# Optimizing Hyper-Phrase Queries

Dhruv Gupta
Max Planck Institute for Informatics, Germany

Klaus Berberich
Max Planck Institute for Informatics & htw saar, Germany

## ABSTRACT

A hyper-phrase query (HPQ) consists of a sequence of phrase sets. Such queries naturally arise when attempting to spot knowledge graph (KG) facts or sets of KG facts in large document collections to establish their provenance. Our approach addresses this challenge by proposing query operators to detect text regions in documents that correspond to the HPQ as combinations of n-grams and skip-grams. The optimization lies in identifying the most cost-efficient order of query operators that can be executed to identify the text regions containing the HPQ. We show the efficiency of our optimizations on spotting facts from Wikidata in document collections amounting to more than thirty million documents.

**ACM Reference Format:**
Dhruv Gupta and Klaus Berberich. 2020. Optimizing Hyper-Phrase Queries. In *Proceedings of the 2020 ACM SIGIR International Conference on the Theory of Information Retrieval (ICTIR '20), September 14–17, 2020, Virtual Event, Norway.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3409256.3409827

## 1 INTRODUCTION

To assist journalists and scholars in humanities in verifying and validating facts, large knowledge graphs (KGs) such as Wikidata have started to substantiate facts concerning named entities with references to news articles or scientific reports available on the Web. However, there still exist many facts in Wikidata that are entered manually without any references or provenance (e.g., see Figure 1). In order to establish the origin of KG facts, we need to retrieve their evidence from the Web or in large document collections. Simply spotting a single fact (e.g., ⟨BILL CLINTON, SPOUSE, HILLARY CLINTON⟩) is a time consuming task, unless we leverage the common sub-phrases underlying the surface forms in the triple.

The process of validating and verifying facts about named entities relies on spotting them in text documents (e.g., news articles or on the Web). In such scenarios, hyper-phrase queries naturally manifest themselves. The problem of spotting KG facts becomes even more relevant when trying to verify false facts about emerging named entities. For emerging named entities it is impossible to apriori annotate large document collections using NLP tools. For example, named entity recognition and disambiguation tools will fail to annotate emerging entities as the KG does not contain their canonical entries for disambiguation. Furthermore, to assist downstream applications such as knowledge acquisition and question answering for emerging named entities, it is important that we can execute such hyper-phrase queries efficiently and at scale.

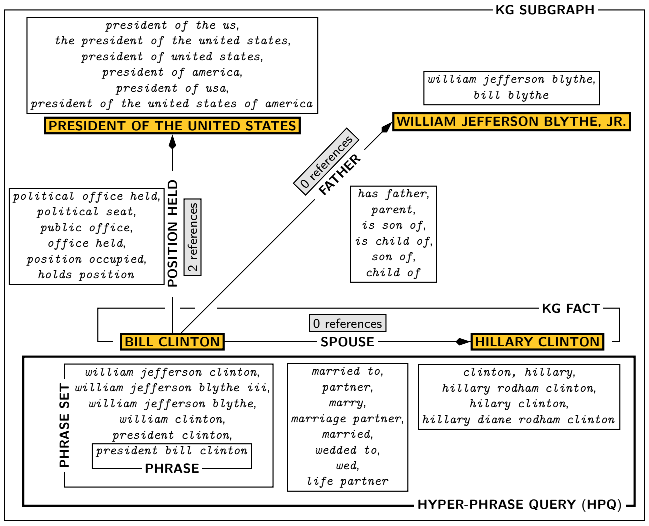

**Figure 1: KG facts regarding BILL CLINTON from Wikidata. Note that there are no references establishing the provenance of two facts.**

## 2 PROBLEM DEFINITION

We now define the basic elements underlying the problem of executing hyper-phrase queries. Key concepts related to the definitions are illustrated in Figure 1. Consider a large document collection, $\mathcal{D} = \{d_1, d_2, \ldots, d_{|\mathcal{D}|}\}$. Each document in the collection $d \in \mathcal{D}$ is a sequence of sentences $d = \langle s_1, s_2, \ldots, s_{|d|} \rangle$. Each sentence further consists of a sequence of words drawn from the vocabulary $\Sigma$ associated with the collection.

**Phrase Query**. A basic *phrase query* is a sequence of words that must be matched contiguously in a document. For example, consider the phrase query: ⟨*bill clinton is son of bill blythe*⟩. The entire phrase with each word occurring in that sequence must be matched in the retrieved document. Formally, a phrase query $p$ can be defined as:

$$p \in \Sigma^+. \tag{1}$$

A document is a match for a phrase query $p$ if it contains a sentence in which the words of the phrase occur contiguously. Formally, a sentence $s$ matches the phrase $p$ if

$$p \sqsubset s \quad \equiv \quad \exists 1 \leq i \leq |s| : \forall 1 \leq l \leq |p| : s[i+l-1] = p[l], \tag{2}$$

and a document $d$ matches a phrase if

$$p \sqsubset d \quad \equiv \quad \exists s \in d : p \sqsubset s. \tag{3}$$

**Phrase-Set Query**. A *phrase-set* query is one that combines multiple phrases, for example, consisting of different paraphrases to increase recall in document retrieval. Consider, as a concrete example, the phrase set query $\{\langle$*alumni of*$\rangle\langle$*attended college*$\rangle\}$. Formally, a phrase set query $P$ can be defined as a subset of all possible phrases that can be generated from the vocabulary:

$$P = \{p_1, p_2, \ldots, p_{|P|}\} \subseteq \Sigma^+. \tag{4}$$

A document is considered a match for a phrase-set query if at least one of the phrases in the set is found in the document according to Equation 3. Concretely, this can be put as follows:

$$P \sqsubset d \equiv \exists p \in P : p \sqsubset d. \tag{5}$$

**Hyper-Phrase Query (HPQ).** A HPQ is defined to be a sequence of phrase sets. An example of a HPQ corresponding to a KG fact is shown in Figure 1. Formally, a HPQ is a sequence of phrase sets:

$$\mathcal{P} = \langle P_1, P_2, \ldots, P_{|\mathcal{P}|} \rangle. \tag{6}$$

A document is said to match a HPQ if one phrase from each phrase set is matched, and matches for adjacent phrase sets occur within $k$ sentences from each other in the document. Formally:

$$\begin{aligned} \mathcal{P} \sqsubset d \;\equiv\; & \exists 1 \le i_1 \le \ldots \le i_{|\mathcal{P}|} : \\ & \forall 1 \le j \le |\mathcal{P}| : \exists p \in P_j : p \sqsubset s_{i_j} \wedge \\ & \forall 1 < j \le |\mathcal{P}| : (s_{i_{j+1}} - s_{i_j}) \le k \end{aligned} \tag{7}$$

The definition ensures that phrase matches across different sentences have to occur in the order specified by the HPQ. Should more than one phrase set be matched by the same sentence, we additionally ensure that their order is respected within the sentence. To reduce formalism, we omit this detail from the above definition.

**Establishing Provenance for KG Facts.** Consider the problem of spotting knowledge graph (KG) facts on the Web or in large document collections [19] as a concrete use case of matching hyper-phrase queries. A KG consists of facts. Each fact consists of vertices that are either named entities or literals and edges that define relationships between them. The facts are usually encoded in the form of a triple, which consists of two vertices (either named entities or literals) and an edge (predicate or relationship). The triples can be succinctly represented as ⟨(S)UBJECT, (P)REDICATE, (O)BJECT⟩. Each component of the triple is a canonical representation of its various surface forms. Let these surface forms of S, P, and O be denoted by: $\{s_1, s_2, \ldots, s_u\}$, $\{p_1, p_2, \ldots, p_v\}$, and $\{o_1, o_2, \ldots, o_w\}$ respectively. A **text region** is considered an **evidence** (thereby establishing provenance for the fact) if it contains at least one phrase from each of the phrase sets within a distance of $k$ sentences and with the particular order as expressed in the fact.

**Problem Difficulty.** Consider the problem of retrieving documents for a HPQ $\mathcal{P} = \langle P_1, P_2, \ldots, P_{|\mathcal{P}|} \rangle$, where each phrase set $P$ can contain at most $m$ surface forms, over a document collection $\mathcal{D}$. Consider that we have access to a standard dictionary and inverted index over words in the document collection. From them we can assemble the text regions for the evidences by looking up the word and its offsets within the documents. A naïve approach is: first retrieve those documents that contain the words from each of the $m$ phrases in $|\mathcal{P}|$ phrase sets. As a second step, we can intersect and pool those documents in which at least one phrase from each phrase set is present. Finally, with this pool of documents we can then scan each document for potential matches using the string matching algorithms from [8, 9, 11, 14, 20]. However, this simple approach is inefficient as we do not leverage common sub-phrases among the phrases in each phrase set for retrieval of posting lists. Furthermore, we also do not leverage any co-occurrence of words among phrase sets that can significantly bring down the cost of processing a HPQ.

## 3 DATA MODEL

There are three key challenges that need to be overcome in order to reduce the cost of processing a HPQ. The first challenge is to capture phrases as simpler combinations of n-grams and skip-grams. Capturing skip-grams is hard as their number grows polynomially with the sentence length. The second challenge is coming up with novel ways of maintaining sentence boundaries such that we can quickly identify the match of two phrases to lie within a distance of $k$ sentences. Third and finally, the data model must allow us to compute different ways to represent combinations of phrases. We can then design algorithms to optimize the order of processing such operators. Figure 2 summarizes the key elements of our data model. First, we must provide a data model that is capable of representing text regions within documents.

**Modeling Text Regions.** A *phrase* in our data model is defined as a contiguous sequence of words with their positional span as:

$$\mathbb{N} \times \mathbb{N} \times \Sigma^+. \tag{8}$$

In Equation 8, the Cartesian product of natural numbers $\mathbb{N} \times \mathbb{N}$ represents the *text region* $[i, j]$ of the phrase $\langle w_i, \ldots, w_j \rangle \in \Sigma^+$.

**Incorporating Sentence Boundaries.** A text document is explicitly structured using syntactical structures such as sentences, paragraphs, pages, sections, chapters etc. In our work, we consider *sentences* as the maximal unit for imposing structure on text. Sentence boundaries can be reliably detected using natural language processing (NLP) tools (e.g., Stanford's CoreNLP toolkit [18]). Our data model records the sentence information by encoding the sentence numbers as identifiers along with phrases. The data model with this augmentation is represented as:

$$\overbrace{\mathbb{N}}^{\text{sentence id}} \times \overbrace{\mathbb{N} \times \mathbb{N} \times \Sigma^+}^{\text{phrase}}. \tag{9}$$

For instance, with this approach, we can represent the bigram and the sentence information in Figure 2 as: $(1, 2, 3)$. Also, with this representation, we can easily compute relaxations of phrase matches to lie within a distance of $k$ sentences.

## 4 INDEXING UNITS AND INDEXES

Documents can be indexed by considering contiguous and non-contiguous combinations of words in our data model as indexing units. The key indexing units we consider are n-grams and skip-grams (shown in Figure 2). Our dictionaries and indexes are stored in HBase, a modern state-of-the-art distributed extensible record store. Our indexes in the HBase record store comprise of tables, where the records contain key-value pairs. In each key-value pair, the key of a record in the table encodes the indexing unit while the value stores the compressed payload for the posting list.

**N-Grams**. By considering the contiguous sequence of words of fixed length we can arrive at unigrams, bigrams, and trigrams as indexing units. These n-grams can immediately help us spot phrases by decomposing them as an overlap of two or more n-grams. For example, to retrieve documents for the phrase "*physics nobel prize*", we can decompose it to be an overlap of bigrams "*physics nobel*" and "*nobel prize*".

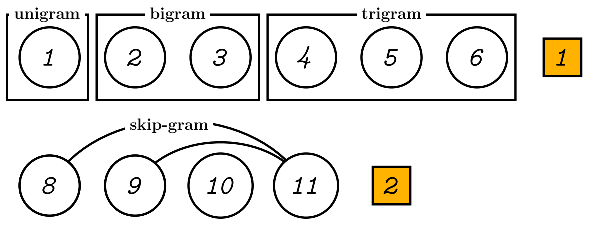

**Figure 2: Indexing units can be obtained by considering ordered contiguous sequences of words (n-grams) or ordered non-contiguous combinations of words (skip-grams). Additionally, to maintain context windows and semantically meaningful text regions we keep track of sentence boundaries. Circular nodes represent words and the numbers their positions. While the square nodes represent sentence boundaries and the numbers represent sentence identifiers.**

**Skip-Grams**. To spot a phrase or sequence of phrases it shall be helpful if we know, in advance, whether a pair of words co-occurs or not. Examples of skip-grams are also shown in Figure 2. However, recording all skip-grams contained within a sentence can lead to index blowup. To curtail the selection of skip-grams to only those which are highly discriminative, we leverage the syntactical structure of sentences within documents to record skip-grams. With this scheme we keep track of only those ordered co-occurrences of words that are within a sentence and within fixed separation of $\ell$ words. To instantiate our indexes we consider skip-grams where the word separation is at most ten words (i.e., $\ell = 10$). By limiting the separation between the skip-grams we also discard those combinations of words that may be part of different compound sentences and not semantically related to each other. Since, we keep track of infrequent skip-grams also, we can detect mentions of emerging entities as well. This however leads to large inverted index sizes. Since, establishing provenance for KG facts is a recall-oriented task, we consider this scheme for recording the ordered co-occurrence of words.

**Dictionaries.** For each n-gram and skip-gram we record their collection statistics in a dictionary. Each entry in the dictionary stores for each indexing unit $p$: the document frequency df($p$) – number of documents containing $p$ and the collection frequency cf($p$) – number of times $p$ was observed in the entire collection.

**Inverted Indexes.** Our inverted indexes store posting lists that consist of compressed lists corresponding to sorted document identifiers, and sorted lists for sentence identifiers, begin and end positions of the indexing elements. For the n-gram indexes we can omit payload corresponding to the end positions as they can be inferred by adding the n-gram length to the begin positions. The compression technique utilized is the patched frame of reference [6, 24].

## 5 QUERY PROCESSING AND OPERATORS

We now discuss the design of operators over text regions that we use to represent the combinatorial space of a hyper-phrase query.

**Basic Query Processing.** First and foremost, we discuss how to process a HPQ to obtain the resulting text regions. Assume that for a HPQ $\mathcal{P}$ we have obtained the posting lists corresponding to each of its constituent phrase sets $P$. To find the resulting text regions that contain the evidences for the HPQ we apply the binary variable-length match operator by processing two phrase sets at a time, from left to right, in $\mathcal{P}$. Algorithm 1 shows how to process a

**Algorithm 1:** Computing a variable-length gap match.

```
Input  : Posting lists $L_l$ and $L_r$ corresponding to the left and right operands
         of the variable length match operator and $k$ indicating the number
         of sentences the match may span.
Output : Posting List containing the resultant text regions containing the
         variable length match.
1  Function match($L_l$, $L_r$, $k$)
2      $R \leftarrow$ find all common documents for postings in $L_l$ & $L_r$
3      $L \leftarrow \varnothing$, $S \leftarrow \varnothing$
4      foreach doc-id $\in R$ do
5          $S \leftarrow$ join(payload for doc-id in $L_l$, payload for doc-id in $L_r k$)
6          $L \leftarrow L$.append(new⟨doc-id, $S$⟩)
7      return $L$
8  Function join($S_l$, $S_r$, $k$)
9      $S \leftarrow \varnothing$
10     if the last position span in $S_r$ lies before the first position span in $S_l$ then
11         return $S$
12     if first position span in $S_r$ is before the first position span in $S_l$ then
13         $S_r \leftarrow$ remove position spans from the front of $S_r$ until the first
           position spans in it is after the first position spans in $S_l$
14     for ($i \leftarrow 1; i \leq |S_l|; i{+}{+}$) do
15         for ($j \leftarrow 1; j \leq |S_r|; j{+}{+}$) do
16             if $|$sentence id for $S_r -$ sentence id for $S_l| \leq k$ then
17                 if ($S_l[i]$ is before $S_r[j]$) then
18                     $S \leftarrow S$.append([$S_l[i]$.begin, $S_r[j]$.end])
19     return $S$
```

variable-length match between two posting lists. With Algorithm 1 acting as a general framework for query processing, we note two avenues for optimization. First, we must minimize the time spent for retrieving the postings corresponding to each phrase set $P$ in HPQ $\mathcal{P}$. Which in turn implies presenting the variable-length match operator with a minimum number of documents to process (line 4 in Algorithm 1). Second, we must minimize the time spent for joining the positions corresponding to each common document (lines 8-19 of Algorithm 1).

**Naïve Optimization.** A naïve strategy to optimize the execution of the HPQ is to identify a common pool of documents for *all* the phrase sets $P$ in HPQ $\mathcal{P}$ before processing them for a variable-length match. Clearly, this strategy is expensive as explained in Section 2. An improvement for matching phrase sets was proposed by Agrawal et al. [7] for named entity extraction from text documents. Their method relies on first computing a set cover over the surface forms of named entities and then utilize Boolean operators over a word index (i.e., no positional information is used) to obtain a final superset of potentially matching documents. However, there is much room for improvement on executing a HPQ by exploiting the order of phrases and optimizing them in our proposed data model.

**Vertical Cover Operator (⊟).** Given our data model, we can leverage n-grams and skip-grams to reduce the time spent for retrieving postings for each phrase. We now describe a *vertical cover* operator that does this. Let $P$ be the phrase set and $\{p_1, p_2, \ldots, p_n\}$ the constituent phrases. We can induce a partitioning for each phrase set using n-grams and skip-grams. The partition that has minimal cost (see Section 6) is then used for retrieval of the posting lists. Using n-grams, we can decompose each phrase in the phrase set using unigrams, bigrams, and trigrams. For instance, for the phrase set $\{abc, bcd\}$ the common unigrams are $\{b, c\}$, while the common bigrams is $\{bc\}$. Using the common n-grams we can induce partitions over the phrase set. For example, the partition induced using unigrams is: $\{a, b, c, d\}$; using bigrams the partition is: $\{ab, bc, cd\}$; and using trigrams it is: $\{abc, bcd\}$.

**Algorithm 2:** Vertical cover operator using skip-grams.

```
Input  : HPQ $\mathcal{P} = \langle P_1, P_2, \ldots, P_n \rangle$.
Output : Set of skip-grams that cover the phrase sets $P$ in HPQ $\mathcal{P}$.
Function cover($\mathcal{P} \leftarrow \langle P_1, P_2, \ldots, P_n \rangle$)
    $S \leftarrow \varnothing$                              // Resulting skip-gram cover.
    /* Compute the set of skip-grams needed to retrieve postings for
       $\mathcal{P}$ by keeping all the phrases across phrase sets in one set. */
    skipGrams[] ← generateSkipGrams(put all phrases of $\mathcal{P}$ in an array)
    return skipGrams
Function generateSkipGrams($p[] \leftarrow \{p_1, p_2, \ldots, p_m\}$)
    $S \leftarrow \varnothing$                              // Resulting set of skip-grams.
    words[] ← $\varnothing$ // Holds the words for the phrase being processed.
    foreach $p \in p[]$ do
        words ← $p$.split()
        /* If the phrase is one-word only, keep it as is. It will be
           retrieved using the unigram index.                        */
        if (words.length = 1) then
            $S$.put($p[i]$)
            continue
        i ← 1
        for (j ← i + 1; j ≤ words.length; j ++) do
            $S$.put(⟨words[i], words[j]⟩)
    return $S$
```

**Algorithm 3:** Horizontal order operator.

```
Input  : Sets of n-grams that cover the phrase sets corresponding to the left
         and right operand of the variable-length phrase match.
Output : Set of selective skip-grams between phrase sets that are indicative of
         the number of positions needed to be merged for the join.
Function order($S_l$, $S_r$)
    $S \leftarrow \varnothing$                         // Resulting skip-gram join cover.
    words$_l$, words$_r$ ← $\varnothing$
    minCostSkipGram, skipGram ← $\varnothing$
    double minCost, cost ← $-\infty$
    foreach ($p_l \in S_l$) do
        words$_l$ ← $p_l$.split()
        minCost, cost ← 0
        foreach ($p_r \in S_r$) do
            words$_r$ ← $p_r$.split()
            for (i ← 1; i ≤ words$_l$.length; i++) do
                for (j ← 1; j ≤ words$_r$.length; j++) do
                    skipGram ← ⟨words$_l$[i], words$_r$[j]⟩
                    cost ← cost(skipGram)
                    if (minCost ≡ $-\infty$) ∨ (minCost > cost) then
                        minCost ← cost
                        minCostSkipGram ← skipGram
            /* Add the min. cost skip gram.                            */
            $S$.add(minCostSkipGram)
    return $S$
```

Using skip-grams, we can induce the partitions over phrase sets by computing ordered co-occurrences with respect to an anchor word. To compute skip-grams, we fix the first word of the phrase as an anchor word and then derive all the skip-grams with respect to it. For instance, for the phrase set $\{abc, adc\}$ the skip-grams induce the partitioning: $\{\langle a, b\rangle, \langle a, c\rangle, \langle a, d\rangle\}$. Algorithm 2 shows how to compute skip-grams that can then be used to retrieve posting lists.

**Horizontal Order Operator ($\boxminus$).** We now discuss how to minimize the time for performing the joins (lines 8-19 of Algorithm 1). When computing the variable-length match for a HPQ we shall like to process those combinations that are highly selective, i.e., possess the least number of positions first. In our data model, we can compute the cost of these joins by using skip-grams between words from phrases belonging to different phrase sets. Concretely, we select that skip-gram for phrases belonging to different phrase sets that contains the least number of positions. By summing up all the cardinalities indicated by the set of skip-grams, output by Algorithm 3, we can determine the cost of joining two phrase sets. Using the vertical cover and horizontal order operators we can now model the execution of hyper-phrase queries using the n-gram and skip-gram indexes.

**Operator Properties.** We note the following operator properties that are helpful in optimizing their order. The vertical cover operator is both associative ($(a \boxbar b) \boxbar c = a \boxbar (b \boxbar c)$) and commutative ($a \boxbar b = b \boxbar a$). However, the horizontal order operator is only associative ($(a \boxminus b) \boxminus c = a \boxminus (b \boxminus c)$). Furthermore, the vertical cover operator is only left distributive over the horizontal order operator ($(a \boxbar b) \boxminus (a \boxbar c) = a \boxbar (b \boxminus c)$).

## 6 QUERY OPTIMIZATION

We now discuss the strategy to select the optimal operator sequence to process a hyper-phrase query.

**Cost Model.** Our cost model depends on two factors. First, we account for the number of postings to be retrieved from the inverted indexes for a given indexing unit based on document frequency – $C_{\text{join}}$. Second, we account for the number of positions in each posting list based on collection frequency – $C_{\text{merge}}$. The first cost $C_{\text{join}}$ gives us an estimate on the cardinality of the document sets we must join for the query operators. The second cost $C_{\text{merge}}$ gives us an estimate of the number of positions that we must merge to compute the resulting posting for the query operators.

**Cost of Vertical Cover Operator ($\boxbar$).** The cardinality of the result of the vertical cover operator is directly proportional to the union of the number of documents associated with the left and right operand. Formally, it can be expressed in terms of document frequency (df(●)) as follows: $|a \boxbar b| \propto (\text{df}(a) + \text{df}(b) - \text{df}(a \cap b))$.

The cost associated with the vertical cover operator can be determined using the number of common n-grams (skip-grams) (see cover(●) in Algorithm 2) that we can uncover from each phrase set of the HPQ. In case of no overlap, cost degenerates to querying the n-gram (skip-gram) inverted indexes for each n-gram (skip-gram) decomposition of the phrases in each phrase set. We can put this formally as:

$$\text{cost}(a \boxbar b) \propto \sum_{x \in \text{cover}(a,b)} \Big( C_{\text{join}} \cdot \text{df}(x) + C_{\text{merge}} \cdot \text{cf}(x) \Big).$$

**Cost of Horizontal Order Operator ($\boxminus$).** The cardinality of the result of the horizontal order operator is proportional to the intersection of the number of documents associated with the left and right operand. Formally, this can be expressed in the number of documents (df(●)) as follows: $|a \boxminus b| \propto (\text{df}(a) + \text{df}(b) - \text{df}(a \cup b))$.

The cost associated with the horizontal order operator depends on the number of skip-gram combinations generated across the phrase sets. To obtain the ordering using skip-grams for the horizontal order operator, we need to generate skip-grams between each two consecutive phrase sets in the HPQ (see order(●) in Algorithm 3). We can model the cost in two ways. First, if it is required that we optimize both for the number of documents and positions, we can write the cost as:

$$\text{cost}(a \boxminus b) \propto \mathop{\arg\min}_{x \in \text{order}(\langle a,b \rangle)} \Big( C_{\text{join}} \cdot \text{df}(x) + C_{\text{merge}} \cdot \text{cf}(x) \Big).$$

However, if the number of documents is equal in both operands, then we are interested in minimizing the number of positions to be merged:

$$\text{cost}(a \boxminus b) \propto \underset{x \in \text{order}(\langle a,b \rangle)}{\arg\min} \left( \frac{C_{\text{merge}} \cdot \text{cf}(x)}{C_{\text{join}} \cdot \text{df}(x)} \right). \tag{10}$$

**Cost Comparison between $\boxbar$ and $\boxminus$.** In general, computing the intersection over large sets of documents is more expensive than computing the union. The common document identifiers can be identified via three strategies. The first strategy uses binary search for looking up an identifier from the first list in the second list of ordered document identifiers. The second strategy is to use galloping (interpolation) search in order to speed up the lookups in ordered lists. The third and final strategy is to use hash-set-based intersections to reduce the time for intersection at the cost of performance degradation due to collisions. The first two strategies benefit from data locality however, on modern hardware architectures, all perform comparably well [12]. Our implementation uses the third strategy. Furthermore, the horizontal order operator leads to a Cartesian product of sets if distributed over the vertical cover operator. That is, $(a \boxbar b) \boxminus (c \boxbar d) = (a \boxminus c) \boxbar (a \boxminus d) \boxbar (b \boxminus c) \boxbar (b \boxminus d)$. Given these two characteristics of the horizontal order operator, it is more expensive to compute than the vertical cover operator. Formally, $\text{cost}(\boxbar) \ll \text{cost}(\boxminus)$.

**Optimization.** Let $\mathcal{P} = \langle P_1, P_2, \ldots, P_n \rangle$ be a hyper-phrase query. Where each phrase set $P_i = \{p_1^i, p_2^i, \ldots, p_m^i\}$ has at most $m$ phrases. Using the aforementioned query operators we can specify the following operator sequence for executing the query $\mathcal{P}$:

$$(p_1^1 \boxbar p_2^1 \ldots p_m^1) \boxminus (p_1^2 \boxbar p_2^2 \ldots p_m^2) \boxminus \ldots \boxminus (p_1^n \boxbar p_2^n \ldots p_m^n). \tag{11}$$

We can rephrase the above formulation as follows:

$$\underset{j=1}{\overset{n}{\boxminus}} P_j = \underset{j=1}{\overset{n}{\boxminus}} \underset{i=1}{\overset{m}{\boxbar}} p_i^j \tag{12}$$

where, $P$ represents the result of the vertical cover operator. At a high level, it may seem trivial to execute the vertical cover operators ($\boxbar$) first to obtain the union of the documents representing the phrase sets ($P_j$) and subsequently the horizontal order operators ($\boxminus$) to obtain the result of sequences of one phrase set following the other ($P_{j-1} \boxminus P_j$). However, naïvely executing this query plan may be expensive. This is because, we may end up intersecting two potentially large posting lists. This can be avoided if we chose to perform a more selective operand (i.e., operand with fewer postings) with an operand with a large posting list in order to eliminate those documents that will not end up in the final result.

**Optimizing Join-Order Sequence.** To optimize the sequence of operators in Equation 12, we first need to uncover the **optimal substructure property** underlying the hyper-phrase queries. It is important to note that a subsequence of phrase sets in a HPQ $\mathcal{P} = \langle P_1, P_2, \ldots, P_n \rangle$ is yet another HPQ. To optimize the original HPQ, we must first identify the optimal sequence of performing the horizontal order operators for sub-hyper-phrase queries that it contains. This problem is similar to that of identifying the optimal order of multiplying a sequence of compatible matrices [10] and optimizing the join order for SQL queries in relational databases [22]. A naïve method of computing such an optimal solution is lower

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| $P_1$ | 100 | 1400 | 2125 | 3400 | 3790 |
| $P_2$ | × | 750 | 1750 | 3275 | 4465 |
| $P_3$ | × | × | 175 | 1975 | 3415 |
| $P_4$ | × | × | × | 725 | 3975 |
| $P_5$ | × | × | × | × | 390 |

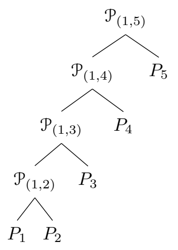

**Figure 3: An illustrative example of generating the optimal execution plan. For the example, consider the cost constant ratio to be $C_{\text{join}}/C_{\text{merge}} = 10$ with following cardinalities of the $\boxminus$ operator $|P_1 \boxminus P_2| = 50$, $|P_2 \boxminus P_3| = 75$, $|P_3 \boxminus P_4| = 100$, and $|P_4 \boxminus P_5| = 300$. Assume that each resulting text regions has only 50 positional spans.**

bounded by the *Catalan Numbers* – $\Omega(4^n/3^{3/2})$ [10]. Let a sub-hyper-phrase query be denoted by $\mathcal{P}_{(i,j)}$ where the subscripts denote the subsequence (without omissions) of phrase sets from the original hyper-phrase query $\mathcal{P}$. The optimal join order sequence can then be stated as:

$$\text{opt}(\mathcal{P}_{(i,j)}) = \begin{cases} \text{cost}(\boxbar_{l=1}^{m} p_l^i), & \text{if } i = j \\ \min\limits_{i \le k < j} \Big[ \text{opt}(\mathcal{P}_{(i,k)}) + \text{opt}(\mathcal{P}_{(k+1,j)}) + \\ \qquad \text{cost}\big(\mathcal{P}_{(i,k)} \boxminus \mathcal{P}_{(k+1,j)}\big) \Big], & \text{if } i < j. \end{cases}$$

The equation above decomposes the optimization of a HPQ $\mathcal{P}_{(1,n)}$ into smaller hyper-phrase queries. The base case of the inductive hypothesis corresponds to computing the cost of the vertical cover operator over each individual phrase set. The induction step builds up the dynamic programming table by first computing the join between a hyper-phrase query with only two phrase sets. Thereafter, we seek to intersect the solutions to those hyper-phrase queries that consist of the least number of documents. Figure 3 shows an example of computing an optimal join order sequence.

## 7 EVALUATION

We now discuss the evaluation setup of our experiments.

**Document Collections and their Indexes.** We build our proposed indexes over four large document collections. The first document collection consists of news articles published in the New York Times (NYT) and is publicly available [3]. The NYT archive consists of approximately two million documents published between 1997 and 2007. The second document collection we utilize is Wikipedia [4]. Wikipedia comprises of around five million documents and is also publicly available. The third document collection consists of news articles published by seven major newswire organizations including the Washington Post, the Associated Press, and the Xinhua News Agency during the 1995-2010 reporting period [1]. This news archive is publicly available as the fifth edition of the English Gigaword. It consists of around ten million documents. The fourth and final document collection is the largest in our evaluation setup. It comprises of a set of news articles obtained by crawling the links available on the real-world event repository GDelt [2]. The GDelt document collection amounts to a total of approximately fourteen million documents. In total, the four document collections amount to more than thirty million documents. A summary of the document collection statistics is given in Table 1.

For each document collection, we created indexes based on the data model discussed in Section 4. For n-grams, we instantiated indexes for unigrams, bigrams, and trigrams. For skip-grams, we instantiated indexes that record skip-grams with word separation of up to ten words (i.e., $\ell = 10$). The index build times are shown in Table 2. Corresponding to each collection the dictionary and index sizes in GB are reported in Table 3. We also show the sizes of the word dictionary and inverted index for each of the collection in Table 3. It can be observed by storing higher length n-grams we blow up our indexes by a factor of at most 7.91×. While, the blow up for the skip-gram index is at most 10.04× compared to a standard word (unigram) index. Despite the large index sizes, we observe that by lowering the word separation $\ell$ between skip-grams we can reduce the skip-gram index sizes.

**Hyper-Phrase Queries.** In order to test the efficiency of our proposed approach we utilize the Wikidata KG [5] to construct a testbed of hyper-phrase queries. The translation of facts in the KG to hyper-phrase queries is done via the following scheme: $\mathcal{P} \equiv \langle \text{S}, \text{P}, \text{O} \rangle$. We evaluate our proposed method and baselines against two kinds of tasks: KG fact and KG subgraph spotting task.

**Queries for KG Fact Spotting (KG-F) Task.** For this task, each test instance consists of a HPQ corresponding to a single KG fact with multiple object arguments. Each query consists of three phrase sets where each phrase is an alias for the subject, predicate, or object. To instantiate the instances for this task, we pursued those entities where multiple objects for the same predicate could be associated. Concretely, we constructed instances for categories and predicates in Table 4. For each instance in this task we record the time to retrieve the text regions as evidences for each HPQ. An example of an instance in the KG fact spotting task is shown in Figure 1.

**Queries for KG Subgraph Spotting (KG-S) Task.** Journalists and scholars in humanities are seldom interested in retrieving evidences for a single KG fact. It is often required that one can query for relationships between multiple entities in a single query. In order to simulate hyper-phrase queries with more than three phrase sets we consider KG subgraphs. In particular, we restrict ourselves to subgraphs with a star topology. To construct queries for the task of KG-S, we take each fact concerning an entity (subject) as a constituent HPQ. Thus, each instance in KG-S task is a list of hyper-phrase queries, where each HPQ represents a fact concerning a common entity. For each instance in this task, we retrieve the text regions as evidences for each HPQ in the list and record the total time to execute all of the constituent HPQ.

To materialize the instances for the KG-S task we focus on prominent named entities. The prominence is chosen by restricting ourselves to famous scientists, artists, and athletes. To select these named entities we looked at the prestigious awards won by scientists (e.g., the Nobel Prize), artists (e.g., the Grammy), and athletes (e.g., a medal at the Summer Olympics). The concrete Wikidata object identifiers corresponding to these awards is shown in Table 5. By obtaining a set of entities where the award occurs as an object, we then focused on common and selective key predicates to further narrow down the facts concerning these entities. The key predicates that we utilized were: *P19* (PLACE OF BIRTH), *P10* (OCCUPATION), *P166* (AWARDS RECEIVED), *P69* (EDUCATED AT), *P800* (NOTABLE WORK), *P22* (FATHER), *P26* (SPOUSE), *P361* (PART OF), *P39* (POSITION HELD), and *P102* (MEMBER OF POLITICAL PARTY). The

**Table 1: Document collection statistics.**

| COLLECTION | $n_{\text{documents}}$ | $n_{\text{words}}$ | $n_{\text{sentences}}$ |
|---|---|---|---|
| NYT | 1,855,623 | 1,058,949,098 | 54,024,146 |
| WIKIPEDIA | 5,327,767 | 2,807,776,276 | 192,925,710 |
| GIGAWORD | 9,870,655 | 3,988,683,648 | 181,386,746 |
| GDELT | 14,320,457 | 6,371,451,092 | 297,861,511 |

**Table 2: Inverted Index build times in minutes.**

| TYPE | NYT | WIKIPEDIA | GIGAWORD | GDELT |
|---|---|---|---|---|
| UNIGRAM | 5.83 | 22.22 | 31.10 | 50.37 |
| BIGRAM | 16.27 | 46.72 | 68.15 | 99.05 |
| TRIGRAM | 39.98 | 104.77 | 94.87 | 147.68 |
| SKIP-GRAM | 79.62 | 222.52 | 191.43 | 259.10 |

**Table 3: Dictionary and index sizes in Gigabytes.**

| | NYT | WIKIPEDIA | GIGAWORD | GDELT |
|---|---|---|---|---|
| COLLECTION SIZE | 3.00 | 21.89 | 9.10 | 77.44 |
| **INDEX TYPE** | **NYT** | **WIKIPEDIA** | **GIGAWORD** | **GDELT** |
| WORD DICTIONARY | 0.04 | 0.30 | 0.10 | 0.14 |
| N-GRAM DICTIONARIES | 4.54 | 19.80 | 10.50 | 19.04 |
| SKIP-GRAM DICTIONARY | 14.40 | 25.70 | 21.30 | 29.30 |
| WORD INDEX | 5.80 | 18.00 | 22.30 | 35.90 |
| N-GRAM INDEXES | 45.90 | 126.30 | 154.40 | 234.80 |
| SKIP-GRAM INDEX | 56.10 | 180.80 | 203.60 | 289.00 |

**Table 4: Statistics regarding the testbed for the KG-F task.**

| CATEGORY | PREDICATES | $n_{\text{queries}}$ | $\mu_{\text{words}}$ |
|---|---|---|---|
| WRITERS | AWARD RECEIVED,NOTABLE WORK | 694 | 57.95 |
| MEDICINE LAUREATES | AWARD RECEIVED,EMPLOYER | 410 | 55.02 |
| PHYSICS LAUREATES | AWARD RECEIVED,EMPLOYER | 406 | 58.48 |
| CHEMISTRY LAUREATES | AWARD RECEIVED,EMPLOYER | 348 | 56.04 |
| MOVIES | CAST MEMBER,FILMING LOCATION | 114 | 75.51 |
| ALL US ELECTIONS | CANDIDATE | 1 | 1081.00 |
| ALL WORLD WAR I BATTLES | LOCATION | 1 | 1669.00 |
| ALL WORLD WAR II BATTLES | LOCATION | 1 | 2563.00 |
| ALL SUMMER OLYMPICS | LOCATION | 1 | 717.00 |
| ALL WINTER OLYMPICS | LOCATION | 1 | 407.00 |

**Table 5: Statistics regarding the testbed for the KG-S task.**

| CATEGORY | KG OBJECT ID | $n_{\text{queries}}$ | $\mu_{\text{words}}$ |
|---|---|---|---|
| ACTRESSES | *Q103618* | 1,434 | 32.72 |
| ACTORS | *Q103916* | 1,547 | 35.00 |
| SINGERS | *Q41254* | 327 | 33.86 |
| WRITERS | *Q37922* | 2,316 | 33.69 |
| US PRESIDENTS | *Q11696* | 367 | 47.22 |
| PHYSICISTS | *Q38104* | 2,056 | 34.12 |
| CHEMISTS | *Q44585* | 1,788 | 33.99 |
| MATHEMATICIANS | *Q28835* | 189 | 33.19 |
| ECONOMISTS | *Q47170* | 274 | 37.57 |
| PACIFISTS | *Q35637* | 1,945 | 36.28 |
| OLYMPIANS | *Q15243387* *Q15889641* *Q15889643* | 144 | 31.79 |

number of facts thus created are reported in Table 5. An illustrative example of an instance of the KG subgraph spotting task is shown in Figure 1. The testbeds for KG-F and KG-S tasks are publicly available at: `http://resources.mpi-inf.mpg.de/dhgupta/data/hpq/`.

**Implementation and Hardware Details.** The entire indexing infrastructure has been built from scratch in Java with indexes stored in a cluster of machines running Cloudera CDH 5.90 version of Hadoop and HBase. Our Hadoop cluster consists of twenty machines in which all machines have up to 24 core Intel Xeon CPUs at 3.50 GHz, up to 128 GB of primary memory, and up to eight 4 TB HDDs. With the Hadoop cluster acting as our storage backend, we evaluate our queries on a high-memory compute node equipped with 1.48 TB of primary memory and a 96 core Intel Xeon CPU with processing speed of 2.66 GHz.

**Baselines and Systems.** We evaluate three baselines against our proposed approach. We first establish a lower bound on how much time a HPQ should take to answer by scanning the entire

Table 6: Baselines and Systems.

| SYSTEM | N-GRAM DICTIONARY | N-GRAM INDEX | SKIP-GRAM DICTIONARY | SKIP-GRAM INDEX |
|---|---|---|---|---|
| $B_{\text{SCAN}}$ | × | × | × | × |
| $B_{\text{BIGRAM}}$ | × | ● | × | × |
| $B_{\text{NGRAM}}$ | × | ● | × | × |
| $A_{\boxbar}$ | ● | ● | × | × |
| $A_{\boxbar\boxminus}$ | ● | ● | ● | ● |

Table 7: Results for Baseline $B_{\text{SCAN}}$ (secs).

| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
|---|---|---|---|---|
| $B_{\text{SCAN}}$ | 111.00 | 322.00 | 396.00 | 604.00 |

document collection on our Hadoop cluster in an embarrassingly parallel manner $B_{\text{SCAN}}$. As a second naïve baseline, $B_{\text{BIGRAM}}$, we retrieve the results corresponding to each individual phrase set in the HPQ using unigram and bigram decomposition only. We use bigrams to construct the result set for phrases as we do not index stopwords in the word index. Subsequent to this we compute the result set only for those document identifiers that are present in all the phrase sets of the HPQ. As a third baseline, $B_{\text{NGRAM}}$, we consider the longest possible n-gram decomposition possible with our indexes. The result is computed in a similar manner as in $B_{\text{BIGRAM}}$. The $B_{\text{BIGRAM}}$ and $B_{\text{NGRAM}}$ baselines, simulate traditional inverted indexes (e.g., Lucene and Elasticsearch) with advantages (i.e., additionally encoding sentence identifiers). As our first system, $A_{\boxbar}$, we test our approach that searches for hyper-phrase queries by only leveraging the optimized vertical cover operator across phrase sets. As our second system, $A_{\boxbar\boxminus}$, we execute each HPQ using the optimized vertical cover and horizontal order operators.

**Parameters and Setup.** For both the KG-F and KG-S task, we sample 100 queries to execute for the two baselines and our systems. For the remaining baseline $B_{\text{SCAN}}$ we measure the time required to scan each document in the entire document collection once to establish a lower bound. We evaluate the sampled queries for three rounds with cold cache settings. To simulate the cold cache setting: we shuffle the order of queries in between rounds. We only show cold cache run times as they are similar to the warm cache setting results. Furthermore, for computing the optimal plan of execution we set the constants ratio ($C_{\text{join}}/C_{\text{merge}}$) to 1 and we use the cost for horizontal-order operator in accordance with to Equation 10. The rationale for choosing this ratio was to consider the cost for merging positions same as documents as we have applied the naïve query optimization to all baselines and systems. We additionally vary the match between phrase sets to lie within a distance of $k \in \{0, 2, 5\}$ sentences of the document containing the evidence.

**Results.** A summary of the dictionaries and indexes used by the baselines and systems is shown in Table 6. We have further computed the statistical significance of the results using the Student's paired t-test at significance level $\alpha = 0.05$. The systems that produce statistically significant results over the $B_{\text{BIGRAM}}$ baseline are marked with △ and over the $B_{\text{NGRAM}}$ are marked with ▲. The results for the first baseline $B_{\text{SCAN}}$ are shown in Table 7. As expected the time to scan the document collection directly depends on the collection size. The NYT being the smallest takes the least time while GDelt takes the most time on our Hadoop cluster to scan.

**KG-F Task Results.** We first discuss the efficiency results of the KG fact spotting task. In this task, we are required to retrieve text regions for a sequence of phrase sets that correspond to a KG fact. The results for the baselines and systems are displayed in Table 8. When restricting ourselves to matching phrase sets within

Table 8: Results for for KG-F Task (secs).

| RUNTIME RESULTS FOR KG-F TASK (SECS) FOR $k = 0$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 7.77 ± 12.91 | 24.41 ± 29.43 | 41.24 ± 62.94 | 67.34 ± 164.75 |
| $B_{\text{NGRAM}}$ | 1.80 ± 2.82 | 7.59 ± 6.37 | 7.89 ± 7.16 | 12.47 ± 17.98 |
| $A_{\boxbar}$ | △ 1.92 ± 3.63 | △ 7.21 ± 5.63 | △ 7.74 ± 7.30 | △ 11.57 ± 16.36 |
| $A_{\boxbar\boxminus}$ | △ ▲ 1.41 ± 2.42 | △ ▲ 6.04 ± 5.57 | △ ▲ 6.45 ± 6.50 | △ ▲ 9.78 ± 15.35 |

| RUNTIME RESULTS FOR KG-F TASK (SECS) FOR $k = 2$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 8.61 ± 18.71 | 21.8 ± 28.25 | 34.77 ± 48.91 | 42.19 ± 68.97 |
| $B_{\text{NGRAM}}$ | 1.91 ± 3.33 | 7.11 ± 5.87 | 8.46 ± 8.66 | 10.50 ± 11.08 |
| $A_{\boxbar}$ | △ 1.82 ± 2.87 | △ 6.97 ± 6.06 | △ 8.21 ± 7.63 | △ 9.90 ± 10.31 |
| $A_{\boxbar\boxminus}$ | △ ▲ 1.52 ± 2.82 | △ ▲ 5.86 ± 5.44 | △ ▲ 6.92 ± 6.81 | △ ▲ 7.65 ± 8.97 |

| RUNTIME RESULTS FOR KG-F TASK (SECS) FOR $k = 5$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 10.45 ± 30.12 | 35.18 ± 120.41 | 39.93 ± 59.29 | 71.81 ± 233.71 |
| $B_{\text{NGRAM}}$ | 2.08 ± 4.64 | 10.02 ± 32.68 | 8.41 ± 7.42 | 12.92 ± 22.46 |
| $A_{\boxbar}$ | △ 2.56 ± 6.06 | △ 9.47 ± 30.57 | △ 10.17 ± 11.39 | △ 13.71 ± 25.07 |
| $A_{\boxbar\boxminus}$ | △ ▲ 1.64 ± 4.35 | △ ▲ 8.26 ± 28.87 | △ ▲ 7.07 ± 7.13 | △ ▲ 11.26 ± 23.68 |

Table 9: Results for for KG-S Task (secs).

| RUNTIME RESULTS FOR KG-S TASK (SECS) FOR $k = 0$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 7.50 ± 8.67 | 35.77 ± 53.78 | 37.66 ± 76.66 | 49.70 ± 66.84 |
| $B_{\text{NGRAM}}$ | 4.07 ± 4.86 | 18.04 ± 9.90 | 15.61 ± 15.92 | 21.18 ± 17.35 |
| $A_{\boxbar}$ | △ 3.96 ± 4.17 | △ 17.26 ± 10.35 | △ 14.71 ± 11.81 | △ 20.52 ± 17.29 |
| $A_{\boxbar\boxminus}$ | △ 2.91 ± 4.22 | △ ▲ 10.53 ± 7.92 | △ 12.05 ± 10.96 | △ 17.45 ± 14.83 |

| RUNTIME RESULTS FOR KG-S TASK (SECS) FOR $k = 2$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 6.20 ± 7.40 | 32.76 ± 31.72 | 46.56 ± 107.71 | 57.10 ± 67.78 |
| $B_{\text{NGRAM}}$ | 3.54 ± 4.11 | 17.79 ± 11.76 | 15.20 ± 15.74 | 21.38 ± 17.27 |
| $A_{\boxbar}$ | △ 3.32 ± 4.28 | △ 16.97 ± 11.11 | △ 14.39 ± 12.80 | △ 20.00 ± 16.27 |
| $A_{\boxbar\boxminus}$ | △ 2.98 ± 5.58 | △ ▲ 10.20 ± 7.74 | △ 12.17 ± 13.71 | △ ▲ 15.38 ± 13.60 |

| RUNTIME RESULTS FOR KG-S TASK (SECS) FOR $k = 5$. | | | | |
|---|---|---|---|---|
| SYSTEM | NYT | WIKIPEDIA | GIGAWORD | GDELT |
| $B_{\text{BIGRAM}}$ | 9.33 ± 10.95 | 31.19 ± 34.11 | 38.09 ± 48.21 | 63.94 ± 72.30 |
| $B_{\text{NGRAM}}$ | 4.47 ± 4.29 | 17.36 ± 12.28 | 15.12 ± 12.11 | 24.47 ± 18.47 |
| $A_{\boxbar}$ | △ 4.22 ± 4.99 | △ 16.86 ± 11.04 | △ 14.76 ± 12.43 | △ 23.36 ± 18.83 |
| $A_{\boxbar\boxminus}$ | △ ▲ 3.06 ± 4.06 | △ ▲ 9.91 ± 7.81 | △ ▲ 11.66 ± 9.89 | △ ▲ 18.47 ± 16.06 |

a sentence we notice a speed up of at least 8.97× using bigrams $B_{\text{BIGRAM}}$ over the simple $B_{\text{SCAN}}$ baseline. Further using trigrams to spot phrases for matching a HPQ brings about a speed up of at least 42.42× over $B_{\text{SCAN}}$. Using our proposed optimization we can achieve a speed up of at least 1.22× over $B_{\text{NGRAM}}$, 4.04× over $B_{\text{BIGRAM}}$, and 53.31× over $B_{\text{SCAN}}$. As we increase the sentence relaxation $k$ size for matching a HPQ we see a speed up of at least 1.15× over $B_{\text{NGRAM}}$ and 3.72× over $B_{\text{BIGRAM}}$. It is also important to observe that with increasing collection sizes the difference between end-to-end runtime results between the best baseline $B_{\text{NGRAM}}$ and our system $A_{\boxbar\boxminus}$ increases significantly.

**KG-S Task Results.** We now discuss the efficiency results of the KG subgraph spotting task. In this task, we are required to retrieve text regions for HPQs that correspond to multiple facts concerning an entity. The results for the baselines and systems are displayed in Table 9. For the KG subgraph spotting task, when restricting ourselves to matching each constituent fact of a subgraph to within a sentence, the baseline $B_{\text{BIGRAM}}$ achieves a speed up of at least 9.00× over $B_{\text{SCAN}}$. While, the baseline $B_{\text{NGRAM}}$ achieves a speed up of at least 17.85× over $B_{SCAN}$. The improvements offered by our proposed optimizations add up to reflect a speed up of at least 1.21× and at most 1.71× over $B_{NGRAM}$. As we increase the sentence relaxation $k$ size for matching phrase sets in a KG subgraph we observe speed ups of at least 1.19× over $B_{NGRAM}$, 2.08× over $B_{BIGRAM}$, and 31.57× over $B_{SCAN}$. Just like the KG fact spotting task, we observe that with increasing collection size the benefit offered by proposed optimization is significant over the baselines $B_{BIGRAM}$ and $B_{NGRAM}$.

**Summary.** At the task of spotting evidences for KG facts, our proposed optimizations show an improvement of at least 1.15× and at most 1.37× over $B_{NGRAM}$ across different levels of sentence separations. At the task of spotting evidences for KG subgraphs, our proposed optimizations have shown an improvement of at least 1.19× and at most a speed up of 1.75× over $B_{NGRAM}$. We also observe that as we move across increasing collection sizes the benefit of optimization also becomes apparent. This speed up however, comes at a cost of maintaining n-gram indexes and skip-gram indexes that are a factor of at most 7.91× and 10.04× larger than keeping only a word index. Despite this, we note that depending upon the application domain, selective choices regarding what kind of skip-grams to index and n-grams can further bring down the storage cost and at the same time offer speed ups based on our optimization for executing hyper-phrase queries.

## 8 RELATED WORK

We now discuss prior studies related to our problem setting.

Variable length pattern matching is an allied area with respect to our problem setting. Prior works [8, 20] have studied how in-memory data structures can help in the design of efficient matching algorithms. For instance, [20] considered matching-lookup table while [8] considered a wavelet tree as an in-memory index to speed up the matching process. Our work in contrast, leverages large-scale inverted indexes that are part of modern IR systems to efficiently execute a more difficult problem. A straight-forward approach to spotting evidences for KG facts is to index document collections annotated with named entities linked to KGs. However, using such an approach we can not spot facts for out-of-KG entities or their emerging relations. A recent work on spotting KG facts uses regular expression based operators at word-level [15, 16]. However, their approach disregards any optimization for efficient execution of hyper-phrase queries. [13, 19] propose a system that retrieves witness documents given a KG fact as a query. However, a limitation of their system is that documents need to be processed apriori and linked to KG facts for their retrieval. Put another way, out-of-KG facts or entities can not be processed with their system. Our approach solves this issue by relying on a data model that can represent n-grams, skip-grams, and sentence boundaries. Relying on our data model, we can then retrieve text regions as evidences for KG facts. [17] investigate how to model query execution plans with respect to recall of relevant documents and the query's execution time. Their approach contrasts between two models: inverted index based approach versus scanning the entire document collection. [7] describes an algorithm that identifies a relevant set of documents for named entities by finding a token-set-cover for various surface forms of the named entity and computing a join of the retrieved documents. [21, 23] describe approaches to query phrases using combinations of inverted, phrase, nextword, and direct indexes. Our work in contrast explores ways to compute an optimal plan of hyper-phrase query execution using dictionaries and indexes over n-grams and skip-grams.

## 9 CONCLUSION

We have shown how to speed up the processing of verbose hyper-phrase queries that help in establishing provenance for knowledge graph facts and subgraphs. Additionally, our approach shall find applications in knowledge acquisition for relationships and facts regarding out-of-knowledge-graph entities. Our solution consists of a data model for text that indexes n-grams and skip-grams along with their sentence identifiers. Furthermore, we presented operators to express the complete combinatorial space for optimizing hyper-phrase queries. We then described an approach based on dynamic programming to generate an optimal query plan using the proposed vertical cover and horizontal order query operators. We showed the efficiency of our system in spotting evidences for knowledge graph facts and subgraphs in document collections amounting to more than thirty million documents.

## REFERENCES

[1] English Gigaword Fifth Edition. `https://catalog.ldc.upenn.edu/LDC2011T07`

[2] The GDELT Project. `https://www.gdeltproject.org`

[3] The New York Times Annotated Corpus. `https://catalog.ldc.upenn.edu/LDC2008T19`

[4] Wikipedia: The Free Encyclopedia. `https://www.wikipedia.org`

[5] Wikidata: The Free Knowledge Base. `https://www.wikidata.org`

[6] D. Lemire. JavaFastPFOR: A Simple Integer Compression Library in Java. `https://github.com/lemire/JavaFastPFOR`

[7] S. Agrawal et al. Scalable Ad-hoc Entity Extraction from Text Collections. *PVLDB* 1, 1 (2008), 945–957.

[8] J. Bader et al. Practical Variable Length Gap Pattern Matching. In *SEA 2016*. 1–16.

[9] P. Bille et al. String Matching with Variable Length Gaps. *Theor. Comput. Sci.* 443 (2012), 25–34.

[10] T. H. Cormen et al. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company.

[11] M. Crochemore et al. Approximate String Matching with Gaps. *Nord. J. Comput.* 9, 1 (2002), 54–65.

[12] J. S. Culpepper and A. Moffat. Efficient Set Intersection for Inverted Indexing. *ACM Trans. Inf. Syst.* 29, 1 (2010), 1:1–1:25.

[13] S. Elbassuoni et al. ROXXI: Reviving witness dOcuments to eXplore eXtracted Information. *PVLDB* 3, 2 (2010), 1589–1592.

[14] K. Fredriksson and S. Grabowski. Efficient Algorithms for Pattern Matching with General Gaps, Character Classes, and Transposition Invariance. *Inf. Retr.* 11, 4 (2008), 335–357.

[15] D. Gupta and K. Berberich. GYANI: An Indexing Infrastructure for Knowledge-Centric Tasks. In *CIKM 2018*. 487–496.

[16] D. Gupta and K. Berberich. Structured Search in Annotated Document Collections. In *WSDM 2019*. 794–797.

[17] P. G. Ipeirotis et al. To Search or to Crawl?: Towards a Query Optimizer for Text-Centric Tasks. In *SIGMOD 2006*. 265–276.

[18] C. D. Manning et al. The Stanford CoreNLP Natural Language Processing Toolkit. In *ACL 2014*. 55–60.

[19] S. Metzger et al.. S3K: Seeking Statement-Supporting Top-k Witnesses. In *CIKM 2011*. 37–46.

[20] Fan Min et al. Pattern Matching with Independent Wildcard Gaps. In *DASC 2009*. 194–199.

[21] K. Panev and K. Berberich. 2014. Phrase Queries with Inverted + Direct Indexes. In *WISE 2014*. 156–169.

[22] P. G. Selinger et al. Access Path Selection in a Relational Database Management System. In *SIGMOD 1979*. 23–34.

[23] H. E. Williams et al. Fast Phrase Querying with Combined Indexes. *ACM Trans. Inf. Syst.* 22, 4 (2004). 573–594.

[24] M. Zukowski et al. Super-Scalar RAM-CPU Cache Compression. In *ICDE 2006*, 59–59.