Horizon 2020

Understanding Europe's Fashion Data Universe

Report on Text Joins

Deliverable number: D4.1

Version 2.0

Funded by the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 732328

Project Acronym:	FashionBrain
Project Full Title:	Understanding Europe's Fashion Data Universe
Call:	H2020-ICT-2016-1
Topic:	ICT-14-2016-2017, Big Data PPP: Cross-sectorial and cross-lingual data integration and experimentation
Project URL:	https://fashionbrain-project.eu

Deliverable type	Report (R)
Dissemination level	Public (PU)
Contractual Delivery Date	31 March 2018
Resubmission Delivery Date	27 February 2019
Number of pages	32, the last one being no. 26
Authors	Alexander Löser, Torsten Kilias, Benjamin Winter - BEUTH Martin Kersten, Ying Zhang - MDBS
Peer review	Alessandro Checco - USFD

Change Log

Version	Date	Status	Partner	Remarks
1.0	20/04/2018	Final	BEUTH, MDBS	Rejected 15/10/2018
2.0	27/02/2019	Resubmitted Final	BEUTH, MDBS	

Deliverable Description

This report will describe our methodology for learning text joins and a robust entity recognizer for the fashion domain.

Abstract

This report presents a new architecture, In-Database Entity Linking (IDEL), in which we integrate the analytics-optimized Database MonetDB with neural text mining abilities. This follows requirements from D.1.2 suggesting to have both, text and tabular data in the same database to permit market research departments to conduct analytic queries for fashion trends.

The most important task is how to recognize entities in a text and to link them to an existing database. Our system design abstracts core tasks of most neural entity linking systems specifically for MonetDB. We leverage the ability of MonetDB to support in-database analytics with user defined functions implemented in Python and these functions call machine learning libraries for neural text mining, such as TensorFlow. This type of integration removes data shipping and transformation overhead by utilizing MonetDB's ability to embed Python processes directly in the database kernel and exchange data with them. IDEL represents text and relational data in a joint vector space with neural embeddings and can compensate errors with ambiguous entity representations. For detecting matching entities, we propose a new similarity function based on joint neural embeddings which are learned via a specific ranking loss.

So far, no corpus exists that has human labeled entities in fashion texts linked to large tables with many densely populated attributes. Therefore we carefully explored potentially corpora that would mimic properties that would come close to a scenario in the fashion domain. Our choice is WebNLG which features many different entity types also used in the fashion domain, such as products, persons, groups, locations and organizations. Moreover, this corpus has sparse and dense populated tables as we would expect in the fashion domain as well. Our first implementation and experiments using the WebNLG corpus show the effectiveness and the potentials of this architecture.

Our system architecture uses in D4.1 our named entity recognition software TASTY. A more recent adoption, guided by recommendations of the EU, also lead to integrate FLAIR (published after this report by Zalando) in MonetDB, the named entity tagger of zResearch.

Table of Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

ANN	Artificial Neural Network
API	Application Programming Interface
BLOB	Binary Large Object
EL	Entity Linking
ELU	Exponential Linear Unit
IDEL	In-Database Entity Linking
INDREX	In-Database Relation Extraction
KNN	k-Nearest Neighbours
NLP	Natural Language Processing
RDBMS	Relational Database Management System
RDF	Resource Description Framework
RELU	Rectified Linear Unit
SQL	Structured Query Language
TAC-KBP	Text Analysis Conference Knowledge Base Population
UDF	User-Defined Function
WP	Work Package

1 Introduction

The FashionBrain project targets at consolidating and extending existing European technologies in the area of database management, data mining, machine learning, image processing, information retrieval, and crowdsourcing to strengthen the positions of European (fashion) retailers among their world-wide competitors.
Retailers, such as Zalando SE, often use a Relational Database Management System (RDBMS) to manage their product catalogue, customer data, sales information, etc. At the same time, there is also an abundance of text information available online, e.g. news messages, fashion articles, customers reviews and public fashion discussions. So, to enrich relational data in a (fashion) data warehouse with text data from the web is therefore an important operation for retailers to learn about their users, monitoring trends and predicting new brands. Therefore, in WP4 "In-Database-Mining and Deep Learning", we focus on developing deep learning empowered in-database text mining technology. To that end we develop a novel architecture, which tightly integrates with RDBMSs doing entity linking directly inside of the database.

We first outline the general architecture in section 2.1, and then discuss the relational neural embeddings and how they form a joint vector space in 2.2, and further implementation details in Section 2.3. We close with the results of our experiments in Section 2.4 and related works in Section 3

1.1 Scope of this Deliverable

In the context of T4.1 "Text joins between Fashion Entities in a RDBMS and Text", we have developed In-Database Entity Linking (IDEL), a novel architecture in which relational data, text data and entity linking are integrated into a single system. IDEL specifically targets the following business scenarios defined by deliverable D1.2

- Scenario 1, in particular challenge 1 "Mapping Search Intentions to Product Attributes", where linking of product attributes first requires a linking of the products themselves.
- Scenario 3, challenge 4 "FashionBrain Taxonomy and Product Taxonomy Linking" and challenge 5 "Linking Entities to Product Catalogue" which directly refer to the Entity Linking problem that is solved by this Deliverable 4.1 and Deliverable 4.2

IDEL leverages two best of breed systems MonetDB and In-Database Relation Extraction (INDREX) [1] for relational and text data analysis, respectively. Then, we

have developed a novel entity linking technology using neural embeddings for IDEL.

Scope and Dependencies This architecture builds on top of and integrates substantially with MonetDB, and as such with the results of Work Package WP2 and specifically task T2.3 and Deliverable D2.4. MonetDB represents the lower execution layer of the software packages implemented in this project and as a Core technology is referred to as CT2 "Infrastructures for scalable cross-domain data integration and management" In particular we make use of their Application Programming Interface (API) for custom user defined functions and the deep integration with python processes, NumPy and machine learning libraries such as TensorFlow that they provide. This results in an advanced in-database text-join system, which uses deep learning techniques to improve entity linking results.

The results of our work in T4.1 are reported in two deliverables. This deliverable, D4.1 "Report on Text Joins", contains the theoretical description of the method which is implemented and demonstrated in deliverable D4.2 and also experiments showing this architectures' effectiveness.

Dataset The deliverable and its task of entity linking requires a very specific type of dataset. Such a dataset would need to map a knowledge base, for example in the form of Resource Description Framework (RDF)-triples to corresponding entities in text. For the fashion domain this could for example be a knowledge base which describes different fashion products or brands and links those entries with news articles, blog posts etc. Unfortunately, to the best of our knowledge, no such dataset is currently available.
Regrettably the FashionBrain project does not compile a fashion themed entity linking dataset itself, as deliverable D3.3 is only defined to provide data for Work Packages WP5 and WP6 and not WP4, due to timeline issues. Therefore this deliverable and deliverable D4.2 by extension focus on implementing and validating a system that can be compared to the state-of-the-art, and is applicable to standard benchmark datasets that match the fashion domain as close as possible. We further plan to compile and build a new dataset in cooperation with USFD, which can then be used in the remainder of work package WP4 to test the reported techniques.
For this reason we have to rely on standard benchmark datasets for the entity linking task. We specifically chose the WebNLG dataset [20] for its idiosyncratic and structural similarities to fashion data, which makes transfer of our architecture to actual fashion data from news articles or blogs straightforward, as demonstrated in deliverable D4.2. WebNLG is a dataset containing 21,855 data and text pairs,

with the data consisting of 8,372 distinct knowledge base entries. these entries come from the broad, community-sourced DBPedia knowledge base. WebNLG offers data for 9 categories of entities. Out of these, the most relevant are Sports Team and University, which map well to any kind of Brand, Food and written works, which are products in essence, and building, which could map to individual stores or venues related to fashion events. Most importantly, WebNLG is quite a diverse dataset, which helps our architecture to generalize across different problems and corpora. We decide to use the 2017 challenge version, since that is the most commonly used benchmark.

The following example from WebNLG shows a candidate text for entities of the type 'building'. It shows for the entity five attributes in structured and text data. Note that some attributes are described over multiple sentences. Attribute names are highly ambiguous and do not match words in texts. Furthermore, the position of attributes in text varies.

```
<entry size="5" eid="Id1" category="Building">
 <modifiedtripleset>  <mtriple>103_Colmore_Row | floorCount | 23</mtriple>
  <mtriple>103_Colmore_Row | completionDate | 1976</mtriple>
  <mtriple>103_Colmore_Row | architect | John_Madin</mtriple>
  <mtriple>103_Colmore_Row | location |
                "Colmore Row, Birmingham, England" </mtriple>
  <mtriple>John_Madin | birthPlace | Birmingham</mtriple>
 </modifiedtripleset>
 <lex lid="Id1" comment="good">
  103 Colmore Row is located on Colmore Row, Birmingham,
  England. It was designed by the architect, John Madin,
  who was born in Birmingham. It has 23 floors and was
  completed in 1976.</lex>
</entry>
```

1.2 In Database Entity Linker (IDEL)

A particular exciting source for complementing relational data is text data. There are many opportunities for enterprises in many domains to gain advantages for their offerings or operations, for example gaining insights into upcoming trends, when the enterprise relational data can be linked to the abundance of text data from the web. The entities represented in the relational data can then be complemented, updated and extended with the information in the text data (which is often more fresh).

Entity linking between text and tables To realize such applications, a basic step is linking entities mentioned in text to entities represented in relational data, so that missing data in relational tables can be filled in or new data can be added. Figure 1.1 shows text data (i.e. Document) already preprocessed by some entity

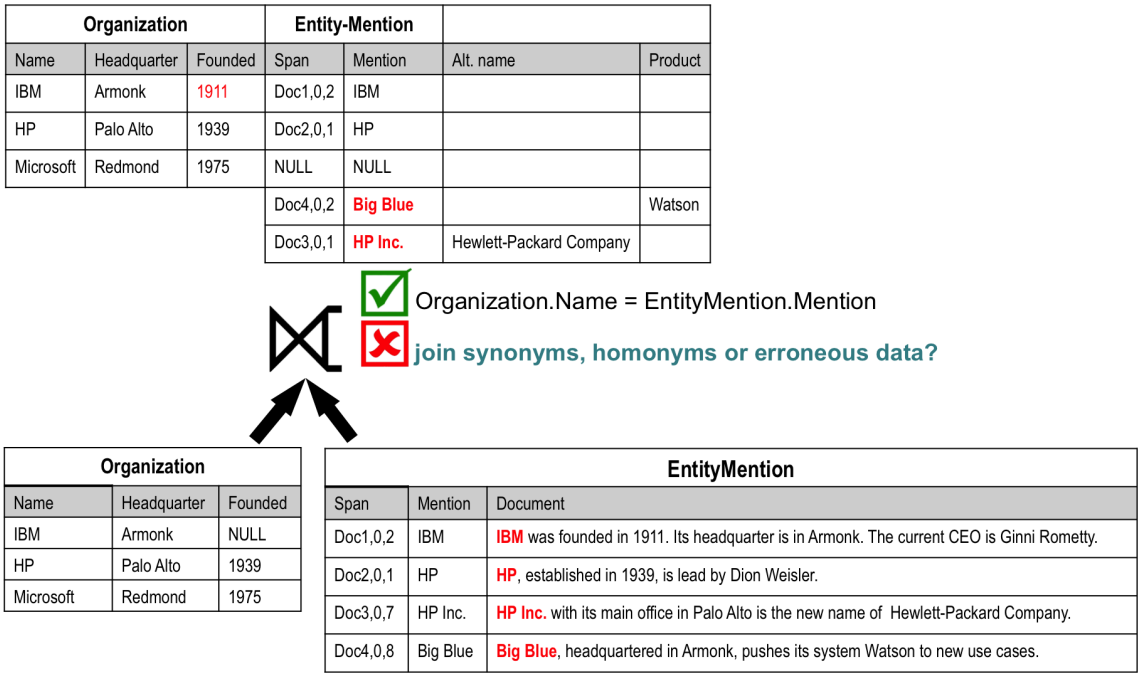


Figure 1.1: An example of joining relational data (i.e. Organization) with text data (i.e. EntityMention). An exact match strategy would have low recall, since it would not be able to match 'Big Blue' with 'IBM' or 'HP Inc.' with 'HP'.

recognizers [2], which annotate the text data with entities recognized (i.e. Mention) and their positions in the text (i.e. Span). An often practiced but limited solution to entity linking are linguistic-features based text join techniques [8, 14]. First, each entity in the relational data is represented as a character sequence; while a text join system is used to extract candidate *entity mentions* from the text data. Next, the text join system executes a cross join between the two sets of entities. Finally, users can apply some lexical filter conditions, e.g. an exact or a containment match, to reduce the result size. This practice often suffers from low recall and precision when faced with ambiguous entity type families and entity mentions. Typical error sources are matches between *homonyms* (e.g. IBM as the IATA airport code or the IT company), *hyponyms* (e.g. "SAP SE" vs. "SAP Deutschland SE & Co. KG"), *synonyms* (e.g. IBM and "Big Blue") and *misspellings in text* (e.g. "SAP Dtld. SE & Co. KG").

The need for Neural Entity Linking in RDBMS Entity Linking (EL) between text and a more structured representation has been an active research topic for many years in the web community and among computational linguists [22]. Systems come stand-alone or as separate tools, while so far, there has been little support inside RDBMSs for such advanced Natural Language Processing (NLP) tasks. Users (e.g. data scientists) often *have to use three systems*: one for relational data (RDBMS), one for text data (often Apache Lucene) and one for EL tasks (often homegrown).

The first problem comes with the choice of a proper EL-system. Although there are many research papers and a few prototypes for several domains available, most work comes with domain specific features, ontologies or dictionaries, and is often not directly applicable for linking data from a particular database or needs extensive fine tuning for a particular domain. To apply EL on relational data, a user has to first move the data from the RDBMS to the EL tool. This approach not only has many technical drawbacks, e.g. huge development effort, high maintenance, data provenance problems, bad scalability due to data conversion, transferring and storage costs, but also non-technical drawbacks, such as the difficulty of hiring people highly trained in both the RDBMS and the NLP worlds.

IDELs Innovations Ideally, an EL system should work without requiring domain specific feature engineering and costly data shipping, and can be triggered by simple Structured Query Language (SQL) queries. Therefore, we propose *IDEL*, a single system in which relational data, text data and entity linking tools are integrated. IDEL stores both relational and text data in MonetDB, the open-source RDBMS developed by partner MonetDB solutions, which is optimized for in-memory analytics [5]. Entity linking components are tightly integrated into the kernel of MonetDB through SQL User-Defined Functions (UDFs) implemented in Python [21]. In this way, machine learning libraries, e.g. TensorFlow, can be used to facilitate entity linking with neural embeddings. We chose neural embeddings so that the system learns 'features' from existing signals in relational and text data as hidden layers in a neural network and thus reduces human costs for feature engineering.

In IDEL, we choose the RDBMS as the basis of the architecture, and integrating text data and entity linking into it for several carefully considered reasons. First, while IDEL is generally applicable for text analysis applications, its primary target is enterprise applications, in which enterprise data is already stored in an RDBMS. Thus, an RDBMS based architecture has the biggest potential of a seamless adaptation. Second, an RDBMS has an extensive and powerful engine for pre- and post-entity-linking query processing and data analysis. Finally, in-database analytics (i.e. bring the computation as close as possible to the data instead of moving the data to the computation) has long been recognized as the way-to-go for big data analytics. Following the same philosophy, we propose an in-database entity linking architecture, which directly benefits from existing in-database analytics features. As a result, the following characteristics are realized in the IDEL architecture:

- **Best of two worlds** Users can seamlessly switch between SQL and Python, so that they can choose the best execution environment for each part of their data analytics.
- **Flexible and extensible** IDEL provides a set of pre-trained neural network models. In addition, it permits users to plug-in their own models or third-party

models for entity linking.

- **Simple user interface** IDEL provides an SQL-based user interface to all parts of the system. The whole workflow of entity linking can be executed by several calls to the implemented SQL UDFs. All intermediate and final results can be stored in the underlying database for further analysis.
- **Robust to language errors** IDEL adopts state-of-the-art neural embeddings for entity linking, which can achieve much higher precision under the four typical error sources (i.e. homonyms, hyponyms, synonyms and misspellings). In addition, the system leverages extra information from the relational data, such as attribute values, and integrity constraints on data type and range.
- **No manual feature engineering** IDEL does not require manual feature engineering; instead the system observes data distributions in the text database to represent best entities in relational and text data.

2 Method

2.1 IDEL Architecture

Figure 2.1 depicts the architecture of IDEL. We assume that relational data is already stored in the RDBMS according to its schema. In IDEL, we also store text data and neural embeddings in the same RDBMS. Text data is simply stored as a collection of strings, e.g. a table with a single string column. In this way, users can manage and query relational data together with text data and neural embeddings. Our approach for entity linking addresses *both mapping directions*, i.e. text to relational data and vice versa. The process of entity linking can be divided into four major steps:

Step 1: Vectorization. First, we compute the respective vector representations (i.e. Tuple_Vector $v_R(r)$ and Text_Vector $v_T(t)$) for the two data sets. Here we choose not to learn a neural network ourselves, but adopt a pre-trained model instead. From the machine learning community, there already exist well-trained networks that are known to be particularly suitable for this kind of work, e.g SkipThought [16]. Further, we can enrich both vector representations with additional discriminative features we can derive from their respective data sets. For tuple vectors, we can use additional constraints in the relational data, such as foreign keys. For text vectors, we can use context from surrounding sentences. This is further discussed in Sections 2.2.1 and 2.2.2.

Step 2: Finding matching candidates. The next step is to find matching candidates for entities in relational data with mentions in text data. Assume a user enters an SQL query such as the following to link relational and text data shown in Figure 1.1:

```
SELECT e.*, o.* FROM EntityMention e, Building b
WHERE LINK_CONTAINS(e.Mention, b.name, $ Strategy) = TRUE
```

This query joins EntityMention and tuples of building and evaluates if a name of a building is contained in the entity mentions. In addition, the function LINK_CONTAINS takes a third parameter $Strategy so that different strategies can be passed. So far, we support an exact match and, most important for this work, a semantic match strategy.

When computing the matches for the first time, there is generally very little knowledge about the data distribution. Therefore, we suggest bootstrapping an initial candidate pool. For example, one can generate exact matches with a join

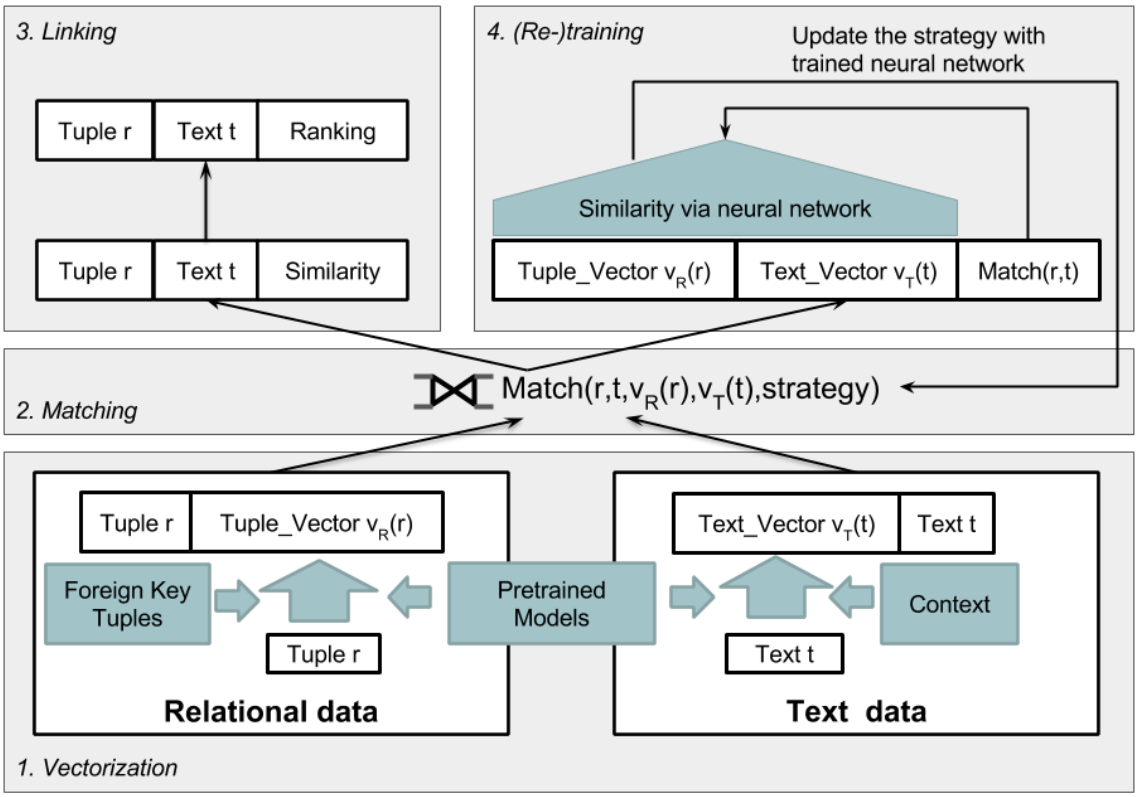


Figure 2.1: Architecture of IDEL with 4 steps in its workflow: vectorization, matching, linking and (re-)training.

between words in entity mentions and words in relational tuples describing those entities. This strategy is inspired by Snowball [1]. The initial matches can be used in later steps, such as linking and retraining. Other sources for matchings are gold standards with manually labeled data. However, this approach is highly costly and time consuming, because it requires expensive domain experts for the labeling and hundreds, or preferably even thousands of matchings.

Step 3: Linking. Now, we create linkings of matching entities. We interpret entity linking as a ranking task and assume that an entity mention in one data set is given and we try to find the k most likely entities in the other data set. This step uses the matching candidates found in step 2 and generates a ranking. If the matching function in step 2 returns similarity values, this steps will leverage that information to compute a ranking and select the top N best matchings for further use. In case the matching function, e.g. LINK_CONTAINS, does not produce similarity values (possibly due to the chosen strategy), all pairs of matching candidates are regarded to have equal similarity and hence will be included in the result of the linking.

Step 4: Retraining. The initial matching strategies mentioned in step 1 are often unable to detect difficult natural language features such as homonyms, hyponyms, synonyms and misspellings. To improve the initial results, IDEL contains a retraining step to enhance the neural models of the semantic similarity function. Thereby, IDEL updates previous models with retrained neural networks and recomputes the matching step. The training - updating - matching circle can also be used to incorporate changes in both relational and text data. If the changes alter the distributions of the data, the neural networks should be retrained with the new data so as to match the new entities reliably while using existing matching models for bootstrapping training data and reducing manual labeling efforts. In the next section, we will describe in details how training is done.

2.2 Embedding Models

Entity linking and deep learning Since very recently, entity linking techniques based on deep learning methods have started to gain more interests. The first reason is significantly improved performance on most standard data sets reported by TAC-KBP [12, 13]. Secondly, deep learning does not require costly feature engineering for each novel domain by human engineers. Rather, a system learns from domain specific raw data with high variance. Thirdly, deep learning based entity linking with character- and word-based embeddings often can further save language dependent costs for feature engineering. Finally, deep learning permits entity linking as a joint task of named entity recognition and entity linking [2] with complementary signals from images, tables and even documents in other languages [13]. These recent findings triggered a move of the entire community to work on entity-linking with deep learning.

Figure 2.2 gives an overview of our methods for representing and matching entities in a *joint embedding space*. It zooms in on the (re-)training step in the IDEL architecture (Figure 2.1). In this section, we first provide a formal description for our transformation of relational and text data into their respective vector representations. Next, we formalize a joint embedding space, in which similar pairs of entities in the relational data and their corresponding entity mentions are kept close to each other, while dissimilar pairs further apart. Then, we learn a common joint embedding space with a pairwise contrastive ranking loss function. Finally, in this joint space we compute a similarity between an embedding vector for relational and text data.

2.2.1 Relational Data Embeddings

Integrating relational signals in single entity embedding. The relational model features many rich signals for representing entities, such as relation and attribute names, attribute values, data types, and functional dependencies between values.

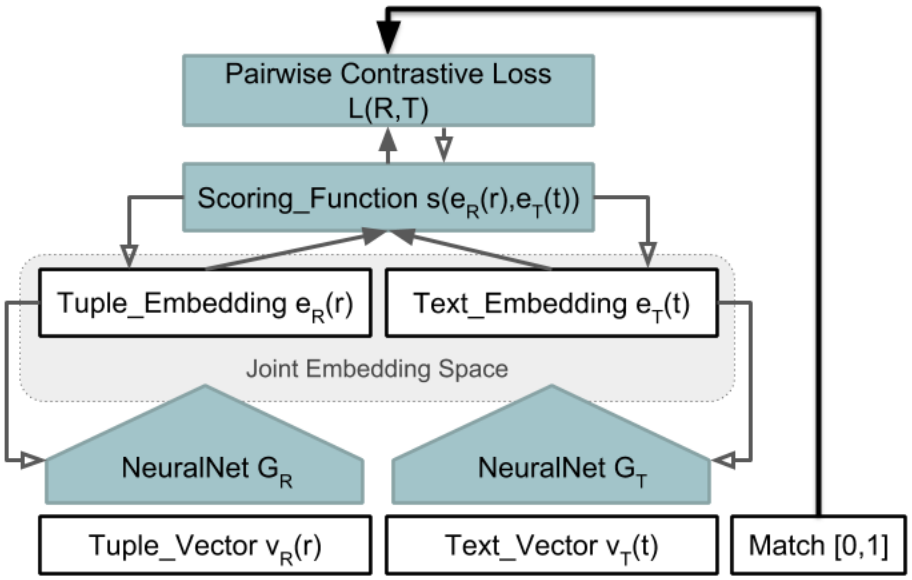


Figure 2.2: Overview of representing and matching entities in a joint embedding space in IDEL.

Moreover, some relations may have further inter-dependencies via foreign keys. These relation characteristics are important signals for recognizing entities. Our approach is to represent the "relational entity signatures" relevant to the same entity in a single entity embedding.

Vector representation of entity relations. To create embeddings we require a vector space representation. Therefore, we transform relations into vectors as follows:

Let $R(A_1, \ldots, A_n, FK_1, \ldots, FK_m)$ be a relation with attributes $A_1, \ldots, A_n$ and foreign keys $FK_1, \ldots, FK_m$ referring to relations $R_{FK_1}, \ldots, R_{FK_m}$. We define the domain of R as $dom(R) = dom(A_1) \times \ldots \times dom(A_n) \times dom(FK_1) \times \ldots \times dom(FK_m)$.

Embedding attribute data types. Another important clue is data type: we transform text data from alpha-numeric attribute values, such as CHAR, VARCHAR and TEXT, in neural embeddings represented by the function $text2vec : TEXT \rightarrow \mathbb{R}^m$; we normalize numerical attribute values, such as INTEGER and FLOAT, with their mean and variance with the function $norm : \mathbb{R} \rightarrow \mathbb{R}$; and we represent the remaining attributes from other data types as a one-hot encoding (also known as 1-of-k Scheme) [4]. Formally, $\forall a_i \in A_i$ we define a vector $v(a)$ of a as:

$$v_{A_i}(a_i) = \begin{cases} text2vec(a_i) & dom(A_i) \subseteq Text \\ norm(a_i) & dom(A_i) \subseteq Numbers \\ onehot(a_i, A_i) & else \end{cases}$$

Embedding foreign key relations. Foreign key relations are another rich source of signals for representing an entity. Analogous to embeddings of entity relations, we encode embeddings for these relations as well. We define $\forall fk_j \in FK_j$ the vector $v_{FK_j}(fk_j)$ of fk_j as the sum of the vector representations of all foreign key tuples $v_{R_{FK_j}}(r_{fk_j})$, where $r_{fk_j} \in R_{FK_j}$is a foreign tuple from R_{FK_j} with fk_j as primary key:

$$v_{FK_j}(fk_i) = \sum_{r_{fk_j} \in R_{FK_j}} v_{R_{FK_j}}(r_{fk_j})$$

Concatenating signature embeddings. Finally, we concatenate all individual embeddings, i.e. attribute embeddings and foreign key embeddings, into a single embedding for each entity, i.e. $\forall r = (a_1, \ldots, a_n, fk_1, \ldots, fk_m) \in R$, the vector $v_R(r)$ of tuple r is defined as:

$$v_R(r) = \begin{array}{c} v_{A_1}(a_1) \oplus \cdots \oplus v_{A_n}(a_n) \oplus \\ v_{FK_1}(fk_1) \oplus \cdots \oplus v_{FK_m}(fk_m) \end{array}$$

2.2.2 Text Embeddings

Representing entities in text as spans. Text databases, such as INDREX [14] and System-T [8], represent entities in text data as so-called *span data type*:

Given a relation $T(Span, Text, Text)$ which contains tuples $t = (span_{entity}, text_{entity}, text_{sentence})$ where $span_{entity} \in Span$ is the span of the entity, $text_{entity} \in Text$ is the covered text of the entity and $text_{sentence} \in Text$ is the covered text of the sentence containing the entity.

The above formalization covers the entity name, the context in the same sentence and long range-dependencies in the entire in-document context of the entity. Thereby it implements the notion of *distributional semantics* [10], a well-known concept in computational linguistics.

Vectorizing text spans and their context. Next, we need to vectorize spans and their context from above. We define the vectorization of text attributes of relations as a function *text2vec* which can be "anything" from a pre-trained sentence embedding or a trainable recurrent network. In our model, we choose the popular and well suited approach *SkipThought* [16] from the machine learning community. Our rationale is the following: First, SkipThought is based on unsupervised learning of a generic, distributed sentence encoder, hence there is no extra human effort necessary. Second, using the continuity of text in a document, SkipThought trains an encoder-decoder model that tries to reconstruct the surrounding sentences of an encoded passage. Finally, a SkipThought embedding introduces a semantic similarity for sentences. This can help with paraphrasing and

synonyms, a core problem in resolving entities between relational and text data. In our implementation, we use the pre-trained sentence embeddings from SkipThought.

2.2.3 Joint Embedding Space

After the vectorization, we compute transformations for the entity-mentions and relational data embeddings in a joint embedding space. In this space similar pairs of entities from relational data and entity-mentions from text are placed close to each other while dissimilar pairs far apart.

Let the first transformation $e_R : R \to \mathbb{R}^m$ to compute an embedding for a tuple $r \in R$, while the second transformation $e_T : T \to \mathbb{R}^m$ to compute an embedding for text $t \in R_T$. We define our transformations as follows:

$$e_R(r) = G_R(v_R(r), W_R) \qquad e_T(t) = G_T(v_T(t), W_T)$$

where G_R denotes a neural network with weights W_R for the transformation of relational data, and G_T an Artificial Neural Network (ANN) with weights W_T for the transformation of text data. Weights W_R and W_T are learnable parameters and will be trained with Stochastic Gradient Descent.

Depending on the vector representations used, G_R and G_T can be feed-forward, recurrent or convolutional neural networks, or any combination of them. In our implementation, we use feed-forward networks, because we transform the attribute values of the relational data and the text with existing neural network models into a common vector representation.

2.2.4 Pairwise Contrastive Loss Function

Scoring function. By nature, text and relational embeddings represent different areas in a vector space created from our feed forward networks. Therefore, we must define a scoring function to determine how similar or dissimilar two representations in this vector space are. We compare these two embeddings with a scoring function $s(e_R, e_T) : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}_{\geq 0}$, where small values denote high similarities while larger values dissimilar entities. Currently, we use the cosine distance as the scoring function $s(e_R, e_T)$, since our experiments with different distance measures, such as euclidean distance, show no notable effect on the accuracy of our results.

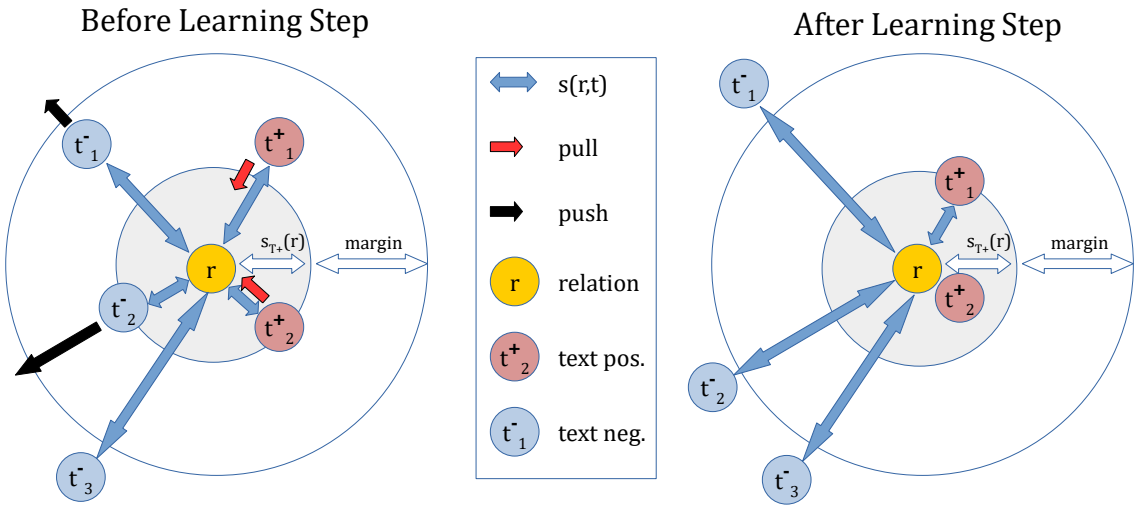


Figure 2.3: A learning step for a relation r with a pairwise contrastive loss function: before and after. Here, r is the center of both figures, surrounded by matching text examples t_1^+, t_2^+ and not matching (i.e. contrastive) text examples t_1^-, t_2^-, t_3^-. The inner circle represents the average score between r and matching text $s_{T_r^+}(r)$ from equation 2.5. The outer circle is the margin m. The loss function pulls matching text examples towards r and pushes contrastive examples towards the outer circle.

Loss function. To train relational and text embeddings, e_R and e_T, we use Stochastic Gradient Descent, which conducts a backwards propagation of errors. Our loss (i.e. error) function is a variation of the *pairwise contrastive loss* [15] applied first to the problem of mapping images and their captions into the same vector space. This loss function has desirable characteristics to solve our problem. First, it can be applied to classification problems with a very large set of classes, hence a very large space of 1000s of different entities. Further, it can predict classes for which no examples are available at training time, hence it will work well in scenarios with frequent updates without retraining the classifier. Finally, it is discriminative in the sense that it drives the system to make the right decision, but does not cause it to produce probability estimates that are difficult to understand when debugging the system. These properties make this loss function an ideal choice for our entity linking problem.

Applied to our entity linking task we consider either 1-N or M-N mappings between relations. We define our loss function as follows:

$$L(R,T) = \sum_{r \in R} L_R(r) + \sum_{t \in T} L_T(t) \tag{2.1}$$

with $L_R(r)$ as partial loss for r and $L_T(t)$ for t:

$$L_R(r) = \sum_{t^- \in T_r^-} max\{0, m + s_{T_r^+}(r) - s(e_R(r), e_T(t^-))\} \tag{2.2}$$

$$L_T(t) = \sum_{r^- \in R_t^-} max\{0, m + s_{R_t^+}(t) - s(e_T(t), e_R(r^-))\} \tag{2.3}$$

where T_r^- denotes the set of *contrastive* (i.e. not matching) examples and T_r^+ denotes the set of matching examples of T for r, such as respectively R_t^- and R_t^+ for t. The hyper-parameter *margin* m controls how far apart matching (positive) and not matching (negative) examples should be. Furthermore, the functions $s_{R_t^+}(t)$ and $s_{T_r^+}(r)$ calculate the average score of all positive examples for r and t:

$$s_{R_t^+}(t) = \frac{1}{|R_t^+|} \sum_{r^+ \in R_t^+} s\left(e_T(t), e_R\left(r^+\right)\right) \tag{2.4}$$

$$s_{T_r^+}(r) = \frac{1}{|T_r^+|} \sum_{t^+ \in T_r^+} s\left(e_R(r), e_T\left(t^+\right)\right) \tag{2.5}$$

Figure 2.3 shows a learning step of this loss function for a relation r. In equations 2.2 and 2.3, the addition of $s_{R_t^+}(t)$ and $s_{T_r^+}(r)$ pulls embedding vectors for positive examples together during the minimization of the loss function by decreasing their score. Conversely, the subtraction for a contrastive example of $s(e_T(t), e_R(r^-))$ and $s(e_R(r), e_T(t^-))$ pushes embedding vectors further apart, because increasing their score minimizes this subtraction. The margin limits the score for a contrastive example, so that the loss function cannot push the embedding vector of a contrastive example further. This is crucial to learn mappings between two different vector spaces.

Overall, we are not aware of any other work where loss functions for mapping pixels in images to characters are applied to the problem of linking entities from text to a table. IDEL is the first approach that abstracts this problem to entity linking. For our specific problem we therefore modified the loss function of [15] by replacing a single positive example with the average score of all positive examples $s_{R_t^+}(t)$ or $s_{T_r^+}(r)$. This pulls all positive examples together, enabling our loss function to learn 1-N and M-N mappings between relational data and text.

Hyper-parameters. We evaluate several configurations for representing neural networks with relational G_R or text data G_T. Our representation for relational data contains three layers: the input layer containing 1024 neurons, the second layer 512 neurons and the output layer 256 neurons. To represent text embeddings G_T we use two layers: an input layer with 1024 neurons and an output layer with 256 neurons. We choose fewer layer for G_T, because the dimensionality of their input is smaller

than that of the relational data. All layers use the activation function *Exponential Linear Unit (ELU)*, because it has the advantages of *Rectified Linear Unit (RELU)* and works for negative inputs, too. We train the model via gradient descent with Adam as optimizer and apply dropout layers to all layers with a keep probability of 0.75. Since we use dropouts, we can choose a higher learning rate of 1e−05 with an exponential decay of 0.9 every 1000 batches and set our margin to 0.001.

2.3 Implementation

Despite the fact that several technologies are investigated by the computational linguistics or the machine learning community, no other database system currently permits the execution of neural text mining inside its kernel. One reason is the choice of the RDBMS, which has a significant impact on the overall system. This section describes the ability of MonetDB to support in-database-analytics through SQL Python UDFs and to solve the overall entity-linkage task.

We integrate the entity linking process into one single RDBMS, MonetDB, as depicted in Figure 2.4, and store text, relational data and embeddings in MonetDB. The computation is either implemented in SQL queries, SQL UDFs or in Python. We briefly describe this integration in this section. works; then we describe the system architecture of IDEL; finally we detail the SQL UDFs used in each step of the entity linking process.

2.3.1 MonetDB/Python/TensorFlow/Annoy

We have integrated the entity linking process into a single RDBMS, MonetDB, as depicted in Figure 2.4. All data, i.e. text, relational data and embeddings, is stored in MonetDB. The computation is either done by vanilla SQL queries or SQL UDFs in Python.

MonetDB is the open-source columnar RDBMS optimized for in-memory processing of analytical workloads [5] of FashionBrain partner MonetDB Solutions. In recent years, we enriched its support for in-database analytics by, among others, introducing MonetDB/Python integration through SQL UDFs [21]. As a result, MONETDB users can specify Python as the implementation language for their SQL UDFs. Basically, any Python libraries accessible by the MonetDB server can be imported. In our work we base on the deep learning library TensorFlow[1] and the nearest neighbor search index for neural embeddings, Spotify Annoy [2]. When such an *SQL Python UDF* is called in an SQL query, MonetDB automatically starts a Python subprocess and executes it.

MonetDB exchanges data of relational tables between the SQL engine and the

[1] https://github.com/tensorflow
[2] https://github.com/spotify/annoy

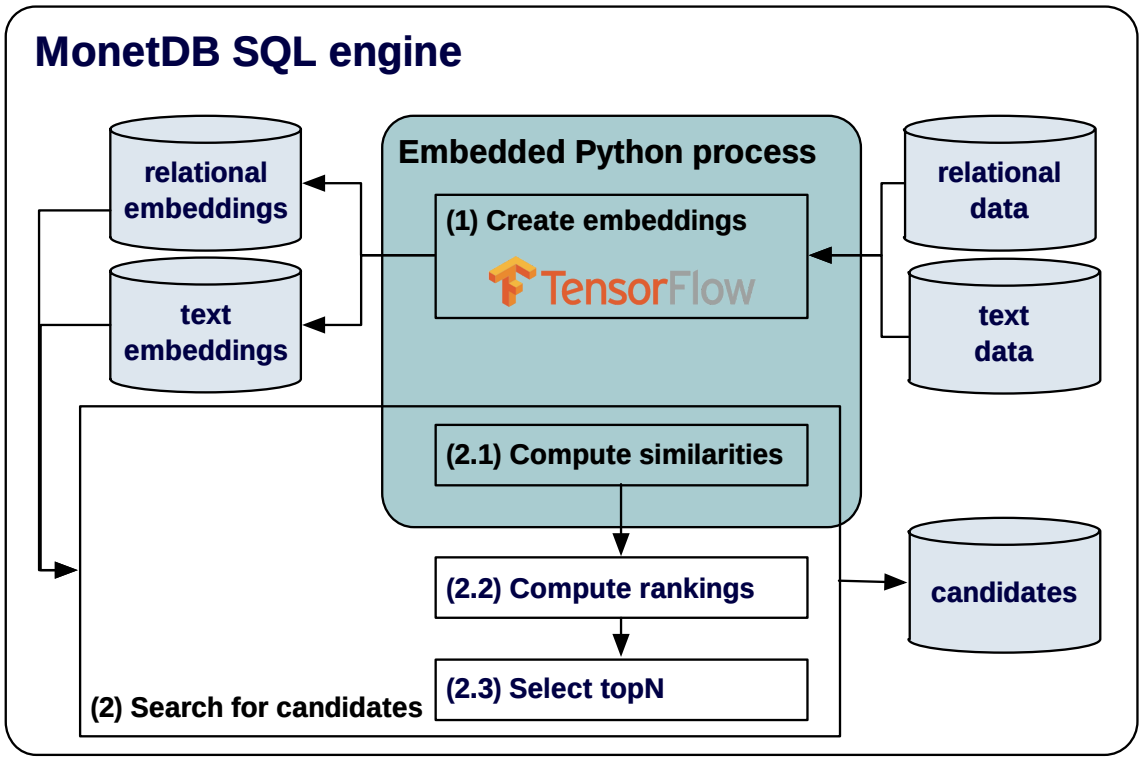


Figure 2.4: System architecture of IDEL: all data and computation are integrated in MonetDB.

embedded Python process by means of NumPy arrays. The MonetDB/Python integration features several important optimizations to allow efficient executions of SQL Python UDFs. *Zero data transfer and version cost*: internally, MonetDB stores data of each column as a C-array, the same as NumPy arrays. Hence, by exchanging data between the SQL engine and the Python subprocess by means of NumPy arrays, it incurs zero conversion and transfer cost. *Parallel execution of SQL Python UDFs:* an SQL Python UDF can declared as parallelizable, so that MonetDB automatically generates parallel execution plan for this UDF.

Figure 2.4 shows the implementation of IDEL in MonetDB. We store relational data in MonetDB according to their schemas and text data in a table with a single string-typed column. First, we create embedding vectors for both relational and text data by two SQL Python UDFs, one for each input table. This step leverages TensorFlow's machine learning features to load the pre-trained neural network and apply it on the input tables. We return embedding vectors as NumPy arrays and store them as BLOBs. The second step finds matching candidates with the highest similarities among embeddings. We employ nearest neighbor search with Annoy for a given embedding, compute a ranked list for each entity according to their similarities and finally return TopN candidates. All steps are implemented in SQL.

The resulting candidates list can be stored in the database for post-entity-linking

analysis.

This system architecture is straightforward but flexible and extensible. First, in step 1, we can easily replace one pre-trained neural network with another. Moreover, we can as easily extend this step to train our own neural network, which is part of our future work. Similarly, in step 2.1, we can choose different similarity functions depending on, for instance, their quality versus their computational complexity, and the characteristics of the input data set. Second, although we have currently decided to split step 2 into three substeps, we can flexibly decide to change it into more or less steps, and decide which steps to do in Python and which ones in SQL. Decisions made here are mainly determined by trying to not break the parallel execution flow in both MonetDB and Python.

Changes inside this architecture, such as different embedding models or similarity functions, are transparent to upper layer applications. The main differences that are noticeable to the applications are the execution time of the entity linking process and the final results

2.3.2 Create Embeddings

We abstract core functionalities for entity linking as SQL UDFs. This design principle permits us exchanging functionalities in UDFs for different data sets or domains.

UDF: EmbedSentences This UDF family embeds text sentences into the joint vector space. They apply v_T and G_T from a trained model to generate e_T. Because of the learned joint vector space, the function G_T is coupled on G_R which computes the tuple embedding for each table to which we want to link. They take as input a NumPy array of strings, loads a trained neural network model into main memory and apply this model to the NumPy array in parallel. TensorFlow allows these UDFs to easily leverage GPUs to compute the embedding vectors. Finally, these functions transform an array of embedding vectors into an array of Binary Large Objects (BLOBs) and returns it to MonetDB. The following example executes the UDF *embed_sentence_for_building* to retrieve sentences about buildings and return their embeddings.

```
CREATE FUNCTION embed_sentences_building(sentences STRING)
RETURNS BLOB LANGUAGE PYTHON
{
 from monetdb_wrapper.embed_udf import embed_udf
 return embed_udf().run("path/to/repo","path/to/model",
                "sentences", {"sentences": sentences })
}

CREATE TABLE embedd_sentences_for_building AS
 SELECT *, embed_sentence_for_building(sentence)
    AS embedding FROM sentences;
```

UDF: EmbedTuples This UDF family embeds tuples of a table into the joint vector space. They apply a trained neural network model on v_R and G_R to generate e_R. As input, they assume arrays of relational columns, and load and apply a trained model in parallel to input relations and outputs embedding vectors as an array of BLOBs. The exact signature of this UDF family depends on the schema of the table. In the following example, we encode the table *building* with attributes name, address and owner in the embedding:

```
CREATE TABLE building_with_embedding AS
 SELECT *, embed_building(name, address, owner)
 AS embedding FROM building;
```

2.3.3 Search for Candidates

UDF: QueryNN The next task is, given a vector (e.g. an entity represented in text), to retrieve a set of similar vectors (e.g. tuples representing this entity in a table). To avoid expensive cross join of all vectors of relational and text data, we represent embeddings for entities in a nearest neighbor search index for neural embeddings. Following benchmarks of [3], we implemented this index with Spotify Annoy because: i) Annoy is almost as fast as the fastest libraries in the benchmarks, ii) it has the ability to use static files as indexes which we can share across processes, iii) Annoy decouples creating indexes from loading them and we can create indexes as files and map them into memory quickly, and iv) Annoy has a Python wrapper and a fast C++ kernel, and thus fits nicely into our implementation.

The index bases on random projections to build up a tree. At every intermediate node in the tree, a random hyperplane is chosen, which divides the space into two subspaces. This hyperplane is chosen by sampling two points from the subset and taking the hyperplane equidistant from them. Annoy applies this technique t times to create a forest of trees. Hereby, the parameter t balances between precision and performance, see also work on Local sensitive hashing (LSH) by [7]. During search, Spotify Annoy traverses the trees and collects k candidates per tree. Afterwards, all candidate lists are merged and the TopN are selected. We follow experiments of [3] for news data sets and choose $t = 200$ for $k = 400000$ neighbors for $N = 10$.

The example below executes a k-Nearest Neighbours (KNN) search for an embedding representing relational tuples of table Building in the space of indexed sentences that represent an entity of the type Building. The query returns the top 10 matching sentences for this relational entity.

```
SELECT *
FROM query_index((
       SELECT id, embedding, 10, 400000,
       index_embedd_sentence_for_building
       FROM embedd_building)) knn,
     building_with_embedding r,
     sentences_for_building_with_embedding s
WHERE r.id = knn.query_key AND s.id = knn.result_key;
```

2.4 Experimental Evaluation

Training, test and cold start scenario In realistic situations, new entities are regularly added to the database. Hence, it is important for our system to recognize such *cold start entity representations* without needing to be re-trained. Hence, we need to consider previously seen entities for which we learn new matchings (hot and running system) and entities we have never seen during training (cold start scenario). To simulate these two scenarios we choose the same setup as described in [9] and split the set of relational entity instances into 20% *unseen* entities for the cold start scenario. We kept the remaining 80% as previously seen entities and split this set again into 80% for training and 20% for testing.

2.4.1 Experimental Setup

System setup We implemented our model using TensorFlow 1.3, NumPy 1.13 and integrated it into MonetDB (release Jul2017-SP1). We installed these software packages on a machine with two Intel® Xeon® CPUs E5-2630 v3 with 2.40GHz, 64 GB RAM, SSD discs and 1 Nvidia K80 GPU. The GPU was used for training the neural networks and computing the embeddings during the runtime measurements.

Measurements Our first set of experiments measures the effectiveness of our embeddings and entity linking. Given an entity representation from the relational data, the output of entity linking is an ordered list of sentences where this entity likely appears. A common measure is *Precision@1*, which counts how often the system returns the correct sentence at rank 1. Analogously, *Precision@5* and *Precision@10* count how often the correct result returned by the system is among the first five and ten results. Our second set of experiments measures the efficiency. We measure execution times for loading and creating embeddings before query run time and for generating candidates, executing the similarity measure on candidates and ranking candidates at runtime.

2.4.2 Experimental Results

Entity Linking with very high precision Table 2.1 shows accuracy for Precision@k for each entity type. We observe high values (≥ 0.80) during testing Precision@1 for all entity types, except City (0.73) and ComicsCharacter (0.76). This indicates that our system can return the correct entity linking with very high precision. If we measure the effectiveness of our system at Precision@5, we observe that our system returns the correct result for each entity, independent of the type, and with a high accuracy of ≥ 0.93. We report an accuracy of ≥ 0.95 at Precision@10 for all entity types with a perfect result for entity type university. Note that the columns *train* denote "ideal systems" for the given training data. We observe that even an ideal system fails in rare cases for Astronaut, WrittenWork, City, Food and Airport.

	Prec@1			Prec@5			Prec@10		
	te	tr	c	te	tr	c	te	tr	c
Airport	0.90	0.98	0.54	0.96	0.99	0.70	0.99	1	0.79
Astronaut	0.91	0.96	0.88	0.97	0.99	0.98	0.98	1	0.98
Building	0.89	0.99	0.77	0.94	1	0.90	0.97	1	0.95
City	0.73	0.98	0.93	0.93	1	0.98	0.96	1	1
ComicsCharacter	0.76	0.99	0.29	0.97	1	0.80	0.98	1	0.97
Food	0.85	0.94	0.69	0.94	0.98	0.90	0.94	0.98	0.91
Monument	0.94	1	0.90	0.98	1	0.98	1	1	1
SportTeam	0.90	1	0.66	0.97	1	0.83	0.99	1	0.92
University	0.95	1	0.93	0.99	1	1	1	1	1
WrittenWork	0.88	0.95	0.63	0.97	0.99	0.79	0.99	0.99	0.86

Table 2.1: Accuracy in Precision@k of the trained model for each *Entity type* (te - test, tr -train, c - cold start).

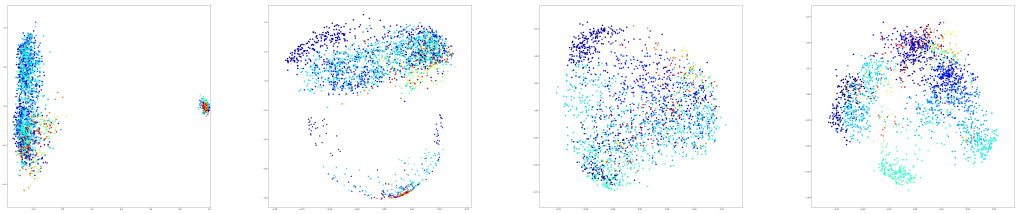
Phase	Step	Runtime (sec)
Load Time	Loading Model	30.50
Load Time	UDF:EmbedTuples	55.00
Load Time	UDF:EmbedSentences	150.00
Load Time	Create Index for Tuples	0.04
Load Time	Create Index for Sentences	3.07
Load Time	Sum over all steps	208.1
Query Time	Cross Join Top10	115.9
Query Time	UDF:QueryNN Top10 Sent.	9.60
Query Time	UDF:QueryNN Top10 Tuples	29.15

Table 2.2: Runtime of different stages of IDEL.

Execution Engine Table 2.2 reports execution times averaged over all entity types. We observe for steps at data loading time, such as embed sentences and relational tuples, an average of 208 seconds. For the query execution time and creating candidate tuples, storing embeddings, applying the similarity metric, ranking and pruning TopK entity mappings, we observe an average of 116 seconds. Our conclusion is that once a user has set up in IDEL an initial query mapping from entity types in relational data to sentences in text data, the system can asynchronous rebuild embeddings in the background to achieve very high Precision@1 values even for unseen entities for the next time the user hits the same query.

2.4.3 Error Analysis and Discussion

Understanding sampling and computing similarity function To understand the behavior of IDEL we conducted a closer inspection on results and individual components. Figure 2.5 shows four snapshots from the joint embedding space in IDEL during the training of the similarity function. For example, Figure 2.5(a) visualizes on the right a cluster of 58 different entities of the type *building* in 380 tuples, while the left cluster denotes 2377 sentences mentioning these entities. Colors



(a) Initial **(b)** 100 Batches **(c)** 200 Batches **(d)** 1200 Batches

Figure 2.5: The vector space of text and relational embeddings during the training period at varying batch samples.

indicate distinct entities[3]. Figure 2.5(b)..(d) show how during training, the shape of these clusters change. Finally, Figure 2.5(d) shows clusters which combine sentence and relational representations for the same entity. However, we also observe "yellow" and "red" entities with fewer training examples compared to the "blue" and "light blue" entities. Our explanation is that the contrastive pairwise loss function does not have sufficient "signals" gained from training samples yet to cluster these entities as well.

Performance for unseen entities suffers from sparse attribute density or too few sentences IDEL can recognize unseen data with a decent Precision@1>0.6, except for Airport and ComicsCharacter. This performance is comparable with other state-of-the-art entity linking systems (see [12]). The low performance for the type Airport is most probably due to the extreme sparse average tuple density. As a result, during training the model often retrieves relational tuples with low information gain and many NULL-values. A closer inspection reveals that several errors for this type are disambiguation errors for potential homonyms and undiscovered synonyms. The type ComicsCharacter also performs poorly for unseen entities compared to other types. This type has the second lowest ratio for Sentence/Instance. Hence, each distinct comic character is represented on average by 18 sentences. The popularity of text data, such as comic characters, often follows a Zipf distribution. In fact, we inspected our set of comic characters and observed that a few characters are described by the majority of sentences, while most characters are described by only a few sentences. As a result, the system could not learn enough variances to distinguish among these seldom mentioned characters.

[3]To keep the colors in the figures somewhat distinguishable, we show here only the most frequent entities, instead of all 58 of them.

3 Related Work

Text Databases Authors of Deep Dive [24], InstaRead [11] and System-T [8] propose declarative SQL-based query languages for integrating relational data with text data. Those RDBMS extensions leverage built-in query optimization, indexing and security techniques. They rely on explicitly modeled features for representing syntactic, semantic and lexical properties of an entity in the relational model. In this work we extend the text database system INDREX [14, 23] with the novel functionality of linking relational data to text data. Thereby, we introduce neural embeddings in a main memory database system, effectively eliminating the need for explicit feature modeling. Our execution system for INDREX is MonetDB as stated earlier. To our best knowledge, no other database system so far provides this functionality for text data.

Embeddings in Databases Only since recently, authors of [6] investigated methods for integrating vector space embeddings for ANNs in relational databases and query processing. Authors focus on latent information in text data and other types of data, e.g. numerical values, images and dates. For these data types they embed the latent information for each row in the same table with word2vec [19] in the same vector space. Finally, they run queries against this representation for retrieving similar rows. Similar to our work, they suggest to compute embeddings for each row and to access embeddings via UDFs. Our approach goes much further, since we embed latent information from at least two tables in the same vector space, one representing entities and attributes while the other representing spans of text data. Because of the nature of the problem, we can not assume that both representations provide similar characteristics in this vector space. Rather, we need to adopt complex techniques such as SkipThought and pair-wise loss functions to compute similarity measures.

Entity Linking and knowledge base completion Entity linking is a well-researched problem in computational linguistics[1]. Recently, embeddings have been proposed to jointly represent entities in text and knowledge graphs [25]. Authors of [18] use an embedding for relations and entities in the triple format based on the structures of graphs. However, they do not incorporate additional attributes for the entities into the embedding; also, they only learn an embedding for binary relations, not for n-ary relations. At a very high level, we also apply similar techniques for representing entities in embeddings. However, our approach is based on SkipThought and a pair wise loss function which works particularly well with

[1] See http://nlp.cs.rpi.edu/kbp/2017/elreading.html

many classes (each entity represents its own class) and for sparse data, two data characteristics often found in practical setups for relational databases. Moreover, our approach is not restricted to triple-based knowledge bases. We can learn an embedding for arbitrary n-ary relations and incorporate their attributes and related entities. Finally, we are not aware of any work that incorporates neural network based knowledge representation methods into the query processor of an RDBMS.

4 Conclusions

To our best knowledge, IDEL is the first working database system which permits executing SQL queries on neural embeddings representing both, text and tables, and in an RDBMS. We plan to investigate other powerful neural architectures, overcoming vocabulary limitations of the pre-trained SkipThought. We also will inspect hybrid models considering large external linguistic corpora but also very specific, potentially domain focused corpora from the text database to improve character embeddings. Finally, we will investigate deeper effects of other distance functions, such as the *word mover's distance* [17].

This report demonstrates a method on how entities can not only recognized, but also linked directly inside and RDBMS. This method works through the use of both MonetDB's contributions in the FashionBrain project and deep recurrent neural network architectures. We contribute a SQL and Python based join operator for text and data in a database, the fully trained model and this generally applicable method.
While we were unfortunately unable to train IDEL directly on a fashion dataset, this method already transfers well, as seen in D4.2 and will be straight forward to apply to a fashion themed entity linking dataset should one become available in the future.

Bibliography

[1] Eugene Agichtein et al. *Snowball*: extracting relations from large plain-text collections. In *ACM DL*, pages 85–94, 2000.

[2] Sebastian Arnold et al. TASTY: Interactive Entity Linking As-You-Type. In *COLING'16 Demos*, pages 111–115, 2016. 00000.

[3] Martin Aumüller et al. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *SISAP 2017*, 2017.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag New York, Inc., 2006. ISBN 0387310738.

[5] Peter A. Boncz et al. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[6] Rajesh Bordawekar et al. Using word embedding to enable semantic queries in relational databases. DEEM'17, pages 5:1–5:4. ACM, 2017.

[7] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *STOC 2002*, 2002.

[8] Laura Chiticariu et al. SystemT: An algebraic approach to declarative information extraction. ACL '10, 2010.

[9] Nitish Gupta et al. Entity linking via joint encoding of types, descriptions, and context. In *EMNLP 2017*, 2017.

[10] Z. S. Harris. Distributional Structure. *WORD*, 10(2-3):146–162, August 1954. ISSN 0043-7956, 2373-5112.

[11] Raphael Hoffmann, Luke Zettlemoyer, and Daniel S. Weld. Extreme Extraction: Only One Hour per Relation. *arXiv:1506.06418*, 2015.

[12] Heng Ji et al. Overview of tac-kbp2016 tri-lingual edl and its impact on end-to-end cold-start kbp. *TAC*, 2016.

[13] Heng Ji et al. Overview of tac-kbp2017 13 languages entity discovery and linking. *TAC*, 2017.

[14] Torsten Kilias et al. INDREX: In-Database Relation Extraction. *Information Systems*, 53:124–144, 2015.

[15] Ryan Kiros et al. Unifying Visual-Semantic Embeddings with Multimodal Neural Language Models. *arXiv:1411.2539 [cs]*, November 2014.

[16] Ryan Kiros et al. Skip-thought vectors. In *NIPS'15*, 2015.

[17] Matt J. Kusner et al. From word embeddings to document distances. In *ICML'15*, 2015.

[18] Yankai Lin et al. Learning entity and relation embeddings for knowledge graph completion. In *Proceedings of AAAI*, 2015.

[19] Tomas Mikolov et al. Distributed representations of words and phrases and their compositionality. In *NIPS'13*, 2013.

[20] Laura Perez-Beltrachini et al. Building RDF content for data-to-text generation. In *COLING 2016*, 2016.

[21] Mark Raasveldt and Hannes Mühleisen. Vectorized UDFs in Column-Stores. SSDBM '16. ACM, 2016. ISBN 978-1-4503-4215-5. doi: 10.1145/2949689.2949703. URL http://doi.acm.org/10.1145/2949689.2949703.

[22] Xiang Ren et al. Automatic entity recognition and typing in massive text data. In *SIGMOD'16*, 2016.

[23] Rudolf Schneider et al. Interactive Relation Extraction in Main Memory Database Systems. *COLING'16*, 2016.

[24] Jaeho Shin et al. Incremental Knowledge Base Construction Using DeepDive. *VLDB 2015*, 2015.

[25] Zhen Wang et al. Knowledge graph embedding by translating on hyperplanes. In *AAAI'14*. Citeseer, 2014.