Featurized-Decomposition Join: Low-Cost Semantic Joins with Guarantees

Sepanta Zeighami
UC Berkeley
zeighami@berkeley.edu

Shreya Shankar
UC Berkeley
shreyashankar@berkeley.edu

Aditya Parameswaran
UC Berkeley
adityagp@berkeley.edu

ABSTRACT

Large Language Models (LLMs) are being increasingly used within data systems to process large datasets with text fields. A broad class of such tasks involves a *semantic join*—joining two tables based on a natural language predicate per pair of tuples, evaluated using an LLM. Semantic joins generalize tasks such as entity matching and record categorization, as well as more complex text understanding tasks. A naive implementation is expensive as it requires invoking an LLM for every pair of rows in the cross product. Existing approaches mitigate this cost by first applying embedding-based semantic similarity to filter candidate pairs, deferring to an LLM only when similarity scores are deemed inconclusive. However, these methods yield limited gains in practice, since semantic similarity may not reliably predict the join outcome. We propose Featurized-Decomposition Join (FDJ for short), a novel approach for performing semantic joins that significantly reduces cost while preserving quality. FDJ automatically extracts features and combines them into a logical expression in conjunctive normal form that we call a *featurized decomposition* to effectively prune out non-matching pairs. A featurized decomposition extracts key information from text records and performs inexpensive comparisons on the extracted features. We show how to use LLMs to automatically extract reliable features and compose them into logical expressions while providing statistical guarantees on the output—an inherently challenging problem due to dependencies among features. Experiments show **up to 10 times reduction in cost** compared to the state-of-the-art while providing the same guarantees.

PVLDB Reference Format:
Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. Featurized-Decomposition Join: Low-Cost Semantic Joins with Guarantees. PVLDB, 19(11): 3048 - 3061, 2026.
doi:10.14778/3836663.3836672
PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/szeighami/FDJ.

1 INTRODUCTION

A *semantic join* between two tables applies a user-specified natural language filter to their cross product using a Large Language Model (LLM). This operation is now supported by many industrial [16, 20, 56, 68] and open-source ones [4, 18, 49, 55]. Semantic joins enable a wide range of downstream applications. These include entity matching tasks [39, 47, 55] such as identifying when two marketplace listings describe the same product [34, 45]; large-scale multi-label classification, such as matching doctors' notes for patients to a long list of medical reactions [15, 49]; and cross-referencing complex documents based on entities they refer to [1, 41, 50]. A real-world instance of the last category that we use as our running example is the *police record* matching task from the Police Records Access Project, a collaborative effort that we are involved in with journalists and public defenders [1]. Here, different police and legal documents (e.g., eyewitness reports, testimonials, court filings, investigation reports) that refer to the same incident need to be matched—a semantic self-join that requires the LLM to determine whether two documents refer to the same incident. Such documents are long and complex, and identifying matching documents requires understanding the incidents they describe, e.g., knowing their date, location, type of police activity, and people involved. This entity matching is useful for journalists analyzing police accountability, e.g., to analyze the outcome of court cases for incidents involving police use of force—without semantic joins, this matching had been done by hand by journalists [1].

Although highly valuable, performing semantic joins is computationally expensive. A naive approach requires invoking an LLM on every record pair in the cross product to determine whether each pair satisfies the join condition. However, making $\Omega(n^2)$ LLM calls for n data records is prohibitive. LOTUS [49] instead leverages a model cascade approach, by first computing the cosine similarity between the vector embedding of the two records as a *proxy score* for matching. Record pairs with low similarity are discarded as non-matches, those with high similarity are accepted as matches, and pairs with inconclusive scores are evaluated by an LLM. This decision is made by setting *cascade thresholds* on proxy scores to determine when to defer to an LLM. To avoid accuracy loss compared to using an LLM on all record pairs, cascade thresholds are set to statistically guarantee the output quality is close to that of using an LLM on every record pair. More recently, BARGAIN [72] refines model cascades by improving when to defer to an LLM (i.e., how to set cascade thresholds), thereby reducing LLM calls.

However, the effectiveness of such model cascade solutions depends on the quality of the embedding vectors. When proxy scores derived from the embeddings closely approximate LLM outputs, they can reduce cost; however, if embeddings poorly encode the join criteria, proxy scores become unreliable and most join decisions will need to be made by an LLM. In practice, text records are often lengthy and contain both relevant and irrelevant information for the join. In police records, each document may include multiple names, locations, dates, and incident descriptions, as well as irrelevant boilerplate. Embedding this information into a single vector

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836672

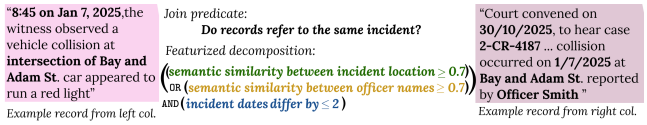

Figure 1: Example of a Featurized Decomposition

fails to capture relevant details with sufficient fidelity (as is theoretically suggested by [67]). Thus, semantic similarity becomes a poor indicator of the join, and methods that rely on it end up using the LLM on most record pairs, providing limited savings. In our police records example, BARGAIN defers to the LLM on around 80% of all record pairs, thus costing close to that of naively comparing all pairs. We see similar results on other real-world datasets.

We present *Featurized Decomposition Join* (FDJ), a novel method for optimizing semantic joins. Our key insight for reducing join cost is to logically rewrite the join condition into inexpensive predicates that extract features from text records and use the features to perform the join. Extracting features can be done cheaply—it requires only a linear pass over the records with an LLM—relative to performing a quadratic number of pairwise comparisons with an LLM to obtain the join result. Thus, a logical expression that accurately represents the join condition using feature-based predicates can significantly reduce the join cost. We call such a logical expression a *featurized decomposition*: a conjunctive normal form (CNF) expression consisting of predicates that extract and compare features using inexpensive distance functions (e.g., lexical, arithmetic, or embedding-based). Fig. 1 illustrates a featurized decomposition for our example which combines predicates on three features—date, location, and police officer—to decide if two records refer to the same incident. Featurized decompositions can be complex, since the join condition may depend on multiple features and the decomposition must be robust to variations, incorrect extractions, or missing feature values.

An accurate featurized decompositions can perform a semantic join with low cost, but automatically constructing one is challenging for two reasons. First, finding relevant features and extracting them correctly is difficult. A naive approach is to ask an LLM to produce all relevant features. However, LLMs often miss important features or produce redundant ones, and may not extract the feature values correctly. Meanwhile, applying LLMs to extract features incurs cost and must be done judiciously. Second, even after finding a set of relevant features and extracting them correctly, designing a featurized decomposition using such features while providing theoretical guarantees on output quality is non-trivial. In fact, we show that constructing the lowest cost decomposition with guarantees is NP-hard. Quality guarantees are particularly important to ensure reliability given that features are automatically identified and extracted by LLMs which inevitably introduces errors. Meanwhile, guaranteeing quality is challenging as it requires an in-depth analysis of the quality of the featurized decomposition when constructing it. Existing model cascades approaches provide statistical guarantees for the simpler setting of finding a single threshold on the proxy score [29, 49, 72].

To automatically construct a reliable featurized decomposition, FDJ begins by extracting a comprehensive set of candidate features through an iterative process. At each iteration, it evaluates a running set of features using a sample of results labeled by the LLM, identifies records where the features fail to accurately estimate the join outcome and generates new features to address these gaps. Next, FDJ uses these features to form the decomposition. This construction phase is also guided by labeled samples: FDJ evaluates alternative decompositions, estimates their join costs, and selects the one with the lowest cost that is estimated to meet the desired quality targets. A crucial part of this step is ensuring that the chosen decomposition satisfies quality guarantees. We theoretically guarantee output quality by appropriately choosing the threshold parameters in our predicates—recall that each predicate in a featurized decomposition evaluates whether the distance between extracted features is below a certain threshold (e.g., whether extracted dates are within *two* days, see Fig. 1). We provide a tight theoretical analysis of our threshold selection procedure, generalizing existing bounds in the model cascade literature that set one-dimensional thresholds [29, 49, 72] to high dimensions.

Using the above methodology, FDJ automatically constructs a featurized decomposition and uses it to perform the join. Our experimental evaluation across 7 different real-world datasets show FDJ provides significant cost savings. **Over BARGAIN applied to semantic joins, FDJ has about half the cost on average, but reduces the cost up to 10 times on some datasets**.

2 PRELIMINARIES

Semantic Joins. We are given two sets of strings (text columns, documents) and a natural language predicate. Our goal is to find the subset of the cross product of the two sets on which the natural language predicate evaluates to true, where the evaluation is done by an LLM. We let $\mathscr{L}$ be a user-specified LLM, $\mathtt{L}$ and $\mathtt{R}$ be two sets of strings and let $\mathtt{p}$ be the natural language predicate, called the *join condition*. $\mathtt{p}$ is a parameterized string (or a langex [49]) with parameters $\mathtt{r} \in \mathtt{R}$ and $\mathtt{l} \in \mathtt{L}$ as input, e.g., "`Do {l} and {r} describe the same incident?`". We define $\mathscr{L}_\mathtt{p} : \mathtt{L} \times \mathtt{R} \to \{0, 1\}$ to be a function that determines whether the join condition is satisfied for a pair from $\mathtt{L} \times \mathtt{R}$. Specifically, given $(\mathtt{l}, \mathtt{r}) \in \mathtt{L} \times \mathtt{R}$, $\mathscr{L}_\mathtt{p}$ first substitutes $\mathtt{l}$ and $\mathtt{r}$ in $\mathtt{p}$ and then passes the resulting string as a prompt to the LLM $\mathscr{L}$ to obtain a boolean output. We say two records, $(\mathtt{l}, \mathtt{r}) \in \mathtt{L} \times \mathtt{R}$ *match* for a join condition $\mathtt{p}$ if $\mathscr{L}_\mathtt{p}(\mathtt{l}, \mathtt{r}) = 1$. Finally, the *true* semantic join answer is defined as [49]

$$\mathtt{Y} = \{(\mathtt{l}, \mathtt{r}); (\mathtt{l}, \mathtt{r}) \in \mathtt{L} \times \mathtt{R}, \mathscr{L}_\mathtt{p}(\mathtt{l}, \mathtt{r}) = 1\}.$$

We define $\bar{\mathtt{Y}} = (\mathtt{L} \times \mathtt{R}) \setminus \mathtt{Y}$ as the set of non-matches. We refer to the pairs in $\mathtt{Y}$ (in $\bar{\mathtt{Y}}$) as *positive (negative) pairs*, to any $(\mathtt{l}, \mathtt{r}) \in \mathtt{L} \times \mathtt{R}$ as a *pair*, and to the value of $\mathscr{L}_\mathtt{p}(\mathtt{l}, \mathtt{r})$ as the *label* for a pair $(\mathtt{l}, \mathtt{r})$.

Approximate Semantic Joins. Evaluating the true semantic join exactly is expensive, as it requires invoking an LLM on every pair of rows from $\mathtt{L}$ and $\mathtt{R}$. To support low-cost processing while still producing answers similar to that of the true answer, we allow the users to specify quality requirements on the recall and precision of the result. For any result set $\hat{\mathtt{Y}} \subseteq \mathtt{L} \times \mathtt{R}$, we define its precision, $\mathcal{P}(\hat{\mathtt{Y}})$, and recall, $\mathcal{R}(\hat{\mathtt{Y}})$, respectively, as $\mathcal{P}(\hat{\mathtt{Y}}) = \frac{|\mathtt{Y} \cap \hat{\mathtt{Y}}|}{|\hat{\mathtt{Y}}|}$ and $\mathcal{R}(\hat{\mathtt{Y}}) = \frac{|\mathtt{Y} \cap \hat{\mathtt{Y}}|}{|\mathtt{Y}|}$. We consider the setting where user specifies a recall requirement T_R and a precision requirement T_P, and requires a result whose precision and recall are at least T_P and T_R, respectively, with high probability. To formalize this, let δ be a user-provided probability of failure and let $\mathtt{A}(\mathtt{L}, \mathtt{R}, \mathtt{p})$ be any (randomized) algorithm that outputs a join result $\hat{\mathtt{Y}}$. We say the algorithm performs an approximate semantic

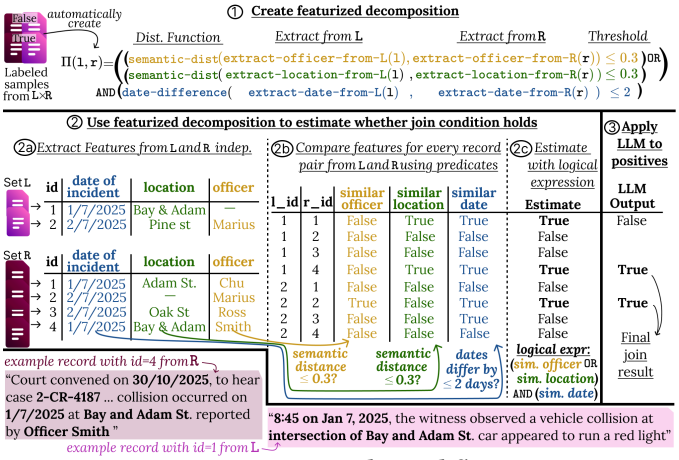

Figure 2: FDJ Example Workflow

join with statistical guarantees if it satisfies

$$\mathbb{P}_{\hat{Y}\sim A(L,R,p)}(\mathcal{P}(\hat{Y}) < T_P \text{ or } \mathcal{R}(\hat{Y}) < T_R) \leq \delta,$$

where the probability is over random runs of the algorithm. Our goal is to design an algorithm with statistical guarantees while minimizing total cost, as we discuss next.

Embedding Models. To reduce costs, we use embedding models that provide low-cost high-dimensional semantic representations of the data. Formally, an embedding model $\mathscr{E}$, which produces embedding vectors, that is, for a string s, $\mathscr{E}(s)$ is a high-dimensional vector. For two strings s_1 and s_2, their semantic similarity is defined as the cosine similarity between the embeddings, $\mathscr{E}(s_1)$ and $\mathscr{E}(s_2)$. For consistency with other distance metrics, we use *semantic distance* for two strings s_1 and s_2 to refer to (1−semantic similarity).

Cost Model. We focus on the cost of using LLMs and embedding models; the cost of non-LLM operations are negligible in comparison. We define the *cost of an algorithm* A, denoted by $C(A(L, R, p))$, as the monetary cost it incurs using LLM $\mathscr{L}$ and embedding model $\mathscr{E}$, determined using the total number of input and output tokens.

Problem Definition. Putting everything together, our goal is to solve the following problem:

Definition 2.1 (Approximate Semantic Joins with Guarantees). Given two datasets L and R, a recall target T_R, a precision target T_P, a probability of failure δ, an LLM $\mathscr{L}$, and an embedding model $\mathscr{E}$, design an algorithm A to perform approximate join with statistical guarantees on precision and recall while minimizing the cost A:

$$\min_{A} \quad C(A(L, R, p))$$
$$\text{s.t.} \quad \mathbb{P}_{\hat{Y}\sim A(L,R,p)}\left(\mathcal{P}(\hat{Y}) < T_P \text{ or } \mathcal{R}(\hat{Y}) < T_R\right) \leq \delta$$

Note that we focus on the setting that a user would like to perform an LLM-powered join with a specific LLM. Our goal is to reduce the cost of this join while providing guarantees to *ensure that the cost reduction optimizations do not incur a regression in quality* when performing this user-specified join.

3 FDJ OVERVIEW & FEATURIZED DECOMPOSITIONS

We present Featurized-Decomposition Join (FDJ), our method for performing approximate semantic joins at low cost. FDJ uses a technique we call *featurized decomposition* to reduce join costs by estimating the join outcome based on features extracted from text records. In this section, we first provide an overview of how FDJ works and present the necessary terminology in Sec. 3.1. Then, we discuss the challenges in performing joins using featurized decomposition and provide an outline for the rest of this paper in Sec. 3.2. For now, we consider the case that $T_P = 1$ and $T_R < 1$, that is, we must provide 100% precision but recall can be imperfect, and we use $T = T_R$ to simplify notation. We extend this approach to a more relaxed precision requirement, i.e., $T_P < 1$ and $T_R < 1$, in Sec. 7. $T_P = 1$ is important for many real-world applications; for example, incorrectly showing non-matching police records (i.e., those that refer to different incidents) to users can corrupt analysis, leading to wrong conclusions with severe consequences. While having recall target $T_R = 1$ may also be important for some applications, it unfortunately would require evaluating all pairs in the cross product (because any pair not evaluated risks having recall<1). Relaxing the recall requirement, as done by FDJ, makes performing such joins feasible—on our largest dataset it would cost about US$13.5M to compare all pairs, while FDJ costs about US$54 with $T_R = 0.9$.

3.1 Overview and Terminology

We next provide a high-level workflow of FDJ using Fig. 2. The figure shows the workflow for our running example where our goal is to join different police records based on whether they refer to the same incident (Fig. 2 shows two toy examples of such text records).

① *Creating Featurized Decomposition.* FDJ first samples a subset of $L \times R$ and labels them using the LLM, $\mathscr{L}$. It then uses these labeled samples to automatically create a *featurized decomposition*: a Boolean function designed to estimate whether the join condition holds for a pair (l, r) using features extracted. A featurized decomposition consists of (1) functions to extract features from records in L and R, (2) logical predicates that check whether extracted features from $l \in L$ and $r \in R$ are *similar* based on a feature-specific distance function; and (3) a logical expression that combines the logical predicates to estimate whether the join condition holds in conjunctive normal form (CNF). In Fig. 2, Step ①, the function $\Pi(l, r)$ is a featurized decomposition; it takes $(l, r) \in L \times R$ as input and produces a binary output to estimate if the join condition holds for (l, r). The function Π is a logical combination of three logical predicates (each in a different color). The predicate in yellow checks whether police officer names are similar. It leverages functions that extract names of police officers from l and r and checks whether the semantic distance between extracted names are within threshold 0.3. Predicates in green and blue are similarly defined for location and date features.

② *Using Featurized Decomposition to Produce Estimates.* After constructing the featurized decomposition, we use it to estimate whether each record pair satisfies the join condition. This estimation process first applies the feature extraction functions to obtain the relevant features, and then evaluates the logical expression over these features to produce the final estimate. Step ② in Fig. 2 shows an example of how the estimates are obtained. First, the feature extraction functions are applied independently to all rows in L and R. Step 2a shows the output of this step for 2 records in L and 4 records in R. The values under the location column for L are obtained by applying `extract-location-from-L` function to every record in L, while the values for R are obtained by applying `extract-location-from-R` to every record in R. After obtaining feature values, Step 2b, shows the result of each of the three predicates for each pair of records, and finally Step 2c shows the final

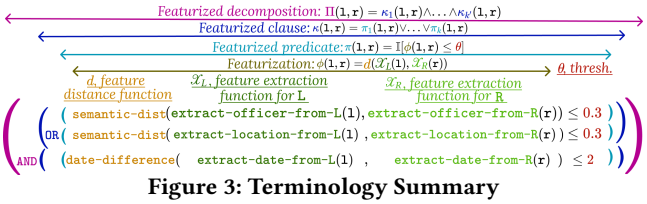

Figure 3: Terminology Summary

result of the expression for every pair of records from $\mathsf{L} \times \mathsf{R}$, where three record pairs are estimated to satisfy the join condition.

③ *Refinement.* Finally, we remove any false positives introduced, by applying the LLM $\mathscr{L}$ to any pair from ②c estimated to satisfy the join condition. In our example, we find one false positive. The final join result output by FDJ is the set of pairs estimated to be positive by the featurized decomposition and confirmed as such by the LLM. Our approach guarantees perfect precision (since every estimated pair is evaluated by an LLM), but depending on the featurized decomposition may have imperfect recall.

Terminology. We next introduce necessary terminology to formalize the above workflow. Fig. 3. shows a summary of the terminology along with the example from Fig. 2. We use the term *feature extraction function* to refer to functions designed to extract features (e.g., `extract-location-from-L` in Fig. 3). We typically denote a feature extraction function for extracting features from L (R) by $\mathscr{X}_L$ ($\mathscr{X}_R$, resp.)—these function can include LLM invocations and have any output signatures. We allow $\mathscr{X}_L$ to be different from $\mathscr{X}_R$ to enable dataset-specific functions. We use *feature distance function* (or distance functions) to refer to real-valued functions designed to compute distances between features, denoted by d (e.g., `semantic-dist` in Fig. 3). A distance function does not perform any LLM invocations and only performs lightweight comparisons on features (e.g., semantic distance, lexical difference, arithmetic operations). We use extraction and distance functions to define logical predicates which we refer to as *featurized predicates*: Boolean functions of the form $\pi(\mathsf{l}, \mathsf{r}) = \mathbb{I}[d(\mathscr{X}_L(\mathsf{l}), \mathscr{X}_L(\mathsf{r})) \leq \theta]$, consisting of feature extraction functions $\mathscr{X}_L$ and $\mathscr{X}_R$, a distance function d, and a threshold, $\theta \in \mathbb{R}$. When presenting our solution, we often discuss the functions $d, \mathscr{X}_L, \mathscr{X}_R$ separately from θ, so for convenience we refer to the triplet $(d, \mathscr{X}_L, \mathscr{X}_R)$ as a *featurization* and θ as a *predicate threshold*. For a featurization $\phi = (d, \mathscr{X}_L, \mathscr{X}_R)$ we denote its *inference function* by $\phi(\mathsf{l}, \mathsf{r}) = d(\mathscr{X}_L(\mathsf{l}), \mathscr{X}_R(\mathsf{r}))$. We abuse notation and refer to the featurization and its inference function interchangeably. The three logical predicates in Fig. 3 are featurized predicates.

To estimate whether the join condition holds, we combine featurized predicates into logical expressions in CNF form. Formally, we define a *featurized clause* $\kappa : \mathsf{L} \times \mathsf{R} \to \{0, 1\}$ as a Boolean function that is a disjunction of featurized predicates, that is, of the form $\kappa(\mathsf{L}, \mathsf{R}) = \pi_1(\mathsf{l}, \mathsf{r}) \vee \ldots \vee \pi_k(\mathsf{l}, \mathsf{r})$ for some integer k and featurized predicates $\pi_1, ..., \pi_k$. Then, a *featurized decomposition*, Π, is a Boolean function $\Pi : \mathsf{L} \times \mathsf{R} \to \{0, 1\}$ defined as a conjunction of featurized clauses, that is, $\Pi(\mathsf{l}, \mathsf{r}) = \kappa_1(\mathsf{l}, \mathsf{r}) \wedge \ldots \wedge \kappa_{k'}(\mathsf{l}, \mathsf{r})$, for some integer k' and featurized clauses $\kappa_1, ..., \kappa_{k'}$. We abuse notation and for any $\mathsf{S} \subseteq \mathsf{L} \times \mathsf{R}$, define $\Pi(\mathsf{S}) = \{(\mathsf{l}, \mathsf{r}) \in \mathsf{S}; \Pi(\mathsf{l}, \mathsf{r}) = 1\}$.

3.2 Challenges and Roadmap

Using the above terminology, to perform joins using featurized decomposition, we first create a featurized decomposition Π (Step ① in Fig. 2), then create a candidate join result $\hat{\mathsf{Y}} = \Pi(\mathsf{L} \times \mathsf{R})$ using the decomposition (Step ② in Fig. 2) and finally use the LLM $\mathscr{L}$ to obtain our final join result as $\{(\mathsf{l}, \mathsf{r}) \in \hat{\mathsf{Y}}; \mathscr{L}_\mathsf{p}(\mathsf{l}, \mathsf{r}) = 1\}$. Recall that we want 100% precision and recall at least T. Steps ① and ② guarantees the latter, while Step ③ (where each $(\mathsf{l}, \mathsf{r}) \in \hat{\mathsf{Y}}$ is checked by $\mathscr{L}$) guarantees the former.

Challeges. The main challenge in the above workflow is Step ①, wherein we automatically construct a featurized decomposition. The recall of FDJ is equivalent to the recall of its featurized decomposition, so Step ① must theoretically guarantee that the featurized decomposition found meets the recall target. Moreover, the cost of FDJ depends on the precision of the featurized decomposition—false positive pairs introduced incur additional cost of LLM calls during refinement in Step ③—in addition to the cost of finding the featurized decomposition (Step ①) and using it to produce estimates (Step ②). The rest of this paper focuses on finding featurized decompositions that minimize cost and guarantee high recall.

Roadmap. We formalize the problem of creating minimum cost featurized decompositions in Sec. 4 and provide an overview of how we construct featurized decompositions; we present the details of the construction procedure in Secs. 5-6. Finally, we present the final FDJ algorithm in Sec. 7, discuss its guarantees and extensions when relaxing the precision requirement. Proofs of our theoretical results are presented in our technical report [71].

4 MIN. COST FEATURIZED DECOMPOSITIONS

In this section, we formalize the problem of finding minimum cost featurized decomposition, show its NP-Hardness and present an overview of our solution for creating the featurized decomposition.

4.1 Problem Statement and NP-Hardness

Finding a minimum-cost featurized decomposition involves searching over all possible decompositions and selecting the one that achieves the lowest cost while meeting the recall target. The search space depends on possible featurizations that can be used to create the decompositions. Let $\mathbb{F}$ denote the set of possible featurizations where each element is a triplet $(d, \mathscr{X}_L, \mathscr{X}_R)$ such that for any $(\mathsf{l}, \mathsf{r}) \in \mathsf{L} \times \mathsf{R}$, $d(\mathscr{X}_L(\mathsf{l}), \mathscr{X}_R(\mathsf{r})) \in \mathbb{R}$. In principle, $\mathbb{F}$ could include any valid combination of feature extraction functions (e.g., functions over string inputs) and distance functions (e.g., real-valued functions over two inputs). In practice, as discussed later, we restrict $\mathbb{F}$ to featurizations automatically generated by an LLM. Given $\mathbb{F}$, the space of possible featurized decompositions consists of all logical expressions in CNF whose predicates are derived from featurizations in $\mathbb{F}$ paired with threshold parameters $\theta \in \mathbb{R}$. Our goal is to find a decomposition with minimum cost among such decompositions.

Definition 4.1. [Minimum Cost Featurized Decomposition (MCFD)] Given a set of possible featurizations $\mathbb{F}$, two sets L and R and a predicate function $\mathscr{L}_\mathsf{p} : \mathsf{L} \times \mathsf{R} \to \{0, 1\}$, find a featurized decomposition with recall at least T while minimizing total cost.

Unfortunately, solving MCFD optimally is computationally difficult:

Theorem 4.2. *MCFD is NP-hard.*

Thm. 4.2 is shown by a reduction from the Set Cover problem. Proof is available in our technical report [71].

4.2 Constructing Featurized Decompositions

We next describe how we solve MCFD. At a high level, we use LLMs to obtain useful features from which we create our decomposition.

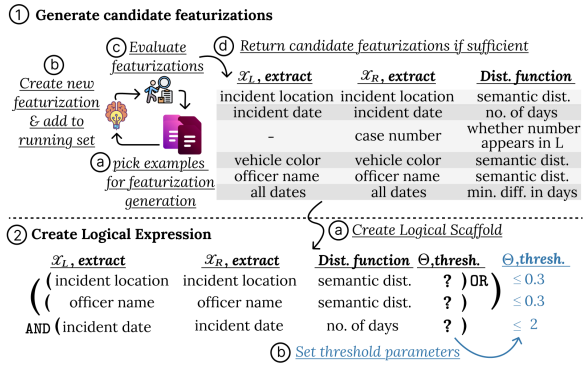

Figure 4: Finding Featurized Decompositions

LLMs help us prune the space of possible featurizations to a few relevant ones that we then algorithmically combine to create decompositions. Nonetheless, obtaining reliable featurizations from LLMs is challenging. As our experiments show, a single-attempt prompting strategy often overlooks important features or creates erroneous extraction functions. Feature omissions often happens for features that are useful only for a small subset of the join matches. The LLM fails to identify such features from a random sample of the data, because the few records for which those features are useful are often either not present in a small random sample, or, if taken a larger sample, become overshadowed by other records. Erroneous extraction functions often occur due to missing values and inconsistencies in the data. For example, to extract a *date and time* feature in a dataset where time may not be present in some records, the distance function must be designed to correctly calculate distance between dates even when time is not present. Moreover, our goal is to guarantee that the recall of the resulting decomposition meets a user-specified target—a non-trivial statistical estimation task.

To address these challenges, we design a multi-step pipeline that leverages LLMs to generate featurizations and uses labeled samples to iteratively refine them, ensuring both reliability. We then use this set of *candidate featurizations* to construct a featurized decomposition with tight theoretical guarantees on recall. As such, we follow a two-step procedure, where we (1) first generate a set of candidate featurizations, from which (2) we choose a subset to create the featurized decomposition. We provide an overview of our approach below with the aid of Fig. 4. Details of the two steps (1) and (2) are presented in Secs. 5 and 6, respectively.

Candidate Generation. We begin by constructing a comprehensive set of featurizations. As shown in Fig. 4 (top half), this process proceeds iteratively: we (1a) select a set of examples as demonstrations for the LLM; (1b) use an LLM to generate new featurizations or refine existing ones; and (1c) evaluate the resulting set of featurizations. If the current set is deemed sufficient to perform the join, (1d) we return it as the final output, or we otherwise repeat the process. Fig. 4 illustrates several candidate featurizations. At each iteration, we select positive examples on which the current featurizations performs worst—measured by the number of false positives that must be introduced to correctly classify the positive example using the current featurizations. These examples are then fed back to the LLM to generate featurizations suitable for those examples or to correct errors in feature extraction, forming a feedback loop to iteratively improve the quality of the featurizations.

Algorithm 1 `get-candidate-featurizations(p, S, `Y_S`)`

```
C, Y_C ← A sample of β elements from S and their labels
Φ ← ∅
for i ← 0 to max_iter do
    Φ ← Φ ∪ get-featurization-from-examples(C, Y_C, p)
    C, Y_C ← evaluate-and-pick-examples(S, Y_S, Φ)
    if C = ∅ then
        return Φ
return Φ
```

Logical Expression Formulation. Next, we use the set of candidate featurizations obtained from the previous step to construct a featurized decomposition that minimizes cost and meets the recall requirement. At a high-level, we evaluate possible decompositions, estimating for each whether it satisfies the recall target, and selecting the lowest-cost one among those that do. However, guaranteeing that the recall target is met is challenging, as testing multiple decompositions introduces a multiple hypothesis testing problem. A naive analysis using union bound across each test leads to loose bounds, and consequently limited cost savings. To obtain tight bounds, we divide the process of constructing the featurized decomposition into two steps. First, we build a *logical scaffold* that specifies which featurizations to include and how to combine them, without specifying threshold parameters. Fig. 4 illustrates this scaffold in our running example. Second, we set the thresholds to ensure the final decomposition meets the target recall. Theoretical analysis of recall is only needed in the latter step, where it is more tractable.

5 CANDIDATE GENERATION

As discussed in the previous section (see Fig. 4), we iteratively generate featurizations, evaluate them, and pick examples as demonstrations to generate new featurizations. This iterative process ensures the featurizations are reliable—the evaluation phase surfaces errors in extraction functions that the LLM can fix—and comprehensive—different examples demonstrate different useful featurizations.

We present this procedure in Alg. 1. The algorithm takes as input a set, S, containing random samples from $\mathtt{L} \times \mathtt{R}$ together with their labels, $Y_S = \{\mathscr{L}_\mathtt{p}(\mathtt{l}, \mathtt{r}); (\mathtt{l}, \mathtt{r}) \in \mathtt{S}\}$, generated by an LLM. S is used to guide the generation process. Alg. 1 iteratively calls two subroutines (detailed later): (1) `get-featurization-from-examples` to generate new featurizations using a set of examples as in-context demonstration and (2) `evaluate-and-pick-examples` to evaluate the featurizations using S and find examples where the featurizations cannot accurately estimate the join outcome. If `evaluate-and-pick-examples` decides the featurizations do accurately estimate the join outcome for the pairs in S, it returns an empty set and the algorithm terminates, returning the current set of featurizations. Alg. 1 otherwise repeats the process up to a maximum number of iterations. Note that at each iteration, a subset of size β from S is chosen to be passed to the LLM as demonstrations, where β is a parameter determined based on the LLM's context limit. We illustrate this process in Fig. 5, where starting with a random sample (Step (1)), we alternate between generating candidate featurizations (Steps (2) and (4)) and evaluating them to pick new examples as demonstrations for generation (Steps (3) and (5)). We next describe featurization generation in Sec. 5.1 and evaluating and picking examples in Sec. 5.2.

5.1 LLM Pipeline for Featurization Generation

We next describe the subroutine `get-featurization-from-examples` to generate factorizations. We use a multi-step pipeline that decomposes featurization generation into several LLM invocations,

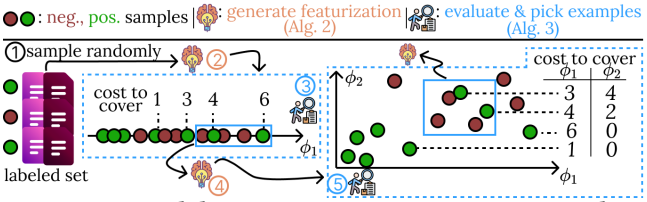

Figure 5: Candidate Featurization Generation Example

Algorithm 2 Featurization Generation

```
1:  procedure get-featurization-from-examples(S, Y_S, p)
2:      U ← get-featurization-descriptions(S, Y_S, p)
3:      for u ∈ U do
4:          X_L ← get-feature-extractors(S, Y_S, p, u, left)
5:          X_R ← get-feature-extractors(S, Y_S, p, u, right)
6:          d ← get-distance-func(S, Y_S, p, u, X_L, X_R)
7:          Φ ← Φ ∪ {(d, X_L, X_R)}
8:      return Φ
9:  procedure get-feature-extractors(S, Y_S, p, u, col)
10:     description ← get-feature-description-for-col(S, Y_S, p, u, col)
11:     if should-use-llm(S, Y_S, p, description) then
12:         prompt ← get-extraction-prompt(S, Y_S, p, description)
13:         X ← create LLM-powered extraction function using prompt
14:     else
15:         X ← get-extraction-code(S, Y_S, p, description)
16:     return X
```

following best practices [11, 32] to improve reliability. Alg. 2 shows this procedure (functions in blue are LLM-powered; we present prompts in our technical report [71]). It takes a set of example pairs and their labels as input that are used as in-context demonstrations of when the join condition holds and doesn't hold for generation.

Specifically, to generate featurizations, we first ask the LLM to provide a set of high-level natural language descriptions for possible featurizations (Line 2) and then instantiate each (Line 3-7). We use the description to both generate feature extraction functions (Lines 4-5) and a distance function (Line 6). The latter is done by asking the LLM to choose among a set of predefined distance functions (including semantic or lexical distance, and arithmetic difference)—we provide the list of predefined distance functions in our technical report [71]. Designing each feature extraction function $\mathcal{X}_L$ and $\mathcal{X}_R$ (Line 10-16) also involves multiple LLM invocations: first to get a detailed description of the feature extraction function for the specific column (Line 10), then to decide whether to use LLM or code for extraction (Line 11), and finally, in each case, another LLM invocation to either obtain the prompt for LLM-powered extraction (Line 12) or Python code (Line 15).

5.2 Evaluation and Picking Examples

We next describe `evaluate-and-pick-examples` for evaluating featurizations using a set of labeled samples, S. To do so, we define the notion of *cost to cover*. Let S_p and S_n be, respectively, the positive and negative subsets of S. For any positive pair $(l, r) \in S_p$, define

$$c_\Phi(l, r) = \min_{\phi \in \Phi} \sum_{(l', r') \in S_n} \mathbb{I}[\phi(l', r') \leq \phi(l, r)], \quad (1)$$

as the *minimum cost to cover* a positive pair (l, r) with featurizations in Φ. The summation $\sum_{(l', r') \in S_n} \mathbb{I}[\phi(l', r') \leq \phi(l, r)]$ represents, for any ϕ, the number of false positives that must be admitted to correctly classify the positive pair (l, r) if we only used ϕ. The minimum cost to cover a positive pair (l, r) with the featurization set Φ is therefore, the minimum number of false positives that must be admitted to correctly classify the positive pair using any one of the featurizations in Φ. If $c_\Phi(l, r)$ is high, none of the featurizations in Φ are suitable to distinguish (l, r) from the negative pairs. Thus, the featurizations should be improved by taking the high cost to cover pair (l, r) into account. We pick such a pair as an example to be passed to the LLM during candidate generation to ensure the LLM generates new featurizations that are suitable for the pair.

Recall that we use the notion of minimum cost to cover to decide which pairs to use for generating new featurizations. Our goal is to identify pairs whose correct classification with a CNF employing a given set of featurizations leads to admitting many false positives. We then generate new featurizations to enable correctly classifying such pairs with fewer false positives. Minimum cost to cover to helps identify such pairs, because it provides an upper bound on the number of false positives in the *best* possible CNF that correctly classifies the pair, i.e., a CNF that introduces the fewest false positives while correctly classifying the pair. Thus, a pair with the highest minimum cost to cover is one whose correct classification (with any CNF) can incur the most false positives in the worst case. Our algorithm uses this pair to generate new featurizations. Formally, minimum cost to cover, $c_\Phi(l, r)$, for a pair, (l, r) provides an upper bound on the number of false positives of the best possible CNF that correctly classifies (l, r) due to the following existence argument. Taking ϕ^* as the featurization that achieves the minimum cost to cover (i.e., in Eq. 1), the CNF consisting of one predicate $\pi(l', r') = \mathbb{I}[\phi^*(l', r') \leq \theta]$ for $\theta = \phi^*(l, r)$ has $c_\Phi(l, r)$ false positives and correctly classifies (l, r).

Examples of cost to cover computation are shown in Fig. 5. In Step 3 and after generating a single featurization ϕ_1, we see that 4 positive pairs have non-zero cost to cover. The cost to cover value is calculated based on number of negative pairs to the left (i.e., with smaller feature distance values) of the positive pairs. A subset of these pairs are used to generate a new featurization, ϕ_2, in Step 4. Then, in Step 5, we again compute cost to cover but now for two featurizations. We see that the cost to cover improves for some of the positive pairs when using the featurization, ϕ_2, added in Step 4.

Alg. 3 specifies this process more formally. We first calculate the minimum cost to cover for all positive pairs (Lines 2-3). If this cost is sufficiently small for all the pairs—according to a system parameter α—we assume the candidate featurizatations are a suitable indicator of the join result and thus return an empty set (Lines 4-5). Otherwise, we return β pairs in total ($\frac{\beta}{2}$ positives and negatives) to ensure they fit in the LLM context. To generate a candidate set that enables a high recall featurized decomposition, we select positive pairs with the highest cost to cover (Line 6). We select negative pairs that have smaller feature distances than at least one selected positive pair (Line 7), randomly selecting $\frac{\beta}{2}$ of such pairs if there are more (Line 8); this choice of negative pairs helps the LLM understand why the cost to cover for positive pair was high.

6 LOGICAL EXPRESSION FORMULATION

We obtain a set of candidate featurizations, Φ, from the previous step. Here, we use a subset of these featurizations to create a featurized decomposition, Π. As discussed in Sec. 4.2, we construct the featurized decomposition by first generating a *logical scaffold* and then generating the final featurized decomposition by setting its parameters. We first formally define the notion of a logical scaffold

Algorithm 3 `evaluate-and-pick-examples`(S, Y_S, Φ)

$\mathsf{S}_p, \mathsf{S}_n \leftarrow$ negative and positive subsets of S
for $(\mathtt{l}, \mathtt{r}) \in \mathsf{S}_p$ **do** ▷ Calculate cost to cover for all positives
  $c_\Phi(\mathtt{l}, \mathtt{r}) \leftarrow \min_{\phi \in \Phi} \sum_{(\mathtt{l}', \mathtt{r}') \in \mathsf{S}_n} \mathbb{I}[\phi(\mathtt{l}', \mathtt{r}') \leq \phi(\mathtt{l}, \mathtt{r})]$
if $\max_{(\mathtt{l},\mathtt{r}) \in \mathsf{S}_p} c_\Phi(\mathtt{l}, \mathtt{r}) < \alpha$ **then**
  return $\emptyset$
$\mathsf{C}_p \leftarrow \frac{\beta}{2}$ pairs $(\mathtt{l}, \mathtt{r}) \in \mathsf{S}_p$ with largest $c_\Phi(\mathtt{l}, \mathtt{r})$
$\mathsf{C}_n \leftarrow \{(\mathtt{l}', \mathtt{r}') \in \mathsf{S}_n; \exists (\mathtt{l}, \mathtt{r}) \in \mathsf{C}_p, \phi \in \Phi \text{ s.t. } \phi(\mathtt{l}', \mathtt{r}') \leq \phi(\mathtt{l}, \mathtt{r})\}$
$\mathsf{C}_n \leftarrow$ random subset of size $\frac{\beta}{2}$ of C_n if $|\mathsf{C}_n| > \frac{\beta}{2}$
return $\mathsf{C}_p \cup \mathsf{C}_n$, labels for $\mathsf{C}_p \cup \mathsf{C}_n$ from Y_S

Algorithm 4 `get-logical-scaffold`$(\Phi, T, \mathsf{S}, Y_S)$

$\mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta) \leftarrow \mathtt{True}$ ▷ Initialize $\mathring{\Pi}$ as a function always returning True
$c_{min}, \phi_{min} \leftarrow \hat{C}_S(\mathring{\Pi}), \emptyset$
while $|\Phi| \geq 0$ **do**
  for $\phi \in \Phi$ **do**
    $\mathring{\Pi}'(\mathtt{l}, \mathtt{r}; \Theta \cup \{\theta\}) \leftarrow \mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta) \wedge \mathbb{I}[\phi(\mathtt{l}, \mathtt{r}) \leq \theta]$
    if $\hat{C}_S(\mathring{\Pi}') < c_{min}$ **then**
      $c_{min}, \phi_{min} \leftarrow \hat{C}_S(\Pi'), \phi$
  $\Phi \leftarrow \Phi \setminus \{\phi_{min}\}$
  if $\phi_{min} \neq \emptyset$ **and** $c_{min} < \hat{C}_S(\mathring{\Pi}) - \gamma$ **then**
    $\mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta \cup \{\theta\}) \leftarrow \mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta) \wedge \mathbb{I}[\phi_{min}(\mathtt{l}, \mathtt{r}) \leq \theta]$
  else
    break
for $\phi \in \Phi$ **do**
  for each clause $\mathring{\kappa}$ in $\mathring{\Pi}$ **do**
    $\mathring{\kappa}'(\mathtt{l}, \mathtt{r}; \Theta \cup \{\theta\}) \leftarrow \mathring{\kappa}(\mathtt{l}, \mathtt{r}; \Theta) \vee \mathbb{I}[\phi(\mathtt{l}, \mathtt{r}) \leq \theta]$
    $\mathring{\Pi}' \leftarrow \mathring{\Pi}$ but replace $\mathring{\kappa}$ with $\mathring{\kappa}'$
    if $\hat{C}_S(\mathring{\Pi}') < \hat{C}_S(\mathring{\Pi}) - \gamma$ **then**
      $\mathring{\Pi} \leftarrow \mathring{\Pi}'$
return $\mathring{\Pi}$

in Sec. 6.1, before presenting our method for creating the scaffold in Sec. 6.2 and setting the thresholds in Sec. 6.3.

6.1 Logical Scaffolds

Recall that a featurized decomposition consists of a set of featurized predicates, with every featurized predicate of the form, $\phi(\mathtt{l}, \mathtt{r}) \leq \theta$, for a featurization, ϕ and a threshold parameter, θ. We define a *predicate scaffold* similarly, except that a predicate scaffold takes the threshold parameter as an input. Thus, a predicate scaffold is a function of the form $\mathring{\pi}(\mathtt{l}, \mathtt{r}; \theta) = \mathbb{I}[\phi(\mathtt{l}, \mathtt{r}) \leq \theta]$ for some $\phi \in \Phi$. We similarly extend the notion of featurized clauses and featurized decomposition to take the threshold parameter as input. Formally, a *clause scaffold* $\mathring{\kappa}$ with predicate scaffolds $\mathring{\pi}_1, ..., \mathring{\pi}_k$ is the function $\mathring{\kappa}(\mathtt{l}, \mathtt{r}; \Theta) = \mathring{\pi}_1(\mathtt{l}, \mathtt{r}; \Theta_1) \vee ... \vee \mathring{\pi}_k(\mathtt{l}, \mathtt{r}; \Theta_k)$, for $\Theta \in \mathbb{R}^k$ and Θ_i its i-th element. A *logical scaffold* with clauses $\mathring{\kappa}_1, ..., \mathring{\kappa}_{k'}$ is a function

$$\mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta) = \mathring{\kappa}_1(\mathtt{l}, \mathtt{r}; \Theta^1) \wedge ... \wedge \mathring{\kappa}_{k'}(\mathtt{l}, \mathtt{r}; \Theta^{k'}),$$

where $\Theta = [\Theta^1, ..., \Theta^{k'}]$ is a list of parameters, each a high dimensional vector of threshold parameters for each clause. We abuse notation and define, for any $\mathsf{S} \subseteq \mathsf{L} \times \mathsf{R}$,

$$\mathring{\Pi}(\mathsf{S}; \Theta) = \{(\mathtt{l}, \mathtt{r}); (\mathtt{l}, \mathtt{r}) \in \mathsf{S}, \mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta) = 1\}.$$

A logical scaffold takes threshold parameters as input, so it does not on its own specify any predicates. Given a logical scaffold, we need to find suitable threshold parameters to create a featurized decomposition. Thus, using the above notation, to find a featurized decomposition, we first find a logical scaffold $\mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta)$; we discuss how in Sec. 6.2. Then, we create the final featurized decomposition by finding a suitable set of threshold parameters Θ^*, so we obtain $\Pi(\mathtt{l}, \mathtt{r}) = \mathring{\Pi}(\mathtt{l}, \mathtt{r}; \Theta^*)$; we discuss how in Sec. 6.3.

6.2 Creating the Logical Scaffold

We use a greedy approach to construct a logical scaffold using the candidate featurizations Φ (recall that finding an optimal featurized decomposition is NP-hard). We use a set of pairs S sampled from $\mathsf{L} \times \mathsf{R}$ together with their labels Y_S obtained from an LLM. We build a logical scaffold to minimize the join cost while satisfying the recall target on these samples. Starting with an empty scaffold, we iteratively extend the current scaffold by adding a predicate that yields the largest reduction in join cost. However, the join cost depends on the choice of threshold parameters for the scaffold. So, we optimistically estimate the join cost when using the scaffold as the minimum achievable join cost across all possible threshold settings for the scaffold. We next present our solution in detail, first discussing how we estimate the join cost when using a scaffold before presenting our algorithm for creating scaffolds.

Cost Estimation. We estimate the join cost for $\mathring{\Pi}$ as the lowest cost achievable when using $\mathring{\Pi}$ while meeting the recall target on the samples, S. We use the false positive rate of $\mathring{\Pi}$ as a proxy for cost. False positive rate determines the cost of refining the join result with an LLM (i.e., Step 3 in Fig. 2) which in practice dominates the join cost—extensions to more fine-grained cost models are straightforward but in practice we see limited benefits. More formally, denote by S_p and S_n respectively the positive and negative subsets of S. Define

$$\mathcal{R}_S(\mathring{\Pi}, \Theta) = \frac{|\mathring{\Pi}(\mathsf{S}_p; \Theta)|}{|\mathsf{S}_p|}, \ \mathcal{F}_S(\mathring{\Pi}, \Theta) = \frac{|\mathring{\Pi}(\mathsf{S}_n; \Theta)|}{|\mathring{\Pi}(\mathsf{S}; \Theta)|},$$

respectively, the observed recall and false positive rate of a logical scaffold $\mathring{\Pi}$, when using thresholds Θ. We define the cost function

$$\hat{C}_S(\mathring{\Pi}) = \min_{\Theta \in \mathbf{\Theta}} \mathcal{F}_S(\mathring{\Pi}, \Theta) \text{ s.t. }, \mathcal{R}_S(\mathring{\Pi}, \Theta) \geq T, \tag{2}$$

where $\mathbf{\Theta}$ is the set of all possible thresholds and depends on the logical scaffold, $\mathring{\Pi}$. In practice, we find this minimum in Eq. 2 through exhaustive search since the set of all possible thresholds is typically small. Note that all possible thresholds must have recall $\geq T$ and among such thresholds we only need to consider thresholds at which cost changes. For small sample sizes, such a search space is typically small and can be exhaustively searched efficiently. We provide further details in our technical report [71].

Scaffold Construction Algorithm. Using this cost function, creating the logical scaffold follows two steps. First, we create a logical scaffold only consisting of conjunction of predicates. Then, we iteratively add disjunctions to each clause if the addition reduces cost. We present this algorithm formally in Alg. 4. We first create the conjunctive clauses (Lines 3-12). To do so, we find a new featurization whose addition to the current logical scaffold decreases the cost the most among all possible featurizations, estimated by $\hat{C}_S$ (Lines 4-7). If such a featurization reduces cost sufficiently over the current logical scaffold, then it is added with a conjunction to the current scaffold; we check if the reduction in cost is at least equal to a threshold γ to avoid adding featurizations that have a marginal difference. After obtaining a logical scaffold containing conjunctions, we iteratively add featurizations to each clause with disjunctions (Lines 13-18). We iteratively consider each featurization and evaluate for the featurization if its addition to a clause reduces cost by more than γ (Lines 15-17), and add it the clause if so (Line 18)—we consider each featurization and clause pair once. The

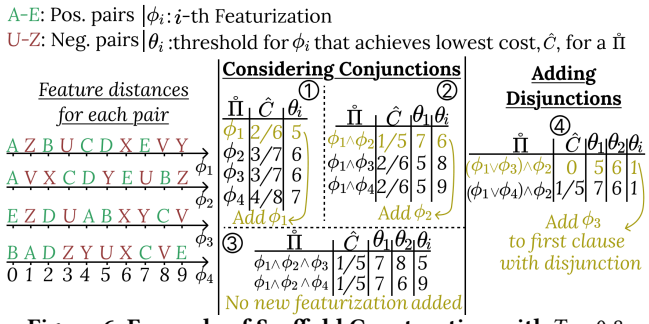

Figure 6: Example of Scaffold Construction with $T = 0.8$

loop to add featurizations with disjunctions (Line 14) differs from the loop to add conjunctions (Line 3) where each featurization is added to one clause when considering conjunctions, while it may be added to multiple clauses when considering disjunctions. This is because the loop in Line 3 in Alg. 4 *creates* clauses while the loop in Line 14 *adds additional featurizations* to the clauses. Thus, in Line 3, adding multiple clauses with the same feature would not improve the cost. However, in Line 14, adding the same feature to multiple (already distinct) clauses creates distinct clauses with overlapping features, which may reduce the cost.

Note that generated featurizations by FDJ may be correlated—we instruct the LLM to not repeat a feature, but in practice some features may end up being similar. The CNF construction algorithm, Alg. 4, avoids picking correlated features in Lines 10 and 18, because adding a feature correlated with the existing features often leads to little cost reduction, tested in Lines 9 and 17. Uncorrelated features often provide more cost reduction and are chosen instead. We present a case study in our technical report [71] that shows that in practice correlated features are not selected.

Example. We illustrate this process using Fig. 6. The figure constructs a scaffold from 4 featurizations ϕ_1, ..., ϕ_4 with target $T = 0.8$ and using a sample set of 10 pairs, 5 positive (denoted by A-E) and 5 negative (denoted by U-Z). The figure (left) shows the distance for each pair for each featurization, e.g., $\phi_1(D) = 5$. All distances for all featurizations are integers between 0 to 9. To construct the scaffold, at Step (1), we find a featurization, ϕ_i, such that setting $\mathring{\Pi}(\mathsf{l}, \mathsf{r}; \theta) = \mathbb{I}[\phi_i(\mathsf{l}, \mathsf{r}) \leq \theta]$ achieves the lowest cost, $\hat{C}(\mathring{\Pi})$. The table in Step (1) shows this cost computation for all four ϕ_i, where the first column shows different possible scaffolds, the second column shows the cost, $\hat{C}$, of these scaffolds calculated based on Eq. 2 and the third column shows the *best threshold setting* for each scaffold that achieves the minimum in Eq. 2. For example, for the scaffold $\mathring{\Pi}(\mathsf{l}, \mathsf{r}; \theta) = \mathbb{I}[\phi_1(\mathsf{l}, \mathsf{r}) \leq \theta]$ (the first row in the table in Step (1)), $\hat{C}(\mathring{\Pi}) = \frac{2}{6}$ and that the threshold, θ_1, at which this cost is achieved is 5. This is because negative pairs Z and U as well as positive pairs A, B, C, D have ϕ_1 distance at most 5, so that false positive rate at $\theta_1 = 5$ is $\frac{2}{6}$, while recall is 0.8. Based on the table at Step (1), ϕ_1 yields the lowest cost and is thus added to the scaffold.

Next, at Step (2), we again consider the cost of adding remaining featurizations to the scaffold. Adding ϕ_2 now leads to the lowest $\hat{C}$—the table at Step (2) shows $\hat{C}$ for different scaffolds, with corresponding best threshold settings that achieve the cost. Observe that the best value of θ_1 (that achieves the cost $\hat{C}$) is different for different logical scaffolds; e.g., compare the best value for θ_1 for the

Algorithm 5 FDJ(L, R, p, T, δ)

```
1: S ← sample subset of size k from L × R uniformly at random
2: Y_S ← {ℒ_p(l, r); (l, r) ∈ S}
3: Φ ← get-candidate-featurizations(p, S, Y_S)
4: Π̊ ← get-logical-scaffold(Φ, T, S, Y_S)
5: S′ ← sample a subset of size k′ from L × R uniformly at random
6: Y_S′ ← {ℒ_p(l, r); (l, r) ∈ S′}
7: Θ* ← arg min_Θ ℱ_S′(Π̊, Θ) s.t. ℛ_S′(Π̊, Θ) ≥ T′        ▷ Y_S′ used here
8: Π(l, r) ← Π̊(l, r; Θ*)                                  ▷ define featurized decomposition function
9: Ŷ ← {(l, r); (l, r) ∈ L × R, Π(l, r) = 1}
10: return {(l, r); (l, r) ∈ Ŷ, ℒ_p(l, r) = 1}
```

scaffold consisting of conjunction of ϕ_1 and ϕ_2 (in Step (2)) and the scaffold only consisting of ϕ_1 (in Step (1)). Next, at Step (3), adding new featurizations with conjunctions does not reduce the cost anymore, and thus the algorithm moves on to adding disjunctions. Step (4) shows the cost of scaffolds when adding disjunctions to the first clause (Alg. 4 considers adding featurizations to all clauses, which we omit to simplify the figure), where now the disjunction of ϕ_1 and ϕ_3 reduces the cost.

6.3 Setting the Thresholds

The previous step returns a logical scaffold $\mathring{\Pi}(\mathsf{l}, \mathsf{r}; \Theta)$. Next, we find the threshold parameters Θ^*, to construct the final featurized decomposition $\Pi(\mathsf{l}, \mathsf{r}) = \mathring{\Pi}(\mathsf{l}, \mathsf{r}; \Theta^*)$. We want to find thresholds that both minimize cost and meet the recall target with high probability. To do so, we first sample a set, S, uniformly at random from $\mathsf{L} \times \mathsf{R}$ and label them (S is a new sample set not the same set as previous sections). We then set the thresholds as

$$\Theta^* \in \arg\min_{\Theta \in \mathbb{R}^r} \mathcal{F}_S(\mathring{\Pi}, \Theta) \text{ s.t. } \mathcal{R}_S(\mathring{\Pi}, \Theta) \geq T', \tag{3}$$

where $T' > T$ is an *adjusted recall target*. Eq. 3 finds a threshold that has a recall higher than T' on set S while providing the lowest false positive rate, where, similar to before, we use the false positive rate as a proxy for cost. To provide guarantees on recall, we show

Theorem 6.1 (Informal). *For a suitable T' and under realistic conditions, for any threshold Θ^* where $\mathcal{R}_S(\mathring{\Pi}, \Theta^*) \geq T'$, we have*

$$\mathbb{P}_{S \sim \mathsf{L} \times \mathsf{R}}(\mathcal{R}_{\mathsf{L} \times \mathsf{R}}(\mathring{\Pi}, \Theta^*) < T) \leq \delta,$$

where, for simplicity, we have presented an informal theorem statement—formal statement and proof are available in our technical report [71]. Thm. 6.1 directly implies a threshold chosen by Eq. 3 meets the recall target with high probability.

7 FDJ ALGORITHM AND DISCUSSION

7.1 Final FDJ Algorithm

We next present the final FDJ algorithm and how to support precision target.

Algorithm. Alg. 5 presents the final FDJ algorithm. We first sample and label a subset of k pairs from $\mathsf{L} \times \mathsf{R}$. We pass this set to `get-candidate-featurizations` defined in Alg. 1 to generate candidate featurizations, Φ. Then, we use Φ to create a logical scaffold—we use the same sample set S in this step. To set the threshold parameters, we sample and label another subset, S′ of size k', $\mathsf{L} \times \mathsf{R}$, and find the threshold with the lowest false positive rate that has observed recall more than the adjusted target, T' (Line 7). Using the threshold parameters, we now have a featurized decomposition that we use to create a candidate join result, $\hat{\mathsf{Y}}$. The join result is

then refined using $\mathscr{L}_\mathsf{p}$ to remove false positives and returned to the user. The output of Alg. 5 provides the required recall guarantees:

Theorem 7.1. *Let $\bar{\mathsf{Y}}$ be output of Alg. 5. Then, $\mathbb{P}(\mathcal{R}(\bar{\mathsf{Y}})) < T) \leq \delta$.*

We provide an analysis of the cost complexity of FDJ as well as impact of system parameters on cost in our technical report [71]. In practice, we set k and k' to ensure a sufficient number of positive pairs to enable statistically significant estimates. In our experiments, we see fixed values perform well across datasets and FDJ is not sensitive to the parameter values.

Considering Precision Target. To take advantage of a relaxed precision target ($T_P < 1$), we introduce a post-processing step after generating the estimated result $\hat{\mathsf{Y}}$. This step identifies positive pairs in $\hat{\mathsf{Y}}$ using the existing featurizations Φ, without invoking $\mathscr{L}_\mathsf{p}$, thereby reducing the join cost. Specifically, for each featurization $\phi \in \Phi$, we create a high-precision subset of $\hat{\mathsf{Y}}$ as $\hat{\mathsf{Y}}_\phi = \{(\mathsf{l}, \mathsf{r}) \in \hat{\mathsf{Y}}; \phi(\mathsf{l}, \mathsf{r}) \leq \theta\}$ where θ is a threshold on the feature distances chosen in order to guarantees $\mathcal{P}(\hat{\mathsf{Y}}_\phi) \geq T_P$ with high probability. Determining θ is a a one-dimensional threshold selection problem given a precision target, for which we directly apply the solution proposed by Zeighami et al. [72]. We then combine these subsets to obtain $\hat{\mathsf{Y}}' = \cup_{\phi \in \Phi} \hat{\mathsf{Y}}_\phi$ and return the final result as $\{(\mathsf{l}, \mathsf{r}); (\mathsf{l}, \mathsf{r}) \in \hat{\mathsf{Y}} \setminus \hat{\mathsf{Y}}', \mathscr{L}_\mathsf{p}(\mathsf{l}, \mathsf{r}) = 1\} \cup \hat{\mathsf{Y}}'$. To preserve theoretical guarantees, we account for the cumulative failure probability across multiple applications of [72] and ensure that the subsets $\hat{\mathsf{Y}}_\phi$ do not overlap. Details are provided in our technical report [71].

7.2 Discussion and Extensions

In this section, we discuss our modeling choices and how they relate to existing learning paradigms such as weak supervision. We also discuss other cost optimizations that can be combined with FDJ.

7.2.1 *Modeling Choices*

FDJ discovers a set of featurizations and learns a CNF to combine the featurizations to predict the join outcome. The problem of combining featurizations is reminiscent of weak supervision [26, 53], which combines noisy labeling functions (analogous to our featurizations) to generate accurate labels. However, there are two main differences between FDJ and weak supervision: (1) we automatically identify featurizations (instead of humans creating labeling functions) and (2) we have access to ground-truth labels so we can learn how to construct and combine the labeling functions from these labels (instead of only considering the relationships between the labeling functions as in weak supervision [52, 53]).

Since we have access to ground-truth labels, we instead treat the problem as one of supervised learning. In fact, any classification model that takes the featurizations as input and predicts whether the join condition holds can be used to combine featurizations. We use CNFs for two reasons. First, we expect the join outcome to depend on logical relationships between the features, e.g., whether two police reports refer to the same incident depends on a logical relationship between the time, location, and the people involved in the incident. Second, our CNF construction algorithm optimizes the cost of performing the join at a specific recall, while training off-the-shelf classifiers minimize training loss (e.g., mean squared loss), which can lead to suboptimal solutions for a specific recall target. We empirically evaluate different modeling choices in Sec. 8.4.

While we focus on constructing CNFs, formulae in disjunctive normal form (DNF) may provide similar benefits. Theoretically, CNFs and DNFs are equivalent in that given a CNF we can always construct an equivalent DNF (and vice versa). However, our theoretical analysis for quality guarantees relies on the CNF form. Specifically, our proof of Thm 6.1 relies on the CNF property that in a CNF expression that meets a recall target T, all clauses must also meet the target T . Although we believe—due to theoretical equivalence between CNFs and DNFs—that similar results to Thm **??** should be possible for DNFs, our proof techniques would need to be adjusted, which we leave to future work.

7.2.2 *Cost Discussion.*

Note that FDJ reduces join cost from the quadratic cost of pair-wise comparisons to a linear cost of feature extraction. For large datasets, a linear pass over the dataset with an expensive LLM can also be infeasible. There are multiple optimizations that FDJ can support to reduce the cost further. As already discussed in Alg. 2, FDJ may generate code to extract features, instead of performing LLM calls, which significantly reduces the cost. If code generation is not possible, feature extraction can also be done with cheaper LLMs. To support the use of smaller models, when creating the logical scaffold in Alg. 4, in addition to picking the featurization to add to the scaffold, we can also pick which model to use for that featurization by estimating the cost of the join when using each model and picking the model that leads to the overall lowest cost. We leave a careful consideration of these optimizations to future work.

8 EXPERIMENTS

We next empirically evaluate FDJ, presenting our setup in Sec. 8.1. Sec. 8.2 shows results comparing FDJ with baselines, Sec. 8.3 evaluates cost/accuracy trade-offs, Sec. 8.4 presents ablation studies and Sec. 8.5 discusses scalability and sensitivity to parameters. Additional results on sensitivity to label quality, sensitivity to other parameters, impact of data characteristics, and discussion of feature extraction failure modes are presented in our technical report [71]

8.1 Setup

Datasets and Tasks. We perform experiments on 7 different real-world datasets and multiple synthetic datasets. We discuss the real datasets here, and synthetic ones are diferred to our technical report [71]. Our datasets cover various domains and applications. Reviews is a dataset of movie reviews and the task is to identify reviews that express the same sentiment about a movie [35]. *Citations* [41] is a dataset of legal arguments and the task is to perform a self-join to find legal arguments that cite each other. *Categorize* [43, 62] and *BioDEX* [15, 49] are multi-label classification tasks; the former classifying a product given its description into a list of categories and the latter classifying patient notes into a list of medical reaction terms. *Products* is a classic entity resolution dataset from [34, 45] that contains product descriptions from different online listings and the task is to match listings that refer to the same product. *Movies* is a dataset of Wikipedia pages of different actors and movies—we created this dataset by crawling Wikipedia pages of popular movies and actors in IMDB [23]. The task is to join movie pages with pages of actors based on whether an actor acts in a given movie. Finally, *Police Records* is as discussed in Sec. 1, where

	|L|	|R|	n^+
Products	973	956	616
BioDEX	8,103	3,718	20,000
Citations	16,161	16,161	20,000
Movies	4,043	4,983	10,475
Police Records	2,093	2,093	56,531
Categorize	19,101	3,144	20,000
Reviews	1.375,738	1,375,738	122,368,284

Table 1: Dataset Size

	Avg.	% Failed
LOTUS	75.4	100
BARGAIN	91.1	9
FDJ	91.2	7

Table 2: Observed recall and failure rate, $T_R = 90\%, \delta = 10\%$

the task is to join records that belong to the same incident. For all datasets except BioDEX, Categorize and Citations, we use them as is; we describe our join prompts in our technical report [71]. For other datasets, to stay within our budgetary constraints, we sampled 20,000 ground-truth pairs, and used only the column values from the 20,000 pairs as the columns to be joined. Table 1 shows data sizes and number of ground-truth pairs, n^+.

Baselines. We compare FDJ with BARGAIN [72] the state-of-the-art model cascade method with statistical guarantees, Optimal Cascade the provides a lower bound on any model cascade approach, as well as ThalamusDB [26] and training classifiers on record embeddings. BARGAIN is designed for filter and classification tasks, but can be applied to joins by treating a join as a filter on the cross product of the columns. We use semantic similarity between the vector embedding of records as the proxy score for BARGAIN. Additionally, to understand the limit of cascade-based approaches that rely on semantic similarity alone, we consider *optimal cascade* as a baseline. This approach chooses the *best possible* cascade threshold by looking at *all* the ground-truth pairs; i.e., it selects the threshold that prunes the most pairs while satisfying the quality target based on the ground-truth pairs. We report the cost of performing the join given the cascade threshold, that is, we ignore the cost of finding the thresholds. Optimal cascade provides a lower bound on the cost of any model-cascade approach as it is allowed to find the best possible cascade threshold for free—this is not possible in practice.

We also considered using LOTUS [49] as a baseline that performs model cascades with some statistical guarantees. However, as was observed by [72], LOTUS [49] that uses SUPG [29] only provides guarantees in the limit as data size goes to infinity, and, in practice, can fail to meet the target on any finite dataset. [72] reports that SUPG misses the target up to 75% of the time with a failure probability of only 10%. We observed a similar trend in our experiments. We report supporting results for BioDEX dataset in Table 2, where LOTUS fails to meet the target recall of 90% *on every run*, and returns average recall of less than 80%, while both FDJ and BARGAIN meet the recall target. We exclude LOTUS from experiments since it does not provide statistical guarantees.

We additionally compare FDJ with two methods that do not provide theoretical guarantees: ThalamusDB [24, 26] and an embedding-based classifier. ThalamusDB uses batching to reduce join cost. Embedding-based classifier is an approach that trains an ML model on top of record embeddings to predict whether the join holds or not. We used XGBoost [2] as our classifier. We considered an MLP and a linear model as well, but XBGoost provided the lowest cost.

Metrics. We are primarily concerned with the LLM cost of semantic joins. By default and for all the datasets except Products (see below), we report the *cost ratio* of the methods that normalizes the cost of a method by the cost of a naive all pairs comparison—normalization enables comparison across datasets and data sizes. Formally, for an algorithm A that costs C_A to join dataset L and R, the *cost ratio* is $\frac{C_A}{C}$, where C is the cost of performing the join by calling an LLM on all the pairs in L×R. We use OpenAI models—GPT-4.1 for feature extraction and performing join, O3 for generating candidate featurizations; and text-embedding-large as the embedding model—so the cost ratio is calculated using the corresponding monetary costs. We note that all the datasets used contain ground-truth labels. To reduce costs, for every invocation of $\mathscr{L}_p$, instead of actually performing the LLM call, we simulate it by returning the known ground-truth and calculate the cost by creating the prompt—containing text records and join instruction—that would have been sent to the LLM, counting the number of tokens in the prompt and computing monetary cost based on OpenAIs cost model. Results in our technical report [71] show the benefits are similar when using LLMs instead of ground-truth labels.

Our cost evaluation differs for Products dataset in order to maintain the original evaluation procedure from [45]. Specifically, [45] uses training and test pairs for evaluation. We evaluate the join condition on the test set and use the training set to create the featurized decomposition (for FDJ) and set model cascade thresholds (for BARGAIN). For this dataset, we calculate the cost of the methods as the cost of finding the featurized decomposition or model cascade threshold on the training set plus the cost of using them to perform the join on the test. Cost ratio is defined as this cost divided by the cost of only using the LLM to perform the join on the test set.

Parameters. For both BARGAIN and FDJ, the total number of samples is set so that we obtain 250 positive pairs for their offline procedures. For BARGAIN, the samples are used only to determine the cascade thresholds, for FDJ, they are used both for generating featurizations, creating the logical scaffold, and setting the thresholds. Featurizations and logical scaffolds are determined on a subset using 50 of the positives, and the rest of the samples are used to set the predicate thresholds. By default we set $T_R = 90\%$, $T_P = 100\%$ and $\delta = 10\%$ in our experiments—we saw limited cost saving from using a more relaxed precision target, T_P, for both BARGAIN and FDJ and thus we keep $T_P = 100\%$ to ensure better output quality. We use BARGAIN with $\beta = 0$ which provides the same theoretical guarantees as FDJ.

8.2 Baseline Comparisons

Table 3 shows the comparison between FDJ and baselines on the real-world datasets. **FDJ improves over BARGAIN across all datasets, with gains up to 10 times over BARGAIN, and up to 8 times over the optimal cascade**.

The relative performance of FDJ compared with BARGAIN depends on dataset characteristics. In our experiments, the datasets can be divided into three categories: (1) Movies and Citations have a feature that is an accurate indicator of the join outcome; FDJ provides the highest benefits here. In Movies, a featurization that extracts movie names and actor names from the corresponding Wikipedia pages represents the join condition accurately. However, BARGAIN and the optimal cascade perform poorly because the pages contain a lot of information beyond a single actor or movie name. (2) In Police Records and Products, relevant features exist but are a weaker indicator of the join outcome. FDJ provides significant

	Citations	Police Records	Categorize	BioDEX	Movies	Products
BARGAIN	0.28	0.69	0.28	**0.74**	0.62	0.36
FDJ	**0.068 (0.24×)**	**0.32 (0.46×)**	**0.23 (0.83×)**	**0.74 (0.99×)**	**0.031 (0.04×)**	**0.25 (0.69×)**
optimal cascade	*0.19*	*0.67*	*0.12*	*0.73*	*0.53*	*0.30*

Table 3: Cost ratio at $T = 90\%$ (number in parenthesis shows cost reduction factor compared with BARGAIN).

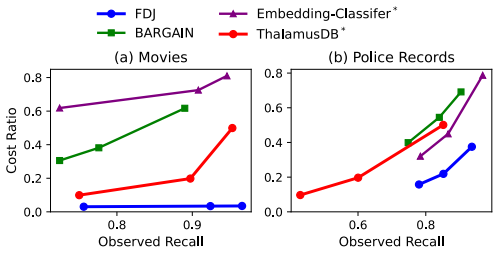

Figure 7: Cost/Accuracy Trade-off (*without guarantees)

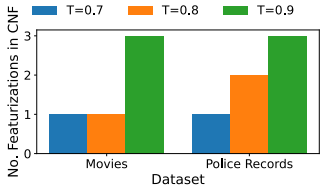

Figure 8: Number of predicates in constructed CNF

	Movies	Police Records
PGM	0.89	0.94
XGBoost	0.048	0.39
XGBoost+AL	0.044	0.43
CNF	**0.031**	**0.32**

Table 4: Cost ratio for different classification methods

gains in both, but less so compared to Movies and Citations. In Products, model numbers in product listings is a featurization that can be used as a join indicator. However, model numbers may not exist for every record or may be noisy (e.g., one description may have complete model numbers but another only a few digits) requiring reliance on other features (e.g., color or brand name) which may also not conclusively imply a match. (3) Categorize and BioDEX are classification tasks. In such datasets the join relationship is not easily decomposable into featurizations—classification may require a complex mapping across many features. Nonetheless, FDJ provides improvements over BARGAIN by extracting relevant information.

8.3 Cost/Accuracy Trade-offs

We compare the cost/accuracy trade-offs FDJ provides with baselines. To do so for FDJ and BARGAIN, we vary the recall target among {0.7, 0.8, 0.9}, and calculate the observed recall and cost ratio. We additionally include baselines that do not provide theoretical guarantees: ThalamusDB and Embedding-Classifier. ThalamusDB uses batching to reduce join cost without any other optimizations, performing the join batch by batch with an LLM. The latter approach, which we call Embedding-Classifier, trains an ML model, specifically XGBoost [2], on top of record embeddings to predict whether they join. We considered MLPs and linear models as well, but XGBoost provided the lowest cost. For ThalamusDB, we vary the batch size used among {2, 5, 10}. For Embedding-Classifier, we vary the decision threshold (we evaluate recall at multiple decision thresholds and pick thresholds that lead to observed recall in the range [0.7, 1] on the test set).

The results are shown in Fig. 7; FDJ provides the best cost/accuracy trade-offs across methods, while ThalamusDB, which does not provide quality guarantees, can lead to arbitrarily low recall.

8.4 Ablation Study and Analysis

Study of Modeling Choice. Here, we evaluate possible alternatives to constructing CNFs. Specifically, we train classifiers on the featurizations to predict the join outcome, where we evaluate XGBoost classifiers as well as probabilistic graphical models, referred to as PGMs (typically used in weak supervision [52]). We furthermore evaluate the methods when using *active learning* (AL) to collect the training set [54]. We specifically use uncertainty sampling [37, 54] to decide which points to label . We iteratively switch between labeling and training until we reach the labeling budget.

Table 4 shows that CNFs perform better than other approaches. We also see no benefit in using active learning, which we believe is due to significant label imbalance (most pairs in the cross product are negative, so most labeled pairs also end up being negative pairs with active learning). When training models without active learning we kept the labels in the training set balanced. Nonetheless, we believe a more comprehensive study of active learning for both CNF construction as well as other classifiers can help improve cost savings, which we leave to future work.

Complexity of Created CNFs. We study the complexity of the constructed CNFs based on the number of predicates that appear in the CNF. Fig. 8 shows this result across two datasets, for different recall targets T. We see that there are at most 3 predicates in the CNF and sometimes only 1. Note that there are a total of 6 featurizations generated in both datasets and FDJ uses only a subset of these featurizations, leading to simple CNF formulae. We provide a case study of features discovered and used in our technical report [71]. When $T = 0.9$, for Movies, the three predicates are in a single clause connected via disjunction, while for Police Report the CNF contains two clauses, one with a disjunction. This low complexity occurs due to parameter γ in Alg. 4, where a predicate is only added if it can improve cost by at least γ. In practice, this means few predicates are chosen where each has a significant impact on accuracy, instead of many predicates each with marginal impact.

8.5 Scalability and Sensitivity Analysis

We next study FDJ's scalability and sensitivity to parameters.

Scalability Evaluation We evaluate scalability of our method on the large scale Reviews dataset from [35]. This dataset consists of reviews of different movies, and the goal is to perform a self-join based on whether the reviewers express a similar sentiment about the same movie. We evaluate scalability of FDJ and BARGAIN on this dataset, both on the full dataset and random subsets of size $\{10^4, 5\times10^5, 10^5, 5\times10^5\}$ to observe impact of data size. To evaluate feasibility of performing the join, we report dollar cost instead of

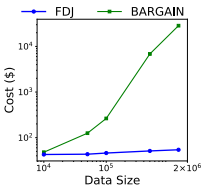

Figure 9: Scalability evaluation

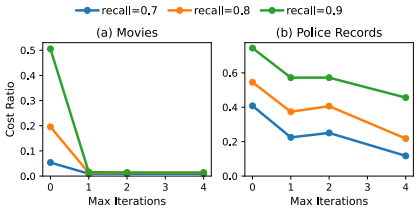

Figure 10: Impact of max_iter

cost ratio using gpt-4.1-nano (we estimate cost based on number of input tokens, as was discussed in Sec. 8.1). The results are shown in Fig. 9, where we see two orders of magnitude difference in cost between FDJ and BARGAIN on the full dataset. FDJ costs only about US$54 on the largest dataset, while BARGAIN costs about US$28,515, showing that FDJ can make large scale joins feasible.

Sensitivity to max_iter and α. Fig. 10 shows the impact of max_iter and α for two of our datasets, plotting how the cost ratio changes with max_iter for different recall targets (results for α and different precision targets were similar). Overall, the trends are almost identical for different recall thresholds, with the best max_iter value being independent of the threshold. In practice, FDJ's cost ratio monotonically decreases with max_iter and increases with α so that any large max_iter (we set max_iter=4 for all datasets) and small α (we set $\alpha = 10$ for all datasets) is suitable. To see why the best max_iter and α are independent of the quality target, observe that max_iter and α only impact how many featurizations are generated—the larger max_iter and the smaller α, the more featurizations are generated—but not how they are used which is decided by our CNF construction algorithm. Although how featurizations are used does depend on the precision and recall thresholds (see Fig. 8 which shows that the CNFs created changes based on recall thresholds), generating more featurizations only provides more options for CNF generation, without causing quality degradation.

9 RELATED WORK

We next survey the related work, discussing methods that perform AI-powered data management, semantic joins and entity resolution.

LLM-Powered Data Management. LLMs have now been applied to a large array of data management problems: data discovery [19, 65], data extraction and cleaning [7, 36, 46, 47, 63], query planning [60], and text to SQL [51]; while other work have built systems to flexibly process data with LLMs in open-ended ways [5, 26, 38, 39, 49, 55, 57, 61, 64, 70]. FDJ reduces costs while guaranteeing quality, making it applicable to any LLM-based system that supports semantic joins, including [16, 20, 49, 55, 56].

Cost-Efficient AI-Powered Data Processing. Many techniques have been used to reduce the cost of ML-powered data processing, some examples include: optimizing aggregations through approximation [26, 27], building indexes to reduce online query processing time [8, 30], performing profiling to estimate model accuracies to decide which model to use [25], and ensembling various models to generate the final output [22]. *Model cascade*, which routes queries through cheaper proxy models before using expensive oracle models, has been widely adopted in traditional ML and deep learning, particularly for video analytics [6, 10, 14, 27, 28, 40]. In particular, recent work [12, 49, 72] has used this framework for LLM-powered data processing, specifically for filters. Featurized decomposition in FDJ parallels model cascades by using a cheap logical expression to approximate outputs before invoking an expensive model. We automatically construct such logical expressions and show they empirically outperforms existing cascade methods for joins.

Semantic Joins. Joining tables with semantically related columns [3, 13, 21, 66] pre-dates LLMs. Recently, LLMs have become the standard solution for joining tables based on a user-specified semantic relationship [4, 16, 18, 20, 49, 55, 56, 68]. Most work does not optimize such joins, except [49] which uses model cascades. As discussed above and empirically shown, FDJ outperforms cascade-based solutions. A recent preprint [59] uses batching to reduce join costs. Batching is orthogonal to our work, and can be combined with FDJ to further reduce costs.

Entity Resolution. Entity resolution (ER) is a well-established problem in data management [9, 17, 31, 42, 44, 48, 58, 69, 73], which can be seen as a special case of semantic joins. Most related work on ER focuses on a relational setting and does not support arbitrary natural language joins on text data. More specifically, related work can be categorized into non-deep learning, e.g., [9, 31, 33] and deep learning work e.g., [17, 58, 69, 73]. The former category assumes existence of a set of attributes (or features) that can be used for blocking, different from FDJ that operates on raw text with any natural language join instruction and thus must automatically discover and extract attributes. Furthermore, [9, 31, 33] leverage blocking rules on pre-specified attributes to remove as many false positive matches without introducing any false negatives, unlike our setting that allows a more relaxed recall target (and therefore much lower costs) but requires statistical guarantees on meeting this user-provided target. Thus, FDJ is not comparable to [9, 31, 33]. The latter category trains task-specific models to learn blocking rules but once again operates on tabular data [17, 58, 69, 73] and designs method to encode such tabular rows, unlike FDJ that operates on raw text columns with any natural language join instruction.

10 CONCLUSION

We presented featurized decomposition, a new technique for optimization semantic joins by defining logical expressions using feature extraction functions. Using this technique, we developed FDJ, a new method for performing semantic joins efficiently and reliably. FDJ automatically constructs featurized decompositions that accurately estimate whether the join condition holds, leading to substantial cost savings. We furthermore performed an in-depth theoretical analysis of FDJ to provide tight theoretical bounds that guarantee meeting user-specified quality requirements. Our results across real-world datasets demonstrate significant cost reductions.

ACKNOWLEDGMENTS

We acknowledge support from grants DGE-2243822, IIS-2129008, IIS-1940759, and IIS-1940757 awarded by the National Science Foundation, funds from the State of California, funds from the Alfred P. Sloan Foundation, as well as EPIC lab sponsors: Adobe, Google, G-Research, Microsoft, PromptQL, Sigma Computing, Bridgewater, and Snowflake. Compute credits were provided by Azure, Modal, NSF (via NAIRR), and OpenAI.

REFERENCES

[1] 2025. California Police Records Access Project. https://bids.berkeley.edu/california-police-records-access-project.

[2] 2026. XGBoost Documentation. https://xgboost.readthedocs.io/en/release_3.2.0/.

[3] Ziawasch Abedjan, John Morcos, Ihab F Ilyas, Mourad Ouzzani, Paolo Papotti, and Michael Stonebraker. 2016. Dataxformer: A robust transformation discovery system. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. IEEE, 1134–1145.

[4] PG AI. [n. d.]. Pgai: Giving PostgreSQL Developers AI Engineering Superpowers. urlhttps://github.com/timescale/pgai?tab=readme-ov-file. Accessed: 2025-06-22.

[5] Eric Anderson, Jonathan Fritz, Austin Lee, Bohou Li, Mark Lindblad, Henry Lindeman, Alex Meyer, Parth Parmar, Tanvi Ranade, Mehul A Shah, et al. 2024. The Design of an LLM-powered Unstructured Analytics System. *arXiv preprint arXiv:2409.00847* (2024).

[6] Michael R Anderson, Michael Cafarella, German Ros, and Thomas F Wenisch. 2019. Physical representation-based predicate optimization for a visual analytics database. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1466–1477.

[7] Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. 2023. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433* (2023).

[8] Favyen Bastani and Samuel Madden. 2022. OTIF: Efficient tracker pre-processing over large video datasets. In *Proceedings of the 2022 International Conference on Management of Data*. 2091–2104.

[9] Mikhail Bilenko, Beena Kamath, and Raymond J Mooney. 2006. Adaptive blocking: Learning to scale up record linkage. In *Sixth international conference on data mining (ICDM'06)*. IEEE, 87–96.

[10] Jiashen Cao, Karan Sarkar, Ramyad Hadidi, Joy Arulraj, and Hyesoon Kim. 2022. Figo: Fine-grained query optimization in video analytics. In *Proceedings of the 2022 International conference on management of data*. 559–572.

[11] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. 2023. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735* (2023).

[12] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176* (2023).

[13] Arash Dargahi Nobari and Davood Rafiei. 2024. Dtt: An example-driven tabular transformer for joinability by leveraging large language models. *Proceedings of the ACM on Management of Data* 2, 1 (2024), 1–24.

[14] Dujian Ding, Sihem Amer-Yahia, and Laks Lakshmanan. 2022. On Efficient Approximate Queries over Machine Learning Models. *Proceedings of the VLDB Endowment (PVLDB)* 16, 4 (2022), 918–931.

[15] Karel D'Oosterlinck, François Remy, Johannes Deleu, Thomas Demeester, Chris Develder, Klim Zaporojets, Aneiss Ghodsi, Simon Ellershaw, Jack Collins, and Christopher Potts. 2023. BioDEX: Large-scale biomedical adverse drug event extraction for real-world pharmacovigilance. *arXiv preprint arXiv:2305.13395* (2023).

[16] Till Döhmen. 2024. Introducing the prompt() Function: Use the Power of LLMs with SQL! https://motherduck.com/blog/sql-llm-prompt-function-gpt-models/. Accessed: 2025-06-22.

[17] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq Joty, Mourad Ouzzani, and Nan Tang. 2017. DeepER–Deep Entity Resolution. *arXiv preprint arXiv:1710.00597* (2017).

[18] Flock. [n. d.]. Integrating LM and IR Capabilities in SQL. urlhttps://dais-polymtl.github.io/flock/. Accessed: 2025-06-22.

[19] Juliana Freire, Grace Fan, Benjamin Feuer, Christos Koutras, Yurong Liu, Eduardo Pena, Aécio Santos, Cláudio Silva, and Eden Wu. [n. d.]. Large Language Models for Data Discovery and Integration: Challenges and Opportunities. *Data Engineering* ([n. d.]), 3.

[20] Google. 2025. Perform intelligent SQL queries using AlloyDB AI query engine. http://cloud.google.com/alloydb/docs/ai/evaluate-semantic-queries-ai-operators.

[21] Yeye He, Kris Ganjam, and Xu Chu. 2015. Sema-join: joining semantically-related tables using big table corpora. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1358–1369.

[22] Keke Huang, Yimin Shi, Dujian Ding, Yifei Li, Yang Fei, Laks Lakshmanan, and Xiaokui Xiao. 2025. ThriftLLM: On Cost-Effective Selection of Large Language Models for Classification Queries. *arXiv preprint arXiv:2501.04901* (2025).

[23] IMDB. 2025. IMDb Non-Commercial Datasets. https://developer.imdb.com/non-commercial-datasets/.

[24] Saehan Jo and Immanuel Trummer. 2016. Thalamusdb: Approximate query processing on multi-modal data (code). https://github.com/itrummer/thalamusdb.

[25] Saehan Jo and Immanuel Trummer. 2024. SMART: Automatically Scaling Down Language Models with Accuracy Guarantees for Reduced Processing Fees. *arXiv preprint arXiv:2403.13835* (2024).

[26] Saehan Jo and Immanuel Trummer. 2024. Thalamusdb: Approximate query processing on multi-modal data. *Proceedings of the ACM on Management of Data* 2, 3 (2024), 1–26.

[27] Daniel Kang, Peter Bailis, and Matei Zaharia. [n. d.]. BlazeIt: Optimizing Declarative Aggregation and Limit Queries for Neural Network-Based Video Analytics. *Proceedings of the VLDB Endowment* 13, 4 ([n. d.]).

[28] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Neural Network Queries over Video at Scale. *Proceedings of the VLDB Endowment* 10, 11 (2017).

[29] Daniel Kang, Edward Gan, Peter Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2020. Approximate Selection with Guarantees using Proxies. *Proceedings of the VLDB Endowment* 13, 11 (2020).

[30] Daniel Kang, John Guibas, Peter D Bailis, Tatsunori Hashimoto, and Matei Zaharia. 2022. Tasti: Semantic indexes for machine learning-based queries over unstructured data. In *Proceedings of the 2022 International Conference on Management of Data*. 1934–1947.

[31] Mayank Kejriwal and Daniel P Miranker. 2013. An unsupervised algorithm for learning blocking schemes. In *2013 IEEE 13th International Conference on Data Mining*. IEEE, 340–349.

[32] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. 2022. Decomposed prompting: A modular approach for solving complex tasks. *arXiv preprint arXiv:2210.02406* (2022).

[33] Kunho Kim, Athar Sefid, and C Lee Giles. 2017. Scaling author name disambiguation with CNF blocking. *arXiv preprint arXiv:1709.09657* (2017).

[34] Hanna Köpcke, Andreas Thor, and Erhard Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 484–493.

[35] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, et al. 2026. SemBench: A Benchmark for Semantic Query Processing Engines. *VLDB* (2026).

[36] Alexander W Lee, Justin Chan, Michael Fu, Nicolas Kim, Akshay Mehta, Deepti Raghavan, and Ugur Cetintemel. 2025. Semantic Integrity Constraints: Declarative Guardrails for AI-Augmented Data Processing Systems. *arXiv preprint arXiv:2503.00600* (2025).

[37] David D Lewis. 1995. A sequential algorithm for training text classifiers: Corrigendum and additional data. In *Acm sigir forum*, Vol. 29. ACM New York, NY, USA, 13–19.

[38] Yiming Lin, Madelon Hulsebos, Ruiying Ma, Shreya Shankar, Sepanta Zeigham, Aditya G Parameswaran, and Eugene Wu. 2024. Towards accurate and efficient document analytics with large language models. *arXiv preprint arXiv:2405.04674* (2024).

[39] Chunwei Liu, Matthew Russo, Michael Cafarella, Lei Cao, Peter Baille Chen, Zui Chen, Michael Franklin, Tim Kraska, Samuel Madden, and Gerardo Vitagliano. 2024. A declarative system for optimizing ai workloads. *arXiv preprint arXiv:2405.14696* (2024).

[40] Yao Lu, Aakanksha Chowdhery, Srikanth Kandula, and Surajit Chaudhuri. 2018. Accelerating machine learning inference with probabilistic predicates. In *Proceedings of the 2018 International Conference on Management of Data*. 1493–1508.

[41] Robert Mahari, Dominik Stammbach, Elliott Ash, and Alex Pentland. 2024. LePaRD: A large-scale dataset of judicial citations to precedent. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. Association for Computational Linguistics, 9863–9877.

[42] Adam Marcus, Eugene Wu, David Karger, Samuel Madden, and Robert Miller. 2011. Human-powered sorts and joins. *arXiv preprint arXiv:1109.6881* (2011).

[43] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. 165–172.

[44] Matthew Michelson and Craig A Knoblock. 2006. Learning blocking schemes for record linkage. In *AAAI*, Vol. 6. 440–445.

[45] 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.

[46] Zan Ahmad Naeem, Mohammad Shahmeer Ahmad, Mohamed Eltabakh, Mourad Ouzzani, and Nan Tang. 2024. RetClean: Retrieval-Based Data Cleaning Using LLMs and Data Lakes. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4421–4424.

[47] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).

[48] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2019. A survey of blocking and filtering techniques for entity resolution. *arXiv preprint arXiv:1905.06167* (2019).

[49] Liana Patel, Siddharth Jha, Carlos Guestrin, and Matei Zaharia. 2025. Lotus: Enabling semantic queries with llms over tables of unstructured and structured data. *VLDB* (2025).

[50] Ellie Pavlick, Heng Ji, Xiaoman Pan, and Chris Callison-Burch. 2016. The gun violence database: A new task and data set for nlp. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. 1018–1024.

[51] Mohammadreza Pourreza, Hailong Li, Ruoxi Sun, Yeounoh Chung, Shayan Talaei, Gaurav Tarlok Kakkar, Yu Gan, Amin Saberi, Fatma Ozcan, and Sercan O Arik. 2024. Chase-sql: Multi-path reasoning and preference optimized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943* (2024).

[52] 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. 269.

[53] Alexander J Ratner, Christopher M De Sa, Sen Wu, Daniel Selsam, and Christopher Ré. 2016. Data programming: Creating large training sets, quickly. *Advances in neural information processing systems* 29 (2016).

[54] Burr Settles. 2009. Active learning literature survey. (2009).

[55] Shreya Shankar, Tristan Chambers, Tarak Shah, Aditya G Parameswaran, and Eugene Wu. 2024. DocETL: Agentic Query Rewriting and Evaluation for Complex Document Processing. *arXiv preprint arXiv:2410.12189* (2024).

[56] Snoflake. 2025. Introducing Cortex AISQL: Reimagining SQL into AI Query Language for Multimodal Data. https://www.snowflake.com/en/blog/ai-sql-query-language/.

[57] Zhaoze Sun, Qiyan Deng, Chengliang Chai, Kaisen Jin, Xinyu Guo, Han Han, Ye Yuan, Guoren Wang, and Lei Cao. 2025. Quest: Query optimization in unstructured document analysis. *VLDB* (2025).

[58] Saravanan Thirumuruganathan, 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), 2459–2472.

[59] Immanuel Trummer. 2025. Implementing Semantic Join Operators Efficiently. *arXiv preprint arXiv:2510.08489* (2025).

[60] Matthias Urban and Carsten Binnig. 2024. Demonstrating CAESURA: Language Models as Multi-Modal Query Planners. In *Companion of the 2024 International Conference on Management of Data*. 472–475.

[61] Matthias Urban and Carsten Binnig. 2024. ELEET: Efficient Learned Query Execution over Text and Tables. *Proc. VLDB Endow* 17 (2024), 13.

[62] Manik Varma, Kush Bhatia, and Kunal Dahiya. 2019. The extreme classification repository: Multi-label datasets & code. http://manikvarma.org/downloads/XC/XMLRepository.html.

[63] David Vos, Till Döhmen, and Sebastian Schelter. 2022. Towards parameter-efficient automation of data wrangling tasks with prefix-tuning. In *NeurIPS 2022 First Table Representation Workshop*.

[64] Jiayi Wang and Guoliang Li. 2025. Aop: Automated and interactive llm pipeline orchestration for answering complex queries. CIDR.

[65] Qiming Wang and Raul Castro Fernandez. 2023. Solo: Data Discovery Using Natural Language Questions Via A Self-Supervised Approach. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–27.

[66] Yue Wang and Yeye He. 2017. Synthesizing mapping relationships using table corpus. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1117–1132.

[67] Orion Weller, Michael Boratko, Iftekhar Naim, and Jinhyuk Lee. 2025. On the theoretical limitations of embedding-based retrieval. *arXiv preprint arXiv:2508.21038* (2025).

[68] Patrick Wendell, Eric Peter, Nicolas Pelaez, Jianwei Xie, Vinny Vijeyakumaar, Linhong Liu, and Shitao Li. 2023. Introducing AI Functions: Integrating Large Language Models with Databricks SQL. https://www.databricks.com/blog/2023/04/18/introducing-ai-functions-integrating-large-language-models-databricks-sql.html. Accessed: 2025-06-22.

[69] Shiwen Wu, Qiyu Wu, Honghua Dong, Wen Hua, and Xiaofang Zhou. 2023. Blocker and matcher can mutually benefit: a co-learning framework for low-resource entity resolution. *Proceedings of the VLDB Endowment* 17, 3 (2023), 292–304.

[70] Sepanta Zeighami, Yiming Lin, Shreya Shankar, and Aditya Parameswaran. 2025. LLM-Powered Proactive Data Systems. *arXiv preprint arXiv:2502.13016* (2025).

[71] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. 2025. Featurized-Decomposition Join: Low-Cost Semantic Joins with Guarantees. https://arxiv.org/pdf/2512.05399.

[72] Sepanta Zeighami, Shreya Shankar, and Aditya Parameswaran. 2026. Cut Costs, Not Accuracy: LLM-Powered Data Processing with Guarantees. *SIGMOD'26* (2026).

[73] 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.