Dima: A Distributed In-Memory Similarity-Based Query Processing System

Ji Sun† Zeyuan Shang† Guoliang Li† Dong Deng‡ Zhifeng Bao*
†Department of Computer Science, Tsinghua University, Beijing, China
‡Department of Computer Science, MIT *Computer Science & IT, RMIT
{sunj16, shangzy13}@mails.thu.edu.cn; liguoliang@tsinghua.edu.cn;
dengdong@csail.mit.edu;zhifeng.bao@rmit.edu.au

ABSTRACT

Data analysts in industries spend more than 80% of time on data cleaning and integration in the whole process of data analytics due to data errors and inconsistencies. It calls for effective query processing techniques to tolerate the errors and inconsistencies. In this paper, we develop a distributed in-memory similarity-based query processing system called `Dima`. `Dima` supports two core similarity-based query operations, i.e., similarity search and similarity join. `Dima` extends the SQL programming interface for users to easily invoke these two operations in their data analysis jobs. To avoid expensive data transformation in a distributed environment, we design selectable signatures where two records approximately match if they share common signatures. More importantly, we can adaptively select the signatures to balance the workload. `Dima` builds signature-based global indexes and local indexes to support efficient similarity search and join. Since Spark is one of the widely adopted distributed in-memory computing systems, we have seamlessly integrated `Dima` into Spark and developed effective query optimization techniques in Spark. To the best of our knowledge, this is the first full-fledged distributed in-memory system that can support similarity-based query processing. We demonstrate our system in several scenarios, including entity matching, web table integration and query recommendation.

1. INTRODUCTION

In big data era, data are full of errors and inconsistencies and create many troubles in data analysis. As reported in a New York Times article, 80% of a typical data science project is cleaning and preparing the data, while the remaining 20% is actual data analysis[1]. Therefore, it is demanding to have efficient and effective query processing techniques to serve the data cleaning job, and some similarity-based algorithms [5, 2, 11, 1, 9, 7, 4, 6, 8, 10, 3] have been proposed. However, they suffer from several limitations. Firstly, they are not full-fledged and cannot support complex data analysis, e.g. SQL-based analysis. Secondly, the algorithms either cannot support large-scale data analysis or have workload unbalance problem.

To address these limitations, we develop a distributed in-memory system `Dima` that can utilize SQL to support similarity-based query processing. In this paper we focus on two core operations, i.e., similarity search and similarity join. Similarity search extends traditional exact search by tolerating errors and similarity join extends traditional exact join by tolerating errors between records. Regarding the similarity metrics adopted in search and join, we focus on two widely adopted ones: set-based similarity and character-based similarity [5]. A big challenge in distributed computing is to avoid expensive data transmission. An effective way is to judiciously assign data into different partitions such that the results must be in the same partition (and avoid the Cartesian product over different partitions). To achieve this goal, we propose effective signatures where two records approximately match if they share common signatures. Using signatures, we build global indexes and local indexes to support similarity search and join. Another challenge is to balance the workload among partitions. To this end, we propose the concept of selectable signatures, which are adaptively selectable based on the workload. Based on selectable signatures and effective indexes, we devise efficient algorithms to support similarity-based query processing. We seamlessly integrate `Dima` into Spark SQL by developing effective query optimization techniques on top of Spark SQL.

We demonstrate our system in three scenarios and all VLDB attendees working on big data integration and analysis will be interested in our demo.

Entity Matching. Given two sets of objects, our system can be used to identify the pairs of objects that refer to the same entity. For example, we want to find the products from Amazon and eBay that refer to the same entity. We also want to find the publications from DBLP and Google scholar that refer to the same entity.

Web Table Integration. Web tables contain a large amount of high-quality relational information and a recent Google study extracted 14 billion raw HTML tables and estimated that 150 million among these are relational tables. These Web tables, if properly integrated, can benefit numerous pragmatic applications such as searching structured data, data discovery and data transformation. Our system can be used to integrate web tables.

Query Recommendation. Users may not type search queries correctly, and thus search engines, e.g., Google, rec-

[1]http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html

Proceedings of the VLDB Endowment, Vol. 10, No. 12

ommend relevant queries based on query logs. Our system can be used to recommend query logs similar to the query.

Our system is available at `http://dbgroup.cs.tsinghua.edu.cn/ligl/dima/` and our source code is publicized at `https://github.com/TsinghuaDatabaseGroup/dima`.

2. SIMILARITY-BASED QUERY PROCESSING FRAMEWORK

2.1 Similarity-Based Query Operations

Given two records r and s, we use a similarity function to compute their similarity and we focus on two types of similarities.

Set-Based Similarity. It tokenizes records as sets of tokens and computes the similarity based on the sets, e.g., Jaccard, Cosine, DICE. Two records are similar w.r.t. Jaccard if their Jaccard similarity is not smaller than a threshold τ. For example, the Jaccard similarity between {`SIGMOD, 2017, NC`} and {`SIGMOD, 2016, CA`} is 1/5.

Character-Based Similarity. It transforms a record to another based on character transformations and computes the similarity by the number of character transformations. The well-known character-based similarity is edit distance, which transforms a record to another by three atomic operations, deletion, insertion and substitution, and takes the minimum number of edit operations as the edit distance. Two records are similar w.r.t. edit distance if their edit distance is not larger than a threshold τ. For example, the edit distance between `SIGMOD` and `SIGMD` is 1.

Next we formally define two similarity operations based on the similarity functions.

DEFINITION 1 (SIMILARITY SEARCH). *Given a collection of records $\mathcal{R}$, a query s, a similarity function f and a threshold τ, the similarity search problem aims to find all similar records from the set, i.e., $\{r \in \mathcal{R} | f(r, s, \tau) = true\}$. For Jaccard, $f(r, s, \tau) = true$ iff. $\text{JAC}(r, s) \geq \tau$; For edit distance, $f(r, s, \tau) = true$ iff. $\text{ED}(r, s) \leq \tau$.*

DEFINITION 2 (SIMILARITY JOIN). *Given two collections of records $\mathcal{R}$ and $\mathcal{S}$, the similarity join problem aims to find all similar record pairs from the two sets, i.e., $\{(r, s) | r \in \mathcal{R}$ & $s \in \mathcal{S}$ & $f(r, s, \tau) = true\}$.*

Our goal is to to support these two operations in distributed in-memory systems.

2.2 Our Framework

Extended SQL. We extend SQL and define `simSQL` by adding two operations to support similarity search and join. We use $\texttt{Sim}(f, \tau)$ to support similarity-based query processing using function f and threshold τ and the two operations are defined as below.

(1) Similarity Search. Users utilize the following `simSQL` query to find records in table T whose $\mathcal{S}$ column is similar to query q w.r.t. similarity function f and threshold τ.

`SELECT` * from `FROM` T `WHERE` q $\texttt{Sim}(f, \tau)$ T.$\mathcal{S}$

(2) Similarity Join. Users utilize the following `simSQL` query to find the records in tables T_1 and T_2 where table T_1's $\mathcal{S}$ column is similar to table T_2's $\mathcal{R}$ column w.r.t. similarity function f and threshold τ.

`SELECT` * from `FROM` T_1, T_2 `WHERE` $T_1.\mathcal{S}$ $\texttt{Sim}(f, \tau)$ $T_2.\mathcal{R}$

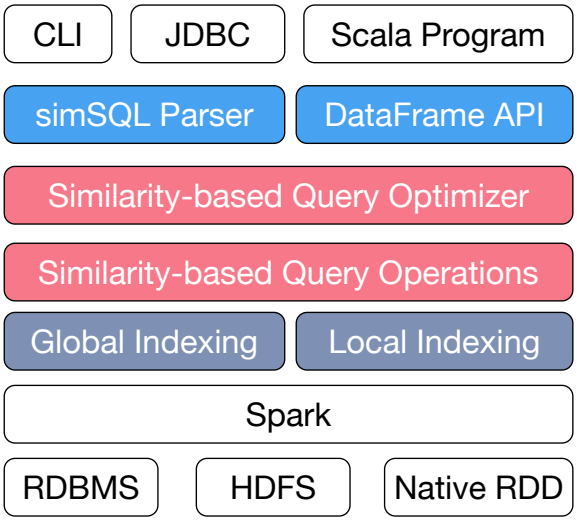

Figure 1: The Framework of `Dima`.

DataFrame. In addition to `simSQL`, users can also perform special operations over DataFrame objects using a domain-specific language similar to data frames in R. We also extend Spark's DataFrame API to support similarity search and join similar to the aforementioned extended `simSQL`.

Index. Users can also create index to support similarity search and join. Our system supports both gloabl indexes and local indexes. Users can utilize the following `simSQL` query to create a local index on the column $\mathcal{S}$ of table T using the `SEGINDEX`, which will be introduced in Section 3.1.

`CREATE LOCAL Index` GlobalIndex `ON` T.$\mathcal{S}$ `USE SEGINDEX`.

Users can also build a global index using the following `simSQL`.

`CREATE GLOBAL Index` LocalIndex `ON` T.$\mathcal{S}$ `USE SEGINDEX`.

Similarity-based Query Processing. We utilize the above signature-based index to process similarity search and join query. For search, we utilize the global index to prune irrelevant partitions and send the query request to relevant partitions. In each local partition, we utilize the local index to compute the local results. For join query, we utilize the global index to make the similar pairs in the same partition and this can avoid expensive data transmission. In each partition, we utilize local index to compute local answers. The details are discussed in Sections 3.2.

Query Optimization. `Dima` extends the Catalyst optimizer of Spark SQL and introduces a cost-based optimization (CBO) module to optimize the approximation queries. The CBO module leverages the (global and local) index to optimize complex `simSQL` queries. Query optimization in `Dima` is discussed in Section 3.3.

`Dima` Workflow. Figure 1 shows the architecture of our `Dima` framework. Next we give the query processing workflow of `Dima`. Given a `simSQL` query or a DataFrame object, `Dima` first constructs a tree model by the `simSQL` parser or a DataFrame object by the DataFrame API. Then `Dima` builds a logical plan using Catalyst rules. Next, the logical optimizer applies standard rule-based optimization to optimize the logical plan. Based on the logical plan, `Dima` applies cost-based optimizations based on signature-based indexes and statistics to generate the most efficient physical plan. `Dima` supports analytical jobs on various data sources such as CVS, JSON and Parquet.

3. DIMA SYSTEM

3.1 Indexing

3.1.1 Selectable Signatures

Basic Idea. Given a data record r and a query record s, we generate two types of indexing signature set for r, $\texttt{iSig}^+$ and $\texttt{iSig}^-$, and two types of probing signature set for s, $\texttt{pSig}^+$ and $\texttt{pSig}^-$. If $\texttt{iSig}^+ \cap \texttt{pSig}^+ = \phi$, we can deduce that r and s have at least 1 mismatched token (or character). If $\texttt{iSig}^+ \cap \texttt{pSig}^+ = \phi$ & $\texttt{iSig}^+ \cap \texttt{pSig}^- = \phi$ & $\texttt{iSig}^- \cap \texttt{pSig}^+ = \phi$, we can deduce that r and s have at least 2 mismatched tokens (characters). We can utilize either of the two properties to do pruning and thus we can select a better way to reduce the cost. Moreover in distributed computing, we can select a better way to balance the workload. Next we take Jaccard as an example to define the signatures and interested readers can refer to [4, 6] for more details for edit distanece.

Indexing Signatures. Given a record r, we generate two types of indexing signatures: indexing segment signatures and indexing deletion signatures.

Indexing Segment Signatures. We partition record r into $\eta_{|r|}$ disjoint segments $seg_1, seg_2, \cdots, seg_{\eta_{|r|}}$. $\texttt{iSig}^+_{r,i,|r|} = (seg_i, i, |r|)$ is an indexing segment signature for $1 \le i \le \eta_{|r|}$. We will introduce how to compute the number of segments $\eta_{|r|}$ later. Let $\texttt{iSig}^+_r = \cup_{i=1}^{\eta_{|r|}} \{\texttt{iSig}^+_{r,i,|r|}\}$ denote the indexing segment signature set of r.

Indexing Deletion Signatures. For each segment signature $\texttt{iSig}^+_{r,i,|r|} = (seg_i, i, |r|)$, we generate an indexing deletion signature $\texttt{iSig}^-_{r,i,|r|,k} = (del_i^k, i, |r|)$ where del_i^k is a subset of seg_i by deleting the k-th token ($1 \le k \le |seg_i|$). Let $\texttt{iSig}^-_{r,i,|r|} = \cup_{k=1}^{|seg_i|} \{\texttt{iSig}^-_{r,i,|r|,k}\}$ denote the set of deletion signatures for the i-th segment. Let $\texttt{iSig}^-_r = \cup_{i=1}^{\eta_{|r|}} \texttt{iSig}^-_{r,i,|r|}$ denote the indexing deletion signature set of r.

Number of Segments $\eta_{|r|}$. If $\textsc{Jac}(r,s) \ge \tau$, we have $1 - \frac{|r \cap s|}{|r \cup s|} \le 1 - \tau$. Thus $|r \cup s - r \cap s| \le |r \cup s|(1-\tau) \le (1-\tau)\frac{|r \cap s|}{\tau} \le \frac{1-\tau}{\tau}|r|$. That is for any set s, if s is similar to r, it has at most $\lfloor \frac{1-\tau}{\tau}|r| \rfloor$ mismatched tokens with r. Thus if we partition r into $\eta_{|r|} = \lfloor \frac{1-\tau}{\tau}|r| \rfloor + 1$ segments, s must share a segment with r if s is similar to r. Note in the partition, we must keep a global order for the tokens, and the same token in different records must be assigned into the same segment. Thus we can keep a hash function Γ that maps a token t to the i-th segment, i.e., $\Gamma(t) = i$.

Probing Signatures. Given a record s, if it is similar to record r, the length difference between s and r should not be too large. In other words, s can only be similar to a record r whose length $|r|$ ranges in $[l^-_{|s|}, l^+_{|s|}]$. Since $|r| \ge |r \cap s| \ge |r \cup s| \cdot \tau \ge |s| \cdot \tau$, we have $l^-_{|s|} = \lceil |s| \cdot \tau \rceil$. Similarly, since $|r| \le |r \cup s| \le \frac{|r \cap s|}{\tau} \le \frac{|s|}{\tau}$, we have $l^+_{|s|} = \lfloor \frac{|s|}{\tau} \rfloor$. As the records with different lengths have different segment strategies, we should consider every length $l \in [l^-_{|s|}, l^+_{|s|}]$.

Probing Segment Signatures for Length l. As the record with length l is partitioned into η_l segments, we also partition s into η_l segments $seg_1, seg_2, \cdots, seg_{\eta_l}$ (using the same global order, e.g. the same hash function Γ). For each $i \in [1, \eta_l]$, we generate a probing segment signature $\texttt{pSig}^+_{s,i,l} = (seg_i, i, l)$.

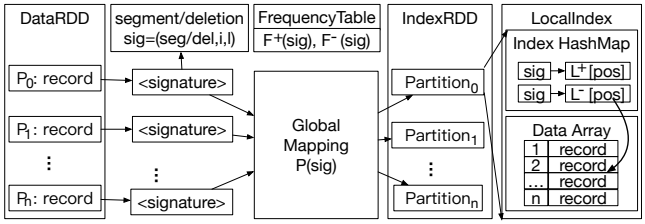

Figure 2: Indexing Structure.

Let $\texttt{pSig}^+_{s,l} = \cup_{i=1}^{\eta_{|r|}} \{\texttt{pSig}^+_{s,i,l}\}$ denote the probing segment signature set of s for length l.

Probing Deletion Signatures for Length l. For each probing segment signature $\texttt{pSig}^+_{s,i,l} = (seg_i, i, l)$, we generate a deletion signature $\texttt{pSig}^-_{s,i,l,k} = (del_i^k, i, l)$ where del_i^k is a subset of sig_i by deleting the k-th token. Then we can get a probing deletion signature set $\texttt{pSig}^-_{s,i,l} = \cup_{k=1}^{\eta_l} \{\texttt{pSig}^-_{s,i,l,k}\}$. Let $\texttt{pSig}^-_{s,l} = \cup_{i=1}^{\eta_l} \texttt{pSig}^-_{s,i,l}$ denote the probing deletion signature set of s for length l.

Pruning Condition. Consider r and s. If r and s, they cannot have two many mismatched tokens. Next we discuss how to compute the bound of mismatched tokens. $|r \cup s - r \cap s| = |r| + |s| - 2|r \cap s| \le |r| + |s| - 2\frac{\tau}{1+\tau}(|r| + |s|) = \frac{1-\tau}{1+\tau}(|r| + |s|)$. Thereby, if r and s are similar, they should have at most $\frac{1-\tau}{1+\tau}(|r|+|s|)$ mismatched tokens. Let $\theta_{|s|,|r|} = \lfloor \frac{1-\tau}{1+\tau}(|r| + |s|) \rfloor + 1$ denote the dissimilar token threshold, i.e., if r and s have more than $\theta_{|s|,|r|}$ mismatched tokens, they cannot be similar to each other. We call $\theta_{|s|,|r|}$ as the dissimilar threshold bound.

Signature Selection. We have two options in selecting the probing signatures.
(1) Selecting the probing segment signature $\texttt{pSig}^+_{s,i,l}$. If $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^+_{r,i,l} = \phi$, r and s have at least 1 mismatched token on the i-th segment.
(2) Selecting the probing deletion signature $\texttt{pSig}^-_{s,i,l}$. If $\texttt{pSig}^-_{s,i,l} \cap \texttt{iSig}^+_{r,i,l} = \phi$ & $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^-_{r,i,l} = \phi$ & $\texttt{pSig}^+_{s,i,l} \cap \texttt{iSig}^+_{r,i,l} = \phi$, r and s have at least 2 mismatched tokens on the i-th segment.

Suppose we select x probing segment signatures and y probing deletion signatures of s such that $x + 2y \ge \theta_{|s|,|r|}$. If there is no matching on the selected signatures, r and s have at least $\theta_{|s|,|r|}$ mismatched tokens, then r and s cannot be similar. Based on this property, we will present how to select probing signatures for similarity search and join.

3.1.2 Distributed Indexing

Given a dataset $\mathcal{R}$, we build a global index and a local index offline. Then given an online query s, we select the probing signatures of s, utilize the global index to locate the partitions that contain s's probing signatures, and send the probing signature to such partitions. The executor that monitors such partitions does a local search to compute the local results. Figure 2 shows the indexing structure.

Offline Indexing. Note that different queries may have different thresholds and we require to support queries with any choice of threshold. To achieve this goal, we utilize a threshold bound to generate the index. For example, the threshold bound for Jaccard is the smallest threshold for all queries that the system can support, e.g., 0.6. Using this threshold bound, we can select the indexing segment/deletion signatures and build a *local index*. In addition, we also keep the

frequency table of each signature to keep each signature's frequency and build a *global index* that keeps a mapping from the signature to partitions that contain this signature.

Frequency Table. For each RDD $\mathcal{R}_i$, for each record $r \in \mathcal{R}_i$, we compute its indexing segment number $\eta_{|r|}$ using the threshold bound. Then we generate the indexing segment signature set of each record r, $\texttt{iSig}_r^+$ and indexing deletion signature set $\texttt{iSig}_r^-$. For each segment signature $g \in \texttt{iSig}_r^+$, we collect its global frequency $\mathcal{F}^+[g]$ and for each deletion signature $g' \in \texttt{iSig}_r^-$, we also collect its global frequency $\mathcal{F}^-[g']$. If we keep the frequency of all signatures, the frequency table will be too large. Thus we only keep the signatures whose frequencies exceed 2. In this way the frequency table is very small, and the frequency table can be easily distributed into every node.

Local Index. Next we shuffle the indexing signatures such that (1) each signature and its inverted list of records that contain this signature are shuffled to one and only one partition, i.e., the same signature will be in the same partition and (2) the same partition may contain multiple signatures and their corresponding records. For each partition, we construct an IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ for indexing signatures in this partition. Each IndexRDD $\mathcal{I}_i^{\mathcal{R}}$ contains several signatures and the corresponding records, which includes two parts. The first part is a hash-map which keeps the mapping from a signature to two lists of records: $\mathcal{L}^+[g]$ keeps the records whose indexing segment signatures contain g and $\mathcal{L}^-[g]$ keeps the records whose indexing deletion signatures contain g. We use $\mathcal{L}[g]$ to denote $\mathcal{L}^+[g].\cup\mathcal{L}^-[g]$. The second part is all the records in this RDD, i.e., $\mathcal{D}_i = \cup_{g\in\mathcal{I}_i^{\mathcal{R}}}\mathcal{L}[g]$. Note that the records are stored in the data list $\mathcal{D}_i$ and $\mathcal{L}^+[g]$ and $\mathcal{L}^-[g]$ only keep a list of pointers to the data list $\mathcal{D}_i$.

Global Index. Then for each signature, we keep the mapping from the signature to the partitions that contain this signature. Note that we do not need to utilize a hash table to keep the mapping. Instead, we only maintain a global function $\mathcal{P}$ that maps a signature g to a partition p, i.e., $\mathcal{P}(g) = p$. Thus the global index is rather small.

3.2 Similarity-based Query Operations

3.2.1 Similarity Search

Given an online query s, `Dima` utilizes the proposed indexes to support similarity search operation in three steps. (1) It first conducts a global search by utilizing the frequency tables to select the probing signatures of s. Specifically, we propose an optimal signature selection method to achieve a balance-aware selection by using a dynamic-programing algorithm. (2) For each selected probing signature, it utilizes the global hash function to compute the partition that contains the signature and sends the search request to the corresponding partition. (3) Each partition further exploits a local search to retrieve the inverted lists of probing signatures and verify the records on the inverted lists to get local answers. Finally, it returns the local answers.

3.2.2 Similarity Join

To join two sets $\mathcal{R}$ and $\mathcal{S}$, a straightforward approach is to first build the index for a set, e.g., $\mathcal{R}$, then take each record $s \in \mathcal{S}$ as a query and invoke the search algorithm to compute its results. However, it is rather expensive for the driver, because it is costly to select signatures for huge number of queries. To address this issue, we propose an algorithm for similarity join consisting of four steps. (1) It generates signatures and builds the indexRDD for one dataset. (2) Then it selects probing signatures for each length l. Since it is expensive to utilize the dynamic-programming algorithm to compute the optimal signatures. Thus we propose a greedy algorithm to efficiently compute the high-quality signatures. (3) For each selected signature it builds the probeRDD for the other dataset. Since the matched probing and indexing signatures are in the same executor, it can avoid data transmission between different partitions. (4) It computes the local join results in each executor based on the indexRDD and probeRDD, and the master collects the results from different local executors.

3.3 Cost-Based Query Optimization

A SQL query may contain multiple operations, it is important to estimate the cost of each operation and thus the query engine can utilize the cost to select a query plan, e.g., join order. Since Spark SQL has the cost model for exact selection and join, we focus on estimating the cost for similarity join and search. If there are multiple join predicates, we also need to estimate the result size. In our system, we adopt cost-based model to optimize a SQL query.

4. DEMONSTRATION SCENARIO

Similarity Join For Entity Matching. First, we will demonstrate how the *SimJoin* operator of Dima helps solve the entity matching problem. We use `Dbpedia`[2] dataset which is a knowledge base of entities.

Similarity Search For QueryLog Recommendation. Next, we will demonstrate how Dima's *Select* operator can help provide instant recommendations to user's search queries that have spelling errors, to support interactive search.

Acknowledgment. This work was supported by the 973 Program of China (2015CB358700), NSF of China (61632016, 61373024,61602488,61422205,61472198), ARC DP170102726, and FDCT/007/2016/AFJ.

5. REFERENCES

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

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

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

[4] 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.

[5] Y. Jiang, G. Li, J. Feng, and W. Li. String similarity joins: An experimental evaluation. *PVLDB*, 7(8):625–636, 2014.

[6] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *PVLDB*, 5(3):253–264, 2011.

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

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

[9] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[10] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[11] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.

[2] http://wiki.dbpedia.org/Downloads2015-04