Improving Hamming distance-based fuzzy join in MapReduce using Bloom Filters

Thi-To-Quyen Tran, Thuong-Cang Phan, Anne Laurent, Laurent D'orazio

Improving Hamming distance-based fuzzy join in MapReduce using Bloom Filters

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

Thuong-Cang PHAN
Cantho University
Cantho, 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*—Join operation is one of the key ones in databases, allowing to cross data from several tables. Two tuples are crossed when they share the same value on some attribute(s). A fuzzy or similarity join combines all pairs of tuples for which the distance is lower than or equal to a prespecified threshold ε from one or several relations. Fuzzy join has been studied by many researchers because its practical application. However, join is the most costly and may even not be possible to compute on large databases. In this paper, we thus propose the optimization for MapReduce algorithms to process fuzzy joins of binary strings using Hamming Distance. In particular we propose to use an extension of Bloom Filters to eliminate the redundant data, reduce the unnecessary comparisons, and avoid the duplicate output. We compare and evaluate analytically the algorithms with a cost model.**

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

I. INTRODUCTION

Join is a critical operation within a data management system, making it possible to enrich data from a source with information stored outside of it. This is why literature is rich in working on join optimization, especially in parallel and distributed systems. In recent years, researches has focused on the problem of efficient joins in large-scale parallel environments. In other words, researchers toward their goals of limiting the use of resources in terms of bandwidth consumption or CPU usage. The first results, concerning the equi-join [1], impose strong constraints on the 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. A query in computer network defense for example: grouping information from a URL written in different ways (eg www.irisa.fr in a source and http://www.irisa.fr in the other), set of IP addresses (192.168.150.30, 192.168.150.31, etc.). As another query example [2] in mining social networking sites where user's preferences are stored as bit vectors (where a "1" bit means interest in a certain domain), applications wants 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]". This query is defined as a fuzzy or similarity join and arosed in many applications, including detecting attacks from colluding attackers [3], mining in social networking sites [4], detecting near duplicate web-pages in web crawling [5], document clustering [6], master data management [7].

When dealing with a very large amount of data, fuzzy join becomes a challenging problem in a distributed parallel computing environment with the expensive cost of data shuffle. As a result, the data redundancy is very difficult to accept. Vernica et al. [2] proposed a similarity join method using 3-stage MapReduce which utilized the prefix filtering method to support set-based similarity functions. Metwally et al. [8] proposed a 2-stage algorithm VSMART join for similarity join on set, multisets and vector. Afrati et al. [9] proposed multiple algorithms to perform fuzzy join in a single MapReduce stage. While recent studies on the fuzzy join have the common limitations as redundancy and duplication of data, the filter-based approaches in our recent studies [1], [10] can solve these problems. Our team was interested in using Bloom Filters [11], Intersection Filter [10]. The idea is to filter irrelevant data as soon as possible to reduce data transfers and workload on different machines.

This study, therefore, focuses on a theoretical analysis of various Hamming distance-based 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 presents the research background by the related works. Various Hamming distance-based similarity join algorithms are analysed as a research context in section 3. We propose the optimizations in section 4. We theoretically compare and evaluate the algorithms by an example in section 5. Finally, section 6 concludes and discusses future works.

II. BACKGROUND

A. MapReduce

MapReduce [12] 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 MapReduce execution.

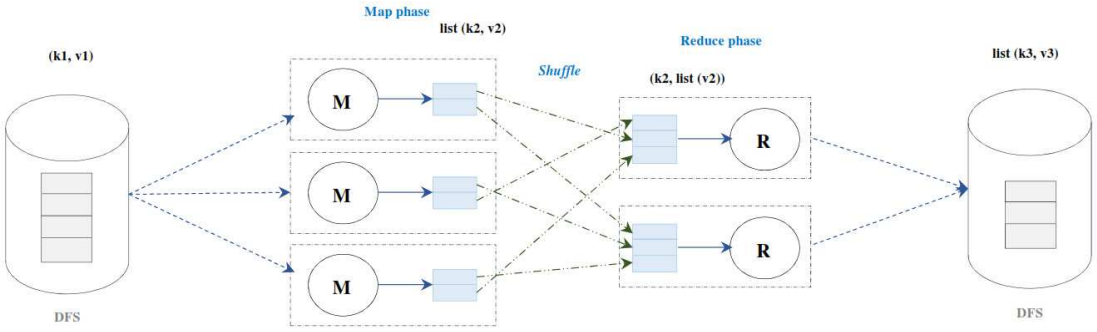

Fig. 1. MapReduce Execution

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 the 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 group 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 bellow some pre-specified threshold ε. Different solutions have been proposed for big data systems [2], [8], [9], [13]–[16]. A survey has been written on MapReduce-based fuzzy join [17] 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). In this paper, we focus on fuzzy join algorithms using Hamming distance [9] with fixed-length data inputs (b-bit strings).

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 a similarity and a threshold ε is

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

. The *ball of radius* d $(B(d))$ can be obtained by flipping the value of at most d bits of any given b-bit string. Thus, it is computed by the following formula [9]:

$$B(d) = \sum_{k=0}^{d} \binom{b}{k} \approx b^d/d!$$

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

$$\forall t \in B_s(d), HD(s, t) \leq d$$

Example: Consider the 3-bit string ($b = 3$),

- $d = 0$, the ball of radius 0 around any given element $B_{000}(0)$ now is itself (000).
- $d = 1$, the $B_{000}(1)$ now has $1 + 3 = 4$ elements $(000, 001, 010, 100)$.
- $d = 2$, the $B_{000}(2)$ now has $1 + 3 + 3 = 7$ elements $(000, 001, 010, 100, 011, 101, 110)$.

C. Bloom filter

A Bloom Filter (BF) [11] 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 bits, k independent hash functions, and a set S of n elements represented by $BF(S)$. $BF(S)$ can be described as follows:

- The set $S = \{x_1, x_2, ..., x_n\}$ of n elements is represented by an array of m bits, initially all set to 0.
- The filter uses k independent hash functions $h_1, h_2, ..., h_k$ with $h_i : x \to \{1..m\}$.
- To insert an element $x \in S$, we compute $h_1(x), h_2(x), ..., h_k(x)$, and set the corresponding positions in the bit array to 1. Once this operation has been done for each element of S, the resulting bit array can be used as an approximate representation of the set.
- To check if $y \in S$, we check whether for each of the k hash functions, the position $h_i(y)$ is set to 1 in the bit array. If at least one position is set to 0, this means that $y \notin S$. Otherwise, all positions are set to 1, that is to say that y may be a member of S with some probability.

BF never returns false negatives. However, it can return false positives. A false positive element of BF is an element that

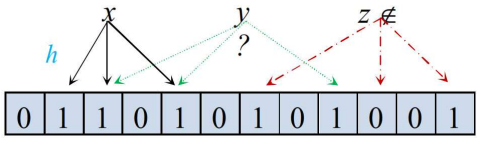

Fig. 2. A Bloom filter $BF(S)$ with 3 hash functions.

does not belong to a set S while testing it on BF lead 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 querying. 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 [18]

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

D. Motivation

This paper aims to improve fuzzy joins using Hamming distance in a MapReduce environment, relying on Bloom Filter. In order to compare the costs of different algorithms, it adapts a previous model (M, C, R) [9], 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
d	Pre-specified threshold of distance
s, t, b	A string s or t and its length
$B(d)$	Ball of radius d
k	Number of hash functions
K	Number of reducers
D	Size of intermediate data for shuffle

III. FUZZY JOIN ALGORITHMS IN MAPREDUCE

This paper studies hamming distance-based fuzzy join algorithms in MapReduce using the (M, C, R) cost model [9]. More precisely, it focuses on Naive join, Ball Hashing, Splitting and Anchor Points algorithms

A. Naive Algorithm

Naive algorithm 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. The details of Naive algorithm for an input set S are specified as follows:

- With a constant $J > 0$, let $K = \binom{J+1}{2} = J(J+1)/2$ or $J \approx \sqrt{K}$ be the number of reducers.
- Each reducer is identified by a pair (i, j), such that $0 \leq i \leq j \leq J$
- During the Map phase, all members X of S are hashed to J buckets so as to be sent to exactly J reducers (i, j) or (j, i) $\forall i = [0, J)$ $(key, value)$ pairs of the form $((i, j), X)$.
$$X \xrightarrow{map} ((i, j), X)$$
The total map cost and communication (data transfer from mappers to reducers) is $M = C = O(|S|J) = O(|S|\sqrt{K})$
- A reducer receives all records with the same key (i, j), computes the distance between each pair of records and outputs the pairs satisfying the similarity, that is to say the threshold ε. Each reducer receives around $|S|J/K = 2|S|(J+1)$ elements, which requires $\binom{2|S|/(J+1)}{2} = O(|S|^2/K)$ comparisons. As a consequence, for K reducers, the total computation cost for all reducers R is $O(|S|^2)$

The challenge is to define K in order for every pairs of elements of S to be sent to exactly one reducer and thus avoid data duplication. Each input records must be compared with all others leading to data redundancy and inefficiency.

B. Ball Hashing Algorithms

Ball Hashing is a family of two algorithms BH_1 and BH_2. These algorithms rely on the "ball of radius d" 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 $n = 2^b$.

1) BH_1:

- The mappers generate all elements t in ball of radius d of each input record s ($B_s(d)$) 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 $i \in [1, d)$ bits.
$$s \xrightarrow{map} \begin{cases} (s, -1) \\ (t, s), \quad \forall t \in B_s(d), t \neq s \end{cases}$$
Thus the map cost is $B(d)$ per input element.
- Call a reducer that receives $(s, -1)$ *"active"*, it infers that s is in the input set and outputs all pairs of similar received strings. Assuming that it is not possible for multiple input records to have the same join value, the average number of strings to be sent to each reducer is $|S|B(d)/n$. The total cost of all $|S|$ active reducers is $|S|^2B(d)/n$.
- A issue with BH_1 is data duplication due to t - s and s - t similarity. A proposed solution is to proceed lexicographically [19]. A mapper only emits (t, s) if $t < s$. However, redundant data still exist in "inactive"

reducers because similar records in $B_s(d)$ are sent to reducers although they are not elements in S.

2) BH_2: BH_2 is an extension of BH_1. The difference is that during the map phase, BH_2 generates ball of radius $d/2$. 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.

C. Splitting Algorithm

Splitting algorithm is based on a principle of which have any of two similarity b-bit strings with a distance less than d, there exists at least one same substring of length $b/(d+1)$.

- Mappers decompose each input string s into $d+1$ equal-length substrings $s_1, s_2, ..., s_{d+1}$ and emits (s_i, s).
$$s \xrightarrow{map} ((i, s_i), s), i = 1..(d+1), s_i \subset s$$
Each substring of length $(d+1)$ has $2^{b/(d+1)}$ possible values. Therefore, the number of reducers is $(d+1)2^{b/(d+1)}$. The total communication cost is $(d+1)|S|$.
- There is at least one reducer that will receive any two similar strings in S. Reducers test each string to see if it is within distance d of all other received strings, similar to the Naive algorithm. The processing cost is $(d+1)|S|^2/2^{b/(d+1)}$. To avoid duplicate results, when a reducer in the ith family finds that s and t are at distance d or less, it checks that there is no $j < i$ for in which jth 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.

D. Anchor Points Algorithm

Anchor Points algorithm is the only randomized algorithm considered. The algorithm chooses a random universe. If the set is large enough, at least one string in the set can be expected to be within distance $\lceil d/2 \rceil$ of any two strings in the input data. This algorithm will not be included in our research since the paper that introduced it showed that it is outperformed by the other algorithms [9].

IV. Bloom Filter-based fuzzy joins

The previous algorithms generate intermediate elements that may be not relevant to the join process in the map phase, because they do not match with any similar record in the input dataset. In this section, we propose to integrate BF into the join algorithms to improve performances. Naive algorithm will be used as a baseline for comparison with other solutions.

A. BF-BH1 Algorithms

During the map phase, BH_1 generates all elements within a distance d from s and sends to them to the reducers for combining with similar input records. It is easy to see that not all elements in the $B_s(d)$ belong to S. Our approach integrates $BF(S)$ to remove elements in $B_s(d)$ that do not belong to S before sending it to the reducers. This solution consists of two stages:

- Stage 1 (Pre-processing) : A filter $BF(S)$ is built on a join key value set of the input dataset S. Figure 3 describe an example of preprocessing stage of $BF-BH1$ for the fuzzy join with 3-bit string.
- Stage 2 (Join processing) : $BF(S)$ 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. An example of join stage of $BF-BH1$ for the fuzzy join with 3-bit string and threshold $d = 1$ is shown in Figure 4.
$$s \xrightarrow[BF(S)]{map} \begin{cases} (s, -1) \\ (t, s), & \forall t \in B_s(d) \cap S, t < s \end{cases}$$
After filtering none relevant data, the join algorithm then proceeds as in BH_1.

Let us recall assumption that hash operation performs in unit time. With k hash functions, the pre-processing cost on all input records is $k|S|$. However, this cost can be amortized by streaming or caching techniques.

Each membership test also uses k hash functions, so the map cost for each record is $kB(d)$.

In the shuffle phase, the number of intermediate elements for each record will be reduced, instead of $B(d)$. Precisely, if we note δ_S the ratio of similar records of S, $f_{BF(S)}$ the false positive probability of the BF of S, then the cost to transfer intermediate data from mappers to reducers is

$$D_{BF-BH1} = |S|[\delta_S B(d) + f_{BF(S)}(1 - \delta_S)B(d)]$$
$$D_{BF-BH1} < |S|B(d)$$

As a consequence, the processing cost in the reduce phase is also improved, the reduction being: $D_{BF-BH1}|S|/n$

BF can also be used in $BH2$ and will be the subject of future works. In particular, we envision to address multiple inputs with Intersection Filters [1], [10].

B. BF-Splitting algorithm

The Splitting algorithm generates redundant data by sending each record to $d+1$ reducers. In fact, each record just need to be sent to some identified reducers if all its actual similar elements present in S and its substrings are known. As a solution we propose to combine Ball Hashing, Splitting and BF.

This approach also requires a pre-processing stage for building $BF(S)$ with a cost of $k|S|$. The join stage is described as follows:

- A mapper generates all elements in the ball of radius d around each input record s. By the membership test in $BF(S)$, it determines which elements $\{t \neq s\}$ in $B_s(d)$ may actually be similar to s. Then each of them is divided in to $d+1$ equal-length substrings $\{s_i\}$ and $\{t_i\}$, $i = 1..(d+1)$. For each s_i, if there exists a substring t_i of t in the intersection of S and $B_s(d)$ that matches with s_i, the pair (s_i, s) will be outputs, and then t will never be considered again.

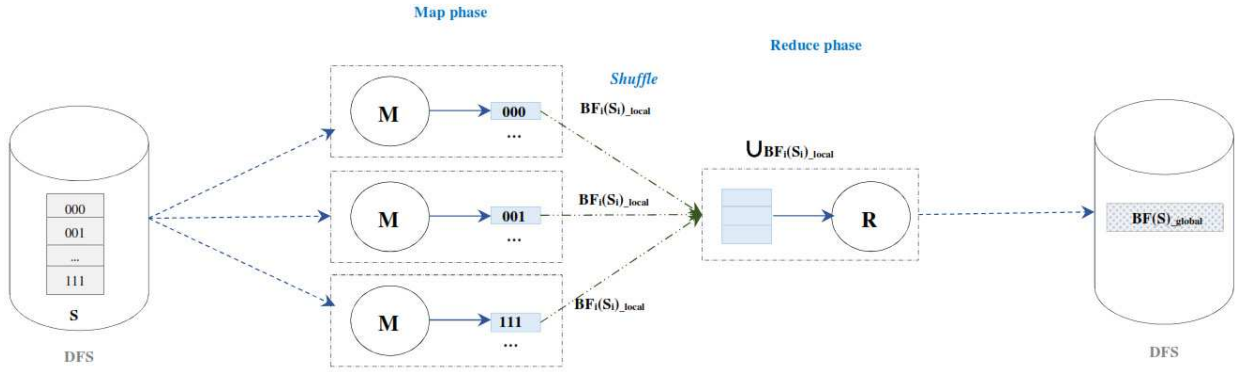

Fig. 3. Pre-processing stage

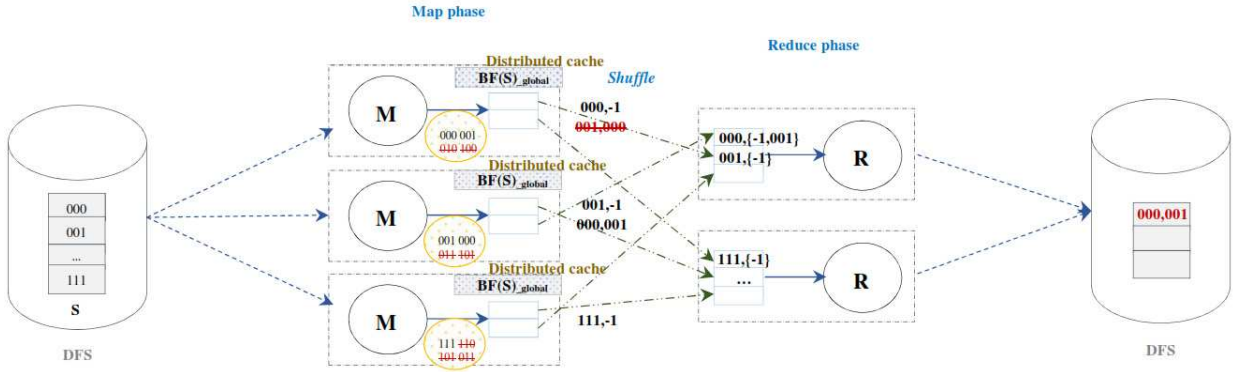

Fig. 4. Join processing stage of BF-BH1 Algorithm

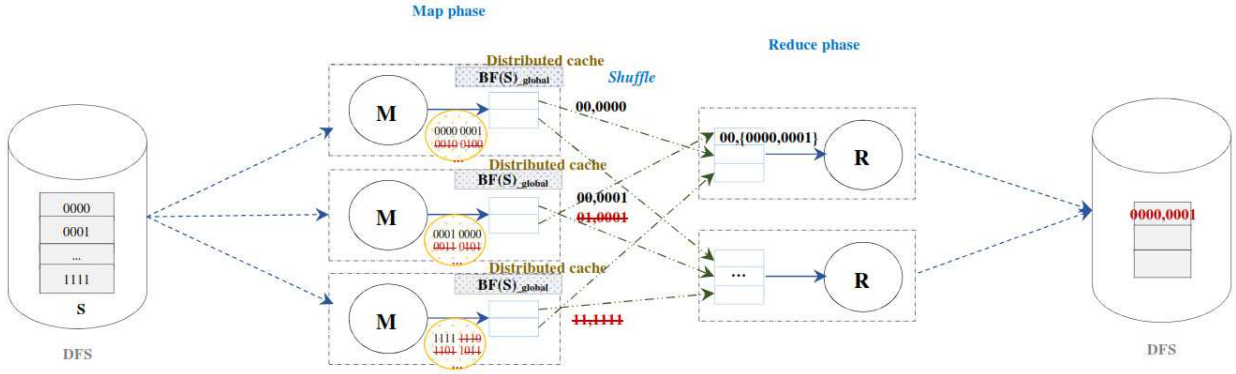

Fig. 5. Join processing stage of BF-Splitting Algorithm

$$s \xrightarrow[BF(S)]{map} (s_i, s) \begin{cases} s_i \subset s \\ \forall t \in B_s(d) \cap S \\ \exists t_i \subset t \equiv s_i \end{cases}$$

An example of join stage of $BF - Splitting$ for the fuzzy join with 4-bit string and threshold $d = 1$ is shown in Figure 5. The map cost is $k|S|B(d)(d+1)$.

- Each record is sent only if there is actual similar elements, with a small false positive. The communication cost is

$$D_{BF-Splitting} = [\delta_S|S| + f_{BF(S)}(1 - \delta_S)|S|]$$

$$D_{BF-Splitting} < (d+1)|S|$$

- The reducers collect, test the distance, and output records as in the Naive algorithm. However, in such an approach, each similar pair in S is sent to at most one reducer, solving the duplicated output problem without a lexicography test. The total computation cost for reducers is $D_{BF-Splitting}|S|/2^{b/(d+1)}$

TABLE II
SUMMARY OF COSTS FOR VARIOUS HAMMING DISTANCE-BASED JOIN ALGORITHMS

Approche	Pre-processing	Map cost per element	# Reducers	Communication	Processing
Naive	0	$J \approx \sqrt{K}$	K	$\|S\|\sqrt{K}$	$\|S\|^2$
BH1	0	$B(d)$	$n = 2^b$	$\|S\|B(d)$	$\|S\|^2 B(d)/2^b$
BF-BH1	$k\|S\|$	$kB(d)$	$n = 2^b$	$D_{BF-BH1} < \|S\|B(d)$	$D_{BF-BH1}\|S\|/2^b$
Splitting	0	$d+1$	$(d+1)2^{b/(d+1)}$	$(d+1)\|S\|$	$(d+1)\|S\|^2/2^{b/(d+1)}$
BF-Splitting	$k\|S\|$	$kB(d)(d+1)$	$(d+1)2^{b/(d+1)}$	$D_{BF-Splitting} < (d+1)\|S\|$	$D_{BF-Spitting}\|S\|/2^{b/(d+1)}$

TABLE III
VALUE OF EXPRESSIONS FROM TABLE II WHEN $b = 20, d = 4, |S| = 10^5, K = 10^4, \delta_S = 1\%, k = 8, f_{BF(S)} = 10^{-4}$

Approche	Pre-processing	Map cost per element	# Reducers	Communication	Processing
Naive	0	100	10^4	10^7	10^{10}
BH1	0	6226	10^6	6.2×10^8	6.2×10^7
BF-BH1	8×10^5	49808	10^6	6.26×10^6	6.26×10^5
Splitting	0	5	80	5×10^5	3.1×10^9
BF-Splitting	8×10^5	249040	80	10^3	6.2×10^3

V. SYNTHESIS

Table II summarizes the costs of the different algorithms. According to the processing cost, Naive algorithm is the most expensive solution, but its cost is independent with the change of distance. With respect to the communication cost, Splitting algorithm is the best approach, while Ball Hashing is the most suitable solution to processing cost. However, Ball Hashing is sensitive to distance. With the greater the distance d, the number of elements in $B(d)$ increases dramatically. Integrating BF in the algorithms implies the following changes according the (M, C, R) model:

- The pre-processing cost is incurred by reading the input to generate $BF(S)$. However this cost can be amortized, especially using streaming or caching techniques (e.g Spark [20]).
- The map phase use k hash functions for the membership test. In the BF-Splitting, the map phase generates $B_s(d)$ for each input record.
- The number of reducer does not change.
- Using $BF(S)$, redundant elements are eliminated, thus the communication cost is reduced. This also leads to a decrease of the computation cost on reducers.

Table III compares the costs of algorithms via a concrete example [9]. We choose $b = 20$, so $n = 2^20 \approx 10^6$. We use $d = 4$, so $B(d) = 6226$. We also take $|S|$ be 10000. For the Naive algorithm, we take $K = 10000$. We assume that the the ratio of similar records of S is $\delta_S = 1\%$, the small false positive of $BF(S)$ is $f_{BF(S)} = 0.0001$. As a conclusion, no algorithm is the best. Choosing a solution depends on the context. However, in a parallel and distributed environment, communication cost is one of the most important factors. Experiments in our previous studies [1], [10] have proved that filtering can significantly improve execution times.

VI. CONCLUSIONS

In this paper, we study theoretical details for the fuzzy join algorithms based on Hamming distance measure in MapReduce, applied for b-bit strings input dataset. We propose the optimization for the Ball Hashing and Splitting algorithms, and show the comparison through the MapReduce cost model. Our approaches eliminate the redundant intermediate data, reduce the unnecessary comparisons and avoid the data duplication. For the fuzzy join of multiple input datasets, Intersection filter [10] is applied instead of Bloom filter. Our optimizations may be extended in the cache or streaming supported framework to reuse the preprocessing cost. In the future works, we continue to validate our works, compare with other approachs and extend the research for other fuzzy join algorithms.

REFERENCES

[1] 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.

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

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

[4] 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.

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

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

[7] 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.

[8] 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

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

[10] 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.

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

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

[13] 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.

[14] Y. N. Silva and J. M. Reed, "Exploiting MapReduce-based Similarity Joins," in *SIGMOD*. ACM, 2012, pp. 693–696.
[15] A. Okcan and M. Riedewald, "Processing Theta-joins Using MapReduce," in *SIGMOD*, 2011, pp. 949–960.
[16] 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.
[17] 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.
[18] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM*, 2006, pp. 1–12.
[19] B. Kimmett, V. Srinivasan, and A. Thomo, "Fuzzy Joins in MapReduce: An Experimental Study," *PVLDB*, vol. 8, no. 12, pp. 1514–1517, 2015.
[20] "Apache SparkTM - Lightning-Fast Cluster Computing." [Online]. Available: http://spark.apache.org/