join2vec: Semantic-Rich and Efficient String Similarity Joins with Vector Embeddings

Manos Chatzakis
emmanouil.chatzakis@epfl.ch
EPFL
Lausanne, Switzerland

Viktor Sanca
viktor.sanca@epfl.ch
EPFL
Lausanne, Switzerland

Anastasia Ailamaki
anastasia.ailamaki@epfl.ch
EPFL, RAW Labs SA
Lausanne, Switzerland

ABSTRACT

This paper presents join2vec, a novel framework that efficiently addresses the problem of performing semantic rich and efficient string similarity joins using word embeddings, by exploiting the full computational capacity of modern servers comprised of multiple cores. Our framework is able to perform both exact and approximate joins by implementing different join algorithms, while it introduces several hardware-conscious optimizations, such as prefetching, SIMD usage and parallelization. Through a wide range of different experimental configurations, including datasets of different sizes, our experimental analysis demonstrates that join2vec is highly scalable and manages to achieve its challenging goals.

KEYWORDS

string similarity joins, embeddings, vector data management, semantic similarity

1 INTRODUCTION

Motivation. Performing string similarity joins is an indispensable task of real-world data analytics. It has applications across diverse scientific fields, as it is used in entity resolution, semantic data integration, data cleaning, anomaly detection, information retrieval and others ([8], [6], [40], [3], [24]). Modern approaches to the topic managed to provide efficient algorithms for string similarity joins with a wide variety of metrics ([35], [4], [34]), but unfortunately they are applicable for scenarios where we are interested in syntactic string similarity. Moreover, approaches that aim to incorporate semantic similarity in such joins require preconfigured sets of similarity rules and predefined taxonomy knowledge [41]. Incorporating deep learning words embeddings for string similarity joins is a promising direction, as there are multiple embedding models that provide interesting results for string similarity, managing to capture multiple different types of similar strings (syntactic, semantic and taxonomy) ([1], [2], [36]).

In this work, we combine deep learning embeddings, able to capture the semantic similarity of words by representing them as vectors using corpus as input, with string similarity joins, through our join2vec framework. In order to produce a model that efficiently performs semantic rich joins, we argue about several optimizations for the algorithms and vector manipulation, that manage to make join2vec highly scalable.

Applications. The importance of a semantic string similarity join can be shown through a real-world application scenario. A core task of data analysis is the semantic data integration, in which data scientists try to merge datasets from different sources in order to produce a duplicate-free unified dataset ([27], [10]). When those datasets contain strings, the merging procedure is done using a classic similarity join, able to only locate similar entities that share syntactic similarity, possibly missing entities that share same meaning but they do not have common syntactic attributes. By performing a semantic-rich similarity join, we could locate every possible similar pair, being able to perceive many different kinds of similarity and thus performing high quality semantic data integration.

Challenge. High dimensional embedding data are very demanding in terms of resources and computational power. Word embeddings are dense vectors, and usually have several dimensions, making them difficult to store and process. Although there is relevant work in the field of vector manipulation for similarity joins and queries [37], there are several significant performance bottlenecks that we need to optimize. (i) The similarity calculation procedure between two high-dimensional vectors, (ii) The in-memory storing and retrieval of such vectors, and (iii) The filtering techniques that could be applied to enhance the search, both for approximate and exact joins.

Contributions. The main contributions of this paper are as follows:

- We present join2vec, an efficient framework for semantic-rich similarity joins with deep learning embeddings, that performs numerous optimizations routines to enhance the performance.
- We describe different string similarity join algorithms, applicable for exact and approximate joins.
- We develop several algorithmic and hardware-conscious optimizations for join2vec, that exploit both parallelism and modern multi-core hardware architectures in order to make the framework highly scalable.
- Finally, we conduct an experimental evaluation with a wide range of configurations using several string datasets of different size. The evaluation demonstrates that join2vec is a highly efficient and scalable framework for semantic rich string similarity joins.

The rest of this paper is organized as follows: Section 1 is the introduction, section 2 presents the preliminary material, section 3 introduces the join2vec framework, section 4 shows the exact join algorithms, 5 presents the approximate join algorithm, section 6 is the experimental evaluation, section 7 shows the related work in the field and section 8 is the conclusion.

2 PRELIMINARIES

In this section we present the essential background and preliminary material.

RS Join. Given two string datasets S, R, an RS string similarity join (or RS Join) aims to find the most similar pairs of strings between

R and S w.r.t. a similarity threshold th [1]. When it is performed in a single dataset, it is called Self Join. When the similarity between two strings takes into account the semantic similarity of the words, it is called semantic (or semantic-rich) string similarity join. An exact string similarity join (or simply, exact join) is the join where all the pairs that share a similarity w.r.t. the threshold must be obtained. An approximate string similarity join (or simply, approximate join) allows compromises in the procedure in order to improve the performance, by using a filtering method. This means that an approximate join in some cases may not obtain some similar pairs, but exposes the performance trade-offs that exact and approximate joins share.

Similarity Measures. A similarity measure (or similarity metric) is the metric that quantifies the similarity between two words. In classic similarity joins, it can be either a character-based metric, such as edit or hamming distance, or a token-based metric, such as jaccard or cosine distance. Based on the selected similarity, two strings can be considered similar if their similarity score is above or below a threshold.

Filter Verification Framework. String similarity join algorithms respect a generalized interface called Filter Verification Framework [7]. The framework performs a quick iteration over the dataset to produce candidate string pairs, and then the candidates are verified using a similarity measure (e.g. edit distance). This procedure never produces false negatives, but it has a trade-off of the quality and cost of the filtering. The filtering technique works by generating and comparing string signatures (i.e. blueprints of the string), and the implementation has numerous different approaches ([29], [13], [17]).

Word Embeddings. Embeddings are representations of words as multidimensional vectors, where their position in the space correspond to their semantic meaning, i.e. words with similar semantic meaning will have vectors that are close in the space. The similarity between word embeddings is calculated w.r.t. a vector metric, e.g. cosine similarity. Word embeddings are produced by models which output the vectors by performing unsupervised learning to a text corpus. These models, such as Word2Vec[25], Glove[28], FastText[5], MOE[14] and many more are able to provide high quality multidimensional word embeddings.

Context-Rich Opportunities. State-of-the-Art word embedding models, such as FastText and especially MOE are tuned to be able to capture multiple kinds of string similarity, including syntactic and semantic (e.g. synonyms, taxonomy). This is why considering the embeddings of those models for string similarity joins could lead us to design high quality join algorithms.

Locality Sensitive Hashing. Locality Sensitive Hashing (LSH) is a method used for approximate similarity search in high dimensional data. LSH aims to hash similar vectors to the same buckets of a hash table, allowing for faster search by obtaining only the entries of a specific bucket [32]. A common technique to hash vectors is by initializing a number of hyperplanes to the vector space, and hash the vectors based on if they are parallel to each hyperplane using binary representation. This way, the dense vectors are hashed into binary vectors, with as many dimensions as the hyperplanes used. LSH is an approximate structure with probabilistic guarantees, while the number of the hyperplanes expose a trade-off of performance and accuracy. Possible alternatives to this approach is minhashing [38] and simhashing [30].

Bloom Filters. A bloom filter is a probabilistic data structure, used to efficiently answer exact group membership queries. A bloom filter uses a bit array and hash functions, which output indices of the bit array. The entities of a collection are hashed to the indices of the array, and after this process the array contains binary 1s and 0s. 1s mean that an entity of the collection is hashed in the corresponding position, while a 0s mean that no entity was hashed in that position. To answer a membership query, the query is hashed using the hash functions, and if the output leads to an index equal to 1, it means that the object might be in the collection. The bloom hash-functions can be either predefined or learned from the data [26], and the hashing process can return false positives, but never false negatives. For the case of vectors, distance sensitive bloom filters can be applied, but this work is unfortunately limited to binary vectors ([21], [16]).

FAISS. FAISS [20] is a framework for high dimensional vector processing. It works by building indices that support fast lookup operations for dense high dimensional vectors, and it is proven to perform efficiently for k-NN similarity search over those vectors.

SIMD. Single-Instruction Multiple-Data (SIMD) refers to a parallel architecture that allows the execution of the same operation on multiple data simultaneously [23]. Using SIMD, we are able to reduce the latency of an operation, because the corresponding instructions are fetched once, and then applied in parallel to multiple data. For the case of high dimensional vectors, SIMD has been useful both for cosine similarity [22] and euclidean distance calculations [33].

3 JOIN2VEC: JOINS OVER EMBEDDINGS

In this section, we present the join2vec framework.

Data Preparation. Every embedding model needs an input corpus in the form of raw sentences to perform unsupervised training and generate the word embeddings. We harvested several GB of data from Wikipedia latest articles, and then we processed them to strip of unrelated information and keep only the raw text in form of sentences. After the preprocessing, we ended up with 30GB of raw sentences which we used to train the model. Using Wikipedia sentences to generate embeddings is a well known and used method in practise ([18], [15], [31], [11], [9])

Unsupervised Learning. We selected FastText as our word embedding model, because of its superiority against Word2Vec and GloVe [12] (MOE is not open source yet) and because of its ability to generate vectors for words that are not present in the learning set, by producing n-grams and partially calculating the vector.

We trained FastText on the aforementioned corpus of Wikipedia through unsupervised learning, and we produced approximately 3 million word vectors of 100 dimensions. We should mention that this process need to be executed only once, and then we only need to load the vectors to join2vec framework. This means that the training process and the model is agnostic to join2vec, and the choice of FastText is not limiting; We could choose any word embedding library available.

[1] RS Join and Full Join are used interchangeably in this paper

From a system perspective, join2vec only requires a mapping of the words to their corresponding vector, which is kept in-memory and it will be used to obtain the word vectors during the join time. Then, join2vec is able to perform exact or approximate semantic string similarity joins, without any need of predefined and preconfigured similarity rules and taxonomies.

4 EXACT JOINS

In this section, we present the join2vec approach for exact similarity joins.

4.1 Nested loop.

Algorithm. The exact similarity join of join2vec is implemented through a nested loop join, which does an online processing over the data, without the need of additional indices, in order to return the similar string pairs. This approach is illustrated in algorithm 1. The algorithm requires the datasets to be joined (R and S), the similarity threshold (th) and the embedding model ($model$), which holds the mapping of the words to their vectors. As we mentioned in section 3, we use FastText as our model. The algorithm iterates over the each word of the first dataset (or relation [2]), it obtains the vector and it compares it with every vector of each word of the dataset. The word pairs that share a similarity greater among the given threshold are kept in a set to be returned when the join completes.

Parallelism. We parallelized algorithm 1 using two methods: Data Parallel (DP) and Fetch-And-Increment (FAI). For DP method, we partition the first dataset into sets equal to the number of threads, so that each thread works on a specific set to perform the join in an embarrassingly parallel way. FAI method uses a shared counter to assign words of the first dataset to threads when they request new words. This way, each thread gets assigned a word to work only when it has finished its previous work, leading to a more balanced scheme.

Hardware Optimizations: SIMD and Prefetching. As we mentioned in section 2, SIMD manages to perform multiple computations in parallel using vectorization, and it is applicable for cosine similarity calculations. As the cosine similarity calculation between two vectors is the most frequent operation in join2vec, we use SIMD instead of the naive calculation. Moreover, accessing the model to obtain the vectors in each iteration is costly. Consider, for example, the following scenario, for a full join of two datasets of size R and S respectively. If the model is accessed in every iteration, we would need $O(R \times S)$ accesses, will if we prefetch all the vectors first, we would only need $O(R + S)$. For this reason, we prefetch all the vectors from the model before the beginning of the join, and we keep them in a direct data structure, such as an array, so that we have direct access to them during the iterations. By this optimization, we skip the lookup cost of the model and we exploit the locality of the data that the arrays offer.

4.2 FAISS: Joins using k-NN

Algorithm. FAISS is an efficient similarity search tool for high dimensional dense vectors, optimized for k-NN similarity search. We adapted join2vec to be able to exploit FAISS, obtaining the similar pairs by fetching batches of k-NN pairs. The procedure is illustrated in algorithm 2. The algorithm has exactly the same requirements with algorithm 1, plus the $batch\_size$ integer, which denotes how many neighbors will be fetched from FAISS every time. It begins by prefetching all the vectors of the two datasets into the corresponding structures (this is an essential part of the algorithm now, in order to build the index on the second relation), and then proceeds to build a FAISS hash-based index over the inner dataset. We should mention here that FAISS also contains structures for approximate search, but we don't argue about such structures in this work, as we discuss exact joins. For every vector of the outer dataset (R), we obtain a sorted k-NN list, containing the top-N similar pairs. If this list contains enough pairs (i.e. the last pairs share similarity which is less than the threshold), or there aren't any more words to search, we validate the list and add the pairs to the result set. We keep fetching more similar vectors, in case there are not enough based on the given threshold. This procedure is done for each vector and stops if we have enough vectors or if the are no other vectors left to fetch.

Optimizations. FAISS internally makes use of parallelism and SIMD to enhance the numeric calculations for vector similarity. The methods of parallelism described in Nested Loop (DP and FAI) are still applicable for this approach. As we mentioned in the algorithm description, in this approach, prefetching the vectors is a mandatory step for the inner relation in order to build the index.

Algorithm 1: Nested Loop join2vec

Input: Dataset R, Dataset S, Threshold th, Model $model$

```
Set results = []
for every word r in R do
    Vector vr = model.getVector(v)
    for every word s in S do
        Vector vs = model.getVector(s)
        Float sim = cosine_similarity(vr, vs)
        if sim ≥ th then
            results.add(pair(r, s, sim))
return results
```

Algorithm 2: FAISS join2vec

Input: Dataset R, Dataset S, Threshold th, Model $model$, Integer $batch\_size$

```
Set results = []
Set lvectors = prefetch(model, R)
Set rvectors = prefetch(model, S)
Index Index = FAISS.buildIndex(rvectors)
for every vector lv in lvectors do
    Sorted Vector List candidates = Index.getKNN(lv, batch_size)
    while cosine_similarity(candidates.getLast(), lv) ≤ th or
      candidates.size() = S.size() do
        batch_size = 2 * batch_size
        candidates = Index.getKNN(lv, batch_size)
    for every vector cv in candidates do
        Float sim = cosine_similarity(lv, cv)
        if sim ≥ th then
            results.add(pair(lv, cv, sim))
return results
```

[2] dataset and relation are used interchangeably

Algorithm 3: LSH join2vec

Input: Dataset R, Dataset S, Threshold th, Model $model$, Integer $hyperplanes$

```
1  Set results = []
2  Set lvectors = prefetch(model, R)
3  Set rvectors = prefetch(model, S)
4  HashTable table = build_hash(rvectors)
5  for every vector lv in lvectors do
6  |   List candidates = table.lookup(hash(lv, hyperplanes)
7  |   for every vector cv in candidates do
8  |   |   Float sim = cosine_similarity(lv, cv) if sim ≥ th then
9  |   |   |   results.add(pair(lv, cv, sim))
10 return results
```

5 APPROXIMATE SIMILARITY JOINS

In this section we describe the approximate similarity joins of join2vec.

Algorithm. To perform an approximate similarity join we combine the implementation of LSH, described in section 2, with our join2vec ideas. The procedure is illustrated in algorithm 3. The algorithm requires the same arguments as algorithm 1, plus the number of the hyperplanes that we want to use for LSH. The procedure begins by prefetching the needed vectors (which is a mandatory step to build the hashtable) and then we proceed to build the hashtable based on the vectors of the inner dataset and the selected hyperplanes. Then, for every vector of the words of the outer relation, we hash using the same method and we obtain the vectors that belong to the bucket that the vectors were hashed. Those vectors are the approximate candidate set, and they need to be validated. Thus, we validate each candidate set, keeping the similar pairs to the corresponding result set. Note that this approach could be modified to use Bloom Filters instead of LSH, but unfortunately Bloom Filters for high dimensional objects are applicable only for binary vectors.

Optimizations. In this version of join2vec join, we can still apply the SIMD cosine similarity calculation during the validation of the candidates. In addition, our parallelism approaches presented for Nested Loop in section 4 can applied as they are for LSH. As we mentioned in the algorithm description, prefetching is a mandatory step for the inner dataset, because the vectors are used to build the hashtable.

6 EVALUATION

In this section, we present our experimental evaluation.

Setup. We performed our experimental evaluation on a server with 24 physical cores and 24 hyperthreads (total 48 threads), with a 2 x Intel Xeon E5-2660 CPU and 126GB of memory, running Ubuntu 18.04.4 LTS.

Datasets. We evaluate our methods with string (word) datasets of different sizes, containing up to 100K words of average length of 7. We created the datasets by sampling words from the Wikipedia corpus mentioned in section 3, excluding stopwords and avoiding duplicates. Our word embeddings are provided by FastText, using 30GB of text corpus, producing approximately 3M vectors of 100 dimensions.

Evaluation Method. In every experiment we measure the total join time elapsed to perform the actual join, including the preprocessing costs of prefetching. Our evaluation deals with in-memory data, which means that we do not report the costs of the time needed to load the model and the datasets in-memory. The reported numbers are the average of of 5 runs in each configuration.

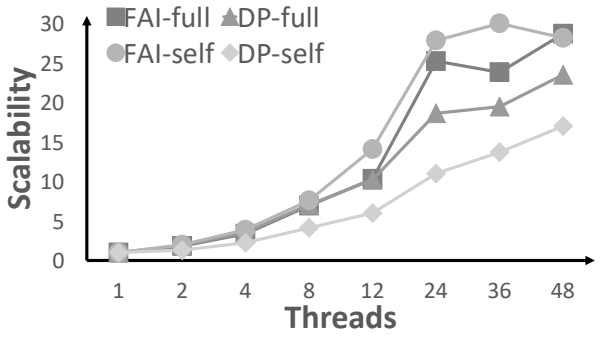
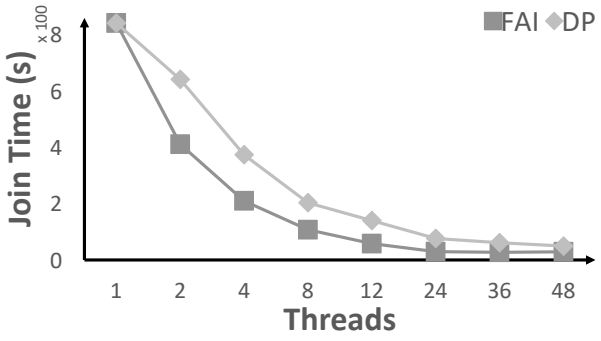


Figure 1: Scalability of the parallelization methods of join2vec.

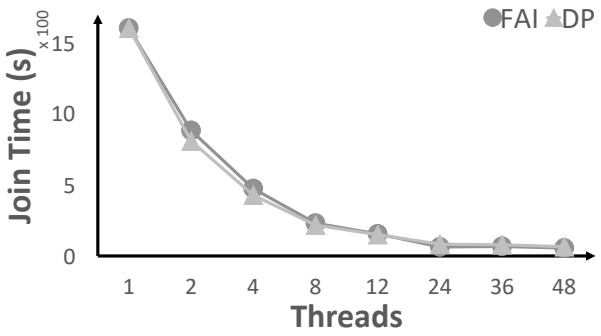
Scalability. Figure 1 shows the comparison of the parallelization methods of nested loop join2vec. We evaluated both Data Parallel (DP) and Fetch And Increment (FAI) methods, for self joins of 100K words and for full joins of 100K*100K words. Y-axis presents the scalability of each method, showing how much time faster it gets for every thread configuration of x-axis, compared to the sequential version. FAI-self and FAI-full are the results of the FAI method for self and full join, while DP-self and DP-full are the results of the DP method for self and full join. In every run, join2vec uses prefetching for data access and SIMD for similarity calculation. We observe that FAI methods in general scale better than the DP methods. This is because the usage of FAI for parallel execution provides a work-balanced scheme, as every thread requests a new word to join every time it has finished its current work, thus a thread gets assigned work only when it is available. The work distribution scheme that DP follows splits the data statically, and every thread gets a static chunk of the dataset to perform the join. This method can sometimes lead to imbalances, especially for self joins, were the first chunks have a lot more work in comparison to the last ones.

Another interesting observation from this graph, is the performance of all methods after 24 threads, were we see that some methods do not scale as much they did for previous thread configurations, while others do not even scale. This happens because up to 24 threads, we assign work to different physical cores, while when we increase the number of threads to 36 or 48 we have started assigning more threads in each core (hyperthreading), resulting in this performance behavior for the parallelization methods. Overall, we conclude that FAI method manages to scale better in both full and self join scenarios, being approximately 30x times faster than their sequential version, because of the load balanced scheme that implements.

Parallelism. In addition to the scalability experiment, we present in figure 2 the direct comparison between Data Parallel method (DP) and Fetch And Increment method (FAI), both for self (figure 2a) and full (figure 2b) joins. Y-axis is the total join time, while x-axis shows the number of threads in each run. We use prefetching and SIMD optimizations for every run. We see that overall, especially



(a) Self Join, 100K words



(b) Full Join, 100K*100K words

Figure 2: Comparison between FAI and DP method for different number of threads.

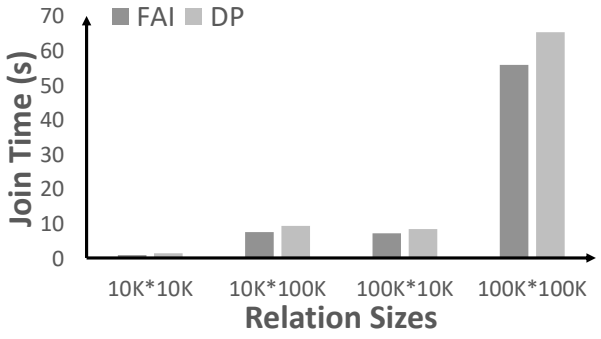


Figure 3: Comparison of FAI and DP method for different relation sizes.

for higher thread configurations, the FAI method performs better, because of the load balancing scheme it implements. From our experiments, we conclude that FAI can be up to 1.29x times faster than DP for full join, and up to 2.6x times faster than DP for the case of self join.

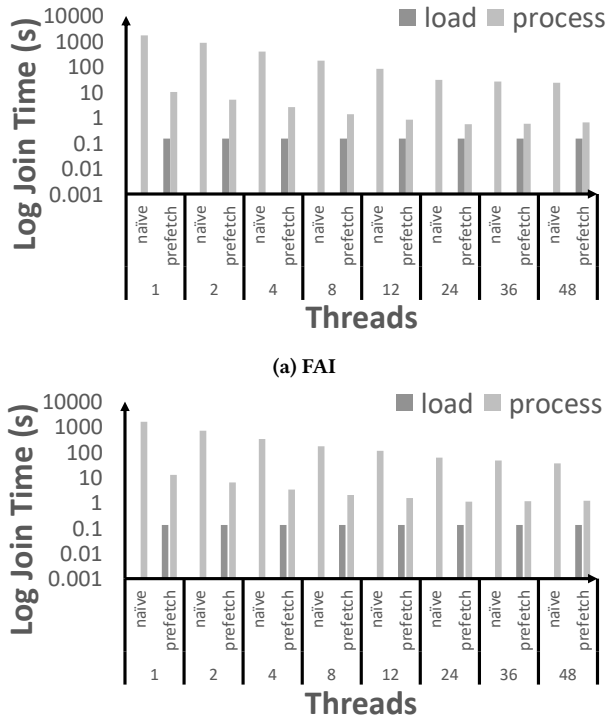
Relation Sizes. Figure 3 presents the comparison between Fetch And Increment (FAI) and Data Parallel (DP) methods of nested loop join2vec for full join, using 48 threads with prefetching and SIMD. Y-axis shows the total join time needed in each configuration, while x-axis contains the corresponding relation sizes for the join. We observe that in every relation size configuration, FAI method manages to give better results than the DP method. Overall, the results show that FAI can be up to 1.74x times faster than DP.

Prefetching. Figure 4 presents the comparison of the prefetching data access pattern against the naive implementation of accessing the model in every iteration, for Fetch And Increment (FAI, figure 4a) and Data Parallel (DP, figure 4b) methods. The experiments correspond to instances of nested loop join2vec running for full joins of 10K*10K words, using SIMD for the numerical calculations of cosine similarity. We compare the performance of the instance of join2vec using prefetching (prefetch) with the naive version that accesses the model in each iteration (naive), reporting separately the actual join time (process bars) and preprocessing time (time to fetch the vectors, load bars). In both figures (a,b) Y-axis corresponds to the log join time, while x-axis shows the results for different thread configurations. Of course, the naive version does not have load bars, because it does not prefetch anything. In both FAI and DP runs, we see that prefetching manages to offer a great performance speedup, because of the reduction of model accesses, described in section 4. We observe that for FAI, using prefetching can be up to 168x times faster than accessing the model in each iteration, while for DP it can be up to 245x times faster. This great speedup is expected. For 10K*10K full joins, the naive method would need a total of 10.000^2 accesses to the model, while prefetching would need only 20.000 accesses. This implies that the model accesses in prefetching are linearly correlated to the relation sizes, while for the naive model are quadratic.

SIMD. Figure 5 presents the comparison of total join time between nested loop join2vec using SIMD against the naive implementation of cosine similarity calculation. For both versions, we use prefetching and the Fetch And Increment (FAI) parallelization method. We present results for both self join with a single dataset of 100K words (figure 5a), and full join for datasets of 100K*100K words each (figure 5b). Y-axis is the total join time needed to perform the joins, while x-axis has the corresponding number of threads. We observe in both self and full joins that SIMD manages to speedup the algorithm, because it achieves to optimize the most frequent operation performed in the join. Overall, we see that using SIMD, we are able to be up to approximately 2x times faster than the naive implementation in both full and self join experiments. Being approximately 2x times faster is an expected speedup, given our server's 2-register architecture and SIMD vectorization.

Cost Models. This set of experiments compares the theoretical cost of different join types supported by join2vec. Nested loop join is denoted as NLJ, while FAISS and LSH join are denoted as HASHJ, as they both belong to a general category of hash joins. We also define NLJ-P (P for prefetching) which corresponds to the cost with the prefetching optimization of nested loop. Note that there is no need to define HASHJ-P, as prefetching is an essential attribute of FAISS and LSH, as described in sections 4 and 5 respectively. We denote the constants R (outer relation size), S (inner relation size), M (cost of accessing the model), sel (selectivity rate) and P (cost of calculating the similarity between a pair of vectors). Based on these notations, we define the our cost models. For NLJ, the cost model is described by

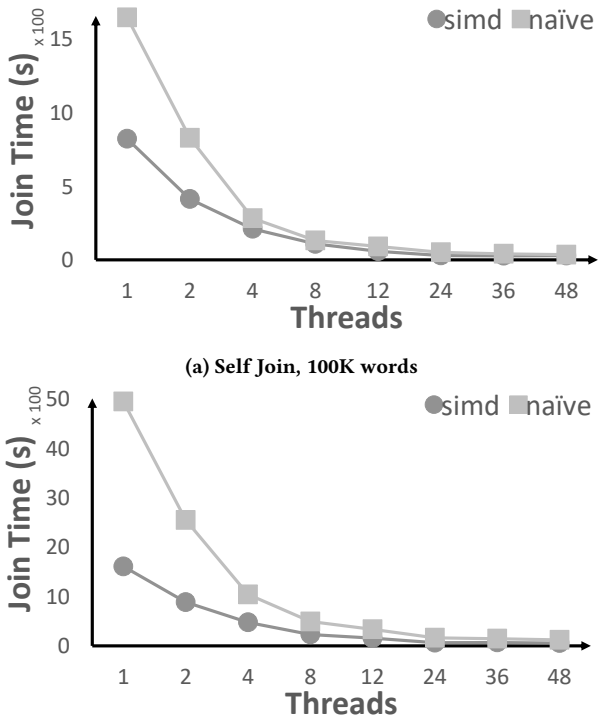
$$cost_{NLJ} = R + (R \times S) + (sel \times R \times S \times P) + (M \times R \times S)$$



(a) FAI

(b) DP

Figure 4: Data access pattern comparison of join2vec, for FAI and DP, for a full join of 10K*10K words



(a) Self Join, 100K words

(b) Full Join, 100K*100K words

Figure 5: Comparison between SIMD and naive cosine similarity calculations for self and full joins.

because we need to traverse all of the outer relation R, in each iteration of this to traverse relation S*R times, and validate the vector pairs. We also access the model a total of M*R*S times, because we do not prefetch anything, thus we need to access it in each iteration.

For NLJ-P, the cost model is described by

$$cost_{NLJ-P} = R + (R \times S) + (sel \times R \times S \times P) + M \times (R + S)$$

because the prefetching allows us to only access the model R+S times. Note that we introduce the selectivity (sel) for both nested loop cost models as a theoretical parameter. In our implementation, we need to validate every possible pair of vectors, thus selectivity = 1. For the HASHJ, the cost model can be described by

$$cost_{HASHJ} = S + R + (sel \times P \times R \times S) + M \times (R + S)$$

Because of the cost of building the hashtable (or FAISS hash-based index) on relation S, the cost of iterating over the vectors of R, the cost of validating the candidate set (here selectivity makes a lot of sense, as FAISS and LSH return a fraction of the total possible pairs) and the cost of accessing the model using prefetching.

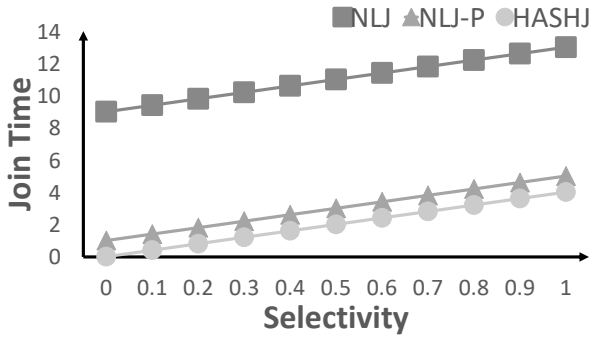
Figure 6a compares the three models, NLJ, NLJ-P and HASHJ for a full join with fixed relation size of 10K*10K words, and different selectivity rates (x-axis, supposing a filtering mechanism for nested loops), showing their corresponding join cost (y-axis). We conclude that HASH-J has the best performance, which is slightly better than NLJ-P. As we expected, we see that NLJ without prefetching leads to a higher cost, because of the need to access the model frequently.

Figure 6b compares the models for full joins with fixed selectivity, showing their cost on y-axis, and the corresponding relation size on x-axis. We compare the NLP and NLP-P model with selectivity 1 (as in join2vec nested loop, where we compare every pair) with different versions of HASHJ (which can be either FAISS or LSH), based on different selectivity rates of 0.25, 0.5, 0.75 and 1. As expected once again, we see that the HASH-J models have less cost, while the usage of NLJ without prefetching leads to higher costs, because of the frequent model access.

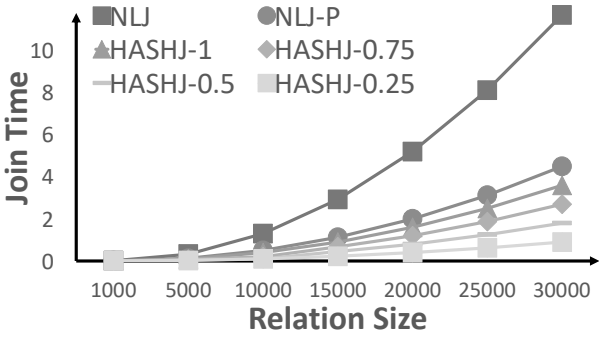
7 RELATED WORK

In this section we discuss the relevant work in the field of string similarity joins.

Classic String Similarity Joins. There is a wide range of algorithms that perform string similarity joins finding pairs of strings that share a degree of syntactic similarity (e.g. typos), which are extensively evaluated [19]. For character-based metrics, such as edit distance, FastSS [4] is a very efficient approach, applicable for strings of small length, and it works by capturing similarity based on the neighbours of each character. For token-based metrics, such as jaccard or cosine distance, PPJoin+ is a dominant approach [39]. because it achieves good filtering quality by ordering the signatures of the tokens of a word. AdaptJoin [35] has also great performance and it is available for both token-based and character-based metrics, while it is based on adaptive signature sizes. The aforementioned algorithms are based on the Filter-Verification framework described



(a) Performance for different selectivity rates of different join2vec join models.



(b) Performance of join2vec models with fixed selectivity for different relation sizes

Figure 6: Cost model comparison for join models of join2vec.

in section 2 and have good performance, but unfortunately they are only available for scenarios we are interested only in syntactic similarity between strings (e.g. typos).

Semantic String Similarity Joins. The Unified Framework algorithm [41] addresses the problem of semantic string similarity joins, and aims to provide an implementation able to capture multiple kinds of string similarity (typos, synonyms, taxonomies) by creating an approximation routine that calculates the pairwise similarities among words. This framework does not use embeddings to capture the string similarity and requires a predefined taxonomy graph and a predefined set of synonym rules.

Summary. To the best of our knowledge, join2vec is the first approach that aims to incorporate word embeddings for string similarity joins and argues about the optimizations of vector processing and implementation with different join types.

8 CONCLUSION

In this work we presented join2vec, a scalable and efficient framework which uses vector embeddings to perform semantic rich and efficient string similarity joins. The framework manages to address a number of problems related to scalability and efficiency for high dimensional vector processing by taking full advantage of the multi-core architecture of modern servers through a range of algorithmic and hardware-conscious optimizations, providing implementations for both exact and approximate joins. In future work, we plan to extend our results using different embedding models, while we also plan to evaluate approximate joins using the approximate hash-based structures of FAISS and compare the different implementations.

REFERENCES

[1] Felipe Almeida and Geraldo Xexéo. 2019. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069* (2019).
[2] Amir Bakarov. 2018. A survey of word embeddings evaluation methods. *arXiv preprint arXiv:1801.09536* (2018).
[3] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. 2007. Scaling up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 131–140. https://doi.org/10.1145/1242572.1242591
[4] Thomas Bocek, Ela Hunt, Burkhard Stiller, and Fabio Hecht. 2007. *Fast similarity search in large dictionaries.* University.
[5] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the association for computational linguistics* 5 (2017), 135–146.
[6] Brent Bryan, Frederick Eberhardt, and Christos Faloutsos. 2008. Compact similarity joins. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 346–355.
[7] Kaushik Chakrabarti, Surajit Chaudhuri, Venkatesh Ganti, and Dong Xin. 2008. An efficient filter for approximate membership checking. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 805–818.
[8] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. 2006. A primitive operator for similarity joins in data cleaning. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 5–5.
[9] Danqi Chen, Adam Fisch, Jason Weston, and Antoine Bordes. 2017. Reading wikipedia to answer open-domain questions. *arXiv preprint arXiv:1704.00051* (2017).
[10] William W Cohen and Jacob Richman. 2002. Learning to match and cluster large high-dimensional data sets for data integration. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. 475–480.
[11] Arjun Das, Debasis Ganguly, and Utpal Garain. 2017. Named entity recognition with word embeddings and wikipedia categories for a low-resource language. *ACM Transactions on Asian and Low-Resource Language Information Processing (TALLIP)* 16, 3 (2017), 1–19.
[12] EDDY MUNTINA Dharma, F Lumban Gaol, HLHS Warnars, and BENFANO Soewito. 2022. The accuracy comparison among word2vec, glove, and fasttext towards convolution neural network (CNN) text classification. *J Theor Appl Inf Technol* 100, 2 (2022), 31.
[13] Sarang Dharmapurikar and John Lockwood. 2005. Fast and scalable pattern matching for content filtering. In *2005 Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE, 183–192.
[14] Bora Edizel, Aleksandra Piktus, Piotr Bojanowski, Rui Ferreira, Edouard Grave, and Fabrizio Silvestri. 2019. Misspelling oblivious word embeddings. *arXiv preprint arXiv:1905.09755* (2019).
[15] Alessio Ferrari, Beatrice Donati, and Stefania Gnesi. 2017. Detecting domain-specific ambiguities: an NLP approach based on Wikipedia crawling and word embeddings. In *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*. IEEE, 393–399.
[16] Mayank Goswami, Rasmus Pagh, Francesco Silvestri, and Johan Sivertsen. 2017. Distance sensitive bloom filters without false negatives. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 257–269.
[17] Luis Gravano, Panagiotis G Ipeirotis, Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugauelayut Muthukrishnan, Divesh Srivastava, et al. 2001. Approximate string joins in a database (almost) for free. In *VLDB*, Vol. 1. 491–500.
[18] Edouard Grave, Piotr Bojanowski, Prakhar Gupta, Armand Joulin, and Tomas Mikolov. 2018. Learning word vectors for 157 languages. *arXiv preprint arXiv:1802.06893* (2018).
[19] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. 2014. String similarity joins: An experimental evaluation. *Proceedings of the VLDB Endowment* 7, 8 (2014), 625–636.
[20] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.
[21] Adam Kirsch and Michael Mitzenmacher. 2006. Distance-sensitive bloom filters. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 41–50.
[22] Robert Lim, Boyana Norris, and Allen Malony. 2018. A similarity measure for GPU kernel subgraph matching. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 37–53.
[23] Chris Lomont. 2011. Introduction to intel advanced vector extensions. *Intel white paper* 23 (2011).

[24] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. 2013. String similarity measures and joins with synonyms. In *ACM SIGMOD Conference*.
[25] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).
[26] Michael Mitzenmacher. 2018. A model for learned bloom filters and optimizing by sandwiching. *Advances in Neural Information Processing Systems* 31 (2018).
[27] Michalis Mountantonakis and Yannis Tzitzikas. 2020. Large-scale Semantic Integration of Linked Data. *Comput. Surveys* 5 (2020), 1–40.
[28] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.
[29] Leonardo Andrade Ribeiro and Theo Härder. 2011. Generalizing prefix filtering to improve set similarity joins. *Information Systems* 36, 1 (2011), 62–78.
[30] Caitlin Sadowski and Greg Levin. 2007. Simhash: Hash-based similarity detection.
[31] Ehsan Sherkat and Evangelos E Milios. 2017. Vector embedding of wikipedia concepts and entities. In *International conference on applications of natural language to information systems*. Springer, 418–428.
[32] Malcolm Slaney and Michael Casey. 2008. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal processing magazine* 25, 2 (2008), 128.
[33] Bo Tang, Man Lung Yiu, Yuhong Li, et al. 2017. Exploit every cycle: Vectorized time series algorithms on modern commodity cpus. In *International Workshop on Accelerating Data Analysis and Data Management Systems, International Workshop on In-Memory Data Management and Analytics*. Springer, 18–39.
[34] Jiannan Wang, Jianhua Feng, and Guoliang Li. 2010. Trie-Join: Efficient Trie-Based String Similarity Joins with Edit-Distance Constraints. *Proc. VLDB Endow.* 3, 1–2 (sep 2010), 1219–1230. https://doi.org/10.14778/1920841.1920992
[35] Jiannan Wang, Guoliang Li, and Jianhua Feng. 2012. Can we beat the prefix filtering? An adaptive framework for similarity join and search. In *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 85–96.
[36] Shirui Wang, Wenan Zhou, and Chao Jiang. 2020. A survey of word embeddings based on deep learning. *Computing* 102, 3 (2020), 717–740.
[37] Yifan Wang. 2022. A Survey on Efficient Processing of Similarity Queries over Neural Embeddings. *arXiv preprint arXiv:2204.07922* (2022).
[38] Wei Wu, Bin Li, Ling Chen, Junbin Gao, and Chengqi Zhang. 2020. A review for weighted minhash algorithms. *IEEE Transactions on Knowledge and Data Engineering* 34, 6 (2020), 2553–2573.
[39] C Xiao, W Wang, X Lin, and JX Yu. 2008. Efficient similarity joins for near duplicate detection. In WWW, pages 131-140. (2008).
[40] Chuan Xiao, Wei Wang, Xuemin Lin, Jeffrey Xu Yu, and Guoren Wang. 2011. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)* 36, 3 (2011), 1–41.
[41] Pengfei Xu and Jiaheng Lu. 2019. Towards a unified framework for string similarity joins. *Proceedings of the VLDB Endowment* (2019).