February 15, 2012

RT0938

Computer Science; Mathematics 12 pages

Research Report

Combinatorial Analysis for Efficient Distributed Top-K Keyword Aggregation

Issei Yoshida, Yuya Unno, Yuta Tsuboi

IBM Research - Tokyo
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Combinatorial Analysis for Efficient Distributed Top-K Keyword Aggregation

Issei Yoshida
IBM Research - Tokyo
Kanagawa, Japan
issei@jp.ibm.com

Yuya Unno
Preferred Infrastructure, Inc.
Tokyo, Japan
unno@preferred.jp

Yuta Tsuboi
IBM Research - Tokyo
Kanagawa, Japan
yutat@jp.ibm.com

ABSTRACT
The Top-k Keyword Aggregation is a kind of Top-k query processing that finds the k most frequent keywords in a document set, where the documents are dynamically selected through keyword searches. Top-k is often used in exploratory text analysis, where the response time for queries is especially crucial in interactive analyses that are seeking higher-level knowledge within a large document collection. In this paper we show that, in a distributed environment where the central master node aggregates the keywords sent from the worker nodes, we can reduce the problem of optimization of the number of keywords each worker node has to return to the master node to a problem of pure combinatorics. To estimate the number of keywords that each worker returns, our combinatorial analysis carefully defines the probability that we obtain the correct answers. There is a reasonable trade-off between the number of keywords and the probability. With high probabilities, say 90% or 95%, we can significantly reduce the number of keywords processed by all the worker nodes. Even when the answer is not correct, our method assures that the end user can know which of the returned k keywords in the answer are actually correct and which of them may be spurious, and the number of the latter is actually very small compared to that of the former. We describe the column-based keyword partitioning approach to implement our algorithm in distributed Top-k. Experimental evaluations on a large archive of medical documents confirmed the results of our theoretical analysis.

Categories and Subject Descriptors
G.2.1 [**Combinatorics**]: Counting problems, Permutations and combinations; H.2.4 [**Systems**]: Query Processing

General Terms
Theory, Algorithm, Performance

1. INTRODUCTION
One of the fastest growing areas of text search and analytics applications has been business intelligence. It became apparent in the late 1990s that human experts could no longer analyze thousands of customer conversation logs (call-logs) generated every week at a call center, although the analysis of this data was vital for early problem detection and customer claim analysis. Text mining tools were widely introduced at call centers to address this problem and are also useful in many other fields such as life sciences and competitive intelligence (patent mining) [8].

Studies on text mining technology have found that computing the distribution of keywords (including terms, named entities, and other types of entities extracted from text) is fundamental for aggregating the content of a document set and in building more advanced analytic functions [16, 13]. More specifically, Top-k keyword aggregation, which gives the k most frequent keywords and their document frequencies in a dynamically given document set, is known to be crucial for exploratory text analysis. Here we mean by exploratory text analysis a sequence of analytic operations by a human analyzer. For example, an analyzer first searches for a specific keyword of interest, then runs Top-k keyword aggregation for the documents that are retrieved by the keyword search. The Top-k result provides information about which keywords are frequently mentioned in the context specified by the search keyword. After investigating the keywords in the result of Top-k, more keywords can be added to the search condition to drill down into the documents. Top-k is then performed again for the new search condition to see what kinds of keywords are frequently mentioned in the new document set.

In order for an analyzer to do analysis comfortably in such a trial-and-error manner, it is essential for the analysis system to provide Top-k keyword aggregation with response times close to those of search engines. Some studies have addressed this problem and proposed efficient algorithms and data structures for Top-k keyword aggregation [20, 18].

Motivated by these applications, we are investigating distributed Top-k query processing considering both scalability and accuracy. These days we often have more than tens of millions of documents for analysis, so we want to process Top-k queries by using large numbers of distributed worker nodes. For usefulness, a system should return either a correct answer or a partly correct answer with information about which of the k keywords are actually correct and which may be incorrect.

In this paper we show that, in a distributed environment where the central master node aggregates the keywords sent from the worker nodes, optimization of the number of keywords that each worker node has to return to the master node is reduced to a problem of pure combinatorics. To estimate the number of keywords that each worker returns, our combinatorial analysis carefully defines the probability that we obtain the correct answers. There is a reasonable trade-off between the number of keywords and the probability. With high probabilities, say 90% or 95%, we can significantly reduce the number of keywords processed by all the worker nodes. Even when the answer is not correct, our method assures that the end user can know which of the returned k keywords in the answer are actually correct and which of them may be spurious, and the number of the latter is actually very small compared to that of the former. For example, we show that when we want to obtain the Top-100 keywords and there are 32 worker nodes available, then it is sufficient to compute only the Top-11 (instead of the Top-100) on each worker node in order to obtain a correct answer with probability 90%.

Our combinatorial method is sufficiently general such that it is available for general distributed Top-k query processing where each scored object to aggregate comes from only one of the worker nodes. To discuss application of our method to keyword aggregation, we describe the column-based keyword partitioning approach for Top-k in a distributed environment. The keyword partitioning approach divides the set of keywords that appear in the whole document set into disjoint subsets, and each worker node corresponds to only one of the subsets and processes only the keywords in the subset. There are two major advantages of our keyword partitioning approach. First, it requires only one-pass between the central master node and the worker nodes. Second, it always gives a correct Top-k answer if each worker node returns its local Top-k to the master node. On top of the efficient calculation of local Top-k in each worker node [20, 18] we can construct an efficient algorithm for distributed Top-k.

The rest of this paper is organized as follows. We present the definitions and notations for our work in Section 2. In Section 3 we review how to calculate Top-k on a single node. In Section 4 we describe a framework to calculate Top-k in a distributed computing environment in which each node calculates its local Top-k by using a keyword partitioning approach. Section 5 is the main part of this paper. We propose a novel method to improve the keyword partitioning approach. Section 6 describes experimental results and presents their analysis. We discuss related work in Section 7 and give our conclusions in Section 8.

2. PROBLEM FORMULATION

2.1 Top-k Keyword Aggregation

In a high-level view, Top-k Keyword Aggregation is to collect keywords that appear most frequently in a specific document set. The document set is given dynamically, usually as a result of keyword search in the context of text analysis.

Definition 1. *Let W be the set of all possible character strings and a* **keyword** *is an element of W. For example, "food", "investigate" and "2011" are elements of W. A* **document** *is a finite subset of W. This means that each keyword represents a binary feature of a document.*

Let us consider a finite multiset of documents $D = \{d_1, \cdots, d_m\}$. We define D as a multiset so that we allow $d_i = d_j$ as a set for $i \neq j$ because it is natural to consider that distinct documents may have exactly the same content. For a subset $D_S \subset D$ and a keyword $w \in W$, let $freq(D_S, w) := |\{d \in D_S \mid w \in d\}|$, which is the number of documents in D_S that contain the keyword w[1]. *We call $freq(D_S, w)$ the* **keyword frequency** *of w in D_S. In particular when $D_S = D$ we call $freq(D, w)$ the* **global frequency** *of w.*

Each document of D is associated with a **document id**, *an integer unique in D. We often identify the symbol of a document $d \in D$ with its document id, and identify D (or its subset) with the set of the corresponding document ids.* ■

Definition 2. *For any $D_S \subset D$, let $W(D_S) := \{w \in W \mid freq(D_S, w) > 0\}$ and sort all of the keywords of $W(D_S)$ in descending order of frequency, $freq(D_S, w_1) \geq freq(D_S, w_2) \geq \cdots > 0$. For a positive integer k and $D_S \subset D$, we define* **Top-k keyword aggregation** *for D_S, or simply* **Top-k** *(if D_S is understood from the context), as the set of k pairs $\{(w_1, freq(D_S, w_1)), \cdots, (w_k, freq(D_S, w_k))\}$. We call each of w_i $(i \leq k)$ a Top-k most frequent keyword or simply a Top-k keyword. When there exist p and q $((p, q) \neq (0, 0))$ such that $freq(D_S, w_{k-p}) = \cdots = freq(D_S, w_k) = \cdots = freq(D_S, w_{k+q})$, there is more than one valid Top-k (See Example 1). Our problem is to get any one of the valid Top-k's for given D_S and k. Note that we always assume that $|W(D_S)| \geq k$, that is, there are sufficiently many distinct keywords in D_S to count.* ■

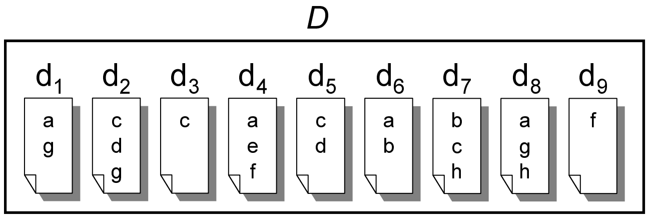


Figure 1: Example of D

Example 1. *Let $W = \{a, b, c, d, e, f, g, h\}$ and D is a document set containing 9 documents as shown in Figure 1. Here we use $a, b, \cdots$ as symbols for keywords, so actually a may be "food", b may be "investigate", and so on. The document id of d_i is i. Top-3 for D is $\{(a, 4), (c, 4), (g, 3)\}$. Let $D_S = \{d_1, d_2, d_4, d_7, d_8\}$, then Top-3 for D_S is $\{(a, 3), (g, 3), (c, 2)\}$. Note that $\{(a, 3), (g, 3), (h, 2)\}$ is another correct answer in the latter case because both c and h are eligible as the third most frequent keyword of Top-3.* ■

A naïve approach to compute Top-k is to count each keyword in each document of D_S with a static counting table that holds the frequency of each keyword that has been seen

[1]There are other methods to give weights to keywords such as the TF-IDF scheme. Refer to Section IV of [8] for details.

so far, and sort the entries of the table after all of the documents of D_S have been investigated.

However, this approach has two major drawbacks. First, it requires $O(|D_S|L)$ time complexity where L is the average size of a document (i.e. the number of keywords in the document) in D_S. This is often unacceptable for interactive analysis that requires a response time within a few seconds, especially when the number of documents of D_S is very large. For text analysis L may be from tens to several hundreds, so this term cannot be ignored. Second, in order to know the keywords in a specific document efficiently, the whole of a keyword-document matrix should be in-memory, because otherwise we cannot avoid slow random access to secondary storage many times (Note that D_S is dynamically determined by the users and we cannot precompute all of the possible answers).

2.2 Typical Scenarios

In the exploratory text analysis described in the introduction, the user initially identifies a “pivot” document set for analysis by using a search engine. For example, if the user wants to investigate descriptions about “cancer” in a large set of biomedical documents, the user can search for “cancer” in the entire set of documents to focus on the documents that contain “cancer” as a keyword. In this case D is the entire document set and $D_S \subset D$ is the document set obtained by searching for “cancer”. By running Top-k query processing on D_S, the user can find out which keywords frequently occur in D_S, in other words, which keywords frequently cooccur with “cancer” in D. Of course, each keyword to be counted is extracted from each document of D in advance, typically by using natural language processing techniques[8].

After the first run of Top-k query processing, the user may find some of the Top-k keywords interesting and may want to see what would happen if the search condition changes from “cancer” to “cancer AND (some of the newly found keywords)”. The user changes the search condition and reruns Top-k query processing against the corresponding new document set D_S. Alternatively, the user may want to remove some unnecessary keywords from the current search condition and re-run Top-k with the new search condition.

To provide an efficient way for users to analyze the document set interactively, the response times of Top-k must be close to those of search engines, limited to a few seconds. Although counting keywords itself is not computationally hard when it is done as batch processing, it is still a big challenge to realize a system that handles real-time Top-k queries for a large document set and a dynamically-given subset of the document set.

At the end of this section we note that the values of k typically used for exploratory text analysis are from tens to several thousands. In explorations of text documents, various kinds of keywords including functional words, common words, and symbols can be clues to discover valuable facts, patterns, or rules in a set of documents. For example, in a large collection of call logs, an exclamation mark in the description of inquiry from a customer may actually indicate that the customer complained about the slow response time and said “Move it!”. Hence it may be worth paying attention to the frequency of exclamation marks in the document set retrieved by a specific keyword to understand how often such problems were reported in association with the keyword. However, a human analyzer cannot know in advance which keywords will be useful for analysis, and so it is important to provide many frequent keywords to the analyzer without regard to the original meanings of the keywords so that the analyzer can scan them to find keywords of interest.

3. TOP-K KEYWORD AGGREGATION ON A SINGLE NODE

In this section, we briefly describe an underlying technique “Early-out” [18, 20], which is designed for efficient Top-k calculation on a single node. This technique plays a fundamental role in the implementation of the distributed processing we will discuss later. It finds the most frequent keywords in a dynamically given document set D_S, usually accessing only a small portion of the keywords of $W(D_S)$.

3.1 Early-Out for Pruning Infrequent Keywords

Here is the idea of early-out: We build an index structure to search for a given keyword w and to retrieve *freq*(D, w), the global frequency of w, with the posting list $P[w]$ for w. A posting list is a list of document ids sorted in ascending order of id. Also, we prepare a sorted list of keywords $w_1, \cdots, w_l$ in $W(D)$ so that we can access each keyword in descending order of its global frequency. For each $w \in W(D)$, *freq*(D_S, w) can be calculated by intersecting D_S with $P[w]$.

Algorithm 1. *Early-Out*

```
1: $P[w_1]$,$P[w_2]$,$\cdots$ $P[w_l]$ : the posting lists sorted by the global
   frequency of the keywords
2: $D_S$ : a (dynamically-given) document subset of D
3: $k$ : Top-k parameter
4:
5: Create an empty priority queue Q of capacity k
6: for $i = 1$ to $l$ do
7:     if size of Q equals k and freq$(D, w_i) \leq$ min. frequency in
       Q then
8:         break    // Early-Out
9:     end if
10:    $f_i = |P[w_i] \cap D_S|$    // computes freq$(D_S, w_i)$
11:    if size of Q is less than k then
12:        push $(w_i, f_i)$ into Q
13:    else if $f_i$ is greater than min. frequency in Q then
14:        pop an entry with min. frequency from Q
15:        push $(w_i, f_i)$ into Q
16:    end if
17: end for
18: return the entries in Q
```

Throughout the process of a query, we maintain a priority queue of capacity k for the final Top-k answer. When we check all of the posting lists then the queue should contain the correct Top-k most frequent keywords with their frequencies in D_S. We investigate the posting lists in the index structure one-by-one, $P[w_1]$, $P[w_2]$, $\cdots$. If the number of elements stored in the queue is less than k, or *freq*(D_S, w_i) is greater than the minimum frequency of the keywords in the queue, then $(w_i, \mathit{freq}(D_S, w_i))$ is pushed into the queue. If necessary, the least frequent keyword in the queue is removed from the queue before adding $(w_i, \mathit{freq}(D_S, w_i))$ to the queue. If *freq*(D, w_i) is not greater than the minimum frequency of the keywords in the queue and the number of elements stored in the queue is k, then none of w_i, w_{i+1}, $\cdots$ can be a Top-k most frequent keyword for D_S, and we can terminate the algorithm. Algorithm 1 describes the complete early-out algorithm. In addition, [18] introduces an effective way to reduce the computational cost of finding the approximate intersection of D_S and $P[w]$. Refer to [18] for details. Here we will analyze a method to calculate exact keyword frequencies.

4. DISTRIBUTED COMPUTING FOR TOP-K KEYWORD AGGREGATION

In this section, we describe our distributed computing approach for Top-k query processing. Let M be the central master node in the distributed environment and N_1, $\cdots$, N_n be worker nodes. M can exchange any kind of data with any N_i, but each N_i can communicate only with M. We can additionally assume that any node can take the role of M, and that all of the nodes can communicate with each other for fault-tolerance, but in this paper we will not discuss such kind of details of the implementation.

Top-k query processing is actively studied in both centralized and distributed settings and many algorithms have been proposed [7, 1, 5, 15, 2, 21]. We will discuss the possibilities of application of these approaches to our problems in Section 7.

4.1 Framework Overview

The framework for Top-k has two logical components. One is the search function and the other is the aggregate function.

The search function receives a search condition S and returns D_S, a list of document (integer) ids. In the following discussion we assume that the search function is provided by a standard keyword search engine and S is given as a set of keywords, although we can use any other definitions of S and a mapping $S \mapsto D_S$. We have already seen a typical example of the search functions in Section 2.2.

The aggregate function receives D_S and k as parameters, and returns the Top-k answers as defined in Definition 2. Here are the steps of the Top-k query processing:

1. For a given S, compute D_S by using the search function.

2. For D_S and k, compute Top-k by using the aggregate function.

3. Return the Top-k answers.

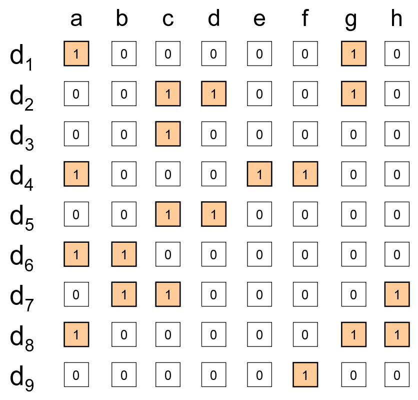
In the distributed environment, each query for Top-k is issued by the central master node M, each N_i receives and processes the query, N_i returns its local result to M, and M merges the results from N_i to produce the final output.

4.2 Two Partitioning Approaches

To get this framework to work in distributed computing, there are two possible approaches, the document partitioning approach and the keyword partitioning approach, which correspond to vertical partitioning and horizontal partitioning for distributed databases respectively [17]. We describe their features in this section. In this paper, we will discuss the techniques for improving the response times for our keyword partitioning approach, although comparison of these approaches is itself research for the future.

In distributed computing, network latency accounts for considerable overhead in response times. We will concentrate on a method that minimizes the number of communications, which is achieved by our keyword partitioning approach.

We again use the example dataset in Figure 1. The relationship between documents and keywords is represented as a matrix such as Figure 2, where the value in a cell (i, j) is 1 if and only if the document i contains the keyword j.



	a	b	c	d	e	f	g	h
d_1	1	0	0	0	0	0	1	0
d_2	0	0	1	1	0	0	1	0
d_3	0	0	1	0	0	0	0	0
d_4	1	0	0	0	1	1	0	0
d_5	0	0	1	1	0	0	0	0
d_6	1	1	0	0	0	0	0	0
d_7	0	1	1	0	0	0	0	1
d_8	1	0	0	0	0	0	1	1
d_9	0	0	0	0	0	1	0	0

Figure 2: A Document-Keyword Matrix

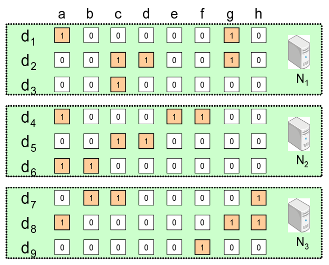


Figure 3: Document Partitioning Approach

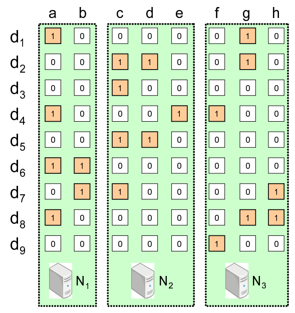


Figure 4: Keyword Partitioning Approach

4.2.1 *Document Partitioning Approach*

The document partitioning approach is a straightforward way to make multiple nodes count the keywords in D_S. In the phase of preprocessing, the whole document set D is partitioned into n subsets (where n is the number of worker nodes) such that each document of D belongs to one and only one N_i. When a query for Top-k is issued by M, search is done in each N_i and the local search result $(D_S)_i$ is obtained on N_i. Note that $\cup_i (D_S)_i = D_S$. Then on each N_i an aggregate function calculates Top-k for $(D_S)_i$. Early-out (Section 3) is available for the aggregate function on N_i. Finally each N_i returns its local result to M.

For example, Figure 3 depicts a situation where $n = 3$ and D is the document set in Figure 1. Each N_i has 3 documents[2] and no two worker nodes share a document in common. Next, let us assume that the search result is $D_S = \{d_1, d_2, d_4, d_7, d_8\}$. Then on each N_i the local Top-3 is calculated (for example N_1 returns $\{(g, 2), (a, 1), (c, 1)\}$), and N_i returns its local result to M. M decides on the most frequent 3 keywords from the total of 9 keywords that M received from all of the N_i.

This approach is clearly scalable with respect to the number of documents. However, the essential problem is that for an original query it cannot return a correct answer in two ways. Let us see this in Figure 3. Assume that N_1 returns $\{(g, 2), (c, 1), (d, 1)\}$, N_2 returns $\{(a, 1), (e, 1), (f, 1)\}$ and N_3 returns $\{(h, 2), (b, 1), (g, 1)\}$ as their local Top-3's for D_S. Then M merges these 9 keywords to produce $\{(g, 3)$, $(h, 2)$, $(e, 1)\}$ as the final Top-3 ($(e, 1)$ may be replaced by any other keyword of frequency one). In Example 1 we have already seen that the correct Top-3 is $\{(a, 3)$,$(g, 3)$,$(c, 2)\}$ or $\{(a, 3)$,$(g, 3)$,$(h, 2)\}$. This example shows that document partitioning may produce (i) incorrect keywords for Top-k (a is missing), or (ii) incorrect frequencies for correct Top-k keywords (even if M can correctly choose a as a Top-3 keyword, the frequency will be incorrect).

The frequency problem can be solved if M sends a request to each N_i about the keywords whose frequencies are unknown, and each N_i returns the frequencies of those keywords to M. Meanwhile, to ensure that M returns the correct Top-k keywords, on each N_i an aggregate function must calculate the Top-k' answers for $k' > k$ and return it to M, but we cannot determine a sufficient value of k' for correctness. Logically, this (and the frequency problem) can be solved by setting $k' = \infty$, but this is very inefficient because the aggregate function has to read all of the documents of D_S and the number of distinct keywords might be practically from millions to tens of millions (We will see this in Section 6).

4.2.2 Keyword Partitioning Approach

As the name implies, this approach splits the whole document set up by the keywords. First, before preprocessing D, the whole keyword set W is split into $W_1, \cdots, W_n$ such that $W = \cup_{i=1}^{n} W_i$ and $W_i \cap W_j = \emptyset$ if $i \neq j$, where n is the number of worker nodes. Although we do not know the set of all the keywords that appear in D before we preprocess D, we can logically split W by defining a rule for partitioning. Specifically, we can define $W_i := \{w \in W \mid (h(w) \bmod n) = i\}$ where h is any function that returns a non-negative integer value for a given keyword such as a standard hash function for the set of character strings.

Here is the strategy of the keyword partitioning approach: For a given D and $1 \leq i \leq n$, define a multiset D_i as

$$D_i := \{d \cap W_i | d \in D\}. \quad (1)$$

Note that $d \cap W_i$ can also be seen as a document, and $d = \cup_{i=1}^{n} (d \cap W_i)$ holds for any $d \in D$ from the construction of D_i. Then we can build on each N_i the Top-k index structure described in Section 3 for D_i. We will summarize in Appendix A the way to build the index structure.

[2]It is not essential in this example that each server has the same number of documents. In most cases it would be desirable for load-balancing of both preprocessing(data indexing) and query processing.

In Top-k query processing, first M runs its search function to obtain D_S. Since a search condition consists of keywords of W, it is sufficient for each N_i to perform a local search to retrieve the posting lists for the keywords in the search condition that belong to W_i. M can collect the posting lists from the relevant N_is to generate D_S. Since the function h for defining W_i is known, M only has to send queries to the N_is that are relevant to the search condition.

Next M sends D_S and k to all of the N_i. Each N_i receives the query parameters and performs Top-k for D_S by using its local index (built on D_i) to produce the local Top-k result. Note that the local Top-k result on N_i consists of only the keywords of W_i, and hence no two distinct local results have the same keyword. Finally, as with the document partitioning approach, M merges the local results from the N_is to produce the final Top-k. We summarize the algorithm of this aggregate function as Algorithm 2.

ALGORITHM 2. *Aggregation by the Keyword Partitioning Approach*

```
1: $M$ : the central master node
2: $N_1, \cdots, N_n$ : the worker nodes
3: $D_S$ : a document set obtained from the search function
4: $k$ : a parameter for the number of keywords
5:
6: Compute $T_i$ := Top-$k$ for $D_S$ and $k$ on each $N_i$
7: Merge $T_i$ $(i = 1, \cdots, n)$ on $M$ to obtain $T$, the set of the
   $k$ most frequent keywords
8: return $T$
```

EXAMPLE 2. *Figure 4 describes the keyword partitioning approach for the document set in Figure 1. The entire keyword set W is partitioned into 3 pieces, $\{a, b\}$, $\{c, d, e\}$ and $\{f, g, h\}$. For $d_1 \in D$, N_1 contains $d_1 \cap \{a, b\} = \{a\}$, N_2 contains $d_1 \cap \{c, d, e\} = \emptyset$, N_3 contains $d_1 \cap \{f, g, h\} = \{g\}$, and so on. When we calculate Top-2 for $D_S = \{d_1, d_2, d_4, d_7, d_8\}$, each N_i calculates its local Top-2 for $(D_S)_i$. The results are $\{(a, 3), (b, 1)\}$, $\{(c, 2), (d, 1)\}$ (or $\{(c, 2), (e, 1)\}$) and $\{(g, 3)$, $(h, 2)\}$ respectively. By merging these results and taking the most frequent 2 keywords, we can get the final answer $\{(a, 3), (g, 3)\}$.*

The next proposition ensures the correctness of keyword partitioning approach for exact calculation of Top-k.

PROPOSITION 1. *The keyword partitioning approach always gives a correct Top-k answer for any Top-k query.*

PROOF. We prove by contradiction. Let $T_i = \{(w_{i1}, f_{i1})$, $\cdots$, $(w_{ik}, f_{ik})\}$ $(f_{i1} \geq \cdots \geq f_{ik})$ be the local Top-k calculated on N_i, and $T = \{(w_1, f_1), \cdots, (w_k, f_k)\}$ $(f_1 \geq \cdots \geq f_k)$ be the final result merged on M. Since each w_j in T comes from a single N_i for some i, f_j, the frequency of w_j, is always correct. Suppose that $(w, f) \in T$ is not a correct keyword of Top-k. There must be at least one keyword $w' \neq w \in W(D_S)$ that is a correct Top-k keyword. If w' exists in one of T_i, then w' must be selected as a member of T when M merges $T_1, \cdots, T_n$, hence w' does not exist in any T_i. By the definition of the keyword partitioning approach, there exists (only one) i_0 such that $w' \in W_{i_0}$, and w' is not in the local Top-k result on N_{i_0}. This means that w' cannot become a member of a correct Top-k, which contradicts the assumption that w' is a correct Top-k keyword. $\square$

5. EFFECTIVE ESTIMATION OF THE NUMBER OF KEYWORDS TO COUNT

Our goal in this section is to substantially reduce the number of keywords that each worker node N_i $(i = 1, \cdots, n)$ returns to the central master node M. The keyword partitioning approach is efficient in that it requires only one-pass between the master node and the worker nodes for exact keyword aggregation. However, during query processing it may generate too many candidate keywords for Top-k most of which are eventually discarded. For example, let us consider $n = 32$ and $k = 100$. We want to know the 100 most frequent keywords, while the keyword partitioning approach always generates 3,200 keywords. In the merging phase, more than 96% $(= 100 \times (3100/3200))$ of the keywords sent from the workers will be thrown away, which is inefficient in both the query processing time in each worker node and the amount of communication between the worker nodes and the master node.

Let us consider that each N_i returns its local Top-t (instead of Top-k) for some $t \leq k$ and M merges nt keywords to produce the final Top-k. Actually, very small t compared to k is sufficient to have the correct Top-k answers with high probabilities as we shall see below. It is possible that M could not return a correct Top-k in such pathological cases as when *all* of the correct Top-k keywords exist in only one of N_is. However, the next proposition holds.

PROPOSITION 2. *Let t be an integer satisfying $t \leq k \leq nt$. Assume that each N_i returns $T_i = \{(w_{i1}, f_{i1}), \cdots, (w_{it}, f_{it})\}$ $(f_{i1} \geq \cdots \geq f_{it})$ to M and M merges nt keywords to decide T, the k most frequent keywords with their frequencies. If for each i there exists $j < t$ such that $(w_{ij}, f_{ij}) \in T$, $(w_{i,j+1}, f_{i,j+1}) \notin T$ and $f_{ij} > f_{i,j+1}$, then T is a correct Top-k.*

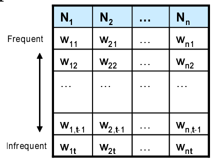


Figure 5: $n \times t$ **keywords from** $N_1, \cdots, N_n$

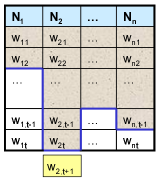


Figure 6: A case Top-k is not determined yet

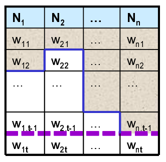


Figure 7: A case Top-k is determined

Let us analyze what Proposition 2 claims. Figure 5 shows the nt keywords that M gathers from $N_1, \cdots, N_n$. Each column corresponds to t keywords of T_i sorted by their frequencies as we defined in the statement of Proposition 2. Let us consider how the k most frequent keywords are chosen by M from these nt keywords. Figure 6 and 7 are examples of the k most frequent keywords, where those keywords are in gray cells. w_{11} and w_{12} are chosen as the k most frequent keywords, while $w_{13}, \cdots, w_{1t}$ are not chosen, and so on. In the case of Figure 6, all of the t keywords of T_2 are chosen as Top-k keywords. This means that the worker node N_2 has many frequent keywords with respect to the search condition S, and N_2 may have additional Top-k keywords besides those in T_2. Specifically, $w_{2,t+1}$ which is not in T_2 may be a correct Top-k keyword and replace one of the keywords of T. In this case we cannot determine that T is a correct Top-k. In contrast, in the case of Figure 7, for each N_i at most $(t-1)$ keywords are chosen as Top-k keywords. Then we can always get a correct Top-k result. We have to take care of keywords of equal frequency in order to give a strict proof. The next proof implies that it is essential to include the condition $f_{ij} > f_{i,j+1}$ in the assumption of the proposition.

PROOF OF PROPOSITION 2. For each N_i, take j as in the assumption of the proposition. Then $(w_{i,j+1}, f_{i,j+1})$ is not a correct Top-k keyword because otherwise it must be chosen as a member of T by M, which contradicts the assumption that $(w_{i,j+1}, f_{i,j+1}) \notin T$. Since T_i is the set of Top-t frequent keywords in N_i, this means that any Top-k keyword in $w \in W_i$ must be in T_i because otherwise $\mathit{freq}(D_S, w) \leq f_{it} \leq f_{i,j+1} < f_{ij} \leq$ (k-th frequency of keywords of T), which is a contradiction (Note that equality does not hold in the inequality). Hence any correct Top-k must be constructed from $T_1, \cdots, T_n$. $\square$

Proposition 2 gives a sufficient condition for getting a correct Top-k. This suggests a possibility for reducing the value of t, but there are two fundamental questions:

1. If T is not a correct Top-k, how much can we know about the correct Top-k from T?
2. For a given t, how likely do we get a correct Top-k?

For the first question, the next proposition gives a strict answer.

PROPOSITION 3. *Let T_i $(1 \leq i \leq n)$ and T be as in Proposition 2, and $I_0 := \{i | (w_{it}, f_{it}) \in T\}$. Assume that $I_0 \neq \emptyset$. Let $f' := max\{f_{it} | i \in I_0\}$. Then $T' := \{(w, f) \in T | f \geq f'\}$ is correct Top-$|T'|$ keywords and hence is a subset of a correct Top-k.*

PROOF. Let i' be an element of I_0 such that $f' = f_{i't}$. It is sufficient to show that $(w_{i't}, f_{i't}) = (w_{i't}, f') \in T_{i'}$ is a correct Top-k keyword. We prove by contradiction. Assume that $(w_{i't}, f')$ is not a correct Top-k keyword. For each $i \in I_0$, any correct Top-k keyword $w \in W_i$ must be in T_i because otherwise $\mathit{freq}(D_S, w) \leq f_{it} \leq f'$ means $(w_{i't}, f')$ is also a correct Top-k keyword, which contradicts the assumption. Also, for each $i \notin I_0$, if $(w_{ij}, f_{ij}) \in T_i$ is not in T then (w_{ij}, f_{ij}) cannot be a correct Top-k keyword because otherwise $f_{ij} \leq f'$ means $(w_{i't}, f')$ is also a correct Top-k keyword, which is a contradiction. Thus any correct Top-k keyword w must be in T. It follows that T is a correct Top-k because $|T| = k$, which contradicts $(w_{i't}, f') \in T$. $\square$

Next we consider the second question. What we want to do is to define a probability that the condition of Proposition 2 is satisfied. Such a probability should be defined as a function of n, k and t. Assuming that we can define the probability $P(n, k, t)$, Algorithm 3 is the improved version of Algorithm 2.

For the applications, it is desirable that a user can determine whether the output of Algorithm 3 gives a correct Top-k, and can determine which of the keywords in the output are correct if the output may include incorrect keywords. These are ensured by Proposition 2 and 3. If Algorithm 3 returns only T, then T is a correct answer. Otherwise, T' consists of only correct Top-k keywords and $T \setminus T'$ may contain one or more incorrect Top-k keywords.

We note that although we defined f_{ij} as the keyword frequency of w_i in the original problem setting, Proposition 2 and 3 do not require any conditions on f_{ij} other than $f_{i1} \geq \cdots \geq f_{it}$. In particular, f_{ij} need not be an integer value or a positive value. This means that Algorithm 3 is sufficiently general such that we can apply it to arbitrary keyword scorings.

Algorithm 3. *Aggregation by using Estimation*

```
1: $M$ : the central master node
2: $N_1, \cdots, N_n$ : the worker nodes
3: $D_S$ : a document set obtained from the search function
4: $k$ : a parameter for the number of keywords
5: $\alpha$ : a threshold for $P(n,k,t)$, $0 < \alpha < 1$
6:
7: Precompute $t$ satisfying $P(n,k,t) \geq \alpha$
8: Compute $T_i :=$ Top-$t$ for $D_S$ and $k$ on each $N_i$
9: Merge $T_i$ $(i = 1, \cdots, n)$ on $M$ to obtain $T$, the set of
   the $k$ most frequent keywords
10: if $T_i$ and $T$ satisfy the condition of Proposition 2 then
11:     return $T$
12: else
13:     return $T$ and $T'$ in Proposition 3
14: end if
```

In the next subsections we show that we can define two kinds of combinatorial probabilities for $P(n,k,t)$ that adequately describe the sufficient condition that Proposition 2 holds. Our idea for defining $P(n,k,t)$ is to use the number of combinations, where each combination represents a way to choose k keywords from the nt keywords collected from $N_1, \cdots, N_n$. In the next subsection we introduce a method using histograms, and then we describe a method using keyword rank combinations.

5.1 The Histogram Method

The histogram method uses all of the possible histograms, each of which corresponds to a way to choose the k most frequent keywords from nt keywords. Let us see an example of the choice of k keywords from the nt keywords. Assume that $n = 3, k = 100, t = 50$. If a worker node N_i $(i = 1, 2, 3)$ eventually provides x_i keywords from T_i (the local Top-50 on N_i) for the most frequent 100 keywords respectively, then $x_1 + x_2 + x_3 = 100$ must hold. Note that $0 \leq x_i \leq 50$ for any i because x_i keywords are chosen from T_i and $|T_i| = t = 50$.

It is then natural to consider how many such combinations (x_1, x_2, x_3) exist and how many of them satisfy the condition of Proposition 2. Since a Top-k answer (and each local Top-t answer) can contain keywords of equal frequency, the condition $f_{ij} > f_{i,j+1}$ in the assumption of Proposition 2 requires us to take into consideration the frequency of each keyword. However, it is actually rare that $f_{ij} = f_{i,j+1}$ holds for some i, so we try to estimate the probability without this condition. This enables us to determine the probability by using only information about the choice of keywords, and the estimation does not depend on frequencies of keywords. We will see in Section 6 that this assumption is practically reasonable. The discussion so far motivates the following definitions.

Definition 3. *Let $n \in \mathbf{N}$ and $k,\ t \in \mathbf{Z}_{\geq 0}$. We define $M(n,k,t) := \{(x_1, \cdots, x_n) \in \mathbf{Z}^n \mid x_1 + \cdots + x_n = k, 0 \leq x_i \leq t\ (1 \leq \forall i \leq n)\}$ and $f(n,k,t) := |M(n,k,t)|$. We define the probability $P_f(n,k,t) := f(n,k,t-1)/f(n,k,t)$.* ■

We have already seen that $f(n,k,t)$ is the number of possible assignments of the number of Top-k keywords to each of the n worker nodes. First let us see that $P(n,k,t)$ actually represents a probability, that is, $f(n,k,t-1) \leq f(n,k,t)$ for any t. By definition of $M(n,k,t)$, any element $(x_1, \cdots, x_n) \in M(n,k,t-1)$ is also an element of $M(n,k,t)$. Hence $M(n,k,t-1) \subset M(n,k,t)$ and $f(n,k,t-1) \leq f(n,k,t)$ follows. By considering the range of x_1 which corresponds to the number of Top-k keywords provided by the first worker of n workers, $f(n,k,t)$ satisfies this recurrence formula:

$$\begin{aligned} f(n,k,t) &= \sum_{i=0}^{\min\{k,t\}} f(n-1, k-i, t) \quad (n > 1), \\ f(1,k,t) &= \begin{cases} 1 & (t \geq k) \\ 0 & (t < k) \end{cases} \end{aligned}$$

This shows that we can compute $f(n,k,t)$ by using the values of $f(n',k',t)$ for $n' < n$ and $k' < k$. Algorithm 4 based on dynamic programming computes $f(n,k,t)$.

Algorithm 4. *Dynamic programming for $f(n,k,t)$*

```
1: $a, b$ : integer arrays of length $k+1$ respectively
2: $a[1] = 1,\ a[i] = 0\ (i > 1),\ b[i] = 0\ (i \geq 1)$
3: for $i = 1$ to $n$ do
4:     $b[1] = a[1]$
5:     for $j = 2$ to $k+1$ do
6:         $s := b[j-1] + a[j]$
7:         if $j - t - 1 > 0$ then
8:             $s = s - a[j-t-1]$
9:         end if
10:        $b[j] = s$
11:    end for
12:    swap$(a, b)$
13: end for
14: return $a[k+1]$
```

It is easy to see that the time complexity of the algorithm is $O(nk)$. We can calculate $f(n,k,t)$ by using Algorithm 4, but in fact $f(n,k,t)$ can be expressed in an explicit form:

Proposition 4. *If $k \leq nt$, $f(n,k,t)$ is given by*

$$f(n,k,t) = \sum_{i=0}^{\lfloor \frac{k}{t+1} \rfloor} (-1)^i \binom{n}{i} \binom{n-1+k-i(t+1)}{n-1} \qquad (2)$$

where $\binom{a}{b} := a(a-1)\cdots(a-b+1)/b!$ is the binomial coefficient.

Proof. Here we give a sketch of the proof due to space limitation. First we note that the assumption that $k \leq nt$ is reasonable because otherwise the sum $x_1 + \cdots + x_n$ cannot be k and $f(n,k,t)$ is trivially zero. Remember that $f(n,k,t) = |M(n,k,t)|$. If $t \geq k$, then it is easy to see that $f(n,k,t)$ equals the number of elements of

$$\begin{aligned} M(n,k) \quad := \quad & \{\ (x_1, \cdots, x_n) \in \mathbf{Z}^n \mid \\ & x_1 + \cdots + x_n = k,\ 0 \leq x_i\ \}, \end{aligned} \qquad (3)$$

and it is well known that

$$f(n,k,t) = |M(n,k)| = \binom{n+k-1}{n-1}. \quad (4)$$

To compute $f(n,k,t)$ for general t, let us consider the elements of $M(n,k) \setminus M(n,k,t)$. For each $(x_1, \cdots, x_n) \in M(n,k) \setminus M(n,k,t)$, at least one of x_i must be greater than t. Let $M(n,k,t;i) := \{ (x_1, \cdots, x_n) \in M(n,k) \mid x_i > t \}$. Then

$$M(n,k,t) \;=\; M(n,k) \setminus \bigcup_{i=1}^{n} M(n,k,t;i)$$

and so

$$f(n,k,t) \;=\; \binom{n+k-1}{n-1} - \left| \bigcup_{i=1}^{n} M(n,k,t;i) \right| \quad (5)$$

holds. From the inclusion-exclusion principle, we get

$$\left| \bigcup_{i=1}^{n} M(n,k,t;i) \right| = \sum_{i=1}^{n} |M(n,k,t;i)|$$

$$- \sum_{i<j} |M(n,k,t;i,j)| + \sum_{i<j<l} |M(n,k,t;i,j,l)|$$

$$- \cdots + (-1)^n |M(n,k,t;1,\cdots,n)| \quad (6)$$

where $M(n,k,t;i_1,\cdots,i_p) := \{(x_1,\cdots,x_n) \in M(n,k,t) \mid x_{i_j} > t,\ j = 1,\cdots,p\} = \bigcap_{j=1}^{p} M(n,k,t;i_j)$.

For each $\{i_1,\cdots,i_p\} \subset \{1,\cdots,n\}$, if $M(n,k,t;i_1,\cdots,i_p) \neq \emptyset$ (or equivalently, $k \geq p(t+1)$) then there exists a one-to-one correspondence between $M(n,k,t;i_1,\cdots,i_p)$ and $M(n, k - p(t+1))$ because, for example, $(x_1,\cdots,x_p,\ x_{p+1},\cdots,x_n) \in M(n,k,t;1,\cdots,p)$ bijectively corresponds to $(x_1 - t - 1, \cdots,\ x_p - t - 1,\ x_{p+1},\ \cdots,\ x_n) \in M(n, k - p(t+1))$.

By using (4), the right-hand side of (6) equals

$$\sum_{p=1}^{\lfloor \frac{k}{t+1} \rfloor} (-1)^{p-1} \binom{n}{p} \binom{n+k-p(t+1)-1}{n-1} \quad (7)$$

By substituting (7) to (5) we get the explicit form (2). Note that the sum in (7) is taken for $1 \leq p \leq \lfloor \frac{k}{t+1} \rfloor$ because $M(n,k,t;i_1,\cdots,i_p) \neq \emptyset \iff k \geq p(t+1) \iff p \leq \lfloor \frac{k}{t+1} \rfloor$ and $\lfloor \frac{k}{t+1} \rfloor \leq n$ since we assume that $k \leq nt$. $\square$

By using this formula, we can calculate $f(n,k,t)$ directly by substituting the values of n, k and t. Note that $f(n,k,t)$ is the coefficient of x^k of a polynomial $(1 + x + x^2 + \cdots + x^t)^n$. However, the formula of Proposition 4 does not follow straightforwardly from this fact.

Next let us consider the time complexity for the computation of $f(n,k,t)$. The time complexity for the computation of $\binom{a}{b}$ is $O(b)$, so the time complexity for the calculation of equation (2) is $O(np'^2)$ where $p' = \lfloor \frac{k}{t+1} \rfloor$. Therefore it is more efficient to calculate $f(n,k,t)$ than the dynamic programming approach when p'^2 is smaller than k, which usually holds for practical values of k and t.

For our application we want to find a value of t such that $P_f(n,k,t) := f(n,k,t-1)/f(n,k,t) > \alpha$ for given n, k, and the threshold α. We can calculate such t in advance because there are at most finite values of t such that $f(n,k,t)$ is non-trivial ($\lceil \frac{k}{n} \rceil \leq t \leq k$). Note that if $t \geq k$ then $P_f(n,k,t) = 1$. Moreover, if $P_f(n,k,t)$ is a monotonic function of t for given n and k, then we can use a binary search to find the value of t much more efficiently. We confirmed that for many pairs of (n,k) the following property holds, although it is not yet proved:

Conjecture 1. *For given n and k, $P_f(n,k,t)$ is monotonically increasing function of t, that is, $P_f(n,k,t) \leq P_f(n,\ k,\ t+1)$ for any $t > 0$.*

This can be rewritten in terms of combinatorics. A sequence $a_1, \cdots, a_m$ of non-negative numbers is said to be **log-concave** if we have $a_i^2 \geq a_{i-1}a_{i+1}$ for all $1 < i < n$ [19]. Then Conjecture 1 can be rewritten as:

Conjecture 2. *For given n and k, $f(n,k,1)$, $f(n,k,2)$, $\cdots$, $f(n,k,k)$ is a log-concave sequence.*

Log-concavity of a sequence of real numbers has been long studied and will be reviewed in Section 7.

5.2 The Rank Method

The definition of $P_f(n,k,t)$ uses only the frequencies of the keywords chosen from each T_i. Another way to define the number of possible ways to choose k keywords is to take the rank of each of the k keywords into consideration.

Again let us assume that $n = 3, k = 100, t = 50$, and 100 "ranked" keywords are chosen from 150 keywords. The 100 keywords are now ordered by the rank in the Top-k such that $w_1 \geq w_2 \geq \cdots \geq w_{100}$. Each w_p is chosen from one of T_i.

Definition 4. *$g(n,k,t)$ is the number of ways to put k distinct balls into n distinct boxes where the capacity of each box is t. We define $P_g(n,k,t) := g(n,k,t-1)/g(n,k,t)$.* ■

In Definition 4, a ball corresponds to a keyword of the k most frequent keywords, and a box corresponds to a worker node. When we put i balls into the first box, then there are $g(n-1, k-i, t)$ ways to put the rest of the balls. Thus $g(n,k,t)$ satisfies this recurrence formula:

$$g(n,k,t) \;=\; \sum_{i=0}^{\min\{k,t\}} \binom{n}{i} g(n-1, k-i, t) \quad (n > 1),$$

$$g(1,k,t) \;=\; \begin{cases} 1 & (t \geq k) \\ 0 & (t < k) \end{cases}$$

By using this formula, $g(n,k,t)$ can be computed with dynamic programming. Algorithm 5 shows the details of the computation. The time complexity of this algorithm is $O(nkt)$. We state a conjecture corresponding to Conjecture 1.

Conjecture 3. *For given n and k, $P_g(n,k,t)$ is a monotonically increasing function of t. In other words, $g(n,k,1)$, $g(n,k,2)$, $\cdots$, $g(n,k,k)$ is a log-concave sequence.*

$g(n,k,t)$ is much more complicated to compute than $f(n,k,t)$. To the best of our knowledge, no explicit formula such as (2) is yet known.

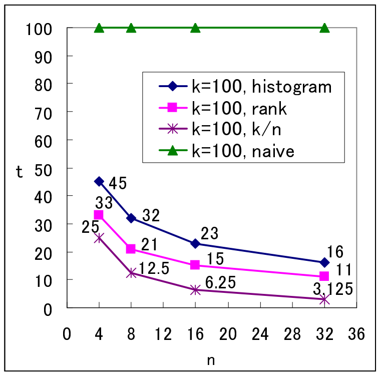


Figure 8: Case $(\alpha, k) = (0.9, 100)$

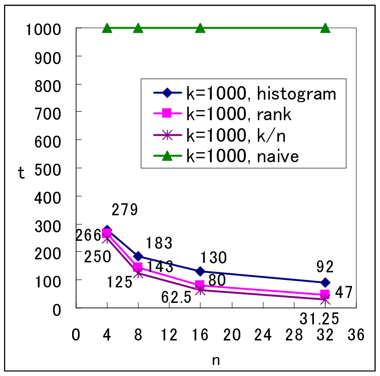


Figure 9: Case $(\alpha, k) = (0.9, 1000)$

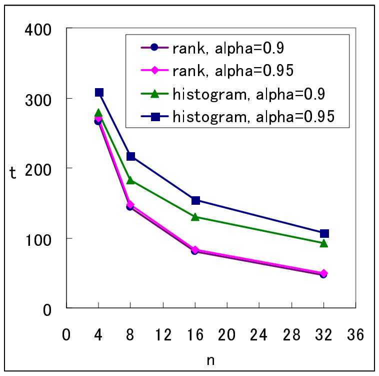


Figure 10: Comparison of $\alpha = 0.9$ and 0.95 $(k = 1000)$

Algorithm 5. *Dynamic programming for $g(n, k, t)$*

```
1: $a, b$ : integer arrays of length $k + 1$ respectively
2: $a[1] = 1,\ a[i] = 0\ (i > 1),\ b[i] = 0\ (i \geq 1)$
3: for $m = 1$ to $n$ do
4:   for $j = 1$ to $k + 1$ do
5:     for $s = 0$ to $t$ do
6:       if $j - s > 0$ then
7:         $b[j] = b[j] + \binom{m}{s} a[j - s]$
8:       else
9:         $b[j] = 0$
10:      end if
11:    end for
12:  end for
13:  swap$(a, b)$
14: end for
15: return $a[k + 1]$
```

5.3 Comparison of Two Counting Methods

Let us compare the two methods in Section 5.1 and 5.2. Since it is very difficult to compare analytically the properties of $P_f(n, k, t)$ and $P_g(n, k, t)$, here we show with examples that the rank method produces a smaller value for t than the histogram method.

Figures 8 and 9 show the values of t for each of the following four cases: (1) $P_f(n, k, t) \geq 0.9$, (2) $P_g(n, k, t) \geq 0.9$, (3) $t = k/n$, and (4) $t = k$. For example, $t = 45$ is the minimum value of t that satisfies $P_f(4, 100, t) \geq 0.9$, which means that the histogram method tells that the system returns a correct Top-100 answer with (at least) 90% probability when each of the 4 worker nodes returns 45 keywords to the central master node. $t = k/n$ is the "luckiest" case that we obtain a correct Top-k result, and $t = k$ is the condition that is necessary and sufficient for the naïve method (Algorithm 2) to work. It is easy to see from Figures 8 and 9 that as a function of n, t is monotonically decreasing in all cases. Also, Figure 10 shows the different values of t with $\alpha = 0.9$ and $\alpha = 0.95$ for both the histogram and the rank methods. The histogram method is more sensitive to the value of α.

Both of these methods have dramatic effects in reducing the number of keywords that each worker returns to the central master. For example, with $n = 32$ and $k = 100$, the naïve method requires 100 keywords from each worker, while the histogram method requires 16 keywords, and when it comes to the rank method, only 11 keywords.

The value of t of the rank method is always smaller than that of the histogram method. The difference of the values depends on n and k. When $n = 32$ and $k = 1{,}000$, the value of the rank method ($t = 47$) is about 51% of that of the histogram method ($t = 92$). Generally speaking, the histogram method estimates the value of t more conservatively than the rank method, which will be confirmed in Section 6 as we show that the histogram method always gave correct answers in all of the test cases.

6. EXPERIMENTAL EVALUATION

In this section we present the results of experimental evaluations of our theoretical analysis. Since we have already seen in Section 5.3 that our method can significantly reduce the number of keywords that each worker node has to calculate, the major goal of this section is to examine the effectiveness of our method in a practical application.

6.1 Setup

System Implementation: We implemented our evaluation system on a network with 1 central master node and 32 worker nodes. The central master node has 6 GB memory and 3.0 GHz $\times$ 4 CPUs, and is running 64-bit KVM Linux. Each worker node has 2 GB memory and 3.0 GHz $\times$ 2 CPUs, and is running 64-bit KVM Linux. Both of preprocessing and query processing are written in Java 6. For a hash function h (Section 4.2.2) we used java.lang.String#hashCode(), the standard implementation of hash code for strings in Java.

Data: We used MEDLINE [14] which is a public collection of medical documents with over 18 million references to journal articles. Each document has its abstract and various kinds of metadata such as the title, authors, publication year, and technical terms related to the document. We preprocessed the abstract and the title of each document by using natural language processing to generate keywords, and

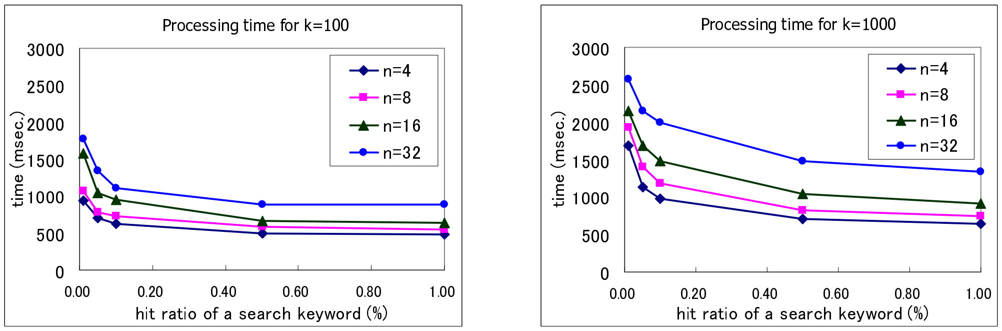


Figure 11: Processing times of the naïve method varying the size of D_S and n (Left: $k = 100$, Right: $k = 1000$)

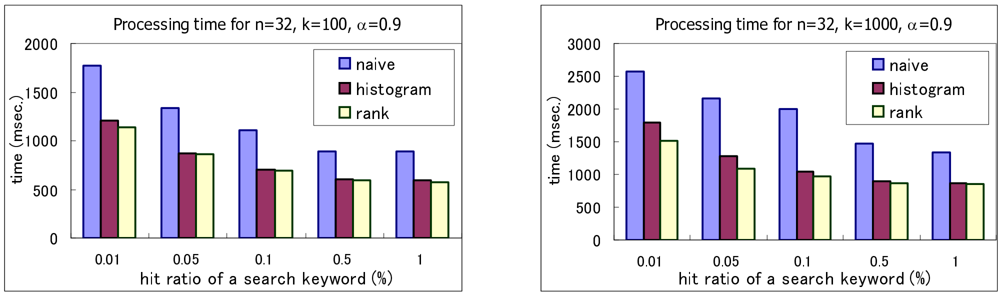


Figure 12: Effectiveness of Estimation Methods for $n = 32$ and $\alpha = 0.9$ (Left: $k = 100$, Right: $k = 1000$)

created the index structure on each worker node by using Hadoop-based batch processing (See Appendix A).

To evaluate the performance of query processing for different numbers of worker nodes, with $n \in \{4, 8, 16, 32\}$, we distributed approximately[3] 425,000 $\times$ n documents to the n worker nodes according to the keyword partitioning approach. For example, when $n = 32$ total 13,600,000 documents were stored in 32 nodes. Note that the number of documents was determined to be linear to n so that we could check the system was scaling out. The number of keywords in W, the set of all the keywords that appear in the document set, is 89,093,177 when $n = 32$. We did not use any kind of filtering methods for the keywords, so common words, functional terms, proper nouns and compound nouns with low frequencies were all included in $W(D)$ (See Section 2.2 for the explanation).

Queries: We prepared keyword search queries for various search hit ratios. For each of the hit ratios $1\%, 0.5\%, 0.1\%, 0.05\%$, and 0.01%, we chose 3 distinct keywords so that each of the keywords gave a search result D_S approximating that hit ratio. In the experimental results we averaged the processing times of 3 keyword queries for each search hit ratio. Our processing time is the total processing time including search and aggregation, and we confirmed that the time for searches was much less than that for aggregation in all of the test cases.

[3]The reason that we did not split the document set in bunches of equal size is that the length of text in a document varies. We prepared the dataset so that the number of 1 in the document-keyword matrix (not the number of documents) was almost linear to n.

6.2 Scalability

Figure 11 shows the processing time for Algorithm 2 (the naïve keyword partitioning approach) for each search hit ratio, with $n \in \{4, 8, 16, 32\}$ and $k \in \{100, 1000\}$. We found that, for all values of (n, k), the processing times increase as the search hit ratio decreases. The reason is that the early-out algorithm on each worker node has this property. As the search hit ratio (that is, $|D_S|$) decreases, more keywords with low frequencies have a chance of being Top-k keywords, and so the early-out algorithm (Algorithm 1) has to scan more posting lists. The evaluation system scaled out well, considering that the overhead to merge keywords at the central master is increasing as n is increasing.

6.3 Effectiveness of Estimation

We compared the runtime performance of our methods (the histogram method and the rank method) with the baseline naïve method. Figure 12 shows the processing time of each method for various search hit ratios and $k \in \{100, 1000\}$ on $n = 32$ worker nodes. Here we used $\alpha = 0.9$ for estimating the probabilities.

We found that our methods are effective in speeding up the query processing. The histogram method performed 1.47 to 1.90 times faster than the naïve method, and the rank method was 1.49 to 2.07 times faster than the naïve method, depending on the search hit ratio and k. We have already seen that our estimation methods reduce the number of keywords from each worker node from half to less than one-tenth of k. The actual processing time was not proportional to the value of t. This is understood as follows. Most of the processing time of the early-out algorithm on each worker node is occupied by the processing of the posting lists of the most globally frequent keywords. For example, in the

Table 1: Correctness of Answers

α	k	Histogram (%)	Rank (%)
0.9	100	100	95.2
0.9	1000	100	96.6
0.95	100	100	97.6
0.95	1000	100	98.4

case that $\alpha = 0.9$ and the search hit ratio is 1%, the average of the number of keywords that the early-out algorithm has read during the processing on each worker was 3,893 when $t = 100$, and was 504 when $t = 11$ that is less than 13% of the amount for $t = 100$. On the other hand, the average of the number of document ids that the early-out algorithm has read from the posting lists during the processing on each worker was 32,409,617 when $t = 100$, and was 26,451,484 when $t = 11$ that is as much as 82% of the amount for $t = 100$. This overhead of reading the posting lists of the most globally frequent keywords explains the gap between the reduction rate of the number of keywords and the effect of speeding up. For the same reason there are not meaningful differences between the histogram method and the rank method with respect to the processing time.

For the correctness of the answers, we ran 500 random search keyword queries with $n = 32$ and $k \in \{100, 1000\}$. The results are shown in Table 1. The histogram method returned correct answers for all of the queries, which indicates that the histogram method behaves conservatively as we expected in Section 5.3. In the case of the rank method, it returned answers with slightly higher probabilities than the estimation calculates. Also, we confirmed that the average of the number of the correct Top-k keywords in each answer that is not a correct Top-k answer was 97.9, 98.1, 994.7 and 996.2 for $(\alpha, k) = (0.9, 100)$, $(0.95, 100)$, $(0.9, 1000)$ and $(0.95, 1000)$ respectively. These results suggest that our estimation methods always provide Top-k answers with satisfactory accuracy.

7. RELATED WORK

Top-k query processing has been intensively studied because it has various kinds of applications including information retrieval, similarity search for multimedia databases, and business intelligence. In particular, the threshold algorithm (TA) [7] and its variants [1, 2] have been proposed. The kernel of TA-based methods is that, during query processing, they compare a lower bound of the score to the current Top-k elements to be aggregated, and an upper bound of the score to the other candidate elements that have not been investigated yet, enabling "early-out" before all of the elements in the index are scanned. TA and its variants assume that random access and/or sorted access to each scored object in each list (worker node) are available, but in distributed settings the repeated use of these access methods causes prohibitive overhead between the master node and worker nodes.

Distributed Top-k processing addresses the situations where data is fragmented over then network and neither random access nor sorted access are performed at a small cost. [5, 21] efficiently estimate the lower bound of the k-th score of the correct Top-k by collecting the local Top-k answer from each worker node and determine which unseen objects have to be additionally collected from the worker nodes. The naïve keyword partitioning approach we introduce in Section 4.2.2 can be seen as a special case of these algorithms when each scored object belongs to only one of the worker nodes.

Most of the previous studies assume that, at least on each worker node, the score of each object is readily available regardless of the kind of access methods. That is, the score is in advance stored in the list, or can be easily calculated. In our problem settings this assumption is not true because the rank of a certain keyword with respect to keyword frequencies depends on the frequencies of other keywords, which are not easily determined because the underlying document set D_S is dynamically given. That is why we adopted the early-out algorithm [18, 20] for efficient calculation of local Top-k keyword aggregation on each worker node.

Dynamic cost estimation for efficient processing is another major topic of Top-k research. [2] proposed an effective scheduling method for sequential scans and random lookups of the underlying index structure, based on the statistics that is available from the collected data during the query processing. Our method also estimates the probability that the central master node successfully obtain a correct Top-k result without communicating with the worker nodes again. [10] classifies the techniques for Top-k processing from many different dimensions.

Log concavity of a sequence of real numbers has long been studied in combinatorics. It is known that in general it is difficult to prove that a certain sequence is log-concave. Unimodality is another property of a sequence that is a necessary condition for log-concavity under an additional hypothesis, and hence it has also been thoroughly studied along with log-concavity. A sequence of real numbers $a_1, \cdots, a_n$ is said to be unimodal if there exists $1 \leq j < n$ such that $a_1 \leq a_2 \leq \cdots \leq a_j \geq a_{j+1} \geq \cdots \geq a_n$. It is easy to see that a log-concave sequence is unimodal if all the elements in the sequence are positive, but the converse does not hold. $f(n, k, t)$ we defined in Section 5.1 trivially forms a unimodal sequence $f(n, k, 1), f(n, k, 2), \cdots$ because $0 = f(n, k, 1) = f(n, k, 2) = \cdots = f(n, k, p-1) < f(n, k, p) \leq \cdots \leq f(n, k, k-1) \leq f(n, k, k) = f(n, k, k+1) = \cdots$, where p=$\lfloor k/n \rfloor$. Note that what we want to prove is that $f(n, k, 1), f(n, k, 2), \cdots$ is *log-concave* (Conjecture 2). In contrast, as a sequence indexed by k, $f(n, 1, t), f(n, 2, t), \cdots$, is log-concave. This follows from the fact that $f(n, k, t)$ is the coefficient of x^k in $(1 + x + \cdots + x^t)^n$, and the next theorem. Note that a polynomial is log-concave if its coefficients form a log-concave sequence.

THEOREM 1 ([11, 12]). *Let $h_1(x)$ and $h_2(x)$ be log concave polynomials with non-negative coefficients and no internal zero coefficients. Then the product $h_1(x)h_2(x)$ is also log-concave.*

[19] surveys a variety of methods to prove that a sequence is log-concave or unimodal. A typical example of a combinatorial proof for log-concavity of an integer sequence is found in [3]. [4] gave another formula for $f(n, k, t)$ for the transformation of a generating series of a sequence of complex numbers, although the formula is not an explicit form. To the best of our knowledge, there have been no studies on the log-concavity of $g(n, k, t)$ (Section 5.2).

8. CONCLUSIONS

We presented a framework for Top-k keyword aggregation in distributed computing based on the keyword partitioning approach. Our novel estimation method can determine the number of keywords that each worker must return to the central master in order to obtain a correct Top-k answer with high probability. We focused on the number of ways to choose the k most frequent keywords from nt keywords sent from the worker nodes, and defined the probability that the aggregation algorithm can terminate with a correct answer. We gave two different kinds of definitions for the probability, whose calculations are reduced to purely combinatorial problems. Also, we introduced some interesting properties of these problems including the conjectures for log-concavity of $f(n, k, t)$ and $g(n, k, t)$. Experimental evaluation with real data showed that our method was effective in practical applications for speeding up the Top-k query processing. The histogram method is more conservative than the rank method with respect to the number of keywords, but in most cases there was no significant difference in the runtime performance. We showed that this comes from the property of the early-out algorithm running on each worker node.

Our future work will involve two directions. First, the estimation methods we presented in this paper work well in more general settings where each worker node returns a list of objects with scores to the central master node. Algorithm 3 is applicable to any real-valued scores. It is interesting to find application scenarios using our method beyond text analysis. Second, there are some open mathematical questions about the combinatorial properties of $f(n, k, t)$ and $g(n, k, t)$. Log-concavity is worth considering because if it holds for $f(n, k, t)$ or $g(n, k, t)$ we can calculate appropriate values of t much more efficiently.

9. REFERENCES

[1] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for top-k queries. In *VLDB*, pages 495–506, 2007.

[2] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[3] M. Bona and R. Ehrenborg. A combinatorial proof of the log-concavity of the numbers of permutations with k runs. *Journal. of Combinatorial Theory, Series A*, 90:293–303, 2000.

[4] F. Brenti and V. Welker. The veronese construction for formal power series and graded algebras. *Advances in Applied Mathematics*, 42(4):545–556, 2009.

[5] P. Cao and Z. Wang. Efficient top-k query calculation in distributed networks. In *PODC*, pages 206–215, 2004.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[8] R. Feldman and J. Sanger. *The Text Mining Handbook - Advanced Approaches in Analyzing Unstructured Data.* Cambridge University Press, 2007.

[9] Hadoop. http://hadoop.apache.org/.

[10] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[11] S. Karlin. *Total Positivity, Vol. I.* Stanford University Press, 1968.

[12] W. Kook. On the product of log-concave polynomials. *Electronic Journal. of Combinatorial Number Theory*, 6, 2006.

[13] J. Liang, K. Koperski, T. Nguyen, and G. B. Marchisio. Extracting statistical data frames from text. *SIGKDD Explorations*, 7(1):67–75, 2005.

[14] MEDLINE. http://www.nlm.nih.gov/pubs/factsheets/medline.html.

[15] S. Michel, P. Triantafillou, and G. Weikum. Klee: A framework for distributed top-k query algorithms. In *VLDB*, pages 637–648, 2005.

[16] T. Nasukawa and T. Nagano. Text analysis and knowledge mining system. *IBM Systems Journal*, 40(4):967–984, 2001.

[17] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems, 2nd Edition.* Prentice-Hall, 1999.

[18] A. Simitsis, A. Baid, Y. Sismanis, and B. Reinwald. Multidimensional content exploration. *PVLDB*, 1(1):660–671, 2008.

[19] R. P. Stanley. Log-concave and unimodal sequences in algebra, combinatorics, and geometry. *Annals of the New York Academy of Sciences*, 576:500–535, 1989.

[20] D. Takuma and I. Yoshida. Top-n keyword calculation on dynamically selected documents. *IBM Research Report*, RT-0760, 2007.

[21] D. Zeinalipour-Yazti, Z. Vagena, D. Gunopulos, V. Kalogeraki, V. J. Tsotras, M. Vlachos, N. Koudas, and D. Srivastava. The threshold join algorithm for top-k queries in distributed sensor networks. In *DMSN*, pages 61–66, 2005.

APPENDIX

A. INDEX BUILDING FOR THE KEYWORD PARTITIONING APPROACH

For efficient query processing by the keyword partitioning approach, we consider building on each N_i the index structure used for early-out (see Section 3). Since it is sufficient for each N_i to have $\{(w, d) | w \in W_i, d \in D, w \in d\}$ for index building on N_i, we can apply the MapReduce[6] programming model to distribute all of the input data (that is, D) to $N_1, \cdots, N_n$, by using an arbitrary number of mappers and n reducers. In the map phase, each document of D is given as input to a mapper in the form of a key-value pair $(j, \{w_1, \cdots, w_p\})$ where j is a document id and $w_i \in d_j$ $(i = 1, \cdots, p)$. For each w_i the mapper emits a key-value pair $(h(w_i), (w_i, j))$ where h is the hash function to define the partitioning of W. Then each reducer receives $(h(w), (w, j))$ satisfying $h(w) = h$ for some $0 \leq h \leq n-1$, and hence the reducer can collect all the pairs for h. In the actual implementation using Hadoop[9], an open-source library for MapReduce, we can use (w, j) instead of $(h(w), (w, j))$ for the intermediate key-value format, because we can specify the hash function that is used internally in the shuffle phase (between the map phase and the reduce phase) so that each reducer can receive only the (w, j) that the reducer should process.