LEMP: Fast Retrieval of Large Entries in a Matrix Product

Christina Teflioudi
Max Planck Institute for Informatics
Saarbrücken, Germany
chteflio@mpi-inf.mpg.de

Rainer Gemulla
University of Mannheim
Mannheim, Germany
rgemulla@uni-mannheim.de

Olga Mykytiuk
Sulzer GmbH
München, Germany
olga.mykytiuk@gmail.com

ABSTRACT

We study the problem of efficiently retrieving large entries in the product of two given matrices, which arises in a number of data mining and information retrieval tasks. We focus on the setting where the two input matrices are tall and skinny, i.e., with millions of rows and tens to hundreds of columns. In such settings, the product matrix is large and its complete computation is generally infeasible in practice. To address this problem, we propose the LEMP algorithm, which efficiently retrieves only the large entries in the product matrix without actually computing it. LEMP maps the large-entry retrieval problem to a set of smaller cosine similarity search problems, for which existing methods can be used. We also propose novel algorithms for cosine similarity search, which are tailored to our setting. Our experimental study on large real-world datasets indicates that LEMP is up to an order of magnitude faster than state-of-the-art approaches.

1 Introduction

Low-rank matrix factorization methods, such as singular value decomposition (SVD), non-negative matrix factorization (NMF), or latent-factor models, have recently gained traction for a number of prediction tasks [1]. In the context of recommender systems, for example, latent-factor models are a popular and successful approach for predicting the preference of users for items from available feedback; see [2] for an excellent overview. Fig. 1a shows a feedback matrix $\boldsymbol{D}$, which contains ratings (1–5 stars) that users gave to movies they had watched. To predict the ratings of the movies users did not yet watch, latent-factor models build two factor matrices: a user matrix $\boldsymbol{Q}$ and an item matrix $\boldsymbol{P}$, in which columns correspond to users and items, respectively, and rows to latent factors. Fig. 1b shows an example with $r = 2$ latent factors, which roughly correspond to action and romance. The predicted preference of user i for item j is given by the (i, j) entry of matrix product $\boldsymbol{Q}^T\boldsymbol{P}$, i.e., by the inner product $\boldsymbol{q}^T\boldsymbol{p} = \sum_{i=1}^{r} q_i p_i$ of the i-th row $\boldsymbol{q}^T$ of $\boldsymbol{Q}^T$ and the j-th column $\boldsymbol{p}$ of $\boldsymbol{P}$. The goal of a recommender system is to recommend high-preference items (among other criteria); we thus need to determine which entries are large. In Fig. 1b, e.g., we marked in bold face all entries > 3.

Another recent application of matrix factorization models is in open information extraction, which extracts and reasons about statements made in natural language text. Riedel et al. [3], for example, construct a *fact matrix* which consists of verb phrases in one dimension (e.g., "was born in") and subject-object-pairs in the other dimension (e.g., ("Einstein", "Ulm")). A nonzero entry indicates that the corresponding fact (a verbal phrase with its subject and object) was observed in a document collection. Matrix factorization techniques are used to predict additional facts, spot unlikely facts, and reason about verbal phrases. As in recommender systems, these methods create factor matrices using a suitable model and subsequently determine the large entries in their product; here large entries correspond to facts with a high predicted confidence.

In this paper, we study the problem of efficiently retrieving large entries in the product of two given factor matrices, which we refer to as *large-entry retrieval problem*. As in the above examples, the rows and columns of the product matrix often correspond to objects or attributes; large entries indicate strong interactions between objects and are often of particular interest in applications. In general, we consider an entry large, if it exceeds a threshold value or if it belongs to the set of largest entries of a row of the matrix. We focus on the setting in which the factor matrices are tall and skinny, each with millions of rows and tens to hundreds of columns.

In applications such as the ones above, the matrix product is significantly larger than the factor matrices themselves and its complete computation is generally infeasible in practice. E.g., the product of two 10M-by-50 factor matrices has 100 trillion entries—each formed by an inner product—and is 100 000 times larger than the factor matrices themselves. If an inner product computation takes about 100 ns on average (as in our experimental study), it takes more than 100 days to multiply the factor matrices (ignoring any other costs such as I/O costs). To avoid the computation of the full matrix product, we propose LEMP, an efficient algorithm to retrieve only the Large Entries of the Matrix Product. LEMP takes as input two sets of vectors (the columns of the factor matrices) and finds pairs of vectors having a large inner product (the entries of the product matrix).

The large-entry retrieval problem is closely related to the problems of top-k retrieval with linear scoring functions and cosine similarity search. The well-known threshold algo-

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2758-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2723372.2747647.

$$\begin{array}{c|ccccc} & \textit{Die Hard} & \textit{Taken} & \textit{Twilight} & \textit{Amelie} & \textit{Titanic} \\ \hline \textit{Adam} & 5 & & 1 & 2 & \\ \textit{Bod} & 5 & 4 & & & 1 \\ \textit{Charlie} & 2 & & 5 & & 4 \\ \textit{Dennis} & & 1 & 5 & 5 & \end{array}$$

(a) Feedback matrix $\boldsymbol{D}$

$$\begin{pmatrix} 1.6 & 1.3 & 0.7 & 1 & 0.4 \\ 0.6 & 0.8 & 2.7 & 2.8 & 2.2 \end{pmatrix} \boldsymbol{P}$$

$$\underbrace{\begin{pmatrix} 3.2 & -0.4 \\ 3.1 & -0.2 \\ 0 & 1.8 \\ -0.4 & 1.9 \end{pmatrix}}_{\boldsymbol{Q}^T} \quad \underbrace{\begin{pmatrix} \mathbf{4.9} & \mathbf{3.8} & 1.2 & 2.1 & 0.4 \\ \mathbf{4.8} & \mathbf{3.9} & 1.6 & 2.5 & 0.8 \\ 1 & 1.4 & \mathbf{4.9} & \mathbf{5.0} & \mathbf{4.0} \\ 0.5 & 1 & \mathbf{4.9} & \mathbf{4.9} & \mathbf{4.0} \end{pmatrix}}_{\boldsymbol{Q}^T\boldsymbol{P}}$$

(b) Factor matrices for users ($\boldsymbol{Q}$) and movies ($\boldsymbol{P}$) as well as corresponding predictions ($\boldsymbol{Q}^T\boldsymbol{P}$)

Figure 1: Example of a simple matrix factorization model

rithm (TA, [4]) can be used with small modifications (see Sec. 5) to retrieve from an inverted list index those vectors $\boldsymbol{p}$ that have the highest inner product with a specific query vector $\boldsymbol{q}$. TA works particularly well for vectors of low dimensionality (say less than 10). Cosine similarity search is also related to, but not equal to, the large-entry retrieval problem: both problems are equivalent when all vectors have unit length. When vector lengths differ, as in our setting, methods for cosine similarity search —such as all-pairs similarity search [5, 6, 7, 8] or locality sensitive hashing (LSH, [9])—cannot be directly used.

Our LEMP algorithm is inspired by TA and techniques from cosine similarity search. It makes use of the simple observation that both the length and the direction of two vectors influence the value of their inner product. In particular, LEMP groups the input vectors into *buckets* of similar lengths and subsequently solves a smaller cosine similarity search problem for each bucket. In this way, LEMP (i) exploits vector lengths for early pruning, (ii) is able to choose a suitable search technique separately for each bucket (and query), and (iii) improves cache locality by fitting the small problem instances into cache. LEMP is able to leverage existing methods for cosine similarity search to process its buckets. We also propose two novel methods termed COORD (for coordinate-based pruning) and INCR (for incremental pruning), which are tailored to our setting of intermediate dimensionality of the input vectors (say, 10–500), i.e., higher than usual for TA and lower than usual for cosine similarity search.

A number of previous approaches to solve the large-entry retrieval problem have been proposed in the literature; most recently the cover tree algorithm in [10]. In contrast to LEMP, none of the existing techniques separates length and direction information from the input vectors. We performed an experimental study on multiple large real-world datasets, where we compared LEMP to existing techniques and also studied the performance of various algorithms for bucket processing. We found that LEMP consistently outperformed existing methods by up to an order of magnitude and was multiple orders of magnitude faster than a naive approach. Although the performance of existing approaches for large-entry retrieval (TA, cover trees) often increased when used within the LEMP framework, LEMP with a variant of our specialized INCR algorithm performed best overall.

2 Problem Statement and Naive Solution

We denote matrices by bold upper-case letters, vectors by bold lower-case letters, and scalars by non-bold lowercase letters. Denote by $\boldsymbol{Q}$ and $\boldsymbol{P}$ two real-valued factor matrices of dimensions $r \times m$ and $r \times n$, respectively. We are interested in large entries in the $m \times n$ product matrix $\boldsymbol{Q}^T\boldsymbol{P}$. We assume throughout that $r \ll m, n$ so that (the transposes of) both factor matrices are tall and skinny; this setting arises, for example, when the input matrices have been obtained by low-rank matrix factorization methods as in Sec. 1. More specifically, we focus on cases where r is in the range of 10–500 and m as well as n take values that are orders of magnitudes larger, e.g., in the order of millions. We write $\boldsymbol{A}_j$ for the j-th column of matrix $\boldsymbol{A}$, $\boldsymbol{v} \in \boldsymbol{A}$ to indicate that $\boldsymbol{v}$ is a column of $\boldsymbol{A}$, and v_i for the i-th entry of $\boldsymbol{v}$. We use shortcut notation to omit the range of indexes in summations when clear from context; e.g., for $\boldsymbol{q} \in \boldsymbol{Q}$ and $\boldsymbol{p} \in \boldsymbol{P}$, we write $\boldsymbol{q}^T\boldsymbol{p} = \sum_i q_i p_i$ for $\sum_{i=1}^{r} q_i p_i$. Let $[n] = \{1, \ldots, n\}$.

We study two variants of the large-entries problem. The first one, termed Above-θ, asks to retrieve all entries that take values above some application-defined threshold θ. This problem is useful, for example, to retrieve all high-confidence facts in an open information extraction scenario.

PROBLEM 1 (ABOVE-θ). *Given a threshold $\theta > 0$, determine the set of large entries*

$$\{ (i, j) \in [m] \times [n] \mid [\boldsymbol{Q}^T\boldsymbol{P}]_{ij} \geq \theta \}.$$

The second problem asks to retrieve the k largest entries on each row of the product matrix, where k is application-defined. This problem is more suited to recommender systems, where we want to retrieve the most relevant items (columns of $\boldsymbol{P}$) for each user (column of $\boldsymbol{Q}$).

PROBLEM 2 (ROW-TOP-k). *Given an integer $k > 0$, find for every $\boldsymbol{q} \in \boldsymbol{Q}$ the set $J \subseteq [n]$ of the k columns of $\boldsymbol{P}$ that attain the k largest values in $\boldsymbol{q}^T\boldsymbol{P}$. Ties are broken arbitrarily.*

Note that if $\boldsymbol{Q}$ has only one column, the Row-Top-k problem is equivalent to top-k scoring with linear scoring function $f(\boldsymbol{p}) = \boldsymbol{q}^T\boldsymbol{p}$. In the general case, in which $\boldsymbol{Q}$ has multiple columns, it is equivalent to multi-query top-k scoring. In analogy to top-k scoring, we refer to $\boldsymbol{Q}$ as the *query matrix* and to $\boldsymbol{P}$ as the *probe matrix.* Similarly, we refer to vectors $\boldsymbol{q} \in \boldsymbol{Q}$ as *query vectors* (or simply *queries*) and to vectors $\boldsymbol{p} \in \boldsymbol{P}$ as *probe vectors.* The top-k values in each column of $\boldsymbol{Q}^T\boldsymbol{P}$ can be found by reversing the roles of $\boldsymbol{Q}$ and $\boldsymbol{P}$.

A simple solution to the above problems is to first compute the full product matrix $\boldsymbol{Q}^T\boldsymbol{P}$, and then select from this product all entries above the threshold (for Above-θ) or the k largest entries in each row (for Row-Top-k). We refer to this approach as *Naive*; it has time complexity $O(mnr)$ and is infeasible for large problem instances.

3 The LEMP Algorithm

In this section, we outline the LEMP algorithm for retrieving large entries in matrix products. We focus on the Above-θ problem throughout and discuss the Row-Top-k problem in Sec. 4.5.

3.1 Length and Direction

LEMP makes use of the decomposition of an inner product of two vectors $\boldsymbol{q}$ and $\boldsymbol{p}$ into a length and a direction part. Denote by $\|\boldsymbol{v}\| = \sqrt{\sum_f v_f^2}$ the length (Euclidean norm) of vector $\boldsymbol{v} \neq 0$, and by $\bar{\boldsymbol{v}} = \boldsymbol{v}/\|\boldsymbol{v}\|$ its *normalization*, i.e., the unit vector pointing in the direction of $\boldsymbol{v}$. Then

$$\boldsymbol{q}^T\boldsymbol{p} = \|\boldsymbol{q}\|\,\|\boldsymbol{p}\|\cos(\boldsymbol{q},\boldsymbol{p}), \tag{1}$$

where $\cos(\boldsymbol{q},\boldsymbol{p}) = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \in [-1,1]$ denotes the cosine similarity between vectors $\boldsymbol{q}$ and $\boldsymbol{p}$. As mentioned previously, the inner product coincides with the cosine similarity if $\boldsymbol{q}$ and $\boldsymbol{p}$ have unit length. The problem of cosine similarity search is thus a special case of the large-entry retrieval problem.

By rewriting Eq. (1), we obtain

$$\boldsymbol{q}^T\boldsymbol{p} \geq \theta \iff \cos(\boldsymbol{q},\boldsymbol{p}) \geq \frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}. \tag{2}$$

The inner product thus exceeds threshold θ if and only if the cosine similarity exceeds the modified threshold $\frac{\theta}{\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|}$, which depends on the lengths of $\boldsymbol{q}$ and $\boldsymbol{p}$. Our goal is to find pairs $(\boldsymbol{q},\boldsymbol{p}) \in \boldsymbol{Q} \times \boldsymbol{P}$ such that $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$. From Eq. 2, we conclude that:

1. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are short in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| < \theta$, we cannot have $\boldsymbol{q}^T\boldsymbol{p} > \theta$ since $\cos(\boldsymbol{q},\boldsymbol{p}) \in [-1,1]$ and $\theta/(\|\boldsymbol{q}\|\,\|\boldsymbol{p}\|) > 1$. Such pairs do not need to be considered.

2. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are of intermediate length in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \approx \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if the cosine similarity $\cos(\boldsymbol{q},\boldsymbol{p})$ is large. Such pairs are best found using a cosine similarity search algorithm.

3. If $\boldsymbol{q}$ and $\boldsymbol{p}$ are long in that $\|\boldsymbol{q}\|\|\boldsymbol{p}\| \gg \theta$, then $\boldsymbol{q}^T\boldsymbol{p} > \theta$ if their cosine similarity is not too small. Such pairs are best found using naive search.

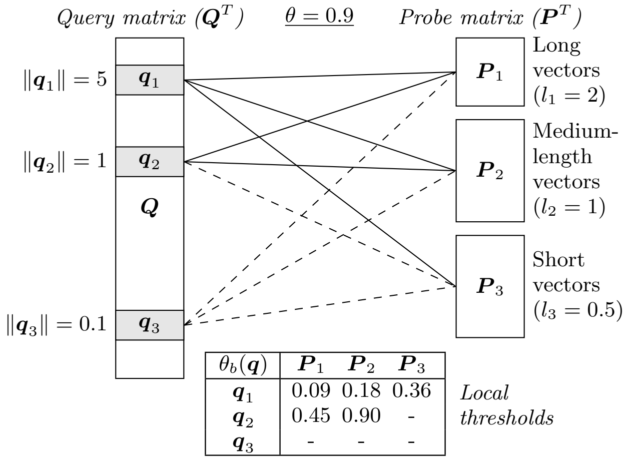
This indicates that vectors of different lengths are best treated in different ways. LEMP exploits this observation as follows. It first groups the vectors of the probe matrix $\boldsymbol{P}$ into a set of small buckets, each consisting of vectors of roughly similar length, and then solves a cosine similarity search problem for each bucket. In particular, we ignore buckets with short vectors, use a suitable cosine similarity search algorithm for buckets with vectors of intermediate lengths, and use (a variant of) naive retrieval for buckets with long vectors. This allows us to prune large parts of the search space and handle the remaining part efficiently.

In more detail, denote by $\boldsymbol{P}_1, \boldsymbol{P}_2, \ldots, \boldsymbol{P}_s$ a set of s buckets and assume that the vectors in each bucket have roughly similar (but not necessarily equal) length. For each bucket $\boldsymbol{P}_b$, $1 \leq b \leq s$, denote by $l_b = \max_{\boldsymbol{p}\in\boldsymbol{P}_b}\|\boldsymbol{p}\|$ the length of its longest vector. Under our assumption, $l_b \approx \|\boldsymbol{p}\|$ for all $\boldsymbol{p} \in \boldsymbol{P}_b$. Fig. 2 shows an example in which $\boldsymbol{P}$ has been divided into three buckets: $\boldsymbol{P}_1$ holds long vectors (approximate and maximum length 2), $\boldsymbol{P}_2$ medium-length vectors (1), and $\boldsymbol{P}_3$ short vectors (0.5).

Fix some bucket $\boldsymbol{P}_b$. From Eq. (2), we obtain that a necessary condition for $\boldsymbol{q}^T\boldsymbol{p} \geq \theta$ for $\boldsymbol{p} \in \boldsymbol{P}_b$ is that

$$\cos(\boldsymbol{q},\boldsymbol{p}) = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q}) \overset{\text{def}}{=} \frac{\theta}{\|\boldsymbol{q}\|\,l_b}. \tag{3}$$

We refer to $\theta_b(\boldsymbol{q})$ as the *local threshold* of query $\boldsymbol{q}$ for bucket $\boldsymbol{P}_b$. Our goal is thus to find all vectors $\boldsymbol{p} \in \boldsymbol{P}_b$ with a cosine similarity to $\boldsymbol{q}$ of at least $\theta_b(\boldsymbol{q})$. The local threshold allows us to determine how to best process bucket $\boldsymbol{P}_b$, analogous to the discussion above. If $\theta_b(\boldsymbol{q}) > 1$, we can prune the entire bucket since none of its vectors can potentially pass the threshold. If $\theta_b(\boldsymbol{q}) \approx 1$, we use a suitable cosine similarity search algorithm for the bucket. Finally, if $\theta_b \ll 1$, we use naive retrieval.



Query matrix ($\boldsymbol{Q}^T$)
$\theta = 0.9$
Probe matrix ($\boldsymbol{P}^T$)
$\|\boldsymbol{q}_1\| = 5$
$\boldsymbol{q}_1$
$\|\boldsymbol{q}_2\| = 1$
$\boldsymbol{q}_2$
$\boldsymbol{Q}$
$\|\boldsymbol{q}_3\| = 0.1$
$\boldsymbol{q}_3$
$\boldsymbol{P}_1$
Long vectors ($l_1 = 2$)
$\boldsymbol{P}_2$
Medium-length vectors ($l_2 = 1$)
$\boldsymbol{P}_3$
Short vectors ($l_3 = 0.5$)
Local thresholds

$\theta_b(\boldsymbol{q})$	$\boldsymbol{P}_1$	$\boldsymbol{P}_2$	$\boldsymbol{P}_3$
$\boldsymbol{q}_1$	0.09	0.18	0.36
$\boldsymbol{q}_2$	0.45	0.90	-
$\boldsymbol{q}_3$	-	-	-

Figure 2: Illustration of LEMP's bucketization

Consider again the example of Fig. 2 and assume a global threshold of $\theta = 0.9$. The figure highlights three query vectors $\boldsymbol{q}_1$, $\boldsymbol{q}_2$, and $\boldsymbol{q}_3$ of decreasing lengths and gives the values of all local thresholds (or "-" if above 1, also indicated by dashed lines). For $\boldsymbol{q}_1$, which is very long, all local thresholds are small so that naive retrieval is well suited for all buckets. For $\boldsymbol{q}_2$, which is shorter, the local threshold is small for bucket $\boldsymbol{P}_1$ (long vectors), large for bucket $\boldsymbol{P}_2$ (medium-length vectors), and above 1 for bucket $\boldsymbol{P}_3$ (short vectors). We use naive retrieval for $\boldsymbol{P}_1$ and a suitable cosine similarity search algorithm for $\boldsymbol{P}_2$. Bucket $\boldsymbol{P}_3$ is pruned. Finally, for $\boldsymbol{q}_3$, which is very short, all local thresholds exceed 1 so that all buckets are pruned.

3.2 Algorithm Description

Alg. 1 summarizes LEMP. It consists of a *preprocessing phase* (lines 1–6) and a *retrieval phase* (lines 8–19).

The preprocessing phase groups the columns of $\boldsymbol{P}$ into buckets of similar length (line 2). There are a number of ways to do this, but we chose a simple greedy strategy in our implementation. In particular, we first sort the columns of $\boldsymbol{P}$ by decreasing length,[1] scan the columns in order, and start a new bucket whenever the length of the current column falls below some threshold (e.g., 90% of l_b). We also make sure that buckets are neither too small nor too large. First, small buckets reduce the efficiency of LEMP due to bucket processing overheads; we thus ensure that buckets contain at least a certain number of vectors (30 in our implementation). The bucket processing overhead of large buckets is negligible. However, when buckets grow larger than the cache size, processing time is negatively affected. For this reason, we select a maximum bucket size that ensures that all relevant data structures fit into the processor cache.

[1]We also sort and normalize query vectors in a manner similar to the bucketization of $\boldsymbol{P}$.

Algorithm 1 LEMP for the Above-θ problem

```
Require: $\boldsymbol{Q}, \boldsymbol{P}, \theta$
Ensure: $S = \{ (i,j) \mid [\boldsymbol{Q}^T\boldsymbol{P}]_{ij} \geq \theta \}$
 1: // Preprocessing phase
 2: Partition $\boldsymbol{P}$ into buckets $\boldsymbol{P}_1, \ldots, \boldsymbol{P}_s$ of similar length
 3: for all $b \in 1, 2, \ldots, s$ do                    // for each bucket
 4:     Sort, normalize, and index $\boldsymbol{P}_b$
 5:     $l_b \leftarrow \max_{\boldsymbol{p} \in \boldsymbol{P}_b} \|\boldsymbol{p}\|$
 6: end for
 7:
 8: // Retrieval phase
 9: $S \leftarrow \emptyset$
10: for all $b \in 1, 2, \ldots, s$ do                    // for each bucket
11:     for all $\boldsymbol{q}_i \in \boldsymbol{Q}$ do                      // for each query
12:         $\theta_b(\boldsymbol{q}_i) \leftarrow \theta \,/\, (\|\boldsymbol{q}_i\|\, l_b)$             // local threshold
13:         if $\theta_b(\boldsymbol{q}_i) \leq 1$ then                          // prune?
14:             Pick a suitable retrieval alg. $A$ based on $\theta_b(\boldsymbol{q}_i)$
15:             Use $A$ to obtain a set of candidates
                $C_b \supseteq \{ \boldsymbol{p}_j \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}_i^T \bar{\boldsymbol{p}}_j \geq \theta_b(\boldsymbol{q}_i) \}$
16:             $S \leftarrow S \cup \{ (i,j) \mid \boldsymbol{p}_j \in C_b$ and $\boldsymbol{q}_i^T \boldsymbol{p}_j \geq \theta \}$
17:         end if
18:     end for
19: end for
```

After bucket boundaries have been obtained, we represent each vector $\boldsymbol{p}$ by two separate components: its length $\|\boldsymbol{p}\|$ and its direction $\bar{\boldsymbol{p}}$. We also store the vectors' column number in the original matrix (denoted `id`) and in the bucket (denoted `lid` for "local id"); see Fig. 4a for an example. This layout allows us to access for each $\boldsymbol{p} \in \boldsymbol{P}_b$ both $\|\boldsymbol{p}\|$ and $\bar{\boldsymbol{p}}$ without further computation. We then create indexes on the contents of each bucket; we defer the discussion of indexing to Sec. 4. For our choice of indexes (Sec. 4.2 and 4.3), the overall preprocessing time, including index computation, is $O(rn \log n)$.

The retrieval phase then iterates over buckets and query vectors. For each query, we compute the local threshold $\theta_b(\boldsymbol{q})$ (line 12) and prune buckets based on their length (line 13). For each remaining bucket $\boldsymbol{P}_b$, we select a suitable retrieval algorithm based on the local threshold (line 14, cf. Sec. 4). The selected retrieval algorithm computes a set C_b of *candidate vectors*, potentially making use of the index data structures created during the preprocessing phase. The candidate set is guaranteed to contain all vectors in $\boldsymbol{p} \in \boldsymbol{P}_b$ that pass the threshold ($\boldsymbol{q}^T\boldsymbol{p} \geq \theta$), but it may additionally contain a set of *spurious vectors* ($\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \geq \theta_b(\boldsymbol{q})$ but $\boldsymbol{q}^T\boldsymbol{p} < \theta$). A verification step (line 16) filters out these spurious vectors by computing the actual values of the inner products $\boldsymbol{q}^T\boldsymbol{p}$ for all $\boldsymbol{p} \in C_b$.

The order of the two loops in the retrieval phase of Alg. 1 is chosen to be cache friendly. Since we process probe buckets in the outer loop and since probe buckets are small, their content remains in the cache for the entire inner loop. The inner loop itself scans query vectors sequentially; these vectors generally do not fit into the cache, but the sequential access pattern makes prefetching effective.

The power of LEMP to prune entire buckets in line 12 depends on the length distribution of the input vectors: generally, the more skewed the length distribution, the more probe buckets can be pruned. Even if bucket pruning is not particularly effective for a given problem instance, however, the organization of the probe vectors into buckets is still beneficial: it allows cosine similarity search algorithms to be applied and is cache-friendly.

4 Retrieval Algorithms

In this section, we propose and discuss a number of algorithms for the retrieval phase of LEMP (line 15 of Alg. 1). Each algorithm takes as input a query vector $\boldsymbol{q} \in \boldsymbol{Q}$ and a bucket $\boldsymbol{P}_b$, and outputs a candidate set $C_b \subseteq \boldsymbol{P}_b$ using some pruning strategy. All algorithms first compute $\|\boldsymbol{q}\|$ and $\bar{\boldsymbol{q}}$; cf. Fig. 4d.

We discuss two kinds of algorithms: those that make use of only the length information to prune candidate vectors and those that use the normalized vectors as well. For the first category, we propose the LENGTH algorithm (Sec. 4.1), which is a simple variant of the naive algorithm that takes length information into account. Existing cosine similarity search algorithms (e.g., [5]) as well as TA fall in the second category. Here we additionally present two novel methods, which are specially tailored to the matrix-product setting. The COORD algorithm (Sec. 4.2) applies coordinate-based pruning strategies. The INCR algorithm (Sec. 4.3) is based on COORD but uses a more effective (but also more expensive) incremental pruning strategy that also takes length into account.

4.1 Length-Based Pruning

Recall that the vectors in bucket $\boldsymbol{P}_b$ are sorted by decreasing length during preprocessing (see also Fig. 4a). Further observe from Eq. (1) that whenever $\|\boldsymbol{q}\|\,\|\boldsymbol{p}\| < \theta$, so is $\boldsymbol{q}^T\boldsymbol{p}$. Putting both together, LENGTH scans the bucket $\boldsymbol{P}_b$ in order. When processing vector $\boldsymbol{p}$, we check whether $\|\boldsymbol{p}\| \geq \theta/\|\boldsymbol{q}\|$; we precompute $\theta/\|\boldsymbol{q}\|$ to make this check efficient. If $\boldsymbol{p}$ qualifies, we add it to the candidate set C_b. Otherwise, we stop processing bucket $\boldsymbol{P}_b$ and immediately output C_b.

Consider for example a bucket $\boldsymbol{P}_b$ as shown in Fig. 4a, query vector $\boldsymbol{q} = (1,1,1,1)^T$, and threshold $\theta = 3.8$. We have $\|\boldsymbol{q}\| = 2$ and $\theta/\|\boldsymbol{q}\| = 1.9$ so that we obtain $C_b = \{1,2,3\}$. (Here and in the following, we give C_b in terms of local identifiers (lid) for improved readability.)

Since LEMP already organizes and prunes buckets by length, we do not expect LENGTH to be particularly effective. In fact, LENGTH degenerates to the naive algorithm in all but one bucket (the "last" bucket that has not been pruned). Nevertheless, since LENGTH has low overhead and a sequential access pattern, it is an effective method when buckets are small or the local threshold is low (i.e., when coordinate-based pruning is not effective).

4.2 Coordinate-Based Pruning

We now proceed to pruning strategies based on the direction (but not length) of the query vector. The key idea is to retain only those vectors from $\boldsymbol{P}_b$ in C_b that point in a similar direction as $\boldsymbol{q}$. In particular, we aim to find all $\boldsymbol{p} \in \boldsymbol{P}_b$ with high cosine similarity to $\boldsymbol{q}$, i.e.,

$$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} = \cos(\boldsymbol{q}, \boldsymbol{p}) \geq \theta_b(\boldsymbol{q}). \qquad (4)$$

Note the usage of normalized vectors here; length information is not taken into account.

Let $\bar{\boldsymbol{q}} = (\bar{q}_1, \ldots, \bar{q}_r)^T$ and $\bar{\boldsymbol{p}} = (\bar{p}_1, \ldots, \bar{p}_r)^T$. Note that $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$ achieves its maximum value for $\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}$ since then $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}^T\bar{\boldsymbol{q}} = \|\bar{\boldsymbol{q}}\|^2 = 1$. In other words, $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$ is maximized when both vectors agree on all their coordinates. Based on this observation, the key idea of the COORD algorithm is to prune $\bar{\boldsymbol{p}}$ if one of its coordinates deviates too far from the respective coordinate in $\bar{\boldsymbol{q}}$. In more detail, we obtain for each coordinate $f \in [r]$ a lower bound $L_f(\bar{\boldsymbol{q}})$ and an upper bound $U_f(\bar{\boldsymbol{q}})$ on $\bar{p}_f$. If $L_f \le \bar{p}_f \le U_f$, we say that $\bar{p}_f$ is *feasible*; otherwise $\bar{p}_f$ is *infeasible*. The bounds are chosen such that whenever a coordinate f of $\boldsymbol{p}$ is infeasible, then $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ so that $\boldsymbol{p}$ can be pruned from the candidate set. Such pruning is particularly effective when $\theta_b(\boldsymbol{q})$ is large or when the query vector is sparse.

In what follows, we provide lower and upper bounds, discuss their effectiveness, and propose the COORD algorithm that exploits them.

Bounding Coordinates. Pick some coordinate $f \in [r]$; we refer to f as a *focus coordinate.* Denote by $\bar{\boldsymbol{q}}_{\text{-}f} = \{\bar{q}_1, \ldots, \bar{q}_{f-1}, \bar{q}_{f+1}, \ldots, \bar{q}_r\}$ the vector obtained by removing coordinate f from $\bar{\boldsymbol{q}}$, similarly $\bar{\boldsymbol{p}}_{\text{-}f}$. Note that $\bar{\boldsymbol{q}}_{\text{-}f}$ and $\bar{\boldsymbol{p}}_{\text{-}f}$ generally have length less than 1. Now we rewrite Eq. (4) as follows

$$\begin{aligned}\theta_b(\boldsymbol{q}) \le \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} = \textstyle\sum_i \bar{q}_i\bar{p}_i &= \bar{q}_f\bar{p}_f + \textstyle\sum_{i\ne f}\bar{p}_i\bar{q}_i \\ &= \bar{q}_f\bar{p}_f + \bar{\boldsymbol{q}}_{\text{-}f}^T\bar{\boldsymbol{p}}_{\text{-}f} \\ &= \bar{q}_f\bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\|\,\|\bar{\boldsymbol{p}}_{\text{-}f}\|\cos(\bar{\boldsymbol{p}}_{\text{-}f}, \bar{\boldsymbol{q}}_{\text{-}f}) \\ &\le \bar{q}_f\bar{p}_f + \|\bar{\boldsymbol{q}}_{\text{-}f}\|\,\|\bar{\boldsymbol{p}}_{\text{-}f}\| \\ &= \bar{q}_f\bar{p}_f + \sqrt{1-\bar{q}_f^2}\sqrt{1-\bar{p}_f^2},\end{aligned}$$

where we used Eq. (1), the fact that the cosine similarity cannot exceed 1, and the property $\|\bar{\boldsymbol{q}}\| = \|\bar{\boldsymbol{p}}\| = 1$.

We now solve the resulting inequality $\theta_b(\boldsymbol{q}) \le \bar{q}_f\bar{p}_f + (1-\bar{q}_f^2)^{1/2}(1-\bar{p}_f^2)^{1/2}$ for $\bar{p}_f$ and obtain the bounds $L_f \le \bar{p}_f \le U_f$, where:

$$\begin{aligned} L'_f &= \bar{q}_f\theta_b(\boldsymbol{q}) - \sqrt{(1-\theta_b(\boldsymbol{q})^2)(1-\bar{q}_f^2)}, \\ U'_f &= \bar{q}_f\theta_b(\boldsymbol{q}) + \sqrt{(1-\theta_b(\boldsymbol{q})^2)(1-\bar{q}_f^2)}, \\ L_f &= \begin{cases} L'_f & \bar{q}_f \ge 0 \text{ or } L'_f > \theta_b(\boldsymbol{q})/\bar{q}_f \\ -1 & \text{otherwise,}\end{cases} \\ U_f &= \begin{cases} U'_f & \bar{q}_f \le 0 \text{ or } U'_f < \theta_b(\boldsymbol{q})/\bar{q}_f \\ 1 & \text{otherwise.}\end{cases}\end{aligned}$$

Note that if the lengths of the vectors within a bucket vary strongly, we are forced to use a low local threshold $\theta_b(\boldsymbol{q})$, which in turn results in looser bounds. This undesirable behavior is avoided by LEMP since it constructs buckets that contain vectors of similar length. The effectiveness of our bounds—and of using normalization and subsequent coordinate-based pruning in general—is thus particularly effective in the context of our LEMP framework.

Effectiveness of Bounds. To gain some insight into the effectiveness of our bounds, we plot the *feasible region* $[L_f, R_f]$ for various choices $\theta_b(\boldsymbol{q})$ in Fig. 3. The x-axis corresponds to the value of $\bar{q}_f$, the y-axis to the lower and upper bounds, and the various oval-shaped gray regions to the feasible regions. Note that $-1 \le \bar{q}_f, \bar{p}_f \le 1$.

The pruning power of our bounds depends on both the value of $\theta_b(\boldsymbol{q})$ and on the properties of matrices $\boldsymbol{Q}$ and $\boldsymbol{P}$. First, the larger the local threshold $\theta_b(\boldsymbol{q})$, the smaller the feasible region and the more vectors can be pruned. In fact,

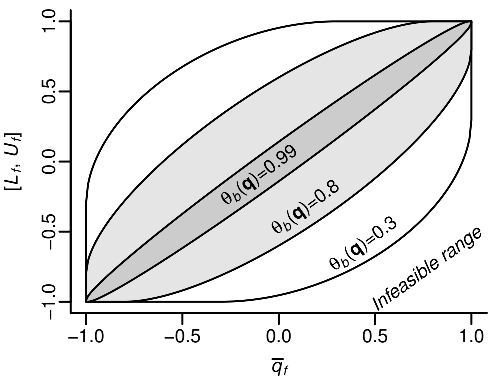


Figure 3: Feasible regions for various values of $\theta_b(\boldsymbol{q})$

for large values of $\theta_b(\boldsymbol{q})$, the feasible region is small across the entire value range of $\bar{q}_f$. Second, the size of the feasible region decreases as the magnitude of $\bar{q}_f$ increases. This decrease is more pronounced when the local threshold is small. Note that a small feasible region may or may not lead to effective pruning; the value distribution of $\boldsymbol{P}_b$ is also important. Nevertheless, the smaller the feasible region, the more effective the pruning will be.

Based on the observations above, we conclude that our bounds can effectively prune a vector $\bar{\boldsymbol{p}}$ with $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} < \theta_b(\boldsymbol{q})$ when $\theta_b(\boldsymbol{q})$ is large or when there is some coordinate f for which only one of $\bar{q}_f$ or $\bar{p}_f$ takes a large value. Since all vectors are length-normalized, the latter property holds if $\bar{\boldsymbol{q}}$ or $\bar{\boldsymbol{p}}$ is sufficiently sparse or has a skewed value distribution. If neither holds and $\theta_b(\boldsymbol{q})$ is small, an algorithm such as LENGTH or INCR may be a more suitable choice.

Exploiting Bounds. The COORD algorithm makes use of the feasible region derived in the previous section to prune unpromising candidates. To do so, LEMP creates indexes for each probe bucket $\boldsymbol{P}_b$ during its preprocessing phase. In the case of COORD, we create r sorted lists $I_1, \ldots, I_r$, one for each coordinate of the vectors in $\boldsymbol{P}_b$. Each entry in list I_f is a (lid, $\bar{p}_f$)-pair, where as before lid is a bucket-local identifier for the corresponding vector $\bar{\boldsymbol{p}}$. As in Fagin et al.'s threshold algorithm (TA, [4]), the lists are sorted in decreasing order of $\bar{p}_f$. Fig. 4c shows the sorted-list index for the example bucket given in Fig. 4a. Although index construction is generally light-weight and fast, LEMP constructs indexes lazily on first use to further reduce computational cost. Buckets with very short vectors, for example, will always be pruned and thus do not need to be indexed.

COORD is summarized as Alg. 2. It takes as input a bucket $\boldsymbol{P}_b$, a query $\boldsymbol{q}$, the global and local thresholds $(\theta, \theta_b(\boldsymbol{q}))$, the bucket indexes $I_1, \ldots, I_r$, and a set of focus coordinates $F \subseteq [r]$. We discuss the algorithm using the example of Fig. 4 with $\theta = 0.9$. Consider the query $\boldsymbol{q}$ shown in Fig. 4d as well as the corresponding inner products shown in Fig. 4b. We have $\theta_b(\boldsymbol{q}) = 0.9/(0.5 \cdot 2) = 0.9$, coincidentally agreeing with the global threshold. Observe that vectors 1 and 5 pass the local threshold $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \ge \theta_b(\boldsymbol{q})$, but only vector 1 additionally passes the global threshold $\boldsymbol{q}^T\boldsymbol{p} \ge \theta$.

COORD does not compute and enforce the bounds for each coordinate but uses a suitable subset $F \subseteq [r]$ of focus coordinates; see below. For each focus coordinate $f \in F$, COORD computes the feasible region $[L_f, U_f]$ (line 3) and determines the start and end of the corresponding *scan range*

lid	id	$\|\boldsymbol{p}\|$	$\bar{\boldsymbol{p}}$			
1	23	2.0	0.58	0.50	0.40	0.50
2	43	1.9	0.98	0	0	0.20
3	12	1.9	0.53	0	0	0.85
4	54	1.8	0.35	0.93	0	0.10
5	18	1.8	0.58	0.50	0.40	0.50
6	20	1.8	0.30	-0.40	0.81	-0.30

(a) Organization of bucket $\boldsymbol{P}_b$

$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$	$\boldsymbol{q}^T\boldsymbol{p}$
0.97	**0.97**
0.79	0.75
0.80	0.76
0.56	0.52
0.97	0.87
0.26	0.23

(b) Results for query $\boldsymbol{q}$ of (d)

I_1

lid	$\bar{p}_1$
2	0.98
1	**0.58**
5	**0.58**
3	**0.53**
4	**0.35**
6	0.30

I_2

lid	$\bar{p}_2$
4	0.93
1	0.50
5	0.50
2	0
3	0
6	-0.40

I_3

lid	$\bar{p}_3$
6	0.81
1	0.40
5	0.40
2	0
3	0
4	0

I_4

lid	$\bar{p}_4$
3	0.85
1	**0.50**
5	**0.50**
2	**0.20**
4	**0.10**
6	-0.30

(c) Sorted-list index (bold rows show scan range for $\boldsymbol{q}$)

	$\|\boldsymbol{q}\|$	$\bar{\boldsymbol{q}}$			
	0.5	0.70	0.3	0.4	0.51
$[L_f, U_f]$		$[0.32, 0.94]$	-	-	$[0.09, 0.83]$

(d) Query $\boldsymbol{q}$ and feasible region for focus coordinates

lid	c
1	**2**
2	1
3	1
4	**2**
5	**2**
6	0

$C_b = \{1, 4, 5\}$

(e) CP array

lid	c	$\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F$	$\|\boldsymbol{p}_F\|^2$	u	$\theta_{\boldsymbol{p}}(\boldsymbol{q})$
1	**2**	**0.66**	**0.59**	**0.32**	**0.9**
2	1	0.10	0.04	0.49	0.95
3	1	0.37	0.28	0.43	0.95
4	2	0.30	0.13	0.47	1
5	2	0.66	0.59	0.32	1
6	0	-	-	-	1

$C_b = \{1\}$

(f) Extended CP array

Figure 4: Illustration of LEMP as well as the COORD and INCR retrieval algorithms for $\theta = 0.9$ and $F = \{1, 4\}$

in sorted list I_f via binary search for U_f and L_f, respectively (line 4). Vectors outside the scan range violate the bound on coordinate f. In the example of Fig. 4, we used $F = \{1, 4\}$. The bounds are shown in Fig. 4d and the corresponding scan ranges in I_1 and I_4 are shown in bold face in Fig. 4c.

COORD subsequently scans the scan range of each sorted list I_f, $f \in F$, in sequence (line 5) and maintains a *candidate-pruning array* (CP array, line 6). The CP array contains for each vector $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ with local identifier *lid* a counter $c[lid]$ that indicates how often the vector has been seen so far. The CP array of our running example is shown in Fig. 4e (with an additional lid column for improved readability). After completing all scans, COORD includes into C_b all those vectors $\bar{\boldsymbol{p}} \in \boldsymbol{P}_b$ that qualified on all focus coordinates, i.e., for which $c[lid] = |F|$ (line 9). In our example, $C_b = \{1, 4, 5\}$ since only those three vectors occurred in both scan ranges. In particular, vectors 2 and 3 are (correctly) excluded because they appear in only one scan range.

We now turn to the question of how to choose the focus set F. One option is to simply set $F = [r]$. However, processing sorted lists can get expensive if F is large or contains coordinates for which pruning is not effective, i.e., for which a large fraction of the corresponding sorted lists needs to be scanned. We make use of a focus-set size parameter ϕ, typically in the range of 1–5; we discuss the choice of ϕ in Sec. 4.4. COORD then uses the ϕ coordinates of $\bar{\boldsymbol{q}}$ with largest absolute value as focus coordinates. The reasoning behind this choice is that large coordinates will lead to the smallest feasible region (cf. Sec. 4.2); the hope is that they also lead to a small scan ranges and a small candidate set.

Algorithm 2 The COORD algorithm

```
Require: $\boldsymbol{q}, \boldsymbol{P}_b, \theta, \theta_b(\boldsymbol{q}), F \subseteq [r], I_1, \ldots, I_r$
Ensure: $C_b \supseteq \{\, \boldsymbol{p}_j \in \boldsymbol{P}_b \mid \bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}_j \geq \theta_b(\boldsymbol{q}) \,\}$
1: $c \leftarrow$ empty CP array
2: for all $f \in F$ do
3:     Calculate feasible region $[L_f, U_f]$
4:     Determine corresponding scan range in sorted list $I_f$
5:     for all $lid$ in scan range of $I_f$ do
6:         $c[lid] \leftarrow c[lid] + 1$        // maintain CP array
7:     end for
8: end for
9: $C_b = \{\, lid \mid c[lid] = |F| \,\}$        // filter
```

To summarize, COORD builds indexes only if needed and uses only a subset of the entries in a subset of the sorted-list indexes. The index scan itself is light-weight; it accesses solely the lid part of the lists and increases the counters of the CP array. Also note that the bounds we use for determining the scan range of the lists are simple and relatively cheap to compute. This is important since these bounds need to be computed per query, per bucket, and per focus coordinate. See Appendix A for implementation details.

4.3 Incremental Pruning

COORD scans the sorted-list indexes to find the set of vectors that qualify in each coordinate $f \in F$, i.e., fall in region $[L_f, U_f]$. Other than checking feasibility, the actual values in the scanned lists are ignored. In contrast, our incremental pruning algorithm INCR makes use of the $\bar{p}_f$ values as well: It maintains information that allows it to prune additional vectors. Such an approach is generally more expensive than COORD, but the increase in pruning power may offset the costs.

When we derived the bounds of a coordinate f of COORD, we assumed that $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f}) = 1$. This is a worst-case assumption; in general, $\cos(\bar{\boldsymbol{q}}_{\text{-}f}, \bar{\boldsymbol{p}}_{\text{-}f})$ will be less (and often much less) than 1. Intuitively, a vector $\bar{\boldsymbol{p}}$ that qualifies barely in all coordinates often does not constitute an actual result. Recall our ongoing example of Fig. 4. Here vector 4 barely qualifies in both indexes I_1 and I_4 and is thus included into the candidate set of COORD. Vector 4 does not pass the local threshold, however, since $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}_4 = 0.56 < 0.9$. COORD is blind to this behavior.

Another potential drawback of COORD is that it does not (and cannot) take into consideration the length distribution of the vectors in each bucket. In the example of Fig. 4, normalized vectors 1 and 5 are identical and both pass the local threshold. However, since vector 1 is slightly longer than vector 5, only vector 1 passes the global threshold and thus the verification step of LEMP.

Similar to COORD, INCR scans the scan ranges of the sorted lists of the focus coordinates. To address the above issues, however, INCR additionally maintains a partial inner product for each of the vectors that it encounters. Generalizing our previous notation, denote by $\bar{\boldsymbol{q}}_F$ ($\bar{\boldsymbol{q}}_{\text{-}F}$) the values of the focus coordinates (of all other coordinates) of the query vector; similarly, $\bar{\boldsymbol{p}}_F$ and $\bar{\boldsymbol{p}}_{\text{-}F}$. We obtain

$$\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} = \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + \bar{\boldsymbol{q}}_{\text{-}F}^T\bar{\boldsymbol{p}}_{\text{-}F} \leq \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + \|\bar{\boldsymbol{q}}_{\text{-}F}\|\,\|\bar{\boldsymbol{p}}_{\text{-}F}\|.$$

Since vectors are normalized, the latter can be computed from $\bar{\boldsymbol{q}}_F$ and $\bar{\boldsymbol{p}}_F$ only. Denote the resulting upper bound on the "unseen" part $\bar{\boldsymbol{q}}_{\text{-}F}^T\bar{\boldsymbol{p}}_{\text{-}F}$ of the inner product $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}}$ by

$$u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F) = \|\bar{\boldsymbol{q}}_{\text{-}F}\|\,\|\bar{\boldsymbol{p}}_{\text{-}F}\| = \sqrt{1-\|\bar{\boldsymbol{q}}_F\|^2}\sqrt{1-\|\bar{\boldsymbol{p}}_F\|^2}.$$

Then $\bar{\boldsymbol{q}}^T\bar{\boldsymbol{p}} \leq \bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. In order to compute this bound, INCR uses an *extended CP array*, which maintains for each probe vector in addition to the frequency counters of COORD (line 6 of Alg. 2) the quantities $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F$ and $\|\bar{\boldsymbol{p}}_F\|^2 = \sum_{f\in F}\bar{p}_f^2$. After the extended CP array has been computed, INCR includes into the candidate set only those vectors $\bar{\boldsymbol{p}}$ that satisfy

$$\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u(\|\boldsymbol{q}_F\|, \|\boldsymbol{p}_F\|) \geq \theta_{\boldsymbol{p}}(\boldsymbol{q}) \overset{\text{def}}{=} \frac{\theta}{\|\boldsymbol{p}\|\cdot\|\boldsymbol{q}\|}. \qquad (5)$$

Here $\theta_{\boldsymbol{p}}(\boldsymbol{q})$ is an improved, probe vector-specific local threshold; it holds $\theta_{\boldsymbol{p}}(\boldsymbol{q}) \geq \theta_b(\boldsymbol{q})$. This improved local threshold cannot be used by the COORD algorithm.

Fig. 4f shows the extended CP array for our running example (to the left of the double vertical lines) as well as the quantities involved in the above pruning condition (to the right; here we write u for $u(\|\boldsymbol{q}_F\|, \|\boldsymbol{p}_F\|)$). For example, for vector 1, $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F = 0.58\cdot 0.70 + 0.50\cdot 0.51 = 0.66$ and $u = \sqrt{1-(0.58^2+0.50^2)}\cdot\|\boldsymbol{q}_{\text{-}F}\|$. The quantity $\|\boldsymbol{q}_{\text{-}F}\|$ (not shown in Fig. 4f) is independent of the probe vectors and thus only computed once. In our example, $\|\boldsymbol{q}_{\text{-}F}\| = \sqrt{1-(0.70^2+0.51^2)} = 0.5$. As can be seen in the example, filter condition $\bar{\boldsymbol{q}}_F^T\bar{\boldsymbol{p}}_F + u \geq \theta_{\boldsymbol{p}}(\boldsymbol{q})$ is passed only by vector 1; thus $C_b = \{\,1\,\}$. Note that the rows of vector 5 and vector 1 agree in the extended CP array; our improved local threshold (0.9 for vector 1 vs. 1 for vector 5), however, allows us to correctly prune vector 5 but retain vector 1.

4.4 Algorithm Selection

Before processing a bucket $\boldsymbol{P}_b$, LEMP needs to decide which retrieval algorithm to use. We have already given some guidance for this choice above: Length-based pruning is suitable for buckets with a skewed length distribution, whereas coordinate-based pruning is suitable for large local thresholds and/or data with a skewed value distribution. In general, the choice of a suitable algorithm is data-dependent.

LEMP uses a simple, pragmatic method for algorithm selection: it samples a small set of query vectors and tests the different methods for each bucket. We observe the wall-clock times obtained by the various methods and select a threshold t_b for each bucket: whenever $\theta_b(\boldsymbol{q}) < t_b$, LEMP will use LENGTH, otherwise it uses coordinate-based pruning. Similarly, we select for each bucket a parameter ϕ_b for the number of sorted lists to scan in coordinate-based pruning; we simply take the choice that performed best on the sampled query vectors. The cost of this sample-based profiling step is negligible since the number of query vectors is large; the overall running time is dominated by the time required to process $\boldsymbol{Q}$ in its entirety.

More elaborate approaches for algorithm selection are possible, e.g., some form of reinforcement learning. Our experiments suggest, however, that even the simple selection criterion outlined above gives promising results.

4.5 Solving the Row-Top-k Problem

Our discussion so far has focused on the Above-θ problem; we now proceed to the discussion of the Row-Top-k problem. Recall that given a query vector $\boldsymbol{q}$, the Row-Top-k problem asks for the vectors $\boldsymbol{p}\in\boldsymbol{P}$ that attain the k largest inner products $\boldsymbol{q}^T\boldsymbol{p}$ (the k largest entries on the corresponding row of $\boldsymbol{Q}^T\boldsymbol{P}$). Row-Top-$k$ is often used in recommender systems for retrieving the best k items for each user.

The Row-Top-k problem is related to the Above-θ problem as follows. Fix a query vector $\boldsymbol{q}$ and denote by θ^* the k-th largest entry in $\boldsymbol{q}^T\boldsymbol{P}$. Given θ^*, the solution of the Row-Top-k problem coincides with the solution of the Above-θ algorithm with threshold θ^* (assuming no duplicate entries). We do not know θ^* but instead make use of a running lower bound θ' on θ^*; the value of θ' increases as the algorithm proceeds.

In more detail, we take the k longest vectors of $\boldsymbol{P}$ (all located at the beginning of bucket $\boldsymbol{P}_1$) and compute their inner product with $\boldsymbol{q}$. The smallest so-obtained value is our initial choice of θ'. We then run the Above-θ algorithm with threshold θ' on the first bucket, determine the top-k answers in the result, and update θ' accordingly. This process is iterated over the following buckets until θ' becomes so large that LEMP prunes the next bucket. At this point, we output the current top-k vectors as a result. This strategy is effective because (1) LEMP organized buckets by decreasing length so that we expect the top-k values to appear in the top-most buckets, and (2) bucket sizes are small (cache-resident) so that the threshold θ' is increased frequently.

Note that the length of $\boldsymbol{q}$ does not affect the result of the Row-Top-k problem. We thus simplify the bounds used by our algorithms by fixing $\|\boldsymbol{q}\| = 1$.

5 Related Work

A number of existing methods that are related to the large-entry retrieval problem have been proposed in the literature. We first review existing algorithms for the large-entry retrieval problem and subsequently turn attention to cosine similarity search algorithms. In general, LEMP differs from existing methods in that it separates the length and direction of the input vectors, prefers inexpensive pruning strategies over more aggressive, expensive ones, and selects suitable retrieval methods dynamically.

Algorithms for large-entry retrieval. To the best of our knowledge, Ram and Gray [11] were the first to pose and address the problem of fast maximum inner-product search, which corresponds to Row-Top-k. They propose to organize the probe vectors in a *metric tree*, in which each node is associated with a sphere that covers the probe vectors below the node. Given a query vector, the spheres are exploited to avoid processing subtrees that cannot contribute to the result. The metric tree itself is constructed by repeatedly splitting the set of probe vectors into two partitions (based on Euclidean distances). In subsequent work [10], the metric tree is replaced a *cover tree* [12]. Both approaches effectively prune the search space, but they suffer from high tree-construction costs and from random memory access patterns during tree traversal. The latter problem was investigated more closely in [13], where a *dual-tree algorithm* that additionally arranges query vectors in a cover tree and processes queries in batches was proposed. The dual-tree methods loosens the bounds for pruning the search space, however, and was found to be not effective in practice (confirmed also in our experimental study).

LEMP differs from these tree-based techniques in that it separates length and direction information, makes use of multiple, light-weight indexing and retrieval methods, and has more favorable memory access patterns. Note that the single-tree approach can also be used within the LEMP framework as a bucket algorithm that solves directly the large-entry retrieval problem. We expect that such a combination will have positive effect w.r.t. indexing time and cache locality. We explore this direction in our experiments.

An alternative approach is taken by [14] in the context of recommender systems: the matrix factorization method used to produce the input matrices is modified, such that all vectors are (approximately) unit vectors and the inner product of user and item vectors and be approximated by standard cosine similarity search. However, this modification may affect the quality of the recommendations. In contrast, LEMP makes no assumption on the source or method used to compute the input matrices.

Approximate methods for large-entry retrieval have also been studied in the literature. One line of work makes use of asymmetric transformations of $\boldsymbol{P}$ and $\boldsymbol{Q}$ to obtain an equivalent nearest-neighbor problem in Euclidean space; this problem is then solved approximately using LSH [15] or modified PCA-trees [16]. Another approach [17] is to cluster query vectors and solve the Row-Top-k problem only for cluster centroids. Although we focus on exact retrieval in this paper, such a method can directly be applied in combination with LEMP.

Threshold Algorithm. Some of our indexing techniques are inspired by the popular threshold algorithm (TA) of Fagin et al. [4] for top-k query processing for monotonic functions. TA arranges the values of each coordinate of the probe vectors in a sorted list, one per coordinate. Given a query, TA repeatedly selects a suitable list (e.g., round robin or heap-based), retrieves the next vector from the top of the list, and maintains the set of the top-k results seen so far. TA uses a termination criterion to stop processing as early as possible. The effectiveness of this criterion depends on the data; if TA is able to stop early, it can be very efficient. Note that TA usually focuses on vectors of low dimensionality (say up to 10), whereas we focus on vectors of medium sizes (say 10 to 500). TA can be used for finding vectors with large inner products almost as is; the only difference is that sorted lists need to be processed bottom-to-top when the respective coordinate of the query vector is negative.

LEMP improves over TA in multiple ways: First, bucket pruning eliminates early all short probe vectors, which otherwise TA would have to consider. Second, TA scans lists from top-to-bottom, whereas LEMP considers only the feasible region. Third, TA immediately computes the inner product of each vector selected from one of the lists in the index; i.e., candidate verification is triggered by individual coordinates. LEMP does not immediately calculate an inner product when it encounters a vector: it first scans multiple list and prunes the vectors before verification based on the so-obtained information. Finally, index scan and verification is interleaved in TA, resulting in a random memory access pattern and a potentially high cache-miss rate. LEMP ensures that all bucket-related data (original vectors and indexes) fits into cache, thereby reducing the cache-miss rate.

In our experimental study we investigated the performance of TA in comparison to LEMP. We also experimented with TA in combination with LEMP, i.e., we used TA as a bucket algorithm. This addresses the first and the final point in the discussion above. Our experimental results indicate that a combination of TA and LEMP can be up to 25x faster than just using TA. Generally, LEMP can improve TA's performance for top-k problems with linear scoring functions (i.e., inner products).

Algorithms for fast cosine similarity search. Cosine similarity search algorithms, like all-pairs similarity search (APSS, [5, 6, 7, 8]) or locality-sensitive hashing (LSH, [9]), cannot be used directly for the large-entry retrieval problem.[2] These methods though can be used (with some modifications) as retrieval methods for LEMP's buckets.

Typical APSS algorithms and applications involve sparse vectors of high dimensionality (tens or hundreds of thousands of coordinates). In such settings, sparsity must be retained during indexing to keep the index size manageable. Thus APSS algorithms generally index only the non-zero values of each coordinate (in contrast to LEMP). In addition, coordinates are often permuted such that dense coordinates (called prefix) appear before sparser coordinates (suffix); only the suffix is indexed. The index is used to obtain candidate vectors, which are further pruned based on properties of prefixes and suffixes [8, 7, 18]. Finally, full similarity scores are computed for each candidate.

L2AP [18] is the state-of-the-art APSS algorithm for cosine similarity search; it exploits the Euclidean norms of suffixes and prefixes for index compression and candidate filtering. L2AP can be used as a bucket algorithm for LEMP after a few modifications. In particular, we create a separate L2AP index for each bucket. In L2AP, like in most APSS algorithms, a lower bound on the cosine similarity threshold needs to be fixed a priori. In our setting, we pick the lower bound $\theta_b(\boldsymbol{q}_{max})$, where $\boldsymbol{q}_{max}$ is the query vector with the largest length.

L2AP follows a similar pruning technique to INCR during candidate generation and verification: it accumulates $\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F$ and precomputes $u(\bar{\boldsymbol{q}}_F, \bar{\boldsymbol{p}}_F)$. INCR differs in the following ways: (i) L2AP scans all indexed lists corresponding to non-zero query coordinates, whereas INCR scans only ϕ of them and only their feasible regions, (ii) L2AP uses sophisticated filtering conditions both during and after scanning. These filtering techniques eliminate the majority of the candidates, but are generally expensive. In contrast, INCR filters candidates only once and after index scanning, which is cheap but may result in a larger number of candidates. See Sec. 6 for an experimental comparison of the two methods.

In the context of APSS, approximate methods like locality-sensitive hashing (LSH) have been used for candidate pruning. A recent approach is BayesLSH-Lite [19], which uses a Bayesian approach over LSH for candidate pruning. BayesLSH-Lite constructs l signatures of k bits (hashes) and uses Bayesian inference to find a (high-probability) lower bound on the number of hash matches between the signatures of the query and the probe vector. It then computes the exact similarity value for the probe vectors with at least that many matches. Precomputation is used to speed up inference. We

[2]LSH is not applicable directly because the triangle inequality does not hold for inner products; see [11] for a discussion as well as the recent asymmetric transformations of [16, 15].

Table 1: Datasets. All with $r = 50$.

Dataset	m	n	CoV of lengths $\boldsymbol{Q}$	CoV of lengths $\boldsymbol{P}$	% Non-Zero	Naive (min)
IE-NMF	771K	132K	1.56	5.53	36.2	112.0
IE-SVD	771K	132K	1.51	4.44	100	113.0
Netflix	480K	17K	0.43	0.72	100	8.4
KDD	1000K	624K	0.38	0.40	100	2910.0

investigate the performance of BayesLSH-Lite as a pruning method within LEMP's buckets in our experimental study.

6 Experiments

The goals of our experiments were (1) to compare LEMP with existing methods for large-entry retrieval—i.e., Naive retrieval (Sec. 5), TA, and the single and dual cover tree approaches (Tree [10], D-Tree [13])—, (2) to investigate whether the combination of length- and coordinate-based pruning is effective, and (3) to study the relative performance of different bucket methods, including COORD, INCR, TA, cover trees, L2AP and BLSH-Lite. Our results can be summarized as follows:

- LEMP consistently outperformed alternative methods and was the best-performing method overall. In particular, LEMP was up to multiple orders of magnitude faster than Naive and between 2x and 20x faster than the best-performing alternative method.
- A combination of length-based and coordinate-based pruning consistently outperformed pure length-based or pure coordinate-based retrieval methods. This indicates that LEMP's approach of using bucket-specific retrieval algorithms is effective.
- The overall most effective retrieval method for LEMP was a combination of LENGTH and INCR.

6.1 Experimental Setup

Datasets and code can be found at `http://dws.informatik.uni-mannheim.de/en/resources/software/lemp`.

Hardware. All experiments were run on a machine with 48 GB RAM and an Intel Xeon 2.40GHz processor.

Algorithms. We implemented LEMP and TA in C++, and used C++ code of Tree and D-Tree provided by the authors of [10] and [13] (at `http://mlpack.org/`). For L2AP and BLSH-Lite, we adjusted publicly available C code (at `http://glaros.dtc.umn.edu/gkhome/l2ap/overview`).

We ran seven "pure" versions of LEMP, in which only one method was used within a bucket. We denote these methods as LEMP-X, where X is: L for LENGTH, C for COORD, I for INCR, TA for TA, L2AP for L2AP, BLSH for BayesLSH-Lite and Tree for cover tree. We also ran the two mixed versions LEMP-LC (LENGTH and COORD) and LEMP-LI (LENGTH and INCR), in which the appropriate retrieval method is chosen as described in Sec. 4.4. For TA, we followed common practice and selected in each step the sorted list i that maximized $q_i p_i$, where p_i refers to the next coordinate value in list i. This strategy selects the "most-promising" coordinate; we implemented it efficiently using a max-heap.

Datasets. We used real-world datasets from collaborative filtering and information extraction applications (cf. Sec. 1). Table 1 summarizes the datasets (described below in more detail); all datasets consist of 50-dimensional vectors. The table also gives the coefficient of variation (CoV) of the lengths of the input vectors and their percentage of non-zero entries; see the discussion in Sec. 6.2.

For our experiments with collaborative filtering data, we used factorizations of the popular Netflix [20] and KDD [21][3] (Yahoo! Music) datasets. Both datasets consist of ratings of users for movies (Netflix) or musical pieces (KDD); matrix-factorization techniques are state-of-the-art methods to predict unknown ratings [2, 22]. For Netflix, we performed a plain matrix factorization with DSGD++ using L2 regularization with regularization parameter $\lambda = 50$ as in [23]. For KDD, we use the factorization of [24], which incorporates the music taxonomy, temporal effects, as well as user and item biases; this dataset has been used in previous studies of the Row-Top-k problem. Since we were ultimately interested in retrieving the top-k movies/songs for each user, we used the collaborative filtering datasets to study the performance of the various methods for the Row-Top-k problem.

For the open information extraction scenario, we extracted around 16M subject-pattern-object triples from the New York Times corpus,[4] which contains news articles, using the methods described in [25]. We removed infrequent arguments and patterns, and constructed a binary argument-pattern matrix: An entry in the matrix was set to 1 if the corresponding argument (subject-object pair) occurred with the corresponding pattern; otherwise, the entry was set to 0. We factorized this binary matrix using the singular-value decomposition (SVD) and non-negative matrix factorization (NMF); we denote the resulting datasets as IE-SVD and IE-NMF, respectively. For SVD, which produces factorization $\boldsymbol{U\Sigma V}^T$, we set $\boldsymbol{Q}^T = \boldsymbol{U}\sqrt{\boldsymbol{\Sigma}}$ and $\boldsymbol{P} = \sqrt{\boldsymbol{\Sigma}}\boldsymbol{V}^T$. For the IE datasets, we studied Above-θ and Row-Top-k, which both are problems relevant in applications. Above-θ aims to find all high-confidence facts, whereas Row-Top-k retrieves the k most probable arguments of a pattern (as in [3]). For the latter problem, we make use the transposed matrices IE-SVD^T and IE-NMF^T.

Methodology. We compare the algorithms both for the Above-θ and the Row-Top-k problem. For the Row-Top-k problem, we experimented with $k = 1, 5, 10, 50$. For the Above-θ problem, we selected θ such that we retrieve the top-10^3, -10^4, -10^5, -10^6 and -10^7 entries in the whole product matrix $\boldsymbol{Q}^T\boldsymbol{P}$. We subsequently refer to the number of results as *recall*.

We compare all methods in terms of overall wall-clock time, which includes preprocessing, tuning, and retrieval time. Preprocessing involves the construction of indexes (cover trees for Tree and D-Tree, sorted lists for LEMP, TA and L2AP, hash-signatures for BLSH) and, for LEMP only, the time required for the normalization, sorting, and bucketization of the input vectors. Tuning refers to the time required to automatically select suitable values for the parameters ϕ and t_b of LEMP.

Choice of parameters. Parameters for LEMP (ϕ and t_b) were tuned on a small sample of the datasets as explained in Sec. 4.4. The base parameter of the cover trees was set to 1.3 as suggested in [13]. For all LEMP algorithms, we used a fine-grained bucketization such that all data structures of a bucket fit into the available processor cache. For

[3]corresponds to Track 1 of the 2011 KDD Cup 2011
[4]`http://catalog.ldc.upenn.edu/LDC2008T19`

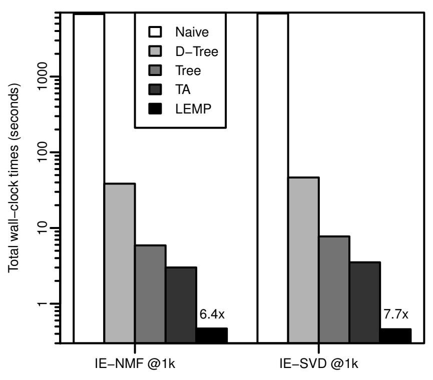


Figure 5: Total wall-clock times (incl. indexing and tuning) for Above-θ @1k recall level on different datasets

LEMP-BLSH, we obtained the best results with using only one signature. The false negative rate ϵ was set to 0.03 and the signature length to 32 as in [19]. For LEMP-L2AP, we used the same combination of filters and bounds that the authors of [18] report as most efficient w.r.t. execution time.

6.2 Comparison with Previous Methods

In this section, we compare LEMP with previous methods for finding large entries in matrix products. Figs. 5 and 6 show the relative performance of LEMP (using the LI bucket algorithm), TA, Tree, and D-Tree for the Above-θ and Row-Top-k problems, respectively. The speedup of LEMP with respect to the best-performing method other than LEMP is marked in the figures. We use Naive as a baseline; its running time is independent of θ for Above-θ and only slightly affected by k for Row-Top-k. To keep our study manageable, we only ran Naive for the Row-Top-1 problem; this is a fair comparison because running times for larger k may be slightly above but not below the times reported here. The wall-clock times for this and additional experiments, as well as average candidate set sizes, can be found in Tables 3 and 4 in the appendix. In the following, we discuss the performance of the algorithms in terms of overall running time, preprocessing time and pruning power.

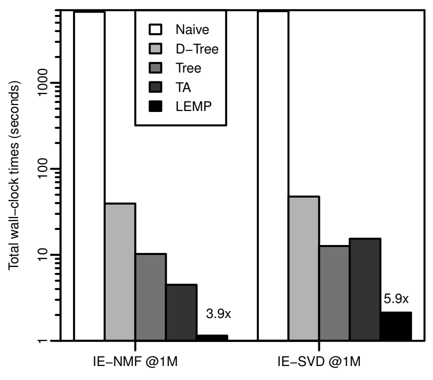
Overall Performance. Overall, LEMP was the fastest method, reaching 14572x speedup over the Naive baseline and up to 22.3x over the next best method. The second fastest method was in the majority of cases Tree, followed by TA and D-Tree. LEMP, Tree and TA appear to have best performance on datasets with large skew in their length distribution, like the IE datasets (high CoV in Table 1) and also on datasets with sparse vectors (IE-NMF). On datasets with little skew in their length distribution, like Netflix and KDD, all methods had difficulties in providing large speedups over Naive. However, for KDD, some methods were still able to offer significant savings in terms of running time: LEMP was able to save up to 41.2 hours of computation time in comparison to naive retrieval. Tables 3 and 4 show that the performance of all methods but Naive deteriorates as the result size or k increases (θ decreases), since the output size increases and pruning opportunities decrease. Generally, there is a break-even point at which any method will be slower than naive. This is the case, for example, for TA on Netflix/KDD and D-Tree on Netflix for $k \geq 1$.

Preprocessing Time. Table 2 shows the preprocessing time for the different datasets and methods. For Tree and D-Tree, we give the wall-clock time of producing the cover tree(s) and for TA the time to create the sorted lists. The preprocessing costs of these methods are fixed and depend on the size of probe matrix (and additionally of the query matrix for D-Tree). For LEMP-LI, we report the sum of maximum indexing and the maximum tuning time (normally the preprocessing times vary from problem to problem since LEMP constructs indexes lazily). LEMP, on the one hand, has the overhead of tuning and, on the other hand, has the benefit of lazy index construction, specially for datasets that have skewed length distribution. The larger the length skew and the size of the probe matrix, the larger the preprocessing savings of LEMP over the other methods and the higher the chances of outweighing the tuning overhead. For example for IE-NMF^T, which has $n = 771K$, LEMP needed 0.92s vs 2.98s for TA and 31.84s for Tree. The highest costs appeared for the Tree and D-Tree methods. Preprocessing costs can be one of the major bottlenecks for these methods. Tables 3 and 4 show that preprocessing can be a large part of the overall running time, specially for datasets with large length skew. For example, D-Tree needed more time to create its trees for Netflix (tree construction was 70% of the overall time) than Naive needed to retrieve the Row-Top-1 entries. Similarly, for the IE datasets (recall $\leq 10^6$, $k \leq 10$), LEMP (and in some cases also TA) terminated before the Tree method finished preprocessing.

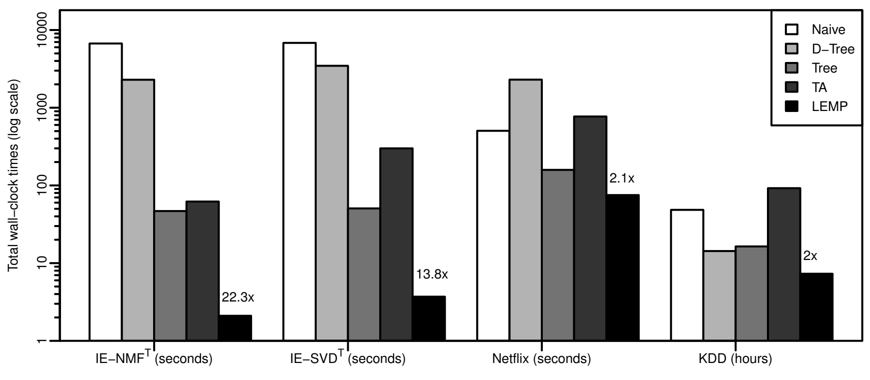
Pruning Power. Tables 3 and 4 show how many candidates remain on average after pruning for each of the different methods. For the Row-Top-k problem, LEMP had the highest pruning power for the IE datasets and for small k for KDD, whereas for Netflix Tree pruned more candidates (and LEMP was second-best). For example, for Netflix, $k = 1$, Tree produced only 1661 candidates per query on average, whereas LEMP produced 1881. Nevertheless, LEMP was faster overall (75.1s vs 158.5s). This speed-up was not due to preprocessing costs (0.24s for Tree vs 0.60s for LEMP), but mainly due to LEMP's efficient pruning.

TA ranks usually last in terms of pruning power. Especially for datasets with low length skew, TA tends to perform poorly. For example, for Netflix, $k = 1$, TA pruned only half of the vectors (8920 candidates per query, out of a total of 17770). We also see the effects of TA's random memory access pattern here: Although TA verified half the number of candidates as Naive, it was 1.5x slower (771.1s vs 504.3s). Also note that sparsity affects the behavior of TA: It checked 4.6x less candidates for the sparse IE-NMF^T dataset, $k = 1$, than for IE-SVD^T (1977 vs. 9226 candidates per query). The main reason for the relatively low pruning power of TA for dense datasets is that it is length-oblivious, i.e., it checks short probe vectors if they have a single, sufficiently large coordinate; these vectors are discarded by LEMP. On the other hand, for sparse datasets, large values for individual coordinates correlate well with the length of the vectors so that essentially TA explores long vectors first. We expect that a combination of LEMP and TA can address the problems of length-obliviousness and random memory accesses (see next section).

For the D-Tree, given a fixed, high θ value (as in the Above-θ problem), the grouping of queries helps to reduce the frequency of visits of the probe-tree nodes (and thus the candidate checking). D-Tree was actually able to prune more

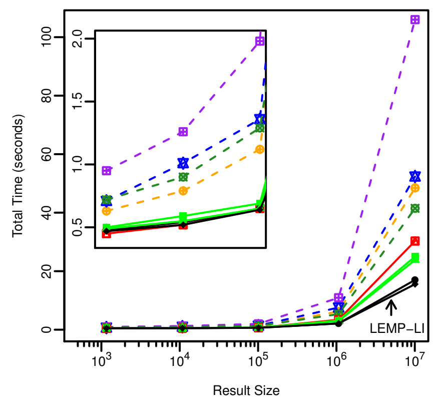


(a) Above-θ @1M

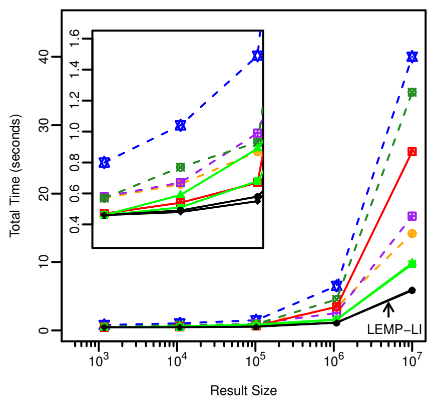


(b) Row-Top-1

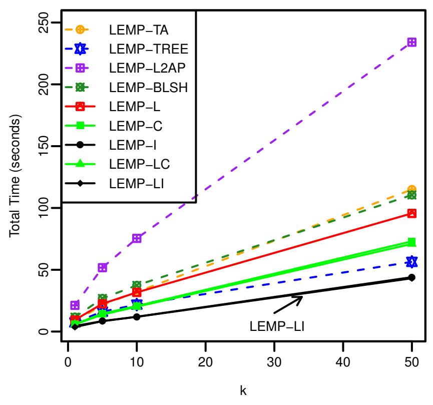
Figure 6: Total wall-clock times (incl. indexing and tuning) for Above-θ @1M and Row-Top-1 on different datasets



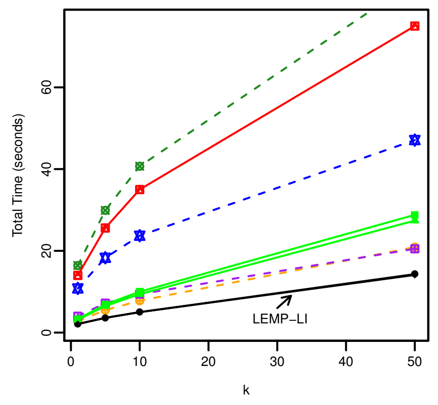
(a) Above-θ IE-SVD



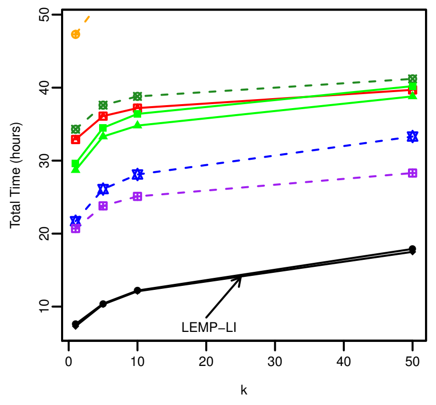
(b) Above-θ IE-NMF



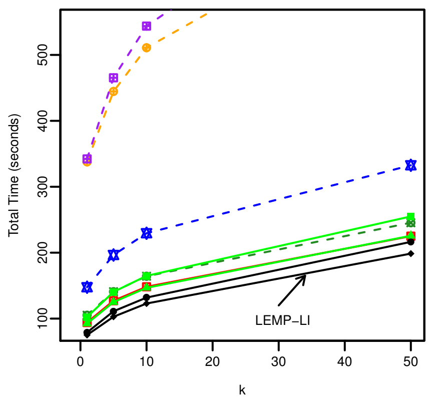
(c) Row-Top-k IE-SVD^T



(d) Row-Top-k IE-NMF^T



(e) Row-Top-k KDD



(f) Row-Top-k Netflix

Figure 7: Comparison of LEMP bucket-algorithms in terms of total wall-clock times (incl. indexing and tuning)

candidates than all other methods for this problem. For the top-k case, the bounds for a group of queries depend on the worst running lower bound θ' among all queries of the group. Thus, for the top-k problem, D-Tree had usually looser bounds and, therefore, less pruning power than Tree.

Caching effects. Recall that LEMP does not create buckets that exceed the cache size. To study the effect of this approach, we experimented with a cache-oblivious version of LEMP in which bucket sizes were unrestricted. We found that for datasets with large length skew, runtime differences were marginal: LEMP creates small buckets anyway

Table 2: Maximum preprocessing times (in seconds) including indexing and tuning.

Dataset	LEMP	TA	Single Tree	Dual Tree
IE-NMF	0.82	0.46	5.33	37.24
IE-SVD	1.22	0.76	7.1	44.5
IE-NMF^T	0.92	2.98	31.84	37.1
IE-SVD^T	1.61	4.88	37.5	44.5
Netflix	1.33	0.09	0.24	1554.9
KDD	115	4.47	204	2022.1

when lengths are skewed. For datasets with less length skew, such as KDD, there was a significant difference in runtime: LEMP created more than 15x more buckets than its cache-oblivious version (26 vs. 403), and was more than twice as fast (16.7h vs. 7.3h).

6.3 Relative Performance of Bucket Algorithms

Fig. 7 shows the relative performance of LEMP's bucket-algorithms. Wall-clock times for all experiments and average candidate set sizes can be found in Tables 5 and 6 in the appendix. In the following, we discuss the performance of each algorithm in turn.

LEMP-L. For the IE datasets, LEMP-L was able to reduce the average candidate set size around 98% (13211 candidates per query vs. 771611 for Naive, IE-SVD^T, $k = 50$), whereas for datasets with less length skew the reduction ranged between 52% and 79% (Netflix) and 14% and 28% (KDD). Overall, LEMP-L was able to provide significant speedup over Naive: up to 15064x (722x) for IE-SVD and 14267x (479x) for IE-NMF for Above-θ (Row-Top-k). Actually, LEMP-L outperformed all other methods for IE-NMF, IE-SVD and result sizes $\leq 100K$. I.e., bucket pruning was very effective for the datasets with large length skew. This indicates that LEMP's separate treatment of short and long vectors is beneficial. The performance of LEMP-L acts as a baseline for the performance of other bucket algorithms: LEMP-L's main filtering mechanism is bucket-level pruning, which is common for all LEMP methods.

LEMP-C, LEMP-I. COORD created up to 7x less candidates per query than LEMP-L (e.g., 271 vs. 1915 for IE-NMF^T, $k = 1$) and its speedup over LEMP-L ranged between 0.7x and 4.3x. INCR reduced the candidates even further (42 candidates for IE-NMF^T, $k = 1$, 46x less than LEMP-L) with up to 6.6x speedup. The difference in the pruning power of COORD and INCR was more prevalent in the case of the KDD dataset (369424 vs. 43160 candidates per query). In the absence of large length skew or sparsity, INCR accumulates as much information as possible for the probe vectors. COORD, on the other hand, is not able to fully take advantage of all the available information. Actually, INCR was the best performing method (when LEMP was used together with only 1 bucket-algorithm) in terms of running time for the majority of datasets and configurations.

LEMP-LI. As discussed above, LEMP-L was the best performing method for datasets with high length skew on small recall levels. On the other hand, LEMP-I showed superior behavior in all other cases. LEMP-LI, for a small extra tuning cost, combines the strong points of both methods. In the majority of cases, it was the fastest method overall. In the remaining cases, the performance of LEMP-LI was similar to that of the best-performing method.

LEMP-TA. LEMP-TA was also able to offer speedup over LEMP-L: up to 2.1x for the Above-θ and up to 5.2x for Row-Top-k. However, it was usually outperformed by COORD and INCR: e.g., LEMP-I was between 1.3x and 3.4x faster. The reason for INCR's superior behavior is that TA is usually not possible to identify good candidates by observing the value of only one coordinate. INCR avoids this problem by gathering information about the vectors from multiple coordinates (lists); based on this information, it prunes as many candidates as possible before actually calculating an inner product. Even for cases that TA prunes more candidates than INCR (e.g., IE-NMF @$10M$) it was slower than INCR, potentially due to the overhead of internally maintaining a max-heap for selecting the best list to explore next. Notice also that LEMP-TA was significantly faster (up to 24.9x for IE-SVD^T, $k = 50$) than the standard TA algorithm, since the length-obliviousness and cache-misses problems are addressed by LEMP. This indicates that a method like LEMP might improve the performance of TA when linear scoring functions are used.

LEMP-L2AP. LEMP-L2AP was the method with the most aggressive pruning for all datasets (e.g., only 18 candidates per query for KDD, $k = 1$). However, this extensive pruning has a high cost: L2AP scans all the lists in the index that correspond to non-zero query coordinates and checks the filtering conditions during and after scanning. Also, the actual threshold used when querying the index can be far away from the lower bound used during index creation, which affects scanning time. For these reasons, INCR consistently outperformed L2AP (1.3x to 6.2x faster).

LEMP-BLSH. As in the case of LEMP-L2AP, the minimum number of hash matches required for a bucket are precomputed by BLSH-Lite based on the largest local threshold, which limits pruning. In our experiments, LEMP-BLSH was able to create on average only up to 0.3% less candidates per query than LEMP-L. This marginal increase in pruning power was unable to outweigh the costs of LSH hashing and Bayesian inference. As a result, LEMP-BLSH was consistently slower than plain LEMP-L (up to 1.6x).

LEMP-Tree. LEMP-Tree creates one tree per bucket (lazy construction), instead one tree from the entire probe dataset. This explains why LEMP-Tree had much better performance than Tree (up to 10.5x faster) for the datasets for which preprocessing was Tree's bottleneck (see Above-θ experiments, small result sizes). In terms of pruning power, LEMP-Tree did not have a consistent behavior w.r.t. Tree. For datasets with large length skew (IE-NMF^T, IE-SVD^T) it checked less candidates per query, whereas for datasets for small skew (Netflix, KDD) it checked more. I.e., in cases where the pruning power of Tree is limited (e.g., KDD), checking multiple trees instead of a single one may slow down retrieval further.

7 Conclusion

We proposed LEMP, a novel algorithm for fast retrieval of large entries in a matrix product. LEMP exploits both the length and the direction information in the data: It bucketizes the input vectors according to their length, prunes buckets that cannot contribute to the result, and dynamically selects a suitable retrieval algorithms for the remaining buckets. Our experiments indicate that LEMP provides significant speedup over the naive approach (up to 14,000x) as well as over state-of-the-art techniques (up to 20x).

8 References

[1] D. Skillicorn, *Understanding complex datasets: data mining with matrix decompositions.* Taylor & Francis Ltd, 2007.

[2] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[3] S. Riedel, L. Yao, B. M. Marlin, and A. McCallum, "Relation extraction with matrix factorization and universal schemas," in *HLT-NAACL*, 2013.

[4] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," in *PODS*, 2001, pp. 102–113.

[5] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007, pp. 131–140.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006.

[7] D. Lee, J. Park, J. Shim, and S.-g. Lee, "An efficient similarity join algorithm with cosine similarity predicate," in *DEXA: Part II*, 2010, pp. 422–436.

[8] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008, pp. 131–140.

[9] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *VLDB*, 1999, pp. 518–529.

[10] R. R. Curtin, P. Ram, and A. G. Gray, "Fast exact max-kernel search," in *SDM*, 2013, pp. 1–9.

[11] P. Ram and A. G. Gray, "Maximum inner-product search using cone trees," in *KDD*, 2012, pp. 931–939.

[12] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *ICML*, 2006, pp. 97–104.

[13] R. R. Curtin and P. Ram, "Dual-tree fast exact max-kernel search," *Statistical Analysis and Data Mining*, pp. 229–253, 2014.

[14] Z. Zhang, Q. Wang, L. Ruan, and L. Si, "Preference preserving hashing for efficient recommendation," in *SIGIR*, 2014, pp. 183–192.

[15] A. Shrivastava and P. Li, "Asymmetric LSH (ALSH) for sublinear time maximum inner product search (MIPS)," in *NIPS*, 2014, pp. 2321–2329.

[16] Y. Bachrach, Y. Finkelstein, R. Gilad-Bachrach, L. Katzir, N. Koenigstein, N. Nice, and U. Paquet, "Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces," in *RecSys*, 2014, pp. 257–264.

[17] N. Koenigstein, P. Ram, and Y. Shavitt, "Efficient retrieval of recommendations in a matrix factorization framework," in *CIKM*, 2012, pp. 535–544.

[18] D. C. Anastasiu and G. Karypis, "L2ap: Fast cosine similarity search with prefix l-2 norm bounds," in *ICDE*, 2014.

[19] V. Satuluri and S. Parthasarathy, "Bayesian locality sensitive hashing for fast similarity search," *Proc. VLDB Endow.*, vol. 5, no. 5, pp. 430–441, Jan. 2012.

[20] J. Bennett and S. Lanning, "The Netflix prize," in *KDD Cup and Workshop*, 2007.

[21] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup'11," *JMLR Workshop and Conference Proceedings*, vol. 18, 2012.

[22] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the Netflix Prize," in *AAIM*, 2008, pp. 337–348.

[23] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *ICDM*, 2012.

[24] N. Koenigstein, G. Dror, and Y. Koren, "Yahoo! music recommendations: Modeling music ratings with temporal dynamics and item taxonomy," in *RecSys*, 2011, pp. 165–172.

[25] N. Nakashole, G. Weikum, and F. Suchanek, "Patty: A taxonomy of relational patterns with semantic types," in *EMNLP-CoNLL*, 2012, pp. 1135–1145.

APPENDIX

A Implementation Details

In this section, we give some guidance on how to implement the COORD and INCR algorithms efficiently.

COORD. In our implementation, we store the sorted-list indexes column-wise to reduce memory bandwidth: the data values are accessed only during binary search to determine the scan range, and the local identifiers are accessed only during the actual scan phase. For efficiency reasons, we also avoid clearing the CP array when moving from one query vector to the next. Instead, we keep the array uninitialized and proceed as follows. When scanning the first sorted list, we set to 1 instead of incrementing the corresponding entry of the CP array and increment while scanning the remaining sorted lists. After all lists have been scanned, we scan the first sorted list again and only consider the corresponding entries of the CP array for inclusion into the candidate set. Since the first sorted list is scanned twice (for CP array initialization and filtering), we take the focus coordinate with fewest elements as the first one.

INCR. Since INCR needs access to both coordinate values and local identifiers during scanning, we store the sorted lists row-wise. The extended CP array is initialized and accessed in the same way as the CP array of COORD. In order to reduce memory bandwidth and avoid excessive checking, in the extended CP array we do not keep the counter information used in COORD: the filtering condition of Eq. (5) is usually pruning vectors more aggressively than the simple check of COORD. Since Eq. (5) contains expensive floating-point operations (such as divisions and square roots), we rewrite the conditions and accept a vector $\bar{\boldsymbol{p}}$ if:

$$\bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| > \theta / \|\boldsymbol{q}\|,$$

for which the right-hand side needs to be computed only once. If this test fails, we accept $\bar{\boldsymbol{p}}$ if and only if:

$$\|\boldsymbol{p}\|^2 \|\boldsymbol{q}\|^2 (1 - \|\bar{\boldsymbol{p}}_F\|^2)(1 - \|\bar{\boldsymbol{q}}_F\|^2) \geq (\theta - \bar{\boldsymbol{q}}_F^T \bar{\boldsymbol{p}}_F \|\boldsymbol{p}\| \|\boldsymbol{q}\|)^2.$$

INCR's strength lies into accumulating partial inner products from many lists. If we decide to use $\phi_b = 1$ for some bucket b, INCR and COORD will produce the same candidate set, but COORD does so faster. We thus use COORD instead of INCR whenever $\phi_b = 1$.

B Additional Experiments

Tables 3 - 6 show running times for Above-θ and Row-Top-k experiments for different retrieval levels and values of k (including those presented in the figures of Sec. 6).

Table 3: Comparison of LEMP with state-of-the-art algorithms for the Above-θ problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small result sizes, resp.).

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M		Preprocessing
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	time [%]
IE-SVD	Naive	6825	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	7.74	*(2.1)*	7.96	*(3.1)*	8.36	*(4.9)*	12.67	*(30.8)*	50.74	*(236.8)*	15% – 95%
	D-Tree	46.47	***(0.2)***	46.71	***(0.4)***	46.73	***(0.7)***	47.60	***(5.9)***	58.21	***(66.2)***	80% – 99%
	TA	3.52	*(2.6)*	3.69	*(4.1)*	4.00	*(6.8)*	15.39	*(107.0)*	453.00	*(3033.0)*	<1% – 21%
	LEMP-LI	**0.47**	*(1)*	**0.52**	*(0.4)*	**0.64**	*(3.8)*	**2.13**	*(21.2)*	**15.57**	*(235.5)*	8% – 90%
IE-NMF	Naive	6703.2	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-	*(132K)*	-
	Tree	5.89	*(2.3)*	6.10	*(3.3)*	6.50	*(5.2)*	10.24	*(29.4)*	39.41	*(185.4)*	14% – 93%
	D-Tree	38.51	***(0.2)***	38.64	*(0.3)*	38.79	*(0.7)*	39.47	***(5.1)***	47.04	***(46.6)***	82% – 99%
	TA	3.01	*(1.5)*	3.08	*(1.8)*	3.17	*(2.3)*	4.49	*(12.6)*	26.39	*(175.9)*	2% – 16%
	LEMP-LI	**0.46**	*(0.7)*	**0.48**	***(0.2)***	**0.55**	***(0.5)***	**1.15**	***(5.1)***	**5.95**	*(51.4)*	14% – 88%

Table 4: Comparison of LEMP with state-of-the-art algorithms for the Row-Top-k problem w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses. The last column gives the preprocessing time as minimum/maximum percentage of the overall time (occurs at large/small k, resp.).

Dataset	Algorithm	k=1		k=5		k=10		k=50		Preprocessing
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	time [%]
IE-SVD^T	Naive	6825	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	50.6	*(357)*	63.7	*(772)*	75.1	*(1119)*	158.5	*(3213)*	25% – 77%
	D-Tree	3464.4	*(24539)*	3246.1	*(23837)*	3224.7	*(24016)*	3234.2	*(25730)*	1.3% – 1.4%
	TA	300	*(9226)*	854.5	*(25885)*	1190.0	*(35717)*	2841.1	*(84056)*	<1% – 1.6%
	LEMP-LI	**3.7**	***(145)***	**8.4**	***(296)***	**11.9**	***(541)***	**43.1**	***(1957)***	3.7% – 15.4%
IE-NMF^T	Naive	6703.2	*(771K)*	-	*(771K)*	-	*(771K)*	-	*(771K)*	-
	Tree	46.8	*(465)*	58.2	*(845)*	67.3	*(1107)*	122.5	*(2591)*	27% – 70%
	D-Tree	2291.6	*(16574)*	2107.0	*(15811)*	2049.5	*(15652)*	1999.4	*(16104)*	1.7% – 1.9%
	TA	62.0	*(1977)*	171.4	*(5510)*	275.2	*(7542)*	462.4	*(14672)*	<1% – 5%
	LEMP-LI	**2.1**	***(35)***	**3.6**	***(92)***	**5.0**	***(161)***	**14.1**	***(357)***	7% – 23%
Netflix	Naive	504.3	*(17K)*	-	*(17K)*	-	*(17K)*	-	*(17K)*	-
	Tree	158.5	***(1661)***	219.9	***(2289)***	259.8	***(2671)***	419.4	***(4045)***	<1%
	D-Tree	2297.8	*(3789)*	>Naive	-	>Naive	-	>Naive	-	- – 70%
	TA	771.1	*(8920)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-LI	**75.1**	*(1881)*	**102.8**	*(2583)*	**122.9**	*(2973)*	**198.6**	*(5604)*	<1%
KDD	Naive	48.5h	*(624K)*	-	*(624K)*	-	*(624K)*	-	*(624K)*	-
	Tree	16.4h	*(86K)*	20.8h	*(110K)*	27.4h	*(122K)*	29.6h	*(155K)*	<1%
	D-Tree	14.3h	*(72K)*	18.0h	*(91K)*	20h	*(101K)*	25.2h	***(129K)***	2.3% – 4%
	TA	90.3h	*(615K)*	>Naive	-	>Naive	-	>Naive	-	<1%
	LEMP-LI	**7.3h**	***(42K)***	**10.3h**	***(77K)***	**12.1h**	***(97K)***	**17.5h**	*(172K)*	<1%

Table 5: Comparison of LEMP bucket algorithms for the Above-θ problem w.r.t. wall-clock time (in seconds). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	@1K		@10K		@100K		@1M		@10M	
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$
IE-SVD	LEMP-L	**0.45**	*(1.2)*	**0.52**	*(2.6)*	0.65	*(5.5)*	3.40	*(71.4)*	30.32	*(716.2)*
	LEMP-LI	0.47	*(1)*	**0.52**	*(0.4)*	**0.64**	*(3.8)*	**2.13**	*(21.2)*	**15.57**	*(235.5)*
	LEMP-LC	0.49	*(1.1)*	0.55	*(2.1)*	0.65	*(3.8)*	2.97	*(43.0)*	23.82	*(436.1)*
	LEMP-I	0.48	*(0.2)*	0.54	*(0.4)*	0.65	*(1.1)*	**2.13**	*(15.4)*	17.03	*(233.5)*
	LEMP-C	0.50	*(1.6)*	0.59	*(2.9)*	0.69	*(4.7)*	3.10	*(43.4)*	25.00	*(432.2)*
	LEMP-TA	0.64	*(0.5)*	0.78	*(1.0)*	1.10	*(1.7)*	6.10	*(16.6)*	46.44	*(299.3)*
	LEMP-Tree	0.71	*(1.7)*	1.01	*(3.1)*	1.36	*(5.8)*	7.58	*(46.6)*	52.35	*(320.5)*
	LEMP-L2AP	0.95	***(0.1)***	1.26	***(0.1)***	1.98	***(0.4)***	10.91	***(1.5)***	105.99	***(14.2)***
	LEMP-BLSH	0.72	*(1.2)*	0.90	*(2.6)*	1.29	*(5.5)*	5.33	*(71.4)*	41.48	*(716.2)*
IE-NMF	LEMP-L	0.47	*(1.5)*	0.54	*(2.9)*	0.67	*(5.8)*	3.46	*(70.0)*	26.14	*(614.0)*
	LEMP-LI	**0.46**	*(0.7)*	**0.48**	*(0.2)*	**0.55**	*(0.5)*	**1.15**	*(5.1)*	**5.95**	*(51.4)*
	LEMP-LC	**0.46**	*(0.8)*	0.59	*(1.2)*	0.89	*(2.6)*	1.60	*(16)*	9.86	*(148.8)*
	LEMP-I	**0.46**	***(0.1)***	0.49	***(0.2)***	0.58	***(0.4)***	1.13	*(4.3)*	5.85	*(44.6)*
	LEMP-C	0.46	*(0.8)*	0.51	*(1.2)*	0.68	*(2.6)*	1.58	*(15.9)*	9.67	*(148.6)*
	LEMP-TA	0.55	*(0.2)*	0.72	*(0.3)*	0.91	*(0.6)*	2.96	*(3.5)*	12.66	*(36.5)*
	LEMP-Tree	0.80	*(2.7)*	1.04	*(4.4)*	1.49	*(7.6)*	6.52	*(43.6)*	40.00	*(251.1)*
	LEMP-L2AP	0.58	***(0.1)***	0.67	***(0.2)***	0.99	***(0.4)***	2.55	***(1.5)***	16.71	***(14.1)***
	LEMP-BLSH	0.57	*(1.5)*	0.77	*(2.9)*	0.93	*(5.8)*	4.53	*(70.0)*	34.81	*(614.0)*

Table 6: Comparison of LEMP bucket algorithms for the Row-Top-k problem w.r.t. wall-clock time (in seconds, unless stated otherwise). We give the average candidate set size per query in parentheses.

Dataset	Algorithm	k=1		k=5		k=10		k=50	
		Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$	Time	$\|C\|/q$
IE-SVD^T	LEMP-L	9.4	*(1272)*	22.5	*(3103)*	31.7	*(4386)*	95.6	*(13211)*
	LEMP-LI	**3.7**	*(145)*	**8.4**	*(296)*	**11.9**	*(541)*	**43.1**	*(1957)*
	LEMP-LC	5.8	*(539)*	13.7	*(1469)*	20.1	*(2188)*	70.9	*(7811)*
	LEMP-I	4.5	*(74)*	8.5	*(310)*	12.0	*(581)*	44.0	*(2402)*
	LEMP-C	6.0	*(541)*	14.5	*(1418)*	20.7	*(2143)*	73.0	*(7803)*
	LEMP-TA	8.1	*(397)*	20.9	*(1145)*	31.5	*(1760)*	114.1	*(6547)*
	LEMP-Tree	7.4	*(217)*	16.1	*(544)*	21.8	*(759)*	56.4	*(2079)*
	LEMP-L2AP	21.2	***(15)***	51.7	***(24)***	75.4	***(35)***	234.1	***(130)***
	LEMP-BLSH	11.7	*(1271)*	26.8	*(3102)*	37.4	*(4385)*	110.5	*(13210)*
IE-NMF^T	LEMP-L	14.0	*(1915)*	25.6	*(3541)*	35.0	*(4869)*	75.0	*(10319)*
	LEMP-LI	**2.1**	*(35)*	**3.6**	*(92)*	**5.0**	*(161)*	**14.1**	*(357)*
	LEMP-LC	3.1	*(274)*	6.5	*(662)*	9.4	*(1010)*	27.4	*(3041)*
	LEMP-I	**2.1**	*(42)*	**3.6**	*(114)*	**5.0**	*(155)*	14.3	*(397)*
	LEMP-C	3.3	*(271)*	6.9	*(659)*	10.0	*(1002)*	28.8	*(3039)*
	LEMP-TA	2.7	*(69)*	5.4	*(225)*	7.7	*(363)*	20.7	*(1154)*
	LEMP-Tree	10.8	*(351)*	18.3	*(649)*	23.7	*(865)*	47.1	*(1735)*
	LEMP-L2AP	4.0	***(11)***	7.2	***(25)***	9.4	***(33)***	20.5	***(127)***
	LEMP-BLSH	16.4	*(1908)*	29.9	*(3537)*	40.7	*(4866)*	86.0	*(10318)*
Netflix	LEMP-L	94.2	*(3673)*	127.1	*(4922)*	148.3	*(5719)*	225.1	*(8416)*
	LEMP-LI	**75.1**	*(1881)*	**102.8**	*(2583)*	**122.9**	*(2973)*	**198.6**	*(5604)*
	LEMP-LC	92.4	*(3332)*	125.0	*(4542)*	146.7	*(5300)*	225.7	*(7903)*
	LEMP-I	79.0	*(2045)*	111.1	*(2755)*	131.9	*(3528)*	216.4	*(5682)*
	LEMP-C	102.4	*(3323)*	140.5	*(4528)*	164.8	*(5288)*	255.0	*(7951)*
	LEMP-TA	336.5	*(3707)*	444.0	*(4912)*	508.9	*(5666)*	736.5	*(8245)*
	LEMP-Tree	147.6	*(1727)*	196.5	*(2314)*	229.5	*(2636)*	332.6	*(3883)*
	LEMP-L2AP	342.2	***(10)***	465.3	***(28)***	543.7	***(48)***	819.4	***(198)***
	LEMP-BLSH	105.1	*(3673)*	140.8	*(4922)*	164.3	*(5719)*	245.4	*(8416)*
KDD	LEMP-L	32.9h	*(445K)*	36.1h	*(488K)*	37.2h	*(504K)*	39.7h	*(537K)*
	LEMP-LI	**7.3h**	*(42K)*	**10.3h**	*(77K)*	**12.1h**	*(97K)*	**17.5h**	*(172K)*
	LEMP-LC	28.7h	*(369K)*	33.3h	*(430K)*	34.8h	*(456K)*	38.8h	*(504K)*
	LEMP-I	7.6h	*(43K)*	10.4h	*(78K)*	12.2h	*(96K)*	17.9h	*(175K)*
	LEMP-C	29.6h	*(369K)*	34.5h	*(433K)*	36.4h	*(455K)*	40.2h	*(503K)*
	LEMP-TA	47.3h	*(440K)*	52.3h	*(483K)*	>Naive	-	>Naive	-
	LEMP-Tree	21.7h	*(204K)*	26.1h	*(247K)*	28.1h	*(266K)*	33.3h	*(317K)*
	LEMP-L2AP	20.7h	***(18)***	23.8h	***(71)***	25.1h	***(122)***	28.3h	***(452)***
	LEMP-BLSH	34.3h	*(445K)*	37.6h	*(488K)*	38.8h	*(504K)*	41.2h	*(537K)*