# Optimization for Large-Scale Fuzzy Joins Using Fuzzy Filters in MapReduce

Thi-To-Quyen TRAN
*Univ Rennes, CNRS, IRISA*
Lannion, France
thi-to-quyen.tran@irisa.fr

Thuong-Cang PHAN
*Can Tho University*
Can Tho, Vietnam
ptcang@cit.ctu.edu.vn

Anne LAURENT
*Univ Montpellier, LIRMM, CNRS*
Monpellier, France
Anne.Laurent@lirmm.fr

Laurent D'ORAZIO
*Univ Rennes, CNRS, IRISA*
Lannion, France
laurent.dorazio@univ-rennes1.fr

***Abstract*—A fuzzy or similarity join is one of the most useful data processing and analysis operations for Big Data in a general context. It combines pairs of tuples for which the distance is lower than or equal to a given threshold $\varepsilon$. The fuzzy join is used in many practical applications, but it is extremely costly in time and space, and may even not be executed on large-scale datasets. Although there have been some studies to improve its performance by applying filters, a solution of an effective fuzzy filter for the join has never been conducted. In this paper, we thus extend our previous work by proposing a novel fuzzy filter to optimize fuzzy joins. This filter is a compact, probabilistic data structure that supports very fast similarity queries by maintaining a bit matrix, with small false positive rate and zero false negative rate. We show that our proposal is more efficient than others because of eliminating redundant data, reducing computation cost and avoiding duplicate output.**

***Index Terms*—Fuzzy join, Similarity join, MapReduce, Fuzzy Filter**

## I. Introduction

Join is a critical operation within a data management system, making it possible to enrich data from a source with external information. That is why literature on join optimization is rich, especially in parallel and distributed systems [1]–[9]. In recent years, researches have focused on the problem of efficient joins in large-scale parallel environments. First results on equi-join [10], impose strong constraints on data (one of the sets having to be small enough to be distributed to all the machines used for the treatment) or their organization (sorting according to the join attribute, placement of data on specific nodes), leading to many data transfers (some unnecessary) and heavy workload on machines or requiring multiple (expensive) execution phases. The problem is even more difficult when the equality constraint is released while this type of query is often necessary. It is motivated by applications requiring similar matching. As a query example [11] in mining social networking sites where user's preferences are stored as bit vectors (where a "1" bit means an interest in a certain domain), applications want to discover the similar interests of users. A user with preference bit vector "[1,0,0,1,1,0,1,0,0,1]" possibility has similar interests to a user with preferences "[1,0,0,0,1,0,1,0,0,1]". Another example, widely used in image content-based search engines as Google, Baidu, Bing, discover images whose features maps into binary code by their similarities are greater than a predefined threshold. This query is defined as a fuzzy or similarity join and plays an important role in various applications, including data cleaning [12], data integration [13], detecting attacks from colluding attackers [14], mining in social networking sites [15], detecting near-duplicate web-pages in web crawling [16], plagiarism detection [17], document clustering [18], master data management [19], bioinformatics [20].

**Stage-of-the-art.** Fuzzy join execution consists of a Cartesian product and processing pair distances. Therefore, this operation is expensive in traditional databases [21], [22]. Moreover, when dealing with large amounts of data, fuzzy join becomes a challenging problem in a distributed and parallel environment with expensive costs due to data shuffling, in addition to space and efficiency constraints. As a result, data redundancy and duplication are very difficult to accept. MapReduce [23] is a famous framework proposed by Google to process large-scale data in parallel. It is widely applied to modeling, processing and calculating costs in large-scale fuzzy join studies [24], [25]. Most of the existing solutions follow a filtering-verification framework in multiple stages to generate candidates, apply principles (prefix filter, length filter, segment techniques) to prune out hopeless pairs, based on similarity functions to determine all pairs within a radius. Vernica et al. [11] proposed a similarity join method using a 3-stage MapReduce approach which utilized the prefix filtering method by inverted index on tokens to support set-based similarity functions. Metwally et al. [26] proposed a 2-stage algorithm VSMART join for similarity join on set, multisets and vector. Deng et al. [27] use signatures to calculate inverted index and process in 3-stage for set similarity join. [28] improved [27] by replacing signatures with q-gram technique in 3-stage. Afrati et al. [29] proposed multiple algorithms to perform a fuzzy join with Hamming, Edit and Jaccard distance in a single MapReduce stage without filters. Other algorithms [30]–[32] use pivots to split data into disjoint partitions by recursive jobs.

**Motivation.** One challenge in distributed computing is to avoid expensive data transmission and large disk I/Os. While recent studies on the fuzzy join have common limitations such as input re-reading by multiple phases, wasteful redundancy and duplication of data, we have introduced filter-based approaches [8], [10], [33]. Our team was interested in using Bloom Filters [34], Intersection Filter [8]. The idea is to filter

978-1-7281-6932-3/20/$31.00 ©2020 IEEE

irrelevant data as soon as possible to reduce data transfers and workload on different machines. Besides, the computation of the Hamming distance is shown faster than the computation of the distance in the input space. Therefore, we take advantage of its theory to propose a new filter for fuzzy joins.

**Contributions.** This study focuses on the improvement for large-scale fuzzy join. The main contributions of our works are

- a novel fuzzy filter (FF) structure for detecting if an element is close to any members in a set. Moreover, FF can determine which records in the set are real similarities of this element.
- large-scale FF based fuzzy join algorithm to avoid useless re-computation.
- a theoretical analysis of various similarity join algorithms in MapReduce and their cost comparison in a map-reduce-shuffle computation.

The remaining part of this paper is organized as follows. Section 2 summarizes the research background of the MapReduce framework, the fuzzy join definition, the previous filter studies, and position of our paper with respect to related work. In section 3, our approaches for modeling the Fuzzy filter (FF) and optimizations for large-scale FF based fuzzy join are presented. Finally, section 4 concludes and discusses future work.

## II. Related work

### A. MapReduce

MapReduce [23] is a parallel and distributed programming model to process large amounts of data on data centers consisting of commodity hardware. This model allows users to focus on designing their applications regardless of the distributed aspects of the execution. Figure 1 illustrates the MapReduce execution.

A MapReduce program consists of two distinct phases, namely, the Map phase and the Reduce phase. Each phase performs a user function on a key/value pair. The function Map **(M)** takes a pair of entries $(k1, v1)$ and emits a list of intermediate pairs $(k2, v2)$.

$$(k1, v1) \xrightarrow{map} (k2, v2)$$

The intermediate values associated with the same key $k2$ are grouped together and then transmitted to the Reduce function which aggregates the values.

$$(k2, v2) \xrightarrow{reduce} (k3, v3)$$

A MapReduce program is executed on multiple nodes. During the Map phase, each Map task reads a subset (called split) of an input dataset and applies the Map function for each key/value pair. The system supports grouping of intermediate data and sends them to the relevant nodes to apply the Reduce phase. This communication process is called Shuffle. Each Reduce task collects the key/value pairs of all the Map tasks, sorts/merges the data with the same key and calls the Reduce function to generate the final results.

### B. Fuzzy join

A fuzzy join aims to combine data based on their similarity. It relies on a distance measure to find all pairs $(x, y)$ in the input dataset(s) with a distance $(d)$ bellow some pre-defined threshold $\varepsilon$. Different solutions have been proposed for big data systems [11], [26], [29], [30], [33], [35]–[37]. The surveys [24], [25] have been written on MapReduce-based fuzzy join studying supported data types (fixed-length string, variable-length string, numeric, vector, set) and distance functions (Hamming distance, Edit distance, Jaccard similarity, Tanimoto Coefficient, Cosine Coefficient, Ruzicka similarity, Dice Similarity, Set Cosine Sim, Vector Cosine Sim). Continuing to develop our previous studies, thus, throughout this paper, we focus on fuzzy join algorithms, using Hamming distance [29], [33] with fixed-length data inputs ($b$-bit strings), and show it as the examples. Our proposal can be easily extended to a general $\Sigma^q$ finite set of the alphabet which has $q$ elements ($q \geq 2$).

Hamming distance $(HD)$ between two strings $s, t$ is the number of positions in which they differ. Given a set, $S$, of $b$-bit strings, a fuzzy join is stated using a Hamming distance used to define similarity and a threshold $\varepsilon$ is

$$\{(s, t) | s, t \in S, HD(s, t) \leq \varepsilon\}$$

The *Ball of radius* $r$ $(B_r)$ signifies all close elements of any given element within a distance $r$. The *Hamming ball of radius* $r$ $(B_r)$ can be obtained by flipping the value of at most $r$ bits of any given $b$-bit string. Thus, it is computed by the following formula [29]:

$$|B_r| = \sum_{k=0}^{r} \binom{b}{k} \approx b^r / r!$$

$B_r(s)$ consists of all similar elements in the ball of radius $r$ around of $s$. In other words,

$$\forall t \in B_r(s), d(s, t) \leq r$$

### C. Filters

A Bloom Filter (BF) [34] is a space-efficient randomized data structure used for testing membership in a set with a small rate of false positives. Figure 2 presents a Bloom Filter structure consisting of $m = 12$ bits, $k = 3$ independent hash functions, and a set $S$ of $n$ elements represented by $BF(S)$. BF never returns false negatives. However, it can return false positives. A false positive element of BF is an element that does not belong to a set $S$ while testing it on BF leads to the opposite result. Indeed, in some cases, a hash function can return the same value for multiple elements. As a consequence, an element that does not belong to $S$ can also have a hash value at its position of 1. BF is a space-efficient structure to accelerate queries. The size of a filter is fixed, independently of the number $n$ of elements. However, there is a relation between the size of the structure $m$ and the false positive probability [38]

$$f_{BF(S)} = (1 - (1 - \frac{1}{m})^{nk})^k$$

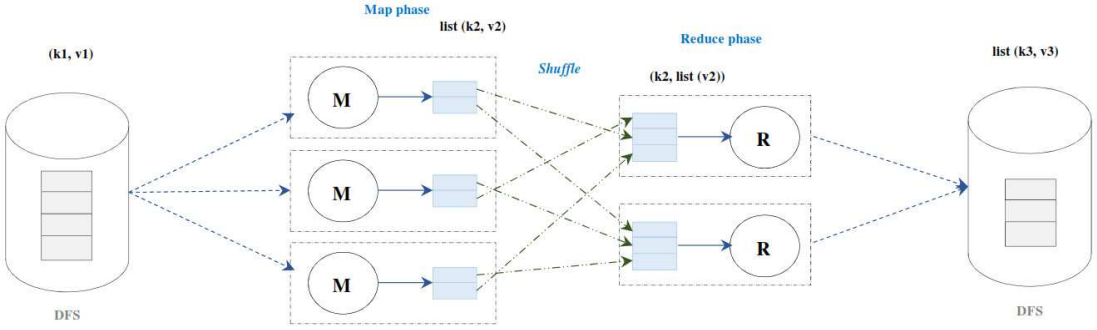


Fig. 1. MapReduce Execution

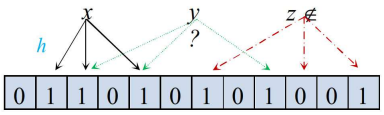


Fig. 2. A Bloom filter $BF(S)$ with 3 hash functions.

The BF and its variations have proven increasingly importance for many applications [39].

Intersection Bloom Filter (IBF) [8] is an extension of BF, represents the intersection of the datasets to be joined. IBF is used to test an element if it is a disjoint element.

SuRF [40] is a fast and compact filter that provides exact-match filtering, range filtering, and approximate range counts. It is a tree-based filter structure. So it does not fit with the fuzzy join problem because it has to browse through all the branches to find its real close elements, lead to inefficient and high costs.

In our study, fuzzy filters have not been proposed yet.

### *D. Fuzzy Join Algorithms in MapReduce*

This paper aims to improve fuzzy joins in a MapReduce environment, relying on Filters. In order to compare the costs of different algorithms, it adapts a previous model $(M, C, R)$ [29], where $M$, $R$, $C$ are used to measure the effectiveness of an algorithm. The notations and parameters are described in Table I.

TABLE I
SYMBOLS AND DESCRIPTION

| Notation | Description |
|---|---|
| $M$ | Total computation (map or preprocessing) cost for all input records |
| $C$ | Total communication cost (network resources) to transfer data from the mappers to the reducers. Other operations such as copying, comparing, hashing are performed at a unit cost |
| $R$ | Total computation cost for all reducers |
| $S, \|S\|$ | Input dataset $S$ and its size |
| $r$ | Pre-specified threshold of distance |
| $s, t, b$ | A string $s$ or $t$ and its length |
| $B_r$ | Ball of radius $r$ |
| $k$ | Number of hash functions |
| $m$ | Size of Bloom filter (bit) |
| $D$ | Size of intermediate data for shuffle |

**Naive Algorithm** [29] can be used for any data type and distance function. It relies on a single MapReduce job. The main idea is to distribute each input record to a small set of reducers so that any two records be mapped to at least one common reducer for computing distance. Each input record must be compared with all others leading to data redundancy and inefficiency.

**Ball Hashing Algorithms** [29] (BH) rely on the "ball of radius $r$" to reduce unnecessary comparisons. This means that each record is compared to the others within its similarity radius. To do this, there is one reducer for each of the $n$ possible strings of length $b$. The number of reducers is thus $n = 2^b$. The mappers generate all elements $t$ in the ball of radius $r$ of each input record $s$ ($B_s(r)$) as (key, value) pairs of the form $(s, -1)$ and $(t, s)$ such that $t \neq s$ and send them to the corresponding reducers. $t$ is a string obtained from $s$ by changing $i \in [1, r)$ bits. An issue with BH is data duplication due to $t$ - $s$ and $s$ - $t$ similarity. A proposed solution is to proceed lexicographically [41]. A mapper only emits $(t, s)$ if $t < s$. However, redundant data still exist because similar records in $B_s(r)$ are sent to reducers although they are not elements in $S$. $\text{BH}_2$ is an extension of BH. The difference is that during the map phase, $\text{BH}_2$ generates balls of radius $r/2$. Because of this, every reducer is active and checks for the similarity between all the possible combinations of two strings it receives and eliminates the duplicate outputs.

**Splitting Algorithm** [29] is based on a principle of which have any of two similarity b-bit strings with a distance less than $r$, there exists at least one same substring of length $b/(r+1)$. Mappers decompose each input string $s$ into $r+1$ equal-length substrings $s_1, s_2, ..., s_{r+1}$ and emits $(s_i, s)$. Reducers test each string to see if it is within distance $r$ of all other received strings, similar to the Naive algorithm. To avoid duplicate results, when a reducer in the $i$th family finds that $s$ and $t$ are at distance $r$ or less, it checks that there is no $j < i$ for in which $j$th substrings are also equal and outputs $s, t$ if there is no such $j$. However, the Splitting algorithm has also the same issue of redundant data as the Ball hashing algorithm.

**Bloom filter based fuzzy joins** [33] During the map phase, BH generates all elements within a distance $r$ from $s$ and sends them to the reducers for combining with similar input records.

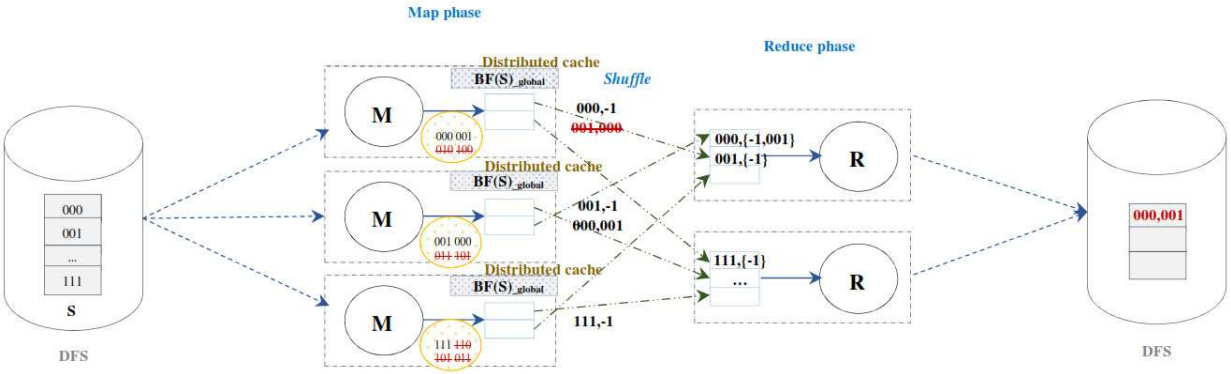


Fig. 3. Join processing stage of BF-BH Algorithm

It is easy to see that not all elements in the $B_r(s)$ belong to $S$. The Splitting algorithm also generates redundant data by sending each record to $r+1$ reducers. In fact, each record just needs to be sent to some identified reducers if all its actual similar elements present in S and its substrings are known. Our approach integrates $BF(S)$ to remove elements in $B_s(r)$ that do not belong to $S$ before sending it to the reducers in the BF-BH algorithm. With the BF-Splitting, we propose to combine Ball Hashing, Splitting and BF. By the membership test in $BF(S)$, it determines which elements $t \neq s$ in $B_r(s)$ may actually be similar to $s$. Then each of them is divided in to $r+1$ equal-length substrings $s_i$ and $t_i, i = 1..(r+1)$. For each $s_i$, if there exists a substring $t_i$ of $t$ in the intersection of $S$ and $B_r(s)$ that matches with $s_i$ , the pair $(s_i, s)$ will be output, and then $t$ will never be considered again. This solution consists of two stages: (1) in pre-processing stage, the filter is built on a join key-value set of the input dataset $S$; (2) in join processing stage, the filter is distributed to all the computing nodes and used to eliminate non-similar elements of the input dataset in each ball of radius $d$ during the map phase. After filtering none relevant data, the join algorithm then proceeds on real similar elements with a small false positive probability. An example of the join stage of BF-BH for the fuzzy join with 3-bit string and threshold $r = 1$ is shown in Figure 3.

Discussion: by the theoretical analysis and cost model in [33], applying filters will improve processing costs. However, Ball of radius $r$ computation is sensitive to distance. With the greater the distance $d$, the number of elements in $B_r$ increases dramatically. [41] has shown the slowness of the BH algorithm without filters. On the other hand, for a finite set of alphabets, the elements in every ball are determined. The ball recalculation for each record is redundant and unnecessary while the processing cost of this calculation is worth considering. From these limitations of the existing solutions, we propose a new filter type called Fuzzy filter to avoid excess calculations.

## III. Modeling Fuzzy Filter

This section shows the approach to build the Fuzzy filter with the criteria: small size, in-memory, quick response, without false negative probability. As illustrated in Figure 4, with a query $y$, a fuzzy filter has to give a quick test:

$$\exists x \in S, d(x, y) \leq r? \text{ Which } x?$$
$$\text{or } \exists x \in S \cap B_r(y)? \text{ Which } x?$$

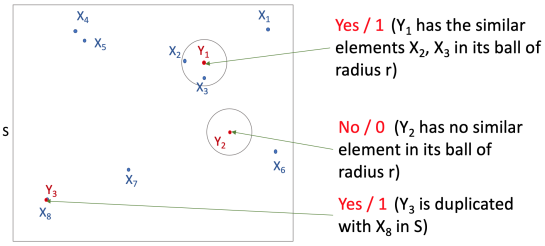


Fig. 4. Fuzzy filter modeling

For convenience during this paper, we assume that all the balls of radius r in the finite alphabet set are known, regardless of the distance function. This calculation will be discussed later.

### A. Fuzzy filter structure

The idea begins with finding the intersection between dataset and balls. The Fuzzy filter $FF(S)$ combines a Bloom filter $BF(S)$ to identify elements, and a table to store real similar elements in the ball of each element. We illustrate this approach by Figure 5

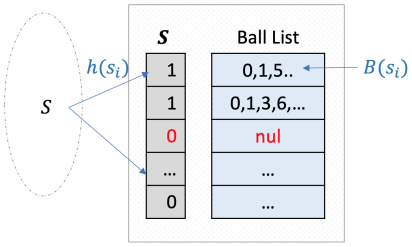


Fig. 5. FF(S) structure

The $m$ bit Bloom filter $BF(S)$ uses one hash function $h$ to calculate positions for an element of $S$ and sets the bit at the resulting positions. The ball list is an array of size $m$ of $m$ bit Bloom Filters (a matrix of $m \times m$ bits), each one stores its ball $BF(B(s_i))$.

The build operation of the fuzzy filter $FF(S)$ is described as follows.

- (0) Hash each ball to a bit array of length $m$. With $b$-bit string, it exists $2^b$ balls, each ball has about $b^r/r!$ elements. Let us recall the assumption that hash operation performs in unit time. This step has the cost $C_{(0)} \approx 2^b b^r/r!$
- (1) Hash $S$ to bit array of length $m$. $C_{(1)} = |S|$
- (2) Ball list compute by the intersection of $BF(S)$ and $BF(B(s_i))$. $C_{(2)} = |2^b|$
- The build cost for $FF(S)$ is

$$C_{FF(S)-build} \approx \frac{b^r}{r!}2^b + |S| + 2^b = (\frac{b^r}{r!} + 1)2^b + |S|$$

For clarity, we consider an example of the FF build with $b = 4, r = 1, S = (0000, 1010, 1110, 1000)$ illustrated in Figure 6

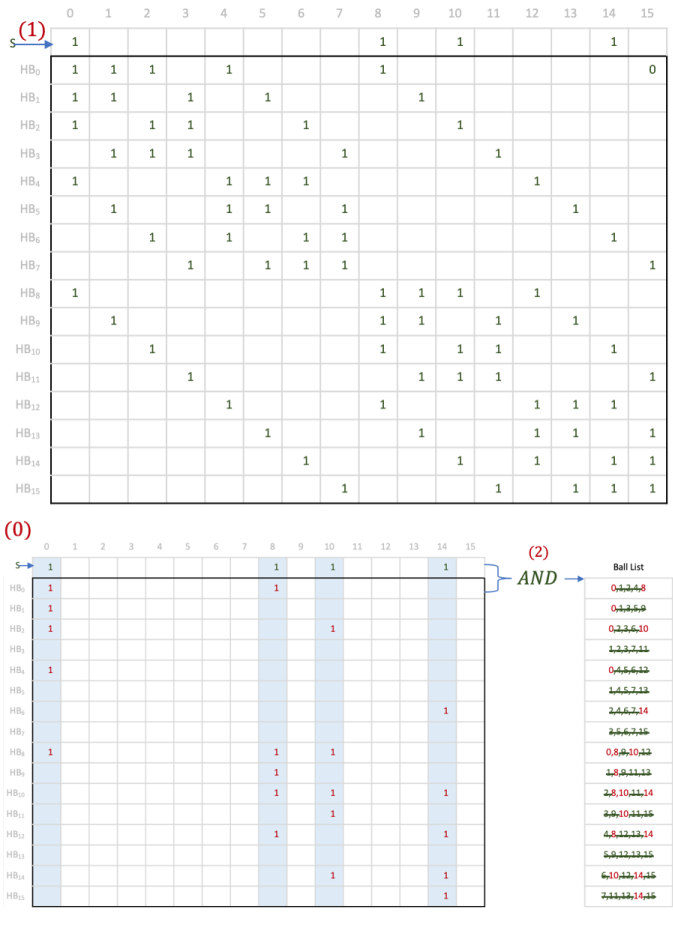


Fig. 6. Example of FF build with b=4, r=1, S=(0000,1010,1110,1000)

Consider the query examples:

- $Y_1 = 0000 \xrightarrow{\text{h(0000)=0}} S(0) = 1, B_1(0000) = \{S_0, S_8\} = \{0000, 1000\}$
- $Y_2 = 0111 \xrightarrow{\text{h(0111)=7}} S(7) = 0, B_1(0111) = \{\emptyset\}$
- $Y_3 = 1011 \xrightarrow{\text{h(1011)=11}} S(11) = 0, B_1(1011) = \{S_{10}\} = \{1010\}$

By this way, each query has a quick response in $O(1)$

### B. Optimizing large-scale fuzzy joins

With the integration of FF, our proposal ignores the costly and redundant ball calculation. Specifically, the FF-FJ algorithm consists of two phases:

- Stage 1 (Pre-processing): A filter $FF(S)$ is built on a join key set of the input dataset $S$. Each worker hashes tuples of input splits to find $h(s_i)$, emits a list of $[h(s_i)]$ to one reducer for the FF building. Thus, the Map cost is $M = |S|$, the communication cost is $D = \#mappers$, the computation cost is the FF building cost $C_{FF(S)-build} \approx \frac{b^d}{d!}2^b + 2^b = (\frac{b^d}{d!} + 1)2^b$. If the ball list is pre-known, the processing cost is only $2^b$ of AND operations. Figure 7 describes an example of the pre-processing stage of FF-FJ for the fuzzy join.
- Stage 2 (Join processing): $FF(S)$ is distributed to all the computing nodes and is used to quickly emit real similar elements of the input dataset during the map phase. Each record $s$ is hashed by $h(s)$. $BF[h(s)]$ returns a list of similar elements. Finally, mappers emit the intermediate tuple with the key is in form $(h(s), h(t_i))$ if $h(s) < h(t_i)$ and vice versa. The number of reducers is $(2^b)(2^b - 1)/2$

$$s \xrightarrow[FF(S)]{map} \begin{cases} ((h(s), h(t_i)), s), & \forall t_i \in B_r(s) \cap S, h(s) < h(t_i) \\ ((h(t_i), h(s)), s), & \forall t_i \in B_r(s) \cap S, h(t_i) < h(s) \end{cases}$$

The map cost for each record in this phase is only 1, instead of $k|B_r|$ in BF-BH algorithm. The number of intermediate elements for each record is optimized, instead of $|B_r|$ in BH algorithm, and may be approximate with BF-BH algorithm because of the same filtering technique.

In the ideal case, regardless of the false positive, our approach has no redundant intermediate data and no duplicated results without verification in reducers. An example of join stage of FF-FJ for the fuzzy join with 4-bit string and threshold $r = 1$ is shown in Figure 8.

### C. FF analysis and optimization

With an overview structure as above, FF can be applied to some data types (string, vector, set), some distance functions (Hamming, edit distance) as long as the balls can be calculated. In the binary space, FF uses $m = 2^b$, the exact probabilities of filtering are guaranteed 100%, without false probabilities. However, in practice, for a large finite alphabet set, a large string length, to optimize memory, the filter size is designed to be smaller than the actual set size. Hence, it may lead to a false probability.

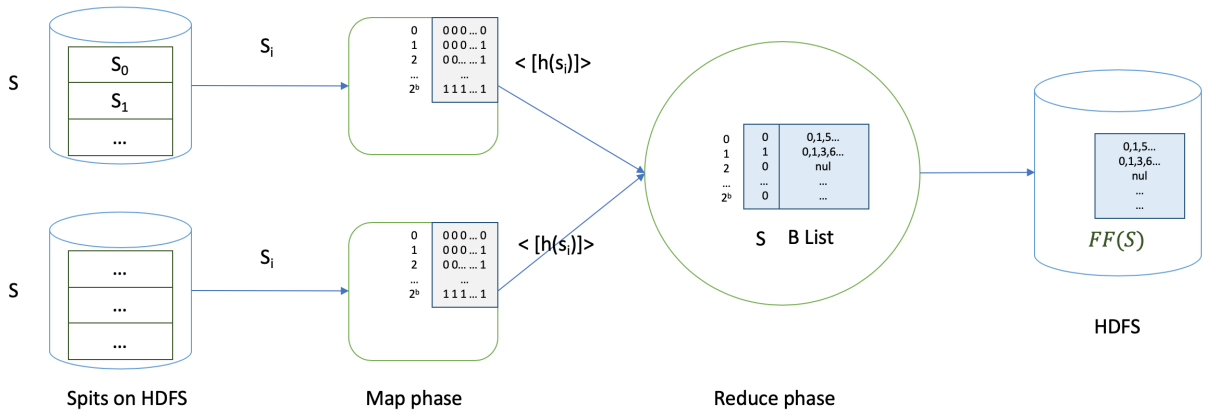


Fig. 7. FF-FJ Pre-processing stage

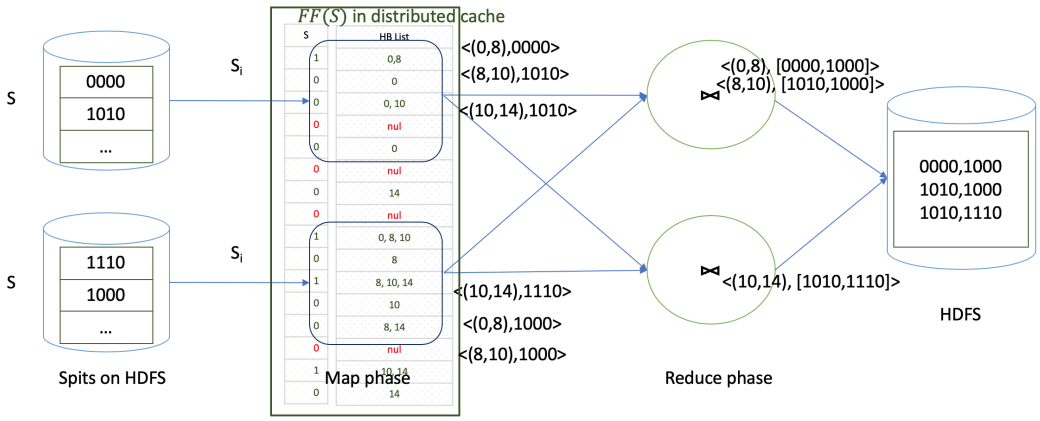


Fig. 8. Join processing stage of FF-FJ Algorithm with b=4, r=1, S=(0000,1010,1110,1000)

In the case of multiple balls that have the same hash index position, the ball in this position is the union of these collision balls. The response will include the real similar elements and also the mistaken records in another collision ball. These records are mistakenly assumed to be a similar element and must be calculated the distance in the join step.

The small false positive probability is caused by one of two cases follows

- (1) for the filter: an element in another collision ball is returned as an answer.
- (2) for the join step: an irrelevant record of $S$ has the same hash index with an exact answer.

Conversely, it does not exist a false negative probability. In other words, no real similar element is not answered in the response. The false positive probability is shown in Figure 9

The precision of the fuzzy filter depends on the similarity function complexity, the size of FF ($m$), the quality of the hash function. For example, with Hamming distance threshold $r$, a dataset $S$ of $b$ bit-strings, for $n$ real elements ($|S| = n < 2^b$), the false positive of a hash bucket list of size $m$ bits is

$$f_S = 1 - (1 - 1/m)^n$$

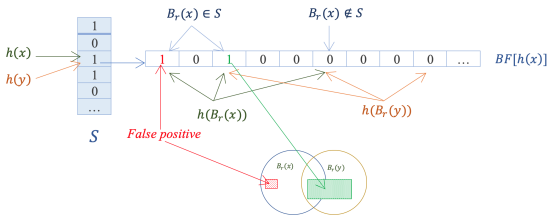


Fig. 9. FF-FJ filtering process

Each hash ball contains about $b^r/r!$ elements. So its number of possible collision bits is approximate $(b^r/r!)(1-1/m)^{b^r/r!}$. If a collision occurs, the probability of a bit 1 is out of the real ball is

$$\frac{b^r(1-1/m)^{b^r/r!}}{mr!}(1 - \frac{b^r(1-1/m)^{b^r/r!}}{mr!})$$

However, this false bit becomes a false positive answer only if its index in hash bucket S is also set.

The building of the matrix that includes all the balls with a reasonable alphabet, an acceptable length of a string is feasible. For large-scale datasets, avoiding repeated calculations for the pre-known balls will reduce a large workload.

In cases where the set of balls cannot be pre-calculated, the input dataset S can be read one time as a distinct key set to compute its balls. This cost can be amortized, especially using streaming or caching techniques (e.g Spark [42]).

Another advantage of FF is the flexibility with distance, capable of equi-join and fuzzy join. The ball list can be easily updated quickly as soon as a new record appears. This can be applied in stream join applications. The solution given is that the AND operation in step (3) during the building phase is not performed. The ball list stores all the balls. The answer to each new query $t$ is the intersection of $S$ and the ball $B(t)$.

## IV. Conclusion

In this paper, we study theoretical details on large-scale fuzzy join algorithms in MapReduce. We propose approaches for building a Fuzzy Filter, a scalable solution with respect to the distance and the volume of the input datasets. This filter is a compact, probabilistic data structure that supports very fast similarity queries by maintaining a bit matrix, with small false positive rate and zero false negative rate. We show the relevance of this structure in fuzzy self join. In addition, our solution for the FF-FJ algorithm is more efficient than previous solutions without filters or with a Bloom Filter since it significantly reduces redundant data, costly and wasteful computations, and thus produces fewer intermediate data, eliminates duplicated results, and avoids unnecessary comparisons. Although FF-FJ algorithm has false positives and an extra cost for the pre-processing step, its efficiency in space-saving and filtering often outweighs these drawbacks. We use the MapReduce cost model to prove it.

Future work includes extending Intersection Fuzzy Filter for fuzzy two-way join, fuzzy multiway join and fuzzy recursive join. Our optimizations may be extended in the cache or streaming supported framework to reuse the pre-processing cost. Perspectives also include validating our solutions, comparing them with other approaches and extending the research for other fuzzy join algorithms. Besides, experimental evaluation and a solution for skewness problems will also be considered.

## References


[1] M. Bamha and G. Hains, "An efficient equi-semi-join algorithm for distributed architectures," in *Proceedings of the 5th International Conference on Computational Science - Volume Part II*. Springer-Verlag, 2005, p. 755–763.

[2] L. Michael, W. Nejdl, O. Papapetrou, and W. Siberski, "Improving distributed join efficiency with extended bloom filter operations," 05 2007, pp. 187–194.

[3] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in mapreduce." Association for Computing Machinery, 2010, p. 975–986.

[4] F. N. Afrati and J. D. Ullman, "Optimizing joins in a map-reduce environment," in *Proceedings of the 13th International Conference on Extending Database Technology*. Association for Computing Machinery, 2010, p. 99–110.

[5] M. A. H. Hassan and M. Bamha, "Semi-join computation on distributed file systems using map-reduce-merge model," in *Proceedings of the 2010 ACM Symposium on Applied Computing*. Association for Computing Machinery, 2010, p. 406–413.

[6] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, "Map-reduce extensions and recursive queries," in *Proceedings of the 14th International Conference on Extending Database Technology*. Association for Computing Machinery, 2011, p. 1–8.

[7] T. Lee, K. Kim, and H.-J. Kim, "Join processing using bloom filter in mapreduce," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*. Association for Computing Machinery, 2012, p. 100–105.

[8] T.-C. Phan, L. d'Orazio, and P. Rigaux, "Toward Intersection Filter-based Optimization for Joins in MapReduce," in *Cloud-I*, 2013, pp. 2:1–2:2.

[9] N. Bruno, Y. Kwon, and M.-C. Wu, "Advanced join strategies for large-scale distributed computation," *Proc. VLDB Endow.*, vol. 7, no. 13, p. 1484–1495, Aug. 2014. [Online]. Available: https://doi.org/10.14778/2733004.2733020

[10] T. Phan, L. d'Orazio, and P. Rigaux, "A Theoretical and Experimental Comparison of Filter-Based Equijoins in MapReduce," *TLDKS*, vol. 25, pp. 33–70, 2016.

[11] R. Vernica, M. J. Carey, and C. Li, "Efficient Parallel Set-similarity Joins Using MapReduce," in *SIGMOD*, 2010, pp. 495–506.

[12] S. Chaudhuri, V. Ganti, and R. Kaushik, "A primitive operator for similarity joins in data cleaning," in *ICDE*, 2006.

[13] A. Doan, A. Halevy, and Z. Ives, *Principles of Data Integration*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[14] A. Metwally, D. Agrawal, and A. El Abbadi, "Detectives: Detecting Coalition Hit Inflation Attacks in Advertising Networks Streams," in *WWW*, 2007, pp. 241–250.

[15] E. Spertus, M. Sahami, and O. Buyukkokten, "Evaluating similarity measures: A large-scale study in the Orkut social network," in *SIGKDD*, 2005, pp. 678–684.

[16] M. Henzinger, "Finding Near-duplicate Web Pages: A Large-scale Evaluation of Algorithms," in *SIGIR*, 2006, pp. 284–291.

[17] T. C. Hoad and J. Zobel, "Methods for identifying versioned and plagiarized documents," *JASIST*, vol. 54, pp. 203–215, 2003.

[18] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig, "Syntactic Clustering of the Web," in *WWW*, 1997, pp. 1157–1166.

[19] M. Sahami and T. D. Heilman, "A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets," in *WWW*, 2006, pp. 377–386.

[20] S. Wandelt, J. Starlinger, M. Bux, and U. Leser, "Rcsi: Scalable similarity search in thousand(s) of genomes," *Proc. VLDB Endow.*, vol. 6, p. 1534–1545, 2013.

[21] Y. Jiang, G. Li, J. Feng, and W.-S. Li, "String similarity joins: An experimental evaluation," *Proc. VLDB Endow.*, vol. 7, p. 625–636, 2014.

[22] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava, "Approximate string joins in a database (almost) for free," in *Proceedings of the 27th International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, p. 491–500.

[23] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[24] F. Fier, N. Augsten, P. Bouros, U. Leser, and J.-C. Freytag, "Set similarity joins on mapreduce: An experimental survey," *Proc. VLDB Endow.*, vol. 11, p. 1110–1122, 2018.

[25] Y. N. Silva, J. Reed, K. Brown, A. Wadsworth, and C. Rong, "An Experimental Survey of MapReduce-Based Similarity Joins," in *Similarity Search and Applications*, 2016, pp. 181–195.

[26] A. Metwally and C. Faloutsos, "V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors," *CoRR*, vol. abs/1204.6077, 2012. [Online]. Available: http://arxiv.org/abs/1204.6077

[27] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng, "Massjoin: A mapreduce-based method for scalable string similarity joins," in *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 340–351.

[28] X. Wang and D. Sun, "Qjoin: A q-sample-based method for large-scale string similarity joins," in *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*, 2018, pp. 45–48.

[29] F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman, "Fuzzy Joins Using MapReduce," in *ICDE*, 2012, pp. 498–509.

[30] Y. N. Silva and J. M. Reed, "Exploiting MapReduce-based Similarity Joins," in *SIGMOD*. ACM, 2012, pp. 693–696.

[31] A. Das Sarma, Y. He, and S. Chaudhuri, "Clusterjoin: A similarity joins framework using map-reduce," *Proc. VLDB Endow.*, p. 1059–1070, 2014.

[32] L. Chen, Y. Gao, B. Zheng, C. S. Jensen, H. Yang, and K. Yang, "Pivot-based metric indexing," *Proc. VLDB Endow.*, p. 1058–1069, 2017.

[33] T.-T.-Q. Tran, T.-C. Phan, A. Laurent, and L. d'Orazio, "Improving hamming distance-based fuzzy join in mapreduce using bloom filters," in *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, 2018, pp. 1–7.

[34] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[35] Y. N. Silva, J. M. Reed, and L. M. Tsosie, "MapReduce-based Similarity Join for Metric Spaces," in *Cloud-I*, 2012, pp. 3:1–3:8.

[36] A. Okcan and M. Riedewald, "Processing Theta-joins Using MapReduce," in *SIGMOD*, 2011, pp. 949–960.

[37] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang, "Efficient Similarity Joins for Near-duplicate Detection," *ACM TODS*, vol. 36, no. 3, pp. 15:1–15:41, 2011.

[38] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM*, 2006, pp. 1–12.

[39] A. Broder, M. Mitzenmacher, and A. B. I. M. Mitzenmacher, "Network applications of bloom filters: A survey," in *Internet Mathematics*, 2002, pp. 636–646.

[40] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Surf: Practical range query filtering with fast succinct tries," in *Proceedings of the 2018 International Conference on Management of Data*. Association for Computing Machinery, 2018, p. 323–336.

[41] B. Kimmett, V. Srinivasan, and A. Thomo, "Fuzzy Joins in MapReduce: An Experimental Study," *PVLDB*, vol. 8, no. 12, pp. 1514–1517, 2015.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*. USENIX Association, 2010, p. 10.