Accelerating Entity Lookups in Knowledge Graphs Through Embeddings

Ghadeer Abuoda[‡], Saravanan Thirumuruganathan[†], Ashraf Aboulnaga[†]
[‡]College of Science and Engineering, HBKU
[†]Qatar Computing Research Institute, HBKU
{gabuoda, sthirumuruganathan, aaboulnaga}@hbku.edu.qa

***Abstract*—Tabular data is widespread on the web and in enterprise data lakes. Recently, there has been increasing interest in developing algorithms for matching tabular data with knowledge graphs. This involves learning correspondences between tabular entities such as cells, rows, and columns and entities in the knowledge graph. Such semantic annotation of tabular entities has numerous applications such as entity disambiguation, knowledge graph expansion, error detection and repair in tabular data, and more. A key first step for all these applications is the *lookup* function that matches a query string to a candidate set of knowledge graph entities. Despite the importance of entity lookup, current implementations are not optimized, not robust to misspellings, and ignore semantic relationships.**

To address these problems, we represent each entity as an embedding – a compact vector representation that is cognizant of syntactic and semantic similarities and supports fast lookup. We propose, EMBLOOKUP**, a novel and efficient approach for learning such an embedding.** EMBLOOKUP **is based on deep metric learning with triplet loss and supports accurate and efficient lookup of knowledge graph entities. We conduct extensive experiments that demonstrate that** EMBLOOKUP **achieves 1-2 orders of magnitude speedup while being tolerant to many types of errors in the query and data. We demonstrate the generality of** EMBLOOKUP **over diverse application scenarios in semantic table annotation, entity disambiguation, and data repair.**

I. INTRODUCTION

Tabular data is the most common format to publish data on the web and in enterprise data lakes. This could be in the form of relational databases, CSV files, or spreadsheets. The usability of tabular data in different applications is often limited by missing or incomplete metadata and other data quality issues. A promising approach for improving the usability of tabular data is to leverage *knowledge graphs (KGs)* as a semantic reference for elements in the tables. Specifically, there has been extensive recent work on *semantic table annotation*. This involves linking individual table elements such as cells, columns, and their respective relations to resources from knowledge graphs such as classes (types), entities (elements), and properties (relations). Once such a mapping is established, it could be used for various downstream tasks such as cell-entity annotation, column type annotation, data error detection and repair, KG expansion and completion, and more.

The Entity Lookup Operation. These seemingly diverse applications rely on a fundamental operation as the first step – *lookup*. Given a keyword q, lookup [11], [19] retrieves a set of entities in the KG that are most relevant to q. For example, looking up the keyword BERLIN on Wikidata might result in entities such as Q64 (capital city of Germany), Q56037 (East Berlin), Q821244, or Q614184 (cities named Berlin in the USA). These entities are then processed in an application-specific manner. For example, entity disambiguation could rank these entities according to some metric and return the top-ranked entity. Note that the keyword q may not necessarily match precisely with the text of the relevant entity, for example, due to misspellings. Thus, the lookup operation should support fuzzy matching based on some similarity metric. We return to this point later in this section. Despite the seeming simplicity, lookup is a crucial building block for many knowledge graph applications. In fact, almost all of the state-of-the-art approaches for semantic table annotation rely on multiple lookup services followed by sophisticated lexical matching [11], [14], [16], [24].

Prior Approaches and Their Limitations. Let us first consider the syntactic lookup setting where the goal is to retrieve a set of candidate entities from a possibly misspelled query string. For example, given strings such as BERLIN or BELRIN, lookup must retrieve entities including Q64, Q56037, Q821244, and Q614184. Prior work on semantic table annotation uses three common approaches for performing the lookup operation. First, one could leverage remote lookup services (such as Wikidata or DBPedia lookup) through their API endpoints. However, this approach suffers from rate limits imposed by the endpoints and, thus, cannot allow extensive lookups. Furthermore, there is limited support for fuzzy queries as they might require an expensive table scan. Second, one could build a local index using the titles of all entities in the KG. Such an index requires substantial storage requirements, and it is often targeted towards a specific similarity metric. For example, Elastic Search[1] uses a weighted combination of word and trigram based BM25 score for fuzzy matching [4]. Support for other metrics is either limited or has to be implemented manually. The final approach solves this issue by implementing the custom similarity metrics from scratch or using fuzzy matching libraries such as FuzzyWuzzy [22][2]. Unfortunately, none of these approaches is well optimized for bulk lookup involving millions of queries, which is often needed. For example, consider the SemTab 2020 challenge [16], which aims to benchmark systems for semantic

[1]https://www.elastic.co
[2]https://pypi.org/project/fuzzywuzzy

table annotation. The third round of this challenge required semantic annotation of 768K cells from 63K tables. Some of the submissions required as much as 2-3 days using remote lookup services [4] and up to 96 hours [12] locally, even when using an optimized Levenshtein distance module.

Support for Sophisticated Lookup Operations. Almost all of the current implementations of lookup are focused on syntactic similarity (such as edit distance). This severely limits the applicability of lookup in several real-world settings such as enterprise data lakes where (a) there is often a wide variety of errors and data cleanliness issues, and (b) semantic similarity is not always captured by syntactic similarity. For example, the lookup operation must be able to determine that the pairs (GERMANY and DEUTSCHLAND), (EUROPEAN UNION and EU), and (BILL GATES and WILLIAM GATES) are all similar. In other words, searching for DEUTSCHLAND should retrieve the entity corresponding to GERMANY, even though DEUTSCHLAND is *not* an entity but an alias for the entity GERMANY.

We would like to note that knowledge graph embeddings are not directly applicable in this scenario. Such embeddings are focused on learning similarities between entities (such as GERMANY and BERLIN) and not on entity mentions/aliases (such as GERMANY and DEUTSCHLAND). While one could efficiently retrieve the embedding for an entity using its *entity-id*, retrieving the embedding based on a *string* requires a two-step process – identify the entity id for the string and then retrieve the corresponding entity embedding. For example, one could efficiently retrieve the embedding for entity BERLIN using the entity-id Q64 but not using the string BERLIN.

To the best of our knowledge, we are not aware of any existing system that can handle both syntactic and semantic similarities and is robust to typos, abbreviations, aliases, etc. In other words, the system must be able to retrieve the entity GERMANY given diverse strings for lookup such as GERMANY, DEUTSCHLAND, or even GERMONEY.

Overview of Our Approach. We propose EMBLOOKUP, a novel and unified approach for entity lookup. Instead of optimizing for individual similarity metrics (such as edit distance), our framework trains a deep learning (DL) model to learn a generic similarity metric in an *unsupervised* manner. Our approach is based on deep metric representation learning with triplet loss [3], [20]. We train our model based on triplet strings of the form (a, p, n). Our goal is to learn an appropriate representation such that the embedding of the anchor a is closer to the positive example p than the negative example n. We generate the training data through a careful process of triplet mining that ensures that the embedding for an arbitrary lookup query q will be closer to entities that are similar to it semantically or syntactically. We design an appropriate DL architecture that can balance both syntactic and semantic similarities. We index the embeddings on the entities in a space-efficient manner using *product quantization* [13]. This allows us to have a compact entity embedding index that is smaller than a wide variety of other indexing techniques while supporting fast queries. Our proposed approach is modular and can be easily customized for specific applications. EMBLOOKUP is robust to a wide variety of noise types and is also optimized for bulk querying.

Problem Scope. EMBLOOKUP is not a new KG embedding algorithm. Nor is it a new algorithm for various semantic annotation tasks. Instead, EMBLOOKUP seeks to optimize the lookup operation that underpins diverse semantic annotation applications over large tabular data and knowledge graphs. Our goal is to be a transparent (and efficient) replacement for syntactic lookup services used by prior work. As an additional bonus, EMBLOOKUP also provides support for semantic lookups.

Summary of Experimental Results. We conduct extensive experiments that show that EMBLOOKUP is fast, compact, and accurate for both syntactic and semantic lookups. Our experiments are conducted over two major knowledge graphs – Wikidata and DBPedia. We demonstrate the generality of EMBLOOKUP by choosing a diverse set of applications and replacing the lookup service of each application with that of EMBLOOKUP. We conduct experiments over six applications from the *four* diverse application scenarios of cell type identification, column type identification, entity disambiguation, and data repair. EMBLOOKUP achieves 1-2 orders of magnitude speedup in the lookup operation in each of these applications. Interestingly, it achieves this feat with almost no reduction in accuracy. We also evaluate these applications under wide variety of noise injection techniques and show that EMBLOOKUP is more tolerant of misspellings and other errors than the original lookup implementations.

Summary of Contributions. We make the following major contributions.

1) We identify the crucial role played by the *lookup* operation in many KG applications. Existing approaches are not expressive and not optimized for the bulk lookup operations needed in enterprise data lake settings.
2) We propose EMBLOOKUP, a novel framework that can perform lookup based on syntactic and semantic similarities and is robust to a wide variety of errors.
3) We conduct experiments over four application scenarios and demonstrate that using EMBLOOKUP results in significant speedup without sacrificing accuracy.

Paper Outline. In Section II, we introduce the relevant terminology and desiderata for the lookup operation. We describe the technical details of EMBLOOKUP in Section III. We rigorously evaluate the efficacy of EMBLOOKUP in Section IV. Relevant prior work is described in Section V followed by parting thoughts in Section VI.

II. PRELIMINARIES

In this section, we introduce the necessary terminology, define the lookup operation, and highlight concrete applications that rely on lookup. Finally, we enumerate the desiderata that must be satisfied by an ideal lookup operation.

Tabular Data. We are given a relational table T with m rows and n columns represented as $T = \{(t_{1,1}, \ldots, t_{1,n}), \ldots, (t_{m,1}, \ldots, t_{m,n})\}$. We represent the i-th row as $r_i = (t_{i,1}, \ldots, t_{i,n})$ and j-th column as $c_j = (t_{1,j}, ..., t_{m,j})$. $t_{i,j}$ represents the cell at the intersection of the i-th row and j-th column. $t_{i,j}$ could represent an entity or could be a literal such as a number.

Knowledge Graph (KG). We represent the KG as a quadruplet $\langle \mathcal{E}, \mathcal{T}, \mathcal{P}, \mathcal{F} \rangle$ [4]. $\mathcal{E} = \{e_1, e_2, \ldots, \}$ is the set of entities, $\mathcal{T} = \{\tau_1, \tau_2, \ldots\}$ is the set of types, and $\mathcal{P} = \{p_1, p_2, \ldots, \}$ is the set of properties. $\mathcal{F} = \{f_1, f_2, \ldots\}$ is the set of facts, where each fact f_i is a triplet $\langle s_i, p_i, o_i \rangle$ with $s_i \in \mathcal{E}$, $p_i \in \mathcal{P}$, and o_i could be an entity or a literal.

Lookup Operation. We use the term *entity mention* to denote a string that refers to an entity. For example, the strings GERMANY, DEUTSCHLAND, FRG, and BRD could all refer to the entity GERMANY. Given an entity mention q and a non-negative integer k, the goal of *lookup*(q, k) is to return a set of entities $C_{q,k} = \{e_{q,1}, \ldots, e_{q,k}\}$ where each $e_{q,i} \in \mathcal{E}$ and is relevant to q. A relevance scoring function $score(q, C_{q,k})$ measures the suitability of each $e_{q,i}$ to q. An ideal lookup operation seeks to efficiently retrieve $C_{q,k}$ that maximizes $score(q, C_{q,k})$. For example, the lookup *lookup*('Berlin', 4) could return {Q64, Q56037, Q821244, Q614184}. Given q, we refer to the set $C_{q,k}$ as a candidate set of entities. The post processing of $C_{q,k}$ is often application specific. Our goal is to improve and accelerate the lookup function *lookup*(q, k) for any q and k.

There are two broad categories of lookup operations. The *syntactic* lookup retrieves the set of candidate entities that are lexicographically similar to the query string according to a distance function such as edit distance or Levenshtein distance (e.g., GERMANY and GERMONEY). The *semantic* lookup retrieves the set of candidate entities that are conceptually related to the entity represented in the query string (e.g., GERMANY and DEUTSCHLAND). Note that semantically similar entities need not be syntactically similar and vice versa.

Semantic Annotation Tasks. We demonstrate the generality and efficacy of EMBLOOKUP for four major tasks.

- Cell Entity Annotation (CEA): Given an entity mention $t_{i,j} \in T$, the goal of CEA is to find entity $e_k \in \mathcal{E}$ that is referred to by $t_{i,j}$.
- Column Type Annotation (CTA): Given a column $c_j \in T$, the goal of CTA is to associate it with the most specific entity type $\tau_k \in \mathcal{T}$.
- Entity Disambiguation (EA): Given a list of entity mentions $\langle q_1, q_2, \ldots, q_K \rangle$, the goal of EA is to output an entity configuration $\langle E_1, E_2, \ldots, E_K \rangle$ where each $E_i \in \mathcal{E}$ is the correct entity referred to by q_i.
- Data Repair (DR): Given a row $r_i = (t_{i,1}, \ldots, t_{i,n})$ with some missing values, DR imputes the missing values using the knowledge graph $\langle \mathcal{E}, \mathcal{T}, \mathcal{P}, \mathcal{F} \rangle$.

Desiderata of Lookup Operation. The goal of EMBLOOKUP is to design an efficient and effective lookup operation that can handle both syntactic perturbations (such as typos, abbreviations) and semantic similarities (such as aliases and synonyms). It should provide fast lookup with accuracy comparable to the traditional syntactic approaches, even for large values of k. Finally, it should support bulk querying and must be able to leverage GPU acceleration when available.

III. EMBEDDINGS FOR ACCELERATING LOOKUPS

In this section, we describe the model architecture for EMBLOOKUP. We first describe how to learn embeddings for an entity lookup involving a single query string. Next, we describe how to compress the entity embedding without affecting retrieval accuracy.

Overview of Our Approach. Figure 1 illustrates the major components of EMBLOOKUP. The embedding learner takes as input a set of triplet strings of the form (a, p, n) that demonstrate similar and dissimilar entity mentions. It seeks to learn an embedding for arbitrary query strings that encodes both syntactic and semantic similarity. Specifically, the goal is to learn an appropriate representation such that the embedding of the anchor a is closer to the positive example p than the negative example n. Once the embedding learner is trained, we generate embeddings for each entity in the knowledge graph. We describe the sub-components of the embedding learner in the following sections and illustrate it in Figure 2. We then compress these entity embeddings using product quantization and store them. Given a new query string, we compute its embedding and search for the k nearest neighbors from the compressed embedding index. We then return the corresponding entities as the output of the lookup operation.

A. From Symbolic to Continuous Representations for Lookup

Consider a hypothetical function $D(\cdot, \cdot)$ that takes two entity mentions m_i and m_j as input and outputs a score that takes syntactic and semantic similarities between m_i and m_j into account. Hence, when looking up an entity mention m_i, D should return a high similarity score for entities m_j that are similar to m_i.

Currently, most of the lookup functions operate on the symbolic representation, i.e., the strings. This representation is appropriate for simple variants such as fuzzy lookups involving simple typos. However, this representation is not expressive enough to handle many common scenarios such as aliases (GERMANY and DEUTSCHLAND) and abbreviations (EUROPEAN UNION and EU), among others. Hence, we propose a transformation to the continuous domain where we transform each entity mention into an embedding such that this embedding encodes both syntactic and semantic similarity. Specifically, we represent each entity as a 64-dimensional vector (requiring $64 \times 4 = 256$ bytes of storage that we shall compress to 8 bytes in Section III-D). We reiterate that our approach only has a limited connection to knowledge graph embeddings. KG embeddings are often expensive to train and are optimized for tasks such as link prediction and node classification and cannot be used for lookup. In contrast,

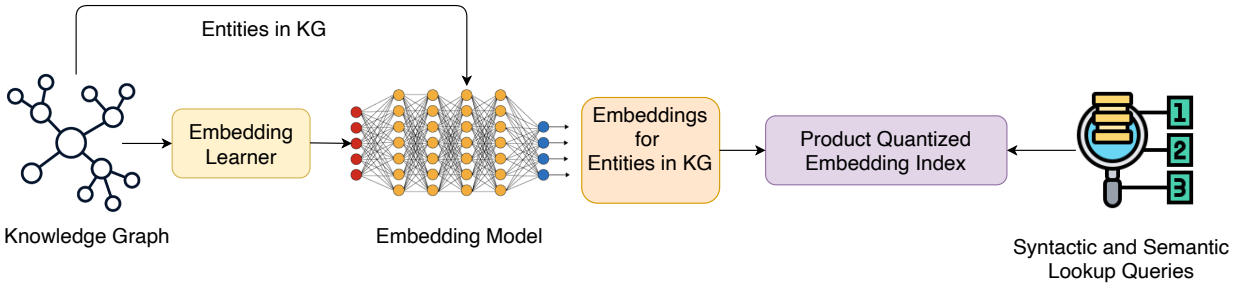

Fig. 1: Overview of EMBLOOKUP.

our approach is lightweight, efficient, and designed explicitly for accelerating lookups.

Specifically, we propose a two-step approach. First, we *learn* an embedding transformation $f(\cdot)$ that takes an entity mention and converts it into an embedding such that a predetermined distance function $d(\cdot, \cdot)$ (such as cosine distance or Euclidean distance) could be used to measure the similarity between the corresponding mentions. Ideally, we would like to ensure that

$$D(m_i, m_j) \approx d(f(m_i), f(m_j)) \tag{1}$$

However, the function D is hypothetical and cannot be directly estimated. Hence, we take an indirect approach for *approximation*. Consider three entity mentions m_i, m_j, and m_k. We would like to ensure that whenever $D(m_i, m_j) < D(m_i, m_k)$

$$d(f(m_i), f(m_j)) < d(f(m_i), f(m_k)) \tag{2}$$

This objective will ensure that the *order* of the distance function is maintained. In other words, if m_j is closer to m_i than m_k according to function D, we would like our learned similarity function to exhibit a similar ordering. This ordering constraint will ensure that the top-l results are more relevant than the top-l' results for $l' > l$. Without loss of generality, for the rest of the paper we assume d to be the Euclidean distance. Our goal then boils down to learning an embedding function $f(\cdot)$.

B. Embeddings for Entity Lookup

In this section, we describe how to convert a single entity mention into an embedding. Note that the mention could consist of a single word (such as 'Germany') or multiple words (such as 'East Berlin'). We use a deep leaning approach for generating the embeddings. Hence, there are two key design choices to be made: (a) the DL architecture for converting entity mentions to embeddings, and (b) the appropriate loss function for optimization.

Data Preprocessing. Let e be an entity in the knowledge graph and let $m_1, m_2, \ldots$ correspond to the entity mentions of e. These could be obtained from the labels of the entities and from aliases obtained from properties such as rdfs:label, skos:altLabel, and dbo:wikiPageWikiLinkText [11]. Let $\mathcal{A}$ be the alphabet of the entity mentions and let L be the length of the longest entity mention. Given a single character c, we can represent it as an L-dimensional vector that has the value 0 for all dimensions except $pos(c)$, where $pos(c)$ provides the positional order of character c in $\mathcal{A}$. Then, given a string m_q, we can extend this one-hot encoding to convert it into a matrix X of dimensions $|\mathcal{A}| \times L$. The i-th column of X stores to the one-hot encoding of the i-th character of m_q. If there are fewer than L characters in m_q, then the last $L - |m_q|$ columns of X are filled with 0. This transformation is widely used, including in [8], [23].

Example. Suppose that $\mathcal{A} = \{a, b, c, d, e\}$ and $L = 4$. Given an entity mention $m_q =$ 'cad', we represent it as a matrix with 5 rows and 4 columns whose *column* values are [0, 0, 1, 0, 0], [1, 0, 0, 0, 0], [0, 0, 0, 1, 0] and [0, 0, 0, 0, 0].

DL Network Architecture. EMBLOOKUP uses two separate embedding models for modeling syntactic and semantic similarities. We empirically found that using a single embedding model that subsumes both syntactic and semantic similarities was less accurate than using two separate models. Furthermore, this two-model approach allows further customization. As we shall describe later, we bootstrap the model for semantic similarity with the model of fastText [2]. The syntactic embedding model uses a CNN-based architecture that has an inductive bias for syntactic perturbations [8].

We use convolutional neural networks (CNNs) for learning an embedding that estimates syntactic similarity. There are two key reasons. First, CNNs can be trained much more efficiently than NLP-based DL architectures. Second, it has been recently observed [8] that CNNs have an appealing property wherein the output of a CNN model with max-pooling over two strings that are transformed using one-hot encoding preserves the bounds on edit distance. Our DL architecture consists of 5 convolutional layers with 8 kernels of size 3 in each of them. We use max-pooling to aggregate outputs.

We piggyback on the extensive work from the NLP community on word embeddings for learning embeddings for semantic similarity. First, we generate training data consisting of entity names and their synonyms. We then use fastText [2] to learn an embedding model such that embeddings of entity names and their synonyms are close together. The fastText model produces a 64-dimensional embedding. Intuitively, the CNN and the fastText model have complementary strengths.

While the embeddings from the CNN model can handle syntactic similarities and misspellings, they cannot handle semantic similarity. On the other hand, the fastText model can handle semantic similarities but cannot easily handle misspellings. Finally, we aggregate the output of the two diverse embeddings using a two-layer linear layer with ReLU activation to produce a single 64-dimensional embedding vector.

Triplet Loss Function. We use the aforementioned model (CNN and fastText) for transforming a string into an embedding. The goal is now to train the model using an appropriate loss function so that it encodes syntactic and semantic similarities. We use the *triplet loss* function [3], [20] that takes a triplet (anchor, positive, negative) and outputs a score. It is formally defined as

$$\max(||f(anchor) - f(positive)||_2^2 - ||f(anchor) - f(negative)||_2^2 + margin, 0) \quad (3)$$

where $f(\cdot)$ is the embedding transformation, and $margin$ is a positive value that accentuates the difference in distance between the pairs (anchor, positive) and (anchor, negative). This formulation will ensure two main properties. First, the distance between the embeddings of anchor and negative is larger than the distance between the embeddings of anchor and positive. Second, the distance between the embeddings of anchor and negative is larger than the margin.

Model Training Procedure. Given a triplet, we encode each of its components using one-hot encoding, pass the encoded matrices for the triplets through the embedding model and use it to compute the triplet loss function. We then back-propagate the loss function to improve the embedding model. This process is repeated for 100 epochs. We use a batch size of 128 and use the Adam optimizer.

Triplet Generation. The core of the training process is triplet mining, which involves collecting the appropriate set of (a, p, n) triplets. For each entity $e_i \in \mathcal{E}$, we generate triplets for encoding *semantic similarity* as follows. First, we generate a set of synonyms (such as rdfs:label, skos:altLabel, dbo:wikiPageWikiLinkText, etc.). Each of these becomes a positive example. We generate the negative examples by choosing labels of randomly chosen entities. For example, given the entity Q183 ('Germany'), some potential triplets include: {(Germany, FRG, blah1), (Germany, Deutschland, blah2), (Germany, Federal Republic of Germany, blah3), (Germany, BRD, blah4), ...}. Here 'blahX' is the label of a randomly chosen entity that is unlikely to be related to Germany.

Heuristics for Triplet Mining. It is possible to use heuristics to generate more informative triplets in order to improve the training dataset. We next describe a couple of such heuristics that are employed by EMBLOOKUP to enable incorporating some domain knowledge. We would like to note that these heuristics are non-comprehensive. First, we generate *additional* triplets to encode *syntactic* similarities. As mentioned above, CNNs could encode simple edit distance-based similarities. Given an anchor, one could produce additional *positive* elements by artificially injecting certain types of errors such as dropping some characters, transposing pairs of characters, adding unrelated characters, and so on. If there are some errors that are pretty common, this approach provides a seamless process to inject such domain knowledge. For example, the triplets generated could be {(Germany, Grmany, blah1), (Germany, Germaney, blah2), (Germany, Gemrany, blah3), ...}. Second, we generate triplets where the anchor and positive belong to the same type or share some property. For example, we generate triplets such as {(Germany, France, blah1), (Germany, Austria, blah2)} where the anchor and positive are both countries. This provides a lightweight way to incorporate some semantic similarity between entities based on their types or properties.

We also vary the training procedure to improve the learning. Recall that we use 100 epochs for learning the embedding. For the first 50 epochs, we follow a static offline approach where we apply the triplet loss on all the triplets in the training data. For the next 50 epochs, we follow an online mining approach where only a subset of 'hard' triplets are used for training. Recall from Equation 3 that given a triplet (a, p, n), a margin m, and a distance function d, the triplet loss function is defined as

$$\mathcal{L} = max(d(a, p) - d(a, n) + m, 0)$$

Here we abuse the notation so that a, p, n also represent their corresponding embeddings $f(a), f(p), f(n)$, respectively. Whenever $d(a, p) + m < d(a, n)$, the loss function becomes 0. These correspond to 'easy' triplets that the embedding model has already learned. By repeatedly including these easy triplets, we reduce the average triplet loss thereby slowing the learning process. There are two types of 'hard' triplets – those with $d(a, n) < d(a, p)$ and those with $d(a, p) < d(a, n) < d(a, p) + m$. When $d(a, n) < d(a, p)$, the embedding of n is closer to the anchor a than p. These are often called hard triplets. When $d(a, p) < d(a, n) < d(a, p)+m$, the embedding of n is not closer to a than p, but it still has a positive loss. For such semi-hard negative triplets, it is often preferable to nudge the embeddings such that loss becomes 0. Hence, we focus only on the harder triplets in the second half of the training procedure and ignore the easy triplets. Figure 2 provides a high-level overview of the training process of EMBLOOKUP.

C. Embedding Lookup as Similarity Search

Once the embedding model is learned, we use it to compute the embedding of each entity $e \in \mathcal{E}$. We represent each entity as a 64-dimensional embedding vector. By default, we use the label of the entity for computing its embedding. For example, the embedding for entity Q183 is derived from its label 'Germany'. One could obtain alternate embeddings for Q183 by evaluating the embedding model on its aliases (such as Deutschland, FRG, and BRD). This could possibly increase the lookup accuracy but with higher storage and retrieval cost.

Let $\mathcal{I}$ be a matrix of dimension $N \times D$ where N is the number of embeddings and D is the embedding dimension

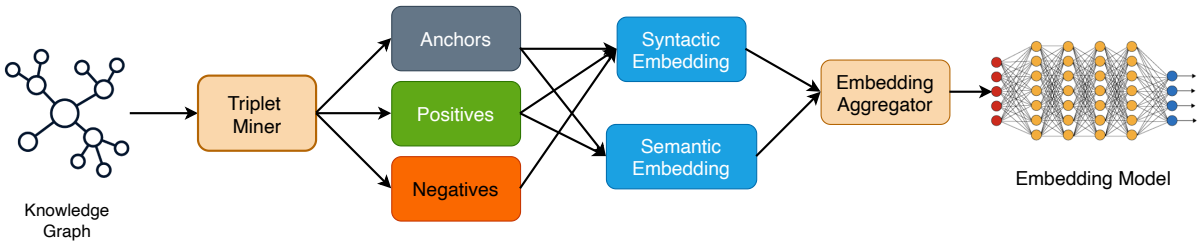

Fig. 2: Process for training embeddings for EMBLOOKUP.

(such as 64). The i-th row corresponds to the embedding of entity e_i. Given a lookup query *lookup*(q, k), we convert q into an embedding vector $f(q)$ and then retrieve the k entities from $\mathcal{I}$ whose embeddings are nearest to $f(q)$ based on the distance function d. A naive brute-force approach would compute the distance between $f(q)$ and each embedding in $\mathcal{I}$ and return the nearest top-k.

One could accelerate the similarity search by trading off some accuracy for efficiency. A number of applications (including those that we evaluate in Section IV) are robust to such approximation. EMBLOOKUP is modular and could accommodate either exact or approximate similarity search. There are several libraries for performing efficient *approximate* similarity search such as FAISS[3], nmslib[4], and annoy[5]. These libraries preprocesses $\mathcal{I}$ and build an index that can accelerate lookups by avoiding distance computation with all embeddings in $\mathcal{I}$. We conducted an empirical analysis and settled on FAISS [17] as it provides a wide variety of indexing options, requires less storage, and allows GPU acceleration.

D. Embedding Compression Through Quantization

Approximate similarity search achieves speedup through indexing and minimizing the number of distance computations. It is possible to achieve further speedup by approximating the embeddings being stored. While there are many approaches, we focus on embedding compression. Intuitively, the goal is to identify an alternate representation for each embedding that requires less storage. The reduced storage results in a smaller index and faster retrieval. The key challenge is to choose an appropriate method for compression that achieves faster retrieval without significant loss of accuracy. By default, we represent each embedding as a 64-dimensional vector requiring 256 bytes. Hence, we need at least 256 MB for storing the embeddings for 1 million entities. We use product quantization to compress the embeddings so that each entity can be represented with just 8 bytes.

Recall that $\mathcal{I}$ is an $N \times D$ matrix storing the embeddings. The goal of product quantization is to reduce the storage cost of $\mathcal{I}$ such that each embedding can be represented with D' bytes. Note that this is different from dimensionality reduction, which transforms each embedding into a low-dimensional vector. We begin by partitioning the embedding vector into groups of length D' each. For example, if $D = 64$ and $D' = 8$, then the groups correspond to dimensions $\{0-7, 8-15, \ldots, 56-63\}$. This will result in 8 matrices of size $N \times D'$, where the j-th sub-matrix stores the embedding dimensions $[j \times D', (j+1) \times D' - 1]$. We next run k-means clustering on the rows of these sub-matrices with $k = 2^{D'}$. Using the example above, we run it with $k = 2^8 = 256$ resulting in k centroids for each of the 8 matrices. We can 'compress' the matrix by treating the k centroids as a codebook. Consider a sub-matrix M_j of size $N \times D'$. First, we replace each row of M_j with the closest centroid resulting in limited approximation. One could further save space by storing the cluster id instead of the entire cluster. Given that there are only 256 clusters, we could represent the cluster id using one byte. This allows us to store the entire sub-matrix using just N bytes and the entire matrix with $N \times D'$ bytes. The mapping between cluster ids and centroid values is stored in a codebook. Please refer to [13] for additional details on product quantization.

Embedding compression is an optional component of EMBLOOKUP. Hence, an accuracy-conscious practitioner could disable the embedding compression component. A practitioner could select an appropriate compression mechanism based on their accuracy-storage-latency tradeoff requirements. For example, EMBLOOKUP uses product quantization since many applications of EMBLOOKUP (such as those that are evaluated in Section IV) retrieve a large number of neighbors (k=20-100) and perform customized post-processing, which minimizes the impact of approximation due to compression.

IV. EXPERIMENTS

We would like to reiterate that EMBLOOKUP is not a new algorithm for KG embedding or semantic annotation. Instead, it provides an alternate lookup implementation that is efficient and robust to misspellings and can be transparently used by various applications. In this section, we experimentally evaluate EMBLOOKUP through this prism. The code for EMBLOOKUP can be found at https://github.com/qcri/EmbLookup.

In our first set of experiments, we show how EMBLOOKUP could be used for accelerating lookups in four tasks related to three different application scenarios – semantic table annotation, entity disambiguation, and data repair. We demonstrate the speedup through three concrete systems for semantic table annotation and one each for entity disambiguation and data repair. These experiments are conducted using three bench-

[3]https://github.com/facebookresearch/faiss
[4]https://github.com/nmslib/nmslib
[5]https://github.com/spotify/annoy

mark datasets over two major knowledge graphs – Wikidata and DBPedia. The experiments show that EMBLOOKUP can achieve 1-2 orders of magnitude speedup with almost no reduction in accuracy. Second, we show that EMBLOOKUP is robust to common types of error such as misspellings. The performance of the five chosen systems experiences a steep drop under extensive noise. However, the impact is comparatively minor for EMBLOOKUP. Third, we show that EMBLOOKUP can support semantic lookups. In contrast, the chosen systems provide poor results when they rely on local indices that can only support syntactic lookups. Finally, we conduct an analysis to study the impact of the major hyperparameters of EMBLOOKUP.

Hardware and Software Platform. We conducted all experiments on an Intel Xeon E5-2686 CPU with 18 cores and 64 GB RAM. We implemented EMBLOOKUP in Python 3.8. The DL models were implemented in PyTorch. The model training and GPU acceleration of FAISS was done on an Nvidia V100 GPU. The end-to-end training of EMBLOOKUP including triplet mining, learning embeddings, and quantization took less than 1 hour on the GPU.

Dataset. The statistics of the tabular datasets used in our experiments can be found in Table I. The ST-Wikidata and ST-DBPedia datasets are from SemTab-2020 [16] and SemTab-2019 [15]. SemTab is an annual competition for benchmarking systems for matching tabular data with knowledge graphs. As the name indicates, ST-Wikidata and ST-DBPedia seek to match tabular datasets with the Wikidata and DBPedia knowledge graphs, respectively. The third dataset, Tough Tables [7], is intended as a challenging dataset for semantic table annotation tasks. It consists of tables that have a wide variety of noise and ambiguous cell values. For each of the tabular datasets, the ground truth labels are available. Specifically, for each cell $T_{i,j}$ to be annotated, we know the correct KG entity id (such as 'Germany'=Q183). For each column c_j, we are provided with the correct type (such as c_j is of type 'country'=Q6256). We used these tabular datasets to evaluate the performance of EMBLOOKUP on the four tasks defined in Section II: Cell Entity Annotation (CEA), Column Type Annotation (CTA), Entity Disambiguation (EA), and Data Repair (DR). In order to evaluate the DR task, we randomly replaced 10% of the cells with missing values. The goal of the DR task is to impute those cells correctly.

TABLE I: Statistics of the tabular datasets.

	ST-Wikidata	**ST-DBPedia**	**Tough Tables**
#Tables	109K	14K	180
Avg #Rows	6.6	26.2	1080
Avg #Cols	4.1	5.1	804
#Cells to annotated	2.03M	877K	663K

For the CEA and CTA tasks, we chose three systems from SemTab 2020: bbw [24], MantisTable [6], and JenTab [1]. In the challenging Round 4 of the competition, each of these submissions achieved a precision higher than 0.98 for the CEA task and 0.93 for the CTA task. Each of them used different lookup services and lexical matching techniques, illustrating that EMBLOOKUP is flexible enough to replace a broad variety of lookup services. Both the ST-Wikidata and ST-DBPedia datasets were evaluated using these three systems. For EA and DR, we chose DoSeR [30] and Katara [5], respectively. Note that each of these systems is chosen as a competitive representative for the corresponding task. We do not necessarily claim that they represent the latest in the state of the art. Instead, our goal is to show that EMBLOOKUP could be easily plugged in and provides acceleration for a wide variety of recent real-world systems.

Each of these systems was evaluated on two variants of the ST-Wikidata and ST-DBPedia datasets: *no error* and *error*. The no error variant consists of the original dataset, whereas in the error variant we injected some noise into 10% of the cells. These include common misspellings such as dropping/inserting one or more letters, transposing letters, swapping the tokens, abbreviations, and so on.

Performance Metrics. We compare EMBLOOKUP to the alternatives based on two criteria: speed and accuracy. We measure speed by reporting the speedup of EMBLOOKUP over the baselines. We use the F-score to measure accuracy.

A. *Speedup of* EMBLOOKUP *for Semantic Annotation Tasks*

In our first set of experiments, we replaced the lookup component of the five systems (bbw, MantisTable, JenTab, DoSeR, and Katara) with EMBLOOKUP. Lookup is one of the key bottlenecks, accounting for as much as 45% of the time taken in each of these systems. We instrumented the time taken by the original lookup function and that of EMBLOOKUP and report the speedup ratio. Note that we instrumented the time taken and not the throughput. This is to provide a fair comparison for remote services that impose rate limits. For example, Wikidata only allows five parallel queries per IP. We evaluate two variants of the lookup operation. The first variant (denoted as EL) corresponds to EMBLOOKUP where the embeddings are compressed through product quantization. The second variant (denoted as EL-NC) corresponds to EMBLOOKUP where the compression component is disabled. This segregation allows us to investigate the impact of compression on both accuracy and speedup.

Performance of EMBLOOKUP with Compression (EL). Table II shows the results for the ST-Wikidata dataset. Note that we used the benchmark results without any injection of noise. We can see that the F-score of the original lookup and EMBLOOKUP are almost identical where the largest difference is just 0.03. EMBLOOKUP does not lose much accuracy even though product quantization results in approximation of the embeddings. This showcases an especially appealing property of EMBLOOKUP: even though EMBLOOKUP is local, it is competitive with remote lookup services (such as SPARQL endpoints and meta-search engines that query multiple services). Since it is local, it can retrieve results at a much faster

TABLE II: Performance of EMBLOOKUP in accelerating lookups of various systems in **ST-Wikidata** dataset. EL denotes EMBLOOKUP while EL-NC denotes EMBLOOKUP without embedding compression.

		Speedup (CPU)		Speedup (GPU)		F-Score		
Task	**System**	**EL**	**EL-NC**	**EL**	**EL-NC**	**Original**	**EL**	**EL-NC**
CEA	bbw	20x	13x	78x	47x	0.92	0.92	0.92
CEA	MantisTable	37x	29x	83x	62x	0.86	0.85	0.86
CEA	JenTab	28x	25x	92x	76x	0.89	0.88	0.89
CTA	bbw	61x	42x	122x	73x	0.82	0.82	0.82
CTA	MantisTable	48x	33x	146x	104x	0.89	0.89	0.89
CTA	JenTab	64x	48x	163x	106x	0.78	0.78	0.78
Entity Disambiguation	DoSeR	40x	35x	54x	42x	0.87	0.84	0.86
Data Repair	Katara	28x	20x	96x	61x	0.76	0.76	0.76

TABLE III: Performance of EMBLOOKUP in accelerating lookups of various systems in **ST-DBPedia** dataset. EL denotes EMBLOOKUP while EL-NC denotes EMBLOOKUP without embedding compression.

		Speedup (CPU)		Speedup (GPU)		F-Score		
Task	**System**	**EL**	**EL-NC**	**EL**	**EL-NC**	**Original**	**EL**	**EL-NC**
CEA	bbw	26x	17x	93x	56x	0.89	0.88	0.89
CEA	MantisTable	42x	33x	102x	76x	0.88	0.85	0.88
CEA	JenTab	38x	34x	118x	97x	0.83	0.83	0.83
CTA	bbw	72x	49x	128x	76x	0.88	0.86	0.88
CTA	MantisTable	56x	38x	155x	110x	0.82	0.80	0.82
CTA	JenTab	73x	55x	168x	107x	0.77	0.77	0.77
Entity Disambiguation	DoSeR	47x	41x	73x	57x	0.81	0.81	0.81
Data Repair	Katara	33x	24x	102x	65x	0.74	0.72	0.74

rate without any rate limits that are imposed by these remote services. As we shall discuss later, EMBLOOKUP achieves this feat even though the index of EMBLOOKUP is comparable to that of other well-optimized indexing systems such as Elastic Search. In addition to achieving high accuracy, we can also see that EMBLOOKUP achieves an order of magnitude speedup when using the CPU. It can achieve further speedup when there is a GPU (up to 2 orders of magnitude). This is partially due to the fact that using product quantization reduces the space requirements of an entity embedding from 256 bytes to 8 bytes. This results in a smaller index allowing for faster retrieval. Table III shows similar trends for the ST-DBPedia dataset. This illustrates that both the accuracy and speedup achieved by EMBLOOKUP are due to the algorithmic choices and not inherently due to the knowledge graph or the tabular datasets used for evaluation.

Performance of EMBLOOKUP without Compression (EL-NC). Next, we investigate the impact of embedding compression on accuracy and speedup. From Tables II and III, we can see that EMBLOOKUP without compression achieves the same F-score as the original systems in all but one case, where the difference is 0.01. Furthermore, EMBLOOKUP achieves significant speedup even without compression. Hence, we can infer that most of the speedup of EMBLOOKUP is achieved by reformulating the lookup problem as finding similar vectors in the embedding space rather than from compression. Nevertheless, compressing the embeddings through product quantization does produce additional boost in both space savings and query time. Since the accuracy loss is at most 0.03, the benefits of compression outweigh the cost.

B. Performance of EMBLOOKUP *under Noisy Tabular Datasets*

In the next set of experiments, we investigate how the performance of EMBLOOKUP is impacted by erroneous data. By default, both ST-Wikidata and ST-DBPedia are largely error free. Hence, we artificially injected errors into 10% of the cells. These include common misspellings such as dropping/inserting one or more letters, transposing letters, swapping the tokens, abbreviations, and so on. In addition to these synthetically created noisy datasets, we also evaluate EMBLOOKUP over the Tough Tables dataset [7], which has a large amount of erroneous cell values and ambiguity to make semantic annotation challenging.

The results of this experiment can be found in Table IV. There is a stark contrast in the performance of the original systems to the no-error scenario (as shown in Tables II and III).

TABLE IV: Performance of EMBLOOKUP under noisy tabular datasets.

Task	System	F-Score for ST-Wikidata		F-Score for ST-DBPedia		F-Score for ToughTables	
		Original	EMBLOOKUP	**Original**	EMBLOOKUP	**Original**	EMBLOOKUP
CEA	bbw	0.59	0.88	0.63	0.83	0.51	0.63
CEA	MantisTable	0.44	0.69	0.51	0.81	0.48	0.68
CEA	JenTab	0.25	0.56	0.38	0.80	0.46	0.64
CTA	bbw	0.48	0.68	0.64	0.82	0.56	0.69
CTA	MantisTable	0.44	0.54	0.54	0.74	0.54	0.68
CTA	JenTab	0.38	0.69	0.47	0.72	0.51	0.66
Entity Disambiguation	DoSeR	0.22	0.69	0.36	0.77	0.24	0.38
Data Repair	Katara	0.36	0.56	0.44	0.66	0.21	0.44

Despite the use of sophisticated lexical and fuzzy matching services, we can see that the performance of many of the competing baselines completely collapsed. In contrast, EMBLOOKUP especially shines when the data is noisy. In fact, the performance of EMBLOOKUP is not that far off from the no-error scenario! The robustness to errors is primarily due to the CNN model and the training based on triplet loss. We would like to note that the retrieval speed of EMBLOOKUP is not affected by the presence or absence of errors. So EMBLOOKUP achieves the same speedup as the no-error scenario.

In summary, EMBLOOKUP achieves comparable accuracy to the original systems when there are no errors and substantially better accuracy in the presence of errors. Finally, it provides 1-2 orders of magnitude speedup irrespective of the level of noise.

C. Comparing EMBLOOKUP with Lookup Services

In the next set of experiments, we perform a head-to-head comparison of EMBLOOKUP against popular local and remote lookup services by evaluating their performance in the CEA task. Both EMBLOOKUP and the baselines are provided with the same set of queries and retrieve the top-10 relevant entities. If the correct entity is in the top-10, the query is considered to be successful. This experiment allows us to quantify the retrieval speed and accuracy of the lookup services.

We focus on *eight* representative approaches that are widely used in semantic annotation tasks. Most of these methods can be executed locally, while two (Wikidata and SearX) involve querying remote services. FuzzyWuzzy [22] is a fuzzy string matching package written in Python. It uses Levenshtein distance to calculate the differences between string sequences. Elastic Search is an optimized local search engine that supports fuzzy queries, also using Levenshtein distance. Locality Sensitive Hashing (LSH) [9] is an approach for efficient approximate nearest neighbor search. We chose a variant optimized for Levenshtein distance [25]. In addition, we also evaluate three syntactic operations based on exact match, q-gram, and Levenshtein distance. We compare EMBLOOKUP against optimized implementations of these operations in Elastic Search. Our first remote lookup service is the Wikidata SPARQL endpoint. Our second remote lookup service is SearX [21], a metasearch engine that aggregates results from more than 70 search engine.

Table V provides the results of the evaluation. As in the previous experiment, we can see that EMBLOOKUP achieves significant speedup over both local and remote lookup services. Even against a highly optimized search engine such as Elastic Search, EMBLOOKUP provides an order of magnitude speedup. EMBLOOKUP achieves higher speedups over the other lookup services, up to 2 orders of magnitude, and even higher speedups when using GPU. Interestingly, EMBLOOKUP is able to achieve these speedups while also improving accuracy. Additionally, while EMBLOOKUP can handle a wide variety of noise types such as abbreviations or aliases, these noise types cannot be appropriately handled by the other lookup services, regardless of whether they are local or remote and which approximation technique they use.

D. Semantic Lookup using EMBLOOKUP

All of the previous experiments were focused on the relatively straightforward *syntactic* lookups. In this section, we evaluate the performance of EMBLOOKUP for semantic lookups where the query string could be an alias or a synonym of the entity mention. We relied on the fact that for all three datasets (ST-Wikidata, ST-DBPedia, and Tough Tables), we are aware of the ground truth mapping between a cell and the corresponding entity. We generated a variant of these three datasets as follows: for each cell in the tabular dataset to be annotated, we identify the corresponding entity from the knowledge graph using the ground truth. For example, we might know that the string 'Germany' corresponds to the entity with id Q183. We retrieve the properties of this entity and replace the string 'Germany' with one of its synonyms/aliases. If there are none, we do not make any change and lookup the original string ('Germany' in this case). If there are multiple aliases, we choose the replacement uniformly at random. For the vast majority of the entities, there were at least 3 aliases/synonyms. We repeated this process to generate 5 perturbed variants of the underlying datasets. For example, in one variant 'Germany' would be replaced with 'Deutschland', while in another, the replacement string could be 'Federal Republic of Germany'. However, we expect that lookup returns

TABLE V: Comparison of EMBLOOKUP with popular lookup services for the ST-Wikidata dataset.

Approach	Speedup (CPU)	Speedup (GPU)	F-Score (no error)		F-Score (error)	
			Original	EMBLOOKUP	**Original**	EMBLOOKUP
FuzzyWuzzy	89x	168x	0.82	0.88	0.78	0.84
Elastic Search	19x	48x	0.77	0.88	0.63	0.84
LSH	48x	124x	0.72	0.88	0.47	0.84
Exact Match	93x	213x	0.85	0.88	0.72	0.84
q-gram	42x	81x	0.79	0.88	0.77	0.84
Levenshtein	53x	96x	0.82	0.88	0.78	0.84
Wikidata API	148x	224x	0.83	0.88	0.69	0.84
SearX API	220x	366x	0.85	0.88	0.72	0.84

TABLE VI: Performance of EMBLOOKUP for semantic lookup queries.

Task	System	F-Score for ST-Wikidata		F-Score for ST-DBPedia		F-Score for ToughTables	
		Original	EMBLOOKUP	**Original**	EMBLOOKUP	**Original**	EMBLOOKUP
CEA	bbw	0.32	0.89	0.63	0.86	0.51	0.59
CEA	MantisTable	0.29	0.84	0.51	0.81	0.48	0.64
CEA	JenTab	0.21	0.86	0.38	0.83	0.46	0.61
CTA	bbw	0.42	0.77	0.64	0.78	0.56	0.63
CTA	MantisTable	0.43	0.82	0.54	0.76	0.54	0.61
CTA	JenTab	0.48	0.68	0.47	0.62	0.51	0.63
Entity Disambiguation	DoSeR	0.24	0.73	0.36	0.71	0.24	0.34
Data Repair	Katara	0.12	0.71	0.44	0.68	0.21	0.42

the entity corresponding to 'Germany' when looking up either of these strings. We report the average F-score achieved by the original systems and EMBLOOKUP for these 5 variants.

Table VI shows the results. Once again, we can see that the performance of the original systems has a steep drop when looking up the aliases. This is not surprising as most of them rely on local indexing systems (such as Elastic Search) or fuzzy matching libraries (such as FuzzyWuzzy) that are unaware of these aliases. Of course, it is possible to tackle this issue by either using remote lookup services or including the aliases in the local index. Both of these methods have disadvantages. The remote lookup services impose rate limit restrictions that significantly increase the response time. Including the aliases in the local index significantly increases the index size. For example, the *compressed* Elastic Search index for ST-Wikidata requires more than 790 MB (as against 63 MB for only entity mentions). In contrast, EMBLOOKUP does not need to store these aliases as the semantic similarity is encoded through the embedding transformation function $f(\cdot)$ trained using the triplet loss function.

E. Sensitivity Analysis

In the next set of experiments, we investigate the impact of various hyperparameters of EMBLOOKUP. We report the results for the ST-Wikidata dataset. The results for other datasets were quite similar.

Varying the Embedding Algorithm. EMBLOOKUP made two key design choices: (a) we used a CNN based model for learning the embeddings instead of a traditional NLP based model such as LSTM, and (b) we chose to train the embeddings from scratch instead of using a pre-trained model. In this experiment, we vary the algorithm for learning the embeddings and evaluate performance on the CEA task. We used the pre-trained models for word2vec [18], FastText [2], and BERT [10]. We also use an LSTM model trained over the labels and aliases of the KG entities. Table VII shows that EMBLOOKUP outperforms all of these popular embedding algorithms, especially in the presence of errors.

TABLE VII: Varying the embedding generation algorithm.

Embedding	F-score (no error)	F-score (error)
EMBLOOKUP	0.88	0.84
word2vec	0.72	0.29
FastText	0.76	0.72
BERT	0.77	0.68
LSTM	0.86	0.78

Varying the Embedding Dimension. By default, EMBLOOKUP uses a 64-dimensional embedding that is then compressed to 8 bytes using quantization. In this experiment, we study how the embedding dimension impacts accuracy. We vary the embedding dimension from 32 to 256 and

measure accuracy with no errors and with errors. We store the entire embedding in FAISS without compression, allowing us to eliminate the confounding factor caused by product quantization. Table VIII shows the results. While reducing the embedding dimension to 32 results in a big drop in accuracy, there is only a slight improvement when increasing the dimension up to 256. The practitioner could choose an appropriate embedding dimension based on the downstream requirements, with a larger dimension slightly increasing accuracy at the cost of a larger index size.

TABLE VIII: Varying the embedding dimension.

Dimension	F-score (no error)	F-score (error)
32	0.64	0.56
64 (default)	0.88	0.84
128	0.90	0.87
256	0.91	0.88

Impact of Training Data. By default, EMBLOOKUP generates 100 triplets for each entity. Since the total number of entity pairs is quadratic in the number of entities, it might seem that using 100 triplets might not sufficiently cover the whole space. However, this effect is mitigated due to two reasons. First, the number of synonyms is less than 50 for at least 95% of the KG entities in both DBPedia and and Wikidata. Hence, we can completely enumerate all the synonyms by using just 50 triplets. We use the remaining budget of triplets for syntactic perturbations. Note that the CNN-based architecture of EMBLOOKUP already has an inductive bias for such perturbations. Second, as mentioned in Section III-B, we use sophisticated techniques for *triplet mining* to generate more informative triplets in order to improve the training dataset.

In this experiment, we investigate the impact of the number of triplets on the four tasks – CEA, CTA, Entity Disambiguation, and Data Repair. Figure 3 shows the results. We can see that increasing the number of triplets slightly increases the accuracy. However, this also proportionately increases the training time. EMBLOOKUP can be trained in under 1 hour using a V100 GPU for 100 triplets per entity. It requires around 1.8 hours for 200 triplets per entity, and rises to 9.2 hours for 1000 triplets per entity.

Impact of Compression on Recall. By default, we use product quantization (PQ) which is an approximate nearest neighbor search method. The applications studied in the paper (CEA, CTA, entity disambiguation, and data repair) are resilient to this approximation as they retrieve as much as 20-100 most similar entities and perform customized post-processing. However, it is known that the retrieval accuracy of PQ is reduced when retrieving a smaller number of neighbors. We investigate this phenomenon in this experiment. Specifically, we focus on the CEA task. We observed similar trends for other tasks. We consider two variants of EMBLOOKUP – with and without embedding compression. Since we study the impact of compression, we treat EMBLOOKUP without compression as the ground truth. Recall from Tables II and III that EMBLOOKUP without compression achieved an F-score that is almost identical to that of the original system. Ideally, EMBLOOKUP with compression should retrieve the same set of entities. For example, if EMBLOOKUP with compression retrieves 80% of the entities retrieved by EMBLOOKUP without compression, we compute the recall as 0.8. We measure the recall for different values of k. Figure 4 shows the results. While the average recall is relatively low for small values of k, the performance recovers for larger values of k.

Alternate Compression Schemes. By default, we use product quantization for compressing the embeddings. A natural alternative, which we investigate in this experiment, is to use techniques based on dimensionality reduction. We choose PCA as the representative dimensionality reduction algorithm. Recall that the uncompressed embedding for a KG entity has 64 dimensions requiring 256 bytes of storage. By default, we reduce the storage requirement to 8 bytes through quantization. In our experiment, we vary the size of the "compressed" vector from 256 bytes to 8 bytes (which corresponds to 64 dimensions to 2 dimensions, respectively, for PCA as each dimension requires 4 bytes of storage). This fixes the space requirements for both the approaches allowing us to compare the impact of compression on accuracy. Figure 5 shows the results for two tasks – CTA and CEA. For both cases, we modify the bbw system as it provided the best results. The PQ and PCA suffixes illustrate the EMBLOOKUP variants that used product quantization and dimensionality reduction for compression, respectively. The PQ lines are almost flat since, as we saw previously, product quantization does not result in significant loss of accuracy. On the other hand, PCA-based compression performs much worse.

V. RELATED WORK

Entity Lookup in KG. Lookup is a fundamental operation in a number of KG tasks. As KGs become larger and larger, it becomes challenging to store them locally. Hence, most of the existing systems rely on remote lookup services. Search in Wikidata is enabled by string-based search engines in the Wikidata API as well as the SPARQL-based Wikidata Query Service (WDQS). DBPedia lookup[6] is a generic entity retrieval service for RDF data. It can be configured to index any RDF data and provide retrieval services that resolve keywords to entity identifiers. The output of this process is usually a list of relevant entities ranked according to a similarity function (i.e., edit distance). Local lookup services often involve building indices based on the entity labels and other relevant properties. The most popular approach is to use a full-text search engine such as Elastic Search. Finally, one could also use existing text similarity libraries such as FuzzyWuzzy. Almost all of these approaches focus on edit-distance-based syntactic similarity and cannot support semantic similarity.

Embeddings for Strings. Popular approaches for word embedding such as fastText [2] and word2vec [18] are trained

[6]https://lookup.dbpedia.org

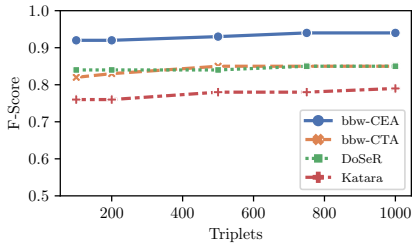

Fig. 3: Impact of training data (#triplets per entity).

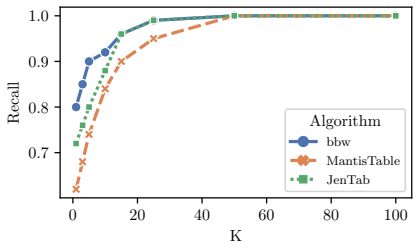

Fig. 4: Impact of compression on Recall@K.

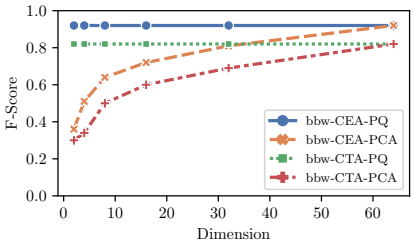

Fig. 5: Comparing the performance of Product Quantization and PCA.

on a large corpus of text and output a vector space where each word in the corpus is represented by a real-valued vector. These methods are not robust to misspellings and cannot be directly used for modeling the rich semantic similarities exhibited by KGs. There has been some recent work on learning string embeddings for specific tasks such as estimating edit distance [8] and string cardinality estimation [23]. However, these embeddings are designed for tasks such as query optimizations and could not be used for lookup.

Knowledge Graph Embedding Models. KG embedding models learn the latent representations of entities and relations that could be used for tasks such as link prediction, node classification, and graph completion. However, these embeddings cannot be directly used for entity lookups. As a matter of fact, a common use case for lookup is to identify relevant entities and then retrieve their entity embeddings from a KG embedding model for downstream tasks such as column type prediction. That is, using a KG embedding model requires separate support for lookup, such as that provided by EMBLOOKUP. Optimizing embedding techniques for similarity applications can be done by incorporating textual information in the embedding method. One class of work accomplishes this by optimizing an objective function that combines entity and text information to learn joint embeddings [26], [28]. These methods show improved performance over pure knowledge graph embeddings. However, they do not handle all forms of semantic similarity in case of misspellings or insufficient text, nor are they very efficient in searching large-scale KGs.

Similarity Search. There has been extensive work on efficiently retrieving similar entities based on their embeddings. A key challenge is the high dimensionality of these embeddings, which obviates the use of traditional relational and text databases. Since exact approaches are inefficient, the embeddings are often preprocessed to answer the queries approximately without scanning the entire database. An overview of optimizations can be found in [27], [29]. In this paper, we use FAISS [17], which supports additional functionalities such as quantization and GPU acceleration.

SemTab Challenge. This competition aims to benchmark semantic annotation systems that match tabular data with knowledge graphs. While SemTab 2019 [15] focused on matching with DBPedia, SemTab 2020 [16], [14] focused on Wikidata. The quality of the submissions is tied with the quality of the lookup functions. Almost all of the submissions rely on multiple lookup services and sophisticated lexical matching. However, none of the systems used are efficient and can handle both syntactic and semantic similarities. SemTab 2020 also introduced Tough Tables [7], which is a challenging dataset for semantic annotation with misspellings and ambiguous entity references. We used three representative and well-performing systems from the SemTab 2020 challenge – bbw [24], MantisTable [6], and JenTab [1] – to illustrate the effectiveness of EMBLOOKUP.

VI. CONCLUSION

We identify the entity lookup operation as an essential workhorse for diverse semantic table annotation tasks. The current implementations of lookup are not optimized, not robust to misspellings, ignore semantic similarities, and are often limited to a single similarity metric (such as edit distance). These limitations are a major bottleneck that impedes the widespread use of knowledge graphs in large data settings such as enterprise data lakes. In this paper, we made some promising progress in accelerating the lookup operation through our framework, EMBLOOKUP, which is efficient to train, requires compact storage, and can overcome a wide range of data errors. The embeddings learned by EMBLOOKUP are generic and encode syntactic and semantic aspects vital for lookup. Specifically, the Euclidean distance between two embeddings can capture multiple hidden syntactic and semantic distances. Our experiments showed that EMBLOOKUP can achieve 1-2 orders of magnitude speedup in multiple real-world applications while being robust to errors.

There are a number of interesting directions for future work such as evaluating other loss functions and DL architectures. Training over the most promising triplets through mining could provide additional speedup. This could allow achieving the same accuracy while training over a smaller number of triplets. Finally, an intriguing direction for future work is to bootstrap the embeddings for lookup from the corresponding KG embeddings that are optimized for semantic similarity and adapt them to handle syntactic similarity.

REFERENCES

[1] N. Abdelmageed and S. Schindler. Jentab: Matching tabular data to knowledge graphs. In *The 19th International Semantic Web Conference (ISWC)*, 2020.

[2] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov. Enriching word vectors with subword information. *TACL*, 5:135–146, 2017.

[3] G. Chechik, V. Sharma, U. Shalit, and S. Bengio. Large scale online learning of image similarity through ranking. *J. Machine Learning Research*, 2010.

[4] S. Chen, A. Karaoglu, C. Negreanu, T. Ma, J.-G. Yao, J. Williams, A. Gordon, and C.-Y. Lin. Linkingpark: An integrated approach for semantic table interpretation. *Semantic Web Challenge on Tabular Data to Knowledge Graph Matching (SemTab). CEUR-WS. org*, 2020.

[5] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. Katara: A data cleaning system powered by knowledge bases and crowdsourcing. In *ACM SIGMOD*, pages 1247–1261, 2015.

[6] M. Cremaschi, R. Avogadro, and D. Chieregato. Mantistable: an automatic approach for the semantic table interpretation. *SemTab@ ISWC*, 2019:15–24, 2019.

[7] V. Cutrona, F. Bianchi, E. Jiménez-Ruiz, and M. Palmonari. Tough tables: Carefully evaluating entity linking for tabular data. In *International Semantic Web Conference*, pages 328–343. Springer, 2020.

[8] X. Dai, X. Yan, K. Zhou, Y. Wang, H. Yang, and J. Cheng. Edit distance embedding using convolutional neural networks. *SIGIR*, 2020.

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SoCG*, pages 253–262, 2004.

[10] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *ACL*, 2019.

[11] V. Efthymiou, O. Hassanzadeh, M. Rodriguez-Muro, and V. Christophides. Matching web tables with knowledge base entities: from entity lookups to entity embeddings. In *International Semantic Web Conference*, pages 260–277. Springer, 2017.

[12] V.-P. Huynh, J. Liu, Y. Chabot, T. Labbé, P. Monnin, and R. Troncy. Dagobah: Enhanced scoring algorithms for scalable annotations of tabular data. *SemTab*, 2020.

[13] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE PAMI*, 33:117–128, 2010.

[14] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, and K. Srinivas. Semtab 2019: Resources to benchmark tabular data to knowledge graph matching systems. In *European Semantic Web Conference*, pages 514–530. Springer, 2020.

[15] E. Jiménez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, and K. Srinivas. Semtab 2019: Resources to benchmark tabular data to knowledge graph matching systems. In *European Semantic Web Conference*, pages 514–530. Springer, 2020.

[16] E. Jimenez-Ruiz, O. Hassanzadeh, V. Efthymiou, J. Chen, K. Srinivas, and V. Cutrona. Results of semtab 2020. In *CEUR*, volume 2775, pages 1–8, 2020.

[17] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with gpus. *IEEE Transactions on Big Data*, 2019.

[18] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[19] D. Ritze, O. Lehmberg, and C. Bizer. Matching html tables to dbpedia. In *WIMS*, pages 1–6, 2015.

[20] F. Schroff, D. Kalenichenko, and J. Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[21] SearX. Searx. https://searx.github.io/searx, 2021.

[22] SeatGeek. Fuzzywuzzy. https://github.com/seatgeek/fuzzywuzzy, 2021.

[23] S. Shetiya, S. Thirumuruganathan, N. Koudas, and G. Das. Astrid: accurate selectivity estimation for string predicates using deep learning. *PVLDB*, 14(4), 2020.

[24] R. Shigapov, P. Zumstein, J. Kamlah, L. Oberländer, J. Mechnich, and I. Schumm. bbw: Matching csv to wikidata via meta-lookup. In *CEUR Workshop Proceedings*, volume 2775, pages 17–26, 2020.

[25] SuperCowPowers. Data hacking. https://github.com/SuperCowPowers/data_hacking, 2021.

[26] K. Toutanova, D. Chen, P. Pantel, H. Poon, P. Choudhury, and M. Gamon. Representing text for joint embedding of text and knowledge bases. In *EMNLP*, pages 1499–1509, 2015.

[27] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

[28] Z. Wang, J. Zhang, J. Feng, and Z. Chen. Knowledge graph and text jointly embedding. In *EMNLP*, pages 1591–1601, 2014.

[29] P. Zezula, G. Amato, V. Dohnal, and M. Batko. *Similarity search: the metric space approach*, volume 32. Springer Science & Business Media, 2006.

[30] S. Zwicklbauer, C. Seifert, and M. Granitzer. Doser-a knowledge-base-agnostic framework for entity disambiguation using semantic embeddings. In *European semantic web conference*, pages 182–198. Springer, 2016.