Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins

Sahaana Suri
Stanford University
sahaana@stanford.edu

Ihab F. Ilyas
University of Waterloo
ilyas@uwaterloo.ca

Christopher Ré
Stanford University
chrismre@cs.stanford.edu

Theodoros Rekatsinas
UW-Madison
thodrek@cs.wisc.edu

ABSTRACT

Structured data, or data that adheres to a pre-defined schema, can suffer from fragmented context: information describing a single entity can be scattered across multiple datasets or tables tailored for specific business needs, with no explicit linking keys. Context enrichment, or rebuilding fragmented context, using *keyless joins* is an implicit or explicit step in machine learning (ML) pipelines over structured data sources. This process is tedious, domain-specific, and lacks support in now-prevalent no-code ML systems that let users create ML pipelines using just input data and high-level configuration files. In response, we propose Ember, a system that abstracts and automates keyless joins to generalize context enrichment. Our key insight is that Ember can enable a general keyless join operator by constructing an index populated with task-specific embeddings. Ember learns these embeddings by leveraging Transformer-based representation learning techniques. We describe our architectural principles and operators when developing Ember, and empirically demonstrate that Ember allows users to develop no-code context enrichment pipelines for five domains, including search, recommendation and question answering, and can exceed alternatives by up to 39% recall, with as little as a single line configuration change.

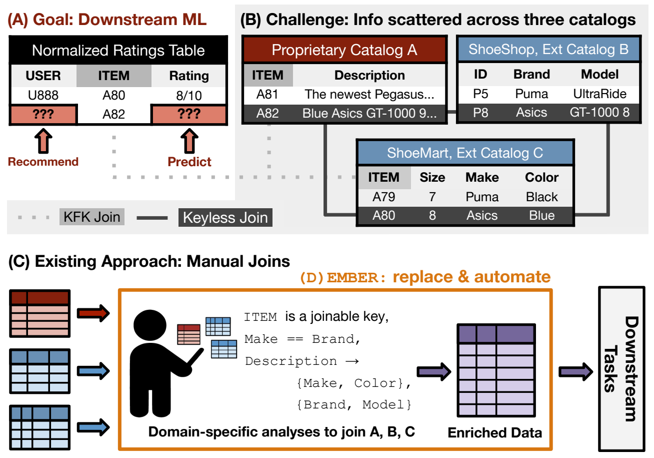
PVLDB Reference Format:
Sahaana Suri, Ihab F. Ilyas, Christopher Ré, Theodoros Rekatsinas. Ember: No-Code Context Enrichment via Similarity-Based Keyless Joins. PVLDB, 15(3): 699 - 712, 2022.
doi:10.14778/3494124.3494149

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sahaana/ember.

1 INTRODUCTION

Machine learning (ML) systems that extract semantic context from unstructured data have revolutionized domains spanning computer vision [47] and natural language processing [24, 66]. Unfortunately, applying these systems to structured and semi-structured datasets with pre-defined schemas is challenging as their context is often

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494149


(A) Goal: Downstream ML
(B) Challenge: Info scattered across three catalogs
Recommend
Predict
KFK Join
Keyless Join
(C) Existing Approach: Manual Joins
(D) EMBER: replace & automate
Domain-specific analyses to join A, B, C
Enriched Data
Downstream Tasks

Normalized Ratings Table

USER	ITEM	Rating
U888	A80	8/10
???	A82	???

Proprietary Catalog A

ITEM	Description
A81	The newest Pegasus...
A82	Blue Asics GT-1000 9...

ShoeShop, Ext Catalog B

ID	Brand	Model
P5	Puma	UltraRide
P8	Asics	GT-1000 8

ShoeMart, Ext Catalog C

ITEM	Size	Make	Color
A79	7	Puma	Black
A80	8	Asics	Blue

```
ITEM is a joinable key,
Make == Brand,
Description →
        {Make, Color},
        {Brand, Model}
```

Figure 1: An end-to-end task requiring context enrichment. Predicting the rating of and recommending a new product (A82), requires relating the Asics products (highlighted in dark gray) via a keyless join (top). This process is manual due to data heterogeneity—we aim to automate it (bottom).

fragmented: they scatter information regarding a data record across domain-specific datasets with unique schemas. For instance, in Figure 1B, information about Asics shoes is scattered across three catalogs with unique schemas. These schemas are optimized for task-specific querying and often lack explicit linking keys, such as primary key-foreign key (KFK) relationships. This constrains users to a single view of an entity, specialized for a specific business need.

Associating these fragmented data contexts is critical to enable ML-powered applications—a preprocessing procedure we denote as *context enrichment*—yet is a heavily manual endeavor due to task and dataset heterogeneity. Engineers develop solutions for context enrichment tailored to their task, such as similarity-based blocking in data integration [62], retriever models in question answering [91], or retrieval functions in search [71] (see Section 2). Constructing these independent solutions is repetitive, time-consuming, and results in a complicated landscape of overlapping, domain-specific methods. For instance, consider the scenario depicted in Figure 1:

An e-commerce company has a proprietary product catalog, and aggregates product information from several external vendors to perform market analysis. Each vendor uses a unique product catalog, each with unique product representations (i.e., schema) ranging from free-form text to tabular product descriptions; products may overlap and evolve

(Figure 1B). Given normalized tables containing user and rating data, an engineer wishes to estimate the rating for a candidate new product (`A82`*) and identify users to recommend the product to (Figure 1A).*

The engineer must first perform context enrichment by joining information across tables to extract features that capture similarities between the new (`A82`) and existing (`A80`, `P8`) products. They can then estimate the product rating, and recommend the new product to users based on how they rated related products.

The classic data management approach is to denormalize datasets using KFK joins. This fails to solve the problem due to two reasons. First, not all tables can be joined when relying on only KFK relationships (e.g., there is no KFK relationship between Catalog B and Catalogs A or C). Second, even when KFK relationships exist, as between Catalogs A and C (`ITEM`), relying on only KFK semantics fails to capture the similarity between `A82` and `A80`.

Alternatively, the engineer can rely on similarity-based join techniques, as in data blocking [62], built to avoid exhaustive, pairwise comparison of potentially joinable records. However, as we show in Section 9.3, the optimal choice of join operator to maximize recall of relevant records is task-dependent, and may not scale at query time when new records arrive. The engineer must first note that the `Description` column in Catalog A relates to the `Brand` and `Model` columns in Catalog B and the `Size`, `Make`, and `Color` columns in Catalog C. They can then select a custom join based on table properties: for matching primarily short, structured data records, they may want to join based on Jaccard similarity, whereas BM25 may be better suited for purely textual records (see Table 3). As database semantics do not natively support these *keyless joins*, joining arbitrary catalogs remains heavily manual—even large companies rely on vendors to categorize listings, resulting in duplicate listings.[1]

To counter this manual process, we draw from recent no-code ML systems such as Ludwig [57], H20.ai [4], and Data Robot [3] that provide higher-level configuration-based abstractions for developing ML applications. Despite their success, these systems leave context enrichment as a user-performed step.[2] In this paper, we evaluate how to bring no-code semantics to context enrichment.

No out-of-the-box query interface surfaces the relatedness between `A80`, `A82`, and `P8` (all `Asics`, two `GT-1000`, and two `blue`) with minimal intervention. The challenge in developing a system to enable this is in constructing an architecture that is simultaneously:

(1) **General**: Applicable to a wide variety of tasks and domains.
(2) **Extensible**: Customizable for domain or downstream needs.
(3) **Low Effort**: Usable with minimal configuration.

Our key insight is to simplify context enrichment by abstracting an interface for a new class of join: a *learned keyless join* that operates over record-level similarity. We formalize keyless joins as an abstraction layer for context enrichment, as traditional database joins are for combining data sources given KFK relationships.

We then propose Ember: a no-code context enrichment framework that implements a keyless join abstraction layer. Ember creates an index populated with task-specific embeddings that can be quickly retrieved at query time, and can operate over arbitrary semi-structured datasets with unique but fixed schema. To

[1]https://sell.amazon.com/sell.html
[2]https://cloud.google.com/automl-tables/docs/prepare

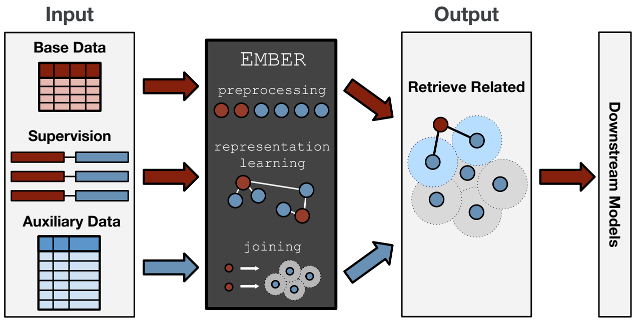


Figure 2: Ember's interface for context enrichment.

provide generality, Ember relies on Transformers [79] for embedding generation, as they have demonstrated success capturing semantic information across textual, semi-structured and structured data [24, 55, 66, 77, 86]. To provide extensibility, Ember is composed of a modular architecture with configurable operators. To provide ease of use, Ember is configurable via a json-based configuration or SQL-style interface, with general default options.

As input, users provide: **(1)** a base data source, **(2)** an auxiliary data source, **(3)** a set of examples of related records across the sources. For each base data record, Ember returns related auxiliary data records as characterized by the examples, which can be post-processed for downstream tasks. We now present Ember's three-step architecture that enables index construction and related record retrieval via Transformer-based embeddings (Figure 2):

Preprocessing. Ember transforms records across all data sources to a common representation, allowing us to apply the same methods across tasks. By default, Ember uses operators that convert each record to a natural language sentence for input to a Transformer-based encoder, which we optionally pretrain using self-supervision.

Representation Learning. Ember tunes the preprocessed representations to identify contextually similar records (characterized by the supervised examples) via a contrastive triplet loss [81]. To evaluate the dependence on labeled data, we examine what fraction of the provided examples Ember needs to match the performance of using all examples, and find that at times, just 1% is sufficient.

Joining. Ember retrieves related records using the tuned representations. Ember populates an index with the embeddings learned to capture record similarity, and uses optimized maximum inner product search (MIPS) to identify a record's nearest neighbors. We show that Ember meets or exceeds the performance of ten similarity join baselines with respect to recall and query runtime.

In the case of the running example from Figure 1, the engineer can replace the manual procedure to collect related information for new product `A82` with keyless joins using Ember's SQL-style API. First, the engineer must collect examples of relevant record pairs that they wish to automatically recover across each data source pair as `supervision`. They can then use the following query to retrieve the closest record in Catalog B to each record in Catalog A, which would associate product `P8` in Catalog B with `A82` in Catalog A:

```
catalog_A INNER KEYLESS JOIN catalog_B USING supervision_B;
```

Given this query, Ember first converts each record in Catalogs A and B into a sentence-based representation. Ember then uses the provided `supervision` to learn a vector representation that encourages clustering of relevant sentences, and transforms all sentences to this representation. Finally, Ember indexes and retrieves the most relevant record from Catalog B for each record in Catalog A based on distance in the learned embedding space.

In this paper, we report on our experience in deploying Ember to perform context enrichment for five tasks: fuzzy joining, entity matching, question answering, search, and recommendation. In summary, we present the following contributions:

- We propose keyless joins with a specification to serve as an abstraction layer for context enrichment. To our knowledge, this is the first work to generalize similarity-based data processing across data integration, natural language processing, search, and recommendation as a data management problem.
- We design and develop Ember, the first no-code framework for context enrichment that implements keyless joins, and provides an API for extending and optimizing keyless joins.
- We empirically demonstrate that Ember generalizes to five workloads by meeting or exceeding the recall of baselines while using 6 or fewer configuration line changes, and evaluate the modular design of Ember's default architecture.

2 CONTEXT ENRICHMENT

In this section, we define context enrichment, and provide example workloads that can be framed as a form of context enrichment.

2.1 Problem Statement

Structured data, or data that adheres to a fixed schema formatted in rows and columns, suffers from context fragmentation. We define structured data to include semi-structured and textual data sources that may be stored and keyed in data management systems, such as content from Wikipedia articles or web search results. Structured data follows a scattered format that is efficient to store, query, and manipulate for specific business needs. This may be in the form of an e-commerce company storing datasets cataloging products and user reviews (Figure 1). To use these fragmented datasets for downstream ML, practitioners unify them to construct discriminative features.

We refer to this preprocessing join procedure as *context enrichment*, which we now define: given a base dataset in tabular form, D_0, context enrichment aligns D_0 with context derived from auxiliary data sources, $D = \{D_1, ..., D_M\}$, to solve a task T. We focus on a text-based, two-dataset case ($|D| = 1$), but propose non-text and multi-dataset extensions in Section 8 as future work. We explore context enrichment in regimes with abundant and limited labeled relevant pairs Y to learn source alignment.

We represent each dataset D_i as an $\mathbf{n_i} \times \mathbf{d_i}$ matrix of $\mathbf{n_i}$ data points (or records) and $\mathbf{d_i}$ columns. We denote C_{ir} as the r^{th} column in dataset i. We denote d_{ij} as the j^{th} row in dataset i. Columns are numeric or textual, and we refer to the domain of d_{ij} as $\mathbf{D_i}$.

Our goal is to *enrich*, or identify all context for, each d_{0j} in D_0: auxiliary records d_{lm} ($l \neq 0$) that are related to d_{0j}, as per task T.

2.2 Motivating Applications

Context enrichment is a key retrieval and processing component in many domains. A general context enrichment framework that can be extended according to domain-specific advances would reduce the developer time needed to bootstrap custom ML pipelines. We now describe five such domains (evaluated in Section 9). Additional domains include entity linkage [61, 73], nearest neighbor machine translation [40], and nearest neighbor language modeling [41].

Entity Matching (EM) and Deduplication. EM identifies data points that refer to the same entity across two different collections of entity mentions [46]. An entity denotes any distinct real-world object such as a person or organization, while its mention is a reference in a dataset. Entity deduplication is a special case of EM, where both collections of entity mentions are the same.

We use context enrichment for entity matching as follows: D_0 and D_1 represent the two collections of entity mentions. Context enrichment aligns entities d_{1m} in the auxiliary dataset D_1 with each entity d_{0j} in the base dataset D_0. In entity deduplication, $D_0 = D_1$. The downstream task T is a binary classifier over the retrieved relevant records, as the aim is to identify matching entities.

Fuzzy Joining. A fuzzy join identifies data point pairs that are similar to one another across two database tables, where similar records may be identified with respect to a similarity function and threshold [20, 80] defined over a subset of key columns.

We use context enrichment for fuzzy joining as follows: D_0 and D_1 represent the two database tables. We let C_0 and C_1 denote the set of key columns used for joining. Records d_{0j} and d_{1m} are joined if their values in columns C_0 and C_1 are similar, to some threshold quantity. This is equivalent to a generic context enrichment task with a limited number of features used. The downstream task T is a top-k query, as the aim is to identify similar entities.

Recommendation. A recommender system predicts the rating that a user would give an item, typically for use in content recommendation or filtering [56]. We also consider the broader problem of determining a global item rating, as in the e-commerce example in Figure 1, as a form of recommendation problem—the global score may be predicted by aggregating per-user results.

We use context enrichment for recommendation as follows: D_0 represents all information regarding the entity to be ranked (e.g., a base product table, or new candidate products), and D_1 represents the information of the rankers, or other auxiliary data (e.g., catalogs of previously ranked products). Context enrichment aligns rankers d_{1m} in the auxiliary dataset D_1 with those that are related to each entity to be ranked d_{0j} in the base dataset D_0. The downstream task T is to return the top-1 query, or to perform aggregation or train an ML model over the returned top-k entries.

Search. An enterprise search engine returns information relevant to a user query issued over internal databases, documentation, files, or web pages [34, 59]. A general web retrieval system displays information relevant to a given search query [15, 28]. Both rely on information retrieval techniques and relevance-based ranking over an underlying document corpus as building blocks to develop complex, personalized pipelines, or for retrospective analyses [13].

We use context enrichment for retrieval and ranking in search as follows: D_0 represents the set of user queries, and D_1 represents the underlying document corpus. Context enrichment aligns documents d_{1m} in the auxiliary dataset D_1 that map to each query d_{0j} in the base dataset D_0. The downstream task T is to return the top-k documents for each query, sorted by their query relatedness.

Question Answering. Question answering systems answer natural language questions given a corresponding passage (e.g., in reading comprehension tasks) or existing knowledge source (e.g., in open domain question answering) [67, 70]. Multi-hop question answering generalizes this setting, where the system must traverse a series of passages to uncover the answer [85]. For instance, the question "what color is the ocean?" may be provided the statements "the sky is blue" and "the ocean is the same color as the sky."

We use context enrichment for the retriever component [91] of a classic retriever-reader model in open domain question answering. Our task is to identify the candidate text spans in the provided passages that contain the question's answer (the retriever component), and a downstream reader task can formulate the answer to the question. We let dataset D_0 represent the set of questions, and D_1 represent the provided passages or knowledge source, split into spans. Context enrichment aligns spans d_{1m} in the auxiliary dataset D_1 that contain the answer to each question d_{0j} in the base dataset D_0. Multi-hop question answering requires an extra round of context enrichment for each hop required. Each round selects a different passage's spans as the base table D_0; to identify context relevant to the question, the user can traverse the related spans extracted from each passage. The downstream task T is to learn a reader model to answer questions using the enriched data sources.

3 LEARNED KEYLESS JOINS

While the applications in Section 2 perform the *same* context enrichment, many state-of-the-art systems for these tasks rediscover primitives (e.g., contrastive learning [81], and optimized indexing [38]) in domains spanning ML, databases, information retrieval, and natural language processing [41, 42, 53, 76]. We now propose learned *keyless joins* as a vehicle towards no-code enrichment.

3.1 Keyless Joins: Definition and Objective

We propose learned *keyless joins* as an abstraction to retrieve related records without primary key-foreign key (KFK) relationships—i.e., facilitate context enrichment. The goal of a keyless join is to quantify and leverage record-level similarity across different datasets.

A keyless join must learn a common embedding space $\mathbf{X}$ for all records d_{ij} across datasets $D_0 \cup D$ that reflects entity similarity. For each $\mathbf{D_i}$, a keyless join learns a function $F_i : \mathbf{D_i} \longrightarrow \mathbf{X}$ that maps elements of D_i to the space $\mathbf{X}$. We denote each transformed data point $F_i(d_{ij})$ as x_{ij}. This mapping must be optimized such that related values map to similar feature vectors in $\mathbf{X}$ and unrelated values map to distant feature vectors in $\mathbf{X}$: $sim(x_{ij}, x_{kl}) > sim(x_{ij}, x_{mn})$ implies that the j^{th} entity in the i^{th} dataset is more closely related to the l^{th} entity in the k^{th} dataset than the n^{th} entity in the m^{th} dataset. For instance, if we define similarity with respect to an ℓ_p-norm, the above condition is equal to optimizing for $\|x_{ij} - x_{kl}\|_p < \|x_{ij} - x_{mn}\|_p$.

3.2 Join Specification for Context Enrichment

We propose a minimal join specification for our applications using keyless joins as a building block (Listing 1). Given a pair of data sources to join (`base_table_ref`, `aux_table_ref`), and examples

```
base_table_ref [join_type] "KEYLESS JOIN" aux_table_ref
"LEFT SIZE" integer "RIGHT SIZE" integer
"USING" supervision;

join_type = "INNER" | "LEFT" | "RIGHT" | "FULL";
```

Listing 1: Keyless join specification (inner join default).

of similar records across them (`supervision`), users first specify the `join type`: an inner join to only return enriched records, or outer join (left, right, full) to return enriched and unenriched records. Users then specify the `join size`: how many records from one data source joins to a single record in the other data source to indicate one-to-one, one-to-many, or many-to-many semantics.

As output, joined tuples (matches) between the two tables are the most similar records across them as learned using the keyless join objective. Results are ranked in order of greatest entity similarity, and the top k results are returned based on `join size`.

For instance, an entity matching application is written as follows to retrieve a single matching record between each data source:

```
entity_mentions_A INNER KEYLESS JOIN entity_mentions_B
LEFT SIZE 1 RIGHT SIZE 1 USING matching_mentions;
```

A search application is written as follows to retrieve 10 documents for each search query, else return the unenriched query:

```
query_corpus LEFT KEYLESS JOIN document_corpus
LEFT SIZE 1 RIGHT SIZE 10 USING relevant_docs_for_query;
```

In the remainder of paper, we describe our prototype system that implements this keyless join abstraction layer for enrichment.

3.3 Background: Transformers and EM

Prior work shows that deep learning models with attention mechanisms can replicate manually-generated rules to perform tasks such as EM in under a day of training, saving months of expert time [58]. As such, we use Transformers [79] to optimize for the objective in Section 3.1, which rely on attention mechanisms to better capture domain-specific semantic information for representation learning.

Transformers have demonstrated wide success across structured, semi-structured, and unstructured domains [24, 55, 66, 77, 86]: Neural Databases utilize Transformers for OLAP over natural language [77]; TaBERT focuses on natural language understanding to answer text queries over tabular data [86]; PicketNet addresses tabular data corruption [55], while TabNet aims to provide interpretable ML models for prediction and regression over tabular data [11]. Just as these methods use Transformers to capture semantic information in structured datasets, we propose a context enrichment architecture powered by Transformers. As we focus on primarily textual datasets, Ember uses BERT-based embeddings [24].

As noted in Section 2.2, context enrichment is prevalent across many domains. Recent work in EM such as AutoBlock [90], BERT-ER [49], DeepBlocker [51], BLINK [83], and DeepER [27] is most related to ours. These methods share primitives such as hierarchical encoding structures [51, 90], attention mechanisms tailored for alignment of tabular attributes [17, 58, 83, 90], hashing or nearest-neighbor retrieval [27, 49, 51, 90], and EM-specific feature engineering and supervision [51, 53]. Unfortunately, isolating and reusing enrichment operators (e.g., blockers) from these systems is challenging due to their end-to-end focus (see Section 9.3). Thus, we propose

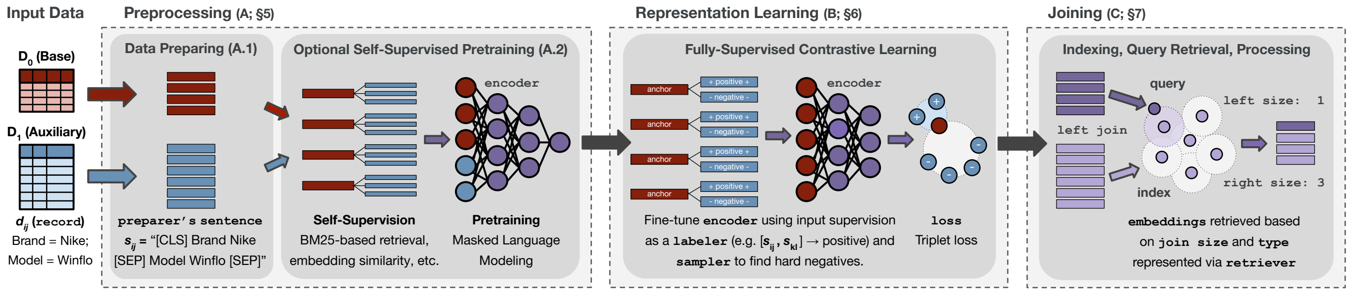

Input Data
Preprocessing (A; §5)
Representation Learning (B; §6)
Joining (C; §7)
D0 (Base)
D1 (Auxiliary)
dij (record)
Brand = Nike;
Model = Winflo
Data Preparing (A.1)
preparer's sentence
sij = "[CLS] Brand Nike [SEP] Model Winflo [SEP]"
Optional Self-Supervised Pretraining (A.2)
encoder
Self-Supervision
BM25-based retrieval, embedding similarity, etc.
Pretraining
Masked Language Modeling
Fully-Supervised Contrastive Learning
anchor
+ positive +
- negative -
encoder
Fine-tune encoder using input supervision as a labeler (e.g. [sij, skl] → positive) and sampler to find hard negatives.
loss
Triplet loss
Indexing, Query Retrieval, Processing
query
left size: 1
left join
index
right size: 3
embeddings retrieved based on join size and type represented via retriever

Figure 3: Ember system architecture.

Table 1: Ember data types and operators used in each step

Data Types		
`record`	`Dict[str, Union[str, int, float]]`	
`sentence`	`str`	
`embedding`	`List[float]`	
Operators		**Step**
`preparer`	`record` ⟶ `sentence`	1
`labeler`	`Tuple[record, record]` ⟶ `bool`	1, 2
`sampler`	`List[Generic[T]]` ⟶ `Generic[T]`	2
`loss`	`Tuple[emb., Union[emb., Tuple[emb., emb.]]]` ⟶ `float`	1, 2
`encoder`	`sentence` ⟶ `embedding`	1, 2
`retriever`	`List[emb.]` ⟶ `List[Tuple[record, emb.]]`	3

an extensible framework to reproduce these methods with minimal overhead: developing a new method should *only* require the construction of new *EM-specific* operators (as defined in Table 1).

4 EMBER

We develop Ember,[3] an open-source system for no-code context enrichment. Ember implements a keyless join abstraction layer that meets the specification in Section 3.2. Ember first represents input records using transformer-based embeddings directly optimized for the condition in Section 3.1. Ember then populates a reusable index with these embeddings based on the `join type`, and configures index retrieval based on the `join sizes`. We now provide an overview of Ember's usage, API and architecture; data types and operators are in Table 1, with an architecture overview in Figure 3.

4.1 Usage

As input, Ember requires: a base data source D_0 ("left"), auxiliary data sources $D = \{D_1, ..., D_M\}$ ("right"), labeled examples of related data points, `join type`, and `join sizes`. Labeled examples are provided in one of two forms: pairs of related records, one from each table, or a triple where a record from D_0 is linked to a related and unrelated record from each D_i. That is, unrelated examples are optional, but related ones are required. Recall that we focus on the $M = 1$ case (see Section 8 for extensions). As output, Ember retrieves enriched records based on the `join type` and `join sizes` as a list of tuples. Users can either configure Ember using a json configuration file or a SQL-like API (see Sections 3.2 and 9.6).

[3]https://github.com/sahaana/ember

4.2 Architecture and Operator Overview

We propose a modular system with three architectural elements to enable general keyless: data preprocessing, representation learning, and data joining. Ember consists of dataflow operators that transform inputs into the formats required for each step (see Table 1).

Ember represents input data records d_{ij} as `records` in key-value pairs. Supervision is represented via `labelers`. An Ember pipeline consists of `preparers`, `encoders`, `samplers`, `losses`, and `retrievers`. Ember uses `preparers` to transform `records` into `sentences`. Ember uses `encoders` to transform `sentences` into `embeddings` that are optimized per a provided `loss`; `samplers` mine negative examples if the provided supervision only provides examples of related records. The trained `embeddings` are stored in an index, and are retrieved using a `retriever`. We provide an API over these operators in the form of a high-level configuration file (see Sections 3.2 and 9.6). Users may use Ember's pre-configured defaults (described below) or implement custom `preparer`, `encoder`, `sampler`, or `retriever` operators as needed (e.g., as in Section 8).

Preprocessing (Figure 3A, §5). Ember ingests any text or numeric data source with a pre-defined schema, and converts its records to a common representation. By default, Ember converts `records` to `sentences` using `sentence preparer` modules, which are fed as input to the pipeline's `encoders`. Ember optionally pretrains the `encoders` in this step via self-supervision, without labels.

Representation Learning (Figure 3B, §6). Ember learns a mapping for each input data source's `records` such that the transformed data is clustered by relatedness. To learn this mapping, Ember's `encoders` are fine-tuned with the input supervision (in the form of a `labeler`) and `loss`. Ember applies the learned mapping to each of the `sentences` to generate `embeddings` passed to the final step.

Joining (Figure 3C §7). Ember populates an index with the learned `embeddings` using Faiss [38]. A keyless join can be completed by issuing a similarity search query given an input `record` (transformed to an `embedding`) against this index. The dataset that is indexed is determined by the `join type`. Ember uses a k-NN `retriever` module to retrieve as many records specified by the `join sizes`.

5 PREPROCESSING

We describe the first step of Ember's pipeline: preprocessing. Ember processes the base and auxiliary input datasets so all records are represented in a comparable format. Ember then optionally pretrains the `encoders` for the representation learning step.

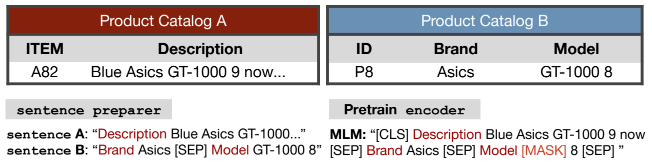

Product Catalog A
ITEM
Description
A82
Blue Asics GT-1000 9 now...
Product Catalog B
ID
Brand
Model
P8
Asics
GT-1000 8
sentence preparer
sentence A: "Description Blue Asics GT-1000..."
sentence B: "Brand Asics [SEP] Model GT-1000 8"
Pretrain encoder
MLM: "[CLS] Description Blue Asics GT-1000 9 now [SEP] Brand Asics [SEP] Model [MASK] 8 [SEP] "

Figure 4: Examples of the data preparing and pretraining phases of the preprocessing architecture step.

Data Preparing (Figure 3A.1). This phase converts each `record` d_{ij} into a format that can be used for downstream Transformer-based `encoders` regardless of schema. Ember's default pipeline uses a `sentence preparer` P to convert input `records` d_{ij} into `sentences` $P(d_{ij}) = s_{ij}$. Each key-value pair in d_{ij} is converted to a string and separated by the `encoder`'s separator token as "`key_1 value_1 [SEP] key_2 value_2 [SEP]...`" (Figure 4).

Optional Pretraining (Figure 3A.2). An out-of-the-box BERT-based `encoder` is pretrained over natural language corpora (BookCorpus and Wikipedia [9, 92]). As structured data is often domain-specific, and of a different distribution than these corpora, we find that bootstrapping Ember's `encoders` via additional self-supervised pretraining improves performance by up to 2.08×. Ember provides and enables a pretraining configuration option by default.

With this option, users can pretrain the pipeline's `encoders` via BM25-based self-supervision that does not use any user-provided labels (option enabled by default). We add a standard Masked Language Modeling (MLM) head to each pipeline `encoder`, and pretrain it following the original BERT pretraining procedure that fits a reconstruction loss. To encourage the spread of contextual information across the two tables, we concatenate one `sentence` from each table to one other as pretraining input as "s_{ij} `[SEP]` s_{kl}" (Figure 4). We select `sentence` pairs that are likely to share information in an unsupervised manner via BM25, a bag-of-words relevance ranking function [71], though any domain-specific unsupervised similarity join can be used, such as from Magellan [45] or AutoFJ [52].

6 REPRESENTATION LEARNING

We describe the second step of Ember's pipeline: representation learning. Given the `sentences` and pretrained `encoders` from the first step, Ember uses labeled supervision to fine-tune the `encoders` such that `embeddings` from related records are close in embedding space. These `embeddings` are passed to the final step.

`Encoder` configuration. Ember either trains several `encoders` independently for each data source, or a single `encoder` common to all data sources. We found that using a single `encoder` always performs best, and is Ember's default configuration. We observe that using separate encoders perturbs the representations such that exact matches no longer share a representation—thus, `encoders` must relearn correspondences, and often fail to (see Section 9.4).

`Encoder` architecture. Ember lets users configure each `encoder`'s base architecture and the size of the learned `embeddings`. Users can choose $\text{BERT}_{\text{base}}$ or $\text{DistilBERT}_{\text{base}}$ as the core architecture; Ember's default is the 40% smaller, and 60% faster DistilBERT model. We use models from HuggingFace Transformers [82]; thus, integrating new architectures requires few additional lines of code.

We remove the MLM head used for optional pretraining, and replace it with a fully connected layer that transforms BERT's output to a user-specified `embedding` dimension, d (default: 200). The output of the fully connected layer is a d-dimensional embedding for each input token. Users can choose one of two aggregation methods that will return a single d-dimensional output per input `sentence`: averaging all of the embeddings, or using the embedding of BERT's leading `CLS` token (default: `CLS`-based aggregation).

`Encoder` training. The `encoders`' goal is to learn a representation (`embedding`), for the data sources such that `sentences` that refer to similar entities are grouped in the underlying embedding space. Recall that to perform a keyless join under an ℓ_p-norm, each `encoder` must learn a function that maps elements of D_i to the space $\mathbf{X}$, such that $\|x_{ij} - x_{kl}\|_p < \|x_{ij} - x_{mn}\|_p$ when the j^{th} entity in the i^{th} dataset is more closely related to the l^{th} entity in the k^{th} dataset than the n^{th} entity in the m^{th} dataset. We directly optimize for this objective function by training `encoders` using a contrastive, triplet loss together with user-provided examples of related `records`.

Given an anchor `record` $d_{0\text{a}}$ from D_0, and `records` $d_{1\text{p}}$ and $d_{1\text{n}}$ in D_1 that are related and unrelated to $d_{0\text{a}}$, respectively, let $F_0(d_{0\text{a}}) = x_{0\text{a}}$, $F_1(d_{1\text{p}}) = x_{1\text{p}}$, and $F_1(d_{1\text{n}}) = x_{1\text{n}}$ be their `embeddings` following the composition of the `sentence preparer` P and `encoder` E_i operators ($F_i = E_i \circ P$). We minimize the triplet loss, defined as:

$$\mathcal{L}(x_{0\text{a}}, x_{1\text{p}}, x_{1\text{n}}) = \max\{\|x_{0\text{a}} - x_{1\text{p}}\|_p, \|x_{0\text{a}} - x_{1\text{n}}\|_p + \alpha, 0\}$$

where α is a hyperparameter controlling the margin between related and unrelated `embeddings`. Users could use an alternative `loss` function: a cosine similarity loss would provide representations that similarly encourage related points to be close and unrelated points to be far, and a standard binary cross-entropy loss may be used as well, where the final layer in the trained `encoder` would be the end embeddings. However, they do not explicitly optimize for *relative* distances between related and unrelated records.

Users often have examples of related pairs, but not triples with unrelated examples: in our tasks, only the search workload provides triples. Thus, we provide negative `sampler` operators to convert `labelers` that operate over pairs to triples. If the user-provided `labeler` contains a related and unrelated example for a `record`, we form a triple using these examples. Otherwise, for each `record` d_{0j} in D_0 with a labeled related record, we use a random or stratified hard negative `sampler`. The random `sampler` selects an unlabeled `record` at random as an example unrelated `record`. The stratified `sampler` weights the degree of relatedness for the negative examples—users can specify a region of hard examples to sample from. We provide a default stratified `sampler` with a single tier of relatedness defined via BM25 or Jaccard similarity.

7 JOINING

We describe the last step of Ember's pipeline: joining. Given the `embeddings` output by the trained `encoders` and user-provided `join type` and `join sizes`, Ember executes the keyless join by identifying related points across the input datasets, and processing them for downstream use. Ember indexes the learned `embeddings` and queries this index to find candidate related records

Indexing and Query Retrieval. Given two records, Ember computes their `embedding` similarity to determine if they are related.

Traditional solutions to our motivating applications perform such pairwise checks across all possible pairs either naïvely or with blocking [58, 60, 62] to identify related records. However, the operator choice both domain-specific, and scales quadratically at query time with the dataset size and blocking mechanisms used (see Section 9.3). We eliminate the need for pairwise checking by indexing our embeddings to cluster related records, and relying on efficient libraries for maximum inner product search (MIPS) [38]—the indexing mechanism used can be optimized based on downstream needs, and impacts both speed and accuracy, as evaluated in Section 9.4.

For a `LEFT` or `RIGHT OUTER JOIN`, Ember constructs an index over the base or auxiliary dataset, respectively. Ember then queries the index with each `embedding` of the remaining dataset, and returns the `record` and `embedding` corresponding to the most similar records using a `retriever` operator. For a `FULL OUTER` or `INNER JOIN`, Ember may jointly index and query both datasets to identify related entries in either direction. By default, Ember only indexes the larger dataset—an optimization that improves query runtime by up to 2.81× (see Section 9.4). Ember's default configuration is an `INNER JOIN`, which we justify via user study in Section 9.6.

Post-Processing. The user-provided `join size` configures the `retriever` to return the top-k records with the closest embeddings in the indexed dataset. Ember additionally supports threshold-based retrieval. The former is useful for applications such as search, where a downstream task may be to display the top-k search results to a user in sorted order. The latter is useful for applications where there may not be related records for each record in the base dataset.

8 EXTENSIONS AND LIMITATIONS

In this section, we discuss extensions and limitations of Ember.

`labeler` Extension. Representation learning requires labeled examples of related pairs. For many complex, real-world tasks, labeling teams typically require extensive domain expertise to accurately label data points [68]. However, such expertise may already be present in the form of existing employees or organizational resources [75] such as click logs, purchase history, and conversion rates—as was used to generate the MS Marco dataset [13]—or can be obtained from an initial unsupervised method [51].

When passive supervision is unavailable, `labelers` that allow users to write heuristic rules (based on embedding similarity or attribute values) is a powerful first step to identify candidate pairs prior to manual identification [75], or to use directly as weak supervision [68]. These rules need not perfectly capture all positive examples, else rule creation would be equivalent to the task.

`encoder` Extension. Ember defaults to a single `encoder` architecture. Users can implement source-specific `encoders` to extend to additional domains where Transformers have demonstrated success, including images [26, 63], video [30, 43, 74], audio [23, 35, 89], and cross-modal settings [36, 37, 65]. Beyond this, developing systems that jointly optimize for keyless joins over such attributes with traditional SQL workloads is an exciting area of future work.

Multi-Dataset Extension. There are two common multi-dataset scenarios: (1) a base table D_0 must be augmented with several auxiliary data sources D_i as in Figure 1, or (2) data sources must be aligned in sequence, as described in Section 2.2 for multi-hop QA.

In either case, the `sentence` preprocessing step remains unchanged. Depending on the encoder configuration, BM25-based pretraining will be applied to all `sentences` across all datasets at once (anchored from the base table), or each dataset pairwise.

For `encoder` training, in the first case, each `encoder` can be trained pairwise for D_0 and each D_i. In the second, a user can (1) sequentially apply Ember over a manually-constructed data DAG, or (2) traverse the space of possible keyless joins using data discovery methods [21] or record cardinality, in ascending order.

For the joining step, in the first case, each data source D_i is indexed, and the retrieval phase will query each index to return candidate related records for each pairwise interaction. In the second case, Ember must chain routines by querying the next index with the records retrieved by the previous routine; we provide an example of this scenario in Section 9.5 under Recommendation.

Limitations. Ember requires sufficient context across data sources to learn record relatedness—either directly, or sequentially in a multi-dataset setting. A first scenario is where information is dense, but not explicitly correlated with one other (i.e., even a human would struggle to identify related records). A second is where information is extremely sparse: related fields are either rarely present or consist of a few categorical values drawn from a high cardinality space. Examples of the first scenario include many pairs of the New York City public datasets [7] that lack join keys; in these cases, Ember will often cluster records by borough, year, or other spurious correlations present in the fields—which may reveal interesting trends to data analysts. An example of the second scenario is to join Yelp's public dataset [10] with another dataset while requiring rarely present business attributes or categories—Ember will heavily weight common attributes, and underrepresent tail attributes.

9 EVALUATION

In this section, we demonstrate that Ember and its operators are:

(1) **General**: Ember enables context enrichment for effective preprocessing across five domains, while meeting or exceeding similarity join baseline recall (Section 9.3).
(2) **Extensible**: Ember provides a modular architecture, where each component affects performance (Section 9.4). Ember enables task-specific pipelines for various similarity-based queries, and provides task performance that can be fine-tuned by state-of-the-art systems (Section 9.5).
(3) **Low Effort**: Ember requires no more than five configuration changes (Table 2) from its default, and does not always require large amounts of hand-labeled examples (Section 9.6).

9.1 Evaluation Metric and Applications

Evaluation Metric. A context enrichment system does not seek to achieve state-of-the-art performance on end-to-end ML workloads, but must identify all related records across a base and auxiliary dataset. This is equivalent to maximizing record-level recall, or the fraction of records for which we recover *all* related records. While naïvely optimizing for recall would return *all* records as being related to one another, a precision metric greatly drops when we retrieve multiple records. Thus, our evaluation metric is **recall@k**, for small `join size` (k) to minimize downstream processing cost.

Table 2: Data source record count, related pair (supervision) count, and number of configuration lines changed for the best recall@k, and for each downstream task in parentheses.

Task	Dataset	# base	# aux	# + train	# + test	# LoC
FJ	IMDb	50000	10000	40000	10000	1
FJ	IMDb-hard	50000	10000	40000	10000	1
QA	SQuAD	92695	64549	86668	6472	2
S	MS MARCO	508213	8.8M	418010	7437	1
R	IMDb-wiki	47813	47813	38250	9563	1 (6)
EM-T	Abt-Buy	1081	1092	616	206	2
EM-T	Company	28200	28200	16859	5640	2
EM-S	BeerAdvo-RateBeer	4345	3000	40	14	1
EM-S	iTunes-Amazon	6907	55923	78	27	1
EM-S	Fodors-Zagat	533	331	66	22	1
EM-S	DBLP-ACM	2616	2294	1332	444	1
EM-S	Amazon-Google	1363	3226	699	234	1
EM-S	DBLP-Scholar	2616	64263	3207	1070	1
EM-S	Walmart-Amazon	2554	22074	576	193	1
EM-D	DBLP-ACM	2616	2294	1332	444	1
EM-D	DBLP-Scholar	2616	64263	3207	1070	1
EM-D	iTunes-Amazon	6907	55923	78	27	1
EM-D	Walmart-Amazon	2554	22074	576	193	1

Applications. We evaluate Ember against workloads from the five application domains summarized in Table 2 and described in detail below. All of the datasets used are publicly available online.

9.1.1 Fuzzy Join (FJ). We construct two workloads using a dataset and generation procedure from a 2019 scalable fuzzy join VLDB paper [20]. The base dataset consists of the Title, Year, and Genre columns from IMDb [5]. The auxiliary dataset is generated by perturbing each row in the first by applying a combination of token insertion, token deletion, and token replacement. The task T, is to join each perturbed row with the row that generated it. We generate two workload versions: 5 perturbations per row (IMDb) and 15 perturbations per row (IMDb-hard), up to 25% of the record length. We randomly sample 10,000 movies and generate 5 perturbed rows for each. We hold out 20% of records as the test set; no records from the same unperturbed record are in both the train and test sets.

9.1.2 Entity Matching (EM). We use all 13 benchmark datasets [1] released with DeepMatcher [58], spanning structured (EM-S), textual (EM-T), and dirty (EM-D) entity matching. The base and auxiliary datasets share the same schema. In EM-T, all data records are raw text entries or descriptions. In EM-S, each table follows a pre-defined schema, where text-based column values are restricted in length. In EM-D, records are similar to EM-S, but some column values are injected into the incorrect column. The task T is to label a record pair as representing the same entity or not. Train, validation, and test supervision are lists of unrelated, and related pairs—we only use the related pairs for training Ember, as we identified mislabeled entries (false negatives) when using Ember to explore results.

9.1.3 Search (S). We use the MS MARCO passage retrieval benchmark [13]. MS MARCO consists of a collection of passages from web pages that were gathered by sampling and anonymizing Bing logs from real queries [13]. The task T is to rank the passage(s) that are relevant to a given query as highly as possible. Supervision is a set of 397M triples, with 1 relevant and up to 999 irrelevant passages for most queries. Irrelevant passages are retrieved via BM25. We report results over the publicly available labeled development set.

9.1.4 Question Answering (QA). We modify the Stanford Question Answering Dataset (SQuAD) [67]. SQuAD consists of Wikipedia passages and questions corresponding to each passage. The task T is to identify the beginning of the text span containing the answer to each question. As described in Section 2, a retriever module, used in retriever-reader models for QA [91], performs context enrichment. We modify SQuAD by splitting each each passage at the sentence-level, and combining these sentences to form a new dataset. The modified task is to recover the sentence containing the answer.

9.1.5 Recommendation (R). We construct a workload using IMDb and Wikipedia to mimic the e-commerce example in Figure 1. For the base dataset, we denormalize four IMDb tables that provide movie title, cast/crew, and rating: title.basics, title.principals, name.basics, and title.ratings [5]. For the auxiliary dataset, we query the latest Wikipedia snapshot for each IMDb movie's summary paragraphs; we extract 47813 overlapping records [9]. We remove the join key (IMDb ID) to induce a need for keyless joins. This workload enables two applications. In Application A, we show that Ember can join datasets with dramatically different schema. In Application B, we show Ember can perform similarity-based analyses to estimate the rating for a new movie. Supervision is provided as exact matches and movie ratings with an 80-20 train-test set split.

9.2 Experimental Setup

Baselines. We first evaluate all workloads compared to ten similarity join baselines with respect to recall@1 and recall@10 in Section 9.3. For EM and search, whose downstream task can be satisfied with Ember's output, we compare downstream performance with existing benchmark solutions [6, 27, 53, 58, 60]. The remainder of our workloads were constructed to isolate context enrichment from the downstream task, and do not have standard baselines.

Ember Default Configuration. Tasks use a `sentence preparer`, and perform 20 epochs of self-supervised pretraining. Ember uses a single DistilBERT$_{\text{base}}$ `encoder` trained with a triplet `loss` and stratified hard negative `sampler`. Self-supervision and the `sampler` use BM25 to identify similar `sentences` to concatenate from each data source, and to mark as hard negatives, respectively. The output `embedding` size is 200, and models are trained with a batch size of 8 using an ADAM optimizer [44] with initial learning rate of 1e−5. The default `join sizes` are 1 and 10. Results are five-trial averages.

Implementation. We use two Intel Xeon Gold 6132 CPUs (56 threads) with 504GB of RAM, and four Titan V GPUs with 12GB of memory. We implement Ember in Python and PyTorch [64] using HuggingFace Transformers [82]. For exact (default) and approximate MIPS indexing, we use Faiss [38]. We use Magellan [45], rank-BM25 [8] and AutoFJ [52] for similarity join baselines, and BERT-ER [49] code from the authors, with our custom processing. We implement DeepBlocker [51] with FastText [39] and Faiss [38].

9.3 Generalizability

We show that Ember's recall and runtime meets or outperforms that of the following similarity join baselines in nearly all tasks:

Levenshtein-Distance (LD). LD is the number of character edits needed to convert one string to another. This join returns the closest

Table 3: Baseline comparison, where we highlight all methods within 1% of the best for each task in blue. No single baseline dominates others, but Ember (EMB) is competitive with or better than alternatives in nearly all tasks with respect to Recall@k.

		Recall@1								Recall@10							
Task	Dataset	**LD**	**J-2G**	**JK-2G**	**JK-ws**	**J-ws**	**BM25**	**AGG**	**EMB**	**LD**	**J-2G**	**JK-2G**	**JK-ws**	**J-ws**	**BM25**	**AGG**	**EMB**
FJ	IMDb	**99.32**	**100.00**	**99.42**	71.54	91.61	91.81	82.77	97.86	**99.99**	**100.00**	**99.96**	96.13	96.30	96.33	89.83	**99.79**
FJ	IMDb-hard	95.54	**98.87**	95.02	33.42	52.29	54.58	31.88	88.92	**99.04**	**99.82**	98.26	42.68	64.34	65.82	46.62	98.37
QA	SQuAD	1.37	34.70	41.26	29.47	27.26	49.17	11.37	**52.91**	1.50	52.82	61.18	44.79	43.18	67.08	27.03	**78.85**
S	MS MARCO	0.26	1.66	1.66	1.15	1.15	2.31	0.10	**16.34**	0.96	7.38	7.38	5.11	5.11	4.10	0.19	**46.98**
R	IMDb-wiki	58.96	11.01	82.57	86.08	18.30	63.25	0.22	**97.02**	64.62	18.80	93.12	91.93	18.30	96.25	0.86	**98.89**
EM-T	Average	18.20	20.06	36.56	36.90	37.21	61.09	6.88	**71.93**	25.73	37.57	49.00	56.15	53.11	71.90	21.45	**83.29**
EM-S	Average	52.26	73.50	72.30	69.25	72.71	**76.59**	38.98	**77.20**	70.26	94.41	**97.90**	93.51	93.33	95.19	55.88	**97.58**
EM-D	Average	18.70	64.69	52.05	53.41	61.94	65.47	30.61	**66.56**	24.60	90.86	80.92	79.41	89.85	92.31	43.92	**97.83**

records with respect to single-character edits over provided key columns. We filter and only return results under a 30 edit threshold.

Jaccard-Similarity, Specified Key (JK-WS, JK-2G). Defining a Jaccard-similarity based join over textual inputs requires tokenizer selection. We consider a whitespace tokenizer (WS) and a 2-gram tokenizer (2G) to capture different granularities. JK-WS and JK-2G return the closest records with respect to Jaccard similarity over provided key columns using a WS or 2G tokenizer. We set a filtering threshold to return results with at least 0.3 Jaccard similarity.

Jaccard-Similarity, Unspecified Key (J-WS, J-2G). J-WS and J-2G return the closest records with respect to Jaccard similarity using a WS or 2G tokenizer, after a `sentence preparer`. We set a filtering threshold to return results with over 0.3 Jaccard similarity.

BM25 (BM25). BM25 is a bag-of-words ranking function used in retrieval [71]. This join returns the closest records with respect to the Okapi BM25 score using default parameters k1=1.5 and b=0.75.

Pretrained-Embedding (BERT). BERT generates embeddings for each prepared `sentence` via a pretrained DistilBERT$_{\text{base}}$ model, and returns the closest records based on the ℓ_2-norm between them.

Auto-FuzzyJoin [52] (AutoFJ). AutoFJ automatically identifies join functions for unsupervised similarity joins. Users must align *matching columns*, which do not always exist in our tasks (e.g., IMDb-wiki): in such cases, we preprocess `records` with a `sentence preparer`. AutoFJ optimizes for a precision target, and does not expose `join sizes`: we set a low precision target of 0.5, and find that a single result is typically returned per record. AutoFJ assumes one input is a "reference table" with few duplicates, which does not always hold in our tasks. Precision estimation may break down if duplicates are present, or if reference table records are clustered (Ember trivially accounts for these cases). As AutoFJ exhaustively computes record similarities for each considered join configuration, we discuss workloads that completed in 1.5 days on our machines. This means we omit results for MS MARCO, EM-T Company, EM-S BeerAdvo-RateBeer, and others with large text spans.

DeepBlocker [51] (AGG). DeepBlocker evaluates self-supervised deep-learning-based blocking methods tailored for EM, and outperforms AutoBlock [90] and DeepER [27]. DeepBlocker tokenizes and generates FastText word embeddings [39], then aggregates them into tuple embeddings prior to a joining step. As basic methods like SIF [12] outperform DeepER, the authors do not include comparison to it, and show that their Hybrid and AutoEncoder methods outperform AutoBlock. Consequently, we implement the Average, SIF, and AutoEncoder methods by following the descriptions in their paper. While they provide self-supervised methods with higher model capacity, these methods rely on EM-specific semantics that do not apply to general enrichment. As BERT similarly performs tuple-level aggregation and indexing, we refer to the best performing method across Average, SIF, AutoEncoder, and BERT as AGG.

BERT-ER [49]. BERT-ER tackles end-to-end EM by jointly training a BERT-based blocker and matcher via multi-task learning. We implement and evaluate BERT-ER, but find that the blocker in isolation performs worse than alternatives, and do not report our results.

For multi-column datasets, we provide a plausible key column if the method requires. In FJ, EM, and R, this is a title or name column.

We now show that Ember is competitive with or exceeds alternatives with respect to Recall@1 and Recall@10. In Table 3, we highlight the methods that are within 1% of the top performing method. No baseline dominates the others across the different workloads. Broadly, character-based joins (LD, JK-2G, J-2G) tend to perform well in scenarios where a join key exists but may be perturbed (FJ, EM-S, EM-D); word-based joins (J-WS, BM25) tend to perform well in scenarios with word-level join keys (EM-S), or where a join key does not exist but common phrases still link the two datasets (R, EM-D). Neither performs well otherwise (S, EM-T). We find that unsupervised tuple-aggregation methods (AGG) cannot tune attribute weights for a specific domain or task, so underperform naive yet exhaustive baselines. As noted in [51], their performance increases significantly as `join size` (k) increases by 2-10×, but this is not our target operating regime. However, they terminate orders of magnitude faster that alternatives—even MS Marco with 8.8M records required on the order of minutes, not hours (see Section 9.4.8). Ember most consistently outperforms or is comparable with alternatives across the tasks: a learned approach is necessary for generalization. The exception is with the synthetic FJ workload that is, by construction, suited for character-based joins.

We do not report AutoFJ in Table 3 due to incompleteness. AutoFJ is tailored for the EM and FJ: many-to-one joins with identical schema. Of those completed, Ember is sometimes outperformed by up to 8% recall@1 (Amazon-Google), but is otherwise comparable or far exceeds AutoFJ when considering recall@10. AutoFJ's recall is 37.9% for QA, and 38.5% for R. As AutoFJ over the Wikipedia summary (task R) did not terminate, we only use the title column.

Ember's strong performance is due to the Transformer-based encoders' ability to capture semantic meaning across the input tables [16, 78], trained via contrastive learning. Prior work in general and automated similarity joins relies on static similarity and distance measures [45, 52], or are tailored to EM workloads [49, 51, 90]. Thus, Ember is able to extend to notions of fuzziness or similarity that are beyond the classically evaluated typos, matching key phrases or terms, or missing or out-of-order attributes [51–53].

9.4 Extensibility: Architecture Lesion

We demonstrate modularity and justify our architecture by evaluating Ember's performance when modifying each system component as follows. Each requires a single line configuration change:

(1) Replacing MLM pretraining and representation learning (RL) with a pretrained `encoder`.
(2) Removing RL, leaving only MLM pretraining.
(3) Replacing the fine-tuned Transformer model with a frozen pretrained embedding and a learned dense layer.
(4) Removing MLM pretraining.
(5) Random negatives compared to hard negative sampling
(6) Using an `encoder` for each dataset versus a single `encoder`.
(7) Removing the joining index-query optimization.
(8) Replacing MIPS-based indexing routines in the joining step.

9.4.1 Pretrained `encoder` *(Figure 5, -mlm, rl).* We eliminate all domain-specific training by replacing MLM pretraining and RL with a pretrained DistilBERT$_{\text{base}}$ encoder—the same as BERT in Section 9.3. Ember's performance dramatically declines, by up to three orders of magnitude (e.g., recall dropped to 0.04 in R). This approach is only feasible when both datasets' contents are near identical, as in certain EM-S and EM-D tasks, and FJ-IMDb.

9.4.2 Removing Representation Learning (Figure 5, -rl). We eliminate all fully supervised learning by only pretraining an `encoder` using the self-supervised BM25-based procedure from Section 5, with no subsequent RL step. Ember's performance declines by up to an order of magnitude, though it outperforms -mlm, rl on tasks whose datasets do not match the original natural text Wikipedia corpus (e.g., EM-S, FJ). Primarily textual datasets do not see as large an improvement over -mlm, rl, and for QA, where the corpus is derived from Wikipedia, -rl performs slightly worse.

9.4.3 Remove Transformer fine-tuning (Figure 5, -ft). We eliminate fine-tuning of our Transformer-based `encoders`. We replace RL with *frozen* pretrained embeddings, followed by a learned fully-connected layer (i.e., we add and train a fully-connected layer on top of -rl). This method slightly outperforms using a fully pretrained encoder (-rl) in text-heavy workloads (QA, S, R, EM-T), as the pretrained encoder provides some semantic information out-of-the-box, which the learned final layer can reweight. Although we do not perform an exhaustive comparison of pretrained embeddings or non-Transformer-based architectures, this demonstrates that Ember captures semantic information to provide a strong baseline.

9.4.4 Remove MLM Pretraining (Figure 5, -mlm). We eliminate self-supervised `encoder` bootstrapping by removing MLM pretraining from the pipeline. We find that Ember meets or exceeds this setting in all but QA, by a significantly smaller margin than previous experiments (up to 20.5% in IMDb-hard). MLM pretraining is ineffective for QA as the corpus is drawn from Wikipedia—one of BERT's training corpora—and both base and auxiliary datasets consist of the same, primarily textual vocabulary. In such cases, users may wish to disable MLM pretraining, although performance is not significantly impacted. In contrast, of the non-EM tasks, MLM pretraining is most helpful for FJ IMDb-hard, where random perturbations result in many records with words that are not in the original vocabulary.

9.4.5 No Negative Sampling (Figure 5, -NS). We replace hard negative sampling with random sampling of unrelated records. Hard negative sampling tends to improve performance by a similar margin as MLM pretraining, and only negatively impacted the EM-T Company dataset (by up to 8.72% absolute recall). We find larger improvements when the join condition is more ambiguous than recovering a potentially obfuscated join key, as in tasks S (30% R@1), QA (15% R@1), and EM-D (15% R@1). We use BM25-based sampling for all but FJ and QA, where we have prior knowledge, and develop custom samplers based on Jaccard similarity, and sentence origin, respectively. This improved performance over the BM25-based sampler by 1% absolute recall for FJ, and 8.7% and 5.6% absolute recall@1 and recall@10, respectively, for QA.

9.4.6 `encoder` *Configuration (Figure 5, te).* We use two `encoders`, one for each dataset, instead of our default single `encoder` configuration. We find that using two `encoders` performs up to two orders of magnitude worse than using a single encoder, especially for strictly structured datasets. We observe that through the course of the `encoder` training procedure, the performance of using two identically initialized `encoders` often *degrades*—inspecting the resulting embeddings when running Ember with identical tables shows that the same terms diverge from one another while training.

9.4.7 Index Optimization. For `INNER` and `FULL OUTER JOINs`, we optimize for join execution time by indexing the larger dataset to reduce the number of queries made by the system. We evaluate this optimization by running the joining step while indexing the larger dataset, and then the smaller dataset for *all* tasks. On average, this optimization reduces join time by 1.76×, and up to 2.81×. However, this optimization is only meaningful at large scale, as in the MS MARCO workload, where we save 2.7 minutes.

9.4.8 Indexing Alternatives. Optimized MIPS improves joining runtime (after model training) by up to two orders of magnitude. Even with CPU-only Faiss, Ember takes 7.24s on average for the joining step. MS MARCO had the longest MIPS runtime of 1.97m when indexing all 8.8M records; excluding it reduces the average to 0.31s. Embedding the query table prior to indexing takes 24.52s on average. Excluding the slowest dataset (EM-T Company, 3.96m), reduces the average to 11.19s. The fastest non-MIPS baseline (JK-WS), took 5.02m on average; excluding MS MARCO (79.73m) reduces the average to 21.18s. The slowest method (LD), took 23.64m on average, with EM-T Company (the most expensive) requiring 321.98m.

We use Faiss with exhaustive search due to small query sizes. To evaluate the loss in recall from a more scalable approximate indexing algorithm, we apply Faiss' optimized approximate cell-probe method that indexes based on product quantization codes [38] to our largest workload, MS MARCO, with the suggested parameter settings. Query time is 691× faster (0.13s), with an extra 169s (18×) indexing overhead in exchange for an 8.62 and 27.8 point decrease in recall@1 and recall@10, respectively. Extending Ember as per Section 9.5 and optimizing parameters as in [42] can improve recall.

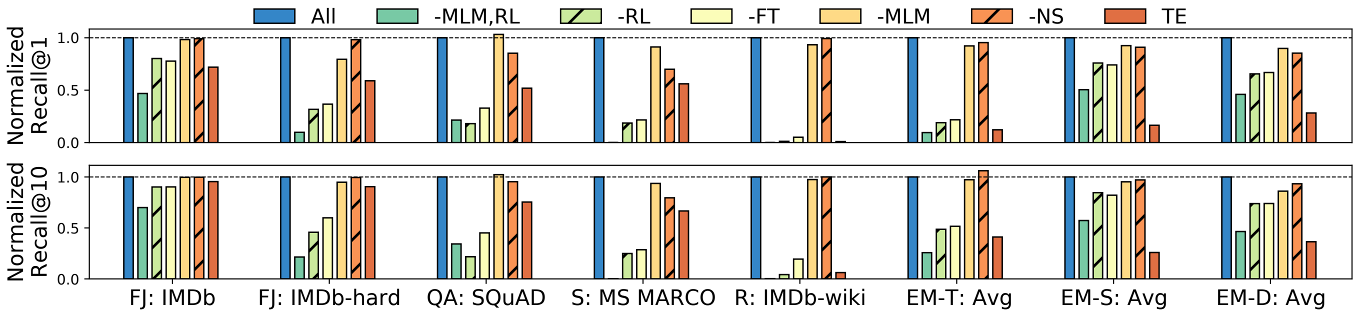


Figure 5: Architecture lesion displaying recall@k normalized to the default seen in Table 3 (All) for: a pretrained DistilBERT$_{\text{base}}$ model with no representation learning (-MLM,RL), MLM pretraining with no RL (-RL), RL with no Transformer fine tuning (-FT), RL with no MLM pretraining (-MLM), no hard negative sampling (-NS), and one encoder per dataset (TE).

9.5 Extensibility: End-to-End Workloads

We show how to extend Ember in an end-to-end context. For tasks whose downstream task (T) can be directly solved with Ember's outputs, we a provide comparison to state-of-the-art. We then demonstrate how to use Ember for more complex downstream tasks.

Entity Matching. EM systems focus on matcher performance in a two-part (i.e., blocker and matcher) end-to-end EM pipeline [27, 53, 58]: a matcher identifies if a given pair of candidate records correspond to the same entity, and is evaluated on model F1 score. To perform context enrichment to efficiently assist with end-to-end EM, Ember must retrieve candidate blocks with a low false negative rate so it can be followed by matchers; we verify this in Table 4 (Recall@10). Perhaps surprisingly, treating Ember results from a top-1 query as binary classifier achieves performance that is comparable to or better than previous, custom-built states-of-the-art with respect to F1 score (Table 4). As Ember is a general enrichment framework, implementing EM-specific operators for data augmentation and domain knowledge integration as in current state-of-the-art like Ditto [53] will allow Ember to bridge this gap.

Search. In MS MARCO passage ranking, results are ranked in relevance order, and rankings are evaluated via mean reciprocal rank (MRR) of the top 10 results. We rank Ember results based on their query distance, and compare MRR@10 with existing MS MARCO baselines. We obtain MRR@10 of 0.266 on the dev set after just 2.5M training examples, outperforming the official Anserini BM25 baseline solution of 0.167. Our results exceed K-NRM, a kernel based neural model, with a dev set MRR@10 of 0.218 [84], and is slightly below the state-of-the-art from the time, IRNet, with 0.278 MRR@10 [6]. For comparison, the first state-of-the-art BERT-based model uses BERT$_{\text{large}}$ to achieve MRR@10 of 0.365 after training with 12.8M examples [60], and current state-of-the-art is 0.439 [25]. By developing an additional joining operator, Ember can implement ColBERT [42], a previous state-of-the-art method that achieves 0.384 MRR@10 in MS MARCO document ranking that operates on larger input passages: we must remove the pooling step from Ember's `encoder` (1 line config change), and develop a `retriever` that indexes and retrieves bags of embeddings for each record.

Recommendation. In Application B of task R, we estimate the IMDb ratings of movies in the test set given a training dataset and Wikipedia corpus. We report the mean squared error (MSE) between the rating estimates and the true rating. The task is easy: predicting the average training rating returns 1.19 MSE, and a gradient-boosted decision tree (GBDT) over the joined data returns 0.82 MSE. We aim to show extensibility while meeting GBDT performance.

There are two approaches to enable this analysis with Ember: similarity defined in a single hop in terms of the labeled training data, or by first going through the Wikipedia data in two hops.

In the former, users configure Ember to index labeled IMDb training data (with ratings) and retrieve records related to the test data (without ratings) via an `INNER JOIN`. Users then average the returned `records`' labels. In the latter, users require two `LEFT JOIN`s that index the Wikipedia training data against the IMDb test dataset, and the IMDb train dataset against the Wikipedia training data. Users retrieve the closest Wikipedia summary to each test IMDb record, and then retrieve the closest labeled IMDb instances to each retrieved Wikipedia summary, whose scores are averaged.

Both approaches reuse the pretrained encoder from the IMDb-wiki task, and require at most 6 configuration changes and a 7-line post-processing module. We report results when aggregating

Table 4: Recall@10 for Ember (R@10) and F1 Scores for the EM workloads using Ember with `join size = 1` (EMB) compared to deep-learning-based EM solutions from DeepMatcher (DM$_*$) and Ditto. Ember meets or exceeds (highlighted) at least one recent, state-of-the-art classifier in most workloads. Ember has a low false negative rate (1 - R@10), and can be used *with* these methods to increase precision

		R@10	F1 score				
Task	Dataset	EMB	EMB	DM$_{\text{RNN}}$	DM$_{\text{att}}$	DM$_{\text{hyb}}$	Ditto
EM-T	Abt-Buy	96.70	85.05	39.40	56.80	62.80	89.33
EM-T	Company	69.89	74.31	85.60	89.80	92.70	93.85
EM-S	Beer	92.86	91.58	72.20	64.00	78.80	94.37
EM-S	iTunes-Amzn	100	84.92	88.50	80.80	91.20	97.06
EM-S	Fodors-Zagat	100	88.76	100	82.10	100	100
EM-S	DBLP-ACM	100	98.05	98.30	98.40	98.45	98.99
EM-S	Amazon-Google	98.94	70.43	59.90	61.10	70.70	75.58
EM-S	DBLP-Scholar	96.29	57.88	93.00	93.30	94.70	95.60
EM-S	Walmart-Amzn	94.97	69.60	67.60	50.00	73.60	86.76
EM-D	DBLP-ACM	99.96	97.58	97.50	97.40	98.10	99.03
EM-D	DBLP-Scholar	95.99	58.08	93.00	92.70	93.80	95.75
EM-D	iTunes-Amazon	100	64.65	79.40	63.60	79.40	95.65
EM-D	Walmart-Amzn	95.39	67.43	39.60	53.80	53.80	85.69

```
"data_dir": "IMDb-wiki",
"join_type": "INNER", # or "LEFT", "RIGHT", "FULL"
"left_size": 1,
"right_size": 1,
```

Listing 2: Core configuration lines annotated with options.

with a `join size` of 1, 10, 20, and 30; both approaches improve performance as the number of neighbors increase, until a plateau is reached. The two-hop approach attains MSE of 1.69, 0.92, 0.89, and 0.89 for `join size` 1, 10, 20, and 30, respectively, while the one-hop approach performs better with 1.59, 0.89, 0.86, and 0.85.

9.6 Low Development Effort

To showcase ease of use, we discuss Ember's configuration, our usability study, and the amount of labeled data required for training.

9.6.1 Configuration. Perhaps surprisingly, we found that exposing only join specification parameters (Listing 2) is a strong out-of-the-box baseline. We require input datasets and supervision to follow a fixed naming convention, under which obtaining the results in Table 3 relies on a default set of configurations options, where only the data directory must be changed. In Table 2, we list how many config changes are required for the best results from Figure 5.

9.6.2 Usability User Study. To understand the amount of domain expertise required to determine the keyless join parameters (i.e., join `size` and `type`), we conducted a user study with 15 graduate students in Computer Science. We provided participants with an introduction to database joins, described context enrichment and our proposed keyless join specification, and displayed EM and recommendation examples similar to those in Section 4.1. We then described search and QA workloads, and asked participants to (1) explain and estimate how long constructing a context enrichment pipeline would take without Ember, and (2) provide `type` and `size` parameters for a keyless join query. All users estimated that constructing a pipeline would take several hours to multiple days, but provided a keyless join specification in minutes, often quicker than *describing* the alternative All users provided correct join `sizes`. Two incorrectly selected an `inner` instead of `left` join, and one erred in the opposite direction. In follow-up interviews, the former asserted that basic postprocessing would admit unmatched results, so an explicit `left` join would be unnecessary sans additional architectural constraints—validating our `inner` join default. The latter misread the survey setup, and revised upon further clarification.

9.6.3 Dependence on Labeled Data. With the exception of MS MARCO, we use all available labeled examples of *relevant pairs* in Table 3 and Figure 5. In MS MARCO, we use 2.5M of the 397M provided labeled *triples* (just 0.63%). For the remainder, we compute how many labeled examples are required to match our reported performance. We found that 9 of the 17 datasets required all of the examples. The remaining required 54.4% of labeled relevant data on average to meet Recall@1 performance, and 44.5% for Recall@10. In the best case (EM-S DBLP-ACM), only 30% and 1% of the data is required to achieve the same Recall@1 and Recall@10, respectively. Consequently, issuing repeated Ember queries and filtering results can help incrementally improve labeled data quality and quantity.

10 RELATED WORK

Similarity Joins. Similarity-based joins often focus on the unsupervised setting [20, 80, 87]. State-of-the art systems such as AutoFJ [52] are tailored for the case where tables can be joined using exact keys or shared phrases that do not differ greatly (e.g., our EM tasks)—not complex context enrichment.

Automated Machine Learning. Automated Machine Learning (AML) systems aim to empower users to develop high-performance ML pipelines minimal intervention or manual tuning. They support various types of data preprocessing, feature engineering, model training and monitoring modules. Examples of AML tools include Ludwig [57], Overton [69], Google Cloud AutoML [2], and H20.ai [4]. However, these platforms do not focus on context enrichment, leaving it as an exercise for users to perform prior to data ingestion.

Relational Data Augmentation. Relational data augmentation systems seek to find new features for a downstream predictive task by deciding whether or not to perform standard database joins across a base table and several auxiliary tables [21, 48, 72]. Similar to context enrichment, these systems aim to augment a base table for a downstream ML workload. However, they assume the existence of KFK relationships that simply must be uncovered.

Data Discovery. Data discovery systems find datasets that may be joinable with or related to a base dataset, and to uncover relationships between different datasets using dataset schema, samples, and metadata [14, 18, 19, 22, 29, 31, 32]. These systems typically surface KFK relationships without tuning for downstream ML workloads.

NLP and Data Management. A recurring aim in data management is to issue natural language commands to interface with structured data [33, 50]. Related to work noted in Section 3.3 are systems that leverage advances in NLP to provide additional domain-specific functionality, such as converting text to SQL [54, 88] or automating data preparation [76]. We focus on the broader problem of context enrichment for downstream tasks as, to our knowledge, most prior work assumes entity information has been resolved [77].

11 CONCLUSION

We demonstrate how entity-level data retrieval and preprocessing, which are components of end-to-end pipelines for a variety of tasks, can be abstracted as context enrichment problems. We propose keyless joins as a unifying abstraction that can power a system for general context enrichment, which allows us to view context enrichment as a data management problem. Consequently, we developed such a system that serves as a vehicle for no-code context enrichment via keyless joins: Ember. We evaluate how developing a keyless join enrichment layer empowers Ember to assist five downstream applications, with no ML code written by the user.

ACKNOWLEDGMENTS

We thank Peter Bailis, Laurel Orr, Vineet Gupta, and the participants of our user study for their insight. This research was supported in part by affiliate members of Stanford DAWN—Ant Financial, Facebook, Google, and VMware—as well as Toyota Research Institute (TRI), and NSF CAREER grant CNS-1651570. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. TRI provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

REFERENCES

[1] 2018. *Datasets for DeepMatcher paper.* https://github.com/anhaidgroup/deepmatcher/blob/master/Datasets.md

[2] 2021. *Cloud AutoML.* https://cloud.google.com/automl

[3] 2021. *Data Robot.* https://www.datarobot.com/platform/automated-machine-learning/

[4] 2021. *h2o.ai.* https://www.h2o.ai/

[5] 2021. *IMDb Datasets.* https://datasets.imdbws.com/

[6] 2021. *MS MARCO.* https://microsoft.github.io/msmarco/

[7] 2021. *NYC Open Data.* https://opendata.cityofnewyork.us/data/

[8] 2021. *rank-bm25.* https://pypi.org/project/rank-bm25/

[9] 2021. *Wikimedia Downloads.* https://dumps.wikimedia.org/

[10] 2021. *Yelp Open Data.* https://www.yelp.com/dataset/documentation/main

[11] Sercan O Arik and Tomas Pfister. 2019. Tabnet: Attentive interpretable tabular learning. *arXiv preprint arXiv:1908.07442* (2019).

[12] Sanjeev Arora, Yingyu Liang, and Tengyu Ma. 2016. A simple but tough-to-beat baseline for sentence embeddings. (2016).

[13] Payal Bajaj, Daniel Campos, Nick Craswell, Li Deng, Jianfeng Gao, Xiaodong Liu, Rangan Majumder, Andrew McNamara, Bhaskar Mitra, Tri Nguyen, et al. 2016. MS MARCO: A human generated machine reading comprehension dataset. *arXiv preprint arXiv:1611.09268* (2016).

[14] Anant Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J Elmore, Samuel Madden, and Aditya G Parameswaran. 2014. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798* (2014).

[15] Andrei Broder. 2002. A taxonomy of web search. In *ACM Sigir forum*, Vol. 36. ACM New York, NY, USA, 3–10.

[16] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *NeurIPS* (2020).

[17] Ursin Brunner and Kurt Stockinger. 2020. Entity matching with transformer architectures-a step forward in data integration. In *International Conference on Extending Database Technology, Copenhagen, 30 March-2 April 2020.* OpenProceedings.

[18] Michael J Cafarella, Alon Halevy, and Nodira Khoussainova. 2009. Data integration for the relational web. *Proceedings of the VLDB Endowment* 2, 1 (2009), 1090–1101.

[19] Raul Castro Fernandez, Dong Deng, Essam Mansour, Abdulhakim A Qahtan, Wenbo Tao, Ziawasch Abedjan, Ahmed Elmagarmid, Ihab F Ilyas, Samuel Madden, Mourad Ouzzani, et al. 2017. A demo of the data civilizer system. In *Proceedings of the 2017 ACM International Conference on Management of Data.* 1639–1642.

[20] Zhimin Chen, Yue Wang, Vivek Narasayya, and Surajit Chaudhuri. 2019. Customizable and scalable fuzzy join for big data. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2106–2117.

[21] Nadiia Chepurko, Ryan Marcus, Emanuel Zgraggen, Raul Castro Fernandez, Tim Kraska, and David Karger. 2020. ARDA: Automatic Relational Data Augmentation for Machine Learning. *Proceedings of the VLDB Endowment* 13, 9 (2020).

[22] Fernando Chirigati, Rémi Rampin, Aécio Santos, Aline Bessa, and Juliana Freire. 2021. Auctus: A Dataset Search Engine for Data Augmentation. *arXiv preprint arXiv:2102.05716* (2021).

[23] Kristy Choi, Curtis Hawthorne, Ian Simon, Monica Dinculescu, and Jesse Engel. 2020. Encoding musical style with transformer autoencoders. In *International Conference on Machine Learning.* PMLR, 1899–1908.

[24] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://www.aclweb.org/anthology/N19-1423

[25] Yingqi Qu Yuchen Ding, Jing Liu, Kai Liu, Ruiyang Ren, Xin Zhao, Daxiang Dong, Hua Wu, and Haifeng Wang. 2020. RocketQA: An Optimized Training Approach to Dense Passage Retrieval for Open-Domain Question Answering. *arXiv preprint arXiv:2010.08191* (2020).

[26] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[27] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *VLDB* (2018).

[28] Christos Faloutsos and Douglas W Oard. 1998. *A survey of information retrieval and filtering methods.* Technical Report.

[29] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A data discovery system. In *2018 IEEE 34th International Conference on Data Engineering (ICDE).* IEEE, 1001–1012.

[30] Rohit Girdhar, Joao Carreira, Carl Doersch, and Andrew Zisserman. 2019. Video action transformer network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 244–253.

[31] Hector Gonzalez, Alon Halevy, Christian S Jensen, Anno Langen, Jayant Madhavan, Rebecca Shapley, and Warren Shen. 2010. Google fusion tables: data management, integration and collaboration in the cloud. In *Proceedings of the 1st ACM symposium on Cloud computing.* 175–180.

[32] Alon Halevy, Flip Korn, Natalya F Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing google's datasets. In *Proceedings of the 2016 International Conference on Management of Data.* 795–806.

[33] Alon Y Halevy, Oren Etzioni, AnHai Doan, Zachary G Ives, Jayant Madhavan, Luke K McDowell, and Igor Tatarinov. 2003. Crossing the Structure Chasm. *CIDR.*

[34] David Hawking. 2004. Challenges in Enterprise Search.. In *ADC*, Vol. 4. Citeseer, 15–24.

[35] Cheng-Zhi Anna Huang, Ashish Vaswani, Jakob Uszkoreit, Noam Shazeer, Ian Simon, Curtis Hawthorne, Andrew M Dai, Matthew D Hoffman, Monica Dinculescu, and Douglas Eck. 2018. Music transformer. *arXiv preprint arXiv:1809.04281* (2018).

[36] Andrew Jaegle, Sebastian Borgeaud, Jean-Baptiste Alayrac, Carl Doersch, Catalin Ionescu, David Ding, Skanda Koppula, Daniel Zoran, Andrew Brock, Evan Shelhamer, et al. 2021. Perceiver IO: A General Architecture for Structured Inputs & Outputs. *arXiv preprint arXiv:2107.14795* (2021).

[37] Andrew Jaegle, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira. 2021. Perceiver: General perception with iterative attention. *ICML* (2021).

[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2017. Billion-scale similarity search with GPUs. *arXiv preprint arXiv:1702.08734* (2017).

[39] Armand Joulin, Edouard Grave, Piotr Bojanowski, Matthijs Douze, Hérve Jégou, and Tomas Mikolov. 2016. Fasttext. zip: Compressing text classification models. *arXiv preprint arXiv:1612.03651* (2016).

[40] Urvashi Khandelwal, Angela Fan, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Nearest neighbor machine translation. *arXiv preprint arXiv:2010.00710* (2020).

[41] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2019. Generalization through memorization: Nearest neighbor language models. *arXiv preprint arXiv:1911.00172* (2019).

[42] Omar Khattab and Matei Zaharia. 2020. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval.* 39–48.

[43] Tae Hyun Kim, Mehdi SM Sajjadi, Michael Hirsch, and Bernhard Scholkopf. 2018. Spatio-temporal transformer network for video restoration. In *Proceedings of the European Conference on Computer Vision (ECCV).* 106–122.

[44] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[45] Pradap Konda, Sanjib Das, Paul Suganthan GC, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, et al. 2016. Magellan: Toward building entity matching management systems. *Proceedings of the VLDB Endowment* 9, 12 (2016), 1197–1208.

[46] Hanna Köpcke and Erhard Rahm. 2010. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering* 69, 2 (2010), 197–210.

[47] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[48] Arun Kumar, Jeffrey Naughton, Jignesh M Patel, and Xiaojin Zhu. 2016. To join or not to join? Thinking twice about joins before feature selection. In *Proceedings of the 2016 International Conference on Management of Data.* 19–34.

[49] Bing Li, Yukai Miao, Yaoshu Wang, Yifang Sun, and Wei Wang. 2021. Improving the Efficiency and Effectiveness for BERT-based Entity Resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 35. 13226–13233.

[50] Fei Li and HV Jagadish. 2014. Constructing an interactive natural language interface for relational databases. *Proceedings of the VLDB Endowment* 8, 1 (2014), 73–84.

[51] Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *Proceedings of the VLDB Endowment* 14, 11 (2021).

[52] Peng Li, Xiang Cheng, Xu Chu, Yeye He, and Surajit Chaudhuri. 2021. Auto-FuzzyJoin: Auto-Program Fuzzy Similarity Joins Without Labeled Examples. *SIGMOD* (2021).

[53] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584* (2020).

[54] Xi Victoria Lin, Richard Socher, and Caiming Xiong. 2020. Bridging Textual and Tabular Data for Cross-Domain Text-to-SQL Semantic Parsing. *arXiv preprint arXiv:2012.12627* (2020).

[55] Zifan Liu, Zhechun Zhou, and Theodoros Rekatsinas. 2020. Picket: Self-supervised Data Diagnostics for ML Pipelines. *arXiv preprint arXiv:2006.04730* (2020).

[56] Jie Lu, Dianshuang Wu, Mingsong Mao, Wei Wang, and Guangquan Zhang. 2015. Recommender system application developments: a survey. *Decision Support Systems* 74 (2015), 12–32.

[57] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. *arXiv preprint arXiv:1909.07930* (2019).

[58] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep learning for entity matching: A design space exploration. In *Proceedings of the 2018 International Conference on Management of Data*. 19–34.

[59] Rajat Mukherjee and Jianchang Mao. 2004. Enterprise Search: Tough Stuff: Why is it that searching an intranet is so much harder than searching the Web? *Queue* 2, 2 (2004), 36–46.

[60] Rodrigo Nogueira and Kyunghyun Cho. 2019. Passage Re-ranking with BERT. *arXiv preprint arXiv:1901.04085* (2019).

[61] Laurel Orr, Megan Leszczynski, Simran Arora, Sen Wu, Neel Guha, Xiao Ling, and Christopher Re. 2020. Bootleg: Chasing the Tail with Self-Supervised Named Entity Disambiguation. *arXiv preprint arXiv:2010.10363* (2020).

[62] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.

[63] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. 2018. Image transformer. In *International Conference on Machine Learning*. PMLR, 4055–4064.

[64] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *arXiv preprint arXiv:1912.01703* (2019).

[65] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2103.00020* (2021).

[66] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[67] Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, and Percy Liang. 2016. Squad: 100,000+ questions for machine comprehension of text. *arXiv preprint arXiv:1606.05250* (2016).

[68] Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB Endowment. International Conference on Very Large Data Bases*, Vol. 11. NIH Public Access, 269.

[69] Christopher Ré, Feng Niu, Pallavi Gudipati, and Charles Srisuwananukorn. 2019. Overton: A data system for monitoring and improving machine-learned products. *arXiv preprint arXiv:1909.05372* (2019).

[70] Matthew Richardson, Christopher JC Burges, and Erin Renshaw. 2013. Mctest: A challenge dataset for the open-domain machine comprehension of text. In *Proceedings of the 2013 conference on empirical methods in natural language processing*. 193–203.

[71] Stephen Robertson and Hugo Zaragoza. 2009. *The probabilistic relevance framework: BM25 and beyond*. Now Publishers Inc.

[72] Vraj Shah, Arun Kumar, and Xiaojin Zhu. 2017. Are Key-Foreign Key Joins Safe to Avoid when Learning High-Capacity Classifiers? *Proceedings of the VLDB Endowment* 11, 3 (2017).

[73] Wei Shen, Jianyong Wang, and Jiawei Han. 2014. Entity linking with a knowledge base: Issues, techniques, and solutions. *IEEE Transactions on Knowledge and Data Engineering* 27, 2 (2014), 443–460.

[74] Chen Sun, Fabien Baradel, Kevin Murphy, and Cordelia Schmid. 2019. Learning video representations using contrastive bidirectional transformer. *arXiv preprint arXiv:1906.05743* (2019).

[75] Sahaana Suri, Raghuveer Chanda, Neslihan Bulut, Pradyumna Narayana, Yemao Zeng, Peter Bailis, Sugato Basu, Girija Narlikar, Christopher Ré, and Abishek Sethi. 2020. Leveraging Organizational Resources to Adapt Models to New Data Modalities. *Proceedings of the VLDB Endowment* 13, 12 (2020).

[76] Nan Tang, Ju Fan, Fangyi Li, Jianhong Tu, Xiaoyong Du, Guoliang Li, Sam Madden, and Mourad Ouzzani. 2020. Relational Pretrained Transformers towards Democratizing Data Preparation [Vision]. *arXiv preprint arXiv:2012.02469* (2020).

[77] James Thorne, Majid Yazdani, Marzieh Saeidi, Fabrizio Silvestri, Sebastian Riedel, and Alon Halevy. 2020. Neural Databases. *arXiv preprint arXiv:2010.06973* (2020).

[78] Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. 2019. Transformer Dissection: A Unified Understanding of Transformer's Attention via the Lens of Kernel. *EMNLP* (2019).

[79] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *NIPS*.

[80] Jiannan Wang, Guoliang Li, and Jianhua Fe. 2011. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *2011 IEEE 27th International Conference on Data Engineering*. IEEE, 458–469.

[81] Kilian Q Weinberger and Lawrence K Saul. 2009. Distance metric learning for large margin nearest neighbor classification. *Journal of machine learning research* 10, 2 (2009).

[82] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. 2019. HuggingFace's Transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771* (2019).

[83] Ledell Wu, Fabio Petroni, Martin Josifoski, Sebastian Riedel, and Luke Zettlemoyer. 2019. Scalable zero-shot entity linking with dense entity retrieval. *arXiv preprint arXiv:1911.03814* (2019).

[84] Chenyan Xiong, Zhuyun Dai, Jamie Callan, Zhiyuan Liu, and Russell Power. 2017. End-to-end neural ad-hoc ranking with kernel pooling. In *Proceedings of the 40th International ACM SIGIR conference on research and development in information retrieval*. 55–64.

[85] Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William W Cohen, Ruslan Salakhutdinov, and Christopher D Manning. 2018. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600* (2018).

[86] Pengcheng Yin, Graham Neubig, Wen-tau Yih, and Sebastian Riedel. 2020. TaBERT: Pretraining for Joint Understanding of Textual and Tabular Data. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 8413–8426.

[87] Minghe Yu, Guoliang Li, Dong Deng, and Jianhua Feng. 2016. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (2016), 399–417.

[88] Jichuan Zeng, Xi Victoria Lin, Caiming Xiong, Richard Socher, Michael R Lyu, Irwin King, and Steven CH Hoi. 2020. Photon: A Robust Cross-Domain Text-to-SQL System. *arXiv preprint arXiv:2007.15280* (2020).

[89] Qian Zhang, Han Lu, Hasim Sak, Anshuman Tripathi, Erik McDermott, Stephen Koo, and Shankar Kumar. 2020. Transformer transducer: A streamable speech recognition model with transformer encoders and rnn-t loss. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 7829–7833.

[90] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and Davd Page. 2020. AutoBlock: A hands-off blocking framework for entity matching. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 744–752.

[91] Fengbin Zhu, Wenqiang Lei, Chao Wang, Jianming Zheng, Soujanya Poria, and Tat-Seng Chua. 2021. Retrieving and Reading: A Comprehensive Survey on Open-domain Question Answering. *arXiv preprint arXiv:2101.00774* (2021).

[92] Yukun Zhu, Ryan Kiros, Rich Zemel, Ruslan Salakhutdinov, Raquel Urtasun, Antonio Torralba, and Sanja Fidler. 2015. Aligning books and movies: Towards story-like visual explanations by watching movies and reading books. In *Proceedings of the IEEE international conference on computer vision*. 19–27.