Customizable and Scalable Fuzzy Join for Big Data

Zhimin Chen Yue Wang Vivek Narasayya Surajit Chaudhuri
Microsoft Research, Redmond, WA, US
{zmchen, wang.yue, viveknar, surajitc}@microsoft.com

ABSTRACT

Fuzzy join is an important primitive for data cleaning. The ability to customize fuzzy join is crucial to allow applications to address domain-specific data quality issues such as synonyms and abbreviations. While efficient indexing techniques exist for single-node implementations of customizable fuzzy join, the state-of-the-art scale-out techniques do not support customization, and exhibit poor performance and scalability characteristics. We describe the design of a scale-out fuzzy join operator that supports customization. We use a locality-sensitive-hashing (LSH) based signature scheme, and introduce optimizations that result in significant speed up with negligible impact on recall. We evaluate our implementation on the Azure Databricks version of Spark using several real-world and synthetic data sets. We observe speedups exceeding 50X compared to the best-known prior scale-out technique, and close to linear scalability with data size and number of nodes.

PVLDB Reference Format:
Zhimin Chen, Yue Wang, Vivek Narasayya, Surajit Chaudhuri. Customizable and Scalable Fuzzy Join for Big Data. *PVLDB*, 12(12): 2106-2117, 2019.
DOI: https://doi.org/10.14778/3352063.3352128

1. INTRODUCTION

Record linkage [25], also known as record matching, is an important task in data cleaning, and helps in preparing data for more accurate analysis. *Fuzzy join* (also referred to as *set-similarity join* or *fuzzy matching*) is a powerful operator used in record matching that can efficiently identify pairs of records that are similar to each other according to a given similarity function. Given a *reference table* R and an *input table* S, for each record $s \in S$ the fuzzy join operator returns all records $r \in R$ such that $sim(s, r) \geq \theta$, where sim is a similarity function and θ is a user-specified threshold. Commonly used similarity functions include Soundex, Levenshtein distance (edit distance), Hamming distance, cosine similarity, Jaro-Winkler similarity, Jaccard similarity etc.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352128

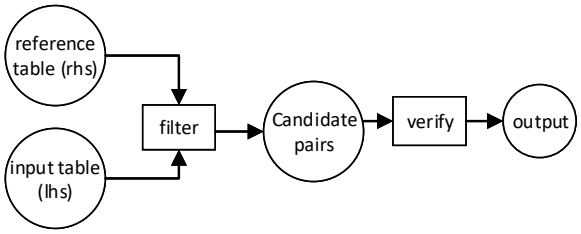

Figure 1: Filtering-Verification architecture for fuzzy join

As an example scenario, if R is a table of existing customers of an enterprise that acquired another company, and S is a table of customers of the acquired enterprise, fuzzy join can be helpful in identifying which customers the two companies share in common and which new customers were acquired.

Many approaches have been adopted in industry and research for supporting fuzzy join. One approach is *vertical* (i.e. domain-specific) solutions. For example, for the important domain of addresses, custom solutions such as Trillium [37] and Melissa [26] have been developed. Another approach in the industry is to provide fuzzy join capability as part of a platform, allowing applications to develop their own record matching solutions. A few examples of such platforms are Informatica [23], Microsoft SQL Server Integration Services (SSIS) [35], Knime [24], and Talend [36]. These approaches expose a fixed menu of similarity functions to use for matching.

The naïve approach of evaluating the similarity function on each pair of records in $S \times R$ is not feasible except for very small data sets. Therefore, most prior techniques e.g., [21, 32, 5] use a filtering-verification architecture as depicted in Figure 1. The filtering step uses a signature-based algorithm to generate a set of signatures for each string in S and R. The signatures have the correctness property: if $sim(s, r) \geq \theta$, then s and r share at least one common signature. Since set overlap can be tested using an equi-join, a big data engine or relational database engine can be used for evaluating this step. Signature schemes have been proposed for several common similarity functions such as edit distance, Jaccard similarity etc. In the *verification* step, the similarity function is invoked for each surviving candidate pair (s, r), and only those pairs for which the similarity exceeds the given threshold are output.

To obtain good recall for record matching it is necessary to capture differences between records s and r that can arise due to various data quality issues such as edit errors, ab-

breviations, and synonyms. It is therefore important that the similarity function is *customizable* by the application. Customization can be domain or application specific. For instance, an application may want to specify that Bob and Robert are synonyms in a column containing first names. The work by Arasu et al. [4] develops a transformation-based framework for fuzzy join where customization is expressed via transformation rules (e.g. $Bob \rightarrow Robert$). To obtain good performance, they use locality sensitive hashing (LSH) [20] to generate signatures for each record, and create an index on the signatures over the reference table R. Unlike prefix filtering, which is an exact method, LSH's randomized algorithm guarantees correctness *with high probability* by generating multiple signatures. The LSH approach for signature generation significantly improves upon previous techniques and results in superior performance, even while supporting customizable similarity. This approach also parallelizes well across multiple cores of a single node since lookups for different s records can proceed in parallel against the index [3].

Once the reference table R becomes large, a single-node solution is no longer feasible. Therefore, *scale-out* approaches including [18, 39, 8, 33, 27, 31, 13, 16, 17, 30] have been developed on MapReduce [15] engines such as Hadoop. A recent experimental study by Fier et al. [19] compared several scale-out techniques. Based on these results, the state-of-the-art scale-out fuzzy join techniques are lacking in two fundamental ways. First, their scalability is limited. For instance, the top ranked technique in the above study: Vernica et al. (VJ) ([39]) uses a variant of a popular signature scheme called *prefix filtering* [10]. However, the scalability of this approach is sensitive to frequent tokens and memory in the reducers becomes a bottleneck, which often leads to job failure or eventual timeout. Intermediate result sizes, and therefore the data shuffling cost during the execution of the equi-join step can be very large. Further, prefix filtering is not very selective except when the similarity threshold θ is very high (e.g. 0.95). Thus, the verification step can also be very expensive. In practice, lower values of θ around 0.8 are often necessary to obtain the desired precision-recall trade-off in record matching. Second, existing scale-out techniques lack the customizability available in single-node approaches such as [4] described above. In fact, as we show in this paper, supporting customizations to handle edit errors, abbreviations and synonyms can further exacerbate the scalability problem of prefix filtering.

We have developed a scale-out fuzzy join operator that supports transformation-based customizability [4]. Analogous to joins in parallel DBMSs, for cases where the input table S is large but the reference table R is small enough for the index to fit into the main memory of a single node, we develop a *broadcast fuzzy join* technique wherein the index on R is broadcast (i.e. replicated) to multiple worker nodes and the input table S is partitioned across those nodes. The more challenging case is when the index on R does not fit into the main memory of a single node. For this case, we develop a *shuffle fuzzy join* technique where both S and R are partitioned across nodes. In contrast to prior scale-out approaches such as VJ [39] that use prefix filtering as the signature scheme, we use LSH to generate signatures. One implication of using LSH, which is a randomized method, is that in practice, all signatures except a few, have very low frequency. As an example, in one of the real-world datasets we have experimented with, less than 1% all signatures are associated with more than 10 rows in the dataset. Although a few signatures are in fact very frequent, these can be pruned with negligible impact on matching recall since there is sufficient redundancy of signatures per row. This pruning results in dramatic reduction in data shuffle cost since the signatures that are very frequent in either one or both tables do not need to be joined. We observe that such pruning is ineffective for prefix filtering based techniques since: (a) prefix filtering does not have the redundancy of signatures present in LSH, and (b) the frequency distribution of signatures is not as skewed; hence the impact of pruning on recall is too high.

We have implemented the broadcast and shuffle versions of scale-out fuzzy join on Spark [40]. Our fuzzy join operator is potentially applicable in multiple data preparation platforms such as Azure Data Factory [1], Microsoft Power Query [29] and Azure Machine Learning Data Prep SDK [2].

One of the key contributions of this paper is a thorough empirical evaluation of performance and scale. We report the results of experiments run on Azure Databricks [6] using several real-world and synthetic data sets. Some of the key findings of our empirical study are:

- Fuzzy join that uses LSH signatures is significantly faster than a prefix filtering based technique.
- Our technique of pruning high-frequency LSH signatures provides large speedups (exceeding 50× for some datasets) with negligible impact on matching recall.
- The shuffle and broadcast versions of fuzzy join scale close to linearly with the number of nodes and size of the data.
- In cases where broadcast fuzzy join is applicable, it is faster than the shuffle version.

The rest of the paper is organized as follows. We first review in Section 2 the need for customization, and how the similarity function can be made customizable. In Section 3 we review prefix filtering and LSH based signature generation schemes, and describe optimized implementations for them. We describe the data flow pipelines on Spark for broadcast and shuffle fuzzy join in Section 4; and provide an analysis of the cost of the pipelines. We present the results of experiments in Section 5, discuss related work in Section 6 and conclude in Section 7.

2. CUSTOMIZING FUZZY JOIN

Formally, fuzzy join is an operator parameterized with a similarity function sim and a threshold θ that takes as input two relations S and R as input and returns for each row in S all rows in R whose similarity is above the specified threshold, that is, $FJ_{sim,\theta}(S,R) = \{\langle s,r\rangle | s \in S, r \in R, sim(s,r) \geq \theta\}$.

The need for *customizing* fuzzy join is ubiquitous. While a particular similarity function (e.g. edit distance) can handle one class of data quality issues, no individual similarity function can handle the large and diverse class of issues, some of which can be domain or application specific. From our prior experience with Microsoft's Bing Maps service [3] as well as other Microsoft internal applications that require fuzzy join we observe several examples. For instance, Bing Maps, needs to match user queries against a reference table

of points of interest (e.g. landmarks, businesses, addresses) in many countries across the world. The similarity function must be able to handle edit errors due to misspellings (Space Needle ↔ Space Neede), token merge and split issues (DisneyLand ↔ Disney Land), abbreviations (United States ↔ US ↔ U.S., and Ave ↔ Avenue), synonyms (Xing ↔ Crossing, 1st ↔ First). Furthermore, synonyms can be country specific. In other applications in different domains, e.g. involving people names, synonyms such as Robert ↔ Bob, which are far apart in terms of string similarity, need to be specifiable via the similarity function.

2.1 Core Similarity Function

We follow the approach in [4] which uses weighted Jaccard as the core similarity function, and a transformation rule based framework for expressing customizations such as edit errors, abbreviations, synonyms etc. Here we briefly review the framework, and provide examples to illustrate how customization is achieved.

Given two strings a and b, we use a tokenization function to convert them into two sequences of tokens $[a_1, a_2, ..., a_m]$ and $[b_1, b_2, ..., b_n]$. A weighting function w assigns a weight to a token. We use the Inverse Document Frequency (idf) weight [7] by default, which models the intuition that less frequent words should carry more weight for determining similarity.

$$w_i(token) = log(\frac{total\_\#rows}{\#rows\_containing\_token\_in\_column\_i})$$

The similarity of a and b is defined as the weighted Jaccard similarity of the multiset $A = \{a_1, a_2, ..., a_m\}$ and $B = \{b_1, b_2, ..., b_n\}$, i.e.,

$$sim([a_1, a_2, ..., a_m], [b_1, b_2, ..., b_n]) = \frac{\sum_{x \in A \cap B} w(x)}{\sum_{x \in A \cup B} w(x)}$$

Since we use multiset similarity, the order of tokens does not affect similarity but multiplicity of a token is counted accordingly.

2.2 Customization Using Transformation Rule

We define a string *transformation rule* as $\langle lhs \rightarrow rhs \rangle$, where lhs and rhs are sequences of tokens and rhs can be empty. An application of a transformation rule on a token sequence s substitutes each matching sequence of lhs in s with the sequence rhs.

For example, the transformation $\langle$Micosoft $\rightarrow$ Microsoft$\rangle$ when applied on a token sequence [Micosoft, Corporation] results in [Microsoft, Corporation]. To apply more than one transformation on a token sequence, we require that the matching of the lhs of the transformation must come from the original token sequence, and not from a sequence of tokens substituted using another rule.

Given a *set* of transformation rules, we apply all subsets of rules that are relevant to s. The application of each subset generates a variant of s. The similarity between s and a record r in the reference table is the Jaccard similarity of the variant with the highest Jaccard similarity. In the case of edit transformations, these transformations are generated *programmatically* by an edit transformation provider, which is a function that uses the distinct tokens in the reference table to identify those variants of tokens in s that are within the specified edit distance. Other classes of transformations such as token merge and split, acronyms, abbreviations etc. can also be generated programmatically by transformation providers. Figure 2 shows an example of how the similarity function is customized under the application of edit transformation rules. Without application of edit transformations, the Jaccard similarity between the original records is $\frac{3}{8}$. Application of edit transformations results in three additional variants. The maximum Jaccard similarity over all variants is boosted to $\frac{5}{6}$.

2.3 Multi-column Records

In the case of multi-column records, we define similarity as the transformation-based weighted Jaccard similarity between tokens across all columns. The token weights are multiplied by configurable column weights, so identical tokens from different columns may have different weights. A user could further specify column level similarity thresholds between pairs of columns as a post-processing step to refine the quality of matched records.

In the rest of the paper, for simplicity, we describe our techniques for a single-column record (or table), and do not distinguish between string similarity and record similarity.

3. SIGNATURE GENERATION

The signature scheme determines the selectivity of the filtering step (see Figure 1), which affects both the cost of joining signatures from tables R and S, and also the cost of the verification step since a more selective signature scheme will require fewer row pairs to be verified. In this section, we first review two well-known signature schemes: prefix filtering and locality sensitive hashing (LSH). We then describe optimized algorithms to generate signatures for the case of customization using weighted tokens and transformations. Finally, we compare the effectiveness of the two signature schemes. We observe that while the techniques and optimizations described below are applicable in both single-node and scale-out approaches to fuzzy join, the pruning optimization for LSH described in Section 3.4 has a much more pronounced impact on performance and scalability in the scale-out scenario.

3.1 Prefix Filtering

Prefix filtering sorts the tokens in a string s by their weights (and use token id as tie breaker because prefix filtering requires a stable global total order among all tokens). Let the sorted sequence denoted as $[s_1, s_2, ..., s_m]$. Let i be the smallest index such that

$$\frac{\sum_{k=1}^{i} w(s_k)}{\sum_{k=1}^{n} w(s_k)} \geq (1 - \theta)$$

where θ is the similarity threshold. Then tokens $\{s_1, ..., s_i\}$ are the signatures of s because any string with weighted Jaccard similarity higher than θ must include one of the tokens in $\{s_1, ..., s_i\}$. In the presence of transformations, let T be all the variants that s generates, then the signatures for s is $\cup_{t \in T}\{signatures(t)\}$. If the transformations are all one-to-one, e.g., in the case with edit distance transformations, then there is no need to enumerate all the variants to generate the signatures and only need to check the variants generated by replacing all tokens with their minimum weight transformed tokens. For example, let s be [a, b, c], and A, B, C are the transformed tokens respectively, let a'

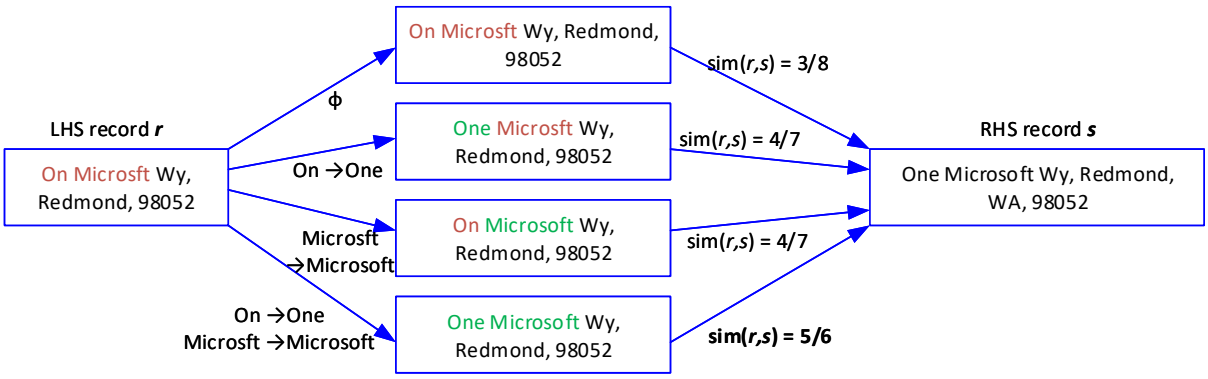

Figure 2: Example of matching in the presence of edit transformations

be the token of minimum weight among $\{a\} \cup A$, b' be the token of minimum weight among $\{b\} \cup B$, to check whether c should be a signature, we only need to check whether c is in the prefix of $\{a', b', c\}$.

3.2 Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing (LSH) [20] is another well-known signature scheme used for set similarity join. It uses $k \times m$ independent hash functions. Let h be one of the hash functions and the set of tokens be $\{s_1, ..., s_n\}$. Then, the min hash token under h is defined as $\arg\min_{x \in \{s_1,...,s_n\}} h(x)$. Let y_i be the min hash token under h_i for $1 \leq i \leq (k \times m)$. LSH divides them into m groups and each group has k min hash tokens, i.e., $((y_1, ..., y_k), (y_{k+1}, ..., y_{2k}), ...)$, then generates m signatures by hashing each group, i.e., $signature_1 = h'(y_1, y_2, ..., y_k)$, $signature_2 = h'(y_{k+1}, y_{k+2}, ..., y_{2k})$, etc. where h' is a different hash function. It can be shown that for any two strings with Jaccard similarity higher than θ the above signature scheme has higher than $1-(1-\theta^k)^m$ probability to generate at least one of $\{signature_1, ..., signature_m\}$ for both strings. In our implementation we use $k = 4$ and $m = 6$ as default, for $\theta \geq 0.8$, $1 - (1 - \theta^k)^m \geq 0.95$

We use the approach in [4] to extend the above LSH scheme to weighted Jaccard similarity. Let s_i be a token in string s and $w(s_i)$ be its weight, and h be one of the hash functions (we use the MurmurHash3 [28] in the standard Scala library). Instead of using $h(s_i)$ as the hash value to compute the min hash token, it first maps $h(s_i)$ uniformly to a number between 0 and 1, denoted as h', and use $-log(h')/w(s_i)$ as the hash value [12]. The rest of procedure to generate signatures is the same.

In the presence of transformations, the baseline method to generate signatures is to enumerate all the variants of s and generate signatures for each of the variants. If the transformations are all one-to-one, sometimes it's more efficient to generate the signatures in alternative way. Here is an example. Let s be a three-token string [a, b, c], and A, B, C are the transformed tokens respectively, so $A' = \{a\} \cup A$, $B' = \{b\} \cup B$, and $C' = \{c\} \cup C$ are the token domains at each position. For $k = 1$ meaning we use a single hash function h to generate a signature, let H be the min hash tokens from all possible combinations of variants:

$$H = \cup_{a' \in A', b' \in B', c' \in C'} \{ \arg\min_{x \in \{a', b', c'\}} h(x) \}$$

which is equivalent to:

$$H = \{y | y \in A' \cup B' \cup C' \wedge h(y) <= v\}$$

where v is defined as $v = min(max_{a' \in A'} h(a'), max_{b' \in B'} h(b'), max_{c' \in C'} h(c'))$. Therefore instead of enumerating the cross product of $A' \times B' \times C'$, it only needs a linear scan of $A' \cup B' \cup C'$.

When $k = 2$, that is, we use 2 hash functions h_1 and h_2 to generate a signature, and then the min hash tokens H is:

$$H = \cup_{a' \in A', b' \in B', c' \in C'} \{ \langle \arg\min_{x \in \{a', b', c'\}} h_1(x), \arg\min_{x \in \{a', b', c'\}} h_2(x) \rangle \}$$

Let $H_1 = \{y | y \in A' \cup B' \cup C' \wedge h_1(y) <= v_1\}$, $H_2 = \{y | y \in A' \cup B' \cup C' \wedge h_2(y) <= v_2\}$, then $H \subseteq H_1 \times H_2$. Note that $H_1 \times H_2$ may be a superset of min hash tokens H because two hash functions may achieve minimum at the same token. For example, let $A' = \{a_1, a_2\}$, $B' = \{b_1, b_2\}$, and $C' = \{c\}$. Suppose $h_1(a_1) < h_1(c) < h_1(a_2) < h_1(b_i)$, and $h_2(a_1) < h_2(c) < h_2(a_2) < h_2(b_i)$. The min hash tokens under $\langle h_1, h_2 \rangle$ is $H = \{\langle a_1, a_1 \rangle, \langle c, c \rangle\}$, which is a subset of $H_1 \times H_2$, where $H_1 = H_2 = \{a_1, c\}$.

Similarly for more hash functions, the set of min hash tokens is a subset of $\prod H_i$. $|\prod H_i|$ can be smaller than the number of variants when the weights of variant tokens are high, suggesting that generating $\prod H_i$ can be more efficient than enumerating variants.

3.3 Comparison

Below we summarize some of the key differences between prefix filtering and LSH.

Exact vs. Probabilistic: Prefix filtering is an exact scheme with respect to the guarantee of not missing any pair of rows with similarity higher than the specified threshold. On the other hand LSH provides this guarantee with high probability. The probability can be tuned by choosing different (k, m) parameters described above. In the default setting of our implementation ($k = 4, m = 6$), the theoretical guarantee is a probability higher than 0.95 for a similarity threshold 0.8. In practice in our target use case of joining relatively short records (around 20 tokens) with similarity 0.8, experiments show that the actual recall is much higher (≥ 0.999) than that theoretical bound.

Signature generation cost: Prefix filtering typically generates fewer signatures per row than LSH, and is less computationally intensive than LSH, which requires invocation

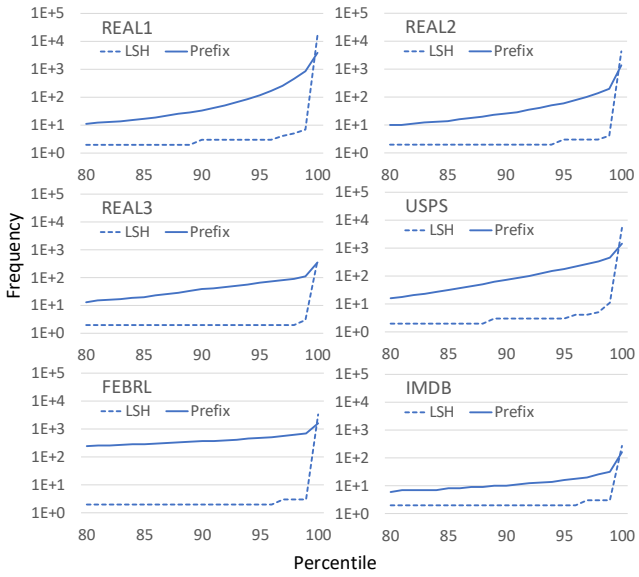

Figure 3: Signature frequency distribution: LSH vs Prefix-Filtering

of several hash functions per record. Thus the signature generation cost of prefix filtering is lower.

Distribution of signature frequency: Although prefix filtering generates fewer signatures per row than LSH does, most signatures generated using prefix filtering are typically associated with many more rows in the table than a signature generated using LSH. For several real-world datasets and synthetic datasets (see Section 5 for descriptions of these datasets), we analyzed the distribution of signature frequency for the reference table (R) using each technique. As we see from Figure 3, one observation is that in the LSH (k=4, m=6) scheme, most of the signatures it generates has very good *selectivity*, i.e., for a particular signature sig, the number of rows in the table for which sig is a signature is very small (below 10 at the 99th percentile), because a signature is effectively a random k-gram from the row. Unless there exists big clusters identical or almost identical rows in the reference table, it is unlikely that a signature will be associated with a large number of rows. Only a small number of "outliers" are associated with a large number of rows. In contrast, the selectivity for prefix filtering is significantly worse. For example, around 20% of all signatures are associated with more than 10 rows, and about 5% of all signatures are associated with more than 100 rows except for one dataset.

The adversarial case for both signature schemes occurs when there are a number of rows that consist of only frequent tokens, either from the original data or generated via applicable transformations. In such cases prefix filtering uses those frequent tokens as signatures, and those signatures are associated with many rows. LSH on the other hand generates hashes of random k tokens as signatures that usually are associated with far fewer rows. Only in rare cases does LSH generate signatures associated with many rows. Given these properties, we use LSH as the default signature scheme in our implementation.

3.4 Signature Pruning

Our signature pruning strategy is based on the following observations. First, the running time of the fuzzy join operator depends heavily on the selectivity of signatures. For LSH, as described above, the vast majority of signatures are associated with only a few rows in the reference table. Thus the join size is dominated by only a few signatures which are very frequent. Second, recall that our LSH scheme repeats the signature generation procedure $m(= 6)$ times per row. This creates redundancy in the signatures per row and provides resilience to dropping signatures. For example, if a pair of rows (s, r) from the two tables S and R has Jaccard similarity greater than 0.8, and we use $k = 4$ minhash tokens as one signature, then using 5 instead of 6 (default m) signatures results in 93% probability of (s, r) sharing at least one signature, and using 4 signatures results in 89% probability. In other words, dropping one signaure decreases the probability of them sharing at least one signature by only around 2% (from 95% to 93%) and dropping two signatures reduces the probability by 6% percent (95% to 89%). These two observations motivate us to use a pruning strategy of removing the signatures associated with more than a certain number rows in the reference table because they contribute the majority to the running time *and* removing them is expected to have minimal impact on recall. We evaluate the effectiveness of such pruning strategy in Section 5, and in practice we find that the impact on recall is negligible (far smaller than the theoretical analysis). One of the factors that further works in favor of such pruning is the use of IDF weighting for tokens. Frequent signatures typically correspond to tokens (or sequence of tokens) with low weight. Dropping such signatures therefore has a small impact on similarity.

It is worth noting that the above pruning technique is not effective for prefix filtering. The distribution of signature frequency is not very skewed and many signatures is associated with a large number of rows in the database; hence an attempt to gain significant speedup via pruning will result in a very large hit on recall. This intuition is confirmed in our experiments in Section 5.

4. SCALE-OUT FUZZY JOIN

In this section we outline how a scale-out implementation of fuzzy join can be implemented on a modern Big Data platform such as Spark [40]. Similar to traditional joins in parallel databases, we implement two variants of fuzzy join. For the special case when the reference table R is small and an inverted index over the signatures of R fits into the memory of a single node, we implement a *broadcast fuzzy join* dataflow pipeline. For the general case where the reference table is large, we implement a *shuffle fuzzy join* pipeline.

Since the actual join step works in the *token-ID space* for efficiency and not with the original strings in the data, both broadcast and shuffle fuzzy join pipelines share the same initial steps to convert the reference table and input table to the token-ID space; which we describe first (Section 4.1). In the pre-processing and broadcast fuzzy join pipelines, we leverage Spark's support for broadcast variables, which makes it easy to get important data structures such as the dictionary of tokens and applied transformations to each worker node.

4.1 Pre-processing

Figure 4 depicts the pipeline of the pre-processing job. At the end of the pre-processing process, we generate an intermediate representation of the reference table rows and input table rows, converted into the space of token-ID and token weights. For rows from the input table S, we also track the transformations applied. Below we discuss for the case of edit transformations. Similar logic can be applied for other kinds of transformations (e.g. synonyms), we omit those details for brevity. The pre-processing pipeline includes the following steps.

Step P1: Count the number of rows in the reference table (R). The count is stored in memory and used in Step $P2$. In principle, this step can be combined with Step $P2$ (compute token frequencies) by inserting an empty token per row and use the count of empty token as the number of rows. Since this step is not typically the bottleneck, such optimization does not significantly reduce the running time.

Step P2: Compute token frequencies in R. We tokenize the strings in each row of the right (reference) table into tokens and group them. We count token frequencies and sort them by token value then collect them as an array of ⟨token, freq⟩. Using the number of rows computed in Step $P1$, it computes the IDF weights for each token and produces an array of ⟨token, weight⟩. The sorting is required because the LSH signature generation scheme that we use requires stable token IDs across runs to be repeatable. We use the index in the array as the token-ID and therefore each run of fuzzy join on the same input data assigns the same token-ID for the same token. We also assume the number of *distinct* tokens in R is relatively small and all of them can be held in memory and broadcast to all the worker nodes. This assumption is borne out in practice: even on large real-world datasets the number of distinct tokens typically does not exceed $1 - 2M$.

Step P3: Compute edit transformation rules. We tokenize the strings in the input table (S) and group them into a dataframe of distinct tokens. We then broadcast the array of ⟨token,weight⟩ array collected in Step $P2$ to all worker nodes and run a map step, where for each token we compute all the tokens that are within k-edits from it ($k = 2$ is effective in datasets we have evaluated). The algorithm we use to compute all the tokens within k-edits leverages a trie. We initialize the trie by inserting all the tokens from the array into it. Then for each token in S, we traverse the trie. If the prefix is already more than k-edits away then we can avoid visiting all the subtrees in the trie. Despite this optimization, such computation is expensive (around 1 millisecond per token for a set of millions of reference tokens); therefore we trade-off a scan and shuffling all the tokens for computing edit transformation rules only once per token. We then collect the transformation rules as an array of ⟨token, array of token-IDs of tokens within k-edits⟩. Once again, since the number of transformation rules is relatively small, this array can be held in memory and broadcast.

Step P4: Prepare reference table rows for the join. We convert each row in reference table into ⟨rid, (⟨token-ID, weight⟩)*⟩. Towards that end we broadcast the ⟨token-ID, weight⟩ array produced in Step $P2$ and build a hash table.

Step P5: Prepare input table rows for the join. We convert each row in input table into ⟨rid, (⟨token-ID, weight⟩)*, (⟨position, (token-ID and weight of transformed token)*⟩)*⟩. We broadcast the token weights array produced in Step $P2$ and the token to similar token-IDs and weights array in Step $P3$ and build hash tables. Then we run a map step that for each tokenized row in the input table looks up each token's ID and weight, and if it matches the edit distance transformation rule, construct a struct of position as well as an array of token-IDs and weights corresponding to the reference table token after transformation. We use position here because the same token can occur more than once in the string and each replacement with a transformation needs to be tracked independently.

4.2 Broadcast Fuzzy Join

Figure 5 depicts the broadcast join pipeline. It takes the "prepared" reference and input table rows produced in Steps $P4$ and $P5$ of the pre-processing pipeline as input. The major steps in the pipeline are:

Step B1: Generate signatures for the reference table. We run a map step, which for each row in the dataframe produced in Step $P4$ in the pre-processing process, call LSH signature generator (section 3.2) to generate signatures and flatten it as a dataframe of ⟨rid, signature⟩. This is the *signature index* over the reference table.

Step B2: Prune signatures. Compute the pruned signature set from the signature index produced in Step $B1$ (see Section 3.4). We group the signature index by signature and filter out those having count higher than a specified *cutoff*. We observe that pruning results in a speedup even for broadcast fuzzy join due to fewer lookups against the signature index (although the gain is much more dramatic for shuffle fuzzy join).

Step B3: Generate and verify candidate pairs. We collect the dataframes produced in Step $B1$ and $B2$ and broadcast them to all worker nodes. Then we run a map step on the dataframe produced in Step $P5$ of the pre-processing process: we invoke signature generator to get signatures, ensure that the signature is not pruned, look up the index to find candidate rows, and finally verify the similarity is above the given threshold.

4.3 Shuffle Fuzzy Join

Figure 6 depicts the shuffle join pipeline. Like the broadcast version, it also takes as input the "prepared" reference and input table rows produced in Step $P4$ and $P5$ of the pre-processing pipeline, then proceeds as follows:

Step $S1$: Generate signatures for the reference table. We run a map step, which for each row in the dataframe produced in Step $P4$ in the pre-processing step, calls the LSH signature generator (Section 3.2) to generate signatures and flatten it as a dataframe of ⟨rid, signature⟩. This is the *signature index* over the reference table.

Step $S2$: Prune signatures. We group the signature index by signature, and extract those with count less than the *cutoff* threshold. They are the remaining signatures after pruning.

Step $S3$: Generate signatures for the input table. We run a map step, for each row in the dataframe produced in Step $P5$ of pre-processing, invoking the LSH signature generator and flattening it as a dataframe of ⟨rid, signature⟩.

Step $S4$: Generate candidate pairs. This step equi-joins the pruned signature index on R produced in Step $S2$ with the left signature index in Step $S3$. We then run a distinct operator on the join output to eliminate duplicate (s, r) pairs. The output is a dataframe of $\langle rid_{left}, rid_{right} \rangle$.

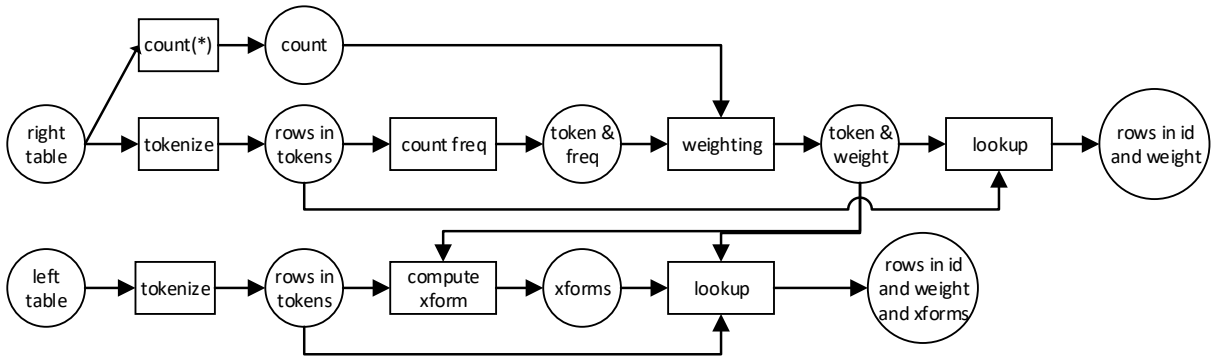

Figure 4: Pre-processing

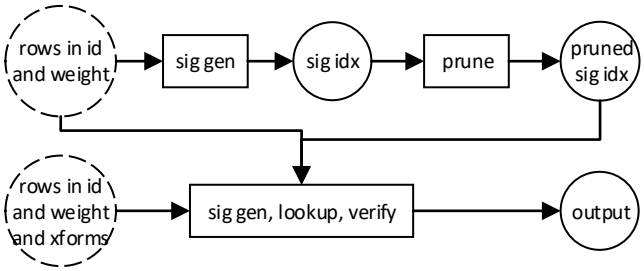

Figure 5: Broadcast Join

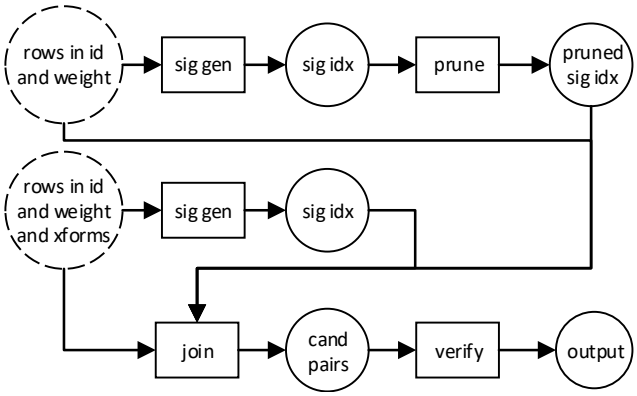

Figure 6: Shuffle Join

Finally, we join this dataframe with the right and left table and each row is a candidate pair to be verified.

Step $S5$: Verify candidate pairs. This map step verifies the candidate pairs by invoking the similarity function sim and outputs the pairs satisfying the given threshold θ.

4.4 Cost Analysis

All the steps in the preparation and broadcast pipeline are either map or group by. The overall cost of the broadcast join is proportional to the size of inputs $O(|R|)+O(|S|)$ where R is the reference table and S is the input table, plus the size of the output of fuzzy join; and it avoids shuffle entirely. In the case of shuffle fuzzy join, Step $S4$ is the most expensive and dominates the cost of pipeline because: (a) there are three joins and the size of join outputs can be much larger than the size of inputs. (b) shuffling large results of intermediate joins involves large amount of I/O. Depending on the selectivity of the signature scheme (quantified by average number of rows in the right table joined with a row in the left table and denoted as α thereafter), the cost can be modeled as $O(\alpha \times (|R| + |S|))$.

For example, in a setting where the size of the input table is much bigger than the size of the reference table, if α is 200, the join size is roughly 200 times the input size. This implies we would need to shuffle 10 TBs of data even for a modest 50 GB input table, which not only dominates the running time of the pipeline, but can cause some of the reducer nodes to run out of memory. In fact, we do see such out-of-memory behavior for prefix filtering signature scheme in some datasets, since its selectivity can be poor; whereas we have not observed this behavior with the LSH signature scheme where effective signature pruning can be applied.

5. EXPERIMENTS

We have implemented the broadcast and shuffle version of a customizable and scale-out fuzzy join operator (Section 4) in Scala for Spark. We implement both prefix filtering and locality sensitive hashing signature schemes (Section 3). The goals of our experiments are:

1. Compare the performance of prefix filtering and LSH signature schemes.
2. Study the effectiveness of signature pruning technique (Section 3.4) for prefix filtering and LSH.
3. Measure the scalability characteristics of scale-out fuzzy join with data size and number of nodes in the cluster.
4. Compare the performance of broadcast and shuffle based fuzzy join methods.
5. Compare the performance of scale-out fuzzy join on nodes (VMs) with varying CPU and memory resources.

5.1 Experimental Settings

We run our experiments on Azure Databricks [6] Spark clusters, runtime version 4.3 with Spark version 2.3.1 and

Scala version 2.11. Unless described otherwise, all experiments run on a standard cluster of 16 worker nodes. Each worker node is an Azure Standard L4s VM with 4 vCPUs, 32 GB main memory and 678 GB SSD local storage. Therefore the aggregate resources across the cluster are 64 vCPUs, 512 GB main memory and 10 TB SSD storage. The driver node is a separate standard L4s VM. We use the default configuration of Azure Databricks for Spark settings and JVM setting except changing *spark.sql.shuffle.partitions* to 512 for all experiments.

For any fuzzy join job that we run on Azure Databricks, we set the maximum running time at 10 hours. If a job does not finish within 10 hours, we report it as timeout. In some cases, a job fails since it gets *out of memory* on a node. We also indicate such error explicitly in the results below.

5.2 Datasets

We use six datasets to test our fuzzy join operator. Table 1 lists the number of rows, the number of distinct tokens, and the average number of tokens per row of each dataset. The first three datasets, REAL1, REAL2, REAL3 are proprietary datasets used by applications within Microsoft. They contain names, addresses and other contact information of organizations. USPS is a dataset of addresses in the United States [38], from which we extract distinct concatenation of street address, city, state and zip code. FEBRL is a synthetic dataset generated using an open source tool [11]. We extract person name, address, suburb, state, and postcode columns from it. IMDB contains movie data from the Internet Movie Data Base [22], in particular the Title, Directors and Genres columns.

The number of distinct tokens is usually much smaller than the number of rows in the dataset and does not exceed $1.2M$ across all datasets. Not surprisingly, all the real-world datasets exhibit a heavily skewed (Zipfian-like) distribution. For the scalability experiments, we generate datasets with scale factors: $1\times$, $2\times$, $3\times$, $4\times$ and $5\times$ the number of rows in the original dataset. We follow the methodology similar to that used in Vernica et al. [39] and Fier et al. [19], which preserves the original set of distinct tokens, their distribution and record lengths; but increases the number of records by replacing a token with a neighboring token in the sorted token frequency order.

For each dataset, we use the table as the reference table R. We generate the input table S as follows: for each row in R, we generate 10 rows by randomly applying some of following operations: inserting token, deleting token or replacing a token with some spelling error. Table 2 summarizes the number of rows of the fuzzy join operations at Scale $1\times$. For scale factor k, the size of both the reference and the input table are increased by a factor of k.

5.3 Performance of Shuffle Fuzzy Join

We run the shuffle pipeline with LSH and prefix filtering as signature generation schemes with and without using transformation rules. Note that we do *not* include the signature pruning optimization described in Section 3.4. In Figure 7 and Figure 8 respectively, we report the running time in minutes and the number of *signature pairs* generated in Step $S4$ (Section 4.3) of the pipeline. The number of signature pairs is a good metric for comparing signature schemes, and is often a good indicator of running time. We see that: (a) Fuzzy join is a very expensive operator, especially when used with transformation rules. Four cases run more than 10 hours (marked by "*"), three of which are with transformation rules. One case fails due to out-of-memory error (marked by "^"). (b) LSH generates significantly fewer signature pairs, and is typically much faster (particularly with transformations).

Table 1: Datasets used in experiments

Dataset	#Rows	#Distinct Tokens	Avg #Tokens per Row
REAL1	25.8M	1.2M	6.4
REAL2	7.7M	1.2M	10.3
REAL3	2.0M	1.2M	9.5
USPS	10.0M	0.3M	5.7
FEBRL	5.0M	0.05M	7.9
IMDB	1.1M	0.3M	6.9

Table 2: Number of rows of input and reference tables to fuzzy join (at scale factor $1\times$.)

Dataset	#Rows in Input Table	#Rows in Reference Table
REAL1	258M	25.8M
REAL2	77M	7.7M
REAL3	20M	2.0M
USPS	100M	10.0M
FEBRL	50M	5.0M
IMDB	11M	1.1M

5.4 Impact of Signature Pruning

As discussed in Section 3.4, we prune all signatures whose frequency (defined as number of rows in the reference table R associated with it) exceeds a particular cutoff value. We vary the cutoff value at 50, 100, 150, 200 for LSH and report the recall compared to the case where there is no pruning. The recall is shown in Figure 9. It can be seen that the recall for all the cutoff values are very close to 1, and the difference is small enough to be acceptable for most real-world scenarios. The running time and number of signature pairs are shown in Figure 10 and Figure 11 respectively. Note that the y-axis uses a log scale. Both running time and number of signature pairs are dramatically smaller compared to the case of no pruning, in some cases by more than an order of magnitude. In contrast, when we apply signature pruning to the prefix filtering scheme, the impact on recall (Figure 12) is drastic. Specifically, it may work when the dataset is small and the signature fanout is small (i.e. REAL3 and IMDB), but it will miss the majority of the result for big datasets. These experiments confirm the intuition that the randomized nature and the signature redundancy inherent in our LSH scheme can be leveraged to improve fuzzy join performance. Overall, we conclude that the signature pruning optimization is very effective for LSH, but inappropriate for prefix filtering.

Finally, in Figure 13 we measure the speedup that is obtained by LSH+Pruning (cutoff value = 50) relative to Prefix filtering (without pruning) for each dataset, since both these approaches have almost identical recall. For REAL1, the speedup is a lower bound (marked by "*") since the fuzzy

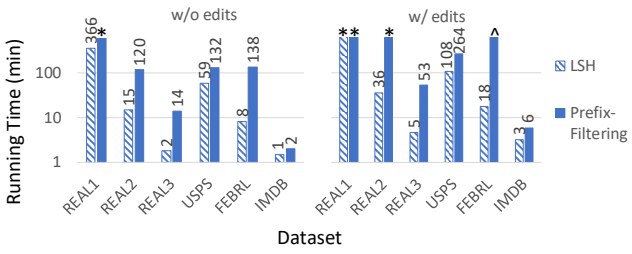

Figure 7: Running time: LSH vs Prefix-Filtering

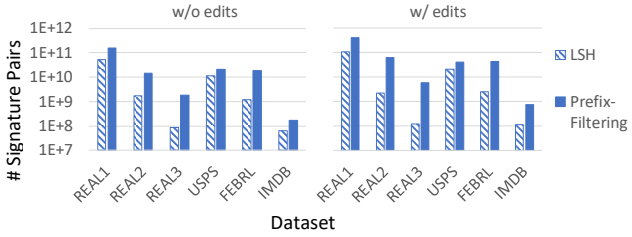

Figure 8: #Signature pairs: LSH vs Prefix-Filtering

join job using prefix filtering times out. We observe that the speedups range from 2.5× to over 50×, and the speedups are over 12× for the big datasets: REAL1, REAL2, USPS, and FEBRL.

In this Subsection 5.4, we only present experimental results without transformations. For cases with transformations, we have similar results for LSH+Pruning, but observe timeout for Prefix+Pruning. We omit the charts due to limited space.

5.5 Scalability of LSH+Pruning

We first evaluate the scalability of the LSH+Pruning approach with the number of nodes in the cluster. We create clusters of 4, 8, 12, 16, 32, and 64 L4s worker nodes, respectively. We run fuzzy join on each cluster and report its running time. The signature pruning cutoff value is set as 50. Figure 14 summarizes the results and we can see that LSH+Pruning scales almost linearly with number of nodes in the cluster.

Next, we evaluate scalability of LSH+Pruning in terms of size of input data. We use scaled datasets, generated as described in Section 5.2, that are 2×, 3×, 4× and 5× bigger reference and input tables. We run LSH+Pruning with cutoff = 50 on a 16-node cluster. From Figure 15 we see that the running time of the LSH+Pruning approach scales roughly linearly with input size as well. As discussed in Section 4.4, the shuffle pipeline running time is primarily determined by number of candidate pairs. In Figure 16, we report the number of signature pairs generated with and without pruning as the dataset size is scaled. The number of signature pairs grows super-linearly without pruning but grows almost linearly with pruning for all datasets.

We also evaluate LSH+Pruning in a scale-up setting. We run experiments on a E64s machine with 64 cores and 432 GB memory. The running time is 20% to 55% more compared to using the 16-node cluster due to the aggregate I/O bandwidth available on 16 nodes.

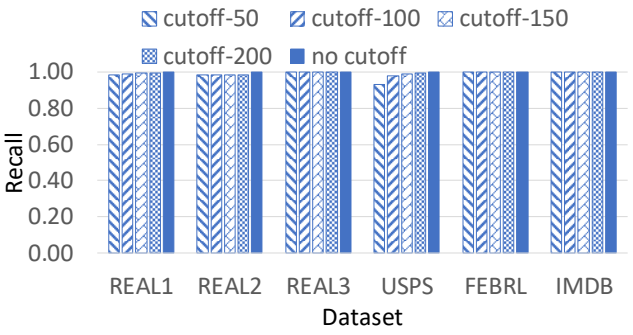

Figure 9: LSH+Pruning: Recall at different cutoffs

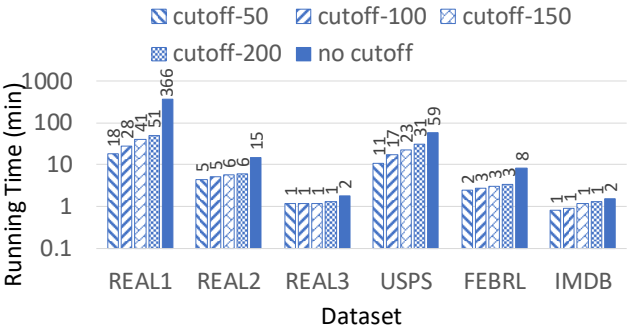

Figure 10: LSH+Pruning: Running time at different cutoffs

5.6 Broadcast vs. Shuffle

We compare broadcast and shuffle versions of fuzzy join. We use LSH+Pruning in both cases. Figure 17 shows that broadcast pipeline is faster than the shuffle pipeline for each of these datasets since our VMs have sufficient memory (32 GB) to hold the signature index over R in memory on each worker node.

Next, we run shuffle and broadcast fuzzy join pipelines on a 16-node cluster with Azure D8_v3 VM. D8_v3 VMs have 8 cores compared in L4s VMs which have 4 cores. Figure 18 and 19 shows the results respectively. Since the bulk of the work in broadcast fuzzy join is done in mappers on each worker node, it is able to exploit the increase in number of cores effectively. In contrast, while shuffle fuzzy join also seems some performance improvement, since its cost is dominated by the I/O cost associated with data shuffles, it does not benefit as much from increasing the number of cores per VM.

Finally, when we use datasets with a larger scale factor, and VMs with less memory, broadcast fuzzy join runs out of memory on the worker nodes. In these cases, shuffle fuzzy join still runs efficiently to completion. For example, for the dataset REAL1 (at scale factor 1×), when we use F4S VM nodes in Azure with 8 GB memory on each node, broadcast fuzzy join runs out of memory. However, shuffle fuzzy join runs to completion in 55 minutes.

Thus, in practice, when users can afford to rent VMs with large amounts of main memory, or the reference table is relatively small, broadcast fuzzy join is faster. However,

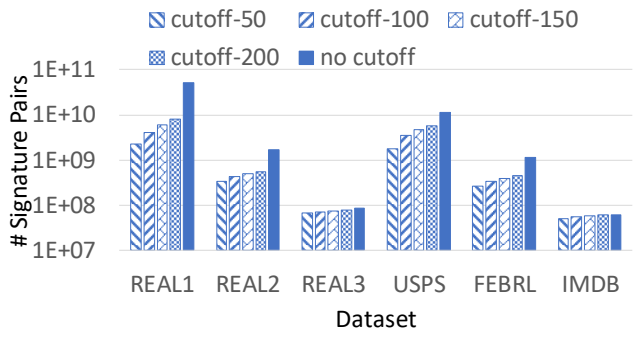

Figure 11: LSH+Pruning: #Signature pairs at different cutoffs

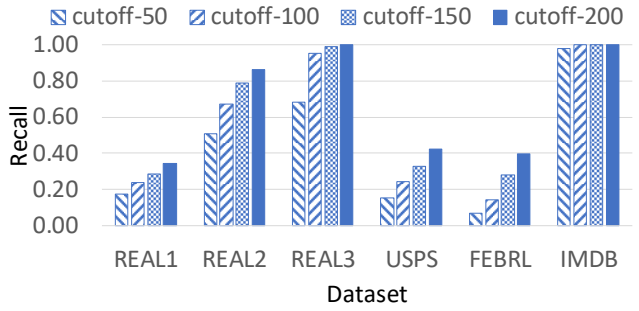

Figure 12: PF+Pruning recall at different cutoffs

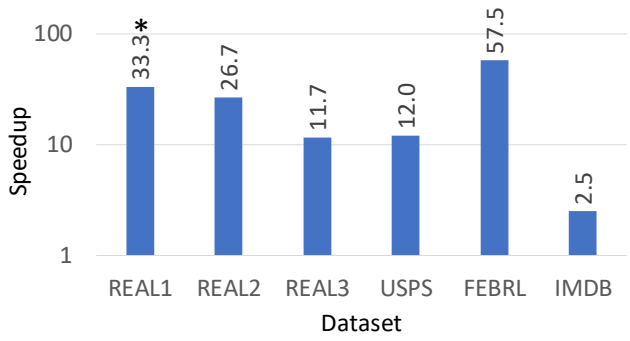

Figure 13: Speedup of LSH+Pruning relative to Prefix filtering

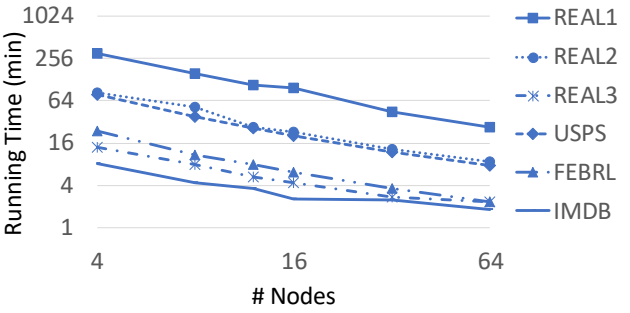

Figure 14: Running time vs #nodes

when datasets are large or per-node memory is constrained, shuffle fuzzy join is the only option.

6. RELATED WORK

There is a large body of work on fuzzy join, also referred to as *set-similarity join* (or *fuzzy matching*), that spans the last two decades both in research and industry. Several papers focused on signature based indexing schemes that would help identify (a smaller number of) candidate pairs on which the similarity function needed to be evaluated. Such schemes include, for example, [9, 21, 32, 5, 10]. These techniques varied in the similarity functions that they could support, e.g. edit distance and variants, cosine similarity, Jaccard similarity etc. While these techniques significantly improved performance of fuzzy join for certain similarity functions, the customization of the similarity function that they allowed was limited.

As noted in the introduction, the paper by Arasu et al. [4] introduced a transformation-based framework for record matching that allowed application or domain specific customizations such as synonyms and abbreviations, but also allowed traditional functions such as edit distance to be expressed in the same framework. This is also the approach we adopt. In this paper we show how via an implementation on Spark, such a framework can be made to perform well in a scale-out setting through optimized signature generation and pruning techniques.

As MapReduce [15] engines such as Hadoop started becoming used for ETL workloads, several new *scale out* fuzzy join techniques for MapReduce were proposed such as [18, 39, 8, 33, 27, 31, 13, 16, 17, 30]. Most of these techniques retained the overall approach used in single-node techniques: signature-based identification of candidate pairs followed by a verification step. For instance, Vernica et al. [39] show how a prefix filtering based technique can be efficiently implemented using Map and Reduce operations. More recently, an experimental comparison of several scale-out fuzzy join techniques was reported in [19]. Although most of the datasets they used to evaluate were relatively modest in size (the largest dataset contained around 10 million rows, and most datasets were below 1 million rows), they found that many of these techniques timeout after 30 minutes, or fail because they run out of memory. Among the techniques, they found VJ [39], which uses a prefix filtering based technique, to be a clear winner since it reported the lowest running time for a majority of the cases across several datasets. They also report that the two techniques that use an alternative approach, metric-based partitioning [13, 33], did not perform well for these datasets. As noted previously, these approaches do not enable the kinds of customization that we believe are important for many applications.

In the industry several fuzzy matching vertical solutions exist for important domains such as addresses (e.g. [37, 26]) and products (e.g. [14]). Many data preparation and ETL platforms (e.g. [23, 35, 24, 36]) also provide generic fuzzy matching capabilities. Similarly, in the open-source ecosystem, there are Spark packages for fuzzy matching, e.g. [34]. However they lack the flexible customization provided in our approach, and many of them do not offer a scale-out option.

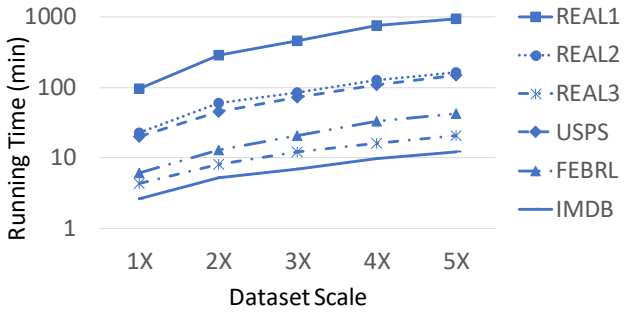

Figure 15: Running time vs. input size

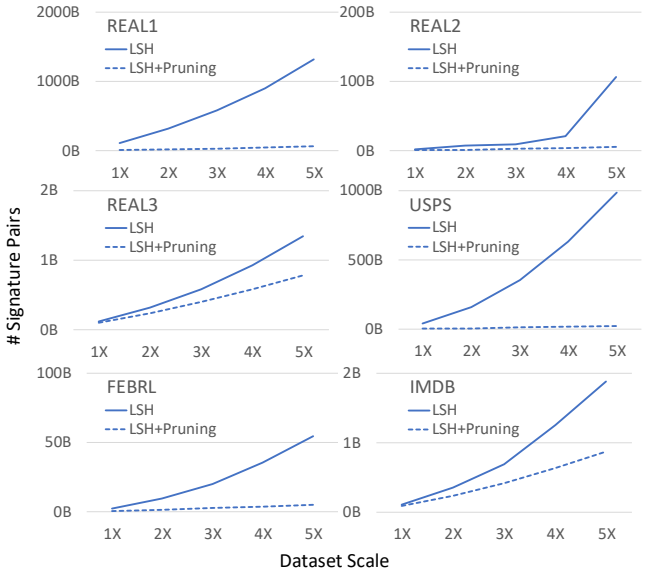

Figure 16: #Signature pairs vs input size

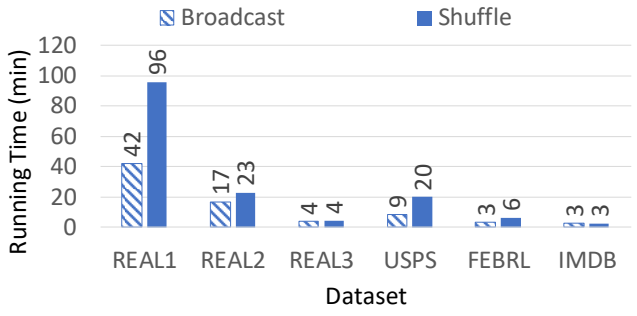

Figure 17: Broadcast vs Shuffle using LSH+Pruning

7. CONCLUSION

We have developed a scale-out fuzzy join operator for Spark. This operator is highly customizable while also exhibiting good performance and scalability characteristics. We have evaluated our fuzzy join operator on Azure Databricks. There are several important avenues of future work. While the empirical results on datasets we have evaluated

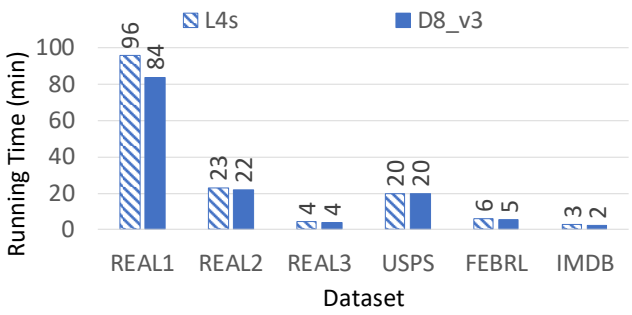

Figure 18: Running time of shuffle fuzzy join with different VM types.

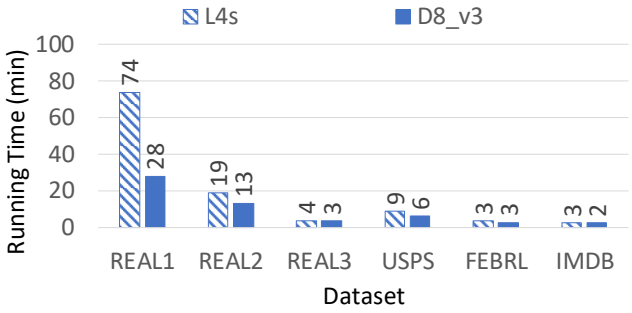

Figure 19: Running time of broadcast fuzzy join with different VM types.

thus far are very promising, we plan to expand datasets to more domains. Second, given that it may not be easy for users to identify whether to use broadcast or shuffle versions of fuzzy join for a given dataset, we plan to investigate technique to automatically recommend or choose the appropriate method. Finally, a closely related operator is fuzzy group-by, which is an important operation for de-duplication. Identifying scalable techniques for de-duplication on Big Data is another important area of future work.

8. ACKNOWLEDGMENTS

We thank Christian König and Yeye He for their insightful and detailed comments on the paper.

9. REFERENCES

[1] Azure Data Factory version 2(v2). `https://docs.microsoft.com/en-us/rest/api/datafactory/v2`.

[2] Azure ML Data Prep SDK. `https://github.com/Microsoft/AMLDataPrepDocs`.

[3] A. Arasu, S. Chaudhuri, Z. Chen, K. Ganjam, R. Kaushik, and V. Narasayya. Experiences with using data cleaning technology for bing services. *IEEE Data Eng. Bull.*, 35(2):14–23, 2012.

[4] A. Arasu, S. Chaudhuri, and R. Kaushik. Transformation-based framework for record matching. In *Proc. ICDE*, pages 40–49, 2008.

[5] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proc. VLDB*, pages 918–929, 2006.

[6] Azure Databricks: Fast, easy, and collaborative Apache Spark based analytics service. `https://azure.microsoft.com/en-us/services/databricks/`.

[7] R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval*. Pearson Addison Wesley, 2011.

[8] R. Baraglia, G. D. F. Morales, and C. Lucchese. Document similarity self-join with mapreduce. In *Proc. ICDM*, pages 731–736, 2010.

[9] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. SIGMOD*, pages 313–324, 2003.

[10] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *Proc. ICDE*, pages 5–5, 2006.

[11] P. Christen. Febrl: an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proc. SIGKDD*, pages 1065–1068, 2008.

[12] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC*, volume 810, pages 812–815, 2008.

[13] A. Das Sarma, Y. He, and S. Chaudhuri. Clusterjoin: a similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[14] Data Ladder Product Matching. `https://www.dataladder.com`.

[15] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[16] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Proc. ICDE*, pages 340–351, 2014.

[17] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *PVLDB*, 9(4):360–371, 2015.

[18] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proc. ACL*, pages 265–268, 2008.

[19] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag. Set similarity joins on mapreduce: an experimental survey. *PVLDB*, 11(10):1110–1122, 2018.

[20] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *PVLDB*, pages 518–529, 1999.

[21] L. Gravano, H. Jagadish, P. G. Ipeirotis, D. Srivastava, N. Koudas, and S. Muthukrishnan. Approximate string joins in a database (almost) for free. In *PVLDB*, pages 491–500, 2001.

[22] Internet Movie Data Base. `http://www.imdb.com`.

[23] Informatica Data Quality. `http://help.informatica.com`.

[24] Knime. `https://www.knime.com/nodeguide/other-analytics-types/text-processing/fuzzy-string-matching`.

[25] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *Proc. SIGMOD*, pages 802–803, 2006.

[26] Melissa Data Matching. `https://www.melissa.com/data-deduplication`.

[27] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *PVLDB*, 5(8):704–715, 2012.

[28] MurmurHash. `https://en.wikipedia.org/wiki/MurmurHash`.

[29] Microsoft Power Query. `https://docs.microsoft.com/en-us/power-query/`.

[30] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, and X. Du. Fast and scalable distributed set similarity joins for big data analytics. In *Proc. ICDE*, pages 1059–1070, 2017.

[31] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. Tung. Efficient and scalable processing of string similarity join. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[32] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proc. SIGMOD*, pages 743–754, 2004.

[33] Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *Proc. SIGMOD*, pages 693–696, 2012.

[34] Spark Package for Fuzzy Matching. `https://spark-packages.org/package/itspawanbhardwaj/spark-fuzzy-matching`.

[35] Fuzzy Lookup in SQL Server Integration Services. `https://docs.microsoft.com/en-us/sql/integration-services/data-flow/transformations/fuzzy-lookup-transformation`.

[36] Talend Fuzzy Matching. `https://help.talend.com`.

[37] Trillium Global Locator. `https://www.syncsort.com/en/Products/DataQuality/Trillium-Global-Locator`.

[38] USPS Database. `https://postalpro.usps.com/`.

[39] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proc. SIGMOD*, pages 495–506, 2010.

[40] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. HotCloud*, pages 10–10, 2010.