21] attempt to reduce the memory required by the index to store a different attribute of documents, namely their lengths (document lengths play a role in the scoring phase). They experiment with several heuristic quantization schemes and measure the ranking degradation on TREC datasets.

We note that the tools and techniques we use in our work touch upon many well-researched problems in theoretical computer science. For ease of presentation, we survey related work around those problems in the context of the exposition rather than in this section.

3. PRELIMINARIES

In this section we formally describe the setting of the Rank Quantization and the Rank Linearization problems.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of n elements and let $\mathcal{Q} = \{\sigma_1, \sigma_2, \ldots, \sigma_s\}$ be a set of s partial orders over V. In the context of this work, the partial orders in $\mathcal{Q}$ will refer to total orders over (typically very small) subsets of V. For a partial order σ, we write $u \succ_\sigma v$ if u dominates v according to σ. Any partial order σ can be represented as a directed acyclic graph $G = (V, T_\sigma)$ where T_σ are the edges of the tournament induced by σ, i.e., $(u, v) \in T_\sigma$ if and only if $u \succ_\sigma v$. Additionally, any σ can also be represented by another directed graph $H = (V, R_\sigma)$ where R_σ are the edges of a directed *path* from the highest ranking node in σ to the lowest ranking node in σ, i.e., R_σ is the longest path in the tournament T_σ.

In the concrete example we consider, V is the set of web pages and the partial orders $\mathcal{Q}$ are given by the ordering of the top t results for each query; we will use the terms query and partial order interchangeably. For a fixed query σ, let $v_1 \succ_\sigma v_2 \succ_\sigma \cdots \succ_\sigma v_t$ be the set of the top t results. The path R_σ simply has one edge from (v_i, v_{i+1}) for $1 \leq i < t$, whereas the tournament T_σ has an edge (v_i, v_j) for any pair $1 \leq i < j \leq t$.

For a given set $\mathcal{Q}$ of partial orders, we define the *query graph* as the multigraph over V whose edge set corresponds to all R_σ (keeping multiple parallel edges) and denote it by $G_\mathcal{Q}$. The closure multigraph $G_\mathcal{Q}^T$ of the query graph has an edge (u, v) any time u appeared prior to (i.e., was ranked higher than) v in some query (keeping multiple parallel edges).

Rank quantization. In the Rank Quantization problem we are given a weighted graph $G = (V, E)$ with a topological order π on the nodes and an integer $k > 0$. The solution to rank quantization is characterized by $k - 1$ topologically decreasing "break point" nodes $\{b_1, b_2, \ldots, b_{k-1}\}$ such that $B_1 = \{v : v \succ_\pi b_1\}$, for $1 < i < k$, $B_i = \{v : b_{i-1} \succeq_\pi v \succ_\pi b_i\}$ and $B_k = \{v : b_{k-1} \succeq_\pi v\}$. Each B_i is called a *bucket* and let us denote by $\mathcal{B}$ the ordered set of all k buckets. The value of the solution, denoted by $\text{cost}(\mathcal{B})$, is the total weight of edges whose both endpoints are in the same bucket, i.e., an edge $(u, v) \in E$ contributes its weight to $\text{cost}(\mathcal{B})$ if there exists a bucket $B \in \mathcal{B}$ s.t. $u, v \in B$. Intuitively, by both endpoints falling in the same bucket, the original order between them is obscured.

We now formally define the rank quantization problem.

PROBLEM 1 (Rank Quantization). *Given a weighted graph $G = (V, E)$, an ordering π on V, and the desired number k of buckets, find the partition of V into an ordered set $\mathcal{B}$ of k buckets that minimizes* $\text{cost}(\mathcal{B})$.

Rank linearization. In practice, there is rarely a single consistent ordering π. In the example of web ranking, although a static score composes a large part of the final order, many query and user specific factors also influence the final ordering of results for a specific query. Therefore, before running the rank quantization algorithms, we must first find the most consistent order from a set of partial orders, such as those given by the individual queries.

Formally, given a collection of partial orders $\mathcal{Q}$, the Rank Linearization problem asks to find a total order π that best corresponds to $\mathcal{Q}$. This is equivalent to the minimum Feedback-Arc-Set problem: given a directed multigraph $G = (V, E)$, or equivalently a weighted directed graph where the weight of an edge (u, v) is set to the number of copies of this edge in the multigraph, find the smallest subset $F \subset E$ such that $(V, E \setminus F)$ is acyclic. It is known that the minimum Feedback-Arc-Set problem is NP-complete [15]. Depending on the multigraph we are working with, we define two versions of the Rank Linearization problem that differ in their objective. In the *query* version we want to remove the fewest number of edges from $G_\mathcal{Q}$, whereas in the *tournament* version we want to remove the fewest number of edges from the closure graph $G_\mathcal{Q}^T$.

4. DATASETS

This section describes the two datasets collected for our experiments. Both are based on Bing query results collected for samples of queries submitted to Yahoo! between 23–25 of August 2011. The first dataset, hereby referred to as the *random* sample, is — as its name suggests — a sample where each instance in the query log was chosen at random with a certain probability. The second dataset, hereby referred to as the *popular* sample, was generated by taking a single instance of each of the million or so most frequently submitted queries over the three days above. For each query instance in both samples, we collected the top-10 Bing results (or all results returned in those rare cases where Bing returned fewer than 10 results per query). Table 1 shows the number of queries and distinct URLs in each sample.

Since the random sample may include multiple instances of the same query string (submitted by different users and/or by the same user at different times), it can be seen as a weighted query sample, where each query is weighted by its popularity, as opposed to the unweighted popular sample. Note that Bing occasionally returned slightly different results for different instances of the same query string.

We checked several properties of the samples that relate to the construction of the multigraphs $G_\mathcal{Q}$ and $G_\mathcal{Q}^T$. Specifically, the sizes of the two largest connected components

Table 1: Statistics of the two query samples.

	Random	Popular
# query instances	2,707,972	1,002,172
# distinct URLs	12,522,552	5,901,147
size of largest component	6,891,145	3,931,689
size of second largest component	261	968

when ignoring the directions of the edges.[1] This is a measure of the overlap between the results of different queries, as edges between nodes (pages) in $G_{\mathcal{Q}}^T$ indicate that the nodes appeared together in at least one set of top-10 query results. As shown in Table 1, both samples have a single, dominating connected component, containing over half the nodes, that is orders of magnitude larger than all other components.

Note that both algorithmic steps, linearization and rank quantization, can be done on each connected component of $G_{\mathcal{Q}}$ separately. In particular, the buckets resulting from all connected components can be trivially aggregated[2] without any loss in the objective. As the large connected component dwarfs all other components, the rest of our experiments were conducted on the large component of each sample rather than on the entire set of fragmented graphs.

5. RANK LINEARIZATION

Section 3 established the equivalence of the Rank Linearization step to solving a minimum Feedback-Arc-Set instance. This section applies known heuristics (and adaptations thereof) for approximating minimum Feedback-Arc-Set on our test collections. The results indicate that static scores play a significant role in ranking search results.

5.1 Rank Linearization: Algorithms

It is known that the minimum Feedback-Arc-Set problem is hard to approximate [11, 14]. The current best algorithm [8] achieves an approximation of $O(\log n \log\log n)$ and requires the computation of a *spreading metric* (this is in fact true for all known approximation algorithms that achieve a polylogarithmic guarantee). We will not define exactly what a spreading metric is, only mention that it is expensive to compute in practice. Thus we resort to heuristic approaches for Rank Linearization detailed below. Let $G = (V, E)$ be a directed multigraph where $|V| = n, |E| = m$.

5.1.1 Sorting by out-degree

One way to compute a full order is to sort the nodes by their weighted out-degree: those with the highest weighted out-degree should come early in the order, while those with out-degree of 0 should come last. In the case of unweighted tournament graphs (where, for any pair of nodes u, v, either $(u, v) \in E$ or $(v, u) \in E$) this simple heuristic returns a 5-approximation to the optimum Feedback-Arc-Set [5]. Although the performance guarantees do not carry over to the more general case, this heuristic is very simple to compute in time $O(n \log n + m)$.

5.1.2 Eigenvector method

The eigenvector method [7] works by computing the stationary distribution of a random walk induced by this graph with a small teleportation probability to make sure the walk is ergodic. Let w_{ij} denote the weight of the edge from v_i to v_j in G. Let the $n \times n$ transition matrix be $M = \{m_{ij}\}$ with $m_{ij} = \max(w_{ij} - w_{ji}, 0)$. Also, we define a constant "teleportation" matrix R with $R_{i,j} = \varepsilon$, for all $1 \le i, j \le n$. These two matrices are incorporated together by considering the matrix $\lambda M + (1 - \lambda)R$ for some predefined constant $0 \le \lambda \le 1$. The algorithm proceeds by finding the eigenvector that corresponds to the largest eigenvalue of the latter matrix, and the final order orders the nodes by their value in this eigenvector.

We use the iterative power method to find the stationary distribution. The algorithm begins with a uniform distribution: $u_0 = (n^{-1/2}, \ldots, n^{-1/2})$. In the ith step, this distribution is updated as: $u_i = (\lambda M + (1 - \lambda)R)u_{i-1}$ and renormalized. The algorithm halts when the change between steps is bounded: $||u_i - u_{i-1}||_2 \le \delta$, for some pre-specified constant $\delta > 0$. Since M is generally sparse, and has at most $m \ll n^2$ values, each iteration can be implemented in $O(n + m)$ time.

5.1.3 Generalized Quicksort

The generalized quicksort algorithm has been shown to lead to near optimal solutions for Feedback-Arc-Set in weighted tournament graphs [1, 7, 17]. This algorithm was subsequently extended to the closely related bucket ordering problem in weighted tournament graphs [23]. Here we adapt it to general graphs, preserving the intuition behind the algorithm, but losing the associated performance guarantees.

This is a recursive algorithm that at each step, chooses a pivot node from G uniformly at random and partitions the remaining nodes of G' into two sets, the *left* set and the *right* set (the algorithm then recurses on the two induced subgraphs). The total order π is obtained by placing the left set to the left of the pivot and the right set to the right of the pivot.

In the tournament version the nodes to the left (resp. right) of the pivot are those that have edges incoming to (resp. going from) the pivot. Below we describe the adaptation to general graphs where some nodes may not be directly connected to the pivot.

As before, the algorithm chooses a node $v_p \in G$ uniformly at random and labels it as the pivot. The next step is to select the *left* and *right* sets. We first consider the nodes that are directly connected to v_p and then we explore the remaining nodes.

Let v_i be a node in G such that either $(v_p, v_i) \in E$ or $(v_i, v_p) \in E$. Equivalently, v_i is such that $w_{ip} + w_{pi} > 0$, where w_{uv} is the total weight of edges from u to v as described above. Intuitively, the higher the value of w_{ip} the higher the probability that v_i should be placed to the left of v_p. At the same time, if there are very few edges between v_i and v_p making an inconsistent choice has a low cost, and hence we skew the probability closer to $1/2$.

[1] Note that both graphs have the same connectivity properties.

[2] The ith bucket of $G_{\mathcal{Q}}$ is simply the union of the ith buckets of all components.

Formally, let θ be a parameter of the algorithm. Denote by L_i the event that v_i was placed in the left set. Then, we have

$$\Pr[L_i] = \begin{cases} \frac{w_{ij}}{w_{ij}+w_{ji}} & \text{if } w_{ij}+w_{ji} \geq \theta, \\ \frac{1}{2} + \frac{w_{ij}-w_{ji}}{2\theta}, & \text{otherwise.} \end{cases}$$

In the second step we consider each remaining node one at a time and decide whether to place it to the left or to the right of the pivot, v_p. Let $\mathcal{L}$ be the nodes already placed to the left, similarly let $\mathcal{R}$ be those already placed to the right. Among the nodes that share an edge with some node in $\mathcal{L}$ or $\mathcal{R}$ the algorithm chooses a node v_i at random. (If no such nodes exist, v_i is chosen at random from all of the remaining nodes and added to $\mathcal{L}$ or $\mathcal{R}$ with equal probability.)

The algorithm again makes a probabilistic decision whether to place the node v_i in $\mathcal{L}$ or in $\mathcal{R}$. To compute the probability of placing v_i in each bucket, we consider a simple Markov Chain with one state for $\mathcal{L}$ and one for $\mathcal{R}$. For a set $S \subset V$, let $w_{iS} = \sum_{v_j \in S} w_{ij}$ be the total weight of the edges from v_i into the S, and $w_{Si} = \sum_{v_j \in S} w_{ji}$ be the total weight of edges from S to v_i.

The probability transition matrix of the Markov chain is the following:

$$P = \begin{array}{c} \\ (\mathcal{L}) \\ (\mathcal{R}) \end{array} \begin{array}{c} \begin{array}{cc} (\mathcal{L}) & (\mathcal{R}) \end{array} \\ \begin{pmatrix} \frac{w_{i\mathcal{L}}}{w_{i\mathcal{L}}+w_{\mathcal{L}i}} & \frac{w_{\mathcal{L}i}}{w_{i\mathcal{L}}+w_{\mathcal{L}i}} \\ \frac{w_{i\mathcal{R}}}{w_{i\mathcal{R}}+w_{\mathcal{R}i}} & \frac{w_{\mathcal{R}i}}{w_{i\mathcal{R}}+w_{\mathcal{R}i}} \end{pmatrix} \end{array}.$$

We add the node v_i to $\mathcal{L}$ with the probability given to $\mathcal{L}$ by the stationary distribution of P:

$$\Pr[L_i] = \frac{\frac{w_{i\mathcal{R}}}{w_{\mathcal{R}i}+w_{i\mathcal{R}}}}{\frac{w_{i\mathcal{R}}}{w_{\mathcal{R}i}+w_{i\mathcal{R}}} + \frac{w_{\mathcal{L}i}}{w_{\mathcal{L}i}+w_{i\mathcal{L}}}}.$$

The above algorithm can be efficiently implemented, with each pivot round taking $O(n+m)$ time.

5.1.4 Local search improvements

None of the above algorithms guarantees that the final linear orders are locally optimal, therefore we perform a local search to further reduce the number of edges removed (see Section 5.2 for the specific use). Given a size parameter s, the algorithm checks whether swapping a pair of nodes within s of each other in the total order improves the cost function, and halts when no such pairs are found.

5.2 Rank Linearization: Experimental results

We executed all three linearization algorithms: sorting by out-degree (SO), the eigenvector heuristic (EG), and the generalized quicksort (GQ), for each of the two query samples (on the largest component of $G_{\mathcal{Q}}$ when viewed as an undirected graph). We also applied the local improvement heuristic (LI) on each of the orderings generated by the algorithms above. The exact settings of our executions were as follows. We ran EG with $\lambda = 0.85$, $\varepsilon = 1$ and $\delta = 10^{-10}$. Since the GQ algorithm is randomized, we report the average results across 32 independent runs for each setting of the parameters. The standard deviation of the results due to different random seeds and different values of θ was under 0.1%. For SO, no special parameters are needed. We chose $s = 512$ for LI.

Table 2: Percentage of edges in the feedback set for each linearization algorithm.

	Random	**Popular**
Lower Bound	3.7	4.4
SO	17.62	19.91
SO + LI	15.54	19.74
EG	17.03	19.02
EG + LI	15.07	18.86
GQ	19.53	19.16
GQ + LI	19.53	19.16

The results are summarized in Table 2, where we measure the performance of each algorithm according to the fraction of the edges of $G_{\mathcal{Q}}^T$ that belong to the feedback set. The results in Table 2 indicate that, for both samples, we can find linear orderings that agree with the direction of over 80% of the edges in $G_{\mathcal{Q}}^T$. Intuitively, this suggests that the static score signal plays a significant role in determining the order among the top-10 results of queries. Another interpretation is that query-dependent score factors are all quite high at the very top of the results list, and so those factors do not dominate the static score there.

Note that unlike SO and EG, the GQ algorithm did not benefit from a local improvement step, but produced qualitatively worse results. To estimate the performance of the linearization algorithms, we find that 3.7% of edges in the random graph and 4.4% of edges in the popular graph must be cut to break cycles of length 2. These cycles are formed when a pair of results appears in reverse order in two different queries.

6. RANK QUANTIZATION

Ultimately, the quality of quantized ranks is task-oriented, namely determined by its impact on the application. In our case we are quantizing search engine static ranks, with the application being ranking query results. Ideally, we would want to check the performance of the search engine with the quantized scores; however, this type of evaluation is beyond our reach. We therefore use the number of ties introduced by quantization as a proxy [13], where a tie occurs when two of the top t results of a query are assigned to the same bucket. Whenever this happens, the quantized scores obscure any difference in the static ranks of these two pages. Conversely, when the quantized scores allow the engine to reconstruct the score relation between all or most pairs of top-t pages, it should be able to utilize the quantized static ranks to approximate its original query results as well. With this measure of success in mind, we present quantization algorithms and their results in the remainder of this section.

6.1 Rank Quantization: Algorithms

We present algorithms for the Rank Quantization problem. Our first algorithm shows how to solve the problem exactly, albeit in time quadratic in the total number of elements,

which is untenable for Web scale. The second algorithm is an adaptation of the classical greedy set cover method to this setting and can be implemented substantially faster for sparse inputs (a typical input in our setting is indeed sparse). The added structure of the problem allows us to prove a $^2/_3$-approximation performance guarantee in the worst case.

Recall that in the Rank Quantization problem we are given a weighted graph $G = (V, E), |V| = n, |E| = m$ and a topological order π on the nodes. The solution to rank quantization has a compact representation. It can be characterized by $k-1$ decreasing *breakpoint* nodes $\{b_1, b_2, \ldots, b_{k-1}\}$ such that $B_1 = \{v : v \succ_\pi b_1\}$, for $1 < i < k$, $B_i = \{v : b_{i-1} \succeq_\pi v \succ_\pi b_i\}$ and $B_k = \{v : b_{k-1} \succeq_\pi v\}$. We will drop the subscript π when it is clear from the context.

6.1.1 Exact algorithm

We exploit the structure of the optimum solution to define a simple dynamic programming algorithm for the problem. We will maintain a table A where $A[i, j]$ for $1 \le i \le n+1, 1 \le j \le k$ stores the cost of bucketing the elements v_i through v_n into j buckets. Our goal is to compute $A[1, k]$. We proceed to fill in $A[i, j]$ in decreasing order of i. When processing element v_i we decide on the last element, v_ℓ of this bucket, and look up the cost of quantizing the subsequent elements $v_{\ell+1}, \ldots, v_n$. Formally the recursion step is:

$$A[i, j] = \min_{i \le \ell \le n} \left(A[\ell + 1, j - 1] + \sum_{v_i \succ v_x \succ v_y \succ v_\ell} w_{xy} \right).$$

Algorithm 1 presents all the details.

Algorithm 1 Dynamic programming.

```
for j = 1, ..., k do
    A[n + 1, j] ← 0
end for
for i = n, ..., 1 do
    for j = 1, ..., k do
        A[i, j] ← min_{i≤ℓ≤n}(A[ℓ + 1, j − 1]
                              + Σ_{v_i ≻ v_x ≻ v_y ≻ v_ℓ} w_xy)
    end for
end for
Output A[1, k]
```

With some preprocessing, we can execute the recursive step in $O(n)$ time, which leads to an $O(n^2 k)$ overall running time for the algorithm. As we stated earlier, this is prohibitive for large n.

6.1.2 Greedy approximation

Since the running time of the exact dynamic programming algorithm is too large, one can settle for a provably approximate solution that can be computed much faster. Recall the objective of the problem is to find a set of buckets that minimizes the number of edges in the subgraphs induced by each bucket:

$$\sum_{i=1}^{k} \sum_{u,v \in B_i} w_{uv}.$$

Alternatively, we can maximize the number of edges between buckets. To proceed with the analysis, we say an edge (u, v) is *cut* by a breakpoint b if $u \succ b \succeq v$. Let $C(b) = \{(x, y) \in E \mid x \succ b \succeq y\}$ be the set of edges cut by a breakpoint. Then the total weight of the cut is:

$$|C(b)| = \sum_{(x,y) \in C(b)} w_{xy}.$$

Extending this definition to multiple breakpoints, for a set $B = \{b_1, b_2, \ldots, b_\ell\}$, the set of edges cut is:

$$C(B) = \bigcup_{b \in B} C(b).$$

Similarly, the total weight is:

$$|C(B)| = \sum_{(x,y) \in C(B)} w_{xy}.$$

Therefore an equivalent formulation of the rank quantization problem is to find a set $B = \{b_1, \ldots, b_{k-1}\}$ of breakpoints that maximizes the total number (weight) of edges cut, $|C(B)|$. This can be viewed as a special case of the well studied Maximum Coverage problem [11]: given a (weighted) ground set $\mathcal{N}$, a collection $\mathcal{S} \subseteq 2^{\mathcal{N}}$ of subsets of $\mathcal{N}$, and a parameter k, find at most k subsets out of $\mathcal{S}$, say $S_1, S_2, \ldots, S_k \in \mathcal{S}$ that maximize $|\cup_{i=1}^{k} S_i|$, i.e., the weight of the elements in the union.

In this formulation each edge (u, v) defines an element of the ground set E, and each breakpoint b_i defines a set $S_i = C(b_i) \subseteq E$. Algorithm 2 shows all the details. With a careful implementation this problem can be solved in time $O(nk + m \log m)$. The greedy algorithm that iteratively selects the set that covers the highest number of yet uncovered elements yields a $1 - 1/e$ approximation for Maximum Coverage. Below we show that due to the special structure of the problem, the same greedy algorithm achieves a $^2/_3 > 1 - 1/e$ approximation for rank quantization. (We conjecture that the correct bound for this algorithm is $^3/_4$.)

Algorithm 2 Greedy algorithm.

```
S ← ∅
T ← ∅
while |S| < k do
    b ← arg max_{b'} |C(b') \ T|
    T ← T ∪ C(b)
    S ← S ∪ {b}
end while
Output |C(S)|
```

THEOREM 2. *The greedy algorithm achieves a $^2/_3$-approximation to the maximization version of Rank Quantization.*

PROOF. Let g_i be the element selected in the ith round, and denote by G_i the solution computed by selecting the first i elements greedily. Thus $G_i = G_{i-1} \cup \{g_i\}$, where $G_0 = \emptyset$. Let H_i be the $k - i$ elements that optimally extend the solution G_i. We will compare the cost of the greedy solution, G_k to that of the optimal solution, H_0.

Finally, for a set S, let

$$\Delta_i(S) = |C(G_i \cup S)| - |C(G_i)|,$$

be the incremental benefit of selecting breakpoints in S, having already selected G_i. To simplify notation for singleton sets S, we will write $\Delta_i(s)$ instead of $\Delta_i(\{s\})$.

We will show that, in every step, the greedy algorithm's decision is nearly optimal.

LEMMA 3. *For $i \geq 1$, $\Delta_{i-1}(H_{i-1}) - \Delta_i(H_i) \leq \frac{3}{2}\Delta_{i-1}(g_i)$.*

Given Lemma 3, we can complete the proof of the theorem by the following argument:

$$\begin{aligned}|C(H_0)| &= \Delta_0(H_0) \\ &= \Delta_k(H_k) + \sum_{i=1}^{k} \Big(\Delta_{i-1}(H_{i-1}) - \Delta_i(H_i)\Big) \\ &\leq \frac{3}{2}\sum_{i=1}^{k} \Delta_{i-1}(g_i) \\ &= \frac{3}{2}|C(G_k)|,\end{aligned}$$

where we use Lemma 3 and the fact that $H_k = \emptyset$ to reach the first inequality. □

Now, we proceed to establish Lemma 3, which uses the specific structure of our problem.

PROOF OF LEMMA 3. To prove the lemma, given the greedy solution after $i-1$ steps, G_{i-1}, its optimal extension, H_{i-1} and the greedily selected breakpoint g_i, we exhibit a set $S \subset H_{i-1}$ with $|S| = k - i$ such that

$$\Delta_{i-1}(H_{i-1}) - \Delta_i(S) \leq \frac{3}{2}\Delta_{i-1}(g_i).$$

Since H_i is the optimal extension of G_i, $\Delta_i(H_i) \geq \Delta_i(S)$ and this establishes the proof.

Denote the elements of $H_{i-1} = \{h_1 \prec h_2 \prec \cdots \prec h_{k-(i-1)}\}$ and let $g = g_i$ be the next point selected by the greedy algorithm.

Case 1. Suppose $g \prec h_1$ (a similar argument holds for, $g \succ h_{k-i+1}$). Consider a solution $S = H_{i-1} \setminus \{h_1\}$.

By the greedy property of g, $\Delta_{i-1}(g) \geq \Delta_{i-1}(h_1)$. Furthermore, g covers no more edges from $C(S)$ than h_1 and hence we have

$$C(g) \cap C(S) \subset C(h_1) \cap C(S).$$

Therefore, $\Delta_i(S) \geq \Delta_{i-1}(S)$, and we have:

$$\Delta_i(S) + \Delta_{i-1}(g) \geq \Delta_{i-1}(S) + \Delta_{i-1}(h_1) \geq \Delta_{i-1}(H_{i-1}).$$

Case 2. Suppose $h_j \prec g \prec h_{j+1}$. To untangle the interactions between $\Delta_{i-1}(h_j), \Delta_i(g)$ and $\Delta_i(g_{j+1})$, consider subdividing the edges cut by h_j and h_{j+1} into the following five disjoint sets:

$E_1 = C(h_j) \setminus C(g) \setminus C(G_i)$,
$E_2 = C(h_j) \cap C(g) \setminus C(G_i)$,
$E_3 = C(g) \setminus (C(h_j) \cup C(h_{j+1})) \setminus C(G_i)$,
$E_4 = C(h_{j+1}) \cap C(g)| \setminus C(G_i)$, and
$E_5 = C(h_{j+1}) \setminus C(g) \setminus C(G_i)$.

Then,

$\Delta_{i-1}(h_j) = |E_1| + |E_2|$,
$\Delta_{i-1}(g) = |E_2| + |E_3| + |E_4|$, and
$\Delta_{i-1}(h_j) = |E_4| + |E_5|$.

Assume that:

$$|E_1| \leq |E_5|. \tag{1}$$

The reverse case is similar. Since g was the point maximizing $\Delta_{i-1}(g)$,

$$\Delta_{i-1}(g) = |E_2| + |E_3| + |E_4| \geq |E_1| + |E_2| = \Delta_{i-1}(h_j). \tag{2}$$

And:

$$\Delta_{i-1}(g) = |E_2| + |E_3| + |E_4| \geq |E_4| + |E_5| = \Delta_{i-1}(h_{j+1}). \tag{3}$$

Let $S = H_{i-1} \setminus \{h_j\}$. Then, compared to $G_{i-1} \cup H_{i-1}$, the solution $G_{i-1} \cup S \cup \{g\}$ will cut the edges in E_3, but will no longer cut those in E_1. Hence:

$$\Delta_{i-1}(H_{i-1}) - (\Delta_i(H_i) + \Delta_{i-1}(g_i)) = |E_1| - |E_3|.$$

Summing Equations (1) and (2) we have:

$$2|E_1| \leq |E_5| + |E_3| + |E_4|.$$

Using Equation (3) we have:

$$\begin{aligned}&2|E_1| \leq |E_2| + |E_3| + |E_3| + |E_4| \\ \Longrightarrow\ &2(|E_1| - |E_3|) \leq |E_2| + |E_4| \\ \Longrightarrow\ &\Delta_{i-1}(H_{i-1}) - (\Delta_i(H_i) + \Delta_{i-1}(g_i)) \leq \frac{1}{2}\Delta_{i-1}(g_1).\end{aligned}$$

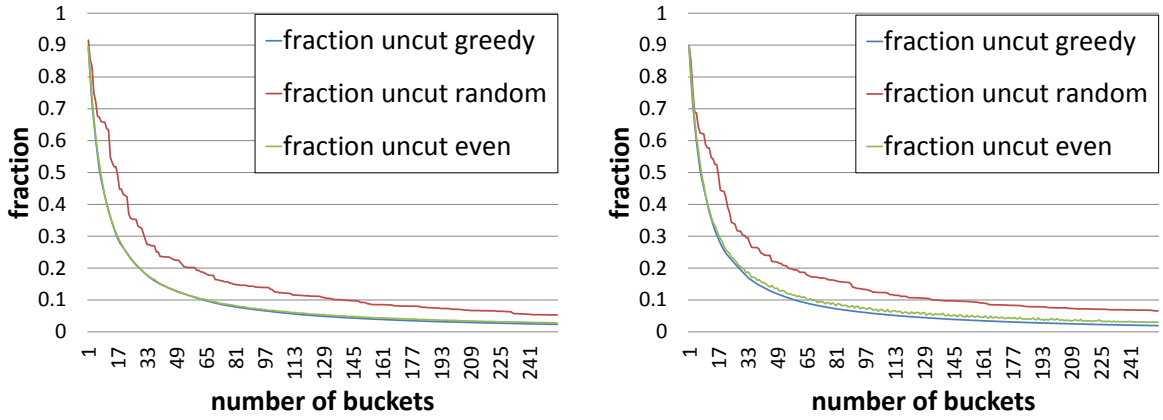
In both cases we bound the loss due to greedy behavior, which allows us to conclude the proof. □

6.2 Bucketing results on synthetic datasets

Recall that the dynamic programming algorithm that optimally solves the rank quantization problem is quadratic in the number of input pages (graph nodes), which is prohibitive for the sizes of our query samples. Therefore, to evaluate the performance of Greedy with respect to the optimal algorithm, and in particular to empirically check whether it exceeds the 2/3 approximation guarantee on the Maximum Coverage instances generated by Rank Quantization (see Section 6.1), we compare the two on synthetic workloads where we can afford to optimally solve the problem instances.

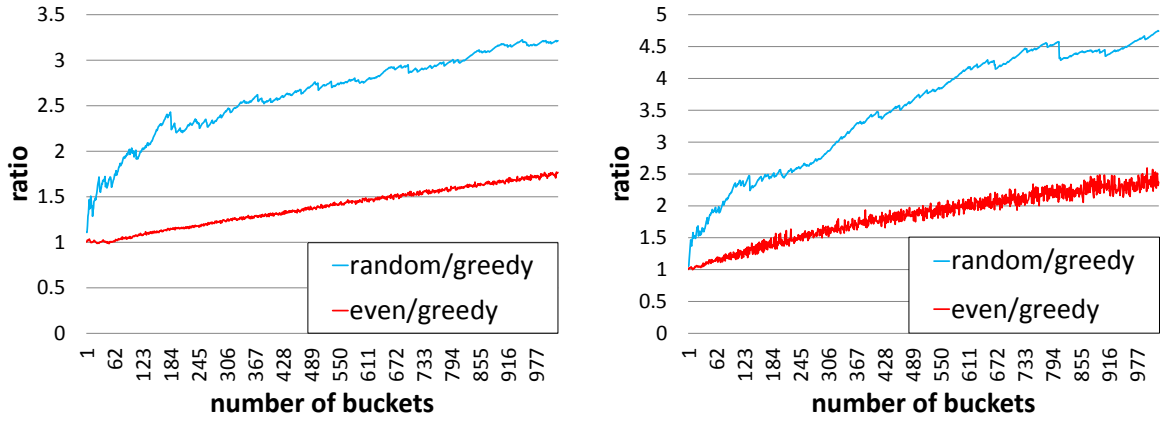
We generated random ordered graphs with the following parameters: $N = 2^{15}$ is the number of nodes, $Q = 2^{13}$ is the number of queries, and $k = 32$ is the number of required buckets. The nodes are ordered $1, \ldots, N$, meaning that all edges are ordered from lower-numbered nodes to higher-numbered nodes. We denoted the number of results per query by t, and experimented with $t = 8$, 10, 12, 14 and 16.

We considered each query as a random subset of t nodes (this is similar to the model used in [13]). We ran both algorithms on these synthetic graphs and measured the performance of the greedy approximation as compared to the optimal solution. Table 3 lists the ratio of the objective functions of the greedy approximation to the optimal solution for both the minimization and maximization versions of the problem. Each entry in the table was computed by averaging 128 independent samples of the input graph. As can be seen from the table, the greedy approximation works extremely well for both objectives, with a deviation from optimum of at most 2.5% for the minimization objective and roughly 0.5% for the maximization objective.



(a) Popular (b) Random

Figure 1: Fraction of edges left uncut by the different rank quantization methods in the *unweighted* (a) popular (b) random query graphs.



(a) Popular (b) Random

Figure 2: The relative performance of the `random` and `even` baselines as compared to the greedy approximation algorithm, in terms of uncut edges, on the *unweighted* (a) popular and (b) random query graphs.

Table 3: The approximation ratio of the greedy algorithm on synthetic random queries.

t	Minimization	Maximization
8	1.024	0.997
10	1.019	0.997
12	1.017	0.997
14	1.019	0.996
16	1.022	0.994

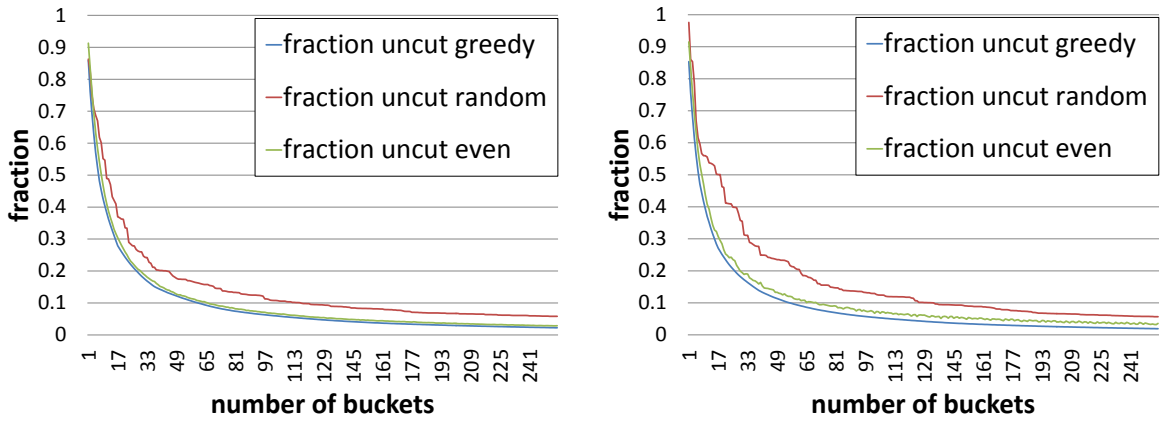
The synthetic graphs above have certain characteristics that are similar to the graphs induced by the real query samples. When linearizing both the random and popular samples (see Section 5.2), the average length of an edge connecting two adjacent results of the same query was approximately $n/12$. Since queries had almost exclusively 10 results, this implies that on average, in both samples, queries are well distributed throughout the linear order, and can be reasonably approximated by a random subset of 10 pages.

6.3 Bucketing results on query samples

We evaluated the performance of the greedy approximation algorithm on weighted and unweighted multigraphs induced by both the popular and random query samples.[3] For each graph we chose the best linearization (as described in Section 5.2). Since the original query results were not fully consistent with the linearized rankings, for the purposes of the quantization experiment we reordered each top-t list to be consistent with the linearized order.

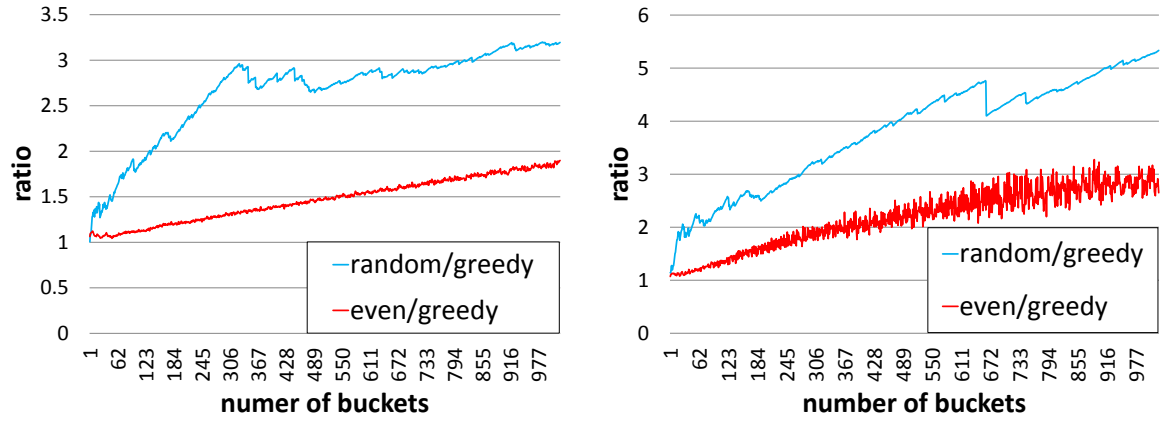
When evaluating unweighted multigraphs, we assigned each (individual) edge a weight of 1. The weighted versions were constructed to account for the fact that inducing a tie between the static ranks of the first two results of a query causes higher potential damage than inducing a tie at the

[3]Even in the popular sample, where queries do not repeat, edges may repeat (hence creating a multigraph) as the same pair of pages may appear in the results of multiple queries.



(a) Popular (b) Random

Figure 3: Fraction of edges left uncut by the different rank quantization methods in the *weighted* (a) popular (b) random query graphs.



(a) Popular (b) Random

Figure 4: The relative performance of the `random` and `even` baseline, in terms of uncut edges, as compared to the greedy approximation algorithm on the *weighted* (a) popular and (b) random query graphs.

bottom of the results page (as ties may cause the engine to erroneously swap the results). Thus, in the weighted version of the problem, we assign an edge connecting results i and $i+1$ a weight of $\frac{1}{\log_2(1+i)}$ (this weighting was motivated by the position discount used in computing DCG).

To evaluate the performance of the greedy approximation algorithm, we consider two additional quantization heuristics. The first, which we call **`even`**, selects evenly spaced breakpoints in the total order. Given an ordering on n points, it selects points in positions $n/k, 2n/k, \ldots, n - n/k$ as the breakpoints. The second, which we call **`random`**, selects $k-1$ points uniformly at random as breakpoints. For all heuristics, when quantizing a total order, we measure the cost of the solution, i.e., the number of *uncut edges* — edges whose endpoints belong to the same bucket. Equivalently, this counts the number of induced static rank ties between pages co-appearing in the results of a query, i.e., pairs of results whose true rank relation has been obscured.

Figure 1 illustrates the number of uncut edges as a function of the number of buckets, in the unweighted versions of the popular and random graphs. Both the greedy approximation and the **`even`** heuristics perform very well, whereas **`random`** lags behind. For example, when the number of bits available to store the static rank is set to 8, corresponding to 256 distinct buckets, the greedy approximation leaves 1.9% of edges in the random sample uncut (and 2.3% in the popular sample). The **`even`** heuristic does not perform as well, leaving 3% and 2.8% of edges uncut respectively. To further illustrate the difference between the three approaches, we plot the relative performance of the two baselines as compared with the greedy algorithm (in terms of uncut edges) in Figure 2. Here, we can clearly see that the gap between the greedy algorithm and the two baselines increases with an increased number of buckets.

We repeated the same set of experiments on the weighted graphs. We again show both the absolute results in Figure 3 and the relative performance of greedy over the baseline in Figure 4. Here the difference between the greedy algorithm and the **`even`** heuristic is more pronounced. The effect is not surprising, since the **`even`** heuristic is independent

of the weights, whereas the greedy algorithm selects more breakpoints near the top of the ranking and fewer near the bottom. The relative performance of `even` is about 50% worse in the weighted case: with 256 buckets on the popular graph it leaves 85% more weight uncut than the greedy algorithm (3.5% versus 1.9%); when the total number of buckets is around one thousand, the `even` heuristic is a factor of 3 worse than the greedy algorithm on the random query graph.

Despite the `even` heuristic not performing as well as the greedy algorithm and not being able to adapt to weighted edges, the loss in performance may be small in practical terms. Note that granting `even` a single extra bit for storing the quantized rank enables `even` to catch up to the performance of the greedy algorithm. In other words, `even` quantization with a bit budget of $b+1$ — using twice the number of buckets — outperforms greedy bucketization with a budget of b bits.

7. CONCLUSIONS

This work formally defined the Rank Quantization problem and presented (quadratic) exact and approximation algorithms for it. Technically, before tackling the Rank Quantization instance, we performed a Rank Linearization step for finding a full order of good agreement with many partial orders over a set of elements.

Our experimental results, over two different samples comprised of millions of queries and pages each, showed that it is possible to find a linear order of the pages that agrees with roughly 80% of the rank preferences expressed by queries. This indicates that static scores play a significant role in deciding the ranking among the top results of queries. Moreover, we find that this linear order can be effectively quantized. Perfectly preserving the ranks in the largest connected component of the graphs we consider requires 23 bits of precision for the random query graph and 22 bits for the popular query graph. However, if a small loss in precision can be tolerated, significantly fewer bits are needed. Specifically, when quantizing with just 8 bits, i.e. when using *just 256* different ranks, the relative rankings of only 2% of pairs of URLs in the top-10 results are obscured.

8. REFERENCES

[1] N. Ailon, M. Charikar, and A. Newman. Aggregating inconsistent information: Ranking and clustering. *JACM*, 55(5), 2008.

[2] L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

[3] C. Botev, N. Eiron, M. Fontoura, N. Li, and E. Shekita. Static score bucketing in inverted indexes. In *CIKM*, pages 311–312, 2005.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *WWW*, pages 107–117, 1998.

[5] D. Coppersmith, L. Fleischer, and A. Rudra. Ordering by weighted number of wins gives a good ranking for weighted tournaments. *TALG*, 6(3), 2010.

[6] N. Craswell, S. Robertson, H. Zaragoza, and M. Taylor. Relevance weighting for query independent evidence. In *SIGIR*, pages 416–423, 2005.

[7] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001.

[8] G. Even, J. Naor, B. Schieber, and M. Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.

[9] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee. Comparing partial rankings. *SIDMA*, 20(3):628–648, 2006.

[10] J. Feng, Q. Fang, and W. Ng. Discovering bucket orders from full rankings. In *SIGMOD*, pages 55–66, 2008.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, San Francisco, 1979.

[12] A. Gionis, H. Mannila, K. Puolamäki, and A. Ukkonen. Algorithms for discovering bucket orders from data. In *KDD*, pages 561–566, 2006.

[13] T. Haveliwala. Efficient encodings for document ranking vectors. In *IC*, pages 3–9, 2003.

[14] V. Kann. *On the Approximability of NP-complete Optimization Problems*. PhD thesis, Royal Institute of Technology, Stockholm, 1992.

[15] R. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–104, 1972.

[16] S. Kenkre, A. Khan, and V. Pandit. On discovering bucket orders from preference data. In *SDM*, pages 872–883, 2011.

[17] C. Kenyon-Mathieu and W. Schudy. How to rank with few errors. In *STOC*, pages 95–103, 2007.

[18] H. Mannila and C. Meek. Global partial orders from sequential data. In *KDD*, pages 161–168, 2000.

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

[20] A. Moffat, J. Zobel, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *VLDB*, pages 352–362, 1992.

[21] A. Moffat, J. Zobel, and R. Sacks-Davis. Memory efficient ranking. *IPM*, 30:733–744, 1994.

[22] A. Ukkonen, M. Fortelius, and H. Mannila. Finding partial orders from unordered 0-1 data. In *KDD*, pages 285–293, 2005.

[23] A. Ukkonen, K. Puolamäki, A. Gionis, and H. Mannila. A randomized approximation algorithm for computing bucket orders. *IPL*, 109:356–359, 2009.

[24] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann, 1999.

[25] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38, 2006.