Termite: A System for Tunneling Through Heterogeneous Data

Raul Castro Fernandez, Samuel Madden

ABSTRACT

Data-driven analysis is important in virtually every modern organization. Yet, most data is underutilized because it remains locked in silos inside of organizations; large organizations have thousands of databases, and billions of files that are not integrated together in a single, queryable repository. Despite 40+ years of continuous effort by the database community, data integration still remains an open challenge.

In this paper, we advocate a different approach: rather than trying to infer a common schema, we aim to find another common representation for diverse, heterogeneous data. Specifically, we argue for an embedding (i.e., a *vector space*) in which all entities, rows, columns, and paragraphs are represented as points. In the embedding, the distance between points indicates their degree of relatedness. We present Termite, a prototype we have built to *learn* the best embedding from the data. Because the best representation is learned, this allows Termite to avoid much of the human effort associated with traditional data integration tasks. On top of Termite, we have implemented a *Termite-Join* operator, which allows people to identify related concepts, even when these are stored in databases with different schemas and in unstructured data such as text files, webpages, etc. Finally, we show preliminary evaluation results of our prototype via a user study, and describe a list of future directions we have identified.

ACM Reference Format:
Raul Castro Fernandez, Samuel Madden. 2019. Termite: A System for Tunneling Through Heterogeneous Data. In *Proceedings of (CIDR'19)*. ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Data integration – combining diverse data sets, from different organizations or with heterogeneous schemas – has been a long standing challenge for the database community, which we continue to struggle with today. At the core of this challenge is the fact that modern relational query processors require data to be carefully organized into a uniform schema. In particular, for relational operators to provide meaningful results, different columns that reference the same concept in different data sets must use exactly the same values and syntax. Duplicates must be eliminated. Values must be normalized. Errors must be cleaned. Although these challenges have been a boon to academic researchers who have published hundreds of papers on each of these topics, they also mean that most data integration projects are hugely time consuming and expensive, and that many data sets that should be integrated never are due to the complexity of creating sufficiently uniform data for relational operations to produce well-defined answers.

In this paper, we advocate an alternative approach. Instead of insisting on clean data and a standardized schema, we argue that we should accept that many closely related data sets will never be fully integrated into a single relational system. Instead, we propose Termite, a"dirt-loving" database system that provides as much of the power of declarative querying as possible, but on top of these non-uniform data sets.

The desiderata for a dirt-loving database are clear:

- It should able to query structured but differently-schema'd tabular data, retrieving related rows from these different tables.
- It should be able to relate structured to unstructured data (i.e., text files), highlighting portions of the text files that are related to specific records in the structured data.

Termite supports these goals through a novel *Termite-Join* capability. Unlike a conventional relational query processor, where most joins are based on exact equality matches, in Termite, the join operation retrieves data that is in *close* proximity to some input query. That proximity indicates degree of relatedness, and it is measured as the distance between vectors of an embedding that represents all cells, rows, columns from relations as well as text from web pages, and emails, and other files. These *relational embeddings* are similar to the word embeddings used for modelling language in text processing [17, 20], but are specially constructed to work well for tabular datasets that are relational in nature.

The key advantage of the embedding representation is that because both structured and unstructured data are represented as points (vectors), understanding whether a tuple in a relation is related to a text file boils down to measuring the distance between their vector representation. The key challenge is to assign vectors to data in such a way that distance between data points in the embedding indicates data is indeed related.

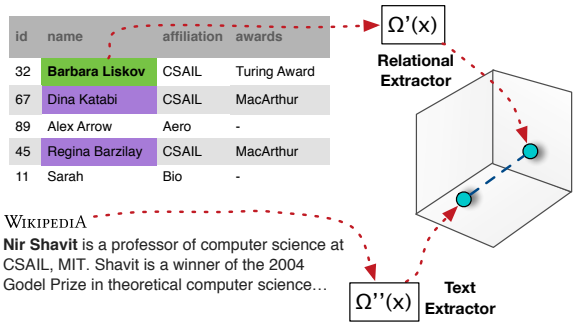

Figure 1: Distance in the embedding indicates relatedness

Consider the example of Fig. 1, which contains a relation and an excerpt from Wikipedia. Suppose we want to know what awards have been granted to professors working at CSAIL. If we only looked at the table, we would miss the Godel prize being awarded

CIDR'19, January 2019,

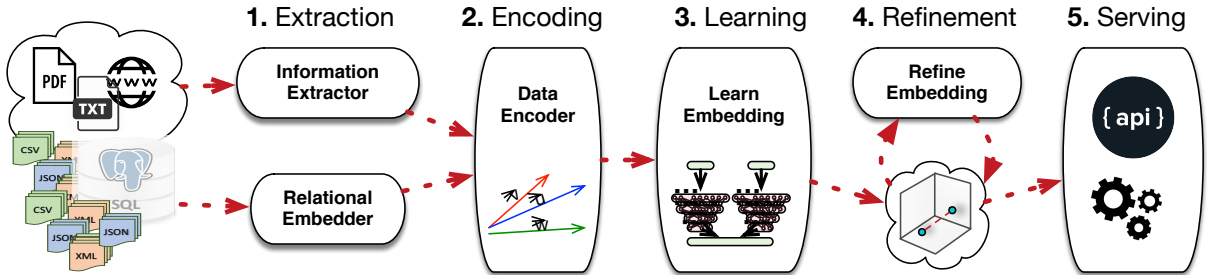

Figure 2: End-to-end overview of the 5 components of the Termite system

to Nir Shavit. In the embedding, the vector representation for *Godel prize* is close to the vector for *Turing award* because: i) it is an award, i.e., it is *granted*, or *won by* someone, and; ii) it is, in this case, awarded to someone who works at *CSAIL*. Note that a traditional TF-IDF based retrieval approach would not be able to identify this relationship. The *Termite-Join* operator relates these two together because they *share* relationships (granted/won award) and entities (CSAIL). With Termite, we can build an embedding on which the *Termite-Join* operator works without writing manual rules. Instead, Termite only needs pairs of elements that are related to each other; the pairs can be generated automatically from the sentence or tuple in which the elements appear.

Building the embedding. A principled way of building the embedding is to represent relational and unstructured data in some multi-dimensional tensor—which would be very sparse—where dimensions correspond to the different entities in the data, and then factorize the tensor to obtain a dense embedding that would contain information about how the entities are related to each other. This approach is so far only theoretical because we do not know how to precisely represent all data in a tensor form, or how to factorize it in such a way that the resulting embedding possesses the desired equivalence between vector distance and data relatedness. Instead, we propose to *learn* the embedding directly from the data. The central theme of this paper is *Termite*, a system to train and build the embedding, and an operator, *Termite-Join* to query the embedding to relate structured and unstructured data.

As an initial step towards Termite, we have built a proof of concept focused on helping with discovery problems. We conducted a user study to understand the benefits of the *Termite-Join* operator to discover data across structured and unstructured data such as MIT News, Wikipedia, personal webpages as well as relational data from the MIT datawarehouse and DBPedia. With Termite, users found faster more relevant content than with a baseline solution consisting of a full-text search index carefully built. We complement our evaluation with results on record linkage and concept expansion, two tasks closely related to the discovery problem.

We discuss the Termite's architecture rationale in section 2, the current learning pipeline in 3, followed by evaluation results (4), related work 5 and a brief discussion in 6.

2 TERMITE'S ARCHITECTURE RATIONALE

The idea of building an embedding with Termite was inspired by the impact of statistical language models such as word embeddings [17, 20] on the NLP, speech recognition and information retrieval communities. We quickly discovered that it is not straightforward to directly use these existing techniques, mainly because the assumption that all those models make—that words that appear often together are related to each other—does not translate to the set-oriented relational world of tuples, attributes, and tables, which carry much more structured information. Furthermore, it is not clear how to merge relational data with unstructured sources.

We have conceptualized the challenges faced by Termite into 5 loosely coupled components (Fig. 2). Each stage presents a number of research opportunities. Rather than exploring in depth each one of them, we decided to first build an end-to-end prototype so we can learn how the different stages are interconnected to each other. We describe these stages and our initial implementation below:

Extraction. The data extraction component converts raw relations and text into a set of bag-of-words (BoW) representation, e.g., one BoW per triple extracted from a sentence, or per cell value from a relation. To do that, it uses different *connectors*. For unstructured data, we use state-of-the-art information extraction platforms such as [10–12] to extract entity-relationship-entity triples. For relational data, a relational discovery tool guesses [6, 7] each relation's key and uses it along with the attribute values to produce triples, e.g., *John:value - age:attribute - 22:value*.

Encoding. The encoding component transforms each BoW into a vector. The vectors must be fixed-size so they can be used as the input to the learning component. A straightforward fixed-sized representation such as one-hot encoding has two big drawbacks. First, its dimensionality depends on the vocabulary size, which is large even for small datasets. Second, the vocabulary size must be known a-priori in order to size the vectors, which is inconvenient.

Our encoding component, instead, dictionary-encodes the vocabulary terms as integers, which are assigned incrementally as new words appear. These integers are indexed into a fixed-sized vector of length F using a hash function in $1 \ldots F$. We size the vector to minimize the number of collisions, which can be achieved using the birthday paradox and the expected number of words per BoW. Because collisions will occur anyway, we make a second attempt to insert the integer using a different hash function. With this encoding strategy we have seen performance improvements during learning of up to 2 orders of magnitude compared with one-hot encoding for a vocabulary size of 1M terms.

Learning and Refinement. These components are explained in detail in section 3. Here, we only mention that given the current extractor and encoder components, which produce triples of the

form *subject-predicate-object*, the training dataset is built by generating pairs from such triples: *subject-predicate*, *predicate-object* and *subject-object*. The pairs from the extracted triples are the positive pairs. Suppose we have positive pairs that always relate a professor to a phone number, and a phone number to an office. Even if we do not have an explicit pair relating the professor to the the office, both entities will appear closer to each other in the embedding: that's a key advantage of *joining* in the embedding.

To obtain negative samples we randomly assemble pairs that are not part of the positive training set, similar to the approach used in [3]. This makes it easy to generate negative pairs, but it introduces anomalies during the learning process i.e., unrelated points that end up close to each other not because they are related, but because they were not explicitly provided as negative pairs. The refinement component ameliorates some of the anomalies.

Serving. The serving component is Termite's raison d'etre. It makes the embedding available to answer database queries. Applications that traditionally take most of the time from analysts who need to perform them—and that are therefore not available to organizations without the luxury of dedicated analysts—become straightforward to perform if an embedding is available.

One example is data exploration, which we refer as the process of *visualizing* schemas to learn the content they represent, *summarizing* relations to get a glimpse of the information they convey, *understanding* how two relations are related to each other without going through the process of figuring out how to join them. Each of these tasks would take a long time to solve, but are really simple to solve in a vector space: 1) plot vectors in a reduced dimensionality to visualize the schema; 2) find a subset of diverse vectors to summarize a relation; 3) find vectors from the two relations we want to join that are close to each other in the embedding.

Another example is the task of discovering how data from relations and unstructured sources is related to each other. Useful for discovery, filling missing values and verifying information that appears in a table among others. This is the first application we have focused on and the reason for the *Termite-Join* operator we have implemented and focus on in the rest of this paper.

3 BUILDING A GOOD DATA EMBEDDING

We have explained how to transform a collection of text and relations into a collection of triples. Here, we explain how to turn the triples into a collection of vectors, (learning component of Fig. 2) in (3.1). We then explain how to refine the embedding (component 4 in the figure) and briefly the *Termite-Join* operator.

3.1 Obtaining a Basic Embedding

Methods such as [3, 16, 22] consume triples from a knowledge base to learn the latent variables that *explain* observable data, which helps among other tasks, with knowledge base completion. Our goal is different. We want to learn a distance metric to measure the relatedness of data coming from databases and text.

The entities we want to represent are the union of the sets of subjects, S, predicates, P, and objects O of the triples, $X = S \cup P \cup O$. We want to find a vector representation for each entity, $f(x_i)|x_i \in X$. In addition, given a distance function, $d()$, we want the vector representation of two related entities, $f(x_i)$ and $f(x_j)$, to be closer to each other according to $d()$ than to a third, unrelated vector $f(x_k)$. Finally, our training data consists of related and unrelated pairs. How can we find such vector representation $f()$?

Can't we just use Word Embeddings?. Word embeddings [17, 20] assume that words that appear often together are related to each other. Using very large text corpora—where words are used many times in different contexts—it is possible to learn a vector representation for each word, and measuring the distance between word vectors, to determine whether they are similar or not. The notion of similarity in word embeddings stems from the usage of words in the *same context*, e.g., *handsome* and *pretty* will be similar to each other because they are often used together in sentences. In our setting, we have the advantage of knowing precisely which entities are related to each other (the pairs): there is no need to infer this from their appearance together. However, we also have the disadvantage that entities won't occur many times in many different contexts. We need to find an alternative.

Vector assignment. Our proposal is to frame the assignment of vectors to entities in X as an optimization problem amenable to learning, so we can train it efficiently by feeding the pairs we have in the training dataset. In particular we want to train a deep network that, when given two input entities, x_i and x_j, assigns a vector to each of them, $f(x_i)$ and $f(x_j)$, computes their distance and predicts whether these two entities are close to each other. Unlike traditional machine learning models built for generalization, i.e., to predict output for unseen data, we are only truly interested in the representation $f()$ learned by the network, so we can use it to encode our data into the embedding. So, how do we train $f()$?

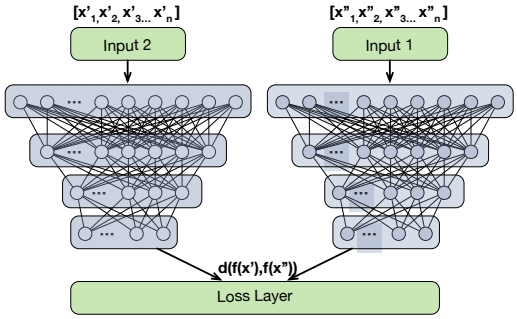

Figure 3: Architecture of the siamese network

Deep metric learning. We were inspired by the siamese networks used in [5] for identifying images of similar faces by using a metric learned by showing examples of similar faces, a task known as deep metric learning. We used a network such as the one in Fig. 3 to learn a metric for data. Once the network is trained and it has learned $f()$, we apply it to each element in X, obtaining the embedding representation of our data. And with that, we are back into database territory.

3.2 Refining the Embedding

We can use a repertoire of techniques from databases to store, index and query the embedding efficiently. We are exploring the best ways to manage and manipulate the embedding, but in this section we focus on how we can further improve the embedding quality.

The learned embedding will contain anomalies: vectors that are close together but are unrelated. This is because we only use a limited number of negative samples during training and because of the curse of dimensionality. If we leave the embedding untouched, we will produce wrong results when querying it. Next, we explain the technique we have implemented in the refinement component as well as ongoing work:

Curse of dimensionality. Since we are working with a high-dimensional embedding, we suffer from the curse of dimensionality [1]. The worst consequence is the phenomenon known as *hubness*, which is the tendency of certain points to be close to many other points. This means that certain data will be artificially related (close) to a lot of other data, which is directly against the quality metric we desire for our embedding.

The good news is that we can largely ameliorate this problem. The main intuition of our technique is that it is possible to compute a *hubness factor* for each entity represented in the embedding, and then remove entities with a high hubness factor in the top-k results. In particular, we compute how many times each point appears close to other points in the embedding. We then take the 75 percentile of the number of appearances as a cutoff parameter. At query time, we filter out those entities of the top-k results with a hubness factor higher than the cutoff parameter and pad the ranking with additional entities until we have K elements. Empirically, this improves the quality of the returned rankings.

Ongoing work: TDA. Can we learn more from the data once it's in a vector format? It seems intuitively interesting to understand the shapes the data forms in the learned embedding, and that may help us further refine the embedding itself. Whether two or more vectors are related—whether they have a shape—boils down to determining if they are within a specific distance, δ, of each other. Then, if they remain close as δ grows, it is possible to determine the strength of the relationship. Using the same intuition, vectors that do not remain close may be categorized as noise. Topological data analysis (TDA) is a mathematical tool that permits reasoning about shapes in high-dimensions algebraically. A central concept in TDA is persistent homology, which indicates which shapes remain in the high-dimensional embedding as δ changes. A straightforward application of persistent homology to our embedding gives us a degree of confidence for each of the results of the top-k list, depending on how *persistent* they are in the embedding. We are currently investigating additional applications of the technique, as well as how to best use TDA to curate the embedding.

3.3 Termite-Join Operator

Given an input entity, *Termite-Join* returns the K closest entities in the embedding. All entities representations in the embedding are computed offline by feeding the entities to the learned network and obtaining their representation.

At query time, given an input query x_i: 1) Obtain the embedding representation, $f(x_i)$ from the collection. 2) Retrieve the K-closest vectors to $f(x_i)$ from the collection. 3) Remove each $k_i \in K$ whose hubness factor is beyond the cutoff parameter computed by the refinement component. 4) Fill in the top-K list if some element has been removed in step 3. 5) Obtain the string representation of the top-k vectors and present the results to the user.

4 EARLY EXPERIENCE

We demonstrate now how we have used our proof of concept implementation of Termite to help users identify related data across heterogeneous schemas and unstructured data (section 4.1). Then, we show additional results on two microbenchmarks on record linkage and concept expansion, in 4.2.

Dataset and setup. We built a dataset with information of faculty at CSAIL. The dataset contains both structured data with different schemas, e.g., MIT datawarehouse and DBPedia, as well as unstructured data, e.g., Wikipedia pages, online news articles. We then used Termite to learn the embedding, using a laptop with 4 cores, 8GB RAM and without access to a GPU. The whole process took around 1 hour.

4.1 Data Discovery with Termite-Join

We did a user study to evaluate *Termite-Join*:

Study Goals. The goals of the study were to determine: i) whether the embedding is an appropriate abstraction to discover data across structured and unstructured data sources; and ii) whether the semantic distance learned is more appropriate for discovery tasks than a traditional full-text search interface based on TF-IDF relevance.

To answer these questions, we built two different interfaces to discover data. One of them, Full-Text-Search (FTS), receives all the data from the data extractor from Fig. 2 and indexes it in elasticsearch [9]. FTS has an API to perform keyword queries and find the matching documents from the system. The second interface, Termite-Search (TMT) is built on top of the embedding. It is similar to the first in that people can query with keywords, but those keywords are used as input to the *Termite-Join* operator. Our goal was to understand which interface was better for a set of discovery tasks we describe next.

Study Procedure. We recruited 8 users with a CS background and that are daily users of web search engines. We asked them to solve 3 tasks. The first is used as a training exercise, and the remaining two, (Task 1 and Task 2), are part of the experiment. We split the users in two groups of 4 people each, and showed a different interface to each group to avoid cross-learning effects, i.e., a person learning the results with one interface and reverse engineering the right query when using the second interface. We then measured the coverage of the results obtained by each group and asked for their feedback on both the questions and the interfaces they used.

We gave each user a 7 minute introduction to the corresponding interface, along with an example walk-through to illustrate the process. An experimenter was present at all times with the user, to clarify questions about the task goal, as well as to suggest ways of using the API when the user had doubts. We first explained the example task, without telling them it was an example. Users were asked to *create a list of forms of recognition (i.e., awards) that have been given to CSAIL faculty*. We asked the users to write their results in a text file, and explained that they could make as many search requests as they needed. We let them use the interface for 5 minutes (we did not stop them abruptly if they were engaged in preparing a query) and then moved on to the next two tasks: Task 1: *Create a list of contributions associated with CSAIL faculty*; and Task 2: *A list of institutions associated with CSAIL faculty*.

Table 1: Results from User Study

	FTS		TMT	
	(worst, avg)	Avg. Time	(worst, avg)	Avg. Time
Task 1	1%, 27%	6.1m	100%	4.2m
Task 2	9%, 18%	5.1m	18%, 43%	5.3m

Results. The users of TMT were far more successful than the users of FTS as the results of table 1 demonstrate. The table shows the percentage of results found by the users when using each interface, distinguishing between the worst result achieved by any of the 4 users using the same interface and the average one. Remarkably, for Task 1, the users that used TMT found a query that led to all results. In the case of Task 2, only one user found all of them. We measured the time the users took to perform each task and found that users of TMT took 2 min less on average than users of FTS. The times were similar, however, for Task 2.

When we asked the users for the difficulty of the different tasks they were solving, users of FTS were consistent in finding Task 1 harder than Task 2, and Task 2 harder than the example task. The users of TMT mentioned that all three tasks were similarly simple.

Examples. We show example results obtained by users of both interfaces in table 2 for both tasks. For Task 1, the users of TMT found algorithmic contributions, such as *LSH, Zero-knowledge proof* when inserting software artifacts such as *Vertica* or *Postgres*, see the second column of table 2. Users of FTS had to *guess* keywords that would indicate contributions, and this led naturally to a lower recall. Similar results were found in the case of Task 2, in which users of the FTS had to try different keywords such as *university*, *degree*, while users of TMT quickly identified that using a known example, e.g., *Harvard* led quickly to many other relevant results.

Conclusion. With TMT, users discover relevant information for their tasks from both unstructured and structured sources more efficiently and easier than with a full-text search index.

4.2 Record Linkage and Concept Expansion

Record Linkage [15] is about finding syntactically distinct records that refer to the same real-world entity, e.g., *Samuel R. Madden* and *Sam Madden*. Concept Expansion [26] is about obtaining instances of a given concept, e.g., given *Harvard*, obtain *MIT*, *Caltech*, *Stanford*, etc. Our hypothesis is that *Termite-Join* can help with these tasks. To understand this empirically, we retrieved ground truth for both tasks on the same CSAIL faculty dataset mentioned above, and then implemented functions on Termite to perform each task. We discuss the results next:

Record Linkage. Our dataset contains 52 faculty members. The minimum number of representations for each faculty member was 2, the average 4, and the maximum was 7. In total there are 210 different representations for the 52 faculty. To conduct this experiment we take one representation of each member and use it to query the embedding with the *Termite-Join* operator. We then measure how many of the found results are true alternative representations of the original query. We could identify 77% (163/210) different representations used to refer to CSAIL faculty.

In particular, the Termite identifies different spellings of faculty such as *David DeWitt* and *Dave DeWitt*, as well as those that include middle names, such as *David J. DeWitt*. More important, the embedding helped to identify entities that were not syntactically similar, such as *Liskov* and *Barbara Jane Huberman*. The first appears in the relational data, while the last name only appears in an unstructured source, but since both representations share relationships and entities they are placed closed to each other in the embedding.

Concept Expansion. For this experiment we compiled ground truth following a procedure similar to the one described in [26]. We found instances of the same concept for 10 different concepts, which are shown in the first column of table 3. The concepts had a number of instances that ranged from 10 to 80. Unlike the original definition of the concept expansion problem [26], which takes both a concept and an example instance as input, we only provide an example instance to the *Termite-Join* operator. We then run queries retrieving lists of different sizes, 2 and 4 times the size of the original (referred to as 2x Top and 4x Top in the table), and reporting the total percentage of values in the ranking result that were correct. The first query returns a list of size equal to the number of instances for the concept. The second and third query subsequently double the previous size.

Conclusions. The best news is that we obtained these results by only pointing out Termite to a data repository. No manual domain-specific engineering was necessary.

5 RELATED WORK

Automatic Knowledge Base Completion. RESCAL [16] models triples from a knowledge base via the pairwise interactions of latent features. Similarly, Structured Embeddings and subsequent work [3, 4, 22] learns embeddings for each relation from the triples. These approaches focus on learning the latent variables that describe the triples, to later fill in values of an incomplete knowledge base. In contrast, Termite learns a metric we use to relate structured and unstructured data based on their distance in the embedding.

Universal schema. [21, 25] decomposes a matrix in which rows represent entity pairs and columns represent relations between entities. Similar to our embedding, they show how to jointly embed text and knowledge bases. They are not focused on data discovery, but rather information extraction applications.

Word Embeddings and Relational Data. In [2], the authors propose a method to learn a vector representation of data items from relational data based on word embeddings [17], and then use those vectors to augment traditional SQL queries with so-called *cognitive* capabilities such as finding elements within a column or row that are similar to an input data item. We are different in that our embedding's goal is to find a relatedness metric for discovery applications beyond only relational data. Our embedding could be applied to extend SQL queries as well, which is interesting future work.

Other related techniques. There are a myriad of applications in NLP which share some characteristics with out goal of learning a good embedding of both structured and unstructured data. We can benefit from: i) sequence learning [13, 24], ii) alternative deep metric methods [14, 23]; iii) alternative embedding methods such as holographic [19] and hyperbolic embeddings [8]. A recent paper [18] explores deep learning for entity resolution, a common

Table 2: Example results by users of the study

Task 1: Contributions of CSAIL Faculty		Task 2: Associated Organizations of CSAIL Faculty	
FTS	**TMT**	**FTS**	**TMT**
arvind co founded company Robert_Tappan_Morris Y_Combinator_(company) stonebraker focused aurora hari balakrishnan commercializing research medusa/aurora project	Shor's_algorithm, karger algorithm Chord_(peer-to-peer), haystack project simultaneous multithreading Multics, VoltDB, Ingres_(database) Wait-free, Public-key Zeroknowledge_proof RSA_(algorithm) Alloy_(specification_language) MOOC **(aprox. 25 more results)**	morris delbarton school dewitt university michigan morris harvard university demaine phd university waterloo meyer phd harvard university	Princeton_University California_Institute_of_Technology University_of_Pennsylvania University_of_California _Berkeley Stanford_University Colgate_University Carnegie_Mellon_University massachusetts institute technology La_Sapienza_University_of_Rome University_of_Michigan IBM_Almaden_Research_Center Rice_University, harvard university **(aprox. 10 more results)**

Table 3: Results for concept expansion

Concept	# Instances	Found Top	Found 2x Top	Found 4x Top
known for	77	27 35%	46 59%	65 **84**%
faculty name	52	30 57%	46 88%	51 **98**%
Institutions	44	26 59%	33 75%	37 **84**%
birth place	35	14 40%	18 51%	24 68%
award	33	19 57%	24 72%	28 **84**%
academic children	54	41 75%	53 **98**%	—
field	21	13 61%	14 66%	16 76%
nationality	10	10 **100**%	—	—
doctoral advisor	31	12 38%	16 51%	23 74%
thesis title	19	8 42%	13 68%	16 **84**%

data integration task. Termite's focus is on leeting users operate on structured and unstructured data without a high upfront cost.

6 DISCUSSION

Termite helps with discovering related data across heterogeneous schemas and unstructured sources. While building Termite we identified several lines of future research. 1) How to build a relational embedding for data exploration. Exploring data is hard and requires many different operations, from visualizing datasets to summarize relations to understand how two schemas are connected to each other. 2) How to support more applications on the embedding, such as verification of relational data, filling values, etc. 3) To find better vector representations of data, where we are currently exploring new techniques for refining the embedding.

We think Termite is the first attempt towards a new tool (and a vector representation of data is a first step towards a new abstraction) for data management. Our vision is geared towards applications that comprise data beyond relations and use cases beyond precise relational query processing.

REFERENCES

[1] Charu C. Aggarwal, Alexander Hinneburg, et al. 2001. On the Surprising Behavior of Distance Metrics in High Dimensional Spaces. In *ICDT*.
[2] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *DEEM*.
[3] Antoine Bordes, Nicolas Usunier, et al. 2013. Translating Embeddings for Modeling Multi-relational Data. In *NIPS*.
[4] Antoine Bordes, Jason Weston, et al. 2011. Learning Structured Embeddings of Knowledge Bases.. In *AAAI*.
[5] Jane Bromley, Isabelle Guyon, et al. 1994. Signature verification using a "siamese" time delay neural network. In *NIPS*.
[6] Raul Castro Fernandez, Abedjan Ziawasch, et al. 2018. Aurum: A Data Discovery System. In *ICDE*.
[7] Zhimin Chen, Vivek Narasayya, et al. 2014. Fast Foreign-key Detection in Microsoft SQL Server PowerPivot for Excel. *VLDB* (2014).
[8] Bhuwan Dhingra, Christopher Shallue, et al. 2018. Embedding Text in Hyperbolic Spaces. In *TextGraphs-12*.
[9] elastic [n. d.]. ElasticSearch. http://www.elastic.com.
[10] J. R. Finkel, Grenager T., and C. Manning. 2005. Incorporating non-local information into information extraction systems by Gibbs sampling. *ACL* (2005).
[11] Angeli Gabor, Johnson Premkumar Melvin, et al. 2015. Leveraging Linguistic Structure For Open Domain Information Extraction. *ACL* (2015).
[12] Cunningham H., Bontcheva K., and Maynard D. 2002. GATE: an architecture for development of robust HLT applications. *ACL* (2002).
[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long Short-Term Memory. *Neural Comput.* (1997).
[14] Elad Hoffer and Nir Ailon. 2015. Deep metric learning using triplet network. In *International Workshop on Similarity-Based Pattern Recognition*. Springer, 84–92.
[15] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. 2006. Record Linkage: Similarity Measures and Algorithms. In *SIGMOD*.
[16] Denis Krompaß, Maximilian Nickel, et al. 2013. Non-negative tensor factorization with rescal. In *ECML workshop*.
[17] Tomas Mikolov, Ilya Sutskever, et al. 2013. Distributed Representations of Words and Phrases and Their Compositionality. In *NIPS*.
[18] Sidharth Mudgal, Han Li, et al. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*.
[19] Maximilian Nickel, Lorenzo Rosasco, et al. 2016. Holographic Embeddings of Knowledge Graphs. In *AAAI*.
[20] Jeffrey Pennington, Richard Socher, et al. 2014. GloVe: Global Vectors for Word Representation. In *EMNLP*.
[21] Sebastian Riedel, Limin Yao, et al. 2013. Relation extraction with matrix factorization and universal schemas. In *NAACL-HLT*.
[22] Richard Socher, Danqi Chen, et al. 2013. Reasoning with Neural Tensor Networks for Knowledge Base Completion. In *NIPS*.
[23] Kihyuk Sohn. 2016. Improved Deep Metric Learning with Multi-class N-pair Loss Objective. In *NIPS*.
[24] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. 2014. Sequence to Sequence Learning with Neural Networks. In *NIPS*.
[25] Patrick Verga and Andrew McCallum. 2016. Row-less Universal Schema. *AKBC* (2016).
[26] Chi Wang, Kaushik Chakrabarti, et al. 2015. Concept Expansion Using Web Tables. In *WWW*.