UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES 1

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par

Thi To Quyen TRAN

Filters based fuzzy big joins

Thèse présentée et soutenue à Lannion , le 17 Décembre 2020
Unité de recherche : IRISA

Rapporteurs avant soutenance :

Marie-Jeanne Lesot	Maître de conférences HDR, Sorbonne Université, LIP6
Sofian Maabout	Maître de conférences HDR, Bordeaux Université, LaBRI, CNRS

Composition du Jury :

Examinateurs :	Bernd Amann	Professeur, Sorbonne Université, LIP6, CNRS
	Christophe Bobineau	MCF, Université de Grenoble Alpes, CNRS, Grenoble INP, LIG
	Olivier Pivert	Professeur, Université de Rennes 1, CNRS, IRISA
Dir. de thèse :	Laurent D'Orazio	Professeur, Université de Rennes 1, CNRS, IRISA
Co-dir. de thèse :	Anne Laurent	Professeur, Université de Montpellier, LIRMM, CNRS
	Thuong Cang Phan	Enseignant, Université de Can Tho

ACKNOWLEDGEMENT

It is a special feeling of great pleasure and happiness to look over the past journey and remember all my advisors, friends and family who have helped and supported me along this long but fulfilling road.

Foremost, I would like to express my heartfelt gratitude to my advisors Professor Laurent D'ORAZIO and Professor Anne LAURENT for the continuous support of my Ph.D study and research, for their patience, facilitation, enthusiasm, and immense knowledge. Their invaluable guidance, advice, and encouragement helped me all the time in doing research and writing this dissertation. I believe that I could not do well my thesis without my advisors.

I would like to say a special thank you to my co-supervisor, Dr. Thuong-Cang PHAN, for his constant encouragement and guidance. He is a great supervisor and also a good collaborator, as well as a good friend. He helped me a lot with being a good researcher and a better person.

Besides my advisors, I would also like to thank my examiners, Professor Marie Jean Lesot and Professor Sofian Maabout, who provided encouraging and constructive feedback. It is not an easy task when reviewing a thesis, and I am grateful for their thoughtful and detailed comments. I would like to thank the rest of my thesis committee: Professor Bernd Amann, Professor Christophe Bobineau and Professor Olivier Pivert for their encouragement, insightful comments, and critical questions.

I wish to sincerely thank Rennes 1 University, IRISA and ENSSAT Lab, especially Mrs. Joëlle Thépault and Mrs. Angélique Le Pennec, who helped me to have a good location for research in the past three years. Thank you to Mr. Christian Sauquet, manager of the ENSSAT library for his availability, his support, and his kindness.

I give sincere thanks to Galactica platform of ISIMA Lab, especially Mr. Frédéric Gaudet who were responsible for the cluster which were very important for my experimental studies. I also thank my fellow labmates in SHAMAN team, for the discussions, and for all the fun we have had in the last three years.

I always remember the share of Doctor Alain SALIOU and my Vietnamese friends, who shared stress during my most difficult moments as well as the beautiful memories at

Lannion.

I would like to express my gratitude for to Excellence Scholarships of the French Embassy in Vietnam, funded by France government's scholarship, for helping me to have the financial resources to do this research.

Last but not least, I must express my very profound gratitude to my family: my parents who provide me with unfailing support and continuous encouragement throughout my life; my loving husband, Thanh Nhan NGUYEN, who has been very patient, supportive and encouraging throughout my years of study; and most of all, my little daughter, Tran An Nhien NGUYEN, who is always beside me and kept me smiling during tough times in the PhD pursuit. Thank you so much!

RÉSUMÉ EN FRANÇAIS

Motivations

Cette ère a vu une croissance massive des applications en ligne et de leurs utilisateurs, ce qui a entraîné une énorme augmentation du volume de données à traiter. En plus, il existe de nombreuses applications qui nécessitent un traitement de données volumineuses de par leur nature. Cela inclut la recherche sur les webs (par exemple, Google), les cartes en ligne (par exemple, Google Maps), les critiques de produits en ligne (par exemple, Amazon Customer Reviews), les réseaux sociaux et professionnels en ligne (par exemple, Facebook et LinkedIn), le streaming vidéo (par exemple, YouTube) et le partage de manèges et de résidences (par exemple, Uber et Airbnb), etc. Cette tendance a posé de plus grands défis pour l'extraction massive de données.

En outre, la jointure est une opération critique au sein d'un système de gestion de données, permettant d'enrichir les données d'une source avec des informations stockées en dehors de celle-ci. C'est pourquoi la littérature est riche en travail sur l'optimisation des jointures, en particulier dans les systèmes parallèles et distribués [72, 19, 73, 90, 2, 4, 55, 91]. Ces dernières années, les chercheurs se sont concentrés sur le problème des assemblages efficaces dans des environnements parallèles à grande échelle. Les premiers résultats, concernant l'équi-jointure [91], imposent de fortes contraintes sur les données (l'un des ensembles devant être suffisamment petit pour être distribué à toutes les machines utilisées pour le traitement) ou leur organisation (tri selon l'attribut de la jointure, placement des données sur des nœuds spécifiques), entraînant de nombreux transferts de données (certains inutiles) et une charge de travail importante sur les machines ou nécessitant plusieurs phases d'exécution (coûteuses). Le problème est encore plus difficile lorsque la contrainte d'égalité est libérée alors que ce type de requête est souvent nécessaire. Il est motivé par les applications demandées par des appariements similaires. Comme exemple de requête [109] dans les services de la recommandation d'amis sur les services de réseaux sociaux en ligne où les préférences de l'utilisateur sont stockées sous forme de vecteurs de bits (où un bit " 1 " signifie un intérêt dans un certain domaine), les applications veulent découvrir les intérêts similaires des utilisateurs. En d'autres termes, il doit détecter toutes

les paires de personnes qui partagent un profil similaire. Par exemple, un utilisateur avec la possibilité de vecteur de bits de préférence " $[1, 0, 0, 1, 1, 0, 1, 0, 0, 1]$ " a des intérêts similaires à ceux d'un utilisateur avec des préférences " $[1, 0, 0, 0, 1, 0, 1, 0, 0, 1]$ " avec leur distance est seulement 1. Il est facile de comprendre que deux personnes peuvent se faire des amis parce qu'elles ont les mêmes passe-temps. Un autre exemple, largement utilisé dans les moteurs de recherche basés sur le contenu d'images comme Google, Baidu, Bing, découvrent des images dont les caractéristiques mappent en code binaire par leurs similitudes sont supérieures à un seuil prédéfini. Un exemple supplémentaire est les intégrations des données pour le nom d'un auteur qui est "Thi-To-Quyen TRAN" et qui est référencé par plusieurs comptes sous les noms légèrement modifiés "Quyen Tran" ou "T.T.Quyen Tran" dans différents articles. Ces requêtes sont définies comme des jointures floues ou des jointures de similarités et jouent un rôle important dans une grande variété d'applications, y compris le nettoyage de données [27], l'intégration de données [39], la détection des attaques [84], l'exploration de sites de réseaux sociaux [100], la détection de pages Web presque en double [56], la détection de plagiat [59], le regroupement de documents [24], gestion des données maîtres [95], la bio-informatique [110].

La montée en puissance du Big Data pose des défis pour évaluations de grandes jointures floues efficaces et évolutives. Il existe deux directions principales pour résoudre ce problème qui ont été examinées dans la littérature. (1) L'un utilise des techniques d'appariement approximatives qui trouvent la plupart des résultats mais pas tous. Lorsque le nombre d'entités est très grand ou que la mesure de similarité est coûteuse à calculer, il peut être préférable en pratique d'appliquer des techniques approximatives. La solution la plus populaire dans ce cas est le hachage, qui transforme un élément en une représentation de faible dimension [112]. L'idée de base est que les éléments similaires ont une probabilité beaucoup plus élevée d'être mappés au même code de hachage que les éléments différents. Ainsi, les techniques de hachage peuvent être exploitées dans la phase de filtrage pour générer des candidats avant les calculs de jointure floue. (2) L'autre utilise des techniques de correspondance exacte qui retournent toujours la sortie correcte. Dans cette thèse, nous nous concentrons sur les jointures floues exactes.

Selon l'exécution de la jointure, un produit cartésien et le calcul de la distance de toutes les paires nécessitent pour une jointure floue. Par conséquent, son traitement est très coûteux dans la base de données traditionnelle [63, 50]. De plus, lorsqu'il s'agit d'une très grande quantité de données, la jointure floue devient un problème difficile dans un environnement informatique parallèle distribué avec le coût élevé du brassage des données,

de l'espace et de l'efficacité. En conséquence, la redondance et la duplication des données sont très difficiles à accepter.

En raison de sa simplicité, de sa tolérance aux pannes et de son évolutivité, MapReduce [35] est de loin le modèle puissant pour les applications qui manipulent intensivement les données. Il devient un cadre populaire utilisé pour l'analyse de données à grande échelle en parallèle. Il est largement appliqué à la modélisation, au traitement et au calcul des coûts dans les études de jointure floue à grande échelle [43, 99]. La plupart des solutions existantes suivent un cadre de filtrage-vérification en plusieurs étapes pour générer des candidats, appliquer les principes (filtre de préfixe, filtre de longueur, techniques de segmentation) pour éliminer les paires sans espoir, baser aux fonctions de similarité pour déterminer toutes les paires dans un rayon. Vernica et al. [109] a proposé une méthode de jointure de similarité utilisant MapReduce en 4 étapes, qui utilisait la méthode de filtrage des préfixes par index inversé sur les jetons pour prendre en charge les fonctions de similarité basées sur des ensembles. Metwally et al. [85] a proposé un algorithme de jointure VSMART en 2 étapes pour la jointure de similarité sur ensemble, multi-ensembles et vecteur. Deng et al. [36] utilise des signatures pour calculer l'index inversé et traiter en 3 étapes pour la jointure de similarité d'ensemble. [113] améliorer [36] en remplaçant les signatures par une technique q-gram en 3 étapes. Afrati et al. [5] a proposé plusieurs algorithmes pour effectuer des jointures floues avec la distance de Hamming, Edit et Jaccard en une seule étape MapReduce sans filtres. D'autres algorithmes [97, 34, 29] utilisent des pivots pour diviser les données en partitions disjointes par des travaux récursifs.

Apache Spark [117, 10] est une cadre de calcul de cluster, développé pour optimiser le calcul interactif à grande échelle. Les caractéristiques les plus importantes de Spark par rapport à MapReduce sont la prise en charge par ses calculs itératifs sur les mêmes données et sa capacité à exploiter efficacement le calcul distribué en mémoire pour un traitement rapide des données à grande échelle tout en conservant l'évolutivité et la tolérance aux pannes de MapReduce. Spark utilise des ensembles de données distribués résilients (RDD) [118] pour restaurer les ensembles de données persistants sur le disque afin de distribuer la mémoire principale et fournir une série de "transformations" et "actions" pour simplifier la programmation parallèle. Spark peut fonctionner jusqu'à 100 fois plus vite que Hadoop MapReduce et bien plus vite que les autres frameworks [117, 10]. Ainsi, nous implémentons des algorithmes de jointure floue basés sur Apache Spark.

Objectifs

En examinant ce problème, les études se concentrent sur les défis impliqués dans l'exécution efficace des jointures floues dans le paradigme MapReduce, y compris les limitations courantes en tant que relecture d'entrée par plusieurs phases, la redondance inutile et la duplication des données intermédiaires conduisant à une transmission de données coûteuse et à un grand E/S. Nous soutenons que les approches basées sur les filtres dans nos études récentes [90, 91] peuvent résoudre ces problèmes. Notre équipe était intéressée par l'utilisation des filtres Bloom [20], filtre d'intersection [90], filtre de comptage de Bloom [42]. L'idée est de filtrer les données non pertinentes dès que possible pour réduire les transferts de données et la charge de travail sur différentes machines. En outre, les jointures floues basées sur la distance de Hamming sont intéressantes pour une variété d'applications. Une application majeure est la correspondance biométrique, où une lecture biométrique est prise et mise en correspondance avec un modèle stocké [102, 62, 106, 76]. Étant donné que les lectures biométriques (par exemple, les empreintes digitales, les analyses d'iris ou les traits du visage) sont un processus bruyant, la chaîne binaire représentant les caractéristiques extraites peut ne pas correspondre exactement à la chaîne de modèle. De plus, le calcul de la distance de Hamming est montré plus rapidement que le calcul de la distance dans l'espace d'entrée. Par conséquent, nous profitons de sa théorie pour proposer de nouvelles approches de filtrage pour les jointures floues.

Contributions

Cette thèse se concentre sur l'amélioration des jointures floues à grande échelle. Les principales contributions de nos travaux sont les suivantes :

— (1) *Filtres flous et filtres flous d'intersection.*
Notre première contribution consiste en un filtre flou et un filtre flou d'intersection basée sur la définition "Boule de rayon r". Ces deux filtres flous sont de petites structures qui peuvent donner un test rapide pour détecter si un élément est similaire à des membres dans des ensembles donnés, et de plus, lesquels sont leurs similarités, avec un taux de faux positifs et sans taux de faux négatifs. Par efficacité spatiale (vecteur bit), vitesse de réponse ($O(1)$), flexibilité (facilement mis à jour), ces filtres flous peuvent être appliqués aux auto-jointures floues, aux jointures bidirectionnelles floues, aux jointures multidirectionnelles floues, au flux jointure

flou.

— (2) *Optimisations pour les auto-jointures et les jointures bidirectionnelles floues basées sur Hamming à grande échelle et l'analyse des coûts en théoriques.*
Notre deuxième contribution est l'étude des optimisations pour les algorithmes de jointure floue à une seule étape utilisant des filtres Bloom, des filtres flous, des filtres flous d'intersection. En effet, nos opérations de jointure peuvent éliminer la plupart des données redondantes avant de les envoyer au traitement de jointure réel. Ils peuvent également éviter le recalcule inutile. En conséquence, ils réduisent considérablement les frais généraux associés. De plus, l'analyse des coûts en théoriques de divers algorithmes de jointure floue est ensuite présentée pour comparer les approches dans un modèle de coût plus convaincant. Cette optimisation est une contribution extrêmement importante pour prendre en charge l'analyse évolutive des réseaux sociaux, l'analyse du trafic Internet et l'analyse des données ADN.

— (3) *Implémentations des grandes jointures floues dans Spark et évaluation expérimentale*
En exploitant pleinement les caractéristiques de calcul à la mémoire de Spark, les algorithmes de jointure floue sont capables d'obtenir de meilleures performances en mettant en cache les résultats intermédiaires et les filtres en mémoire. Nos différents algorithmes de jointures floues avec divers paramètres ont été validés avec des expériences dans le cadre Spark. Les résultats expérimentaux indiquent que les opérations de jointure utilisant nos filtres sont plus efficaces que les autres. L'efficacité est ici examinée par rapport à la quantité de données intermédiaires, la quantité totale de sortie et le temps d'exécution total. L'évaluation expérimentale nous aide à évaluer en profondeur les performances des algorithmes de jointure.

— (4) *Orientations de recherche futures.*
À partir des résultats de recherche obtenus, nous étudions et proposons des directions de recherche possibles dans le futur. Un modèle de filtre flou de différence est proposé pour optimiser les jointures floues de flux et les jointures floues récursives. De plus, une solution efficace pour optimiser l'algorithme de jointure de Vernica [109] est étudiée. Cette solution utilise les filtres de comptage de Bloom [42] pour calculer les fréquences générales des jetons et éliminer deux tâches MapReduce (l'une du tri des jetons et l'autre de l'étape de déduplication) par rapport à l'algorithme d'origine. Cela conduit à de meilleures performances avec moins d'opérations d'E/S et de communication.

Résumé, l'objectif de notre travail est de propositions pour les opérations de grande jointure floue et contributions à l'optimisation en général de la gestion de données à l'aide du paradigme MapReduce sur des infrastructures distribuées à grande échelle.

Table of Contents

Introduction **17**

Context And Motivation 17
Contributions Of The Thesis 20
Thesis Outline 21

1 Related Work **23**

1.1 Fuzzy Join Operations 23
1.2 MapReduce And Big Join Parallelism 27
1.2.1 MapReduce and HDFS 27
1.2.2 Basic Join Algorithms in MapReduce 29
1.2.3 M-C-R Cost Model 32
1.2.4 Spark 32
1.3 Fuzzy Join Algorithms In MapReduce 35
1.3.1 Single Job Fuzzy Big Join Algorithms 36
1.3.2 Multiple Jobs Fuzzy Join Algorithms 40
1.4 Filtering Techniques 46
1.4.1 Bloom Filter And Bloom Join Algorithm 46
1.4.2 Intersection Bloom Filter And Intersection Bloom Join Algorithm . 48
1.4.3 Counting Bloom Filter 49
1.5 Conclusion 50

2 Fuzzy Big Joins Improvement Using Bloom Filters **53**

2.1 Previous Works 53
2.2 Improvement Of Fuzzy Big Joins Using Bloom Filters 55
2.2.1 BF-BH Algorithm 55
2.2.2 BF-Ball-Splits Algorithm 61
2.3 Cost Analysis 64
2.4 Experimental Validation 66
2.4.1 Cluster and Datasets Descriptions 66

2.4.2 Fuzzy Self Join Evaluation . . . 66
2.4.3 Fuzzy Two-way Join Evaluation . . . 72
2.5 Summary . . . 77

3 Fuzzy Filters And Fuzzy Big Joins Optimization 79
3.1 Previous Works . . . 80
3.2 Fuzzy Filters . . . 82
3.2.1 Fuzzy Filter . . . 83
3.2.2 Intersection Fuzzy Filters . . . 86
3.2.3 Extended Intersection Fuzzy Filter . . . 91
3.3 Optimization Of Fuzzy Big Joins . . . 92
3.3.1 Fuzzy Self-Joins Using Fuzzy Filters . . . 92
3.3.2 Fuzzy Two-way Joins Using Intersection Fuzzy Filters . . . 94
3.3.3 Fuzzy Filters Analysis And Optimization . . . 97
3.4 Cost Analysis . . . 99
3.5 Experimental Validation . . . 100
3.5.1 Fuzzy Self Join Evaluation . . . 100
3.5.2 Fuzzy Two-way Join Evaluation . . . 103
3.6 Summary . . . 106

Conclusion 107
Thesis Conclusions . . . 107
Future Work . . . 109

Publications Involved in the Thesis 115

Bibliography 117

List of Figures

1.1 MapReduce Execution 28
1.2 An example of Map-side join algorithm in MapReduce 30
1.3 An example of Reduce-side join algorithm in MapReduce 31
1.4 Spark cluster architecture 33
1.5 Example of a join in Spark 34
1.6 A simple example of Naive algorithm execution with $J = 2, d = 1$ 37
1.7 BH1 algorithm execution 38
1.8 An example of BH1 algorithm execution with $b = 3, d = 1$ 39
1.9 An example of Splitting algorithm execution with $b = 4, d = 1$ 40
1.10 A workflow of set based fuzzy joins 41
1.11 A dataflow of full filtering joins 43
1.12 An example of Full Filtering Joins with $\tau = 0.6$ 43
1.13 A dataflow of Vernica joins 44
1.14 Token ordering for Vernica joins 45
1.15 Example of Job 3 for Vernica joins 46
1.16 A Bloom filter $BF(S)$ with 3 hash functions 47
1.17 $R \bowtie S$ using BF in MapReduce 48
1.18 Three approaches of IBF structure 49
1.19 $R \bowtie S$ using IBF in MapReduce 50
1.20 Counting Bloom Filter 50

2.1 An example for limitations of BH1 algorithms execution with $b = 3, d = r = 1$ 54
2.2 An example for limitations of Splitting algorithms execution with $b = 6, d = 2$ 55
2.3 An example of pre-processing stage 56
2.4 An example of join processing stage of BF-BH1 Algorithm 58
2.5 Flowchart for preprocessing stage for $BF(S)$ building in Spark 60
2.6 Flowchart for join processing stage of BF-BH1 algorithm in Spark 60
2.7 Flowchart for join processing stage of BF-BH1 two-way join algorithm . . . 61

2.8 An example for join stage processing of BF-Ball-Splits Algorithm with 6-bit strings and a threshold $d = 2$ . . . 63
2.9 Flowchart for join processing stage of BF-Ball-Splits algorithm in Spark . . 64
2.10 Exp-1,2: Running time of fuzzy self joins approaches in various thresholds . 70
2.11 Intermediate data of fuzzy self joins approaches in various thresholds . . . 71
2.12 Output result of fuzzy self joins approaches in various thresholds 73
2.13 Intermediate data of Exp-3 . . . 74
2.14 Total output results of Exp-3 . . . 74
2.15 Running time of fuzzy self join approaches in various datasets in the threshold $\tau = 3$ . . . 75
2.16 Intermediate data of Exp-4 and Exp-5 . . . 76
2.17 Total output results of Exp-4 and Exp-5 . . . 76
2.18 Running time of Exp-4 and Exp-5 . . . 77

3.1 BF-BH1 limitations . . . 80
3.2 Fuzzy filter modeling . . . 82
3.3 FF(S) structure . . . 83
3.4 Example of building FF with $b = 4, r = 1, S = (0000, 1010, 1110, 1000)$ . . 84
3.5 FF modeling with false positives . . . 86
3.6 Intersection Fuzzy Filters idea . . . 87
3.7 $IFF(S \bowtie_r T)$ . . . 88
3.8 Example of building IFF with $b = 4, r = 1, S = \{0000, 1010, 1110, 1000\}, T = \{0000, 0100, 1101\}$ . . . 90
3.9 EIFF structure . . . 92
3.10 FF-FJ Pre-processing stage . . . 93
3.11 Join processing stage of FF-FJ algorithm with $b = 4, r = 1$ . . . 94
3.12 Example of FF-FJ pre-processing stage . . . 95
3.13 Example of join processing stage of IFF-FJ algorithm . . . 96
3.14 Example of false positive of FF . . . 98
3.15 Intermediate data and output results of fuzzy self joins approaches on 10GB in various thresholds . . . 101
3.16 Exp-6: Running time of fuzzy self joins approaches on 10GB in various thresholds . . . 102
3.17 Running time of fuzzy self join approaches in various datasets in the thresold $\tau = 3$ . . . 103

3.18 Intermediate data of Exp-8 and Exp-9 104
3.19 Total output results of Exp-8 and Exp-9 105
3.20 Running time of Exp-8 and Exp-9 105
3.21 Difference Fuzzy Filter structure 110
3.22 Example for limitations of Vernica joins 112
3.23 Optimization for Vernica joins 113

List of Tables

1.1 Given Datasets 26

2.1 Notation summary 56
2.2 Summary of costs for various Hamming distance-based join algorithms . . 65
2.3 Value of expressions from Table 2.2 when $b = 20, d = 4, |S| = 10^5, K = 10^4, \delta_S = 1\%, k = 8, f_{BF(S)} = 10^{-4}$ 65
2.4 Input datasets used in experiments 66
2.5 Summary of input keys of datasets used in fuzzy self joins experiments . . 67
2.6 Parameters of filters used in experiments. m: the length of the Bloom filter, n: the number of elements being filtered, k: the number of hash functions . 68
2.7 Exp-1 - Intermediate data of fuzzy self joins of size 2GB on various thresholds 68
2.8 Exp-2 - Intermediate data of fuzzy self joins of size 10GB on various thresholds 68
2.9 Running time of the fuzzy self join approaches on 2GB in various thresholds 69
2.10 Running time of the fuzzy self join approaches on 10GB in various thresholds 71
2.11 Output results of fuzzy self join algorithms on 2GB in various threshold . . 72
2.12 Output results of fuzzy self join algorithms on 10GB in various threshold . 72
2.13 Intermediate data of fuzzy self joins in the threshold $\tau = 3$ for various datasets 73
2.14 Output results of fuzzy big join approaches in the threshold $\tau = 3$ for various datasets 74
2.15 Running times of fuzzy self joins in the threshold $\tau = 3$ on various datasets 74
2.16 Input dataset keys used in fuzzy two-way joins experiments 75

3.1 List of notations 81
3.2 Summary of costs for various Hamming distance-based join algorithms . . 99
3.3 Running time of the fuzzy self join approaches on 10GB in various thresholds 102
3.4 Running times of fuzzy self joins in the theshold $\tau = 3$ on various datasets 103

INTRODUCTION

Context And Motivation

This era has seen a massive growth of online applications and their users, which has resulted in an enormous increase in the volume of data that needs to be processed. Besides, there are numerous applications that require big data processing by their nature. This includes web-search (e.g., Google Search), online maps (e.g., Google Maps), online product reviews (e.g., Amazon Customer Reviews), online social and professional networks (e.g., Facebook and LinkedIn), video streaming (e.g., YouTube), and sharing rides and residences (e.g., Uber and Airbnb), etc. This trend has put forward greater challenges for massive data retrieval.

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 [72, 19, 73, 90, 2, 4, 55, 91]. In recent years, researchers have focused on the problem of efficient joins in large scale parallel environments. The first results, concerning the equi-join [91], 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. It is motivated by applications requested by similar matching. As a query example [109] in friends recommendation services on online social networking services 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. In other words, what it has to do is to detect all the person pairs which share a similar profile. For instance, 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 "$[1, 0, 0, 0, 1, 0, 1, 0, 0, 1]$" with their distance is only 1. It is easy to understand that two people may make friends because they have the same hobbies. Another example, widely used in image content-based search

engines as Google, Baidu, Bing, discover images whose features map into binary code by their similarities are greater than a predefined threshold. An additional instance is the integration data for an author's name that is "Thi-To-Quyen TRAN" and is referred by multiple accounts under the slightly-edited names "Quyen Tran" or "T.T.Quyen Tran" in different papers. These queries are defined as fuzzy or similarity joins, and play an important role in a large variety of applications, including data cleaning [27], data integration [39], detecting attacks from colluding attackers [84], mining in social networking sites [100], detecting near duplicate web-pages in web crawling [56], plagiarism detection [59], document clustering [24], master data management [95], bioinformatics [110].

State-of-the-art. The rise of Big Data poses challenges for efficient and scalable fuzzy big join evaluations. There are two main directions to solve this problem that have been considered in the literature. (1) One uses approximate matching techniques that find most of but not all results. When the number of entities is very large or the similarity measure is costly to compute, it may be preferable in practice to apply approximate techniques. The most popular solution in this case is hashing, which transforms an item to a low dimensional representation [112]. The basic idea is that similar items have a much higher probability to be mapped to the same hash code than dissimilar ones. Thus, hashing techniques can be exploited in the filtering phase to generate candidates before the fuzzy join computations. (2) Another one uses exact matching techniques that always return the correct output. In this thesis, we focus on exact fuzzy joins.

According to the join execution, a Cartesian product and all pairs distance calculation need for a fuzzy join. Therefore, its processing is very expensive in traditional database [63, 50]. Moreover, 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, space and efficiency. As a result, data redundancy and duplication are very difficult to accept.

Due to its simplicity, fault tolerance, and scalability, MapReduce [35] is by far the powerful model for data intensive applications. It becomes a popular framework used for large scale data analysis in parallel. It is widely applied to modeling, processing and calculating costs in large scale fuzzy join studies [43, 99]. Most existing solutions follow a filtering-verification framework in multiple stages to generate candidates, apply the principles (prefix filter, length filter, segmentation techniques) to prune out hopeless pairs, based on similarity functions to determine all pairs within a radius. Vernica et al. [109] proposed a similarity join method using 4-stage MapReduce, which utilized the prefix

filtering method by inverted index on tokens to support set-based similarity functions. Metwally et al. [85] proposed a 2-stage algorithm VSMART join for similarity join on set, multisets and vector. Deng et al. [36] use signatures to calculate inverted index and process in 3-stage for set similarity join. [113] improve [36] by replacing signatures with a q-gram technique in 3-stage. Afrati et al. [5] proposed multiple algorithms to perform fuzzy joins with Hamming, Edit and Jaccard distance in a single MapReduce stage without filters. Others algorithms [97, 34, 29] use pivots to split data into disjoint partitions by recursive jobs.

Apache Spark [117, 10] is an open source cluster computing framework, developed to optimize large-scale interactive computation. The most significant characteristics of Spark in comparison with MapReduce are Sparks' support of iterative computations on the same data and its capability of efficient leveraging of distributed in-memory computation for fast large-scale data processing while it retains the scalability, fault tolerance of MapReduce. Spark uses Resilient Distributed Datasets (RDDs) [118] to restore the persistent datasets on disk in order to distribute main memory and provide a series of "transformations" and "actions" to simplify parallel programming. Spark can perform up to 100 times faster than Hadoop MapReduce and significantly faster than other frameworks [117, 10]. Thus, we implement fuzzy join algorithms based on Apache Spark.

Motivation. In addressing this problem, the studies focus on the challenges involved in efficiently performing fuzzy joins in the MapReduce paradigm including the common limitations as input re-reading by multiple phases, wasteful redundancy and duplication of intermediate data leading to expensive data transmission and large disk I/Os. We argue that the filter-based approaches in our recent studies [90, 91] can solve these problems. Our team was interested in using Bloom Filters [20], Intersection Filter [90], Counting Bloom Filter [42]. The idea is to filter irrelevant data as soon as possible to reduce data transfers and workload on different machines. Besides that, Hamming distance based fuzzy joins are interesting for a variety of applications. One major application is biometric matching, where a biometric reading is taken and matched with a stored template [102, 62, 106, 76]. Since biometric readings (e.g., fingerprints, iris scans, or facial features) are a noisy process, the binary string representing the extracted features may not be an exact match for the template string. In addition, 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 new filter approaches for fuzzy joins.

Contributions Of The Thesis

This thesis focuses on the improvement for large scale fuzzy joins. The main contributions of our works are the following:

— (1) *Fuzzy Filters and Intersection Fuzzy Filters.*
Our first contribution consists of a Fuzzy Filter and an Intersection Fuzzy Filter based on the "Ball of radius r" definition. These two fuzzy filters are small structures that can give a quick test for detecting if an element is similar to any members in given sets, and moreover, which ones are their similarities, with a false positive rate and without a false negative rate. By space efficiency (bit vector), response speed ($O(1)$), flexibility (easily updated), these fuzzy filters can be applied to fuzzy self joins, fuzzy two-way joins, fuzzy multi-way joins, fuzzy stream joins.

— (2) *Optimizations for large scale Hamming based fuzzy self joins and fuzzy two ways joins and theoretical cost analysis.*
Our second contribution is the study of optimizations for single stage fuzzy join algorithms using Bloom Filters, Fuzzy Filters, Intersection Fuzzy Filters. This is because our join operations can eliminate most redundant data before sending them to the actual join processing. They also can avoid the useless re-computation. As a consequence, they significantly reduce the associated overheads. Moreover, theoretical cost analysis of various fuzzy join algorithms is then presented to compare among the approaches in a map-reduce-communication cost model more convincing. This optimization is an extremely important contribution to support scalable social network analysis, internet traffic analysis, DNA data analysis.

— (3) *Fuzzy big join implementations in Spark and experimental evaluation.*
By making full use of the in-memory computation characteristics of Spark, fuzzy join algorithms are able to achieve better performance by caching the intermediate results and the filters in memory. Our different fuzzy joins algorithms with various parameters have been validated with experiments in Spark framework. Experimental results indicate that join operations using our filters are more efficient than the others. The efficiency here is examined with respect to the intermediate data amount, the total output amount, and the total execution time. Experimental evaluation helps us thoroughly evaluate the performance of the join algorithms.

— (4) *Future research directions.*
From the research results achieved, we study and propose possible research direc-

tions in the future. A Difference fuzzy filter model is proposed to optimize fuzzy stream joins and fuzzy recursive joins. In additional, an efficient solution for optimizing the Vernica join algorithm [109] is studied. This solution uses the Counting Bloom Filters [42] to compute the general token frequencies and eliminate two MapReduce jobs (one of tokens sorting and one of deduplication stage) versus the original algorithm. This leads to a better performance with less I/O operation and the communication.

Thesis Outline

In Chapter 1 - Related work, we present the definitions that help us gain a better understanding of the basic characteristics and features of important fuzzy joins. It then covers specific fuzzy join operations that are used in this thesis such as fuzzy self joins, fuzzy two-way joins, fuzzy multi-way joins and fuzzy recursive joins. Besides, we also summarize background components that are the essentials of the MapReduce framework, parallelization of the join operation in MapReduce, M-C-R cost model to compare the costs of different algorithms, Apache Spark and its supports for fuzzy joins. Basic concepts, terminologies, and characteristics of the Bloom filter family are described. Notably, we present the classification and details of fuzzy join algorithms in MapReduce. We specifically analyze the advantages and disadvantages of each method to point out their limitations related to our proposals.

In Chapter 2 - Fuzzy Big Joins Improvement Using Bloom Filters, we present our first two contributions, including the optimizations for Ball Hashing fuzzy joins and Splitting fuzzy joins. They are implemented for fuzzy self joins and fuzzy two-way joins. We first show existing problems of the previous works that need to be addressed. We then present our proposal to improve these limitations. In addition, we analyze fuzzy join approaches based on the cost model and make comparisons of the different fuzzy join algorithms. At the end of the chapter, we present experiments on the performance of the fuzzy big joins on Spark and compare our proposal to previous methods mentioned in the literature.

In Chapter 3 - Fuzzy Filters And Fuzzy Big Joins Optimization, we present our contributions, including the Fuzzy Filter, Intersection Fuzzy Filter and optimizations for fuzzy self joins and fuzzy multi-way joins. We describe approaches to building the Fuzzy Filters and their false probabilities. We then use the Fuzzy Filters to optimize fuzzy self joins and multi-way joins. In addition, we analyze and make comparisons of the different algorithms

based on the cost model. At the end of the chapter, we also present experiments on the performance of the fuzzy big joins on Spark and compare our approaches that use the Fuzzy Filters to others that use the Bloom Filters.

At Chapter Conclusion, we present the conclusions of the thesis. Besides, we discuss open challenges and perspectives in optimization for join operations.

Chapter 1

Related Work

Fuzzy joins (also referred to as similarity joins) have been widely studied in the research community and extensively used in real world applications. A fuzzy join aims to group information based on their similarity. It relies on a distance measure to find all pairs of tuples (x, y) in the input dataset(s) with a distance below some pre-specified threshold τ.

MapReduce has emerged as a popular large-scale data processing model because of its attractive programming interface with the abstraction of parallelism, scalability, and reliability. Basic relational operators such as selection, projection, group and aggregation can be implemented easily and efficiently in MapReduce. In contrast, theta-join, equi-join, multi-way joins and fuzzy join operations are much more difficult and expensive [75]. Significant efforts have been made to develop efficient join algorithms in recent years. Thus, fuzzy big join processing in MapReduce became an interesting challenge for researchers.

In this chapter, we first present the foundations of fuzzy join queries. In the second part of the chapter, we summarize the fundamentals of MapReduce to describe parallelization of the fuzzy join operation. In addition, we enumerate the various parameters used to analyze fuzzy join algorithms. We also present a distributed in-memory computing engine, namely Spark to support our optimizations. The third part presents the classification and details of recent approaches to improving the fuzzy join computation in MapReduce. We then review some basic concepts, terminologies, characteristics of the filters as well as filters based join algorithms in the fourth part. They are used as optimization techniques for fuzzy big joins in our approaches. Finally, we conclude all elements help us make better optimizations for the fuzzy big joins in this research.

1.1 Fuzzy Join Operations

The join operation [30, 31] is a fundamental operation and has been studied widely in the database literature because it is a time consuming and data-intensive operation in data processing. Based on a Cartesian product of relations, it combines related tuples

from relation(s) according to a condition on different attribute schemes to form a new relation with columns selected from across the multiple relations. The equi-join is a join where the join condition uses an equality operator (=) to relate the tuples of two datasets. Self joins, two-way joins, multi-way joins, recursive joins, etc., are instances of the equi-join. An extended, more complex operation of the equi-join is the fuzzy join that the join condition is specified flexible by a similarity function. We start by formally defining the problem of this research.

Definition 1.1 (Similarity/Distance function)**.** *Let $\mathcal{D}$ be the domain of all possible tuples. A similarity/distance function is a function $Sim : \mathcal{D} \times \mathcal{D} \to \mathbb{R}$*

The distance function gives us a dissimilarity criterion to compare objects. Thus, the smaller the distance between two objects, the more similar they are. Fuzzy joins have been studied for a wide variety of entity types and representations, including strings (e.g., entity names), sets (e.g., tokens extracted from entity profiles), binary vectors (e.g., feature vectors extracted from image or audio objects), graphs (e.g., biological structures), data series (e.g., time series, trajectories), etc [12]. Based on the entity type and representation, a wide variety of similarity and distance functions can be used, such as string edit distance, set overlap, Hamming distance, graph edit distance, Euclidean distance, etc. Furthermore, the similarity/distance threshold depends on the data characteristics and the application requirements, and is typically assumed to be provided as input to the fuzzy join task. Our works focus on Hamming distance and Jaccard similarity for fuzzy join evaluation.

Hamming distance (HD) is introduced in [54]. The Hamming distance function is well-defined on bit strings, binary vectors and naturally extends to more general strings, vectors, multi-field records, etc. Hamming distance between two strings s, t is the number of positions in which they differ. It has been used in some disciplines like information theory, cryptography, coding theory, and etc.

Jaccard similarity of sets is the ratio of the size of the intersection of the sets to the size of the union. The more the two sets have in common, the closer the Jaccard similarity will be to 1.0.

Definition 1.2 (Fuzzy/Similarity join)**.** *A fuzzy-join predicate $F = (Sim, \tau)$ is defined by a similarity function Sim and a threshold τ. The result of applying F to a set of tuples $\mathcal{S} \subseteq \mathcal{D}$ is $F(R) = \{(x, y) \mid x, y \in R, Sim(x, y) \geq \tau\}$. For $(x, y) \in F(R)$ we say $F(x, y) = 1$.*

Thus a fuzzy join is stated using a distance measure used to define the similarity, where we are required to find all pairs (x, y) with a distance of at most some pre-specified threshold. Our works aim to find distributed parallel algorithms that can efficiently return all the $(x, y) \in F(R)$

Most of the existing work has concentrated on the **fuzzy two-way joins**: Given two finite datasets S and T, a fuzzy two-way join is defined as a combination of tuples $s \in S$ and $t \in T$, such that $d(s.x, t.y) \leq \tau$; where x and y are columns in S and T respectively, d is a distance function. This specification is represented as:

$$S \bowtie_\tau T = \{(x, y) \mid x \in S, y \in T, d(x, y) \leq \tau\}$$

A **fuzzy self join** is a specified fuzzy join case in which a dataset is joined with itself. Fuzzy self joins are often applied for data cleaning tasks.

$$S \bowtie_\tau S = \{(x, y) \mid x, y \in S, d(x, y) \leq \tau\}$$

Various fuzzy joins can be extended from the idea for the multi-way joins [103, 61] and the recursive joins [88], referred as the fuzzy multi-way joins and the fuzzy recursive joins.

— **Fuzzy multi-way join** is defined as a composition of multiple fuzzy two-way joins, noted as:

$$S_1 \bowtie_{\tau_1} S_2 \bowtie_{\tau_2} S_3 \bowtie_{\tau_3} S_4 ... \bowtie_{\tau_{n-1}} S_n$$

— **Fuzzy recursive join**: Given a dataset K, a fuzzy recursive join is defined as the transitive closure of the dataset K:

$$F(x, y) = K(x, y) \cup F(x, z) \bowtie_\tau K(z, y)$$

For clarity, we illustrate with the following example queries relating to researchers, their papers and their affiliations.

Example. Given three datasets $\mathcal{S}, \mathcal{T}, \mathcal{U}$ shown in Table 1.1.

We have the kinds of fuzzy join queries expressed as follows.

— Q_1**-Fuzzy self join**: Explore the authors who have similar research topics in Dataset $\mathcal{S}$ for a recommendation application.

$$Q_1 = S \bowtie_{S.R \approx S.R} S$$

Table 1.1 – Given Datasets

Dataset $\mathcal{S}$

Profile (ID)	**Author (A)**	**Research topics (R)**
s_0	Thi-To-Quyen Tran	001001010
s_1	Anne Laurent	001011101
s_2	Thuong-Cang Phan	011001100
s_3	Van-Hoang Tran	101001010
s_4	Foto N. Afrati	101110110
s_5	Jeffrey D. Ullman	101011101
s_6	Thanh-Nghi Do	101101010
s_7	Laurent D'Orazio	111001100

Dataset $\mathcal{U}$

Affiliation (AF)	**Country (C)**
Rennes University	France
Blaise Pascal University	France
Can Tho University	Vietnam
Stanford University	America
National Technical University of Athens	Greece

Dataset $\mathcal{T}$

Paper (P)	**Author (A)**	**Affiliation (AF)**
p1	Laurent D'Orazio	Univ Rennes,CNRS,IRISA,Lannion,France
p1	Thi-To-Quyen Tran	Univ Rennes,CNRS,IRISA,Lannion,France
p1	Thuong-Cang Phan	Can Tho University,Can Tho,Vietnam
p1	Anne Laurent	Univ Montpellier, LIRMM, CNRS,Montpellier,France
p2	TC. Phan	Univ Blaise Pascal
p3	Jeffrey Ullman	Stanford University
p3	Foto N. Afrati	National Technical University of Athens, Greece
p4	Jeffrey D. Ullman	Stanford Univ
p5	L. DOrazio	Rennes University
p6	F. Afrati	National Technical University, Greece
p7	Foto Afrati	National Technical University of Athens
p8	A. Laurent	LIRMM University of Montpellier CNRS
p9	Quyen T.T Tran	Can Tho University
p9	Cang T. Phan	Can Tho University

— Q_2**-Fuzzy two-way join**: Merge authors who have similar names in Dataset $\mathcal{S}$ for a record linkage application.

$$Q_2 = S \bowtie_{S.A \approx T.A} T$$

— Q_3**-Fuzzy three-way join**: Statistics of research topics by region.

$$Q_3 = S \bowtie_{S.A \approx T.A} T \bowtie_{T.AF \approx U.AF} U$$

Definition 1.3 (Hamming distance based fuzzy join)**.** *Given a set $\mathcal{S}$ of b-bit strings, and an integer $0 \leq d \leq b$, a Hamming distance based fuzzy join is $\{(s_1, s_2) \mid \mathrm{HD}(s_1, s_2) \leq d\}$*

To illustrate this concept, let us consider an example, 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. In other words, what it has to do is to detect all the person pairs which share a similar profile. For instance, 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 "$[1, 0, 0, 0, 1, 0, 1, 0, 0, 1]$" with their distance is only 1.

Consider another example of the Hamming distance based fuzzy join for the dataset S in Table 1.1. By the Hamming distance threshold $d = 1$, the output of this fuzzy self join is $\{(s_0, s_3), (s_1, s_5), (s_2, t_7), (s_3, s_6)\}$

Definition 1.4 (Ball of radius r)**.** *The Ball of radius r (B_r) contains all close elements of any given element within a distance r.*

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

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

This definition is applied to compute fuzzy joins.

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 [5] [1]:

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

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

— $r = 0$, the ball of radius 0 around any given element $B_0(000)$ now is itself ($\{000\}$).

— $r = 1$, the $B_1(000)$ now has $1+3 = 4$ elements. $B_1(000) = B_0(000) \cup \{001, 010, 100\}$.

— $r = 2$, the $B_2(000)$ now has $1+3+3 = 7$ elements. $B_2(000) = B_0(000) \cup B_1(000) \cup \{011, 101, 110\}$.

1.2 MapReduce And Big Join Parallelism

1.2.1 MapReduce and HDFS

MapReduce [35] 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 1.1 illustrates MapReduce execution.

In this model, a MapReduce program is executed on multiple nodes. 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.

1. [5] assumes that d is much smaller than b

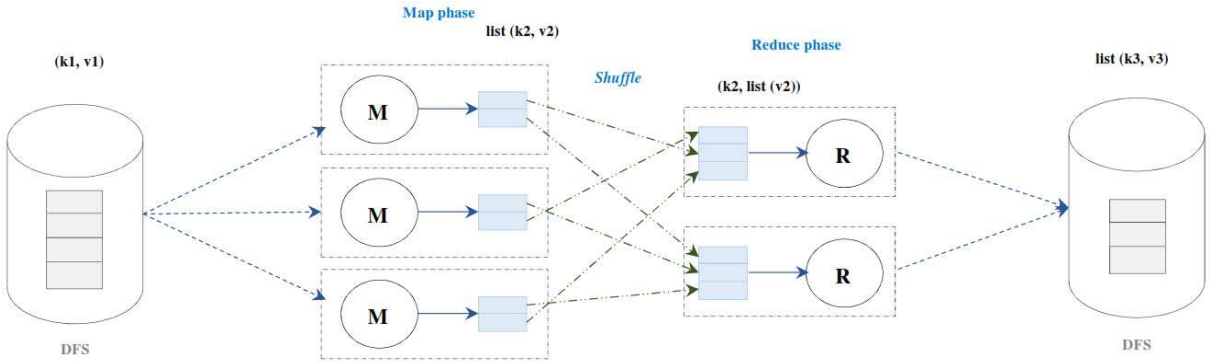


Figure 1.1 – MapReduce Execution

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. As illustrated in Figure 1.1. The Map function **(M)** takes a pair of entries $(k1, v1)$ from a Distributed File System (DFS) and emits a list of intermediate pairs $(k2, v2)$.

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

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. It starts with Map function calls emitting data to an in-memory buffer. Once the buffer fills up, its content is partitioned by Reduce task (using the Partitioner) and the data for each task is sorted by key (using a key comparator than can be provided by the programmer). The partitioned and sorted buffer content is written to a spill file in the local file system. At some point spill files are merged and the partitions are copied to the corresponding Reducers. It constitutes a synchronization barrier between the Map and Reduce phase in the sense that no Reduce function call is executed until all Map output has been transferred. Therefore, shuffle phase sometimes takes time, network bandwidth, and other resources more than two main functions, Map and Reduce [40, 19]. In other words, the shuffle phase can be the most expensive part of a MapReduce execution.

Each reduce task collects the intermediate key/value pairs from all the map tasks, sorts/merges the data with the same key, and then calls the reduce function to process the value list and generate the final results. These results are then written back to DFS. As illustrated in Figure 1.1, the intermediate values associated with the same key $k2$ are grouped together and then transmitted to the Reduce function **(R)** which aggregates the values.

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

Hadoop [8] is an open source implementation of MapReduce. Hadoop is a compound of two parts: a data storage component called Hadoop Distributed File System (HDFS) and a data processing component called Hadoop MapReduce Framework to achieve distributed processing.

HDFS is a fault-tolerant distributed file system. It is designed to recognize and respond to individual machine failures. It divides files into blocks, replicates them, and stores them across the cluster. HDFS provides high throughput access to application data in a distributed environment [8, 60, 78]. To support this characteristic, HDFS leverages unusually large (for a file system) block sizes and data locality optimizations to reduce network input/output (I/O) [60]. It is therefore suitable for applications handling large datasets. On the other side, random access to file parts is essentially costly in comparison with sequential access since HDFS is optimized for streaming access of large files. Files are possible to be only appended; there is no file update support [40, 19].

Hadoop MapReduce is the processing component that distributes the workload for operations on files stored in HDFS and automatically restarts failed work.

In our experiments, we use HDFS as a distributed storage of large data.

1.2.2 Basic Join Algorithms in MapReduce

Most join algorithms in MapReduce are derived from the literature on join processing in parallel RDBMSs [37, 18, 47, 13, 14] and distributed RDBMSs [17, 14, 82] such as sort-merge join, repartition join, hash join, semi-join, Bloomjoin, etc. However, they are not always straightforward to implement within MapReduce environment because MapReduce is originally designed to read a single input. Based on where the join processing takes place in a MapReduce phase, we can show two main classifications of the join operation including Map-side join and Reduce-side join [19, 91]. In this section, therefore, we describe their implementation details and discuss the difference between these two important join algorithms. The problem of skewed data processing in the join operation is outside of our research scope.

Map-side join

Map-side join [72, 19] performs join operations in the map side without a shuffle and reduce phase. This algorithm, however requires under certain strict conditions on input

datasets.

— Each input dataset must be divided into the same number of partitions, be sorted by the join key, and has the same set of the keys.

— All the tuples associated with the same key must reside in the same partition in each dataset.

When a join job satisfies all the mentioned requirements for two input datasets, map tasks are initiated and each map task retrieves two partitions, one from each dataset. The join computation is conducted by the map task before the data reaches the map function and then the result can be directly written to DFS using the map function. We illustrate the Map-side join algorithm in an example as Figure 1.2. The two datasets S and T have the same k partitions ($k = 3$) with k join keys sorted. Thus, the join job uses k mappers to process the partitions. The two partitions with the join key 1 from the two datasets are read by the same first mapper. The first mapper builds the cross product of all tuples with the same join key 1. Then, the results are directly written to DFS.

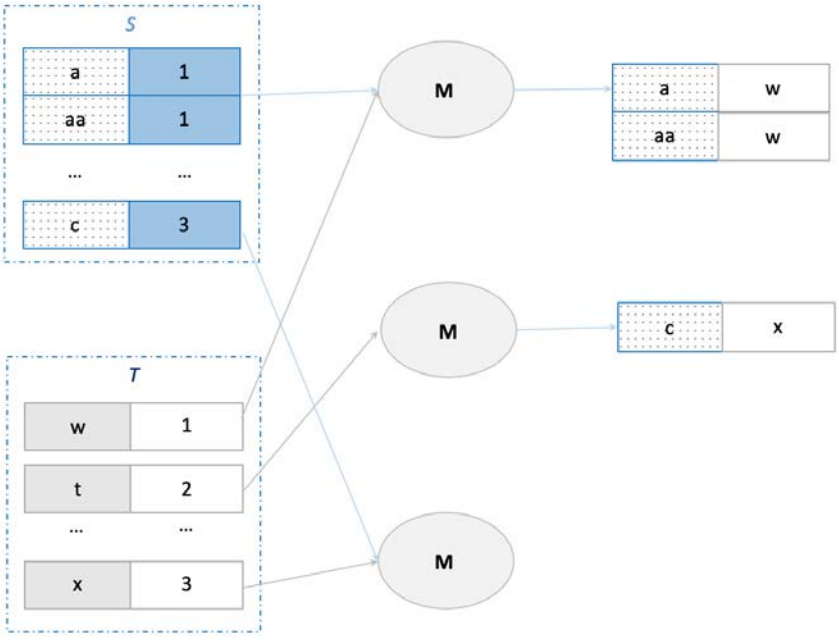


Figure 1.2 – An example of Map-side join algorithm in MapReduce

The algorithm does not create intermediate data as well as has no the cost incurred for the shuffle and reduce phases because it only scans the input datasets and performs the join computation in the map phase. Consequently, it is the most efficient join algorithm if its

input datasets meet all the mentioned conditions. However, this join algorithm rigorous requirements on the input datasets. For arbitrary input datasets, therefore, the problem can be solved by passing the input datasets through additional MapReduce jobs as a pre-processing step that sorts and partitions the datasets in the same way. However, that also means that this algorithm must take additional costs for the jobs of the pre-processing step related to generating a large volume of intermediate data, shuffling them to the reducers and performing local and remote I/O operations. In addition, this algorithm bufferres both the two joined partitions that can lead to a memory overflow for the compute node when the size of the two joined partitions is larger than the size of physical memory allocated for the mapper or the case of skewed datasets.

Reduce-side join

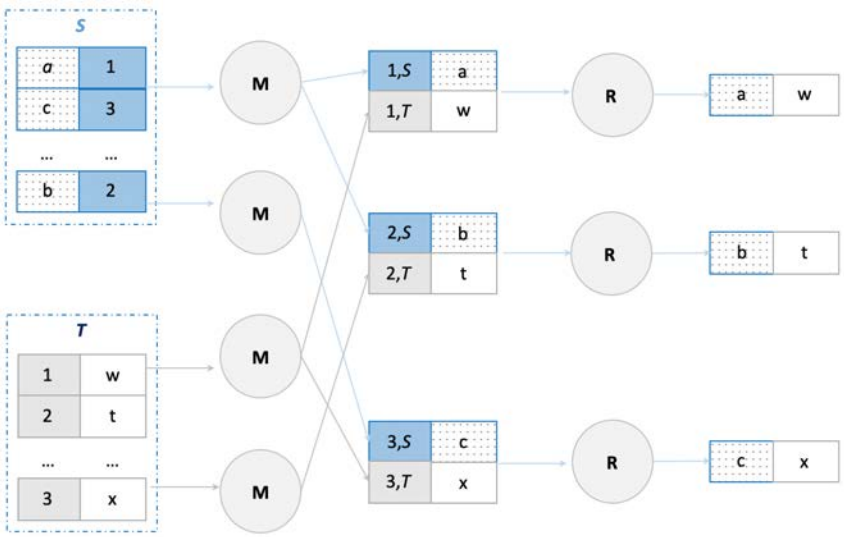


Figure 1.3 – An example of Reduce-side join algorithm in MapReduce

Reduce-side join [114, 3, 19, 91] is a popular join algorithm because it is based on the nature of the MapReduce framework. As implied by its name, the actual join computation is only conducted on the reduce side.

— First, the map phase only pre-processes the tuples of input datasets to organize them in terms of the join key. However, to identify which dataset each tuple belongs to, it is marked by an input tag. Mappers partition all tuples of input datasets according to the join key into intermediate ((key,tag)/value) pairs.

— Then, the immediate pairs are shuffled (repartitioned) to the corresponding reducers to compute the join. All the pairs with the same join key must be sent to the same reducer and sorted by the (join key,tag).
— When all the mappers are complete, the reducers call a reduce function for each the join key to compute its Cartesian products on its values. The output of the reduce function can be directly written to DFS. The Reduce-side join algorithm is depicted as Figure 1.3.

Based on the basic operation in MapReduce, Reduce-side join is the most general type of join algorithms without any constraints on input datasets. However, because the join computation is only conducted on the reduce side, the entire input datasets must be sent across the network from the mappers to the reducers. Besides that, the skewness problem of the input datasets affects also the performance of this join algorithm.

1.2.3 M-C-R Cost Model

In order to compare the costs of different algorithms, this thesis adapts a cost model $(M - C - R)$ [5], where M, R, C are used to measure the effectiveness of an algorithm:

— Total map or preprocessing cost across all input records (M).
— Total communication cost (C) of passing data from the mappers to the reducers. It represents the total amount of network resources needed for the computation. We assume that other operations such as copying, comparing, hashing are performed at a unit cost. The size of the intermediate data D is critical since it often correlates with the I/O cost which dominates the overall execution time.
— Total computation cost of all reducers (R).

In this model, the number of mappers is never considered. They assume that the algorithm uses as many mappers as is appropriate to handle the input. Since the mappers typically operate on one input element at a time, the total map cost is not really affected by the number of mappers, although if we use too few, then the finishing time of the entire job will be unnecessarily high. They do not consider possible communication between mappers to deal with data skew, since it is not in the basic MapReduce model.

1.2.4 Spark

Since Spark [117, 10] is one of the widely adopted distributed in-memory computing system, we have developed our optimization to support fuzzy join technique in Spark.

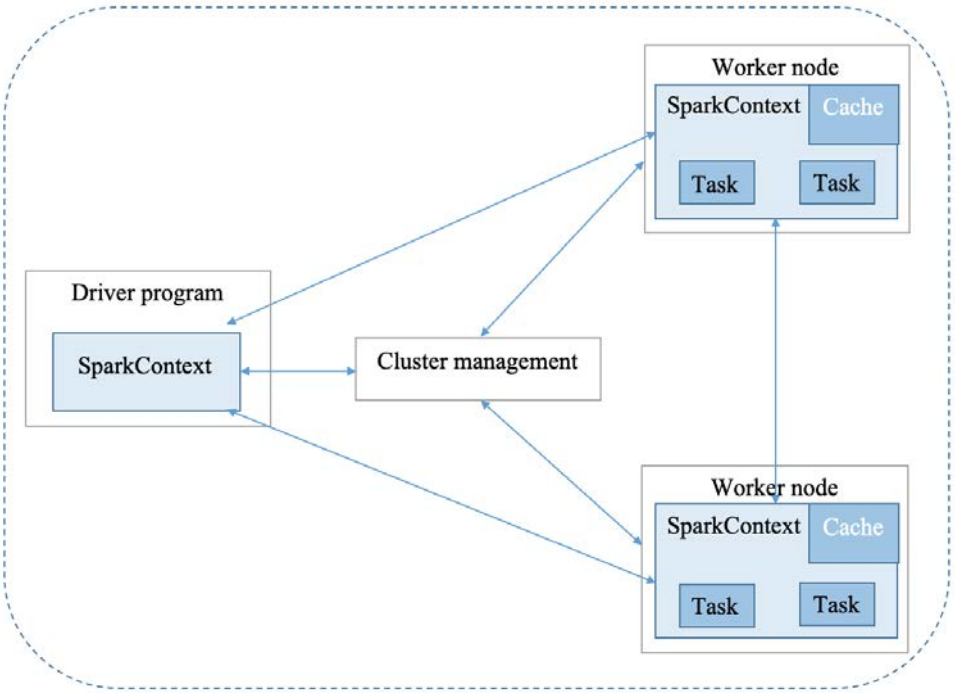


Figure 1.4 – Spark cluster architecture

Spark is a fault-tolerant, distributed in-memory computing engine. Spark can run over a variety of cluster managers, a simple cluster manager included in Spark itself known as Standalone Scheduler, on Hadoop YARN [108], or on Apache Mesos [57]. For distributed data storage, Spark is compatible with many file storage systems such as HDFS [114], Cassandra [71, 7], HBase [9] and Amazon S3 [6]. Spark has a processing speed 100 times faster than Hadoop MapReduce when cached in memory, or 10 times faster if cached on disk [117, 10]. Spark provides advanced APIs in Java, Scala, Python and R, and an optimized engine that supports general execution graphs. It also supports some advanced components, including Spark SQL for structured data processing, MLlib for machine learning, GraphX for computing graphs, and Spark Streaming for stream computing.

Spark cluster architecture. As shown in Figure 1.4, each Spark application runs as an independent process on the cluster, coordinated by SparkContext in the driver program. First, SparkContext requests the executors on the worker nodes in the cluster from the cluster manager. These executors are processes that can run tasks and store data in memory, or on disk for application. Next, SparkContext will send tasks to the executors to perform. Finally, the executors return the results to SparkContext after the tasks are executed. In Spark, an application generates multiple jobs. A job is split into several

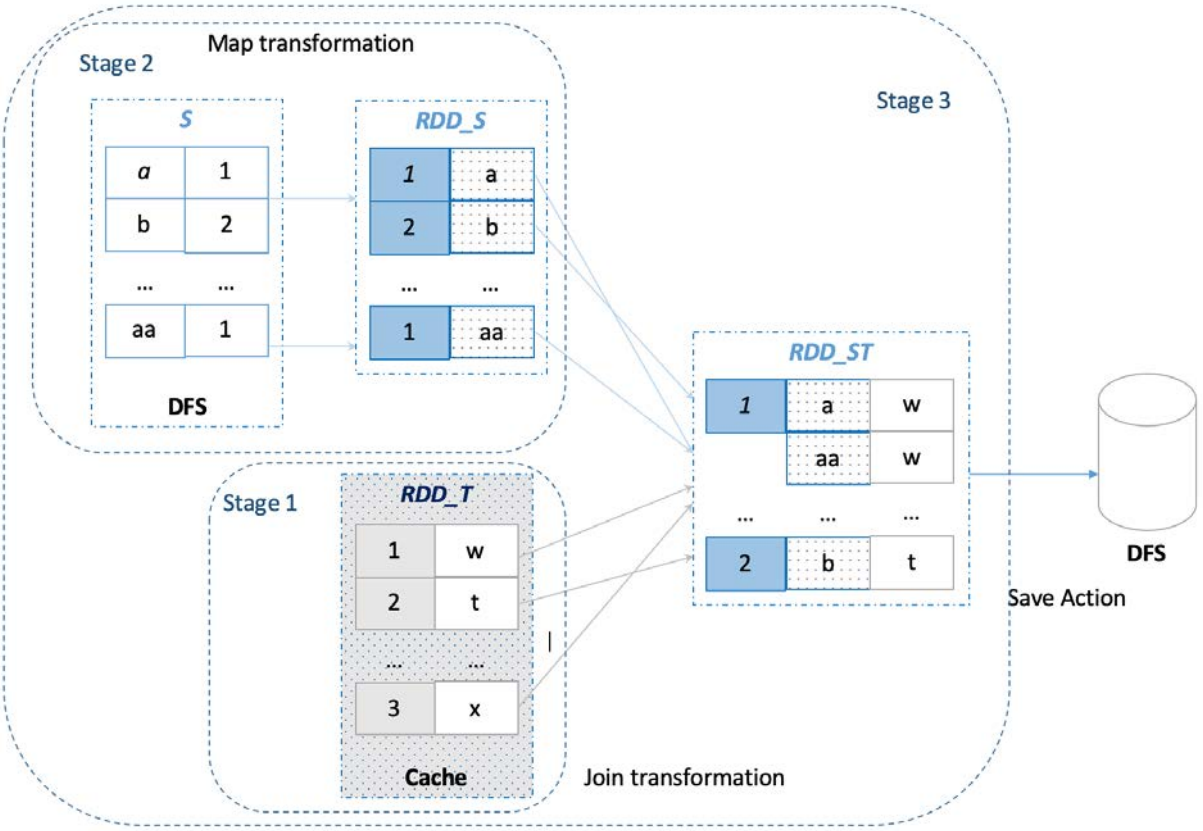


Figure 1.5 – Example of a join in Spark

stages. Each stage is a task set containing several tasks, which performs calculations and produces intermediate results. A task is the smallest unit of work in Spark, completing a specific job on an executor.

Spark's Serial Job Processing Capacity. Spark is a powerful tool to support serial job processing of large-scale datasets using RDD (Resilient Distributed Datasets). RDD is a fault-tolerant parallel data structure that can be cached in memory and used again for future transformations. RDDs support two types of operations: transformations and actions. Transformations are "Lazy" operations, meaning that these operations will not be performed immediately, but only remember their execution steps. Transformations are only performed when an action is called. Basically, the evaluation of RDD is lazy in nature. It means a series of transformations on an RDD is being delayed until it is really needed. This saves a lot of time and improves efficiency.

The serial job processing of Hadoop MapReduce is carried out as a sequence of operations in which the intermediate results must be written to DFS then being read as input

to the following tasks. Meanwhile, Spark will read data from DFS, perform repeated operations with RDDs, and finally write to DFS.

Spark's Cache Mechanism. Due to the fact that RDD will be recalculated after an Action, it is costly and time consuming if a RDD is being reused many times. Therefore, Spark provides a mechanism called persist, allowing RDDs to be reused efficiently. Indeed, the caching mechanism is an optimization technology for (iterative and interactive) Spark computations. It helps save interim partial results so it can be reused in subsequent stages. These interim results as RDDs are thus kept in memory (default) or more solid storages like disk and/or replicated.

Figure 1.5 shows an example of a join computation $S \bowtie T$ in Spark. RDD_T was calculated and was already cached in memory. So, the join job runs stage 2, and then stage 3, and finally, writes results to DFS.

1.3 Fuzzy Join Algorithms In MapReduce

A brute force algorithm of fuzzy join computing for a dataset S is to compare every pair of objects, thus bearing a prohibitively $O(|S|^2)$ time complexity. Most of the existing algorithms for fuzzy join computing are in-memory and central approaches [27, 11, 16, 49, 79, 96, 115].

In large scale data applications like social networks, computational biology, this query may perform the join of trillions of tuples. Therefore, the parallel model like MapReduce is a good solution to this problem and significantly improves the response time. So it is widely applied to modeling, processing and calculating costs in large scale fuzzy join studies [43, 99]. We classify fuzzy big join algorithms into two groups based on the number of MapReduce processing jobs.

— Multiple stage solutions: The most popular of existing solutions [41, 85, 109, 93, 94, 15, 36] follow a filtering-verification framework in multiple stage to generate candidates, apply the principles (prefix filter, length filter, segmentation techniques) to prune out hopeless pairs, based on similarity functions to determine all pairs within a radius. Besides, the metric partitioning based approaches [97, 34] use pivots to split datasets into sub-clusters respectively.

— Single stage solutions: [5] proposed the algorithms using only one MapReduce job to fuzzy big join computing.

In this section, we present the parallel approaches for fuzzy join in MapReduce related to this study.

1.3.1 Single Job Fuzzy Big Join Algorithms

[5] studies on character based fuzzy join approaches for fixed length string using Hamming distance. Beside that, they also propose their extensions for variable length strings using Edit distance and Jaccard distance. Their algorithms process in a single MapReduce job.

Naive Algorithm

Naive algorithm [5] 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 $0 \leq i \leq j < J$
Example: Consider a naive fuzzy join with $J = 4, K = 10$, the reducer labels are specified as follows:

$$\begin{array}{cccc} (0,0) & (1,1) & (2,2) & (3,3) \\ (0,1) & (1,2) & (2,3) & \\ (0,2) & (1,3) & & \\ (0,3) & & & \end{array}$$

— 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 $M = C = O(|S|J) = O(|S|\sqrt{K})$
In the example above, with the hashed values $j = 2$, the tuples are sent to $(0, 2), (1, 2), (2, 2), (2, 3)$
Besides, with the hashed values $j = 3$, the tuples are sent to $(0, 3), (1, 3), (2, 3), (3, 3)$
Therefore, these tuples whose hashed values is 2 or 3 will be mapped to at least one common reducer $(2, 3)$ for computing their distance.

— 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 τ. 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 pair of elements of S to be sent to exactly one reducer and thus avoid data duplication. Each input record must be compared with all others leading to data redundancy and inefficiency. The duplicated pairs exist in the result. These limitations are illustrated with a simple example in Figure 1.6

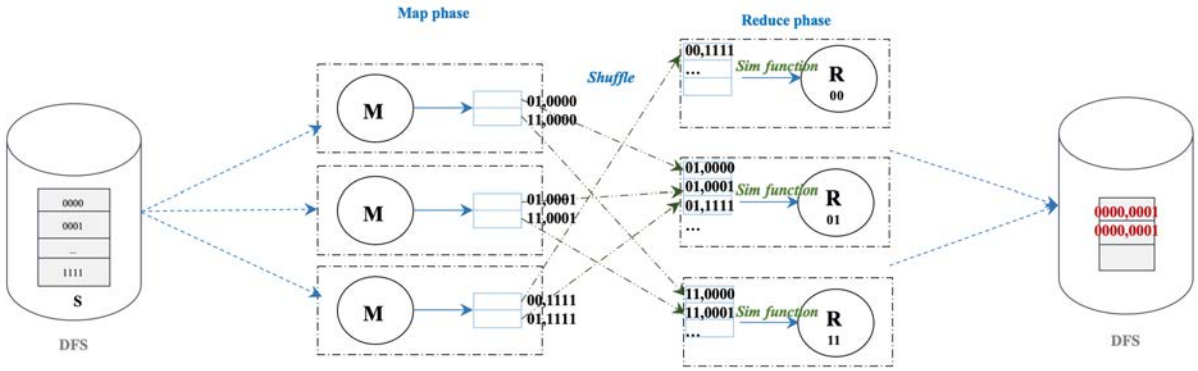


Figure 1.6 – A simple example of Naive algorithm execution with $J = 2, d = 1$

Ball Hashing Algorithms

Ball Hashing [5] is a family of two algorithms BH_1 and BH_2. These algorithms rely on the "ball of radius r" ($d = r$) 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 $n = 2^b$.

BH_1

— The mappers generate all elements t in the ball of radius r of each input record s ($B_r(s)$) 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.

$$s \xrightarrow{map} \begin{cases} (s, -1) \\ (t, s), \quad \forall t \in B_r(s), t \neq s \end{cases}$$

Thus the map cost is B_r per input element.

Example: The tuples 000 of $B_1(000)$ is mapped as follows:

$$000 \xrightarrow{map} \begin{cases} (000, -1) \\ (001, 000), (010, 000), (100, 000) \end{cases}$$

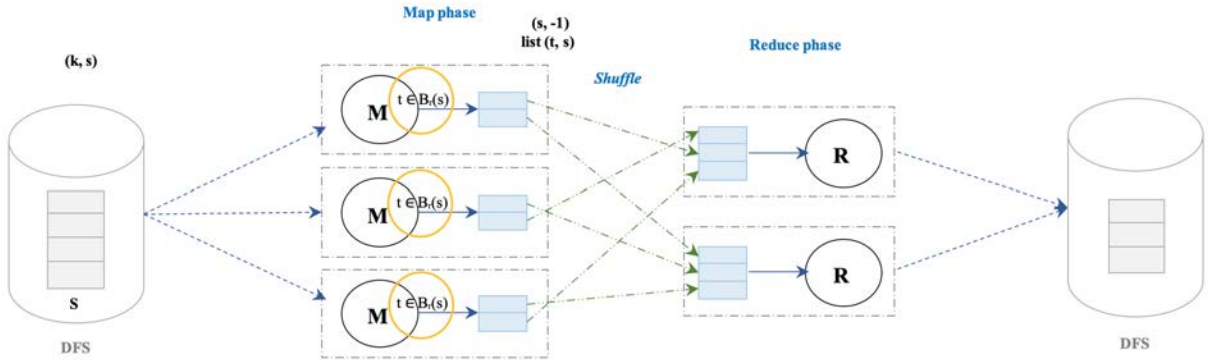


Figure 1.7 – BH1 algorithm execution

Figure 1.7 illustrate BH1 algorithm execution.

— Call a reducer that receives only pairs $(s, -1)$ or does not receive a pair $(s, -1)$ *"inactive"*. On the contrary, remaining reducers that do not only receive pairs $(s, -1)$ but also receive pairs (t, s) are considered *"active"* reducers. It infers that t is really in the input dataset 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 $|S||B_r|/n$. The total cost of all $|S|$ "active" reducers is $|S|^2|B_r|/n$.

— An issue with BH_1 is data duplication due to t - s and s - t similarity. A proposed solution is to proceed lexicographically [66]. A mapper only emits (t, s) if $t > s$. However, redundant data still exist in "inactive" reducers because similar records in $B_r(s)$ are sent to reducers although they are not elements in S. We illustrate the BH1 algorithm on an example as Figure 1.8.

BH_2

BH_2 is an extension of BH_1. The difference is that during the map phase, 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.

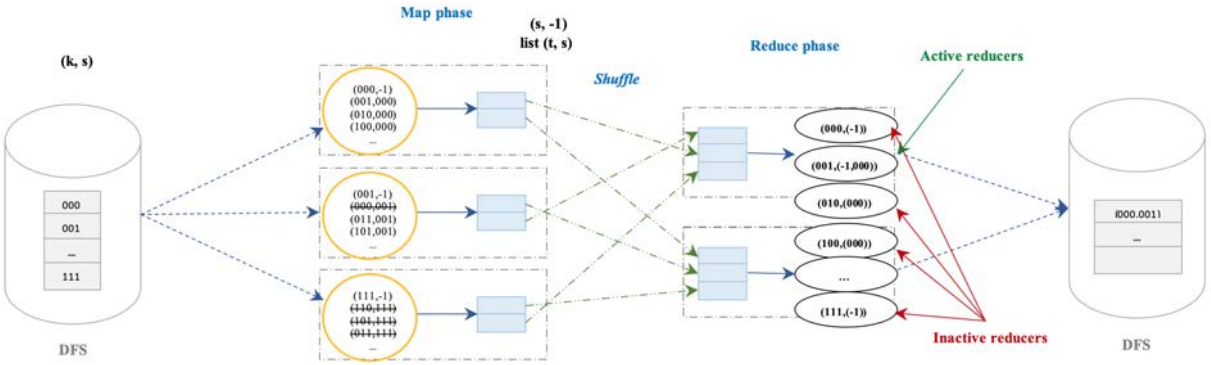


Figure 1.8 – An example of BH1 algorithm execution with $b = 3, d = 1$

Splitting Algorithm

Splitting algorithm [5] is based on a principle below

Lemma 1.3.1. *s, t are b-bit strings. s, t are partitioned into $d + 1$ substrings of length $b/(d + 1)$. Let s_i and t_i $(1 \leq i \leq (d + 1))$ denote each equal length substring in s and t respectively. If $HD(s, t) \leq d$, there exists at least one partition i such that $s_i \equiv t_i$*

Proof. To prove this lemma by contradiction, assume that $\nexists i$ such that $s_i \equiv t_i$ or in another words, $HD(s_i, t_i) \geq 1$. Therefore, $HD(s, t) = \sum_{i=1}^{d+1} HD(s_i, t_i) \geq d+1$. It contradicts that $HD(s, t) \leq d$ □

The Splitting algorithm is composed of a single MapReduce job and is based on splitting strings into substrings.

— Mappers decompose each input string s into $d+1$ equal-length substrings $s_1, s_2, ..., s_{r+1}$ [2] 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 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

2. [5] assumes that $d + 1$ divided by b evenly. If otherwise, some of the last pieces will be one shorter than the first pieces.

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.

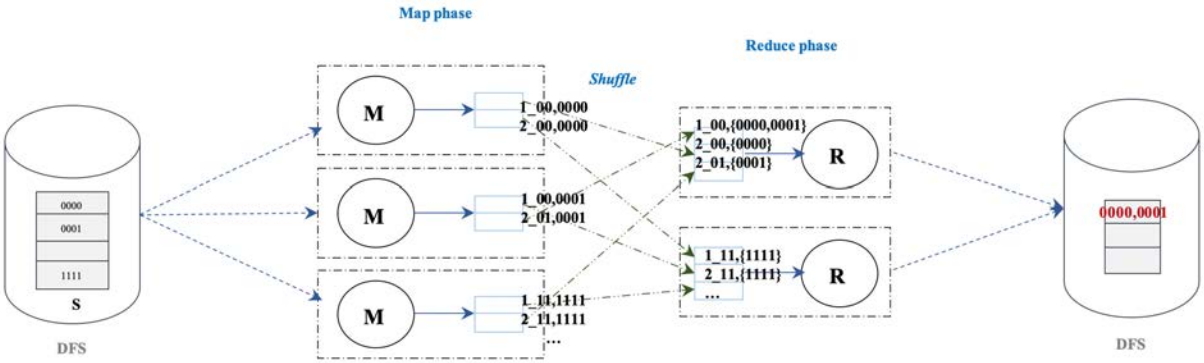


Figure 1.9 – An example of Splitting algorithm execution with $b = 4, d = 1$

Figure 1.9 illustrates an example of the Splitting algorithm execution with $b = 4, d = 1$.

Anchor Points Algorithm

Anchor Points [5] 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 $\lceil r/2 \rceil$ of any two strings in the input data. This algorithm will not be included in our research since the paper that introduced it shown that it is outperformed by the other algorithms [5].

Hamming Code

The Hamming Code algorithm [5] is a special case of fuzzy join technique proposed for string data and the Hamming Distance. Since this algorithm only works when $\tau = 1$ and the strings' length is one less than a power of 2, it is not included in our research.

1.3.2 Multiple Jobs Fuzzy Join Algorithms

The multiple job fuzzy join algorithms rely on computing the token set overlap measure. The previous researches follow the filtering verification framework. In the filter stage, it uses a lightweight filtering technique to identify a set of candidate pairs and prune lots of hopeless pairs. In the verification stage, it verifies every candidate pair and remove

the false positives. On the other hand, the communication cost is a significant problem in MapReduce environment. In order not to have to transfer all entire tuples of vast datasets over the network multiple times, filtering-verification based fuzzy joins require a tokenizing step (preprocessing step) and an aggregation step (postprocessing step) to process the tuples. Hence, its processing usually consists of three steps.

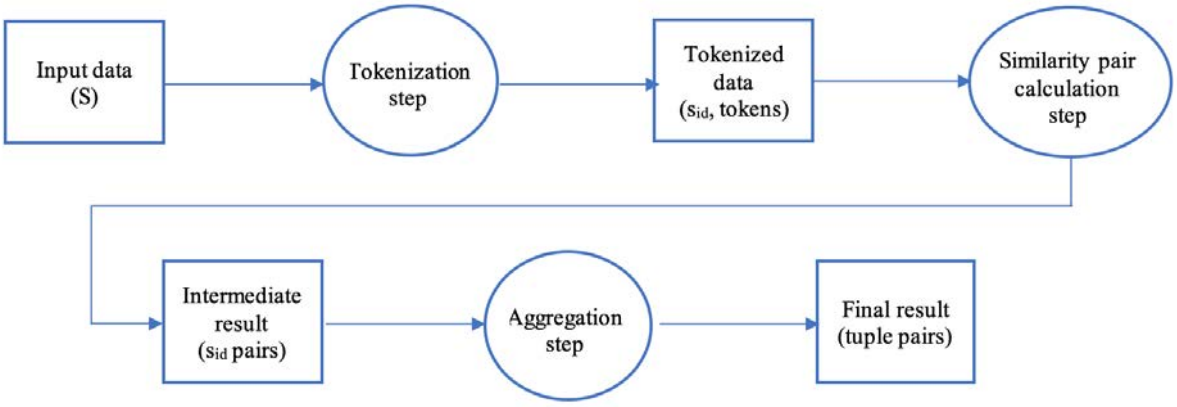


Figure 1.10 – A workflow of set based fuzzy joins

1. Preprocessing step (1) converts input tuples into a set of tokens in order to simplify the computation. A number of tokenization approaches can be adopted such as splitting the string into words (e.g., string "fuzzy big join" would be tokenized into set {fuzzy, big, join}) or computing the q-grams, which are overlapping strings of length q (e.g., 3-gram set for the string "Quyen" is {Quy, uye, yen}). In this step, each tuple is identified by a unique ID.
2. The tuple pair generation step (2) is the core of fuzzy joins. The second stage computes all similar pairs of sets and outputs the respective tuple ID pairs.
3. Postprocessing step (3) joins the original objects to the tuple IDs to produce pairs of tuples as the final result.

Figure 1.10 illustrates a workflow of multiple stage fuzzy big joins. Ignoring the efficient tokenizations and the trivial aggregation, the optimization of the algorithms is evaluated by the second step. In recent years, a number of solutions for the distributed set based fuzzy join have been proposed, most of which are founded on the MapReduce framework. These researches include FullFiltering [41], VernicaJoin [109], SSJ-2R [15], V-SMART

[85], MRSimJoin [97], MG-Join [93] ClusterJoin [34], MassJoin [36], FS-Join [94]. By the comparative evaluations in the survey [43, 99], the algorithm considered to be the most effective is the Vernica join[109]. So, in this thesis, we concentrate to study and optimize the Vernica join algorithm. To illustrate the algorithm, we consider an example as follows.

Example: Given a dataset $S = \{s_1, s_2, s_3\}$, $s_1 = \{A, B, C, D\}$, $s_2 = \{B, C, D, E, F\}$, $s_3 = \{A, C, D, F\}$, a similarity function is $J(s_i, s_j) = |s_i \cap s_j|/|s_i \cup s_j|$ (Jaccard similarity), a similarity threshold is $\tau = 0.6$. Since $J(s_1, s_2) = 1/2$, $J(s_1, s_3) = 3/5$, and $J(s_2, s_3) = 1/2$, the fuzzy join result is (s_1, s_3).

Full Filtering Join

Full Filtering Join [41] is a basic distributed algorithm for computing the similarity of every document pair, which we extended with a simple post-filtering. The proposed algorithm runs two consecutive MapReduce jobs, the first to build an inverted index [44] and the second to compute the tuples overlap and the final result, as follows:

— Indexing job: for each document s_i, the mapper M_1 emits the token as the key, and a tuple consisting of document ID and length as the value, i.e. $(tok, < s_{id}, l >)$. The shuffle phase of MapReduce, groups these tuples by tokens, and delivers these inverted lists to the reducers R_1, that write them to disk.
$M_1 : s \xrightarrow{map} list\{< tok, (s_{id}, l) >\}$
$R_1 :< tok, (s_{id}, l) > \xrightarrow{reduce} < tok, list\{(s_{id}, l)\} >$ //inverted lists

— Similarity pair generation job: For each token tok in the inverted list, the mapper M_2 generates all tuples that are associated with a key $< s_{idi}, s_{idj} >$ consisting in the pair of document IDs and a value $< l_i, l_j, 1 >$ including their lengths and a number 1 that represents for each their occurs. For any document pair, the reducer R_2 uses their lengths and the count of the numbers 1 to compute the Jaccard similarity score and verify each similarity pair. Since each tuple pair is verified by a single reducer, the output is duplicate free.
$M_2 :< tok, list\{(s_{id}, l)\} > \xrightarrow{map} list\{< (s_{idi}, s_{idj}), (l_i, l_j, 1) >\}$ //candidates
$R_2 :< (s_{idi}, s_{idj}), list\{(l_i, l_j, 1)\} > \xrightarrow{reduce} < s_{idi}, s_{idj} >$ //verification

A dataflow of the second step of full filtering joins is illustrated in Figure 1.11.

The running example is represented in the Figure 1.12. $M1$ generates inverted indexes for all tokens of each record s_1, s_2, s_3. Then $R1$ combines and produces inverted lists for every token generated. It explores the groups of records who have common tokens. Next,

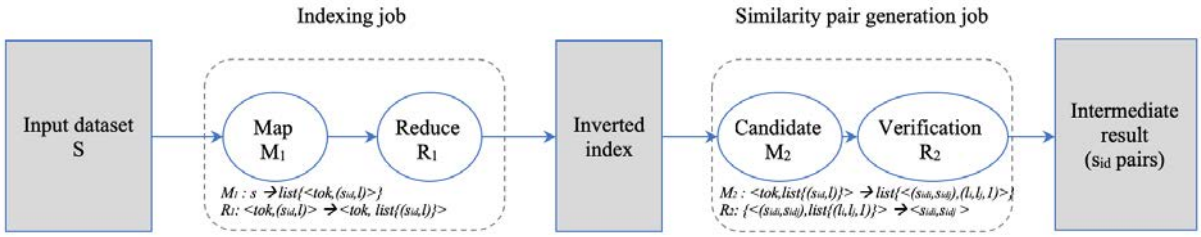


Figure 1.11 – A dataflow of full filtering joins

$M2$ generates all record pairs by combining every two of the records in the list produced. Finally, $R2$ uses the record lengths, which are stored with the respective records, to compute the Jaccard similarity and verify each record pair $(s_1, s_2), (s_1, s_3), (s_2, s_3)$. Hence, the result in this example is only (s_1, s_3).

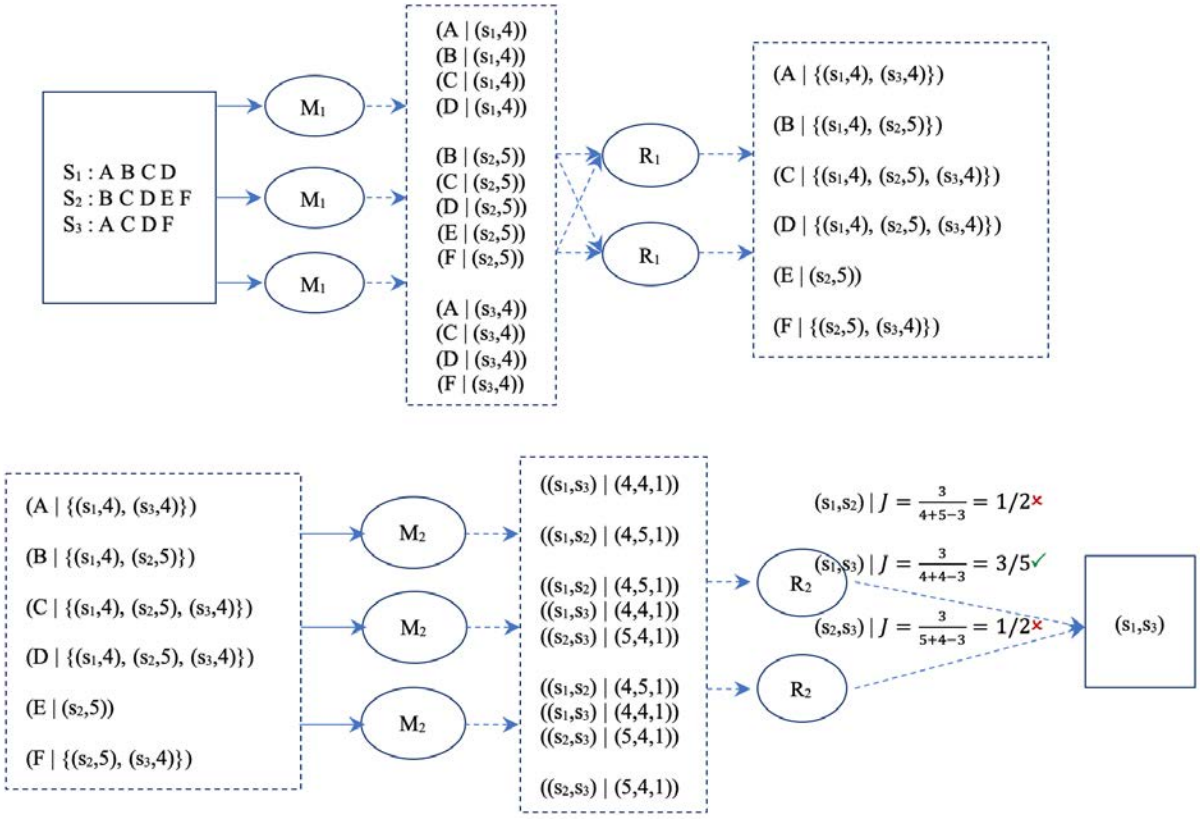


Figure 1.12 – An example of Full Filtering Joins with $\tau = 0.6$

Vernica Join

Vernica et al. [109] proposed an approach that is based on prefix filtering. Vernica joins support set-based distance functions like Jaccard Distance and Cosine Coefficient. There are multiple options presented for each stage, however, the paper states that BTO-PK-BRJ is the most robust and reliable option. Thus, this option is used in this research as the representative of this technique. First, it computes prefix tokens (job 1) and builds an inverted index on them (job 2). Then, it generates candidate pairs from the inverted lists, using additionally the length, positional and suffix filters to prune candidates (job 3). A deduplication step is finally employed to remove duplicate result pairs generated from different reducers (job 4). A dataflow of Venica joins is illustrated in Figure 1.13.

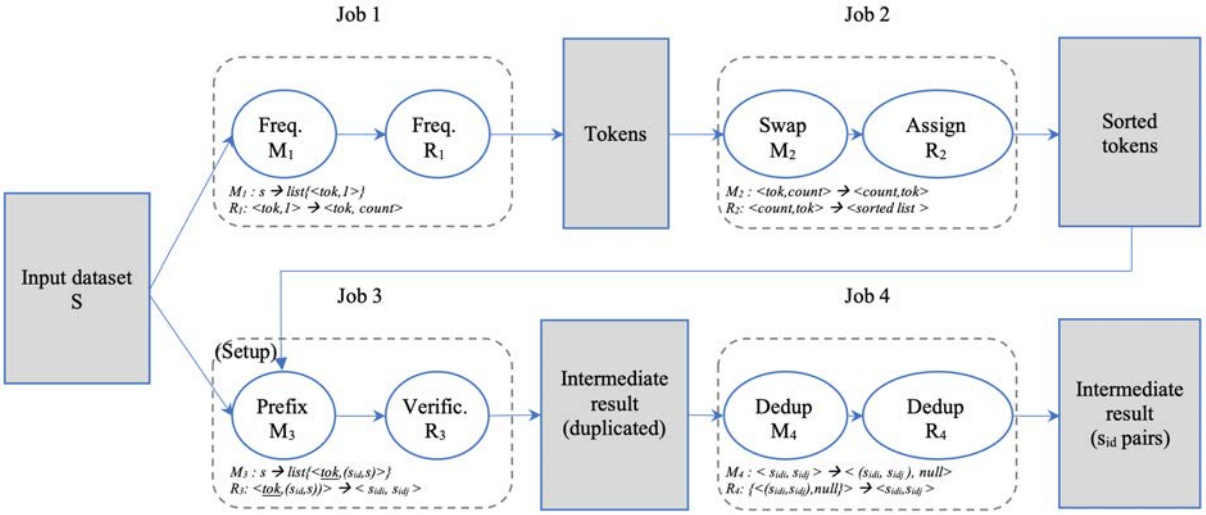


Figure 1.13 – A dataflow of Vernica joins

The set elements are first ordered based on some global token order. For each set, the k-prefix of a set are its k first elements in an arbitrary yet fixed order. With appropriate prefix sizes, a candidate pair of sets can be pruned if their prefixes have no common element. To improve the pruning power of the prefix filter, the sets are ordered by ascending global token frequency (GTF), i.e., rare tokens appear first in the prefix. This way, we try to eliminate higher frequency elements from the prefix filtering and thereby expect to minimize the number of comparisons. The prefix size depends on the similarity threshold (small prefix for high similarity), the record length, and the similarity function.

Lemma 1.3.2. *(Prefix filtering principle) [115, 27] Consider an ordering $\mathcal{O}$ of the token universe $\mathcal{U}$ and a set of records, each with tokens sorted in the order of $\mathcal{O}$. Let the p-prefix of a record s $(p(s,\tau) \subseteq s)$be the first p tokens of s. If $Sim(s,t) \geq \tau$, then the p-prefix of s and the p-prefix of t must share at least one token.*

$$Sim(s,t) \geq \tau \Rightarrow p(s,\tau) \cap p(s,\tau) \neq \emptyset$$

In other words, if there is an intersection between their prefixes, these two strings could be a result. The length of prefix for each similarity function can be computed as follows.

Overlap similarity: $p = |s| - \tau + 1$

Jaccard similarity: $p = \lceil (1-\tau) \cdot |s| \rceil + 1$

Cosine similarity: $p = \lceil (1-\tau^2) \cdot |s| \rceil + 1$

The first job computes the frequency of each token and the second job sorts the tokens based on their frequencies.

$M_1 : s \xrightarrow{map} list\{< tok, 1 >\}$

$R_1 :< tok, 1 > \xrightarrow{reduce} < tok, count >$ //global token frequency

$M_2 :< tok, count > \xrightarrow{map} < count, tok >$

$R_2 :< count, tok > \xrightarrow{reduce} < tok >$ //sorted list of tokens, single reducer

A dataflow of the token ordering for running example is illustrated in Figure 1.14.

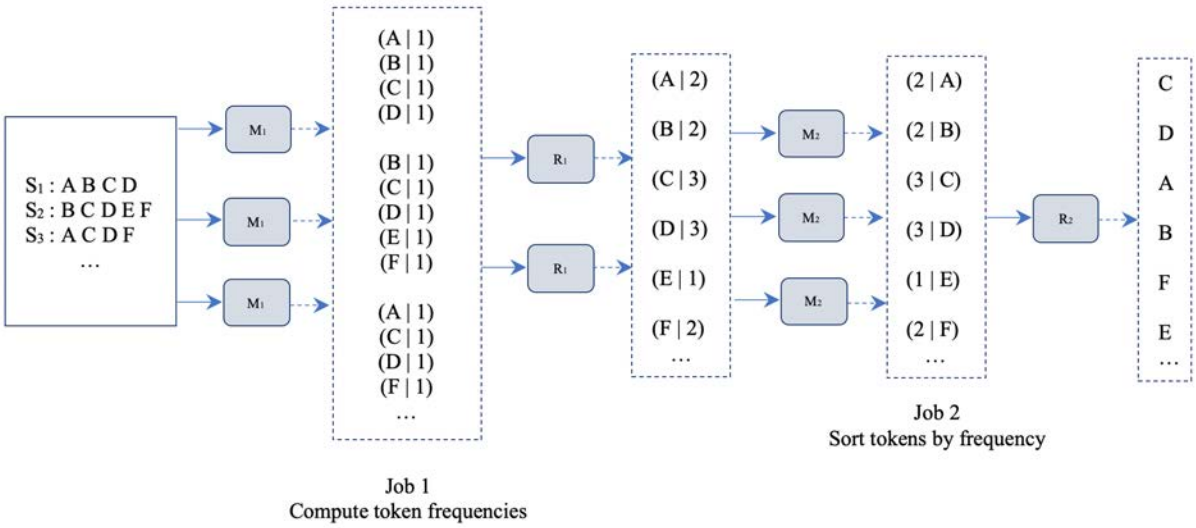


Figure 1.14 – Token ordering for Vernica joins

Mapper M_3 loads the resulting sort order in the setup function and creates the inverted index on the tokens in the prefix (prefix tokens are underlined, tok). Prefix length of each record s is calculated as $|s| - \lceil \tau \cdot |s| \rceil + 1$. Reducer R_3 generates candidate pairs from the inverted lists that are immediately verified.

$M_3 : s \xrightarrow{map} list\{< \underline{tok}, (s_{id}, s) >\}$ //prefix inverted lists

$R_3 :< \underline{tok}, (s_{id}, s) > \xrightarrow{reduce} < s_{idi}, s_{idj} >$ //verified pairs

Since different reducers may generate identical result pairs, a final deduplication step is required (Job 4). Figure 1.15 illustrates Job 3 for our running example.

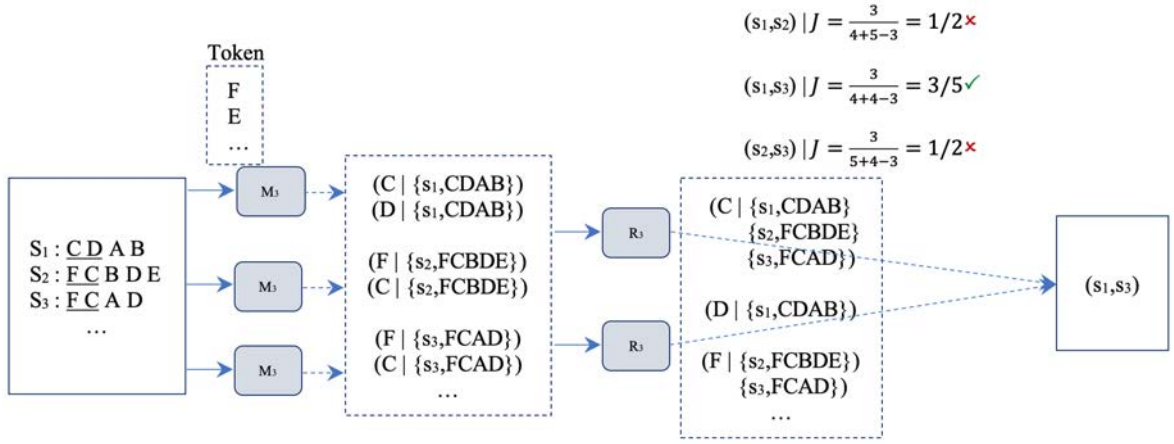


Figure 1.15 – Example of Job 3 for Vernica joins

1.4 Filtering Techniques

Our research aims to improve fuzzy joins in a MapReduce environment, relying on Filters. The main idea of filter techniques is to filter out the redundant data before it is transferred by applying filter structures as Bloom Filter, Intersection Bloom Filter and Counting Bloom Filter.

1.4.1 Bloom Filter And Bloom Join Algorithm

A Bloom Filter (BF) [20] is a space-efficient randomized data structure used for testing membership in a set with a small rate of false positives. Figure 1.16 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 $h_i : x \rightarrow \{1..m\}$.
— 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.

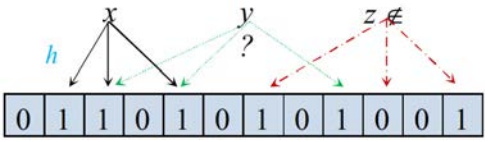


Figure 1.16 – A Bloom filter $BF(S)$ with 3 hash functions

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 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 [52]

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

These expressions are minimized when $k = ln2 \cdot (m/n)$, giving a false positive probability $f_{min} \approx (1/2)^k \approx (0.6185)^{m/n}$. In practice, k must be an integer, and both m/n (the number of bits per set element) and k should be thought of as constants. For example, when $m/n = 21$ and $k = 8$ the false positive probability is just over 0.0001.

The Bloom Join algorithm was proposed in [73] as an efficient query processing that focuses on minimizing the amount of data transfer between different sites, based on Bloom Filters data structure, using the MapReduce framework. Bloom filter is built based on the

key of each record. In the reduce side join, map phase takes the role of filtering the input records. In the reduce phase, copy/shuffle, sorting and reduce phase are taken place, in the reduce function, records are selected and disjoint ones are discarded. By the Bloom filter, this solution improves the efficiency of distributed join for large datasets.

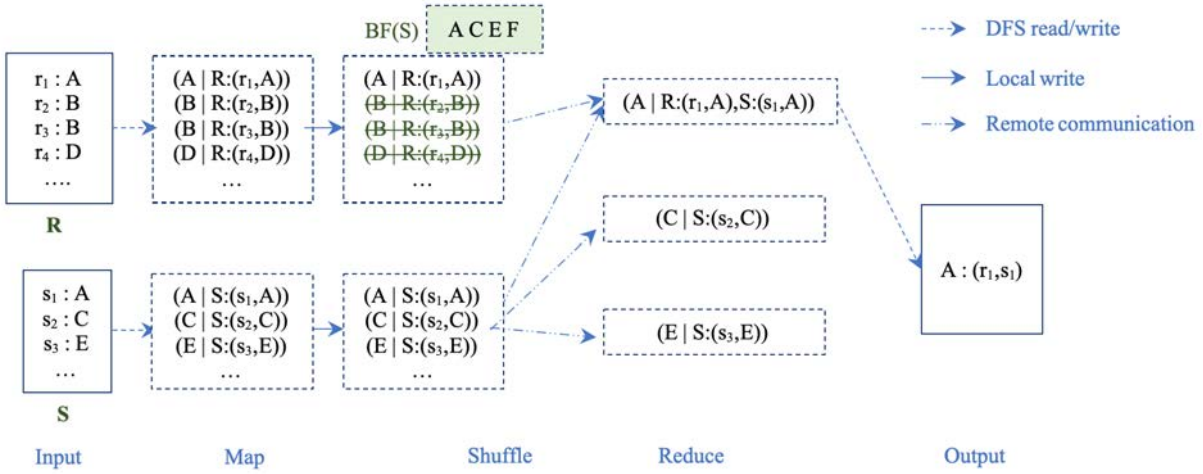


Figure 1.17 – $R \bowtie S$ using BF in MapReduce

Figure 1.17 describes an illustration of a basic join evaluation $R \bowtie S$ using the Bloom filter in MapReduce. A Bloom filter $BF(S)$ is first built for an input dataset S on a join key column and is delivered across all the mappers. Each mapper receives tuples from R or S, it eliminates tuples whose join keys are not in $BF(S)$ and emits key/value pairs for the remaining tuples. Then, the pairs are passed to corresponding reducers to be joined. However, with using only the filter $BF(S)$ for both the two inputs R and S, the algorithm can only eliminate non-joining tuples of the dataset R (e.g., tuples with the join key values of B and D) without eliminating non-joining tuples of the dataset S (e.g., tuples with the join key values of C and E).

1.4.2 Intersection Bloom Filter And Intersection Bloom Join Algorithm

Intersection Bloom filter (IBF) was introduced by our research group in [91, 90]. IBF is a probabilistic data structure that was designed for performing the intersection of two sets and used to denote common elements of sets with a probability of false positive. IBF-based joins [90] are studied to solve the problem of network overhead caused by

non-join data elements that produced and transmitted over the network in filter-based join algorithms using the popular MapReduce. Figure 1.18 presents three approaches proposed to compute the intersection filter: intersection to Bloom filters, unpartitioned and Partitioned Bloom Filters. It filters out disjoint elements or non-joining tuples from both datasets, not only on one input dataset. These approaches in [90] were conducted based on cost analysis of the join process to prove the efficiency of intersection filter. Its result showed that the proposed intersection filter gives high accuracy and reduces the cost of both communications and disk I/O, by reducing redundant data and efficiently filtering datasets.

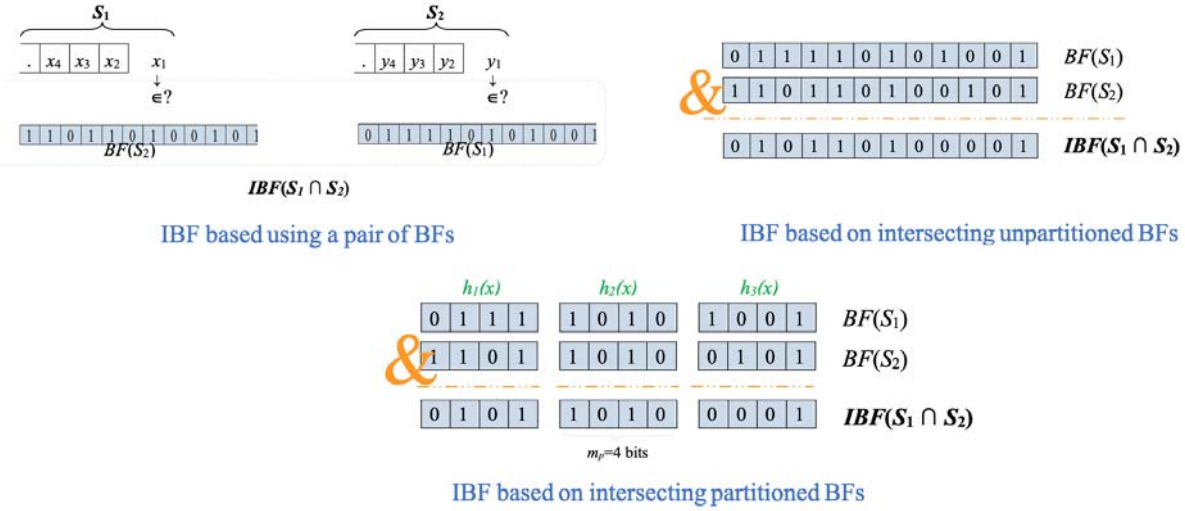


Figure 1.18 – Three approaches of IBF structure

Figure 1.19 describes an illustration of an improvement for the example in the figure 1.17 using the Intersection Bloom filter. This join optimization takes the data redundancy in both R and S before sending to the join processing.

1.4.3 Counting Bloom Filter

The Counting Bloom Filter (CBF) suggested by Fan et al. [42] is a variant BF, in which each hash entry contains a counter with a fixed size of b counters bits, instead of a single bit in BF. Hence CBF also needs b times the memory space consumed by BF. Counters size b must be chosen large enough to avoid overflow. For most applications, four bits suffice to practically obtain a negligible overflow probability [42, 23].

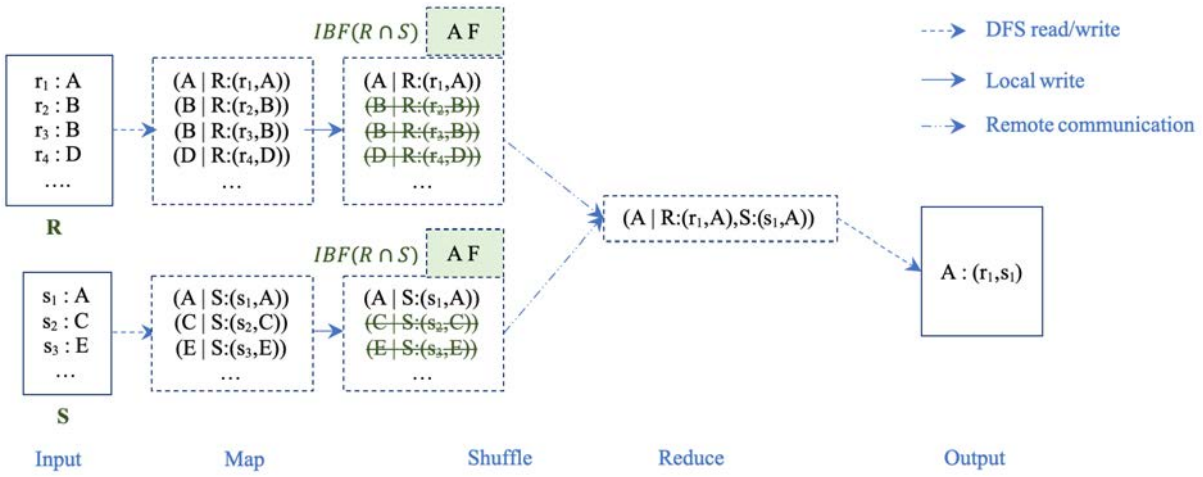


Figure 1.19 – $R \bowtie S$ using IBF in MapReduce

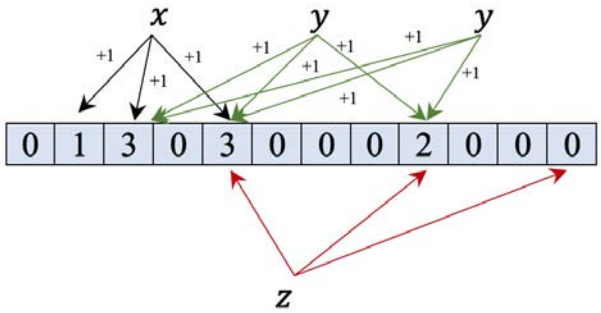


Figure 1.20 – Counting Bloom Filter

As shown in Figure 1.20, to insert an element, all the corresponding hashed counters are incremented by 1. Likewise, to delete it, all of its hashed counters are decremented. To determine if an element $z \in S$, we check if all of its hashed entries are positive. Given only insertions, the false positive rate of CBF is the same as for BF. Based on CBF ideal, we can build a Global Token Frequency by using their counters to improve Vernica joins.

1.5 Conclusion

This chapter presents the overview of related work about the fuzzy big joins. We have reviewed the popular and important techniques for handling large scale datasets, the MapReduce framework, Spark and the filters. The MapReduce programming model enables easy development of scalable parallel applications to process vast amounts of

data. Spark, a MapReduce-like cluster computing engine, extends the MapReduce model to better support cache and serial job processing. The Bloom filter and the Intersection filter based on space efficiency have been applied to optimize for the join processing. The Counting Bloom Filter saves the number of occurrences of an element and then can be applied to build Global Token Frequency.

Furthermore, we provide a state of the art on the status of studies on fuzzy big joins with MapReduce and the recent research. We present an overview of the prominent parallel fuzzy join algorithms and categorize them with respect to their strategies: single job and multiple jobs. In particular, we have studied their limitations by using the cost model $M - C - R$. Through the survey, we realize that there remain a lot of non-joining data sent to the reducers in the existing join algorithms as well as duplicated pairs in results. Therefore, we need to look for a type of filter that has the ability to eliminate all hopeless tuples that do not participate in the fuzzy join result.

There are some important variations of the Bloom filter such as compressed Bloom filter [86], spectral Bloom filter [32], Bloomier filter [28], space code Bloom filter [68], distance-sensitive Bloom filter [67], etc. A variant called Counting Bloom Filters (CBF) [42] allows deletion of elements from the Bloom filter by using counters instead of a single bit at every position. Besides, another version of the Bloom filter is Invertible Bloom Filters (InvBF) [48] that supports not only the insertion, deletion, and lookup of elements, but also enables a listing of its contents with a probability. However, all these variant filters are not designed for our purposes. Furthermore, a SuRF [119] 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 true close elements, lead to inefficient and high costs. As a result, a fuzzy filter for optimization of fuzzy joins should be proposed to devise better optimizations for the fuzzy big joins that are the main subject of this research.

Chapter 2

Fuzzy Big Joins Improvement Using Bloom Filters

Afrati et al. [5] proposed multiple algorithms to perform fuzzy join in a single MapReduce stage. While their algorithms have the common limitations as redundancy and duplication of data, we were interested in using Bloom Filters [20] to improve them. 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 $M - C - R$ computation.

This chapter is formed as follows. Section 2.1 provides a short description of previous work as well as points out its limitations. We then introduce an overview of our contributions, definitions and notations. The remainder of the chapter therefore presents our proposals in detail. Section 2.2 describes our approaches to optimize fuzzy big joins. In addition, we take advantage of Spark to implement our algorithms. A comparative analysis of the costs of fuzzy joining algorithms can be found in Section 2.3. Next, the evaluation environments, experimental protocols and experiments are reported in Section 2.4. Finally, Section 2.5 includes conclusions on our work.

2.1 Previous Works

Ball Hashing is based on the ball of radius r to compute the fuzzy joins. The mappers generate and send all elements t in the ball of radius r of each input record s to the corresponding reducers. However, the number of elements in each Ball of radius r is large (exponential r, depends on the string length) while not all its elements exist in the input dataset S. That means a significant number of elements generated in each ball are sent to the inactive reducers. As a result, they lead to data redundancy, even a serious bottleneck for the Ball hashing algorithm [99]. Lexicographically deals only partially with

redundant data and duplication. Figure 2.1 clearly illustrates these limitations. The pairs $(000, 001), (110, 111), (101, 111), (011, 110)$ are removed by the lexicography.

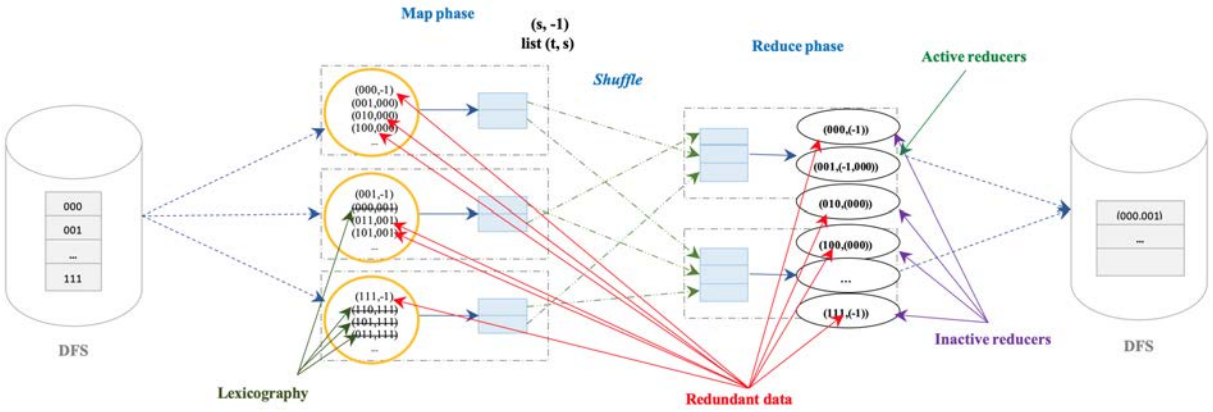


Figure 2.1 – An example for limitations of BH1 algorithms execution with $b = 3, d = r = 1$

Besides, Splitting algorithm generates redundant data by sending each record to $d + 1$ reducers. In fact, each record just needs to be sent to some identified reducers if its actual similar elements present in S and its substrings are known. Furthermore, a considerable quantity of duplicate data is produced because similar strings have more than one common substring. It wastes network resources, computing resources and reduces the efficiency of the algorithm. Figure 2.2 presents limitations of the Splitting algorithm. With $b = 6, d = 2$, each record generates 3 splits. We can see that there are hopeless candidates, which are non matching substrings (eg. $(3_01, 000001), (2_11, 111111), (3_11, 111111)$). Moreover, 000000 and 000001 have more than one common split $(1_00, 2_00)$ which leading to wastes of verification and duplication results problem.

For these reasons above, we can take advantages from filters to be able to filter out non-joining data of the input datasets. This chapter, therefore, makes the following main contributions: (a) optimization for Ball Hashing and Splitting algorithms using the Bloom filters; (b) implementations of our approaches for fuzzy self joins and fuzzy two-way joins in MapReduce as well as in Spark; (b) comparison among the fuzzy joins using different approaches through cost models and experiments. In order to present our contributions and to compare the costs of different algorithms, we supply notations and parameters used in this chapter as shown in the Table 2.1.

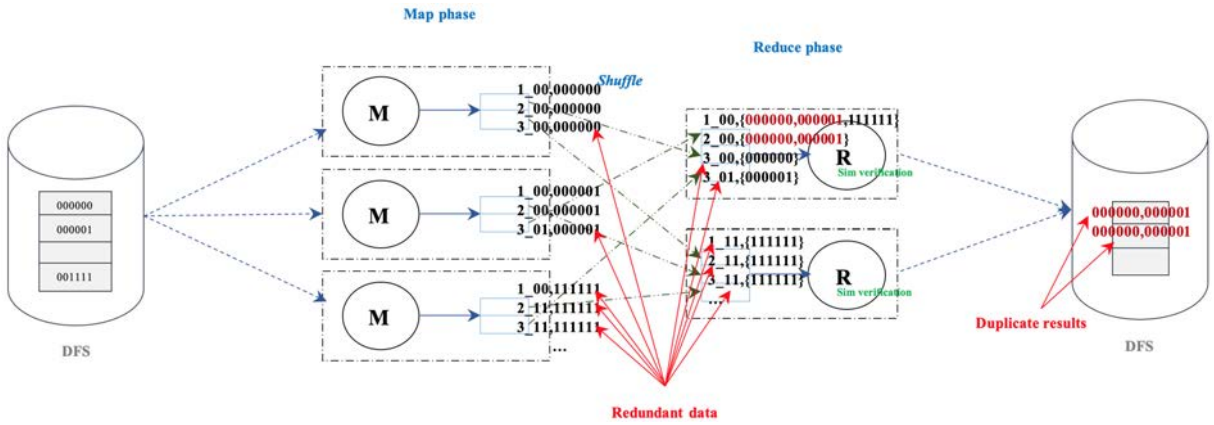


Figure 2.2 – An example for limitations of Splitting algorithms execution with $b = 6, d = 2$

2.2 Improvement Of Fuzzy Big Joins Using Bloom Filters

From the limitations of data redundancy, in this chapter, we propose to integrate Bloom Filter (BF) into the fuzzy join algorithms to improve their performances.

2.2.1 BF-BH Algorithm

During the map phase, BH_1 generates all elements within a distance d from s and sends them to the reducers for combining with similar input records. It is easy to see that not all elements in the $B_d(s)$ belong to S. $BF(S)$ represents all the input records. A membership test for some key value s on BF tells whether s participates or not to the join result. Our approach integrates $BF(S)$ to remove elements in $B_d(s)$ that do not belong to S before sending it to the reducers. This solution consists of two stages:

1. Pre-processing stage: A filter $BF(S)$ is built on a join key value set of the input dataset S. Figure 2.3 describes an example of the preprocessing stage of $BF-BH1$ for the fuzzy join with 3-bit strings. The mappers scan splits of input dataset S, extract the join key column for each tuple, and insert the join keys of S into local Bloom filters $BF(S_i)_{local}$ on tasktrackers. The mappers emit the local filters to one reducer. The reducer receives all the local filters from all the mappers, merges these filters into the global filter $BF(S)_{global}$ using the bit-wise OR the bit arrays. By the

Table 2.1 – Notation summary

Notation	Description
M	Total computation cost (map or preprocessing) 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
τ, d, r	Pre-specified threshold of distance and radius of Ball
s, t, b	A string s or t and its length
B_r	Ball of radius r
k	Number of hash functions
K, n	Number of reducers
D	Size of intermediate data for shuffle
$BF(S)$	Bloom filter built for a set S
$f_{BF(S)}$	False positive probability of $BF(S)$

same size m and k hash functions, then

$$BF(S)_{global} = BF(\bigcup S_i) = \bigcup BF(S_i)_{local} \quad [53]$$

The global filter is then stored into a file on Distributed File System (DFS).

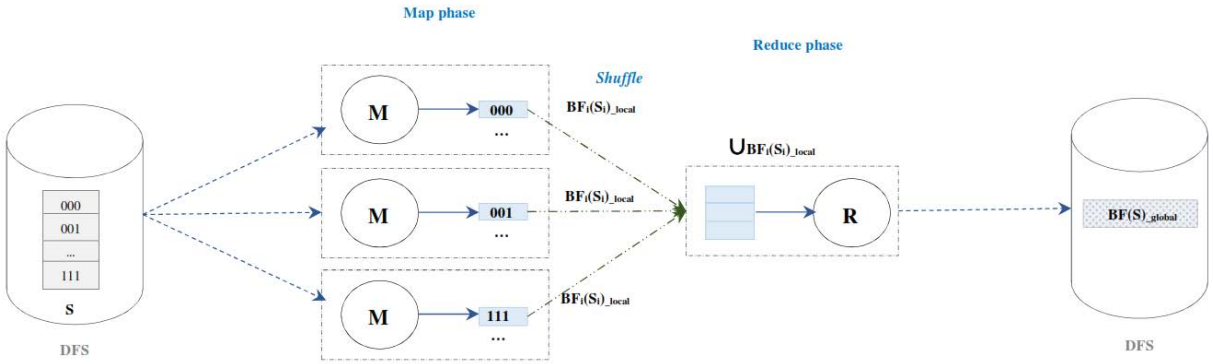


Figure 2.3 – An example of pre-processing stage

Each input record s is hashed by k functions. So, the map cost of preprocessing stage is $k|S|$. The number of $BF(S_i)_{local}$ that is transferred depends on the number of mappers, and thus lead to the number of OR computations in the reducer.

As an option, when the size of filters is large, the filter files will be compressed in

formats such as gzip, bzip2, etc. This compression is efficient for delivering filters to all distributed nodes.

2. Join processing stage: $BF(S)$ is distributed to all the computing nodes and used to eliminate early non-similar elements of the input dataset in each ball of radius d during the map phase. In order to start the job, the bloom filter file $BF(S)$ is distributed to all the compute nodes in a cluster using a distributed cache. Then, the jobtracker will create map tasks for the input dataset S, reduce tasks, and assign each split to one map task run on a tasktracker. Its implementation includes the following two phases.

   — Map phase using the filter: Each mapper uses an initialization function to load the file $BF(S)$ into memory. The mappers first read each tuple from its split, generate all elements t in the ball of radius r ($d = r$) of each input record s ($B_r(s)$). t is a string obtained from s by changing from 1 to r bits. The map function then queries each $t_i (i = 1..|B_r|)$ of the tuple s into the $BF(S)$. Its map output result is given as follows:

   - If all elements t_i are not in $BF(S)$, the tuple s is ignored.
   - Conversely, if it exists at least one t_i in $BF(S)$, this means that t_i is capable of a real similar element of s in S. To avoid creating duplicate pairs, we consider this possibility as one of two cases:
     - If $t_i < s$, the tuple is mapped as (key, value) pairs of the form (t_i, s).
     - On the contrary, if $t_i > s$, a pair $(s, -1)$ is emitted.

     The number of reducers is thus $n = 2^b/2 = 2^{b-1}$

   Then these filtered pairs are sent to the corresponding reducers. By this way, both "inactive" reducers that do not receive any pair $(s, -1)$ as well as "useless" reducers that receive only one pair $(s, -1)$ are almost non-existent. In other words, there are no needless pairs any more because non joining elements are almost filtered. A tiny amount remains due to the false positive probability of the $BF(S)$.

$$s \xrightarrow[BF(S)]{map} \begin{cases} (s, -1) \\ (t, s), \quad \forall t \in B_s(d) \cap S, t < s \end{cases}$$

   When the mapper emits data, these intermediate pairs are partitioned, sorted, merged and written to disk in a single intermediate file. Then, the framework sends the pairs across the network to the corresponding reducers.

   — Reduce phase: After filtering none relevant data, the join algorithm then proceeds

as in BH_1. This reduce phase is the same as the one of the basic join. The reducer receives the list of tuples of the form (key, value) with the same key. Then the active reducers that receive -1 in the value call the reduce function to perform the cross product for each key. On the contrary, the inactive reducers that do not receive -1 are ignored. Finally, the BF-BH fuzzy join is completed by writing the output to DFS.

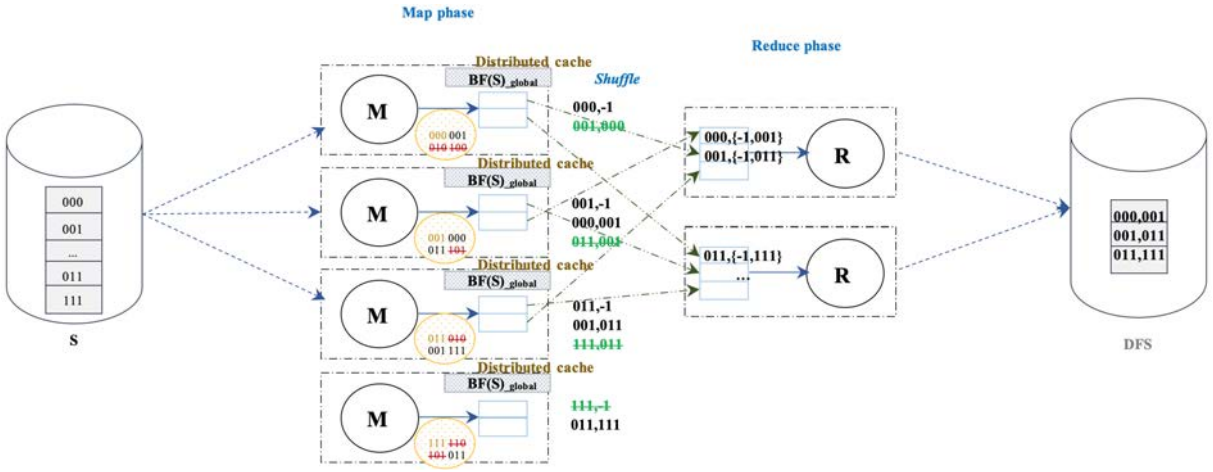


Figure 2.4 – An example of join processing stage of BF-BH1 Algorithm

An example of join stage of $BF-BH1$ for the fuzzy join with 3-bit strings and a threshold $d = 1$ is shown in Figure 2.4. In the map phase, for each B_d, the redundant elements (in red color) that is not in $BF(S)$ are eliminated. Then, the elements (in black color) that pass over the BF, are considered again.

— For 000, $BF(S)$ returns 001 that is greater than 000, thus the mapper emits a $(000, -1)$ pair.

— For 111, $BF(S)$ returns 011 that is smaller than 111, thus the mapper emits a $(011, 111)$ pair.

— Similarly, all the green elements are filtered again. The black elements are sent to corresponding reducers and performed join operations.

Based on these optimizations, redundant data are removed and the duplicate pairs are prevented.

This approach will also be applied to $BH2$. Let us recall the 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 $k|B_d|$.

In the shuffle phase, after filtering, 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, with $n = 2^{b-1}$ is the number of reducers, the processing cost in the reduce phase is also improved, the reduction being: $D_{BF-BH1}|S|/n$.

Regarding the extension for this fuzzy two-way joins $S \bowtie_d T$, we can consider the possibility of filter building on one or two input datasets. If the filter is established on both data sets, one is built on the key set of S, then the other is built on all balls of the key set of T. This leads to a high cost for the preprocessing stage. Furthermore, the generation of balls can be redundant if the input data contains duplicate elements. Therefore, the Bloom filter is just necessary to be built for one of the two input datasets. The preprocessing stage is also executed as in Figure 2.3 above. In the join processing stage:

— For the input dataset S which is used to build $BF(S)$ in the preprocessing stage, each element is mapped one to one without filtering.

$$s \xrightarrow{map} (s, s)$$

— For the remaining input dataset T, each tuple t generates its ball. Each generated element is filtered by $BF(S)$ and then is mapped in the form of

$$t \xrightarrow[BF(S)]{map} (t_i, t) \quad \forall t_i \in B_t(d) \cap S$$

With this solution, one of the two input datasets is not filtered, hence, the non-joining tuples in T are also transferred over the network. On the other hand, the cost of ball calculations is worth considering. Thus, if the sizes of the two input datasets are not equal, in order to balance between the preprocessing cost and redundant data filtering cost, we should choose to build the filter on the larger dataset. The remaining smaller dataset will be computed its balls for filtering.

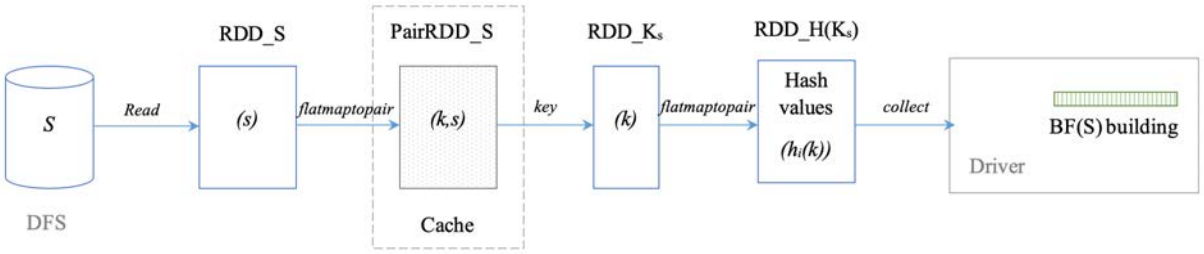


Figure 2.5 – Flowchart for preprocessing stage for $BF(S)$ building in Spark

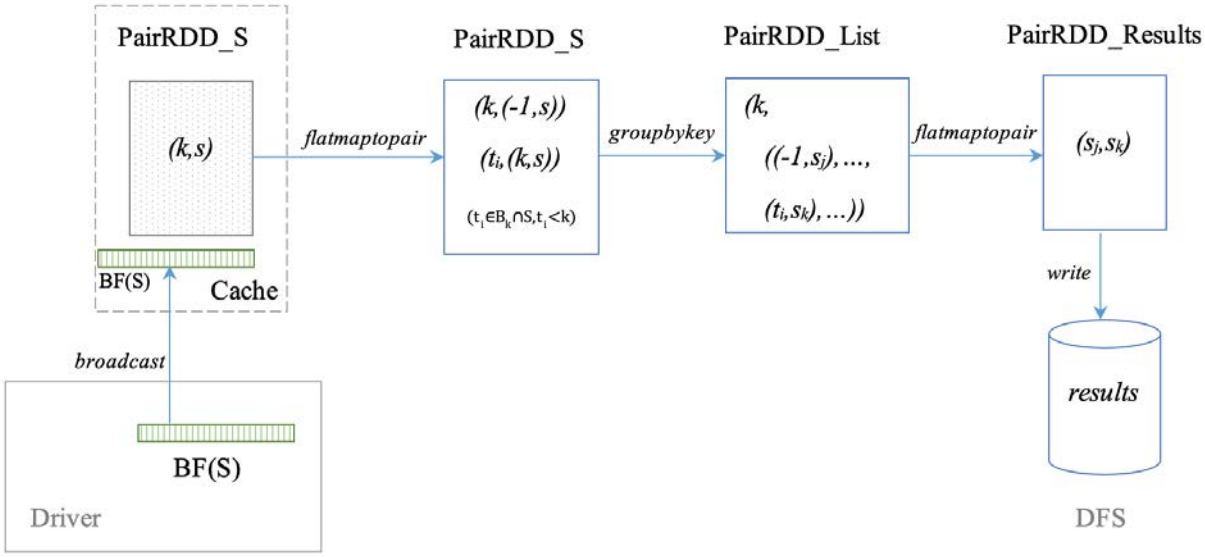


Figure 2.6 – Flowchart for join processing stage of BF-BH1 algorithm in Spark

Taking advantage of cache, we implement BF-BH proposals in Spark framework as in Figure 2.5 and Figure 2.6. *Flatmaptopair* is a map transformation which maps each partition to a new one without shuffling data. *GroupByKey* and *join* need to shuffle the data to build new partitions. The input dataset S is read and mapped to PairRDD_S to identify its main join keys one time, hence, its PairRDD_S partitions are cached to reuse in the join processing stage. Its main keys k_s then are extracted and create its hash values $(h_i(k_s))$. These hash values are collected at the driver program for the filter building. Next, the $BF(S)$ is broadcast to all worker nodes and begins the join processing stage. PairRDD_S generates its balls and uses the $BF(S)$ to determine intermediate elements to transfer and the ones to prune out. Next, these filtered elements are grouped by their keys to create their join list. Finally, each pair elements is mapped as a result without verification and written to DFS.

Figure 2.7 illustrates for the join processing stage of BF-BH1 two-way join algorithm in Spark. A PairRDD_S is created by dataset S stored in the distributed file system to build $BF(S)$, and also is cached in memory in the preprocessing stage. Dataset T is read, computed and filtered by $BF(S)$ in the cache to create a PairRDD_T. Two PairRDDs S and T then join by their keys to create their join list. The PairRDD_S in memory is not filtered in this case. Therefore, the non-joining elements in S are also transferred to the join transformation. The final results are mapped without verifications or duplication.

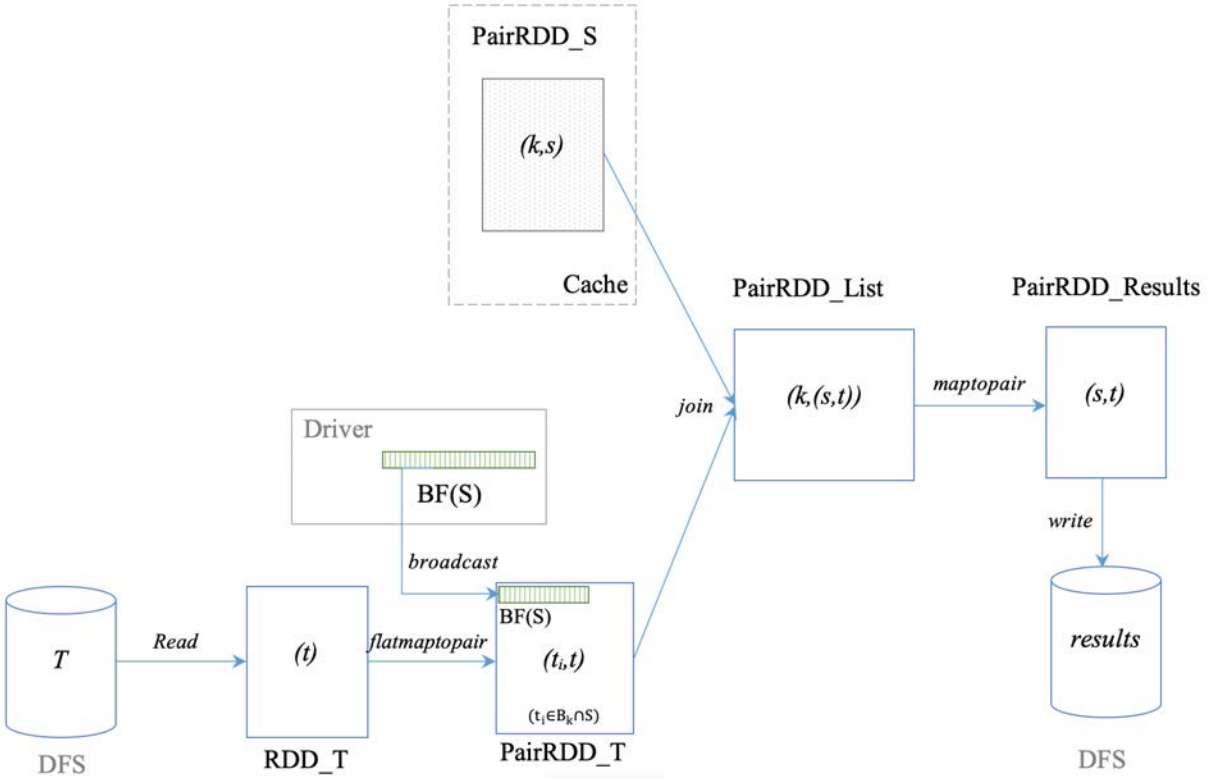


Figure 2.7 – Flowchart for join processing stage of BF-BH1 two-way join algorithm

2.2.2 BF-Ball-Splits Algorithm

As mentioned above, Splitting algorithm generates a considerable amount of redundant data from the non real matching substrings. As a solution for this problem, we propose to combine Ball Hashing, Splitting and BF. Thus, this approach also requires two stages:

1. Pre-processing stage has to build a $BF(S)$ with a cost of $k|S|$. This job is the same with BF-BH1 algorithm.

2. Join stage details is described as follows:

   First, each mapper uses an initialization function to load the $BF(S)$ from distributed caches into memory.

   Then, mappers read each tuple from its splits and generate 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.

   Next, each of them is divided into $d+1$ 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 (i_s_i, s) will be outputs, and then t and s_i will never be considered again. In other words, only one first t_i that matches with s_i is mapped. By this way, mappers emit only real matching substrings of s with existing similar elements in the input dataset S. Non matching substrings are filtered out.

$$s \xrightarrow[BF(S)]{map} (i_s_i, s) \begin{cases} s_i \subset s, i = 1..d+1 \\ \forall t \in B_s(d) \cap S \\ \exists t_i \subset t, t_i \equiv s_i \end{cases}$$

   In this join stage, each record s generates $|B_d|$ elements. Each element is cut into $d+1$ splits. Thus, with the k hash functions of $BF(S)$, the map cost of this join stage is $k|S||B_d|(d+1)$.

   When mappers emit data, these intermediate pairs are partitioned, sorted, merged and written to disk in a single intermediate file. Then, the framework sends pairs across the network to the corresponding reducers.

   Each record is sent only if there are actual similar elements, with a small false positive. Even if more than one similar element of s exists a common substring, they are sent only once to a corresponding reducer. Therefore, with δ_S is the ratio of similar records of S, $f_{BF(S)}$ the false positive probability of the BF of S, the communication cost is

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

$$D_{BFBallSplits} < (d+1)|S|$$

   Finally, reducers collect, compute the distance, and output results as in Naive algorithms. In this approach, each similar pair in S is sent to at most one re-

ducer in general. Thus, the total computation cost for reducers is smaller than $D_{BFBallSplits}|S|/2^{b/(d+1)}$. However, duplicate pairs are also created when elements share multiple common splits with different elements. For example,

— 000110 shares 1_00 with 000111, 2_01 with 010110.

— Besides, 000111 also shares 1_00 with 000110, 2_01 with 010111.

— Hence, 000110 and 000110 are sent to both 1_00 and 2_01 reducers. They produce duplicate pairs in results.

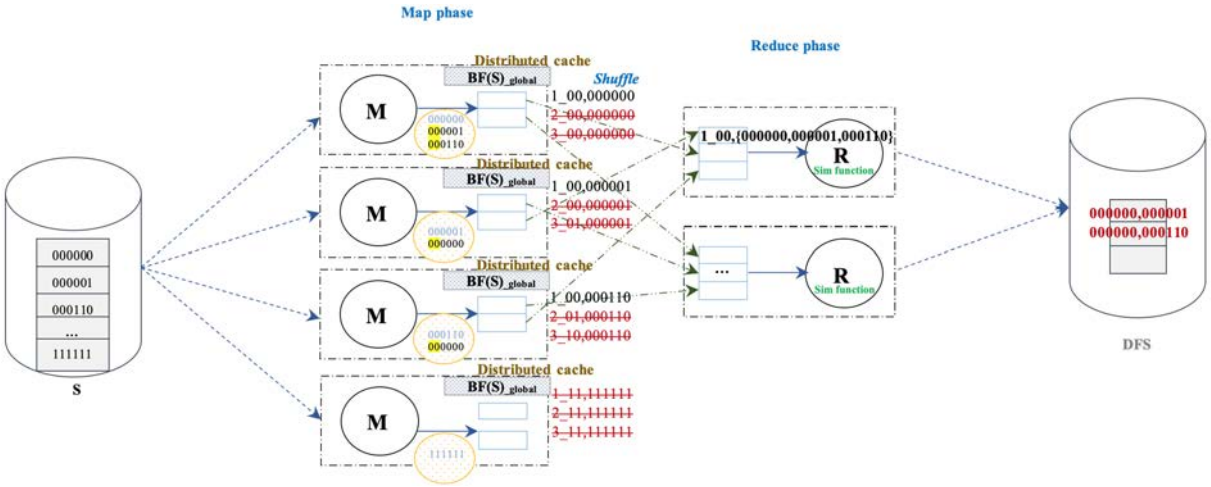


Figure 2.8 – An example for join stage processing of BF-Ball-Splits Algorithm with 6-bit strings and a threshold $d = 2$

An example of a join stage execution of $BF - Ball - Splits$ for fuzzy joins with 6-bit strings and a threshold $d = 2$ is shown in Figure 2.8. We consider the filtering efficiency in this example as follows:

— For 000000, Only two similar elements $(000001, 000110)$ in its ball of radius 2 passes the $BF(S)$. They are cut into 3 substrings: $1_00, 2_00, 3_00$. Considering 000001, there is a first 1_00 substring that matches with 000000. Thus, a pair $(1_00, 000000)$ is emitted. So, 1_00 and 000001 are skipped. The next 000110 has then no remaining splits match with 000000.

— Similarity, 000001 and 000110 are also mapped with the first 1_00 split.

— For 111111, all elements in its balls are filtered out by $BF(S)$.

— In this example, these 4 records create only 3 intermediate tuples instead of 12 as in Splitting algorithm. They are sent to only one reducer and emit two pairs of results. There is no redundant data as well as duplicate pairs in this case.

Figure 2.9 illustrates the join processing stage of BF-Ball-Splits algorithm in Spark. As in BF-BH1 algorithm, a PairRDD_S is created by dataset S stored in the distributed file system to build $BF(S)$, and also is cached in memory in the preprocessing stage. Next, the BF(S) is broadcast to all worker nodes. PairRDD_S generates its balls and its splits, and then use the $BF(S)$ to determine which splits are transferred, which ones are pruned out. Next, these filtered elements in PairRDD_SSplits are grouped by their split keys to create their join list. Finally, each pair elements is verified and mapped as a result.

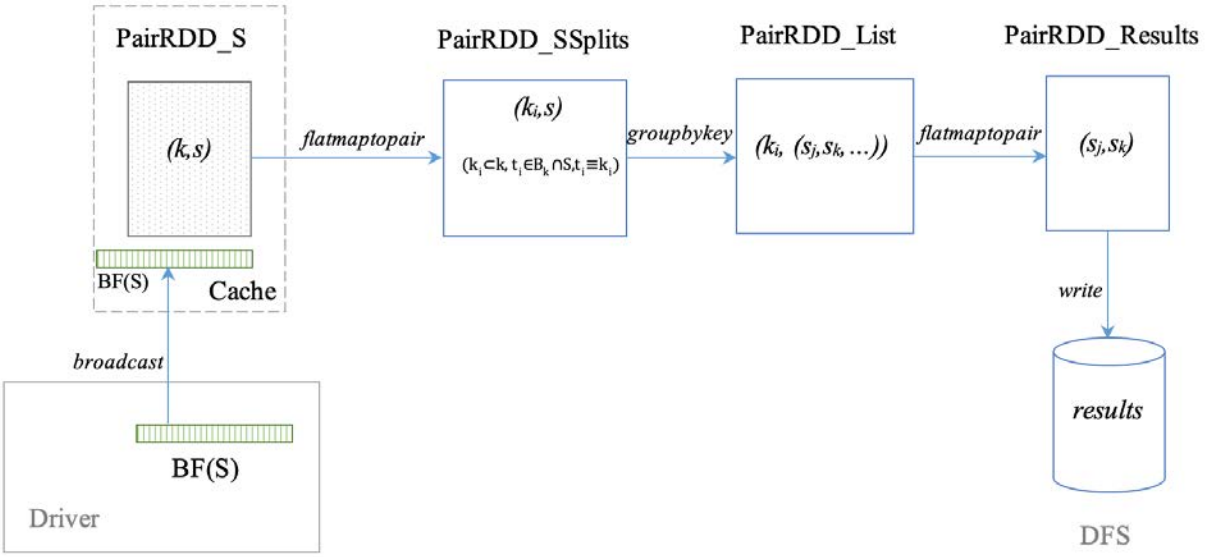


Figure 2.9 – Flowchart for join processing stage of BF-Ball-Splits algorithm in Spark

Considering the extension for this fuzzy two-way joins $S \bowtie_d T$, to avoid ball calculations for both input datasets, an Intersection Bloom Filter of splits $IBF(SSplits \cap TSpits)$ can be applied. The problem becomes a normal IBF two-way join. However, dissimilar tuples have also common splits. Thus, the problem of redundant intermediate data has not been fully resolved in this solution. In addition, similar tuples have more than one common splits lead to duplicate results. In general, this solution only filters out dissimilar splits.

2.3 Cost Analysis

Table 2.2 summarizes the costs of the different algorithms by the M-C-R cost model. According to the processing cost, Naive algorithm is the most expensive solution, but its

Table 2.2 – Summary of costs for various Hamming distance-based join algorithms

Approach	Pre-proc.	M cost/element	#Reducers	Communication cost	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	$k\|S\|$	$k\|B_d\|$	$n = 2^{b-1}$	$D_{BF-BH1} < \|S\|\|B_d\|$	$D_{BF-BH1}\|S\|/2^{b-1}$
Splitting	0	$d+1$	$(d+1)2^{b/(d+1)}$	$(d+1)\|S\|$	$(d+1)\|S\|^2/2^{b/(d+1)}$
BFBallSplits	$k\|S\|$	$k\|B_d\|(d+1)$	$(d+1)2^{b/(d+1)}$	$D_{BFBallSplitting} < (d+1)\|S\|$	$D_{BFBallSpitting}\|S\|/2^{b/(d+1)}$

Table 2.3 – Value of expressions from Table 2.2 when $b = 20, d = 4, |S| = 10^5, K = 10^4, \delta_S = 1\%, k = 8, f_{BF(S)} = 10^{-4}$

Approach	Pre-pro.	M cost/element	#Reducers	Communication	Processing
Naive	0	100	10^4	10^7	10^{10}
BH1	0	6226	1048576	6.2×10^8	6.2×10^7
BF-BH1	8×10^5	49808	524288	6.26×10^6	6.26×10^5
Splitting	0	5	80	5×10^5	3.1×10^9
BFBallSplits	8×10^5	249040	80	10^3	6.2×10^3

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 the processing cost. However, Ball Hashing is sensitive to distance. With the greater the distance d, the higher number of elements in $B(d)$. 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 [10]).

— The map phase uses k hash functions for the membership test. In the BFBallSplitting, the map phase generates $B_d(s)$ for each input record.

— The number of reducers does not change.

— Using $BF(S)$, redundant elements are eliminated, thus the communication cost is reduced. This also decreases the computation cost on reducers.

We use expressions in Table 2.2 to see how these costs grow as the data size grows via a concrete example. 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 ratio of similar records of S is $\delta_S = 1\%$, the small false positive of $BF(S)$ is $f_{BF(S)} = 0.0001$. Table 2.3 compares the cost values of various algorithms.

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 [90, 91] have proved that filtering

can significantly improve execution times.

2.4 Experimental Validation

In this section, we present experimental results obtained from the execution of fuzzy self joins and fuzzy two-way joins using the different approaches. Together with this, our discussion focuses on their performance aspects.

2.4.1 Cluster and Datasets Descriptions

All experiments were run on the Galactica cloud project platform [45] with a computer cluster of 5 nodes. Each machine has 16 CPUs, 32GB RAM and 320GB disks. The operating system is 64-bit Ubuntu server 14.04 LTS, and the java version is 1.7.0.21. We installed Hadoop [8] version 2.7.2 and Spark [10] version 3.0.1 on all nodes.

Spark is configured to run in standalone mode with 14 executors, 10GB memory per executor and 5 cores per executor. HDFS is configured for data storage. The master node is dedicated to run the HDFS NameNode and the Spark Master. We set up 50 minutes for the timeout of all experiments.

All test datasets were produced by a data generation script of the Purdue MapReduce Benchmarks Suite [83], called "PUMA". The maximum number of columns in the datasets is 39. Each column is separated by a comma and each field contains 19 characters.

We used four test datasets. These tests have the different sizes, namely, 1GB, 2GB, 5GB, and 10GB. Table 2.4 summarizes the various dataset sizes used in our experiments.

Table 2.4 – Input datasets used in experiments

Size (GB)	1GB	2GB	5GB	10GB
Cardinalities (records)	2 683 526	5 367 222	13 419 624	26 841 213

All the datasets are saved in the same text file format.

2.4.2 Fuzzy Self Join Evaluation

Experimental protocol

The experimental evaluations in this section perform fuzzy self joins using Hamming Distance over the last 6 characters of the column 36 (fixed-length: $b = 6$, alphabet:

$\Sigma = 10$). The approaches are performed in our experiments include: Naive, BH1, BF-BH1, Splitting, BF-Ball-Splits (BFBS). The following join query is used.

$$Q_1 = S \bowtie_{S.36 \approx S.36} S$$

Table 2.5 summarizes the input dataset keys used in our experiments.

To compare the scalability and the performance of the different algorithms, we conducted two types of tests.

— First, we investigated the scalability of all methods to compute a fuzzy self joins of the datasets $2GB$ (Exp-1) and $10GB$ (Exp-2) in Table 2.4 by increasing the Hamming distance threshold from 0 to 4 .
Exp-1: $Q_1 = 2GB \bowtie_\tau 2GB, \tau = 0, 1, 2, 3, 4$
Exp-2: $Q_1 = 10GB \bowtie_\tau 10GB, \tau = 0, 1, 2, 3, 4$

— Second, we investigated the scalability of the algorithms by increasing the size of the datasets in Table 2.4 with the Hamming distance threshold of fuzzy self joins is 3 (Exp-3).
Exp-3: $Q_1 = S \bowtie_\tau S, \tau = 3, S = 1GB, 2GB, 5GB, 10GB$

For each experiment, we run it 4 times and get the average result. Subsequently, we show results of experiments, and summarize our conclusions for each algorithm.

Table 2.5 – Summary of input keys of datasets used in fuzzy self joins experiments

Dataset	Column 36	
	Number of distinct keys	Total of keys
1GB	3 780	20 681
2GB	4 553	41 711
5GB	5 522	103 519
10GB	6 170	206 868

In addition, to execute these experimental evaluations, we use the constant $J = 6$ for the Naive algorithm and the parameters of Bloom filters as Table 2.6. The probability of a false positive f of $BF(S)$ is

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

We use the same parameters: the length of the Bloom filter $m = 2147483637$, the number of hash functions $k = 8$ for all experiments. However, we can determine optimal parameters for filters in practice to reduce computational overhead and memory cost.

Table 2.6 – Parameters of filters used in experiments. m: the length of the Bloom filter, n: the number of elements being filtered, k: the number of hash functions

Experiment	Dataset	$f_{BF(S)}$	n	k	m
Exp-1	2GB	10^{-38}	4 553	8	2 147 483 637
Exp-2	10GB	8.10^{-38}	6 170	8	2 147 483 637
Exp-3	1GB	2.10^{-39}	3 780	8	2 147 483 637
	2GB	10^{-38}	4 553	8	2 147 483 637
	5GB	3.10^{-38}	5 522	8	2 147 483 637
	10GB	8.10^{-38}	6 170	8	2 147 483 637

Evaluation Of Approaches

As mentioned above, to compare the scalability and the performance of the different approaches, we conducted two types of tests: (1) increasing threshold T, (2) increasing dataset size. Besides, for comparing the efficiency of the join algorithms, we are especially interested in three main aspects for each algorithm evaluation. They include the number of intermediate tuples generated (i.e. shuffle read), the execution time, and the output. The results of our experimental evaluations are described as follows.

Increasing threshold τ.

First, it is important to focus on comparing the amount of intermediate data listed in Table 2.7 and Table 2.8. We mark the enforcement of the timeout by the letter "T" and the out of memory by "OM".

Table 2.7 – Exp-1 - Intermediate data of fuzzy self joins of size 2GB on various thresholds

Algorithm	0	1	2	3	4
Naive	250 266	250 266	250 266	250 266	250 266
BH1	41 711	1 044 380	23 462 057	OM	OM
BFBH1	41 711	50 259	220 627	2 043 918	13 016 781
Splitting	41 711	83 422	125 133	166 844	208 555
BFBS	41 711	46 775	112 327	166 791	208 555

Table 2.8 – Exp-2 - Intermediate data of fuzzy self joins of size 10GB on various thresholds

Algorithm	0	1	2	3
Naive	T	T	T	T
BH1	206 868	5 177 006	OM	OM
BFBH1	206 868	262 956	1 399 853	13 604 835
Splitting	206 868	413 736	620 604	827 472
BFBS	206 868	239 772	585 586	827 418

The intermediate data is a decisive factor that affects the total execution time of

the algorithms. The intermediate data of the Naive fuzzy joins is not affected by the threshold. With $J = 6$ each record is copied 6 times. Thus, the number of records of dataset is increased 6 times. On the dataset 2GB, it is always a high constant 250266. However, the Naive fuzzy joins on 10GB runs into timeouts.

With the threshold $\tau = 0$, the algorithms are considered as the general equi-joins while they are performed as the fuzzy joins. In this case, the filters do not affect the amount of intermediate data. Hence, the numbers of intermediate tuples are equal for all approaches.

The worst case of these experiments is BH1. This result is reasonable because the ball calculation depends greatly on parameter thresholds τ. It even leads to the over memories on the threshold 3,4 for Exp-1 or 2,3 for Exp-2.

Besides, the Bloom filter helps to eliminate almost all redundant data for the algorithm BF-BH1 lead to the better result. Thus, BFBH1 executions do not turn into over memories in these experiments. The intermediate data of BFBS is also better than Splitting. This result proves that the application of the filter is effective. The Splitting algorithms generate less intermediate tuples than BH algorithms. For larger thresholds, Splitting generates more splits and shorter splits. Shorter splits lead to a greater probability of occurrence. Therefore, the filtering efficiency of BFBS tends to decrease with a larger threshold. For example, in these experiments, with the thresholds 3 and 4, the lengths of splits are {2,2,1,1} and {2,1,1,1,1,1} respectively. The number of intermediate data of BFBS is approximated with Splitting in the thresholds 3 and 4.

The results of these experiments are illustrated in the Figure 2.11.

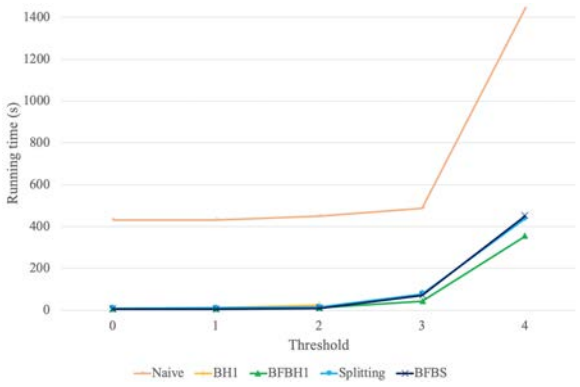
Table 2.9 – Running time of the fuzzy self join approaches on 2GB in various thresholds

Approach	0		1		2		3		4	
	preproc.	join	preproc.	join	preproc.	join	preproc.	join	preproc.	join
Naive		432		432		450		486		1440
BH1		9		10		25		T		T
BFBH1	5	6	5	7	5	10	5	42	5	354
Splitting		9		10		15		78		438
BFBS	5	6	5	6	5	9	5	72	5	450

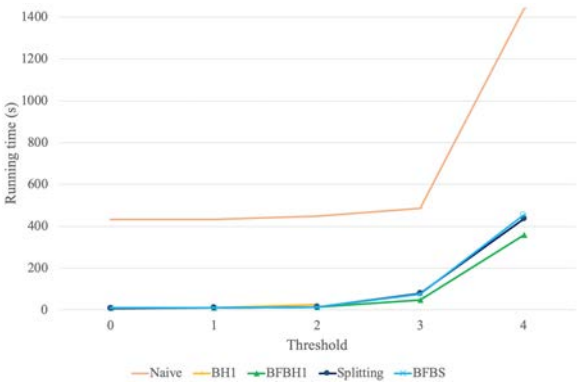
Next, we evaluate the efficiency of these fuzzy self join algorithms by comparing their total execution time. Table 2.9 and 2.10 identify in detail the execution time of the preprocessing stage and the join stage for the fuzzy self join algorithms.

With $J = 6$, the number of records of dataset is increased 6 times. Each pair of results have to be verified. Thus, Naive is always the slowest approach.

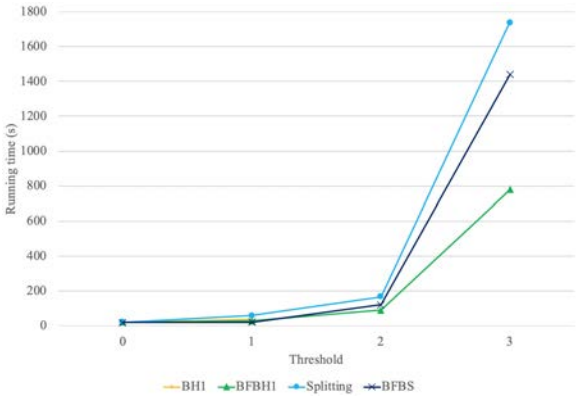
The preprocessing time of filters based approaches is the time to build $BF(S)$, 5s and



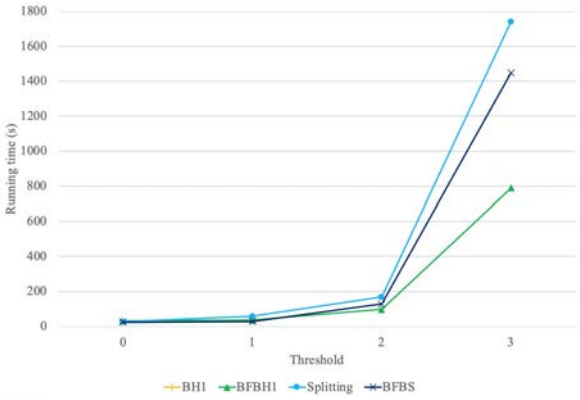
(a) Joining time on 2GB



(b) Total execution time on 2GB



(c) Joining time on 10GB

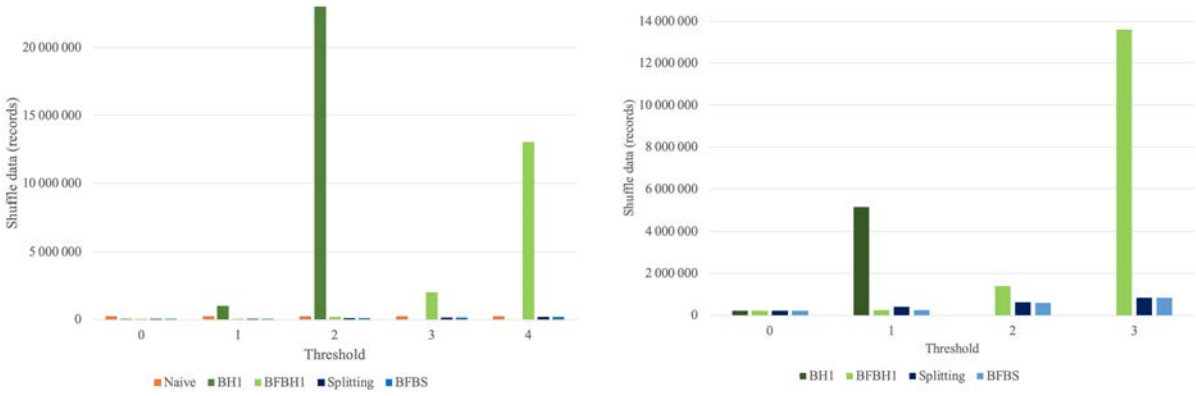


(d) Total execution time on 10GB

Figure 2.10 – Exp-1,2: Running time of fuzzy self joins approaches in various thresholds

9s for 2GB and 10GB respectively. On the small dataset (2GB) and the small threshold (1,2), the execution of the Splitting and BFBS is faster than BH1 and BFBH1 due to less intermediate data. However, on the large dataset or the larger threshold, the Splitting and BFBS is slower than BH1 and BF-BH1 because of the distance verification at joining stage and the duplicate output. Beside that, for a small size dataset 2GB and a small threshold 0,1,2, the filter efficiency is not clear because of the influence of the preprocessing time. For the dataset 10GB, even with the preprocessing time, the joining time and the total execution time of filters based approaches are always better. The Figure 2.10 demonstrates that the BF-BH1 is the best solution about running time.

Finally, we should compare their output result. This output shows the duplication of algorithms. The results of the total output are presented in Table 2.11, Table 2.12 and Figure 2.12. The Splitting produces a large number of duplicate results. The BFBS partially improves this amount. The BH1, BF-BH1 give the exact answers, without duplication.



(a) Exp-1 (b) Exp-2

Figure 2.11 – Intermediate data of fuzzy self joins approaches in various thresholds

Table 2.10 – Running time of the fuzzy self join approaches on 10GB in various thresholds

Approach	0		1		2		3	
	preproc.	join	preproc.	join	preproc.	join	preproc.	join
Naive		T		T		T		T
BH1		22		37		OM		OM
BFBH1	9	22	9	27	9	90	9	780
Splitting		22		58		168		1740
BFBS	9	19	9	21	9	120	9	1440

Increasing Dataset Size.

We investigated the scalability of the algorithms by increasing the size of the datasets. First, we consider the total amount of intermediate data generated by each fuzzy self join algorithm in the threshold $\tau = 3$ for various datasets as in Table 2.13 and Figure 2.13.

The intermediate data has significant overheads involving the communication costs. Naive is not shown in this experiment because its execution cannot finish from the test of 5GB . BH1 is always turned into over memory on the threshold $\tau = 3$. BFBS does not improve much over Splitting about intermediate data. BF-BH1 is the winner for filtering. This means that its redundant data is the least.

Next, we examine the total output of the fuzzy big join algorithms presented in Table 2.14 and Figure 2.14. Naive and Splitting produce a large number of duplicate results. BFBS partially improves this amount. BH1, BF-BH1 give the exact answers, without duplication.

Lastly, the running times of the Exp-3 are demonstrated in the Table 2.15.

Figure 2.15 presents the total execution time of the fuzzy self join using the different

Table 2.11 – Output results of fuzzy self join algorithms on 2GB in various threshold

Approach	0	1	2	3	4
Naive	2 892 882	3 014 344	5 967 271	36 610 447	220 853 437
BH1	482 147	554 899	2 143 631	OM	OM
BFBH1	482 147	554 899	2 143 631	18 719 562	119 394 407
Splitting	482 147	1 037 046	3 513 634	28 671 501	184 892 802
BFBS	482 147	591 441	3 173 038	28 664 535	184 892 802

Table 2.12 – Output results of fuzzy self join algorithms on 10GB in various threshold

Approach	0	1	2	3
Naive	T	T	T	T
BH1	11 885 491	13 659 772	OM	OM
BFBH1	11 885 491	13 659 772	53 057 520	460 714 501
Splitting	11 885 491	25 545 263	86 781 567	706 374 942
BFBS	11 885 491	14 883 015	82 022 287	706 334 923

algorithms from $1GB$ to $10GB$. With the small size dataset the execution time of these approaches is not much different. Beside Naive, Splitting is the slowest in this case because of its redundant intermediate data shuffle, verification and also duplication, even its data shuffle is less than BH algorithms. When the dataset size increases, Splitting and BFBS are the most severely affected algorithms. Their execution time increases rapidly. Besides, BF-BH1 and BF-Ball-Splits demonstrate again the efficiency of filtering. Finally, BF-BH1 is also the winner for the execution time in this experiment.

2.4.3 Fuzzy Two-way Join Evaluation

Experimental protocol

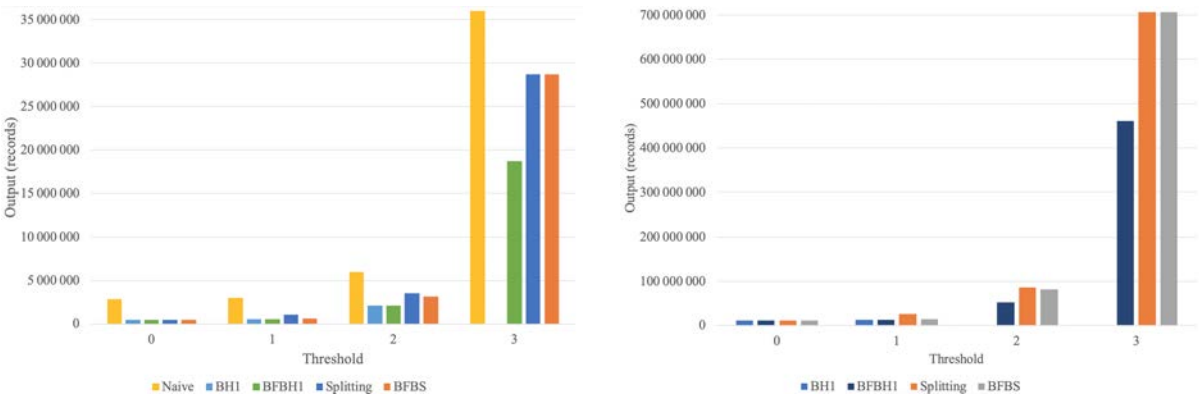
The experimental evaluations in this section perform fuzzy two-way joins using Hamming Distance over the last 6 characters of the column 36 (fixed-length: $b = 6$, alphabet: $\Sigma = 10$). The approaches are performed in our experiments include: Naive, BH1, BF-BH1, Splitting, BF-Ball-Splits (BFBS), IBF-Splits (IBFS). The following join query is used.

$$Q_2 = S \bowtie_{S.36 \approx T.38} T$$

Table 2.16 summarizes the input dataset keys used in our experiments.

Similar to the experiments above, we conducted two types of tests.

— First, we investigated the scalability of all methods to compute a fuzzy two-way



(a) Exp-1 (b) Exp-2

Figure 2.12 – Output result of fuzzy self joins approaches in various thresholds

Table 2.13 – Intermediate data of fuzzy self joins in the threshold $\tau = 3$ for various datasets

Approach	1GB	2GB	5GB	10GB
Naive	124 086	250 266	T	T
BH1	OM	OM	OM	OM
BFBH1	846 746	2 043 918	6 111 074	13 604 835
Splitting	82 724	166 844	414 076	827 472
BFBS	82 579	166 791	414 057	827 418

joins of the datasets $10GB \bowtie 10GB$ (Exp-4) by increasing the Hamming distance threshold from 0 to 4.

Exp-4: $Q_2 = S \bowtie_\tau T, S, T = 10GB, \tau = 0, 1, 2, 3, 4$

— Second, we investigated the scalability of the algorithms by increasing the size of the datasets with the Hamming distance threshold of fuzzy two-way joins is 3 (Exp-5).

Exp-5: $Q_2 = S \bowtie_\tau T, S, T = 1GB, 2GB, 5GB, 10GB, \tau = 3$

For each experiment, we run it 4 times and get the average result. Subsequently, we show results of experiments, and summarize our conclusions for each algorithm.

In addition, to execute these experimental evaluations, we use $J = 6$ for the Naive algorithm. The Bloom filters are built on the join key set of the column 36 of dataset S. Hence, S is cached in memory after the preprocessing stage. The parameters of Bloom filters and the probability of a false positive f of $BF(S)$ are used as Table 2.6 above.

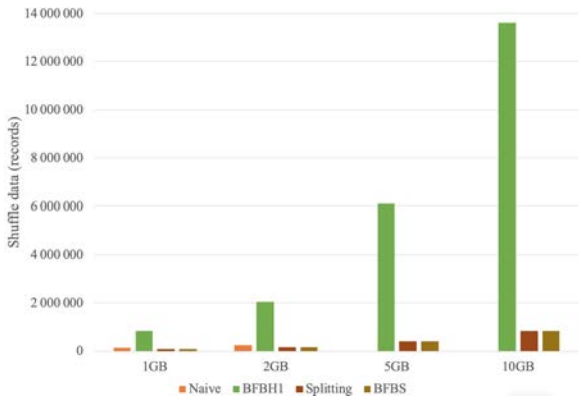


Figure 2.13 – Intermediate data of Exp-3

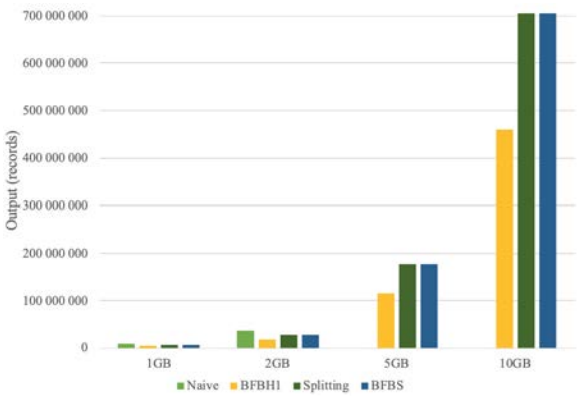


Figure 2.14 – Total output results of Exp-3

Table 2.14 – Output results of fuzzy big join approaches in the threshold $\tau = 3$ for various datasets

Approach	1GB	2GB	5GB	10GB
Naive	8 997 954	36 610 447	T	T
BH1	OM	OM	OM	OM
BFBH1	4 596 714	18 719 562	115 452 868	460 714 501
Splitting	7 040 313	28 671 501	176 911 974	706 374 942
BFBS	7 028 642	28 664 535	176 904 982	706 334 923

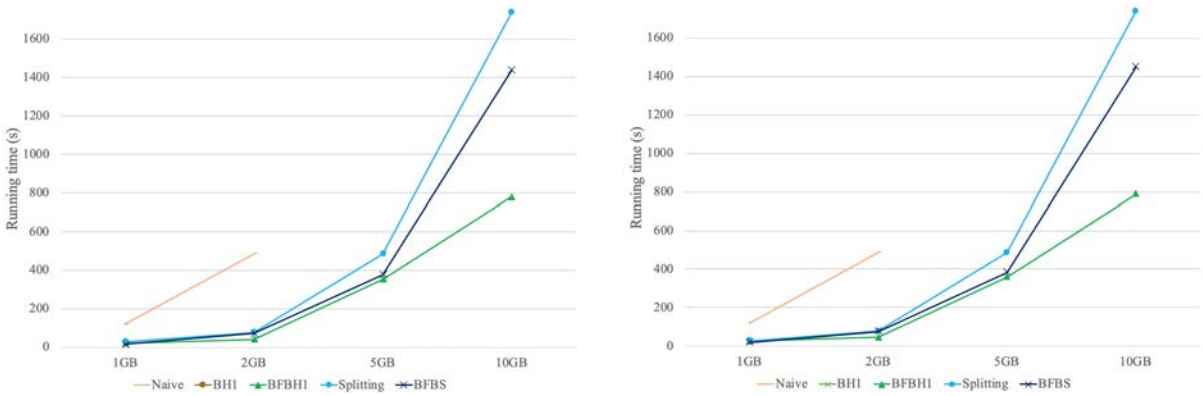
Evaluation Of Approaches

Figure 2.16 presents the intermediate data for the experiments Exp-4 and Exp-5. BH1 generates a large amount of shuffle records. BF-BH1 shows the effectiveness of the filter when it removes a large amount of redundant data. Splitting and IBFS have the approximate amount of data transferred. BFBS prunes out redundant intermediate data clearly on small datasets and small thresholds.

The results of the total outputs for approaches are shown in Figure 2.17. Splitting, BFBS, IBFS generate duplicate results. BF-BH1 gives the exact answers without the

Table 2.15 – Running times of fuzzy self joins in the threshold $\tau = 3$ on various datasets

Algorithm	1GB		2GB		5GB		10GB	
	preproc.	join	preproc.	join	preproc.	join	preproc.	join
Naive		120		486		T		T
BH1		OM		OM		OM		OM
BFBH1	4	22	5	42	6	354	9	780
Splitting		29		78		486		1740
BFBS	4	16	5	72	6	378	9	1440



(a) Joining time

(b) Total execution time

Figure 2.15 – Running time of fuzzy self join approaches in various datasets in the threshold $\tau = 3$

Table 2.16 – Input dataset keys used in fuzzy two-way joins experiments

	Column 38		Column 36	
	# distinct keys	Total of keys	# distinct keys	Total of keys
1GB	2334	7456	3780	20681
2GB	3111	14991	4553	41711
5GB	4183	37371	5522	103519
10GB	4866	74670	6170	206868

duplication.

Figure 2.18 reports the running times of the algorithms for various thresholds and various datasets. Naive timed out on datasets of size 10GB. BH1 is also out of memory from the threshold 3. We observe that the running time of Splitting increases rapidly along thresholds and dataset sizes. IBFS is almost similar to Splitting, even slower than Splitting on large datasets and large thresholds. BFBS improves better. BF-BH1 is the winner in these experiments.

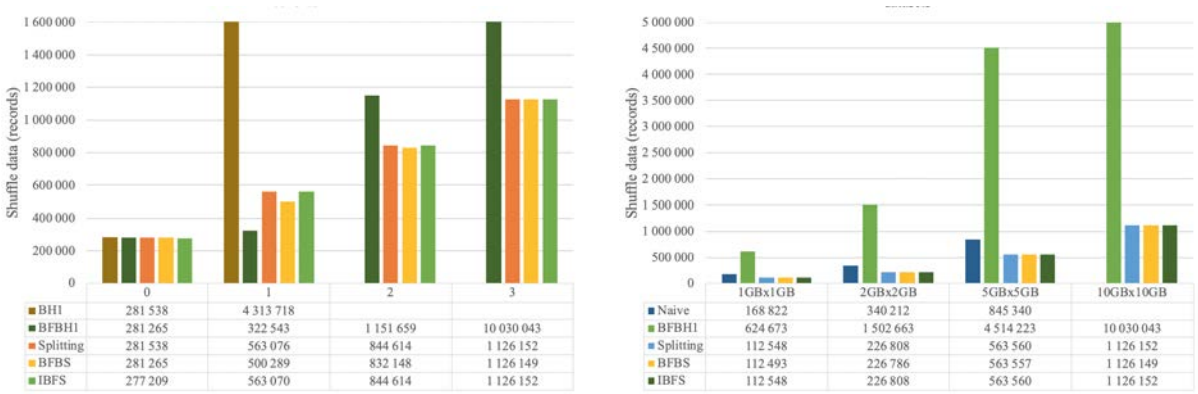
We summary some conclusions of the experiments in this section.

— Naive is the lowest-performing algorithm.

— BH1 is sensitive to large thresholds or large datasets, where the memory becomes a bottleneck and limits scalability.

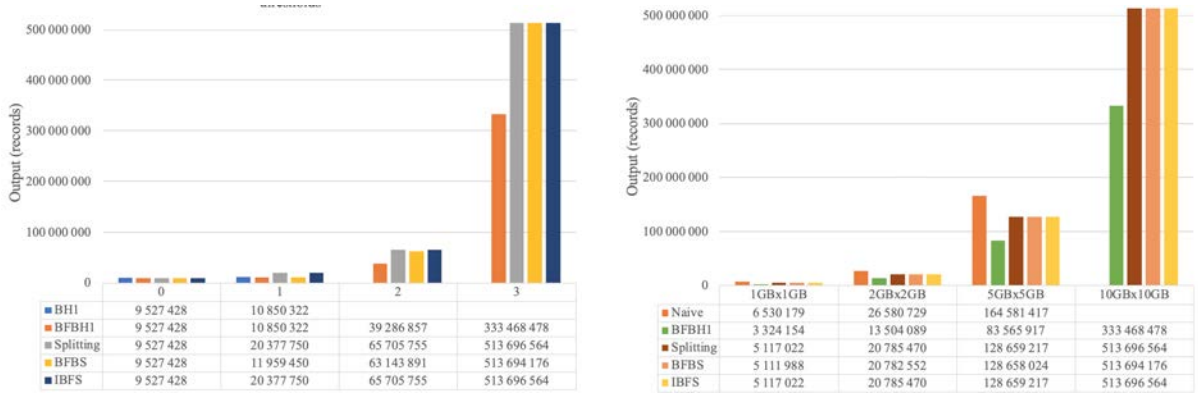
— Filters based approaches better cope when the size or the threshold increase. Our experiments prove that filtering can significantly improve the execution times in site of the preprocessing stage.

— Splitting, Bloom filter based Splitting are also sensitive to large thresholds. For



(a) Exp-4 (b) Exp-5

Figure 2.16 – Intermediate data of Exp-4 and Exp-5



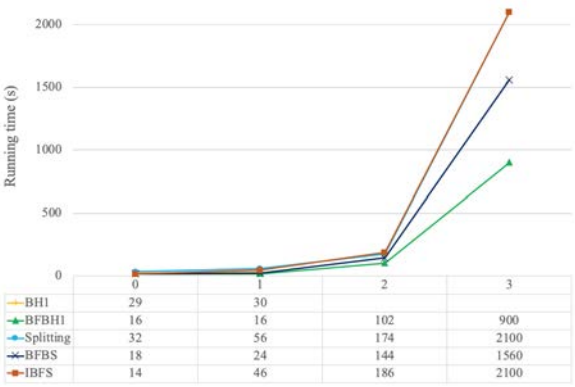
(a) Exp-4 (b) Exp-5

Figure 2.17 – Total output results of Exp-4 and Exp-5

larger threshold, Splitting generates more splits and shorter splits. Shorter splits lead to a greater probability of occurrence. Therefore, filtering efficiency of BFBS tends to decrease with a larger threshold. Still, IBF is inefficient for Splitting.

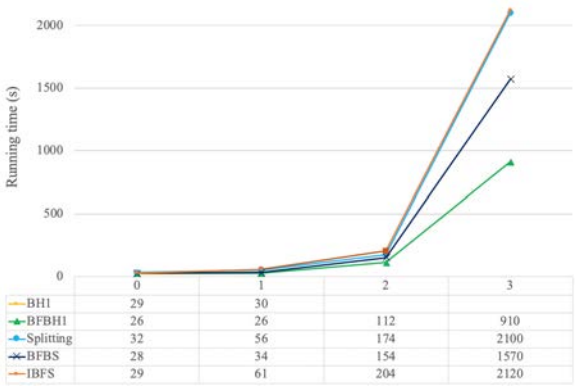
— Splitting and Bloom filters based Splitting approaches generate less intermediate data than Ball Hashing approaches. However, their running times are slower because of their verification at the joining stage and their duplicate outputs.

— Overall, BF-BH1 is the best choice due to their least redundant data and without the duplication.



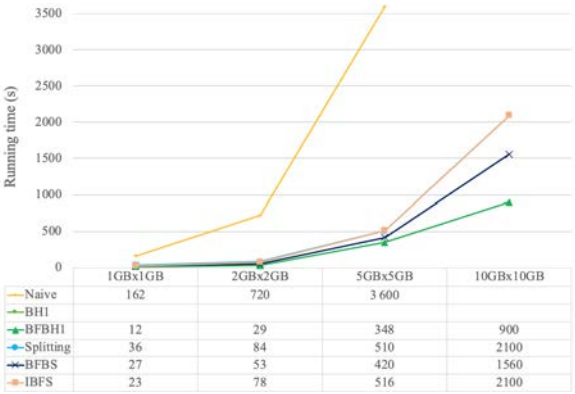
	0	1	2	3
BH1	29	30		
BFBH1	16	16	102	900
Splitting	32	56	174	2100
BFBS	18	24	144	1560
IBFS	14	46	186	2100

(a) Exp-4: Joining time



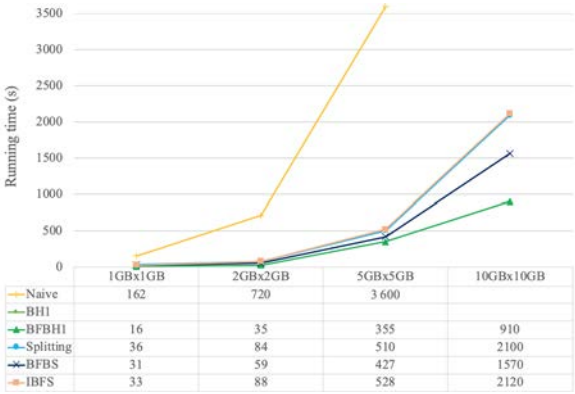
	0	1	2	3
BH1	29	30		
BFBH1	26	26	112	910
Splitting	32	56	174	2100
BFBS	28	34	154	1570
IBFS	29	61	204	2120

(b) Exp-4: Total execution time



	1GBx1GB	2GBx2GB	5GBx5GB	10GBx10GB
Naive	162	720	3 600	
BH1				
BFBH1	12	29	348	900
Splitting	36	84	510	2100
BFBS	27	53	420	1560
IBFS	23	78	516	2100

(c) Exp-5: Joining time



	1GBx1GB	2GBx2GB	5GBx5GB	10GBx10GB
Naive	162	720	3 600	
BH1				
BFBH1	16	35	355	910
Splitting	36	84	510	2100
BFBS	31	59	427	1570
IBFS	33	88	528	2120

(d) Exp-5: Total execution time

Figure 2.18 – Running time of Exp-4 and Exp-5

2.5 Summary

In this chapter, we study theoretical details for the fuzzy big 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. Through experiments, we demonstrate that the proposed solutions bring significant efficiency in comparison with current available solutions. Performing in Spark, we exploit the capacities of Spark such as distributed and parallel processing, iterative processing, caching mechanism, and fast computing on memory. Our optimizations can be applied for Edit distance, Jaccard distance and further extended for variable length strings [5].

This work has led to one publication [104] in Proceedings of the 2018 IEEE International Conference on Fuzzy Systems (FUZZIEEE 2018).

Chapter 3

Fuzzy Filters And Fuzzy Big Joins Optimization

From our study of fuzzy big joins in MapReduce in the previous chapter, we observed various limitations of previous works. We were interested in using Bloom Filters [20] and Intersection Filter [90] to remove irrelevant data as soon as possible to reduce data transfers and workload on different machines. However, these improvements imply redundant calculations that can affect the performance of fuzzy joins. Moreover, they filter only one input dataset in fuzzy two-way and fuzzy multi-way joins. In this chapter, we want to optimize the performance of fuzzy big joins by addressing such these limitations. The main contributions of our works are:

- Fuzzy filter (FF) and intersection fuzzy filter (IFF) structure for detecting if an element is close to any members in a set(s). Moreover, FF, IFF can determine which records in the set(s) are real similarities of this element.
- Large scale FF based fuzzy join algorithm to avoid useless re-computation.
- Optimizations for fuzzy two-way and multi-way joins based on IFF.
- Theoretical analysis of various fuzzy join algorithms in MapReduce and their cost comparison in a $M - C - R$ cost model.
- Experimental evaluations of proposed approaches.

The remaining part of this chapter is organized as follows. We discuss previous work and propose our solution for optimizing fuzzy big joins in Section 3.1. Some definitions and notations are also introduced. Each remaining section of this chapter therefore highlights the contribution of our work. Section 3.2 provides fuzzy filters to answer which similarities of given element(s) are in set(s). The existing problems, concepts, and design details for the fuzzy filter and the intersection fuzzy filter are described. Especially, the false difference probability that affects the filtering performance is also considered and analyzed thoroughly. Next, Section 3.3 presents an optimization for the fuzzy self joins as well as the fuzzy multi-way join algorithms. The general algorithm and its implementation in

Spark are detailed in this section. We compare the proposal to the previous approach through a cost model, and show the advantages of our approach in Section 3.4. The evaluation environments, experimental protocols and experiments are reported in Section 3.5. Finally, we conclude our contributions in Section 3.6.

3.1 Previous Works

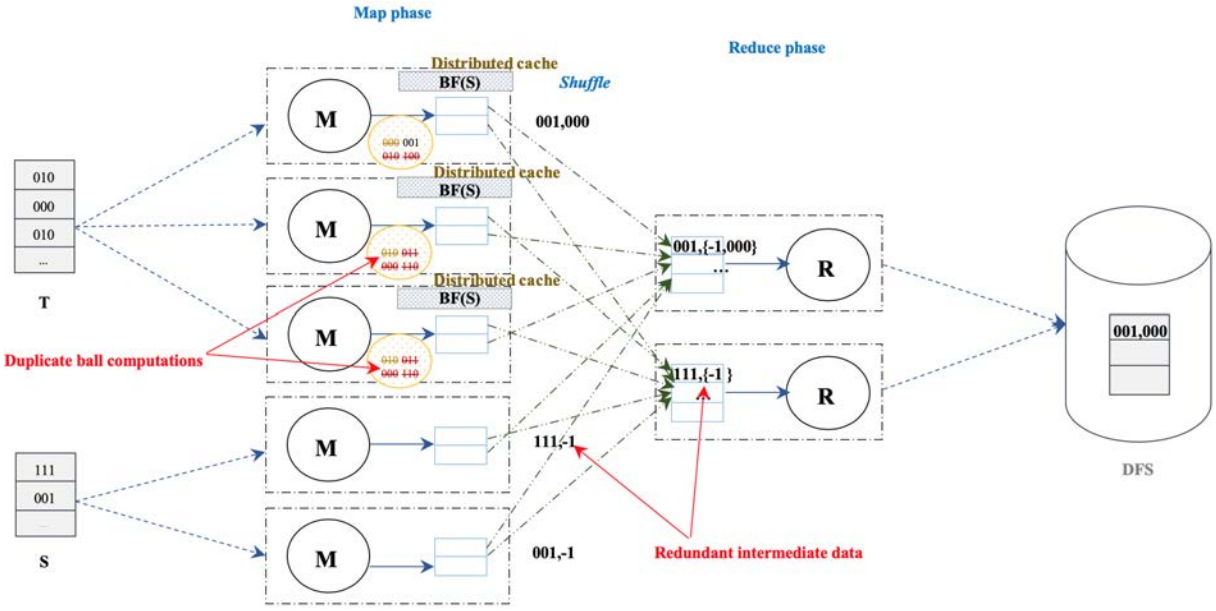


Figure 3.1 – BF-BH1 limitations

By the theoretical analysis and experiments in the previous chapter, applying filters will improve the processing costs of fuzzy big joins. However, Ball of radius r computation is sensitive to distance. The number of elements in B_r increases with the distance d. [66] has shown the slowness of the Ball Hashing algorithm without filters. Besides, 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. On the other hand, the standard Bloom filter in BF-BH approach only has the ability to remove non-joining tuples from one of the input datasets instead of both. As a result, there remains a large amount of non-joining data from other input dataset(s) sent to the reducers for the join processing.

Figure 3.1 describes an illustration of these limitations of the Ball Hashing algorithm using the Bloom filter in MapReduce.

— A Bloom filter $BF(S)$ is first built for an input dataset S on a join key column and is delivered across all the mappers.
— Each mapper receives tuples from T, it generates balls of their join keys, eliminates elements not in $BF(S)$ and emits key/value pairs for the remaining elements. In this example, tuples of duplicate keys (e.g. 010) are calculated balls multiple times.
— Each mapper receives tuples from S, it maps key/value pairs without filtering. Hence, non joining tuples (e.g. 111) are also emitted. This redundancy considerably increases associated overheads in cases of multi-way joins and iterative joins.
— Then, the pairs are passed to corresponding reducers to be joined.

Table 3.1 – List of notations

Notation	**Description**
M	Total computation cost (map or preprocessing) 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
Σ	Finite alphabet of symbols
$S, \|S\|$	Input dataset S and its cardinality
τ, d, r	Pre-specified threshold of distance and radius of Ball
s, t, b	A string s or t and its length
B_r	Ball of radius r
k	Number of hash functions
K, n	Number of reducers
D	Size of intermediate data for shuffle
$\setminus$	Difference operator
$\cap$	Intersection operator
$\cup$	Union operator
$FF(S)$	General Fuzzy Filter built for a set S
$FF(S, r)$	Fuzzy Filter built for a set S with a distance r
$IFF(S \bowtie_r T)$	General Intersection Fuzzy Filter built for $S \bowtie_r T$
$EIFF$	Extended Intersection Fuzzy Filter
m	Length of fuzzy filters
$f_{FF(S)}, f_{IFF(S \bowtie_r T)}$	False positive probability of $FF(S), IFF(S \bowtie_r T)$

For these problems, we need to build new filter types representing the similarities of the input datasets to be able to filter out non-joining data in one or both of input datasets, to avoid excess calculations.

The BF, IBF and its variations [86, 32, 28, 68, 67, 48] cannot solve these problems.

SuRF [119] is a fast and compact filter that provides an 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, leading to inefficient and high costs. In our study, fuzzy filters have not been proposed yet. Therefore, we propose Fuzzy Filter, Intersection Fuzzy Filter and these filters based approaches to optimize fuzzy big joins. The complex joins can take advantages from our proposed filters.

We supply notations used in this research as in Table 3.1.

3.2 Fuzzy Filters

This section shows the approach to build the fuzzy filter with the criteria: small size, stored in-memory, giving quick answers, no false negative probability. With a query y, a fuzzy filter has to give a quick test:

$\exists x \in S, d(x, y) \leq r$? Which x?

or $\exists x \in S \cap B_r(y)$? Which x?

As illustrated in the Figure 3.2, we consider queries:

— $y_1 \rightarrow 0$/NO, because $B_r(y_1) \cap S = \emptyset$

— $y_2 \rightarrow 1$/YES, because $B_r(y_2) \cap S = \{x_5, x_6\}$

— $y_3 \rightarrow 1$/YES, because $B_r(y_3) \cap S = \{x_7\}$

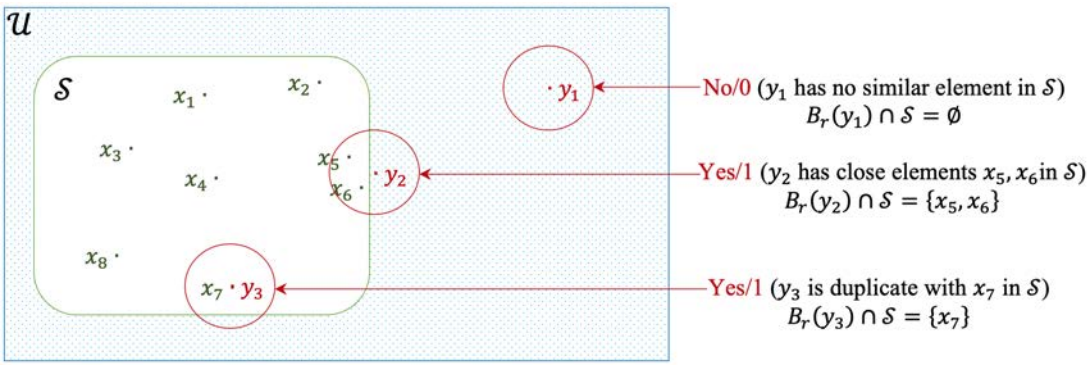


Figure 3.2 – Fuzzy filter modeling

As shown in Figure 3.2, we can see that the output are elements that belong to the intersection of S and $B_r(y)$:

$$FF(y) = B_r(y) \cap S$$

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.

3.2.1 Fuzzy Filter

Definition 3.1 (Fuzzy Filter)**.** *A Fuzzy Filter $FF(S, \tau)$ is a probabilistic data structure designed to represent the similarities of a set with a false positive probability. It is defined by a similarity function Sim and a threshold τ. The result of query y in $FF(S, \tau)$ is $FF(y, S, \tau) = \{x \mid x \in S, Sim(x, y) \geq \tau\}$.*

The idea begins with finding the intersection between dataset and balls. We propose a fuzzy filter structure as follows. A 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 the Figure 3.3

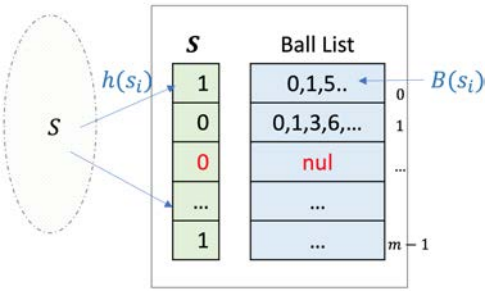


Figure 3.3 – FF(S) structure

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 m bit Bloom Filters, each one stores its real ball elements $BF(B(s_i) \cap S)$. We assume that m is large enough, with a perfect hash function, it will not have duplicates balls at the same hashed position. The problem of false positive will be discussed later. The Fuzzy Filter accepts an input y and returns outputs that is one of these possibilities as follows:

— NO - Null: y is NOT in S and it has no any similar elements in S.

— NO - list of $\{x\}$: y is NOT in S and $\{x\}$ may be its similar elements in S.

— YES - list of $\{x\}$: y may be an element in S and $\{x\}$ may be also its similar elements in S.

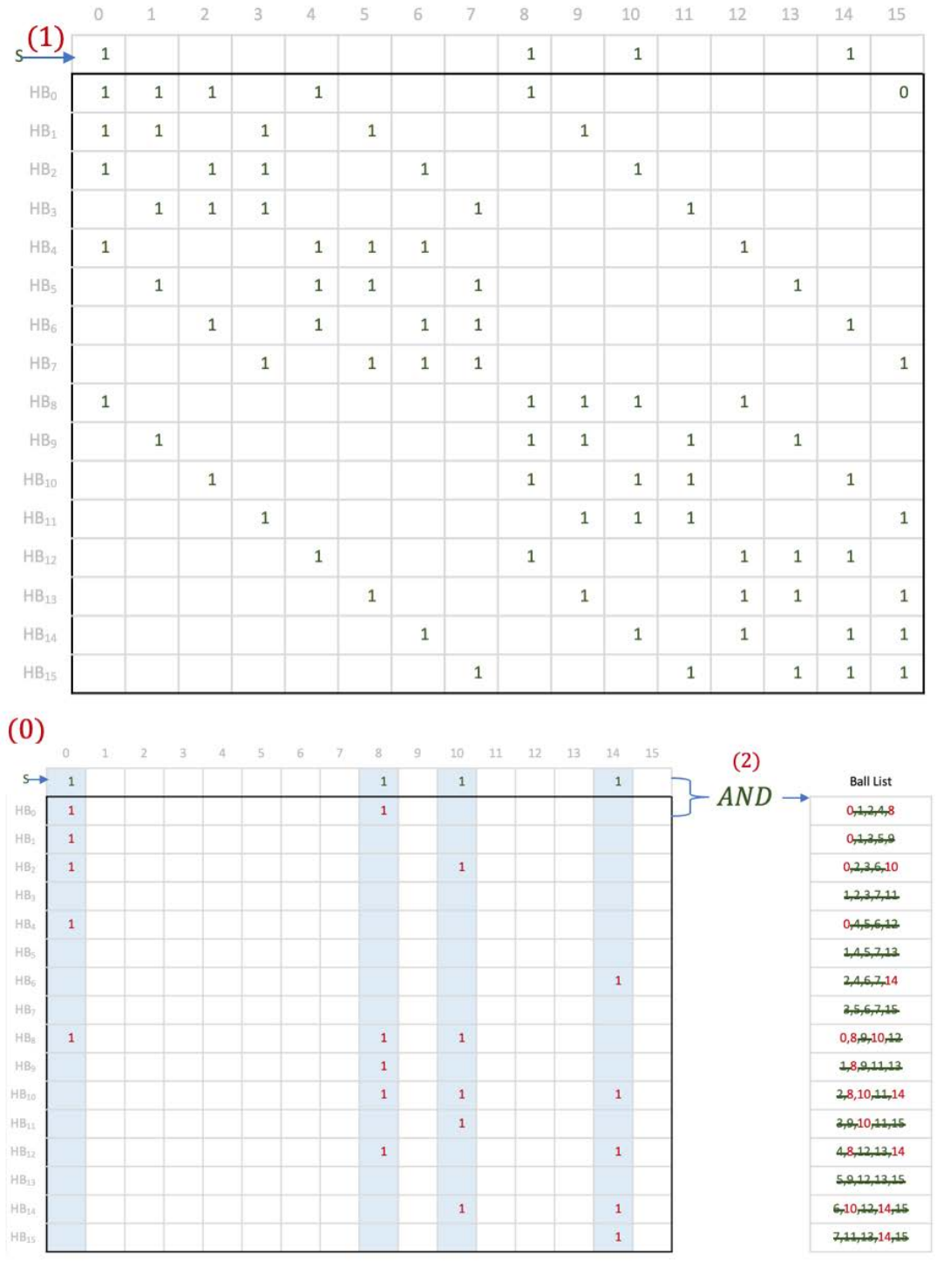


Figure 3.4 – Example of building FF with $b = 4, r = 1, S = (0000, 1010, 1110, 1000)$

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

— (0) Hash each ball to a bit array of length m. In Σ space, it has $|\Sigma|^b$ ball of radius d, with b is length of each element. In general, $m = |\Sigma|^b$. While each ball has $|B_r|$ elements, the cost of this step is $C_{(0)} = |\Sigma|^b|B_r|$.
For example, in Hamming space of b-bit strings, there exist 2^b balls, each ball has about $b^r/r!$ elements. This step has the cost $C_{(0)} \approx 2^b b^r/r!$
— (1) Hash S to a bit array of length m. $C_{(1)} = |S|$
— (2) Ball list is computed by the intersection of $BF(S)$ and $BF(B(s_i))$.
$BallList_i = S \cap B(s_i), 1 \leq i \leq m$
$BF(S \cap B(s_i))$ is formed by intersecting two standard Bloom filters $BF(S)$ and $BF(B(s_i))$ with the bitwise AND operator. This can be expressed by the following form:
$BallList_i = IBF(BF(S \cap B(s_i))) = BF(S)\,\&\,BF(B(s_i))$
It performs the bitwise AND operation between the two bit arrays $BF(S)$ and $BF(B(s_i))$ with the same size m. The $BallList_i$ is now an approximate representation of the similar elements of s_i in S.
The cost of step (2) is $C_{(2)} = |\Sigma|^b$
In Hamming space, $C_{(2)} = 2^b$
— The build cost for $FF(S)$ is

$$C_{FF(S)-build} = |\Sigma|^b|B_r| + |S| + |\Sigma|^b = (|B_r| + 1)|\Sigma|^b + |S|$$

In Hamming space,

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

For clarity, we consider an example of building FF with $b = 4, r = 1, S = (0000, 1010, 1110, 1000)$ illustrated in Figure 3.4

— (0) Hash 2^4 ball to bit array of length $m = 2^4$.
For example, Hamming ball of radius $r = 1$ of 0000 is hashed in $BF(B(0000)) = \{0, 1, 2, 4, 8\}$.
— (1) Hash S to bit array of length $m = 2^4$: $BF(S) = \{0, 8, 10, 14\}$
— (2) Ball list is computed by AND operations. For example,
$BallList_{0000} = BF(S)\,\&\,BF(B(0000)) = \{0, 8, 10, 14\}\,AND\,\{0, 1, 2, 4, 8\} = \{0, 8\}$

Consider the query examples:
— $y_1 = 0000 \xrightarrow{\text{h}(0000)=0} BF(S(0)) = 1, B_1(0000) = \{s_0, s_8\} = \{0000, 1000\}$

— $y_2 = 0111 \xrightarrow{\text{h(0111)=7}} BF(S(7)) = 0, B_1(0111) = \emptyset$
— $y_3 = 1011 \xrightarrow{\text{h(1011)=11}} BF(S(11)) = 0, B_1(1011) = \{s_{10}\} = \{1010\}$

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

With this design, when the Fuzzy Filter returns an answer "NO", the answer is always the correct response. Since each of the Bloom filters has the false positive probability, there exist "false" similar elements discovered by the Fuzzy Filter. $BF(S)$ covers S^* that is a small superset of S. $BF(B(s_i))$ covers $B^*(s_i)$ that is a little bigger than $B(s_i)$. Thus, $BF(S) \cap BF(B(s_i))$ includes $S^* \cap B^*(s_i)$. It may lead to false positives illustrated in Figure 3.5.

— $S^* = S \cup \{x_9, x_{10}\} \rightarrow FF(y_2) = \{x_5, x_6, x_9\}$
— $B^*(y_3) = B(y_3) \cup \{x_{11}, x_{12}\} \rightarrow FF(y_3) = \{x_5, x_{11}\}$

Therefore, an answer "YES" may be the incorrect response because x may be NOT a real similar element(s). It also means that the Fuzzy Filter returns "YES" answers with a false positive probability. As a result, the Fuzzy filter enables us to specify a superset of similar elements including the "YES" elements, and eliminate dissimilar elements that are the "NO" elements. Accordingly, we should minimize false positives for the Fuzzy Filter.

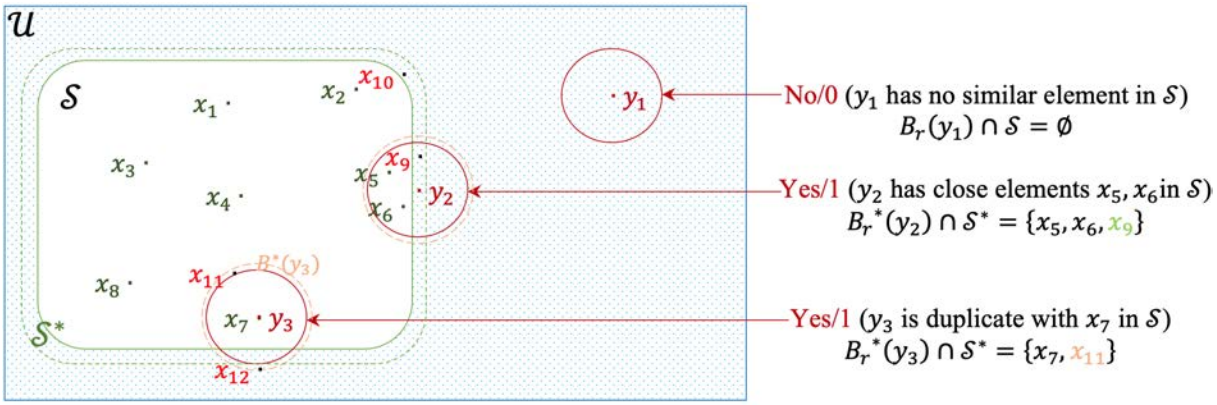


Figure 3.5 – FF modeling with false positives

3.2.2 Intersection Fuzzy Filters

Definition 3.2 (Intersection Fuzzy Filter)**.** *An Intersection Fuzzy Filter* $IFF(S, T, \tau)$ *is a probabilistic data structure designed to represent the similarities of sets with a false positive probability. It is defined by a similarity function Sim and a threshold* τ*. The result*

of query y of a set T to $IFF(S,T,\tau)$ is its similarities in the remaining set S and vice versa.

$$IFF(y,S,T,\tau) = \{x \mid x \in S, y \in T, Sim(x,y) \geq \tau\}$$

Inheriting the idea above, we propose the Intersection Fuzzy Filter (IFF) to apply to Fuzzy two-way join ($S \bowtie_r T$). The model of IFF is presented in Figure 3.6. Consider queries:

— $y_2 \rightarrow YES$, because $B_r(y_2) \cap T^* = \{x_1, x_4\}$
— $x_2 \rightarrow YES$, because $B_r(x_2) \cap S^* = \{y_1, x_3\}$
— $y_4 \rightarrow NO$, because $B_r(y_4) \cap S^* = \emptyset$
— $x_6 \rightarrow NO$, because $B_r(x_6) \cap T^* = \emptyset$

We observe that the similar elements of S in T are $\bigcup_{i=1}^{|S|}(B_r(s_i) \cap T^*)$; the similar elements of T in S are $\bigcup_{j=1}^{|T|}(B_r(t_j) \cap S^*)$; the dissimilar elements of two datasets are $(S \backslash T^*) \cup (T \backslash S^*)$.

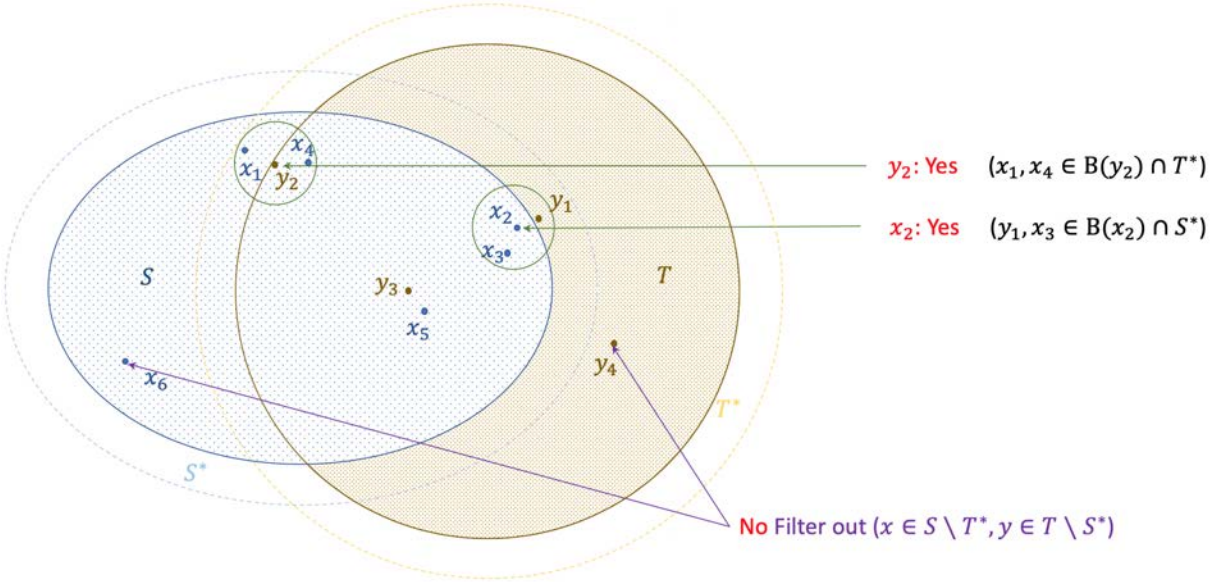


Figure 3.6 – Intersection Fuzzy Filters idea

From these above expressions, we can specify the set fuzzy intersection by eliminating all elements of the difference between the sets. Precisely, the Intersection Fuzzy Filter recognizes similar elements in the set S by the intersection of balls of T and $BF(S)$; similar elements in the set T by the intersection of balls of S and $BF(T)$. To achieve this work, we propose a structure of Intersection Fuzzy Filter illustrated as in Figure 3.7.

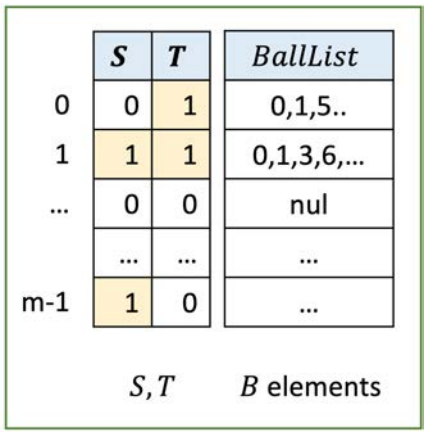


Figure 3.7 – $IFF(S \bowtie_r T)$

An Intersection Fuzzy Filter structure includes: A 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. Similarly, a m bit Bloom filter $BF(T)$ is also used to cover T. The ball list is an array of m bit Bloom Filters of size m ($m \times m$ bits matrix), each one stores its real ball(s) $BF(B(s_i) \cap T)/BF(B(t_i) \cap S)/BF((B(s_i) \cup B(t_i)) \cap (S \cup T))$. Let us recall the assumption that m is large enough, with a perfect hash function, it will not have duplicate balls at the same hashed position. The problem of false positive will be discussed later.

The Fuzzy Filter accepts an input z and returns outputs that is one of these possibilities as follows:

— NO - Null if
  - z is NOT in S, T
  - or z is in S but it has no any similar elements in T.
  - or z is in T but it has no any similar elements in S.

— YES - list of $\{x\}$: z may be an element in T and $\{x\}$ may be its similar elements in S.

— YES - list of $\{y\}$: z may be an element in S and $\{y\}$ may be its similar elements in T.

The build operation of the intersection fuzzy filter $IFF(S, T)$ is described as follows.

— (0) Hash each ball to a bit array of length m. In Σ space, it has $|\Sigma|^b$ ball of radius d, with b is length of each element. In general, $m = |\Sigma|^b$. While each ball has $|B_r|$ elements, the cost of this step is $C_{(0)} = |\Sigma|^b |B_r|$.

For example, in Hamming space of b-bit strings, there exist 2^b balls, each ball has about $b^r/r!$ elements. This step has the cost $C_{(0)} \approx 2^b b^r/r!$

— (1) Hash S to a bit array of length m.
T is also hashed to a bit array of length m.
$C_{(1)} = |S| + |T|$

— (2) Ball list is computed for each ball at the index $i, i = 0..m-1$ as follows.

- If $S_i = T_i = 0$, it means that element at the hashed position i belongs to neither S nor $T \rightarrow BallList$ at i is set to be empty.
- If $S_i = 1, T_i = 0$, it means that element at the hashed position i may belong S, but it does not belong $T \rightarrow BF$ at i is used to determine the similar elements of s_i in T. Thus, $BallList_i = BF(B(s_i) \,\&\, BF(T))$
- Similar, if $S_i = 0, T_i = 1$, it means that element at the hashed position i may belong T, but it does not belong $S \rightarrow BF$ at i is used to determine the similar elements of t_i in S. Hence, $BallList_i = BF(B(t_i) \,\&\, BF(S))$
- If in the case of $S_i = T_i = 1$, it means that element at the hashed position i may belong to both S and $T \rightarrow BF$ at i must store all similar elements of s_i and t_i in two datasets. Therefore, $BallList_i = BF_i \,\&\, (BF(S) \,|\, BF(T)))$
  It performs the bitwise operations between bit arrays with the same size m. The $BallList_i$ is now an approximate representation of the similar elements of s_i, t_i in datasets .
  The cost of step (2) is $C_{(2)} = |\Sigma|^b$
  In Hamming space, $C_{(2)} = 2^b$

— The build cost of IFF is similar to FF.

$$C_{IFF(S \bowtie_r T)-build} = |\Sigma|^b |B_r| + |S| + |T| + |\Sigma|^b = (|B_r| + 1)|\Sigma|^b + |S| + |T|$$

In Hamming space,

$$C_{IFF(S \bowtie_r T)-build} \approx \frac{b^d}{d!} 2^b + |S| + |T| + 2^b = (\frac{b^d}{d!} + 1)2^b + |S| + |T|$$

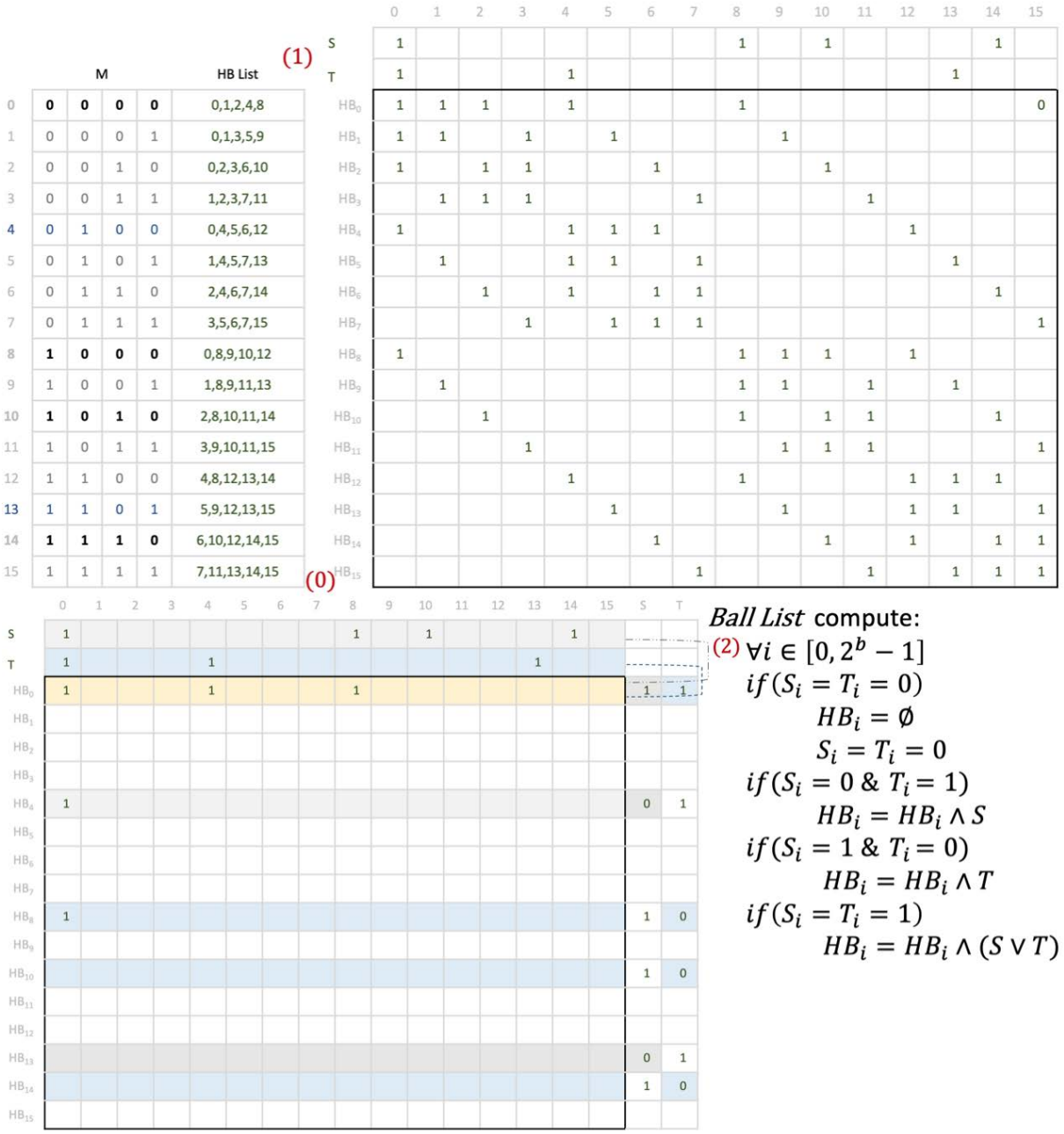


Figure 3.8 – Example of building IFF with $b = 4, r = 1, S = \{0000, 1010, 1110, 1000\}, T = \{0000, 0100, 1101\}$

We illustrate an example of building IFF in Figure 3.8. $S = \{0000, 1010, 1110, 1000\}, T = \{0000, 0100, 1101\}, b = 4, r = 1$

— (0) Hash each ball to a bit array of length $m = 2^b = 2^4$.

— (1) Hash S in $BF(S) = \{0, 8, 10, 14\}$

Hash T in $BF(T) = \{0, 4, 13\}$

— (2) Ball List computing:

- With $i = \{1, 2, 3, 5, 6, 7, 9, 11, 12, 15\}$, $S_i = T_i = 0$, thus, the balls at these hashed positions i are empty.
- With $i = 0$,$S_0 = T_0 = 1$, thus the ball at this position 0 is $HB_0 = BF(B(0000)) \,\&\, (BF(S) \,|\, BF(T)) = \{0, 1, 2, 4, 8\} \wedge (\{0, 8, 10, 14\} \vee \{0, 8, 10, 14\}) = \{0, 4, 8\}$
- With $i = 4$,$S_4 = 0, T_4 = 1$, thus the ball at this position 4 is $HB_4 = BF(B(0100)) \,\&\, BF(S)) = \{0, 4, 5, 6, 12\} \wedge \{0, 8, 10, 14\} = \{0\}$
- With $i = 8$,$S_8 = 1, T_8 = 0$, thus the ball at this position 8 is $HB_8 = BF(B(1000)) \,\&\, BF(T)) = \{0, 8, 9, 10, 12\} \wedge \{0, 4, 13\} = \{0\}$
- It performs similar operations with $i = 10, 13, 14$. But the results of these operations is empty. In other words, the elements at these hashed positions have no similar elements in the remaining dataset.

Consider the query examples:

— $s_0 = 0000 \xrightarrow{\text{h(0000)=0}} BF_{s_0} = BF_{t_0} = 1, B_1(0000) = B_1(0000) \wedge T = \{0, 4, 8\} = \{0000, 0100\}$

It means that s_0 may have two similar elements $0000, 0100$ in T

— $t_0 = 0000 \xrightarrow{\text{h(0000)=0}} BF_{s_0} = BF_{t_0} = 1, B_1(0000) = B_1(0000) \wedge S = \{0, 4, 8\} = \{0000, 1000\}$

It means that t_0 may have two similar elements $0000, 1000$ in S

— $s_{10} = 1010 \xrightarrow{\text{h(1010)=10}} BF_{s_{10}} = 1, BF_{t_{10}} = 0, B_1(1010) = \emptyset$

It means that s_{10} has no similar elements in T

— $t_4 = 0100 \xrightarrow{\text{h(0100)=4}} BF_{t_4} = 1, BF_{s_4} = 0, B_1(0100) = \{0\}$

It means that t_4 may have! a similar element 0000 in S

— $y = 1001 \xrightarrow{\text{h(1001)=9}} BF_{s_9} = BF_{t_9} = 0, B_1(1001) = \emptyset$

It means that y belongs to neither S nor T, thus it is filtered out.

3.2.3 Extended Intersection Fuzzy Filter

Expand the above idea, we propose an Extended Intersection Fuzzy Filter structure to represent similar elements of multiple input datasets. We consider a fuzzy multi-way join

$$S_1 \bowtie_r S_2 \bowtie_r S_3 \bowtie_r S_4 ... \bowtie_r S_n$$

The threshold of all joins are the same r. The similar elements of S_j in S_k $(j, k = 1..n)$ are $\bigcup_{i=1}^{|S_j|}(B_r(s_i) \cap S_k^*)$. Thus, we store BF of each datasets and a ball list to calculate the similar elements of each pair of datasets by the same way in IFF. Figure 3.9 describes EIFF structure.

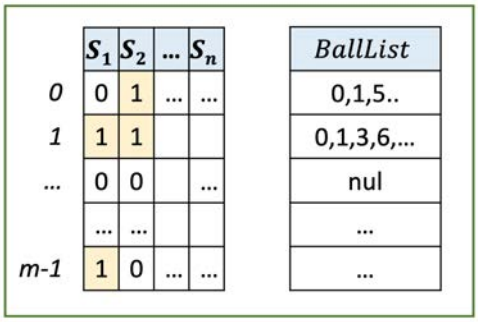


Figure 3.9 – EIFF structure

EIFF is useful for fuzzy multi-way joins and fuzzy recursive/iterative joins, is not present in the existing studies. In the case of the threshold of every joins are different $(\tau_1, \tau_2, .., \tau_{n-1})$, this fuzzy multi-way join has to use IFF for each pair of datasets to calculate their similarities.

3.3 Optimization Of Fuzzy Big Joins

From the limitations of BF-Ball Hashing algorithm that we have pointed out above, we propose optimizations for fuzzy self joins and fuzzy two-way join by using Fuzzy Filter and Intersection Fuzzy Filter. Our optimizations reduce processing excess calculations and remove the data redundancy in all input datasets.

3.3.1 Fuzzy Self-Joins Using Fuzzy Filters

With the integration of FF, our propose ignore the costly and redundant ball calculations. Specifically, the Fuzzy Filter based fuzzy self join algorithm (FF-FJ) consists of two phases:

— Stage 1 (Pre-processing): A filter $FF(S)$ is built on a join key value 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 $FF(S)$ building. Thus, the Map cost is

$M = |S|$, the communication cost is $D = \#mappers$. A small modification for the Ball List computation at reducer is applied to improve the build cost of FF. The goal of FF is to determine which elements in an input dataset are similar to a given element. But in fuzzy self joins, FF just needs to return the similarities of the elements within S. Hence, the Ball List has to compute for only real elements in S. In other words, ball computation operations are only executed at the position i that $BF_i = 1$. Therefore, the computation cost is the Ball List computation cost of $FF(S)$

$$C_{BallList-computation} = |B_r||S| + |S| = (|B_r| + 1)|S| \approx (\frac{b^d}{d!} + 1)|S|$$

If the ball list is pre-known, the processing cost is only $|S|$ AND operations. Figure 3.10 describe an example of pre-processing stage of $FF - FJ$ for the fuzzy join.

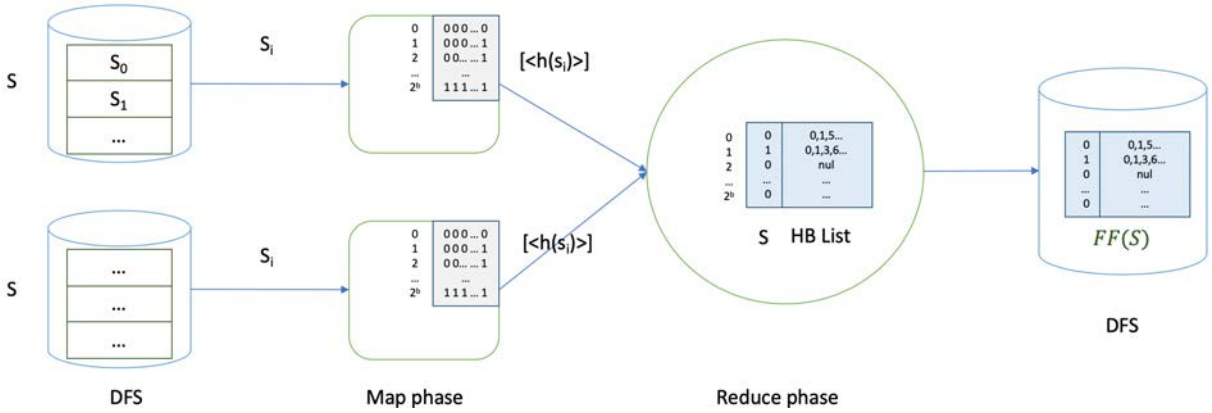


Figure 3.10 – FF-FJ Pre-processing stage

— Stage 2 (Join processing): $FF(S)$ is distributed to all the computing nodes and used to quickly emit real similar elements of the input dataset during the map phase. Each record s is hashed by $h(s)$ and use the filter FF to emit the intermediate tuple in form $< h(s), (-1, s) >$ if $h(s) > h(t_i)$; $< h(t_i), (h(s), s) >$ if $h(s) < h(t_i)$ for all t_i in $BallList_{h(s)}$. The number of reducers is $m = |\Sigma|^b/2 = 2^{b-1}$

$$s \xrightarrow[FF(S)]{map} \begin{cases} < h(s), (-1, s) >, & \forall t_i \in B_r(s) \cap S, h(s) > h(t_i) \\ < h(t_i), (h(s), s) >, & \forall t_i \in B_r(s) \cap S, h(t_i) > h(s) \end{cases}$$

In ideal cases, regardless the false positive, our approach has no redundant intermediate data and no duplicated results without verification in reducers. The output results are computed by the joins on list of elements for every keys. They join all

-1 with all remaining elements. An example of join stage of $FF-FJ$ for the fuzzy join with 4-bit string and threshold $r = 1$ is shown in Figure 3.11.

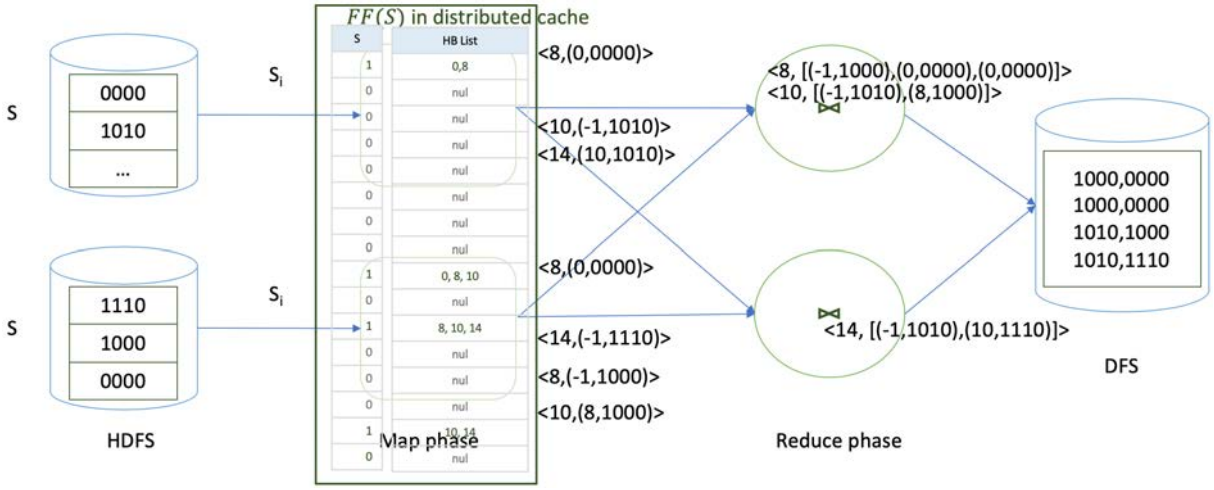


Figure 3.11 – Join processing stage of FF-FJ algorithm with $b = 4, r = 1$

S maps to intermediate tuples as follows:

- 0000: $h(0000) = 0, 0 \xrightarrow{FF(S)} = \{0, 8\}$. Because $0 < 8, 0000 \xrightarrow[FF(S)]{map} < 8, (0, 0000) >$.
- 1010: $h(1010) = 10, 0 \xrightarrow{FF(S)} = \{8, 10, 14\}$. Because $8 < 10$, $1010 \xrightarrow[FF(S)]{map} < 10, (-1, 1010) >$. On the contrary, $10 < 14$, $1010 \xrightarrow[FF(S)]{map} < 14, (10, 1010) >$.
- There are two elements 0000 that are similar to 1000 in S, these three elements are sent to only one reducer to compute the results. Reducer receives $< 8, [(-1, 1000), (0, 0000), (0, 0000)] >$ and emits two pairs $(1000, 0000)$ as results.

We optimize the pre-processing costs of $FF-FJ$ by implementing in Spark with its cache techniques. As a result, the cost of input reading is reused.

3.3.2 Fuzzy Two-way Joins Using Intersection Fuzzy Filters

In the previous approach, we use BF to improve fuzzy two-way joins. We proved that this algorithm reduces significantly redundant intermediate data. However, besides the limitation of ball calculation for all elements, the algorithm can only eliminate non-joining tuples of one dataset, without eliminating non-joining tuples of remaining dataset. This redundancy considerably increases associated overheads in cases of fuzzy multi-way joins and fuzzy iterative joins. For this reason, we propose optimizations for fuzzy two-way

joins using Intersection Fuzzy Filter. The Intersection Fuzzy Filter based fuzzy two-way join (IFF-FJ) consists also two phases:

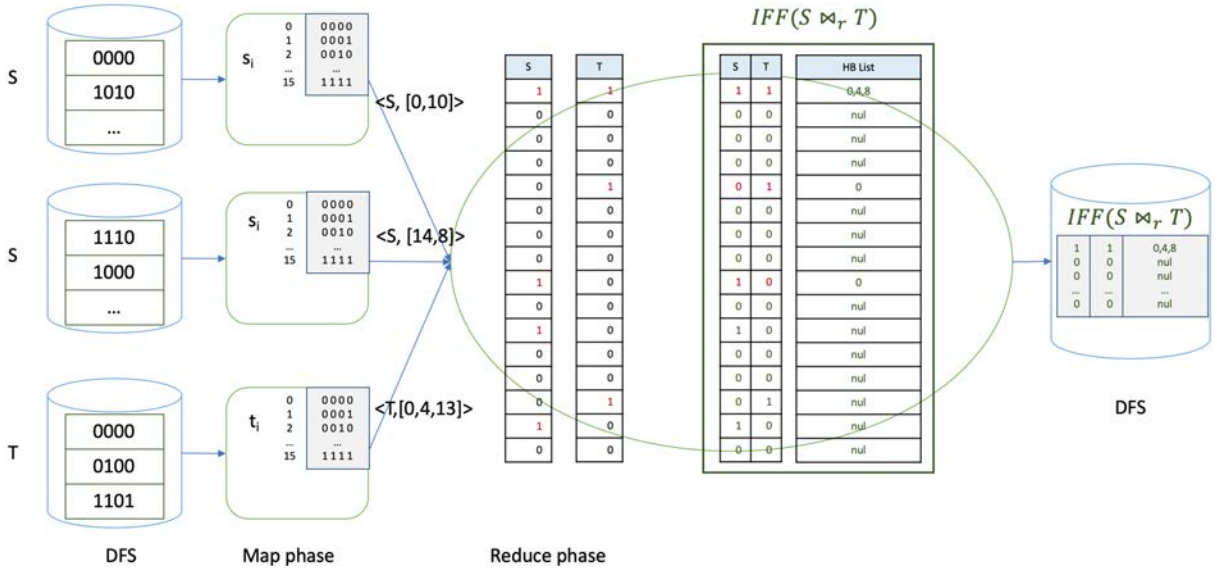


Figure 3.12 – Example of FF-FJ pre-processing stage

— Pre-processing stage: A filter $IFF(S \bowtie_r T)$ is built on a join key value set of the input datasets S and T. Each worker hashes tuples of input splits to find $h(s_i), h(t_j)$, emits a list of $[S, < h(s_i) >]$ or $[T, < h(t_i) >]$ to one reducer for $IFF(S \bowtie_r T)$ building. We also improve the build cost of IFF by calculating only balls at positions i that $BF_i(S) = 1$ or $BF_i(T) = 1$. Figure 3.12 describes an example of pre-processing stage of $IFF - FJ$. Ball computations are only executed at positions $\{0, 4, 8, 10, 13, 14\}$

The Intersection Fuzzy Filter is then stored on Distributed File System (DFS). When the size of filters is large, the filter files will be compressed in formats such as gzip, bzip2, etc. This compression is really efficient for delivering filters to all nodes.

Besides, the IFF building can detect the empty filter to early end the join operation. This interesting characteristic, which is very useful for fuzzy multi-way joins and fuzzy recursive/iterative joins, is not present in the existing studies. At the end of the pre-processing stage, if the IFF is empty, the join stage is skipped and the fuzzy join operation is finished.

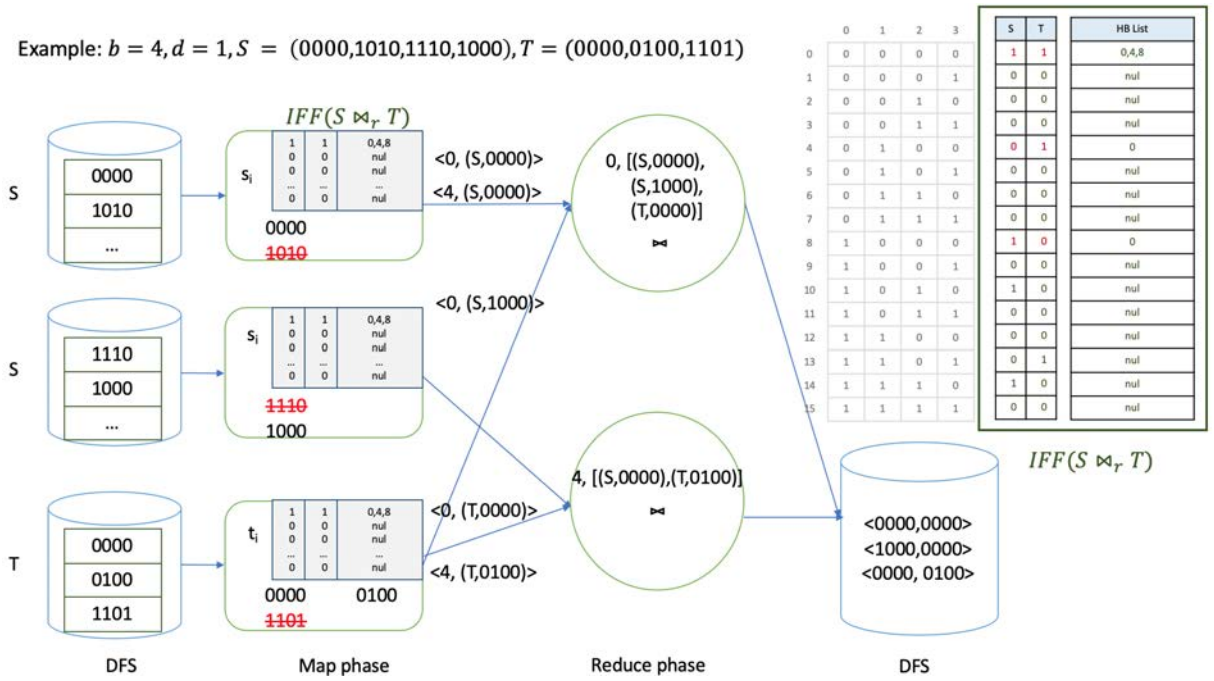


Figure 3.13 – Example of join processing stage of IFF-FJ algorithm

— Join processing stage: Similar to $FF-FJ$, $IFF(S \bowtie_r T)$ is distributed to all the computing nodes and used to quickly emit real similar elements of the input datasets during the map phase.

- Each record s is hashed by a hash function and use the filter IFF to get its ball list. Mappers emit the intermediate tuples in form $< h(t_i), (tag_S, s) >$ for all t_i in $BallList_{h(s)}$.

$$s \xrightarrow[IFF(S \bowtie_r T)]{map} < h(t_i), (tag_S, s) >, \forall t_i \in B_r(s) \cap T$$

- Each record t is also hashed by a hash function and is filtered by IFF. t is only mapped to $< h(t), (tag_T, t) >$ if $BallList_{h(t)}$ is not empty

$$t \xrightarrow[IFF(S \bowtie_r T)]{map} < h(t), (tag_T, t) > \quad if \quad BallList_{h(t)} \neq \emptyset$$

In ideal cases, regardless of the false positive, our approach has no redundant intermediate data and no duplicated results without verification in reducers. The reduce function performs the cross product of the tuples of tag S that are buffered and each incoming tuple of tag R. It is completed by writing the output to DFS. Figure 3.13 illustrates an example of join stage of $IFF-FJ$ for the fuzzy two-way join with 4-bit string and threshold $r = 1$.

For dataset S,

- $BallList_{1010} = BallList_{1110} = \emptyset$. Thus, $1010, 1110$ are pruned out.
- $0000 \xrightarrow{h(0000)0}$, $BF_0(T) = 1$, thus $BallList_0 = BallList_0 \cap T = \{0, 4, 8\} \cap \{0, 4, 13\} = \{0, 4\}$.
  $0000 \xrightarrow[IFF(S \bowtie_r T)]{map} < 0, (S, 0000) >, < 4, (S, 0000) >$
- $1000 \xrightarrow{h(1000)8}$, $BF_8(T) = 0$, thus $BallList_8 = \{0\}$.
  $1000 \xrightarrow[IFF(S \bowtie_r T)]{map} < 0, (S, 1000) >$

For dataset T,

- $BallList_{1101} = \emptyset$. Thus, 1101 are pruned out.
- $BallList_{0000} \neq \emptyset$, $0000 \xrightarrow[IFF(S \bowtie_r T)]{map} < 0, (T, 0000) >$
- $BallList_{0100} \neq \emptyset$, $0100 \xrightarrow[IFF(S \bowtie_r T)]{map} < 4, (T, 0100) >$

The output results of key 0 are $(0000, 0000), (1000, 0000)$. The output result of key 4 is $(0000, 0100)$.

3.3.3 Fuzzy Filters 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 Hamming 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.

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 3.14

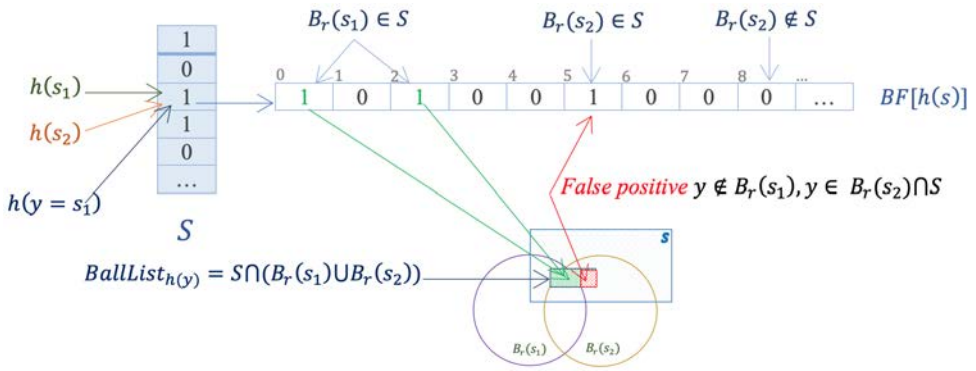


Figure 3.14 – Example of false positive of FF

— s_1, s_2 hash to the same position $h(s_1) = h(s_2) = 2$.

— Consider a query $y = s_1$: $BallList_2 = (B_r(s_1) \cup B_r(s_2)) \cap S = \{0, 2, 5\}$
$\{0, 2\}$ is real similarities of s_1 in S
$\{5\}$ is a false positive, because it belongs $B_r(s_2)$ and is not a real similar element of s_1.

— In another hand, we consider s_1, s_2 whose hash index is 2. 2 is one of elements in results of the query $y = s_1$. If s_1 is not similar to s_2, s_2 is a mistaken similarity of s_1.

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 ($|S| = n \leqslant |\Sigma|^b = 2^b$), the false positive of a hash bucket list of size m bits is

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

Each hash ball contains $|B_r| \approx b^r/r!$ 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. In other words, this false bit becomes a false positive answer only if its elements is real in S.

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. We implement these approaches using cache in Spark to amortize this cost.

IFF helps to optimize fuzzy two-way joins. In pre-processing stage, both of two datasets are read to build the filter. The caching technique also takes advantage of this approach.

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 $B(t)$.

3.4 Cost Analysis

Table 3.2 summarizes the costs of the different algorithms in the previous chapter and our propose Fuzzy Filter based Fuzzy join using Hamming distance by the $M - C - R$ cost model.

Table 3.2 – Summary of costs for various Hamming distance-based join algorithms

Approach	**Pre**	**M per elem**	**# R**	**C**	**R**
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	$k\|S\|$	$k\|B_d\|$	$n = 2^{b-1}$	$D_{BF-BH1} < \|S\|\|B_d\|$	$D_{BF-BH1}\|S\|/2^{b-1}$
Splitting	0	$d+1$	$(d+1)2^{b/(d+1)}$	$(d+1)\|S\|$	$(d+1)\|S\|^2/2^{b/(d+1)}$
BFBS	$k\|S\|$	$k\|B_d\|(d+1)$	$(d+1)2^{b/(d+1)}$	$D_{BF-Splitting} < (d+1)\|S\|$	$D_{BF-Spitting}\|S\|/2^{b/(d+1)}$
FF-FJ	$\|S\|$	1	$n = 2^{b-1}$	$D_{FF-FJ} = D_{BF-BH1}$	$D_{FF-FJ}\|S\|/2^{b-1}$

For the communication cost, BFBS is the best approach. Ball Hashing algorithms are the most suitable solution to processing cost. Besides, integrating FF brings the following changes according the (M, C, R) model:

— The pre-processing cost is incurred by reading the input to generate $BF(S)$. But FF uses $k = 1$ hash function, thus the preprocessing cost of FF-FJ is better than BF-BH1. In addition, this cost can be amortized, especially using streaming or caching techniques (e.g Spark [10]). In this case, the preprocessing cost of FF-FS can be omitted.

— For each tuple, the map phase spends $O(1)$ to determine its similarities. Thus, this M cost is already the best optimization.

— The communication cost of FF-FJ is equal to BF-BH1 because of the same filtering way. Similarity, their processing cost are also equal.

In summary, filters based approaches improved fuzzy join evaluation. In the previous chapter, BF-BH1 is more effective than BF-BS even for the larger communication cost. FF-FJ is thus the best solution based on BF-BH1 optimizations.

For fuzzy two-way joins, IFF-FJ is clearly more improve because it filters both input datasets.

3.5 Experimental Validation

In this section, we present experimental results obtained from the execution of fuzzy self joins and fuzzy two-way joins using the different approaches. All experiments were run on the same cluster, datasets and parameters in the previous Chapter.

3.5.1 Fuzzy Self Join Evaluation

Experimental protocol

The experimental evaluations in this section perform fuzzy self joins using Hamming Distance over the last 6 characters of the column 36 (fixed-length: $b = 6$, alphabet: $\Sigma = 10$). The approaches are performed in our experiments include: BF-BH1, BF-Ball-Splits (BFBS) and FF. We use $m = 10^6$ to build Fuzzy Filters.

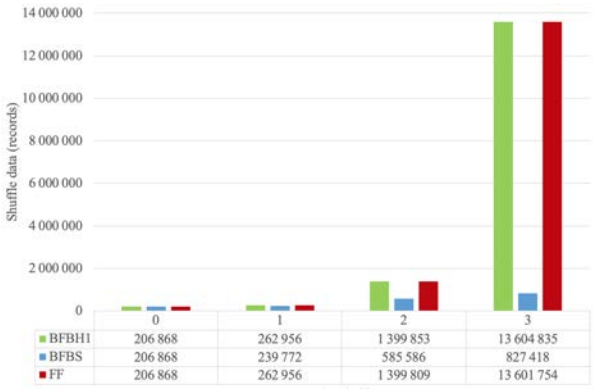
The following join query is used.

$$Q_1 = S \bowtie_{S.36 \approx S.36} S$$

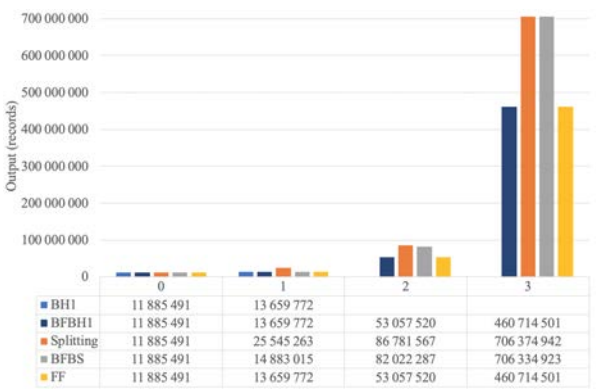
To compare the scalability and the performance of the different algorithms, we conducted two types of tests.

— First, we investigated the scalability of all methods to compute a fuzzy self joins of the datasets $10GB$ (Exp-6) in Table 2.4 by increasing the Hamming distance threshold from 0 to 4 .
Exp-6: $Q_1 = 10GB \bowtie_\tau 10GB, \tau = 0, 1, 2, 3, 4$

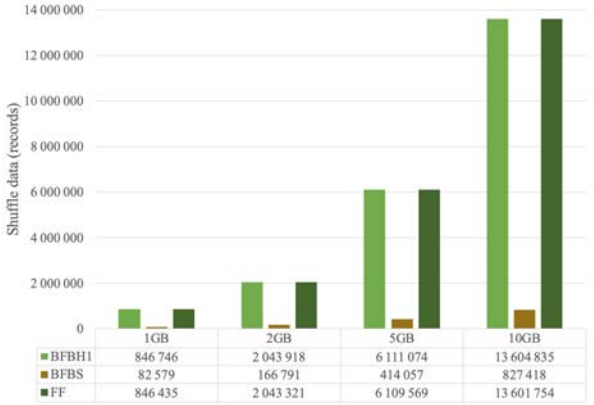
— Second, we investigated the scalability of the algorithms by increasing the size of the datasets in Table 2.4 with the Hamming distance threshold of fuzzy self joins is 3 (Exp-7).



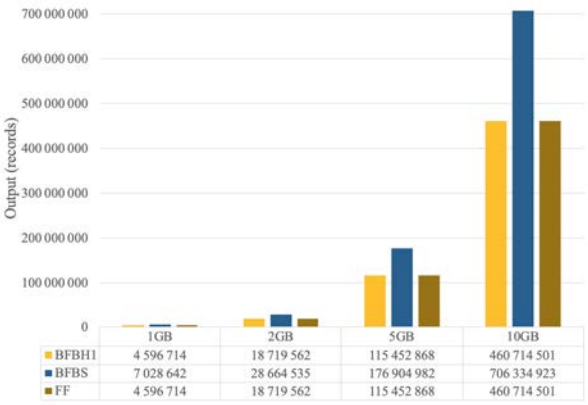
(a) Intermediate data of Exp-6



(b) Total output results of Exp-6



(c) Intermediate data of Exp-7



(d) Total output results of Exp-7

Figure 3.15 – Intermediate data and output results of fuzzy self joins approaches on 10GB in various thresholds

Exp-7: $Q_1 = S \bowtie_\tau S, \tau = 3, S = 1GB, 2GB, 5GB, 10GB$

For each experiment, we run in 4 times and get the average result. Subsequently, we show results of experiments, and summarize our conclusions for each algorithm.

Evaluation Of Approaches

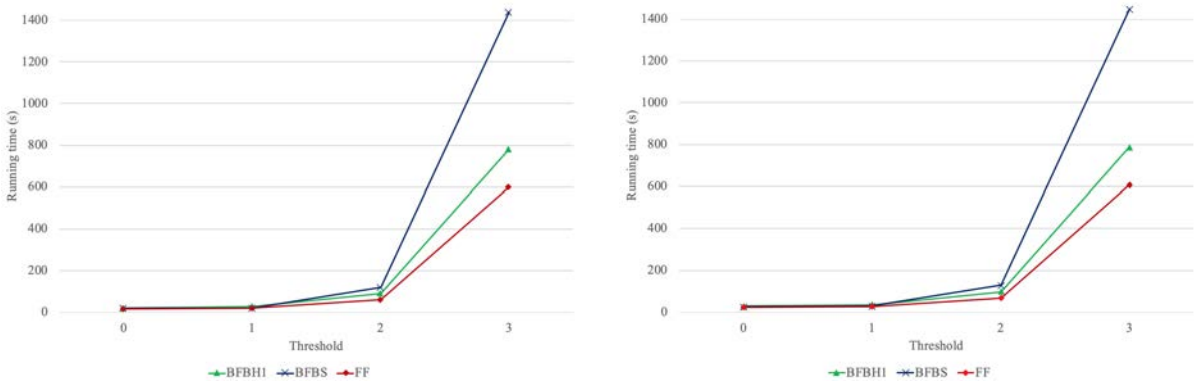
First, we compare the amount of intermediate data and total output results of Exp-6 and Exp-7 in Figure 3.15.

With the threshold $\tau = 0$, the algorithms are considered as the general equal joins while they are performed as the fuzzy joins. In this case, the filters do not affect the amount of intermediate data. Hence, the numbers of intermediate tuples are equal for all approaches.

The intermediate data and the total output results of BF-BH1 are always equal to

FF-FJ because of the same filtering technique. BFBS algorithm generates intermediate tuples less than BFBH1 and FF-FJ algorithms.

However, the BFBS produces a large number of duplicate results. The BH1, BF-BH1 give the exact answers, without duplication.



(a) Joining time (b) Total execution time

Figure 3.16 – Exp-6: Running time of fuzzy self joins approaches on 10GB in various thresholds

Table 3.3 – Running time of the fuzzy self join approaches on 10GB in various thresholds

Approach	0		1		2		3	
	preproc.	join stage	preproc.	join stage	preproc.	join stage	preproc.	join stage
BFBH1	9	22	9	27	9	90	9	780
BFBS	9	19	9	21	9	120	9	1440
FF	9	16	9	19	9	60	10	600

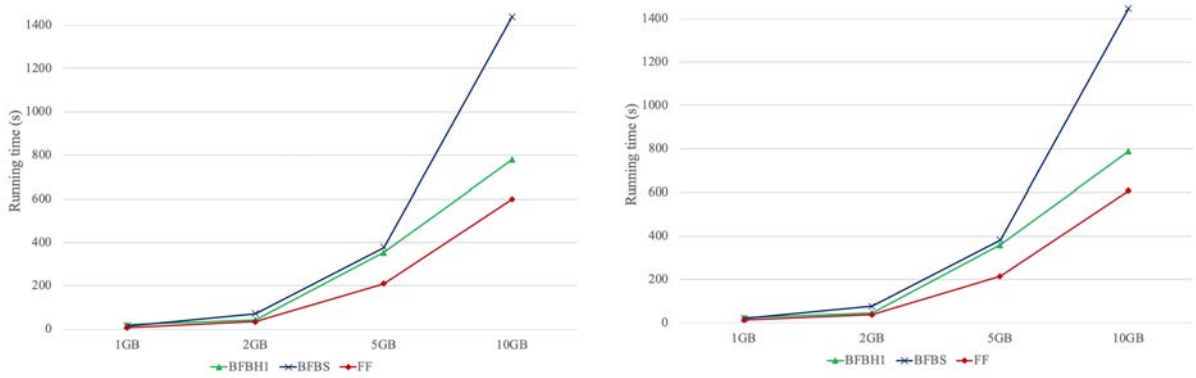
Next, we evaluate the efficiency of these fuzzy self join algorithms by comparing their total execution time. Table 3.3 and Figure 3.16 identify in detail the execution time of the pre-processing stage and the join stage for the fuzzy self join algorithms on various thresholds.

The preprocessing time of Bloom filter based approaches is the time of $BF(S)$ building, 9s for all threshold 0,1,2,3. Similarity, the preprocessing time of FF is the time of $FF(S)$ building, but it has a bit increase in threshold 3 (10s). On the small threshold (0,1), the execution of the BFBS is faster than BFBH1 due to the less of intermediate data. However, on the larger threshold, BFBS is slower than the BF-BH1 because of the distance verification at joining stage and the duplicate output. FF-FJ is always the best solution about running time because its optimizations.

Table 3.4 – Running times of fuzzy self joins in the theshold $\tau = 3$ on various datasets

Approach	1GB		2GB		5GB		10GB	
	preproc.	join stage	preproc.	join stage	preproc.	join stage	preproc.	join stage
BFBH1	4	22	5	42	6	354	9	780
BFBS	4	16	5	72	6	378	9	1440
FF	4	8	5	35	6	210	10	600

Lastly, the running times of the fuzzy self join using the different algorithms from $1GB$ to $10GB$ are demonstrated in the Table 3.4 and Figure 3.17.



(a) Joining time

(b) Total execution time

Figure 3.17 – Running time of fuzzy self join approaches in various datasets in the thresold $\tau = 3$

By the dataset size increment, BFBS are the most severely affected algorithms. Its execution time increase rapidly. BF-BH1 is better than BFBS. FF-FS demonstrates the efficiency of Fuzzy Filter and it is also the winner for the execution time in these experiments.

3.5.2 Fuzzy Two-way Join Evaluation

Experimental protocol

The experimental evaluations in this section perform fuzzy two-way joins using Hamming Distance over the last 6 characters of the column 36 (fixed-length: $b = 6$, alphabet: $\Sigma = 10$). The approaches are performed in our experiments include: BF-BH1, BF-Ball-Splits (BFBS), and IFF-FJ(IFF). The following join query is used.

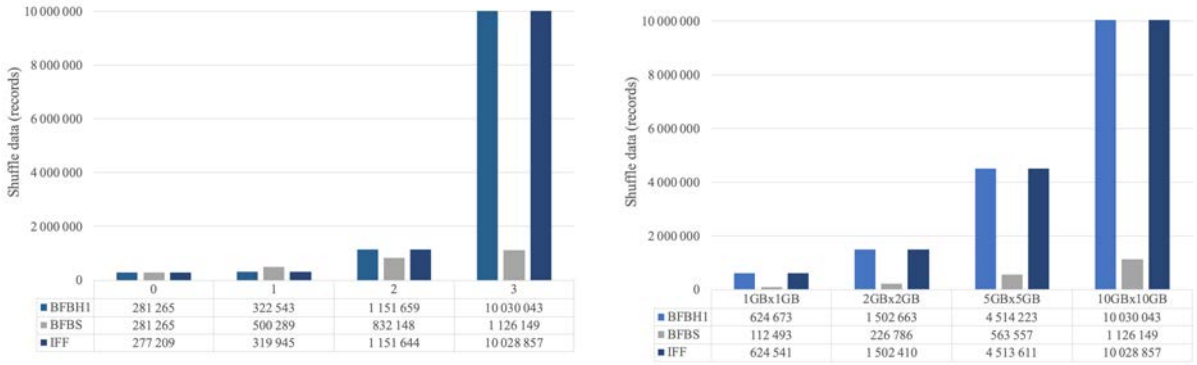
$$Q_2 = S \bowtie_{S.36 \approx T.38} T$$

Similar to the experiments above, we conducted two types of tests.

— First, we investigated the scalability of all methods to compute a fuzzy two-way joins of the datasets $10GB \bowtie 10GB$ (Exp-8) by increasing the Hamming distance threshold from 0 to 4.
Exp-8: $Q_2 = S \bowtie_\tau T, S, T = 10GB, \tau = 0, 1, 2, 3, 4$

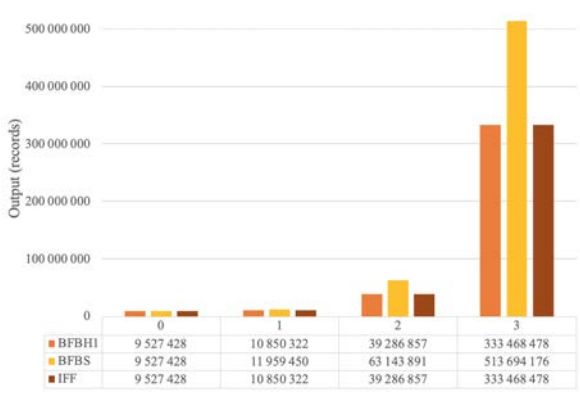
— Second, we investigated the scalability of the algorithms by increasing the size of the datasets with the Hamming distance threshold of fuzzy two-way joins is 3 (Exp-9).
Exp-9: $Q_2 = S \bowtie_\tau T, S, T = 1GB, 2GB, 5GB, 10GB, \tau = 3$

Evaluation Of Approaches

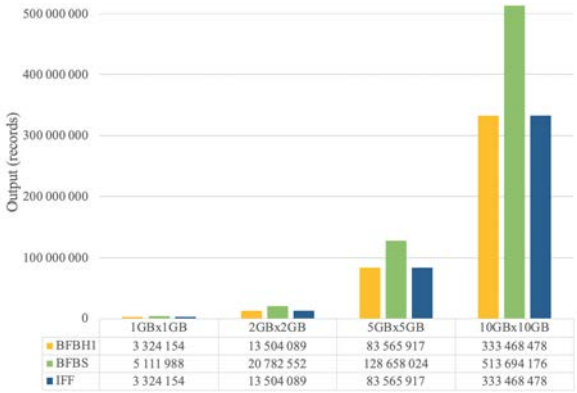


(a) Exp-8 (b) Exp-9

Figure 3.18 – Intermediate data of Exp-8 and Exp-9

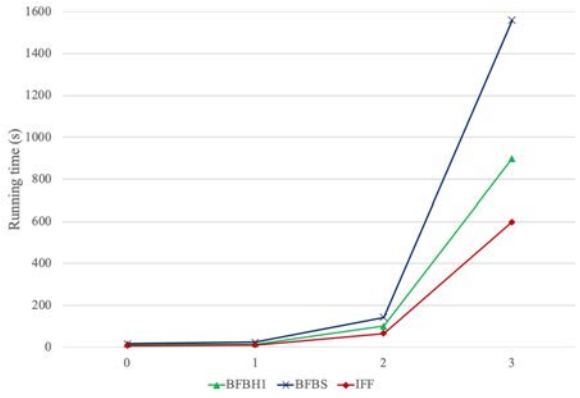


(a) Exp-8

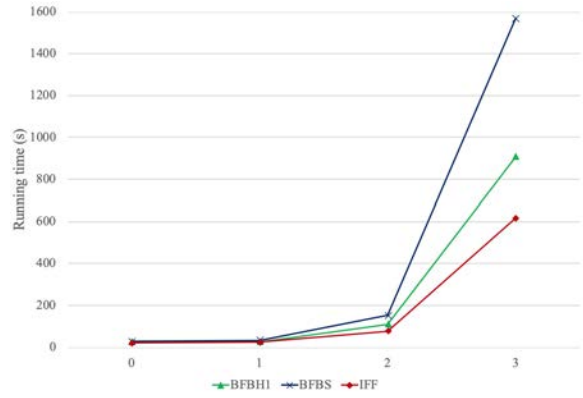


(b) Exp-9

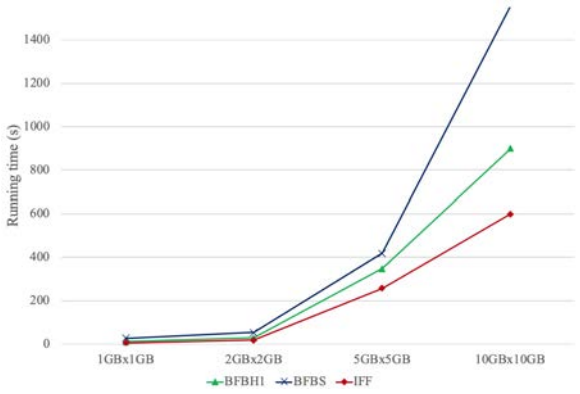
Figure 3.19 – Total output results of Exp-8 and Exp-9



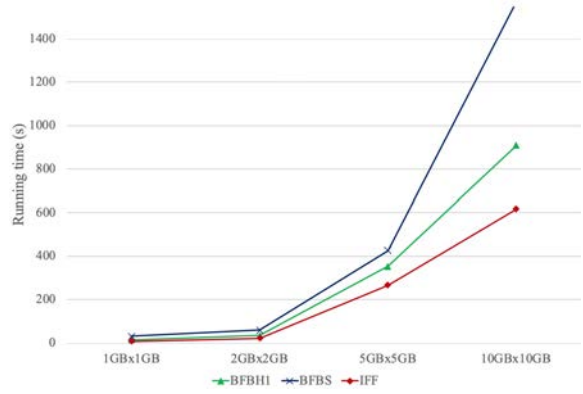
(a) Exp-8: Joining time



(b) Exp-8: Total execution time



(c) Exp-9: Joining time



(d) Exp-9: Total execution time

Figure 3.20 – Running time of Exp-8 and Exp-9

Figure 3.18 shows the intermediate data for the experiments Exp-8 and Exp-9. On the small threshold (1), BFBS generates more intermediate data than BFBH1 and IFF. On

the large dataset or large threshold, BFBS takes advantage on shuffle data. IFF processes less shuffle data than BFBH1 because it filters on both datasets.

Figure 3.19 presents total output results for the experiments Exp-8 and Exp-9. BFBS contains large amount of duplication. BFBH1 and IFF always return the exact answers.

Figure 3.20 reports the running times of the algorithms for various thresholds and various datasets. IFF is the winner in all experiments inspite of the preprocessing time.

3.6 Summary

In this chapter, 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 and experiments to prove it.

We propose also an Intersection Fuzzy Filter for fuzzy two-way join optimization. Beside the same benefits with Fuzzy Filter, IFF prunes out redundant data in both datasets. An Extended Intersection Fuzzy Filter model is proposed to improve fuzzy multi-way join, fuzzy recursive join that will present in future work.

Another advantage of these fuzzy filter is easily updated. Thus, they can be applied to fuzzy stream joins, fuzzy incremental joins.

This work leads to one publication [105] in Proceedings of the 29th IEEE International Conference on Fuzzy Systems (FUZZIEEE 2020).

Conclusion

In this final chapter, we will conclude by describing the results of the optimization for the fuzzy big joins using the Bloom filter and the Fuzzy filters in MapReduce. We will also suggest some future research directions which would further extend its applicability.

Thesis Conclusions

MapReduce is widely considered one of the key processing frameworks for Big Data. However, MapReduce has its own limitations. Complex operations in MapReduce are used extensively and expensively, especially the join operation. The Fuzzy Join is one of the key operations for analyzing large datasets in many application scenarios. With the goal of our study, we have described and analyzed how the fuzzy big join operation can be supported efficiently in MapReduce. For existing solutions, which is surveyed in chapter 1, it has been shown that much redundant data is involved in the fuzzy join operations. Therefore, this thesis is dedicated to solve the problems of the joins in more efficient ways. The main contributions of our research are the following:

— In Chapter 1, we first described the foundations of fuzzy big joins. Different strategies for performing similarity joins using MapReduce have been studied. Then, we summarized the current states of the art for parallel solutions in two types: (1) single stage parallel fuzzy join algorithms and (2) multiple stage parallel fuzzy join algorithms. The focus is on algorithms that can operate in a single MapReduce step in order to avoid the overhead associated with initiating multiple MapReduce jobs. Besides, we present a theoretical analysis of different methods, showing that different algorithms provide different tradeoffs with respect to map cost, reducer cost and communication cost.

— In Chapter 2, we presented our approaches to improve fuzzy joins in MapReduce by using Bloom filters. We described MapReduce implementations of Hamming based single stage algorithms. For each solution, we proposed a filter building stage and a Bloom filter integration stage to reduce redundant data and duplication results. Additionally, each proposed algorithm is implemented in Spark for fuzzy

self join and fuzzy two-way join. We closed this chapter with a theoretically cost analysis, experiments of the various approaches and our recommendations for the best approach.

— Based on the probabilistic model, we define new filter types in Chapter 3:
  - The Fuzzy Filter, to represent a set and answer which similarities of an element are present in the set. It is used to remove most of dissimilar elements in a set and therefore ameliorate fuzzy self joins.
  - The Intersection Fuzzy Filter, an extension of the Fuzzy Filter to represent sets and their similarities. It can determine where are close ones of a given element in another set. It is thus applied to optimize fuzzy two-way joins.
  - We propose an Extended Intersection Fuzzy Filter to represent multiple sets and their similarities. It benefits on fuzzy multi-way joins.

  These filters are dynamic. They are independent with similarity measure functions, threshold distance and easily updated. Thus, they can benefit not only static fuzzy joins but also dynamic fuzzy joins as stream joins and iterative/recursive joins.
  We explained the performance optimizations of these techniques in a theoretical study involving both fuzzy self join and fuzzy two-ways join queries. At the end of this chapter, we deploy experiments of fuzzy big joins implemented by the different algorithms using Spark. Experimental comparisons of the different algorithms for each the fuzzy join are examined with respect to the intermediate data amount, the total output amount, and the total execution time.

Our improvements bring benefits that can be applied to solve popular problems in various fields such as fuzzy join operations, reconciliation and deduplication, error-correction, image content based search, data cleaning, data integration, recommendation system, similarity genome search, etc.

Both the cost models and the experiments show that a fuzzy join operation using the Bloom filter is more efficient than using the other solutions without filtering since it significantly reduces redundant data, and thus produces less intermediate data. These significantly reduce I/O and communication overheads. However, Fuzzy filters improve even more fuzzy join computations. Although the filters have small false positives and an extra cost for the pre-processing job, their efficiency in space-saving and filtering often outweighs these drawbacks.

Future Work

A number of open problems should be solved to allow the complete development of large-scale fuzzy joins processing in MapReduce. These problems suggest some research directions as follows.

Fuzzy Big Joins Optimization Using Other Distance Measures

We considered a number of distance measures, but concentrated on Hamming distance because it is in a sense the simplest measure and lets us offer the clearest view of the various algorithmic approaches. We can continue this line of work and consider the more challenging cases of Edit and Jaccard distances [5, 65].

Edit distance (ED) also known as Levenshtein distance, measures the minimum number of edit operations needed to transform one string into another, where edit operations is an insertion, deletion or substitution of a single character. It can be calculated via dynamic programming [107].

Jaccard similarity of sets is the ratio of the size of the intersection of the sets to the size of the union. The more the two sets have in common, the closer the Jaccard similarity will be to 1.0

We may also consider other similarities, as semantic ones [38], based on knowledge bases to help to find out close terms. These similarities could be based on taxonomies and ontologies, on polarities, etc. By taking such similarities into account, we may also extend our approach to better taking into account the graduality of the similarity degrees. This work will be based on the formal definitions of fuzzy relations [26] and fuzzy similarities, as studied in [21, 22, 74].

Optimization For More Complex Fuzzy Join Queries

Extended Intersection Fuzzy Filter can be applied to optimize fuzzy multi-way joins on the same join attribute and the same distance threshold. In the case of multi-way joins on different join attribute or different distance threshold, we can use pairs of Intersection Fuzzy Filter. These challenges will be studied more thoroughly in the future based on multi-way join [103, 2] in MapReduce.

Fuzzy stream joins are fuzzy joins, where new data arrive continuously over time has many important applications such as data cleaning [120] and data mining [121, 77],

Internet traffic analysis [33], sensor network monitoring [101], etc. For example, fuzzy stream join queries can be used to help clean sensor data collected from various sources that might contain inconsistency [101]. As another example [120], in the stock market, it is crucial to find correlations among stocks so as to make trading decisions timely. In this case, we can also perform fuzzy stream over price curves of stocks in order to obtain their common patterns, rules, or trends.

Fuzzy recursive joins are fuzzy iterative/incremental joins where the fuzzy join operations repeat multiple times.

In these fuzzy join types, the new tuples appear continuously. Therefore, their optimizations need dynamic filters that can be easily updated. Our proposed fuzzy filters satisfy this.

Specifically, in the case without false positives, when a new tuple t arrives that does not exist yet in S ($BF(t, S) = 0$), the $BallList_{h(t)}$ will be easily recalculated by $B_d(t) \cap S$. And of course, the fuzzy filter does not change if t has already existed in S.

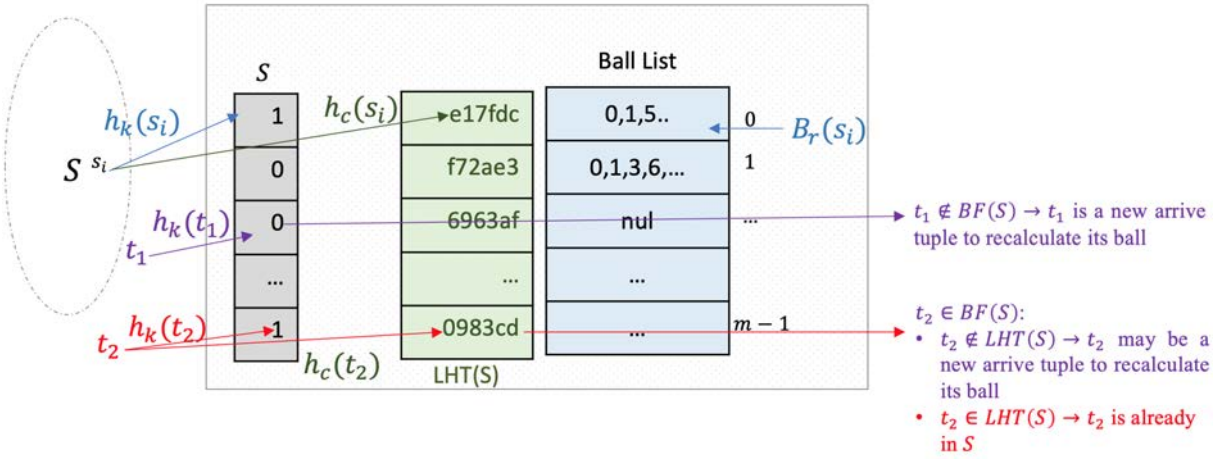


Figure 3.21 – Difference Fuzzy Filter structure

Contrarily, with a false positive, to determine if t is already in S or not, we inherit the idea of Difference Filter [89] to propose a Difference Fuzzy Filter structure as in Figure 3.21. It combines FF and a Lossy Hash Table $LHT(S)$ to detect duplicate elements. $LHT(S)$ consists of an array of buckets, where each bucket contains fingerprints of elements of S. The fingerprint is created by applying a cryptographic hash function h_c (e.g. MD5, SHA-2, and SHA-3) for an element. However, the position of an existing element in $LHT(S)$ can be overwritten by other elements. This overwriting is known as a collision. The size of

the LHT is m, the same with $BF(S)$. In addition, the size of each the bucket is specified by the cryptographic function h_c, namely, typical MD5 or SHA-3 fingerprints are 128 or 1600 bits in length. When a tuple t arrives, $h_k(t)$ determines $t \in S$ or not.

— If $t \notin S \rightarrow t$ is a new arrive tuple. The $BallList_{h(t)}$ will be easily recalculated by $B_d(t) \cap S$.

— If $t \in S$, $h_c(t)$ is used to calculate the fingerprint of t and compare with existing fingerprint in LHT at position $h_k(t)$.

- If the fingerprint of t is not in $LHT(S) \rightarrow$ it may be a new one or an overwritten one. We thus also recalculate the ball of t by $B_d(t) \cap S$.
- If the fingerprint of t exists in $LHT(S) \rightarrow t$ is already in S. Hence, the result is returned without an update.

Filter-Verification Based Fuzzy Big Joins Optimization

Filter - verification techniques use set prefixes or signatures followed by an explicit verification of candidate pairs to compute fuzzy joins. The parallel fuzzy big join algorithms are compared in experimental studies [43]. In these surveys, Vernica Join achieved the best performance in most experiments. Hence, we focus on Venica joins for filter - verification based fuzzy big joins optimization.

Vernica et al. [109] proposed a 4-MapReduce jobs approach that is based on prefix filtering technique to compute parallel distributed fuzzy joins. First, it uses two MapReduce jobs to build an inverted index on tokens of input records. Then, it generates candidate pairs from the inverted lists, using additionally the length, positional and suffix filters to prune candidates. A deduplication step is finally employed to remove duplicate result pairs generated from different reducers. In general, this algorithm exists the limitations:

— Multiple MapReduce jobs algorithm: The resource for each initial MapReduce task affects the performance of a parallel distributed algorithm. Multiple jobs algorithms require a significant cost of resources for initial, input reading, shuffle and close. In particular, the token index has to calculate by two MapReduce jobs.

— Redundant tokens: The prefix filtering technique is based on an ordering of the token universe to reduce candidates. It tries to eliminate higher frequency elements from the prefix filtering and thereby expect to minimize the number of comparisons. Clearly, by this way, rarest tokens that exist in only one record are emitted as redundant candidates while these tokens should be omitted right from the start.

— Duplicated results and deduplication job: Duplicate pairs in the result of the verifi-

cation job are generated from different reducers by candidate pairs have more than one common token in their prefix. Thus, it wastes the distance measure computation and output many times for these duplicated pairs. Moreover, this algorithm has to pay for one more deduplication job.

These limitations are illustrated in Figure 3.22. The rarest tokens L, J, K should be skipped. The candidate pair (s_1, s_4) has two common tokens Q, N. Hence, it produces duplicate result pairs.

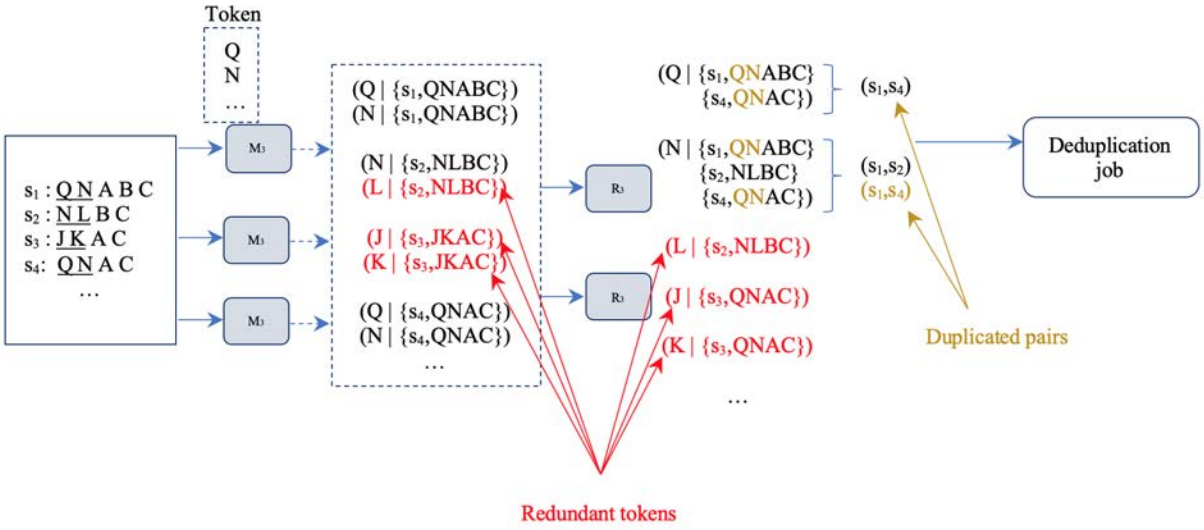


Figure 3.22 – Example for limitations of Vernica joins

From the limitations of Vernica joins as above, we propose the main improvements:

— Replace two jobs of index tokens building by only one job using Counting Bloom Filters [42]. All tokens insert to a CBF to count their frequencies.

— Prune out redundant tokens in the map phase of job 3 by their counters. Each tuple queries their tokens on built CBF to determine their prefix. If the counter of a token in CBF is one, this token is eliminated.

— Remove early duplicated result pairs and eliminate the deduplication job by prefix tokens overlap computing. For example, in Figure 3.22, the candidate pair (s_1, s_4) has two common tokens Q, N. Thus, it produces results for only the first common token Q. All remaining common tokens are skipped to deduplication. Therefore, job 4 is also removed.

Our propose reduces Vernica join processing from 4 to 2 jobs. Figure 3.23 describes our optimization.

— Preprocessing job: M_1 reads splits of S, computes tokens and hashes to an CBF_{local}. R_1 aggregates CBF_{local}s to a CBF_{global}.

— Join job: M_2 use $CBF(S)$ to filter tokens of every tuple s and generates their prefixes. R_2 computes token overlaps of every candidate pair, skips redundant pairs, verifies their similarities and finally, produces the results.

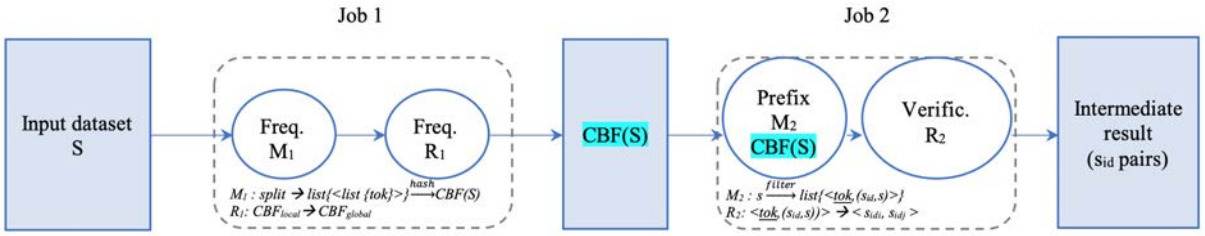


Figure 3.23 – Optimization for Vernica joins

Skewness Problem

Besides, skew data challenge [69, 51, 92, 46] has not been considered in the solutions presented. However, it affects the efficiency of MapReduce algorithms. Hence, researches for skewness problems will also be attended in the future. An option can benefit is to replace the Bloom Filter in Fuzzy Filter structure by a Counting Bloom Filter. CBF helps us predict the possibilities of the number of processing in reducers. Therefore, we will try to balance this cost.

In conclusion, as listed above, there are interesting research directions to explore, and this work, we hope, should become a contribution to the general context of the data science.

Publications Involved in the Thesis

Thi-To-Quyen Tran et al., « 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.

Thi-To-Quyen Tran et al., « Optimization for Large-Scale Fuzzy Joins Using Fuzzy Filters in MapReduce », *in*: *29th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2020, Glasgow, UK, July 19-24, 2020*, IEEE, 2020, pp. 1–8.

BIBLIOGRAPHY

[1] Foto N. Afrati and Jeffrey D. Ullman, *A New Computation Model for Cluster Computing*, Technical Report, 2009.

[2] Foto N. Afrati and Jeffrey D. Ullman, « Optimizing Joins in a Map-Reduce Environment », *in*: *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, Association for Computing Machinery, 2010, pp. 99–110.

[3] Foto N. Afrati et al., « Cluster Computing, Recursion and Datalog », *in*: *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, Springer-Verlag, 2010, pp. 120–144.

[4] Foto N. Afrati et al., « Map-Reduce Extensions and Recursive Queries », *in*: *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, Association for Computing Machinery, 2011, pp. 1–8.

[5] Foto N. Afrati et al., « Fuzzy Joins Using MapReduce », *in*: *ICDE*, 2012, pp. 498–509.

[6] *Amazon Simple Storage Service (Amazon S3)*, URL: `https://aws.amazon.com/s3/`.

[7] *Apache Cassandra*, URL: `http://cassandra.apache.org/`.

[8] *Apache Hadoop*, URL: `https://hadoop.apache.org/`.

[9] *Apache HBase™*, URL: `http://hbase.apache.org/`.

[10] *Apache Spark™ - Lightning-Fast Unified Analytics Engine*, URL: `http://spark.apache.org/`.

[11] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik, « Efficient Exact Set-similarity Joins », *in*: *Proceedings of the 32Nd International Conference on Very Large Data Bases*, 2006, pp. 918–929.

[12] Nikolaus Augsten and Michael H. Bhlen, *Similarity Joins in Relational Database Systems*, 1st, Morgan & Claypool Publishers, 2013.

[13] E. Babb, « Implementing a Relational Database by Means of Specialzed Hardware », *in*: *ACM Trans. Database Syst.* 4 (Mar. 1979), pp. 1–29.

[14] Mostafa Bamha and Gaétan Hains, « An Efficient Equi-semi-join Algorithm for Distributed Architectures », *in*: vol. 3515, May 2005, pp. 755–763.

[15] R. Baraglia, G. De Francisci Morales, and C. Lucchese, « Document Similarity Self-Join with MapReduce », *in*: *2010 IEEE International Conference on Data Mining*, 2010, pp. 731–736.

[16] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant, « Scaling Up All Pairs Similarity Search », *in*: *Proceedings of the 16th International Conference on World Wide Web*, 2007, pp. 131–140.

[17] Philip A. Bernstein et al., « Query Processing in a System for Distributed Databases (SDD-1) », *in*: *ACM Trans. Database Syst.* 6 (Dec. 1981), pp. 602–625.

[18] Spyros Blanas, Yinan Li, and Jignesh M. Patel, « Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs », *in*: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, Association for Computing Machinery, 2011, pp. 37–48.

[19] Spyros Blanas et al., « A Comparison of Join Algorithms for Log Processing in MaPreduce », *in*: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, Association for Computing Machinery, 2010, pp. 975–986.

[20] Burton H. Bloom, « Space/Time Trade-offs in Hash Coding with Allowable Errors », *in*: *Commun. ACM* 13.7 (1970), pp. 422–426.

[21] Bernadette Bouchon-Meunier, Marie-Jeanne Lesot, and Maria Rifqi, « Similarities in fuzzy data mining: from a cognitive view to real-world applications », *in*: *Computational Intelligence: Research Frontiers - IEEE World Congress on Computational Intelligence, WCCI 2008*, vol. 5050, Lecture Notes in Computer Science, Hong Kong, China: Springer, June 2008, pp. 349–367.

[22] Bernadette Bouchon-Meunier et al., « Towards a conscious choice of a fuzzy similarity measure: a qualitative point of view », *in*: *International conference on Information Processing and Management of Uncertainty in knowledge-based systems, IPMU 2010*, vol. 6178, Lecture Notes in Computer Science, Dortmund, Germany: Springer, June 2010, pp. 1–10.

[23] A. Broder and M. Mitzenmacher, « Using multiple hash functions to improve IP lookups », *in*: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, vol. 3, 2001, 1454–1463 vol.3.

[24] Andrei Z. Broder et al., « Syntactic Clustering of the Web », *in*: *WWW*, 1997, pp. 1157–1166.

[25] Andrei Broder, Michael Mitzenmacher, and Andrei Broder I Michael Mitzenmacher, « Network Applications of Bloom Filters: A Survey », *in*: *Internet Mathematics*, 2002, pp. 636–646.

[26] James J. Buckley and Esfandiar Eslami, « Fuzzy Relations », *in*: *An Introduction to Fuzzy Logic and Fuzzy Sets*, Heidelberg: Physica-Verlag HD, 2002, pp. 115–139.

[27] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik, « A Primitive Operator for Similarity Joins in Data Cleaning », *in*: *Proceedings of the 22Nd International Conference on Data Engineering*, 2006, pp. 5–.

[28] Bernard Chazelle et al., « The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables », *in*: *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Society for Industrial and Applied Mathematics, 2004, pp. 30–39.

[29] Lu Chen et al., « Pivot-Based Metric Indexing », *in*: *Proc. VLDB Endow.* (2017), pp. 1058–1069.

[30] E. F. Codd, « A Relational Model of Data for Large Shared Data Banks », *in*: (1970), pp. 377–387.

[31] E. F. Codd, « Relational completeness of data base sublanguages », *in*: *Database Systems*, Prentice-Hall, 1972, pp. 65–98.

[32] Saar Cohen and Yossi Matias, « Spectral Bloom Filters », *in*: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, Association for Computing Machinery, 2003, pp. 241–252.

[33] Chuck Cranor et al., « Gigascope: A Stream Database for Network Applications », *in*: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, San Diego, California: Association for Computing Machinery, 2003, pp. 647–651.

[34] Akash Das Sarma, Yeye He, and Surajit Chaudhuri, « ClusterJoin: A Similarity Joins Framework Using Map-Reduce », *in*: *Proc. VLDB Endow.* (2014), pp. 1059–1070.

[35] Jeffrey Dean and Sanjay Ghemawat, « MapReduce: Simplified Data Processing on Large Clusters », *in*: *Commun. ACM* 51.*1* (2008), pp. 107–113.

[36] D. Deng et al., « MassJoin: A mapreduce-based method for scalable string similarity joins », *in*: *2014 IEEE 30th International Conference on Data Engineering*, 2014, pp. 340–351.

[37] David DeWitt and Jim Gray, « Parallel Database Systems: The Future of High Performance Database Systems », *in*: *Commun. ACM* 35 (June 1992), pp. 85–98.

[38] Natalia Díaz Rodríguez et al., « A fuzzy ontology for semantic modelling and recognition of human behaviour », *in*: *Knowledge-Based Systems* 66 (2014), pp. 46–60.

[39] AnHai Doan, Alon Halevy, and Zachary Ives, *Principles of Data Integration*, 1st, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[40] Christos Doulkeridis and Kjetil NØrvåg, « A Survey of Large-Scale Analytical Query Processing in MapReduce », *in*: (2014), pp. 355–380.

[41] Tamer Elsayed, Jimmy Lin, and Douglas W. Oard, « Pairwise Document Similarity in Large Collections with MapReduce », *in*: *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, Association for Computational Linguistics, 2008, pp. 265–268.

[42] Li Fan et al., « Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol », *in*: *IEEE/ACM Trans. Netw.* 8 (June 2000), pp. 281–293.

[43] Fabian Fier et al., « Set Similarity Joins on Mapreduce: An Experimental Survey », *in*: *Proc. VLDB Endow.* 11 (2018), pp. 1110–1122.

[44] William B. Frakes and Ricardo Baeza-Yates, eds., *Information Retrieval: Data Structures and Algorithms*, Prentice-Hall, Inc., 1992.

[45] *GALACTICA*, URL: `https://galactica.isima.fr/`.

[46] Yantao Gan, Xiaofeng Meng, and Yingjie Shi, « Processing Online Aggregation on Skewed Data in Mapreduce », *in*: *Proceedings of the Fifth International Workshop on Cloud Data Management*, CloudDB '13, San Francisco, California, USA: Association for Computing Machinery, 2013, pp. 3–10.

[47] Georges Gardarin and Patrick Valduriez, « Join and Semijoin Algorithms for a Multiprocessor Database Machine », *in*: *ACM Transactions on Database Systems* 9 (Mar. 1984), pp. 133–161.

[48] Michael Goodrich and Michael Mitzenmacher, « Invertible Bloom Lookup Tables », *in*: Jan. 2011, pp. 792–799.

[49] Luis Gravano et al., « Approximate String Joins in a Database (Almost) for Free », *in*: *Proceedings of the 27th International Conference on Very Large Data Bases*, 2001, pp. 491–500.

[50] Luis Gravano et al., « 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, pp. 491–500.

[51] Benjamin Gufler et al., « Handling Data Skew in MapReduce. », *in*: Jan. 2011, pp. 574–583.

[52] D. Guo et al., « Theory and Network Applications of Dynamic Bloom Filters », *in*: *INFOCOM*, 2006, pp. 1–12.

[53] D. Guo et al., « The Dynamic Bloom Filters », *in*: *IEEE Transactions on Knowledge & Data Engineering* 22.*01* (2010).

[54] R. W. Hamming, « Error detecting and error correcting codes », *in*: *The Bell System Technical Journal* 29 (1950), pp. 147–160.

[55] M. Al Hajj 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*, SAC '10, Association for Computing Machinery, 2010, pp. 406–413.

[56] Monika Henzinger, « Finding Near-duplicate Web Pages: A Large-scale Evaluation of Algorithms », *in*: *SIGIR*, 2006, pp. 284–291.

[57] Benjamin Hindman et al., « Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center », *in*: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, Boston, MA: USENIX Association, 2011, pp. 295–308.

[58] Timothy C. Hoad and Justin Zobel, « Methods for Identifying Versioned and Plagiarized Documents », *in*: *JASIST* 54.*3* (2003), pp. 203–215.

[59] Timothy C. Hoad and Justin Zobel, « Methods for Identifying Versioned and Plagiarized Documents », *in*: *JASIST* 54 (2003), pp. 203–215.

[60] Alex Holmes, *Hadoop in Practice*, Manning Publications Co., 2012.

[61] Stratos Idreos, Erietta Liarou, and Manolis Koubarakis, « Continuous Multi-Way Joins over Distributed Hash Tables », *in*: *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, Association for Computing Machinery, 2008, pp. 594–605.

[62] Anil Jain, Karthik Nandakumar, and Abhishek Nagar, « Biometric Template Security », *in*: *EURASIP Journal on Advances in Signal Processing* (2008).

[63] Yu Jiang et al., « String Similarity Joins: An Experimental Evaluation », *in*: *Proc. VLDB Endow.* 7 (2014), pp. 625–636.

[64] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii, « A Model of Computation for MapReduce », *in*: *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, Society for Industrial and Applied Mathematics, 2010, pp. 938–948.

[65] B. Kimmett, A. Thomo, and V. Srinivasan, « Fuzzy joins in MapReduce: Edit and Jaccard distance », *in*: *IISA*, 2016, pp. 1–6.

[66] Ben Kimmett, Venkatesh Srinivasan, and Alex Thomo, « Fuzzy Joins in MapReduce: An Experimental Study », *in*: *PVLDB* 8.*12* (2015), pp. 1514–1517.

[67] Adam Kirsch and Michael Mitzenmacher, « Distance-Sensitive Bloom Filters », *in*: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, Society for Industrial and Applied Mathematics, 2006, pp. 41–50.

[68] Abhishek Kumar et al., « Space-Code Bloom Filter for Efficient Traffic Flow Measurement », *in*: *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, Association for Computing Machinery, 2003, pp. 167–172.

[69] YongChul Kwon et al., « Managing Skew in Hadoop », *in*: *IEEE Data Eng. Bull.* 36 (2013), pp. 24–33.

[70] Y Kwon et al., « Managing skew in hadoop », *in*: *IEEE Data Eng. Bull* 36 (2013), pp. 24–33.

[71] Avinash Lakshman and Prashant Malik, « Cassandra: A Decentralized Structured Storage System », *in*: (2010), pp. 35–40.

[72] Kyong-Ha Lee et al., « Parallel Data Processing with MapReduce: A Survey », *in*: *SIGMOD Rec.* 40 (2012), pp. 11–20.

[73] Taewhi Lee, Kisung Kim, and Hyoung-Joo Kim, « Join Processing Using Bloom Filter in MapReduce », *in*: *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, Association for Computing Machinery, 2012, pp. 100–105.

[74] Marie-Jeanne Lesot, Maria Rifqi, and Hamid Benhadda, « Similarity measures for binary and numerical data: a survey », *in*: *International Journal of Knowledge Engineering and Soft Data Paradigms* 1.*1* (Dec. 2008), pp. 63–84.

[75] Feng Li et al., « Distributed Data Management Using MapReduce », *in*: (2014).

[76] Qiming Li, Yagiz Sutcu, and Nasir Memon, « Secure Sketch for Biometric Templates », *in*: 2006, pp. 99–113.

[77] X. Lian and L. Chen, « Efficient Similarity Join over Multiple Stream Time Series », *in*: *IEEE Transactions on Knowledge and Data Engineering* 21.*11* (2009), pp. 1544–1558.

[78] Jimmy Lin, Shravya Konda, and Samantha Mahindrakar, *Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework*, 2009.

[79] Jiaheng Lu et al., « String Similarity Measures and Joins with Synonyms », *in*: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 373–384.

[80] Wei Lu et al., « Efficient Processing of k Nearest Neighbor Joins Using MapReduce », *in*: *Proc. VLDB Endow.* 5 (2012), pp. 1016–1027.

[81] Lailong Luo et al., « Optimizing Bloom Filter: Challenges, Solutions, and Comparisons », *in*: *IEEE Communications Surveys & Tutorials* PP (Apr. 2018).

[82] Lothar F. Mackert and Guy M. Lohman, « R* Optimizer Validation and Performance Evaluation for Distributed Queries », *in*: *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, Morgan Kaufmann Publishers Inc., 1986, pp. 149–159.

[83] *MapReduce Benchmarks*, URL: `https://engineering.purdue.edu/~puma/pumabenchmarks.htm`.

[84] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi, « Detectives: Detecting Coalition Hit Inflation Attacks in Advertising Networks Streams », *in*: *WWW*, 2007, pp. 241–250.

[85] Ahmed Metwally and Christos Faloutsos, « V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors », *in*: *CoRR* abs/1204.6077 (2012), URL: `http://arxiv.org/abs/1204.6077`.

[86] M. Mitzenmacher, « Compressed Bloom filters », *in*: *IEEE/ACM Transactions on Networking* 10.*5* (2002), pp. 604–612.

[87] Alper Okcan and Mirek Riedewald, « Processing Theta-joins Using MapReduce », *in*: *SIGMOD*, 2011, pp. 949–960.

[88] Carlos Ordonez, « Optimizing Recursive Queries in SQL », *in*: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, Association for Computing Machinery, 2005, pp. 834–839.

[89] Thuong-Cang Phan, « Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce », Theses, Université Blaise Pascal - Clermont-Ferrand II, July 2014.

[90] Thuong-Cang Phan, Laurent d'Orazio, and Philippe Rigaux, « Toward Intersection Filter-based Optimization for Joins in MapReduce », *in*: *Cloud-I*, 2013, 2:1–2:2.

[91] Thuong-Cang Phan, Laurent d'Orazio, and Philippe Rigaux, « A Theoretical and Experimental Comparison of Filter-Based Equijoins in MapReduce », *in*: *TLDKS* 25 (2016), pp. 33–70.

[92] Smriti R. Ramakrishnan, Garret Swart, and Aleksey Urmanov, « Balancing Reducer Skew in MapReduce Workloads Using Progressive Sampling », *in*: *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, San Jose, California: Association for Computing Machinery, 2012.

[93] C. Rong et al., « Efficient and Scalable Processing of String Similarity Join », *in*: *IEEE Transactions on Knowledge and Data Engineering* 25 (2013), pp. 2217–2230.

[94] Chuitian Rong et al., « Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics », *in*: Apr. 2017, pp. 1059–1070.

[95] Mehran Sahami and Timothy D. Heilman, « A Web-based Kernel Function for Measuring the Similarity of Short Text Snippets », *in*: *WWW*, 2006, pp. 377–386.

[96] Sunita Sarawagi and Alok Kirpal, « Efficient Set Joins on Similarity Predicates », *in*: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, 2004, pp. 743–754.

[97] Yasin N. Silva and Jason M. Reed, « Exploiting MapReduce-based Similarity Joins », *in*: *SIGMOD*, ACM, 2012, pp. 693–696.

[98] Yasin N. Silva, Jason M. Reed, and Lisa M. Tsosie, « MapReduce-based Similarity Join for Metric Spaces », *in*: *Cloud-I*, 2012, 3:1–3:8.

[99] Yasin N. Silva et al., « An Experimental Survey of MapReduce-Based Similarity Joins », *in*: *Similarity Search and Applications*, 2016, pp. 181–195.

[100] Ellen Spertus, Mehran Sahami, and Orkut Buyukkokten, « Evaluating similarity measures: A large-scale study in the Orkut social network », *in*: *SIGKDD*, 2005, pp. 678–684.

[101] S. Subramaniam et al., « Online Outlier Detection in Sensor Data Using Non-Parametric Models », *in*: *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, Seoul, Korea: VLDB Endowment, 2006, pp. 187–198.

[102] Yagiz Sutcu, Qiming Li, and Nasir Memon, « Protecting Biometric Templates With Sketch: Theory and Practice », *in*: *Information Forensics and Security, IEEE Transactions on* (2007), pp. 503–512.

[103] Kian-Lee Tan and Hongjun Lu, « A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems », *in*: *SIGMOD Rec.* (1991), pp. 81–82.

[104] Thi-To-Quyen Tran et al., « 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.

[105] Thi-To-Quyen Tran et al., « Optimization for Large-Scale Fuzzy Joins Using Fuzzy Filters in MapReduce », *in*: *29th IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 2020, Glasgow, UK, July 19-24, 2020*, IEEE, 2020, pp. 1–8.

[106] Pim Tuyls et al., « Practical Biometric Authentication with Template Protection », *in*: 2005, pp. 436–446.

[107] Esko Ukkonen, « Algorithms for approximate string matching », *in*: *Information and Control* (1985), pp. 100–118.

[108] Vinod Kumar Vavilapalli et al., « Apache Hadoop YARN: Yet Another Resource Negotiator », *in*: *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California: Association for Computing Machinery, 2013.

[109] Rares Vernica, Michael J. Carey, and Chen Li, « Efficient Parallel Set-similarity Joins Using MapReduce », *in*: *SIGMOD*, 2010, pp. 495–506.

[110] Sebastian Wandelt et al., « RCSI: Scalable Similarity Search in Thousand(s) of Genomes », *in*: *Proc. VLDB Endow.* 6 (2013), pp. 1534–1545.

[111] J. Wang et al., « LS-Join: Local Similarity Join on String Collections », *in*: *IEEE Transactions on Knowledge and Data Engineering* 29.*9* (2017), pp. 1928–1942.

[112] Jingdong Wang et al., « Hashing for Similarity Search: A Survey », *in*: (Aug. 2014).

[113] 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.

[114] Tom White, *Hadoop: The Definitive Guide*, 4th, O'Reilly Media, Inc., 2015.

[115] Chuan Xiao et al., « Efficient Similarity Joins for Near-duplicate Detection », *in*: *ACM TODS* 36.*3* (2011), 15:1–15:41, (visited on 11/20/2017).

[116] Minghe Yu et al., « String Similarity Search and Join: A Survey », *in*: 10 (2016), pp. 399–417.

[117] Matei Zaharia et al., « Spark: Cluster Computing with Working Sets », *in*: *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*, USENIX Association, 2010, p. 10.

[118] Matei Zaharia et al., « Resilient Distributed Datasets: A Fault-Tolerant Abstraction for in-Memory Cluster Computing », *in*: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, USENIX Association, 2012, p. 2.

[119] Huanchen Zhang et al., « 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, pp. 323–336.

[120] Yunyue Zhu and Dennis Shasha, « StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time », *in*: *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, Hong Kong, China: VLDB Endowment, 2002, pp. 358–369.

[121] Yunyue Zhu and Dennis Shasha, « Efficient Elastic Burst Detection in Data Streams », *in*: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, Washington, D.C.: Association for Computing Machinery, 2003, pp. 336–345.

UNIVERSITÉ DE RENNES 1

Titre : Jointures Floues Massives Basé Aux Filtres

Mot clés : Jointure floue, MapReduce, Filtre floue

Résumé : Une jointure floue est l'une des opérations de traitement et d'analyse de données les plus utiles pour le Big Data dans un contexte général. Il combine des paires de tuples dont la distance est inférieure ou égale à un seuil donné. La jointure floue est utilisée dans de nombreuses applications pratiques, mais elle est extrêmement coûteuse en temps et en espace, et peut même ne pas être exécutée sur des ensembles de données à grande échelle. Bien qu'il y ait eu quelques études pour améliorer ses performances en appliquant des filtres, une solution d'un filtre flou efficace pour la jointure floue n'a jamais été réalisée. Dans cette thèse, nous nous concentrons donc sur l'optimisation des grandes jointures floues à l'aide de filtres.

Pour atteindre ces objectifs, nous appliquons d'abord des filtres Bloom aux opérations de grande jointure floue afin d'éliminer la plupart des éléments non joints dans les ensembles de données d'entrée avant d'envoyer les données au traitement de jointure réel. Ainsi, il réduit les données intermédiaires redondantes, les comparaisons inutiles et évite la duplication des données.

Une autre proposition importante est les filtres flous, qui sont des structures de données probabilistes conçues pour représenter des ensembles et leurs éléments de similarité. Ils sont utilisés pour détecter rapidement les éléments proches de l'ensemble dans des seuils donnés avec un faible taux de faux positifs et zéro taux de faux négatifs. De plus, ces filtres sont non seulement indépendants avec des seuils et des fonctions de mesure de distance, mais aussi facilement mis à jour. Par conséquent, il peut être appliqué à un large éventail de problèmes courants tels que la déduplication, la correction d'erreurs, le nettoyage des données, l'intégration de données, les jointures récursives et les jointures de flux. Notre amélioration réduira considérablement le nombre de calculs redondants et les frais généraux associés tels que les données intermédiaires et la communication pour les opérations de déduplication.

Nous effectuons ensuite des comparaisons d'analyse des algorithmes de jointure floue convaincantes sur la base d'un modèle de coût $M - C - R$. En conséquence, en utilisant les filtres proposés, les opérations de jointure floue peuvent minimiser les coûts d'E/S disque et de communication. De plus, les opérations de jointure floue basées sur des filtres se sont avérées plus efficaces que les solutions existantes grâce à des évaluations expérimentales dans Spark. Des comparaisons expérimentales de différents algorithmes pour les jointures floues sont examinées en ce qui concerne la quantité de données intermédiaires, la quantité totale de sortie, le temps total d'exécution et en particulier les calendriers des tâches.

Bref, nos améliorations sur les opérations de jointure contribuent à la scène mondiale d'optimisation de la gestion des données pour les applications MapReduce sur des infrastructures distribuées à grande échelle.

Title: Filters Based Fuzzy Big Joins

Keywords: Fuzzy join, Similarity join, MapReduce, Fuzzy Filter

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. 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 thesis, we thus focus on optimizing fuzzy big joins using filters.

To achieve these objectives, we first apply Bloom filters to fuzzy big join operations to eliminate most non-joining elements in input datasets before sending data to actual join processing. Thus, it reduces redundant intermediate data, unnecessary comparisons and avoid the data duplication.

Another important proposal is Fuzzy Filters, which are probabilistic data structures designed to represent set(s) and its similarity elements. They are used to fast detect close elements of the set in given thresholds with small false positive rate and zero false negative rate. Moreover, these filters are not only independent with thresholds and distance measure functions but also easily updated. Therefore, it can be applied to a wide range of popular problems such as deduplication, error-correction, data cleaning, data integration, recursive joins and stream joins. Our improvement will significantly reduce the number of redundant computations, and the related overheads such as intermediate data, and communication for the deduplication operations.

We then make analysis comparisons of the fuzzy join algorithms persuasive based on a $M - C - R$ cost model. As a result, with using the proposed filters, the fuzzy join operations can minimize disk I/O and communication costs. Moreover, the filters based fuzzy join operations are demonstrated to be more efficient than existing solutions through experimental evaluations in Spark. Experimental comparisons of different algorithms for fuzzy joins are examined with respect to intermediate data amount, the total output amount, the total execution time, and especially task timelines.

Summary, our improvements on the join operations contribute to the global scene of optimizing data management for MapReduce applications on large-scale distributed infrastructures.