



# Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce

Thuong-Cang Phan

## ► To cite this version:

Thuong-Cang Phan. Optimization for big joins and recursive query evaluation using intersection and difference filters in MapReduce. Other [cs.OH]. Université Blaise Pascal - Clermont-Ferrand II, 2014. English. NNT : 2014CLF22474 . tel-01066612

**HAL Id: tel-01066612**

**<https://theses.hal.science/tel-01066612>**

Submitted on 22 Sep 2014

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

**BLAISE PASCAL UNIVERSITY - CLERMONT II**

Engineering Doctoral School of Clermont Ferrand

LIMOS - CNRS UMR 6158

**P H D T H E S I S**

To obtain the degree of

**Doctor of Philosophy**

**Specialty: Computer Science**

Defended by

**Thuong-Cang PHAN**

---

**Optimization for Big Joins and Recursive  
Query Evaluation using Intersection and  
Difference Filters in MapReduce**

---

*publicly defended on July 07<sup>th</sup>, 2014*

**Committee:**

*Reviewers:*

- |                           |  |
|---------------------------|--|
| Pr. Dominique Laurent     | - University of Cergy-Pontoise, France |
| Dr. Genoveva Vargas Solar | - (HDR) LIG-LAFMIA, CNRS, France       |

*Examiners:*

- |                       |                                       |
|-----------------------|---------------------------------------|
| Pr. Mohand-Said Hacid | - University C.Bernard Lyon1, France  |
| Pr. Farouk Toumani    | - University of Blaise Pascal, France |

*Advisors:*

- |                      |                                       |
|----------------------|---------------------------------------|
| Pr. Philippe Rigaux  | - CNAM Paris, France                  |
| Dr. Laurent d'Orazio | - University of Blaise Pascal, France |

## Author



**Thuong-Cang PHAN** - Ph.D student.

Email: [ThuongCang.PHAN@isima.fr](mailto:ThuongCang.PHAN@isima.fr)

[ptcang@cit.ctu.edu.vn](mailto:ptcang@cit.ctu.edu.vn)

[ptcang@gmail.com](mailto:ptcang@gmail.com)

Thuong-Cang earned a bachelor's degree from Can Tho University of Vietnam in 1998. Then, he received the M.E. degree in Computer Science from Asian Institute of Technology, Thailand, in 2006. He obtained his PhD in Computer Science from Blaise Pascal University, LIMOS UMR 6158, CNRS, France (17/10/2011-07/07/2014). His research interests include probabilistic data structures, Big data, Big joins, large-scale recursive queries, grid computing, cloud computing, SOA, MapReduce, web service, semantic web, and information systems.

# **Declaration**

This dissertation has been completed by Thuong-Cang PHAN under the supervision of Professor Philippe RIGAUX and Assoc. Professor Laurent D’ORAZIO and has not been submitted for any other degree or professional qualification. I declare that the work presented in this dissertation is entirely my own except where indicated by full references.

SIGNATURE

# Acknowledgements

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 Philippe RIGAUX and Assoc. Professor Laurent D’ORAZIO 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 and mentors.

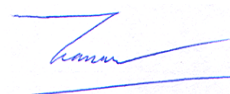
Besides my advisors, I would also like to thank my reviewers, Professor Dominique LAURENT and Madam (HDR) Genoveva VARGAS-SOLAR, who provided encouraging and constructive feedbacks. It is not an easy task when reviewing a thesis, and I am grateful for their thoughtful and detailed comments. I would like to thank the rest of my thesis committee: Professor Farouk TOUMANI and Professor Mohand-Said HACID, for their encouragement, insightful comments, and critical questions.

I would like to thank Blaise Pascal University, LIMOS and ISIMA Lab, especially Ms. Pascale Gouinaud and Mr. Antoine Mahul (CRRI Clermont Université), who were responsible for making sure the cluster used to run the experiments worked fine. I also thank my fellow labmates in Big Data Group, for the discussions, and for all the fun we have had in the last three years.

I always remember the share of my brother, PHAN Anh-Cang, and my Vietnamese colleague, TRAN T.T. Quyen, who shared the work and stress during my most difficult moments.

Last but not least, I would like to thank my family: my parents-in-law, particularly my parents NGUYEN Thi-Tiep and PHAN Van-Mua, who always supported me spiritually throughout my life and in all my pursuits; my loving wife, TANG Dinh Ngoc Thao, who has been very patient, supportive and encouraging throughout my PhD; and most of all, my little son, PHAN Thuong-Lam, who kept me smiling during tough times in the PhD pursuit. Thank you so much !

Clermont-Ferrand, July 07 2014



Thuong-Cang PHAN

# Abstract

The information technology community has created unprecedented amount of data through large-scale applications. As a result, the Big Data is considered as gold mines of information that just wait for the processing power to be available, reliable, and apt at evaluating complex analytic algorithms. MapReduce is one of the most popular programming models designed to support such processing. It has become a standard for processing, analyzing and generating large data in a massively parallel manner. However, the MapReduce programming model suffers from severe limitations of operations beyond simple scan/grouping, particularly operations with multiple inputs. In the present dissertation we efficiently investigate and optimize the evaluation, in a MapReduce environment, of one of the most salient and representative such operations: *Join*. It focuses not only on two-way joins, but also complex joins such as multi-way joins and recursive joins.

To achieve these objectives, we first devise a new type of filter called *intersection filter* using a probabilistic model to represent an approximation of the set intersection. The intersection filter is then applied to two-way join operations to eliminate most non-joining elements in input datasets before sending data to actual join processing. In addition, we make an extension of the intersection filter to improve the performance of three-way joins and chain joins including both cyclic chain joins with many shared join keys. We use the Lagrangian multiplier method to indicate a good choice between our optimized solutions for the multi-way joins.

Another important proposal is a *difference filter*, which is a probabilistic data structure designed to represent a set and examine disjoint elements of the set. It can be applied to a wide range of popular problems such as reconciliation, deduplication, error-correction, especially a recursive join operation. A recursive join using the difference filter is implemented as an iteration of one join job instead of two jobs including a join job and a difference job. This improvement will significantly reduce the number of executed jobs by half, and the related overheads such as data rescanning, intermediate data, and communication for the deduplication and difference operations. Besides, this research also improves the general semi-naive algorithm, as well as the evaluation of recursive queries in MapReduce.

We then provide general cost models for two-way joins, multi-way joins, and recursive joins. Thanks to these cost models, we can make comparisons of the join algorithms more persuasive. As a result, with using the proposed filters, the join operations can minimize disk I/O and communication costs. Moreover, the intersection filter-based join operations are demonstrated to be more efficient than existing solutions through experimental evaluations. Experimental comparisons of different algorithms for joins are examined with respect to intermediate data amount, the total output amount, the total execution time, and especially task timelines.

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

**Key words:** Big data, MapReduce, Bloom filter, Join, Recursive query evaluation, Optimization.

# Résumé

La communauté informatique a créé une quantité de données sans précédent grâce aux applications à grande échelle. Ces données massives sont considérées comme une mine d'or, ces informations n'attendant que la puissance de traitement sûre et appropriée à l'évaluation d'algorithmes d'analyse complexe. MapReduce est un des modèles de programmation les plus réputés, connu pour la gestion de ce type de traitement. Il est devenu un standard pour le traitement, l'analyse et la génération de grandes quantités de données en parallèle. Cependant, le modèle de programmation MapReduce souffre d'importantes limites pour des opérations non simples (scans ou regroupements simples), en particulier les traitements avec entrées multiples. Dans ce mémoire, nous étudions et optimisons l'évaluation, dans un environnement MapReduce, d'une des opérations les plus importantes et représentatives : la jointure. Notre travail aborde, en plus de la jointure binaire, des jointures complexes comme la jointure multidimensionnelle et la jointure récursive.

Pour atteindre ces objectifs, nous proposons d'abord un nouveau type de filtre appelé *filtre d'intersection* qui utilise un modèle probabiliste pour représenter une approximation de l'intersection des ensembles. Le filtre d'intersection est ensuite appliqué à l'opération de jointure bidirectionnelle pour éliminer la majorité des éléments non-joints dans des ensembles de données d'entrée, avant d'envoyer les données pour le processus de jointure. De plus, nous proposons une extension du filtre d'intersection pour améliorer l'efficacité de la jointure ternaire et de la jointure en cascade correspondant à un cycle de jointure avec plusieurs clés partagées lors de la jointure. Nous utilisons la méthode des *multiplieurs de Lagrange* afin de réaliser un choix pertinent entre les différentes solutions proposées pour les jointures multidimensionnelles.

Une autre proposition est le *filtre de différence*, une structure de données probabiliste formée pour représenter un ensemble et examiner des éléments disjoints. Ce filtre peut être appliqué à un grand nombre de problèmes, tels que la réconciliation, la déduplication, la correction d'erreur et en ce qui nous concerne la jointure récursive. Une jointure récursive utilisant un filtre de différence est effectuée comme une répétition de jointures en lieu et place d'une jointure et d'un processus de différenciation. Cette amélioration réduit de moitié le nombre de tâches effectuées et les associés tels que la lecture des données, la génération des données intermédiaire et les communications. Ceci permet notamment une amélioration de l'évaluation de l'algorithme semi-naïf et par conséquent l'évaluation des requêtes récursives en MapReduce.

Ensuite, nous fournissons des modèles de coût généraux pour les jointures binaire, à n-aire et récursive. Grâce à ces modèles, nous pouvons comparer les algorithmes de jointure les plus représentatifs. Ainsi, nous pouvons montrer l'intérêt des filtres proposés, grâce notamment à la réduction des coûts E/S (entrée/ sortie) sur disque et sur réseau. De plus, des expérimentations ont été menées, montrant l'efficacité du filtre d'intersection par rapport aux solutions, en comparant en particulier des critères tels que la quantité de données intermédiaires, la quantité de données produites en sortie, le temps d'exécution et la répartition des tâches.

Nos propositions pour les opérations de jointure contribuent à l'optimisation en général de la gestion de données à l'aide du paradigme MapReduce sur des infrastructures distribuées à grande échelle.

**Mots clés:** données massives, MapReduce, Filtre Bloom, Jointure, évaluation de requêtes récursives, optimisation.



# Contents

<b>INTRODUCTION.....</b>	<b>1</b>
1.1 Context and motivation.....	1
1.2 Goal of the thesis .....	3
1.3 Thesis outline .....	5
 <b>BACKGROUND AND RELATED WORKS.....</b>	 <b>8</b>
2.1 Background .....	8
2.1.1 Join operation .....	8
2.1.2 MapReduce framework.....	11
2.1.3 Parallelization of a join operation in MapReduce .....	13
2.1.4 Iteration in MapReduce .....	15
2.2 Basic join algorithms in MapReduce.....	17
2.2.1 Map-side join.....	17
2.2.2 Reduce-side join .....	19
2.2.3 Broadcast join.....	22
2.2.4 Semi-join.....	24
2.3 Bloomjoin algorithm in MapReduce .....	27
2.3.1 Bloom filter .....	27
2.3.2 Bloomjoin algorithm description.....	31
2.4 Summary .....	34
 <b>OPTIMIZATION FOR TWO-WAY JOINS AND IMPORTANT MULTI-WAY JOINS .....</b>	 <b>38</b>
3.1 Introduction .....	38
3.1.1 Previous work.....	38
3.1.2 Definitions and notations .....	40
3.2 Modeling intersection filter.....	41
3.2.1 Approach 1: a pair of Bloom filters.....	41
3.2.2 Approach 2: intersecting unpartitioned Bloom filters .....	42
3.2.3 Approach 3: intersecting partitioned Bloom filters .....	43
3.2.4 The false intersection probability .....	45

3.3	Optimization for two-way joins using intersection filters in MR .....	47
3.3.1	Implementation overview .....	47
3.3.2	Optimized two-way join algorithm .....	50
3.3.3	Cost analysis for two-way joins in MapReduce.....	52
3.4	Optimization for multi-way joins using intersection filters in MR .....	56
3.4.1	Extended intersection filter .....	56
3.4.2	Three-way join using intersection filter .....	56
3.4.3	Chain join using intersection filter .....	62
3.4.4	Star join using intersection filter .....	65
3.4.5	Cost analysis of three-way joins in MapReduce .....	66
3.4.6	Cost analysis of chain joins in MapReduce .....	69
3.5	Experimental evaluation .....	70
3.5.1	Two-way joins .....	70
3.5.2	Chain joins .....	77
3.6	Summary .....	81

## **OPTIMIZATION FOR RECURSIVE JOINS AND SEMI-NAIVE ALGORITHM..... 83**

4.1	Introduction.....	83
4.1.1	Previous work.....	83
4.1.2	Proposal for recursive join using filters.....	86
4.1.3	Definitions and notations .....	88
4.2	Modeling difference filter .....	89
4.2.1	Existing solutions .....	89
4.2.2	Problem definition .....	94
4.2.3	Difference filter design .....	95
4.2.4	Dynamic Bloom Filter .....	98
4.2.5	False difference probability.....	99
4.3	Optimizing recursive joins and semi-naive algorithm.....	105
4.3.1	Implementation model .....	105
4.3.2	Optimized semi-naive algorithm in MapReduce.....	108
4.4	Cost analysis for recursive joins.....	113
4.4.1	Cost model .....	113
4.4.2	Cost comparison .....	115
4.5	Summary .....	117

## **CONCLUSIONS AND FUTURE WORK..... 120**

5.1	Thesis conclusions .....	120
5.2	Discussion and future work.....	122
5.2.1	Two-way and multi-way joins.....	122
5.2.2	Recursive joins .....	125
5.2.3	Query language for NoSQL databases.....	128

## **Bibliography..... 130**

# List of Figures

2.1: Types of joins .....	10
2.2: MapReduce Execution.....	11
2.3: Difference between traditional parallelism and MapReduce.....	13
2.4: Parallel implementation of the join operation in MapReduce .....	15
2.5: HaLoop vs. Hadoop Programming Model [13] .....	16
2.6: Map-side join in MapReduce.....	17
2.7: Reduce-side join in MapReduce .....	20
2.8: Broadcast join in MapReduce .....	22
2.9: Semi-join in MapReduce .....	25
2.10: A Bloom filter $BF(S)$ with 3 hash functions.....	28
2.11: Approximate representation of $S$ with false positives.....	28
2.12: False positive rate and number of hash functions .....	29
2.13: Bloomjoin in MapReduce .....	32
3.1: Basic join operation using $BF$ in MapReduce .....	39
3.2: Intersection Filter returning an output with two possibilities .....	41
3.3: Intersection filter using a pair of Bloom filters.....	42
3.4: Intersection filter based on intersecting <i>unpartitioned</i> Bloom filters.....	43
3.5: Partitioned Bloom filter $BF(S)$ .....	43
3.6: Intersection filter based on intersecting <i>partitioned</i> Bloom filters .....	44
3.7: Set intersection representation using Bloom filters.....	45
3.8: Join implementation using intersection filter in MapReduce .....	47
3.9: Extended intersection filter - $EIF(BF_1, BF_2, ..., BF_k)$ .....	56
3.10: Distributing tuples of $R$ , $K$ , and $L$ among $r = m^2$ reducers.....	57
3.11: Three-way join operation using intersection filter .....	58
3.12: A chain join .....	62
3.13: Implementation of a chain join using a Bloomjoin cascade .....	62
3.14: Implementation of a chain join using a cascade of two-way joins using intersection filters.....	63
3.15: Optimization of a chain join using extended intersection filters.....	64
3.16: A star join .....	65
3.17: Implementation of a star join .....	66
3.18: Comparison of Map output among the intersection filter-based joins .....	73
3.19: Total execution time .....	74
3.20: 70GB Task timelines during the execution of the join job .....	75
3.21: Threshold of redundant data amount for the joins with 2GB inputs.....	76
3.22: Total intermediate data.....	79
3.23: Total output data (Map output + Reduce output) .....	80
3.24: Total execution time .....	81

4.1: Relationship between join and dup-elim tasks.....	84
4.2: Semi-naive implementation of recursive joins in MapReduce .....	86
4.3: Filter-based optimization for the semi-naive algorithm and recursive joins .....	87
4.4: Invertible Bloom filter .....	91
4.5: InvBF Subtraction. ....	92
4.6: Difference filter returning an output with three possibilities .....	94
4.7: Design of difference filter .....	95
4.8: Example of difference filter .....	96
4.9: Dynamic Bloom filter $DBF(F_{i-1})$ .....	98
4.10: Output of difference filter .....	99
4.11: Representation of elements on each filtering level.....	100
4.12: Graph of the false difference probability with $m_2=100$ .....	102
4.13: False difference probability with variation of $n_r$ and $m_2$ .....	103
4.14: Example of filtering different elements .....	104
4.15: Pre-processing job for building initial filters .....	105
4.16: Recursive Join execution based on Intersection-Diff filter in MapReduce ....	106
5.1. A kind of Bushy join trees in PM approach .....	124
5.2. Two kinds of join trees in CM approach.....	124
5.3. Data skew in recursive join .....	127

# List of Tables

Table 3.1: List of notations .....	40
Table 3.2: Cost model parameters for two-way joins.....	52
Table 3.3: Input datasets used in three tests.....	71
Table 3.4: The number of intermediate tuples (Map output).....	72
Table 3.5: Parameters of filters used in experiments.....	72
Table 3.6: Execution of pre-processing job and join job.....	74
Table 3.7: Input datasets used in three tests.....	77
Table 3.8: Parameters of filters used in experiments.....	78
Table 3.9: The total number of intermediate tuples (all map outputs) .....	79
Table 4.1: List of notations .....	89
Table 4.2: Cost model parameters for recursive join.....	113

---

## INTRODUCTION

### 1.1 Context and motivation

Since the advent of applications that proposes Web-based services to a worldwide population of connected people, the information technology community has been confronted to unprecedented amount of data, either resulting from an attempt to organize an access to the Web information space (search engines), or directly generated by this massive amount of users (e.g., social networks). Companies like Google or Facebook, representative of those two distinct trends, have developed for their own needs data processing platforms that combine an infrastructure based on millions of servers, data repositories where the least collection size is measured in Petabytes, and finally data processing softwares that massively exploit distributed computing and batch processing to scale at the required level of magnitude. Besides, although the Web is a primary source of information production, the Big Data Issue can now be generalized to other areas that constantly collect data and attempt to make sense of it. Sensors incorporated in electronic devices, satellite images, web server logs, bioinformatics are now considered as gold mines of information that just wait for the processing power to be available, reliable, and apt at evaluating complex analytic algorithms.

The MapReduce programming model [1], published ten years ago, has become a standard for processing, analyzing and generating large data in a massively parallel manner. Its success comes from its simplicity: users only define a map function that maps a key/value pair into intermediate key/value pair(s), and a reduce function that processes all intermediate values associated with the same intermediate key. In addition, MapReduce proposes an abstraction of the underlying parallel execution, and enjoys nice properties in terms of fault tolerance, a necessary feature when hundreds or even thousands of commodity machines are involved in a job that may extend of days or weeks.

However, the MapReduce programming model suffers from severe limitations when it comes to implement algorithms that require data access patterns beyond simple scan/grouping operation. In particular, it is not suited for operations with multiple inputs. In the present dissertation we efficiently investigate and optimize the evaluation, in a MapReduce environment, of one of the most salient and representative such operations: *Join*. A join combines related tuples from datasets on different column schemes and thus raises at a generic level the problem of combining several inputs with a programming framework initially design for scanning, processing and grouping a single dataset. Joins are basic building blocks used in many sophisticated data mining algorithms. An important first step toward the efficient processing of the large-scale data analysis is therefore the optimization of the join operation.

This problem has become a hot research topic in recent years [2][3][4][5][6][7][8][9]. Many studies have been conducted so far on join calculating in a MapReduce environment. Two main classifications of the join computing operation includes *Map-side join* [10][2][3] and *Reduce-side join* [10][2][3]. Besides, some variants and improvements of the join operation like broadcast join [3], semi-join [3], Bloomjoin [4][11], multi-way join [6], and recursive join [7][12][13][14] have been proposed.

Although joins in MapReduce can be implemented in many ways, the relative performance of the various algorithms depends on certain assumptions such as the size of inputs, strict constraints on data, joined rates between inputs, etc. Map-side joins would be better to perform the entire joining operation in the map phase since it may save the shuffle and reduce phases. But this solution is also limited in running extra MapReduce jobs to repartition the data sources to be usable. Meanwhile, Reduce-side joins are more flexible and general to process a join operation as a standard MapReduce job without any constraints, but they are quite inefficient solutions. Joining does not take place until the reduce phase. In addition, the shuffle phase is really expensive since it needs to shuffle all data, sort and merge.

Observing Reduce-side joins shows that many intermediate pairs generated in the map phase may not actually participate in the joining process due to no matching with any pairs in another input dataset. Consequently, it would be much more efficient if we eliminate the non-joining data right in the map phase. This problem can be solved by using a distributed cache to disseminate a hashmap of one of input datasets across all the mappers, then dropping tuples whose join key not in the hashmap. The main obstacle in this way resides at the hashmap because the hashmap may not fit in memory and its replication across all the mappers may be inefficient. In this situation, therefore, a probabilistic structure called *Bloom filter* [15] is a worthy replacement for the hashmap. It consists of an  $m$ -bit array and  $k$  distinct hash functions for doing existence tests in less memory than a full list of keys from the hashmap. However, the filtering efficiency of all the solutions has not yet been taken into consideration, even both recent research efforts [4][6]. There remain a lot of non-joining data after filtering because the filters have only the ability to filter on one of input datasets instead of both. Thus, it is necessary to have a better filter to address this problem (a).

In addition to the above two-way joins, the researchers are also confronted big challenges that come from multi-way joins and recursive joins in MapReduce. The multi-way join extends the two-way join by handling multiple input datasets, whereas the recursive join represents a computation of a repeated join operation. Both of them are still open issues and their existing solutions from traditional distributed and parallel databases cannot be easily extended to adapt to a shared-nothing distributed computing paradigm as MapReduce. For this reason, the evaluation of the complex joins has become an urgent requirement and should be thoroughly considered by the following problems.

Computing the multi-way join often generates intermediate results that may be inputs of component joins of the multi-way join. These intermediate results contain a lot of non-joining data that considerably increases total overheads for I/O, CPU, sort, merge, and especially communication. We need to figure out optimized solutions that can prevent the non-joining data involved in the intermediate results. Besides, minimizing the intermediate data amount sent to the reducers should be addressed appropriately (b).



Finally, the recursive join is a fundamental operation in computing the transitive closure that is required in many significant applications such as the reachability analysis of transition networks representing system [16], the construction of parsing automata in compiler [17], and especially recursive database queries [7][18][19]. The transitive closure (TC) of a binary relation  $K$  with attributes  $x$  and  $y$  can be defined as:

$$K^+ = \bigcup_{1 \leq i \leq l} K_i = \bigcup_{1 \leq i \leq l} (K_{i-1} \bullet K_0)$$

where  $K_0 = K$ , and

$K_{i-1} \bullet K_0$  is the composition of  $K_{i-1}$  and  $K_0$  and is identified:

$$K_{i-1} \bullet K_0 = \{(x, y) \mid \exists z (x, z) \in K_{i-1} \text{ and } (z, y) \in K_0\}$$

$l$  is the longest path length in the relation graph of  $K - 1$

The above presentation shows that the " $\bullet$ " operator is actually the projection-join of the two relations  $K_{i-1}$  and  $K_0$ , and thus is related to transitive closure algorithms. There are several major existing algorithms for computing the transitive closure of a relation. They are classified into two main groups, iterative and direct algorithms. The iterative algorithms (e.g., naive [20], semi-naive [21][22], smart [23][24], minimal evaluations [23], etc.) are applied to a tabular representation of the base relation. The main idea behind iterative algorithms is to evaluate the transitive closure breadth-first, with a loop containing algebraic expressions that derive new tuples, until no new element is generated. The direct algorithms (e.g., Warshall [25] and Warren [26] algorithms) are used for a matrix representation of a graph, and operate depth-first. In this research, we consider the semi-naive iterative algorithm in MapReduce.

The semi-naive evaluation is a simple variation of the naive evaluation. It is an algorithm for computing the least fixpoint. The main idea is that each iteration only uses new tuples derived from the previous iteration (denoted as incremental relation) to join with initial relation,  $K_0$ . This variation reduces the amount of redundant computation and duplicate data mentioned by the naive evaluation. We can specify the incremental relation by computing the difference between tuples generated in current iteration and tuples generated in previous iterations. As a result, the semi-naive is an efficient transitive closure algorithm because the cardinality of the incremental relation involved in the joins is reduced. However, the overheads of the join and difference operations are very expensive and complex in a MapReduce environment. Therefore, we should consider the optimized possibilities for recursive joins using the semi-join algorithm toward to evaluating transitive closures as well as recursive queries (c).

## 1.2 Goal of the thesis

The four main results of this dissertation are the following:

### (1) *Intersection filter and Difference filter*

Our first result is to devise two new types of Bloom filters, *Intersection filter* and *Difference filter*. The intersection filter is a method for representing the intersection of two sets, which is used to test whether an element is a common member of the two sets with a false *positive* rate. Meanwhile, the

difference filter represents a set, which is used to check for set difference (NOT membership) with a false *negative* rate. Namely, three approaches to building the intersection filter and one approach for the difference filter are proposed in this research.

Based on space efficiency, our filters can be applied to deal with a wide range of common problems such as join operation in databases [4][5][6][27]; reconciliation and deduplication in networking and distributed systems [28][29][30][31], bioinformatics [32][33], as well as databases [34][35][36]; error-correction in networking applications [37][38][39]; etc.

(2) *Optimization for two-way and multi-way joins and cost models*

Our second result is that optimizations for two-way joins and multi-way joins using the different approaches of the intersection filter are more efficient than the prior join algorithms [5]. This is because our join operations can eliminate most non-joining data in both input datasets before sending them to the actual join processing. As a consequence, intermediate results of multi-way join operations now contain no redundant data, reducing significantly the associated overheads.

An interesting characteristic of the intersection filter-based join operations is that they can be completed without doing anything if the intersection filter is empty (i.e. joined input datasets are distinct).

Besides, we also make an extension of the intersection filter to improve the performance of three-way joins and chain joins including both cyclic chain joins with many shared join keys. Thanks to the Lagrangian multiplier method, we can indicate a good choice between our optimized solutions.

Moreover, our intersection filter-based join algorithms are developed by pseudo codes.

Lastly, two cost models for two-way joins and multi-way joins are then proposed to compare among the join algorithms more convincing.

All these help us address the problems (a) and (b).

(3) *Optimization for recursive join and cost model*

Our third result is to provide a simple and efficient solution for optimizing the general semi-naive algorithm as well as the recursive join in MapReduce. This solution uses the intersection and difference filters to compute the join operation and the incremental relation all in one job. Consequently, the recursive join is processed in the fixed number of computation iterations,  $l$  rather than of  $2 \times l$ . This leads to a better performance with less I/O operation and the communication. The join implementation is then illustrated by an algorithm in form of pseudo code.

In addition, a complex cost model for recursive joins is designed to demonstrate the efficiency of our solution compared with others.

This optimization is an extremely important contribution to support scalable social network analysis, internet traffic analysis, DNA data analysis, and general recursive query. That also means the problem (c) is tackled by this work.

### (4) *Experimental evaluation*

Our last result consists of experiments of the different join algorithms using various parameters. 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, the total execution time, and especially task timelines.

Experimental evaluation helps us thoroughly evaluate the performance of the join algorithms.

## 1.3 Thesis outline

In Chapter 2, we define the basic join types that help us gain a better understanding of the basic characteristics and features of important equi-joins. It then covers specific equi-join operations that are used in this thesis such as two-way joins, multi-way joins and recursive joins. Besides, we also summarize background components that are the essentials of the MapReduce framework, parallelization of the join operation, and iteration in MapReduce supporting iterative data analysis applications. Basic concepts, terminologies, and characteristics of the Bloom filter are described. Notably, we present the classification and details of dominant join algorithms in MapReduce. We specifically analyse advantages and disadvantages of each method to point out their limitations related to our proposals.

In Chapter 3, we present our first two contributions including the intersection filter and optimizations for two-way joins and multi-way joins. We first provide a short survey of previous works, and show existing problems of the joins and especially Bloomjoin that need to be addressed. Then, we describe three approaches to building the intersection filter. The false intersection probability of the approaches is defined. We use the intersection filter to optimize two-way joins and important multi-way joins. In addition, we show cost models for two-way joins and multi-way joins, and make comparisons of the different join algorithms. In the end of the chapter, we present experiments on the performance of the joins and compare the joins using the intersection filter to previous methods mentioned in the literature.

In Chapter 4, we detail our last contribution that is an optimization for recursive joins. First, we examine prior solutions for evaluating the semi-naive algorithm. The problems of the solutions are shown in detail. Next, we describe basic concepts and design details of the difference filter. A false difference of the filter is identified by a probability. Then, we propose an optimization for the general semi-naive algorithm using the intersection and difference filters. Our semi-naive strategy computes the recursive join as an iteration of one join job instead of two MapReduce jobs. Finally, the optimization for recursive joins using the filters is proved more efficient than the existing solutions through a cost model-based comparison.

In Chapter 5, we present the conclusions of the thesis. Besides, we discuss open challenges and perspectives in optimization for join operations.

## **Part I**

### **Background and related works**



---

## BACKGROUND AND RELATED WORKS

MapReduce has emerged as a popular large-scale data processing model because of its attractive programming interface with 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, a join operation is much more difficult and expensive. Significant efforts have been made to develop efficient join algorithms in recent years. Therefore, we have conducted a survey of various join algorithms to support our research. This work investigates related works, strategies as well as advantages and disadvantages of the join algorithms in MapReduce.

The chapter is organized in the following way. The first section includes the definition of the basic join types that supply a better understanding of important equi-joins. It then moves to the join ways such as two-way joins, multi-way joins and recursive joins. In addition, the section summarizes the fundamentals of MapReduce and Apache Hadoop to start making sense of the join processing in the real world. It also describes parallelization of the join operation and iteration in MapReduce environment. Section 2.2 presents the classification and details of recent approaches to improving the join computation. The advantages and disadvantages of the approaches are also discussed. Next, we review some basic concepts, terminologies, and characteristics of the Bloom filter in Section 2.3. It is used as an optimization technique for joins in our approach. Besides, we present the Bloomjoin algorithm in detail. Finally, Section 2.4 concludes all elements help us make better optimizations for the joins in this research.

### 2.1 Background

#### 2.1.1 Join operation

The join operation [40][41] 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 relations according to a condition on different attribute schemes to form a new relation with columns selected from across the multiple relations.

*Equi-joins* are a common type of joins and are considered as a default type of joins. We therefore consider the equi-joins as our main research object for optimizing joins.

The equi-join is a join where the join condition uses an equality operator (=) to relate the tuples of two datasets. Two-way equi-joins, multi-way equi-joins, recursive equi-joins, etc., are instances of the equi-join. Most of the existing work has concentrated on the two-way joins and has left readers to extend the idea for the

## 2.1 Background

multi-way joins and the recursive joins. Our work will mention all of them. Before moving any further, we define the context of this research:

- **Two-way join:** Given two datasets  $R$  and  $L$ , a two-way join is defined as a combination of tuples  $r \in R$  and  $l \in L$ , such that  $r.x = l.y$ ; where  $x$  and  $y$  are columns in  $R$  and  $L$  respectively. This specification is represented as:

$$R \bowtie_{x=y} L$$

If the join columns in the datasets have the same name, we rewrite the two-way join in a short form of  $R \bowtie_x L$

- **Multi-way join** [42]: Given  $n$  datasets  $R_1, R_2, \dots, R_n$ , a multi-way join is defined as a composition of multiple two-way joins, noted as:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie \dots \bowtie R_n$$

We specifically consider multi-way joins using equi-joins, noted as:

$$R_1 \bowtie_{x2=x2} R_2 \bowtie_{x3=x3} R_3 \bowtie \dots \bowtie_{xn=xn} R_n$$

- **Recursive join** [43][44]: Given a dataset  $K(x, y)$ , a recursive join is defined as the transitive closure of the dataset  $K$ :

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

Two methods for computing the recursive join are the iterative and direct methods. Our research focuses on the recursive join evaluation based on the iterative method.

For convenience, we use the following example during the research.

**Example.** Given a *user* dataset  $R(uid, uname, location)$ , a *log* dataset  $L(uid, event, logtime)$  and an *acquaintance* dataset  $K(uid1, uid2)$ . We have the kinds of queries expressed by expressions in the relational algebra as follows.

**Q<sub>1</sub> - Two-way join:** Find the names and events of all users who have accessed before 07/07/2014

$$A_1(uname, event) = \prod_{uname, event} (R \bowtie_{uid=uid} \sigma_{logtime < 12/02/2014} (L))$$

This query uses a two-way join for combining two datasets  $R$  and  $L'$  selected from  $L$ . The two-way join is a basic type of joins. In this case, we consider the popular two-way equi-join that uses equality operator on a common column  $uid$ .

**Q<sub>2</sub> - Multi-way join:** Find the ids, events and times of all users who are known by 'Laurent dOrazio'

$$A_2(uid, event, logtime) = \prod_{uid, event, logtime} (\sigma_{uname='Laurent dOrazio'} (R) \bowtie_{uid=uid1} K \bowtie_{uid2=uid} L)$$

The query uses a multi-way join, precisely, a *chain join*. It links the datasets including  $R'$  selected from  $R$ ,  $K$  and  $L$ . The chain join is an important special case of multi-joins, in which datasets are strung together to produce the result.

**Q<sub>3</sub> - Recursive join:** List the ids of all users who may be friends of 'Philippe Rigaux'

$$A_3(uid1, uid2) = K \bowtie_{uid1=uid} \sigma_{uname='Philippe Rigaux'} (R)$$

$$A_3(A_3.uid1, K.uid2) = A_3 \bowtie_{uid2=uid1} K$$

The query  $Q_3$  is defined by a *recursive join* to deduce a new dataset called *friend*  $A_3$ . The new dataset includes the acquaintance dataset  $K'$  selected from  $K$  and all new tuples  $(a', c')$  so that there exist  $K(a', b')$  and  $K(b', c')$ . Actually, the query uses a compound operation that involves in repeating the join operation.

Although this research only addresses problems of the equi-join, we look at the context of general joins to be able to distinguish the equi-join from various types of joins. The join types can be grouped in two primary classifications including *inner joins* and *outer joins* [45].

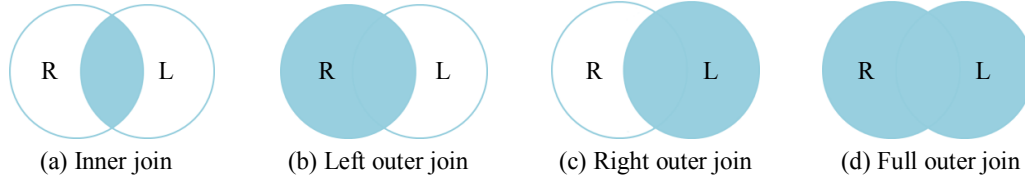


Figure 2.1: Types of joins

- **Inner join:** a join chooses only tuples that match a join condition in both joined relations (as in Figure 2.1(a)). It uses a comparison operator to match tuples from two relations based on some columns from each relation. This class is a typical join operation, which includes types as equi-joins, non-equi-joins, natural joins, and cross joins.
  - *Equi-join:* an equi-join uses an equivalence operation or an equality operator ( $=$ ) to match tuples from different datasets.
  - *Non-equi-join:* a non-equi-join uses a non-equality comparison operator, e.g.,  $\neq$ ,  $\leq$ ,  $\geq$ ,  $>$ ,  $<$  or BETWEEN, etc.
  - *Natural join:* a natural join offers a further specialization of equi-joins. The join compares all columns in both datasets that have the same column-name in the joined datasets. The resulting joined dataset contains only one column for each pair of equally-named columns.
  - *Cross join:* it produces the Cartesian product of all the tuples in both datasets. This type of join occurs when we do not specify a condition.
- **Outer join:** a join returns all tuples from at least one of datasets. As shown in Figure 2.1(b-d), there are three types of outer join:
  - *Left outer join:* it returns all the tuples that would be returned by an inner join, plus all the tuples from the left (or first-listed) data set that do not match any tuple from the right data set.
  - *Right outer join:* it returns all the inner-join tuples, plus all the tuples from the right dataset that do not match any tuple from the left dataset.
  - *Full outer join:* it retains all tuples from datasets, regardless of matches.

From the above classification, our main research subject, which is the equi-join, belongs to an inner join.



### 2.1.2 MapReduce framework

MapReduce [1] is a parallel and distributed programming model for large-scale data analysis run on computer clusters that can scale to thousands of nodes in a fault-tolerant manner. The use of MapReduce has become widespread since Google first introduced it in 2004. It allows users to concentrate only on designing their data-processing operations regardless of the inherent parallel or distributed nature of the cluster itself.

In this model, a MapReduce job consists of two distinct phases, namely, *map phase* and *reduce phase*. Each the phase executes a user-defined function as a distributed execution of parallelizable computations, which acts upon a pair of key and associated value(s). Figure 2.2 below describes MapReduce execution.

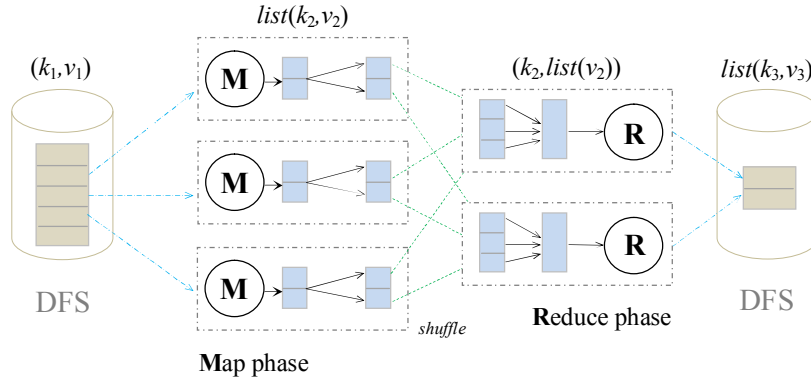


Figure 2.2: MapReduce Execution

The user-defined map function (**M**) takes an input pair  $(k_1, v_1)$  from a Distributed File System (DFS) and transforms into a list of intermediate key/value pairs  $list(k_2, v_2)$ . The intermediate values associated with the same key  $k_2$  are grouped together and then passed to nodes that perform the reduce function.

$$map(k_1, v_1) \rightarrow list(k_2, v_2)$$

The reduce function (**R**) is called for each intermediate key  $k_2$  and a list of values for that key to generate a new list of values.

$$reduce(k_2, list(v_2)) \rightarrow list(k_3, v_3)$$

As illustrated in Figure 2.2, a typical MapReduce job is executed within the two phases across multiple nodes. The map phase and the reduce phase include map tasks and reduce tasks, respectively. These tasks run simultaneously on the nodes.

In the map phase, each map task reads a split of one input dataset, calls the map function for each key/value pair to produce intermediate key/value pair(s). The map task sorts the intermediate data and then calls a partition function on each key to calculate its reducer node index. It means that the partition function operates on the intermediate key/value pairs  $(k_2, v_2)$ , and returns the partition index. The number of partitions is equal to the number of reducers.

The reduce phase has three steps, shuffle, sort and reduce. Shuffle is where the intermediate data is collected by the reduce task from each map task. This can happen while map tasks are generating data since it is only a data transfer. On the other hand, sort and reduce can only start once all the map tasks are done. 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.

Intermediate pairs in the reduce phase are processed in increasing key order. This ordering guarantee makes it convenient to generate a sorted data file on demand and useful to support the reduce function that requires the order of keys.

All of the reduce tasks can launch at the same time. Of course, the execution time of one reduce task  $r_1$  can be different from others and  $r_1$  can run for a long period of time. In other words, some reduce tasks finish faster, but other reduce tasks may be executed longer.

An optional combiner function pre-aggregates the map output in order to reduce the amount of data to be transferred across the network. It runs after the map function and before the reduce function. This means that the combiner function is executed on the same node as the map node, receives data emitted by the map function on a given node and emits output to the reduce nodes. Many data processing jobs use this function such as search engine, machine learning and deduplication.

After running the map function, if there are many identical key/value pairs, the MapReduce framework has to send all those pairs to the reduce function. This can incur a considerable overhead. To remove this redundant data, we can use the combiner function. The example can be illustrated as follows.

Map:	{(a, b), (a, b), (a, b), (a, d), (c, d)}
Combining:	{(a, b), (a, d), (c, d)}
$\downarrow$ <i>shuffle</i>	
Reduce:	{(a, [b, d]), (c, d)}

We can see that the duplicate tuples (a, b) are eliminated by the combiner rather than transferred to the reducer.

Apache Hadoop [46] is an open source MapReduce framework written in Java for executing applications on large clusters. While Google's MapReduce framework is not available to the public, several other implementations of MapReduce such as DISCO [47] and Sphere [48] are also available but not as popular as Hadoop. Hadoop includes a data storage component called Hadoop Distributed File System (HDFS) and a data processing component called Hadoop MapReduce Framework. These components correspond to the Google File System (GFS) and the general MapReduce computing paradigm. Hadoop's HDFS is a fault-tolerant distributed file system. 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 [46][49][50]. 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) [49]. It is therefore suitable for applications handling large

## 2.1 Background

datasets. Hadoop's 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 Hadoop to run MapReduce programs that examine tackling large data joins using parallel processing.

### 2.1.3 Parallelization of a join operation in MapReduce

Before presenting this section in detail, we should look at the difference between the MapReduce model and the existing parallel programming models like MPI, OpenMP, CUDA, etc. MPI means Message Passing Interface, which is one of the most portable high-performance computing programming models. It was designed to enable parallel programming by communication on distributed-memory or shared memory systems in which messages are data packets exchanged between processes. MPI allows programs on one node to send data to another, or conversely receive. In other words, tasks can use their own local memory during computation and exchange data through communications by sending and receiving messages.

Figure 2.3 depicts how a typical traditionally parallel application and MapReduce application work.

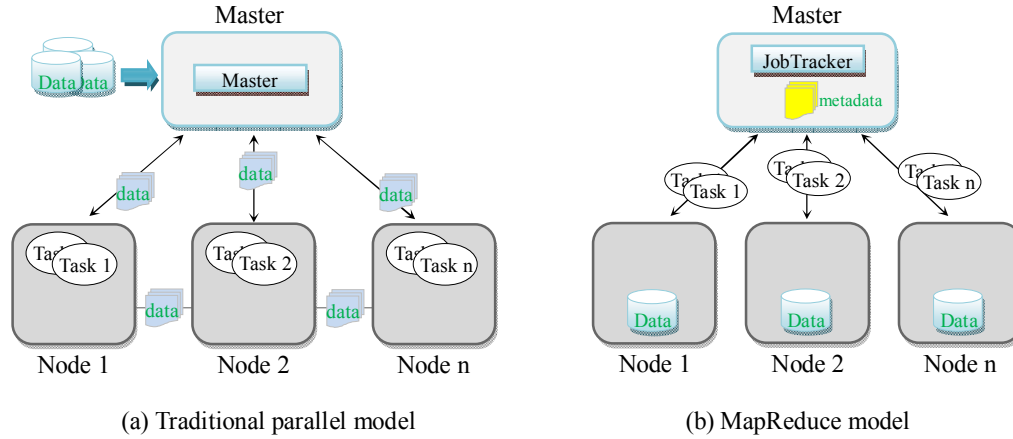


Figure 2.3: Difference between traditional parallelism and MapReduce

Figure 2.3(a) illustrates a traditional parallel model. The input data is resided some remote storage devices such as a file server serving files over NFS or a general parallel file system (GPFS), etc. The compute nodes or workers are represented by rounded rectangles. The tasks depicted eclipses can be MPI tasks, or threads on a shared-memory system. The parallel application is executed as follows. A master parallel worker (MPI rank, thread, etc.) reads the input data. The master worker then splits the input data into chunks and sends them to each of the other workers. The parallel workers compute on their chunk of the input data. The parallel workers communicate their results with each other, and then continue the next computation. For this model, the data is separated from the compute resources, e.g., the chunks of computation are unavailable on compute nodes for the tasks. Besides, the master worker can get around the bottleneck of reading performed serially, splitting and distributing the large-scale input data to the compute nodes. These lead to limitations for scalability.

In contrast, MapReduce model operates in a completely different way. It conveys the computation close to where the data is, instead of moving the data to where the computation is executed as the traditional model. It is shown in Figure 2.3(b). The model does not have to move any data because splits of the data already reside on the compute nodes.

As an illustration of using MapReduce, we consider the join operation of two large datasets  $R$  (*user* dataset) and  $L$  (*log* dataset) to list the user names and corresponding events that the users accessed to the system as the following query.

$$R(uname, uid) \bowtie_{uid=uid} L(uid, event)$$

In large-scale data applications like social networks, 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.

We show how to bring parallelism with MapReduce into the join execution as depicted in Figure 2.4. The first step in building this parallel execution is specifying sets of tasks that can run concurrently and partitions of data that can be concurrently processed. Hence, there are two opportunities for parallelism in the join operation, namely, parallel input processing and parallel join processing. They correspond to parallel map tasks and parallel reduce tasks.

The map tasks are responsible for processing splits of the inputs  $R$  and  $L$  simultaneously. Because the splits of the two input datasets already reside on the compute nodes, the parallel map tasks can handle its split independently and instantly without moving any data. Even if the split is not available for the map task, the task is closed to where the split is residing because of data locality in MapReduce. As in Figure 2.4, the jobtracker creates three mappers to be able to process three splits of the inputs concurrently. The first mapper computes on the first split consisting of three tuples of  $R$ . The second split with one tuple of  $R$  and the third split with two tuples of  $L$  are processed by the second and third mapper, respectively. The mappers transform the tuples of  $R$  and  $L$  as follows.

$$\begin{aligned} r \text{ is a tuple of } R: & \quad r(uname, uid) \rightarrow r'(uid, uname) \\ l \text{ is a tuple of } L: & \quad l(uid, event) \rightarrow l'(uid, event) \end{aligned}$$

It means that the join key  $uid$  is treated as the intermediate key for all the tuples.

As illustrated in Figure 2.4, we can show the parallel processing of the mappers in detail.

The mapper 1 for the first split of  $R$ :

$$\{(A, B), (C, B), (A, F)\} \rightarrow \{(B, A), (B, C), (F, A)\}$$

The mapper 2 for the second split of  $R$ :

$$\{(C, D)\} \rightarrow \{(D, C)\}$$

The mapper 3 for the split of  $L$ :

$$\{(B, C), (D, F)\} \rightarrow \{(B, C), (D, F)\}$$

The intermediate tuples  $r'$  and  $l'$  are then shuffled to the reduce tasks.

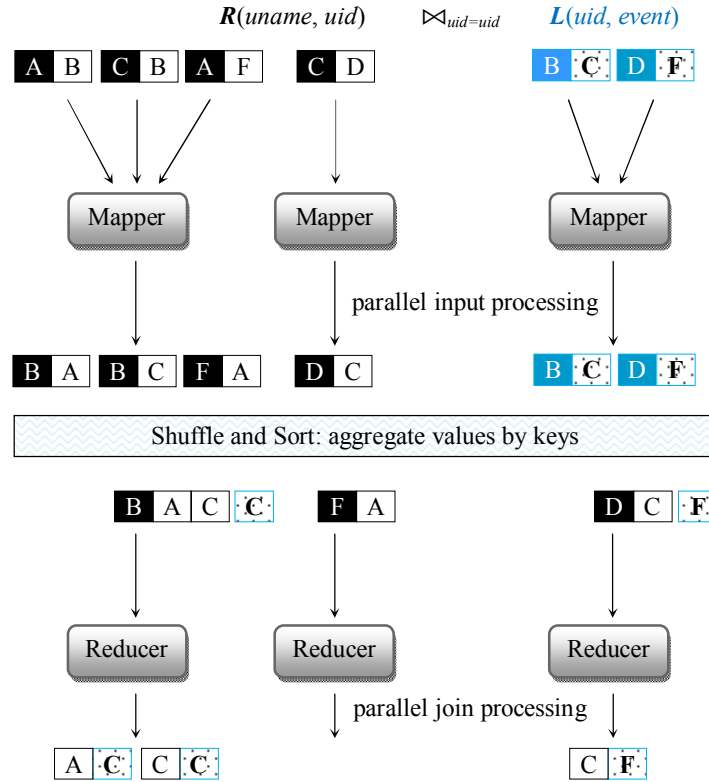


Figure 2.4: Parallel implementation of the join operation in MapReduce

The reduce tasks perform joining for each key  $uid$  concurrently. The intermediate tuples associated with the same key are passed to the same reducer. The reduce function is called for each unique key with a list of values. The function simply takes each tuple of  $R$  and finds tuples of  $L$  with the same join key to generate the results. Figure 2.4 shows that the join key  $B$  has two tuples  $A$  and  $C$  of  $R$  joined with one tuple  $C$  of  $L$  to produce two result tuples  $(A, C)$  and  $(C, C)$ . We can see the parallel join processing of the reducers in detail.

The reducer 1 for the key join  $B$ :  $(B, [R:A, R:C, L:C]) \rightarrow \{(A, C), (C, C)\}$   
 The reducer 2 for the key join  $F$ :  $(F, [R:A]) \rightarrow \{\text{empty}\}$   
 The reducer 3 for the key join  $D$ :  $(D, [R:C, L:F]) \rightarrow \{(C, F)\}$

The above execution shows that the parallel processing with MapReduce for the join operation would be necessary to achieve high scalability and fault tolerance in massive data joining.

#### 2.1.4 Iteration in MapReduce

The fact is that the standard MapReduce framework lacks built-in support to perform these iterative data processing applications. Instead, users must execute iterative programs by manually linking. The general idea for these iterative algorithms in MapReduce is to chain multiple jobs together, using the output of the last round as the input of the next round. The program termination condition must be calculated

within a MapReduce driver program. Hadoop has added a feature, called *Counters*, to execute this task. To check for a termination condition with Hadoop counters, we run a job and collect statistics while it is running. Then, we obtain values of the counters to compute the stop condition. Finally, we decide whether the loop is to stop or to continue.

HaLoop [13] is a modified version of the Hadoop framework, that is designed to efficiently support such iterative MapReduce applications. HaLoop still uses the Hadoop distributed file system for storing input and output data of MapReduce jobs. The basic Hadoop framework is modified to accommodate the requirements of the iterative applications. Task scheduler and task tracker modules are modified, and the loop control, caching, and indexing modules are new. The task tracker not only manages task execution, but also manages caches and indices on the slave node. Besides, HaLoop provides a new application programming interface to simplify iterative MapReduce programs as follows.

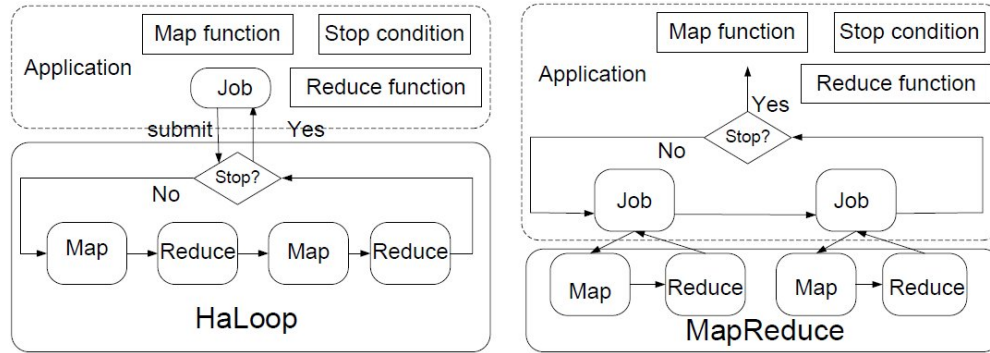


Figure 2.5. HaLoop vs. Hadoop Programming Model [13]

Figure 2.5 shows the difference between HaLoop and Hadoop for supporting iterative applications. HaLoop knows and controls the loop of map and reduce tasks, while Hadoop only knows jobs with one map-reduce pair. Specifically, HaLoop receives loop settings from a driver program and controls the loop execution, whereas a driver program in Hadoop must control the loops.

HaLoop provides three types of caches, namely, Reducer Input Cache (RIC), Reducer Output Cache (ROC), and Mapper Input Cache (MIC).

- Reducer Input Cache: stores and indexes reducer inputs across all reducers. This cache type is used to avoid reprocessing the same data with the same mapper on iterations (e.g., a loop-invariant input relation  $K_0$ ).
- Reducer Output Cache: caches the most recent local output on each reducer node and create a local index for the cached data. It is used in applications where fixpoint evaluation should be conducted after each iteration.
- Mapper Input Cache: caches and indexes mapper inputs across all mappers. It is used to avoid non-local data scans in mappers during non-initial iterations.

Besides, HaLoop introduces the risk of *recursive recovery*, where a failure in one step of one iteration may require re-execution of tasks in all preceding steps in the same iteration or all preceding iterations. If there is a reduce task failure, the failure recovery is similar to reducer task recovery in Hadoop and the cache must be

reconstructed from the mapper output of iteration 0. In another case, if an entire node fails, the mapper output of iteration 0 may be lost. Thus, the corresponding map tasks are re-executed as needed.

## 2.2 Basic join algorithms in MapReduce

Most join algorithms in MapReduce are derived from the literature on join processing in parallel RDBMSs [51][8][52][53][54] and distributed RDBMSs [55][54][56] 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* [3]. In this section, therefore, we describe their implementation details and discuss the difference between these two important join algorithms. Then, some variants and improvements of the join algorithms like broadcast join and semi-join are presented. The problem of skewed data processing in the join operation is outside of our research scope.

### 2.2.1 Map-side join

*Map-side join* [2][3], which is similar to sort-merge join [51][52][57] in RDBMSs, works by joining two datasets on the map side without a shuffle and reduce phase. This algorithm however requires under certain 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 on an example as Figure 2.6.

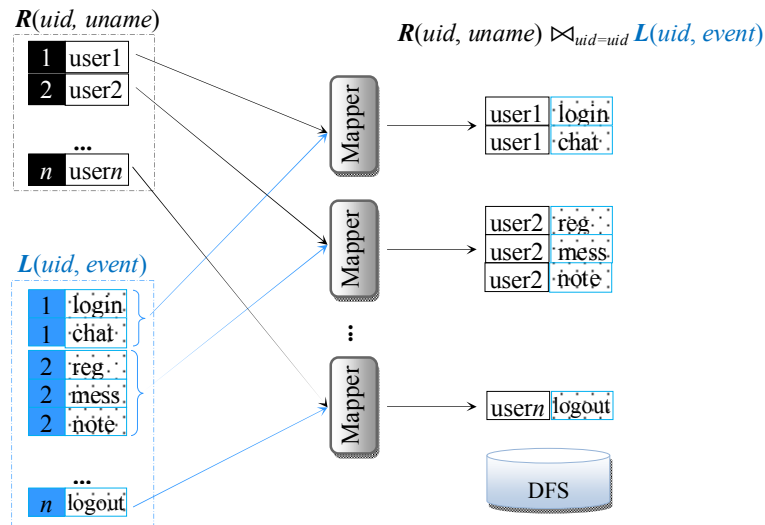


Figure 2.6: Map-side join in MapReduce

The illustration shows the join of two datasets *user* and *log*,  $R(uid, uname) \bowtie_{uid=uid} L(uid, event)$ . The two datasets have the same  $n$  partitions with  $n$  join keys sorted. Thus, the join job uses  $n$  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.

$$\{(1, 'user1')\} \times \{(1, 'login'), (1, 'chat')\} \rightarrow \{('user1', 'login'), ('user1', 'chat')\}$$

The mapper then sends the output to the map function for storing into DFS. This is similar to the other partitions with the key joins 2, ...,  $n$ .

The pseudo code of the Map-side join is shown in Listing 2.1.

---

**Algorithm 1** - Map-side join algorithm

---

**Job1:** partition dataset  $R$  into  $n$  sorted partitions as requirement

**Job2:** partition dataset  $L$  into  $n$  sorted partitions as requirement

**Job3:** join two input datasets  $R$  and  $L$

```

Init_Map() // init function for map phase
    buff_R  $\leftarrow$  load(partitioni_R); //loading partitioni of  $R$ 
    buff_L  $\leftarrow$  load(partitioni_L); //loading partitioni of  $L$ 
    result  $\leftarrow$  empty; //for storing join computation
    if (buff_R  $\neq$  null & buff_L  $\neq$  null) then
        for each  $r$  in buff_R do
            for each  $l$  in buff_L do
                result.add(pair(  $r$ ,  $l$  ));
            end if
    end if

Map( $k$ : null,  $v$ : null)
    for each  $t$  in result do
        emit(null,  $t$ );

```

---

Listing 2.1: Pseudo code for Map-side join algorithm

The Map initialization function, `Init_Map()`, defines an action to run before the mapper processes any input. It loads the partition  $i$  of  $R$  and the partition  $i$  of  $L$  into two memory buffers *buff\_R* and *buff\_L*, respectively, to then perform joining.



We consider both advantages and disadvantages of using the Map-side join algorithm

- Advantages:
  - 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.
- Disadvantages:
  - The drawback of the algorithm is the rigorous requirements on the input datasets. They must be divided into the same number of partitions, be strictly sorted by the join key, and have the same set of the join keys. 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.
  - Another limitation of the algorithm is the buffering of both the two joined partitions that can lead to a memory overflow for the compute node. The two joined partitions consist of all the tuples with the same join key from all the input datasets. As a result, the Map-side join can quickly run out of memory 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.

### 2.2.2 Reduce-side join

*Reduce-side join* [10][2] is also known as *repartition join* [3]. As implied by its name, the actual join computation is only conducted on the reduce side. The algorithm is based on the nature of the MapReduce framework. It partitions all tuples of input datasets according to the join key into intermediate key/value pairs. Then it shuffles (repartition) the immediate pairs 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.

When all the mappers are complete, the reducer calls the reduce function for each the join key. The reduce function buffers only the pairs of one input dataset. It then performs joining the buffered pairs with each pair of another input dataset reaching. The output of the reduce function can be directly written to DFS. The Reduce-side join algorithm is depicted as Figure 2.7.

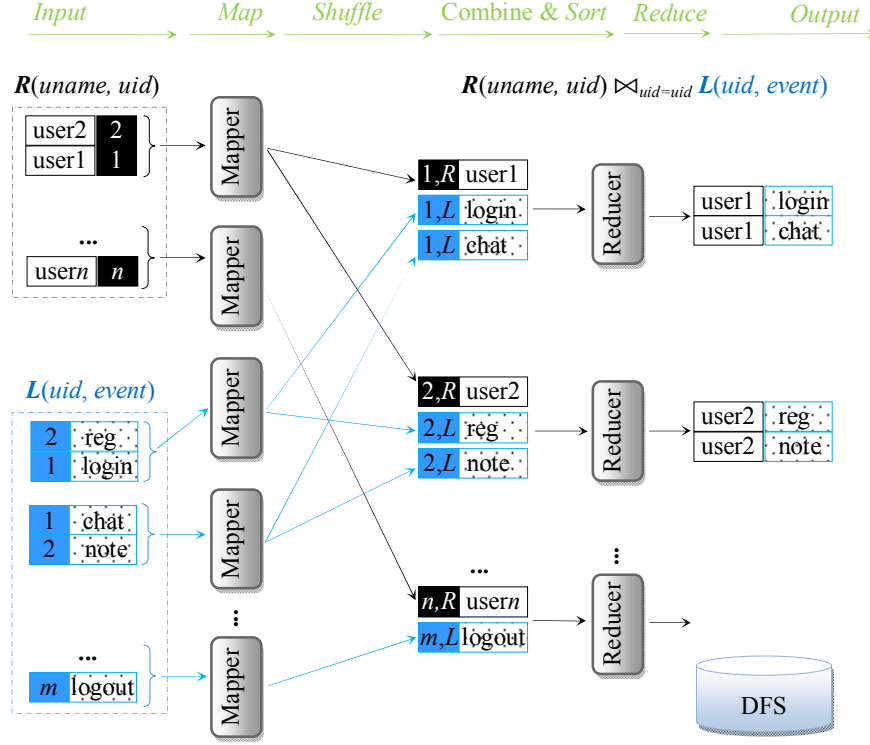


Figure 2.7: Reduce-side join in MapReduce

The illustration shows the join of two datasets *user* and *log*,  $R(\text{uname}, \text{uid}) \bowtie_{\text{uid}=\text{uid}} L(\text{uid}, \text{event})$  similar to the example of the Map-side join. The mappers scan all fixed-size splits (e.g. two tuples) of  $R$  and  $L$ , extract the join key *uid* from the tuples, and tick each tuple with a tag that indicates one input dataset containing this tuple. The tuples in black are ticked with a tag 'R' and the tuples in blue are ticked with a tag 'L'. Then the mappers emit tagged tuples with composite keys of the form  $(\text{uid}, \text{tag})$ .

We have to override the default partitioning function, namely a *partitioner()* function. This function ensures that partitioning the tagged tuples takes into consideration only the join key part (*uid*) and ignores the tag part (*tag*). As a result, the reducers receive the tagged tuples of the form  $((\text{uid}, \text{tag}), \text{tuple})$  with the same *uid*. For instance, the first reducer receives the three tagged-tuples with the same join key 1:  $\{((1, 'R'), \text{'user1'}), ((1, 'L'), \text{'login'}), ((1, 'L'), \text{'chat'})\}$ .

The tagged tuples are then sorted by the composite key or the tag part to reach to the reduce function. The input to the reduce function is one list of the tuples in tag order. The reduce function forms the cross product of the tuples of  $R$  that are buffered and each tuple of  $S$  coming to complete the join and outputs new key/value pair(s) with the *uname* as the key and *event* as the value.

The output of the first reducer is a set of tuples  $\{(\text{'user1'}, \text{'login'}), (\text{'user1'}, \text{'chat'})\}$ . The second reducer produces a set of tuples  $\{(\text{'user2'}, \text{'reg'}), (\text{'user2'}, \text{'note'})\}$  and the last reducer generates the empty output because it does not find any tuples with the same join key.

The pseudo code of the Reduce-side join is shown in Listing 2.2.

---

**Algorithm 2** - Reduce-side join algorithm

---

```

Map(k: null, v: a tuple from an R or L split)
    tag ← a bit 0 or 1 corresponding to name of R or L;
    key ← extract the join key from v;
    emit(pair(key, tag), v);

Partitioner(k': taggedkey, v: value, p: the number of reducers)
    return hash_func(k'.key) mod p;

Init_Reduce() // init function for the reduce phase
    currentKey ← '0'; //for storing current key
    buff ← empty; //for storing tuples with same key of R

Reduce(k': taggedkey, v': list of values v with key k')
    if k'.key != currentKey then
        clear(buff);
        currentkey = k'.key;
    endif
    if k'.tag == '0' then
        for each l in v' do
            add tuple l to buff;

        else if k'.tag == '1' then
            for each l in v' do
                for each r in buff do
                    emit(null, pair(r, l));
            end if
    end if

```

---

Listing 2.2: Pseudo code for Reduce-side join algorithm

A reduce initialization function, `Init_Reduce()`, defines an action to run before the reducer processes any input. It creates a memory object and allocates a memory buffer for storing tuples with the same key of *R*.

The Reduce-side join algorithm has the following advantages and disadvantages:

- Advantages:
  - This algorithm uses the natural and flexible way of the MapReduce framework to process a join operation as a standard MapReduce job. It is the most general type of join algorithms without any constraints on input datasets. Besides, it can address the problem of the memory overflow better than the Map-side join because it buffers only tuples with the same key of one input dataset instead of two input datasets as the Map-side join done.

- Disadvantages:
  - Two major obstacles of this algorithm are the high I/O and communication costs for intermediate data that is generated during the map phase and transferred from map tasks to reduce tasks. It means that the entire input datasets are sent across the network from the map nodes to the reduce nodes.
  - Like the Map-side join, the Reduce-side join can still run out of memory when its input datasets are skewed.

### 2.2.3 Broadcast join

*Broadcast join* [3] in MapReduce is similar to a hash join in RDBMSs. The first hash join algorithm has been mentioned in [53] and then in [51][8], etc. It is a special variant of the Map-side join algorithm. However, it does not require the strict restrictions on input datasets such as the same sorted-partitions and the same set of join keys.

For this join, a small input dataset is sent or broadcasted to all the compute nodes. The mapper loads the small dataset into memory and calls the map function for each tuple  $t$  from a larger input dataset. The map function probes the in-memory dataset and finds matches with the tuple  $t$  to perform the join computation. It then writes the joined tuples to DFS. Figure 2.8 shows an illustration of the broadcast join algorithm.

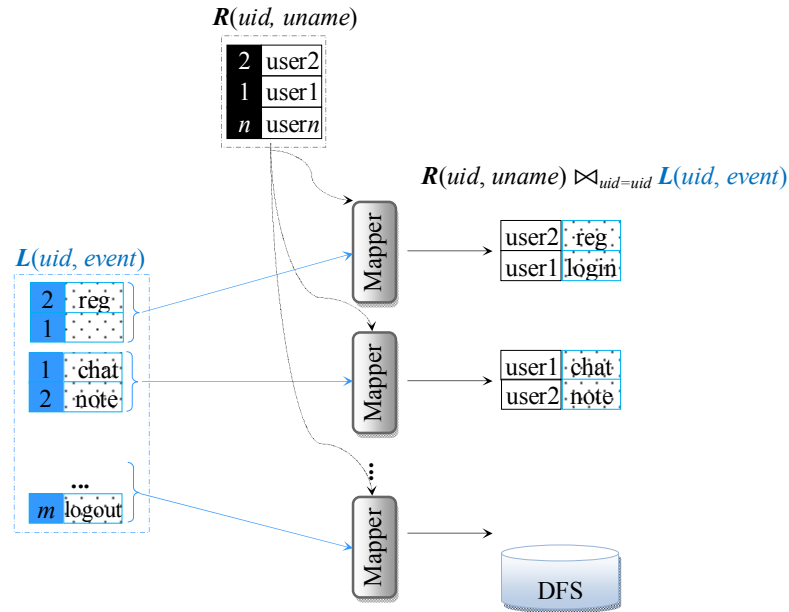


Figure 2.8: Broadcast join in MapReduce

Assume that we compute a join of two datasets  $user\ R(uname, uid)$  and  $log\ L(uid, event)$ , where  $R$  is a small dataset consisting of three tuples, and  $L$  is stored in many splits, each containing two tuples. We can distribute the small dataset  $R$  to all the mappers and each mapper can load it into memory by an initialize function. Next,

## 2.2 Basic join algorithms in MapReduce

---

the mappers read splits of the dataset  $L$ . The first mapper reads the first split with two tuples  $\{(2, 'reg'), (1, 'login')\}$ . The mapper calls the map function for each tuple  $t$  of  $L$ . The map function goes through all tuples of the  $R$  in memory and joins them with the tuple  $t$  if they have the same join key. We can look at the first mapper:

$$\begin{aligned} \{(2, 'user2'), (1, 'user1'), (n, 'usern')\} \times \{(2, 'reg')\} &= \{('user2', 'reg')\} \\ \{(2, 'user2'), (1, 'user1'), (n, 'usern')\} \times \{(1, 'login')\} &= \{('user1', 'login')\} \end{aligned}$$

The mapper emits each joined tuple to DFS. This work is similar to the other mappers.

There are some advantages and disadvantages to this broadcast join algorithm

- Advantages:
  - The algorithm takes the advantages of the Map-side join algorithm, which only includes the map phase without intermediate data and transmission of the datasets over the network. It is even more efficient than the general Map-side and Reduce-side join algorithms because it does not need the pre-processing step for sorting the input datasets, buffers only one dataset, and computes join at the map phase.
  - Furthermore, it is important to note that the algorithm is not affected by the problem of data skew because each mapper reads one split of the larger input dataset in which the split fits in memory.
- Disadvantages:
  - The broadcast join can be used for two arbitrary input datasets but one of them should be quite small to be distributed to all the mappers and fit in memory.

The pseudo code of the broadcast join is shown in Listing 2.3.

---

**Algorithm 3** - Broadcast join algorithm

---

```
Init_Map() // init function for map phase
     $buff\_R \leftarrow load(R)$ ; //loading all tuples of  $R$ 

Map( $k$ : null,  $v$ : a tuple from an  $L$  split)
     $refKey \leftarrow$  extract the join column from  $v$ 
    for each  $r$  in  $buff\_R$  do
         $joinKey \leftarrow$  extract the join column from  $r$ 
        if ( $joinKey == refKey$ ) then
            emit(null,  $pair(r, v)$ );
        end if
```

---

Listing 2.3: Pseudo code for Broadcast join algorithm

### 2.2.4 Semi-join

*Semi-join* in MapReduce [3][9] is derived from the semi-join strategy that is a popular technique for query processing in parallel and distributed database systems [52][53][55][54][58][59]. This approach avoids sending non-joining tuples on the network by eliminating tuples from one input dataset where their join keys are not matching any value in another input dataset.

We consider the join of two datasets *user*  $R(uname, uid)$  and *log*  $L(uid, event)$  as the previous examples. The detail of the semi-join algorithm is described in three computing stages:

- *Stage 1*: projects all tuples of the input dataset  $L$  on the join key column  $uid$ , and stores these distinct keys into a file  $L.uid$ , assuming it is small enough to fit in memory.
- *Stage 2*: distributes the file  $L.uid$  to all the compute nodes, and reduces each split  $R_i$  to  $R'_i$  by eliminating tuples whose join keys are not matching any of  $L.uid$
- *Stage 3*: broadcasts all the files  $R'_i$  ( $R'$ ) to all the compute nodes, and computes the cross product of each split  $L_i$  and  $R'$ .

To implement the algorithm, we use three MapReduce jobs corresponding to the three stages as shown in Figure 2.9. These jobs are executed in the following sequence.

- *Job 1*: determines a set of unique join keys of  $L$  ( $L.uid$ ) by projecting tuples of  $L$  on the join key column ( $uid$ ). This job is a typical MapReduce job with only one reducer. The mappers scan splits of  $L$ , extract the join key column for each tuple, and pass these join keys to the reducer if they are not in a table storing the previous distinct join keys. The reducer receives all the join keys from all the mappers in which duplicate keys are eliminated, and saves the distinct join keys to a file  $L.uid$  on DFS.
- *Job 2*: determines a filtered version of  $R$  (denoted  $R'$ ) by removing keys not in  $L.uid$ . It is a map-only job without the reduce phase. The file  $L.uid$  is distributed to all the compute nodes using a *distributed cache* and loaded into a hash table by an initialization function for the map phase. The mappers scan splits of  $R$ , extract the join key column for each tuple of  $R$  to check in the hash table of  $L.uid$ , and then emit the tuples if their join keys are in the hash table. Each mapper generates the output file  $R'_i$  and the dataset  $R'$  includes all the output files  $R'_i$ .
- *Job 3*: computes the join of  $L$  and  $R'$ . This job is also a map-only job and similar to the broadcast join. First, we can use a distributed cache to broadcast the filtered version  $R'$  (all  $R'_i$ ) to all the compute nodes. The mappers then load  $R'$  into a hash table to perform joining with their split  $L_i$ . The results of the join are written to DFS.

## 2.2 Basic join algorithms in MapReduce

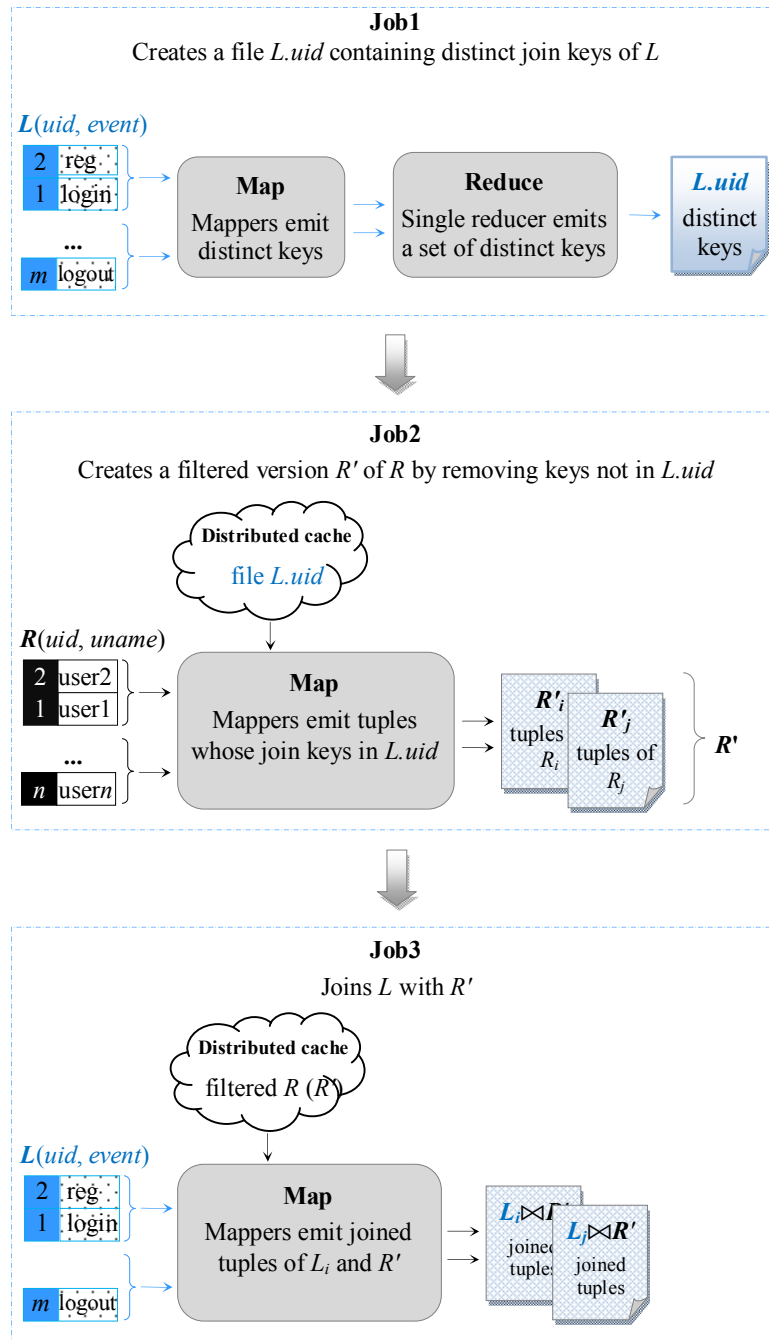


Figure 2.9: Semi-join in MapReduce

Listing 2.4 presents the pseudo code of the semi-join algorithm.

---

**Algorithm 4** - Semi-join algorithm

---

**Job 1:** Determine  $L.uid$  which is a set of unique join keys of  $L$

**Init\_Map()** // init function for map phase

$keyTable \leftarrow \text{empty}$ ; //storing distinct keys

**Map**( $k$ : null,  $v$ : a tuple from an  $L$  split)

$joinKey \leftarrow \text{extract the join column from } v$

**if** ( $joinKey$  not in  $keyTable$ ) **then**

    add  $joinKey$  to  $keyTable$ ;

**emit**( $joinKey$ , null);

**end if**

**Reduce**( $k'$ : a unique join key,  $v'$ : a list of null)

**emit**(null,  $k'$ );

**Job 2:** Determine  $R'$  which is a filtered version of  $R$  using  $L.uid$

**Init\_Map()** // init function for map phase

$refKeyTable \leftarrow \text{load}(L.uid)$ ; //loading  $L.uid$  to hashtable

**Map**( $k$ : null,  $v$ : a tuple from an  $R$  split)

$joinKey \leftarrow \text{extract the join column from } v$

**if** ( $joinKey$  in  $refKeyTable$ ) **then**

**emit**(null,  $v$ );

**end if**

**Job 3:** Compute the broadcast join of  $R'$  and  $L$  for the final result

**Init\_Map()** // init function for map phase

$buff\_R' \leftarrow \text{load}(R')$ ; //loading all  $R_i$

**Map**( $k$ : null,  $v$ : a tuple from an  $L$  split)

$refKey \leftarrow \text{extract the join column from } v$

**for** each  $r$  in  $buff\_R'$  **do**

$joinKey \leftarrow \text{extract the join column from } r$

**if** ( $joinKey = refKey$ ) **then**

**emit**(null,  $pair(r, v)$ );

**end if**

---

Listing 2.4: Pseudo code for Semi-join algorithm



---

There are advantages and disadvantages to this semi-join algorithm

- Advantages:
  - The semi-join is a type of the Reduce-side join, which is used for joining two arbitrary input datasets. For the algorithm, its input datasets are filtered to remove the non-joining tuples in the map phase before they are sent over the network, and thus this reduces the amount of intermediate data and communication costs.
  - In practice, this approach is used when many tuples of one input dataset may not be actually joined with any tuples of another input dataset. A typical example is the Facebook social network application with the user dataset  $R$  containing more than 1.23 billion users [60]. We would like to know information about user's activities in a certain period of time (e.g. an hour). Meantime, the log dataset  $L$  contains the activities of only a few million unique users and most of the users are not present in this log at all. To get the information, we need to join the two datasets  $R$  and  $L$ . In this context, the broadcast join is not appropriate because a large amount of the non-joining tuples in  $R$  are still broadcasted across the network (via the distributed cache) and loading entire  $R$  into the hash table can result in running out of memory. Therefore, the semi-join algorithm becomes a more suitable choice to avoid these problems.
- Disadvantages:
  - The algorithm uses the three MapReduce jobs paying extra costs as job initialization cost, intermediate data, local and remote I/O operations, and communication cost.
  - It must scan the input datasets multiple times. Moreover, in the third job, broadcasting the filtered version of  $R$  (all  $R'_i$ ) to all the compute nodes is very inefficient, even it can cause a memory overflow on all the compute nodes if  $R'$  is enough large and this is not rare.
  - A per-split semi-join algorithm [3] is designed to improve the semi-join algorithm by only sending  $R'_i$  to the mapper which holds  $L_i$  instead of sending the whole filtered version  $R'$ . Although this may reduce the network overhead for the third job, the algorithm also has the three jobs, depends on the size of  $R'_i$ , and is sensitive to skewed data.

## 2.3 Bloomjoin algorithm in MapReduce

### 2.3.1 Bloom filter

#### 2.3.1.1 Bloom filter basics

Bloom filter ( $BF$ ) [15] was introduced already in 1970 by Burton Bloom. It is a space-efficient randomized data structure used for testing membership in a set with a small rate of false positives.

Building the Bloom filter is based on the following definition of a *key/value map*:

**Definition 2.1.** A *key/value map* is a function  $H: U \rightarrow V \cup \{\perp\}$  where  $U$  is the input space,  $V$  is the value set and  $\perp$  is the null value.  $|U| = u$ ,  $|V| = v$  and the support  $S = \{x \in U \mid H(x) \in V\}$  has size  $n$ . A *key* is an element  $x \in U$ .

From the general *key/value map* model, if we set  $V = \{\text{true}\}$  and let  $\perp$  signify *false*, we obtain a set membership tester that identifies whether  $x \in U$  is a member of a set  $S$ .

The Bloom filter  $BF(S)$  for representation of a set  $S$  is described as follows.

- The set  $S = \{x_1, x_2, \dots, x_n\}$  of  $n$  elements is represented by an array of  $m$  bits, initially all set to 0.
- The filter uses  $k$  independent hash functions  $h_1, h_2, \dots, h_k$  with  $h_i: x \rightarrow \{1..m\}$ .
- To **insert** an element  $x \in S$ , we compute  $h_1(x), h_2(x), \dots, h_k(x)$ , and set the corresponding positions in the bit array to 1. Once we have done this operation for each element of  $S$ , we should have a bit array that acts 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 of these positions is set to 0, it is clear that  $y \notin S$ . Otherwise, all the positions are set to 1, we know that  $y$  may be a member of  $S$  with some probability.

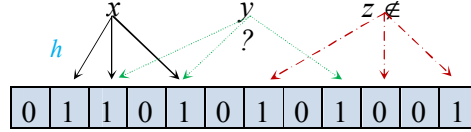


Figure 2.10: A Bloom filter  $BF(S)$  with 3 hash functions

As shown in Figure 2.10, an element  $x \in S$  is inserted into  $BF(S)$ . We check an element  $y$  and  $y$  may be in the set  $S$  since all the hash positions are set to one. An element  $z$  is not in the set  $S$ , because it hashes to one bit-array position containing 0. For this  $BF(S)$ , the size of the filter is  $m=12$  and the number of hash functions is  $k=3$ .

### 2.3.1.2 False positive probability and hash functions

It is possible that multiple elements from the set  $S$  will map to the same position in the bit array and even  $y$  is not in  $S$  but all  $h_i(y)$  are set to 1. As a result, the Bloom filter can have *false positives* and it represents a superset of the set  $S$ . The actual elements in  $S$  and the false positive elements of  $BF(S)$  are represented in Figure 2.11.

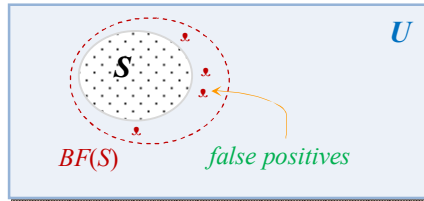


Figure 2.11: Approximate representation of  $S$  with false positives

## 2.3 Bloomjoin algorithm in MapReduce

*False positives* are test results that indicate elements belong to  $S$  but in reality they are not in  $S$ . In contrast, *false negatives* are test results that specify elements not in  $S$  but actually they are in  $S$ .

It is very important that Bloom filters will only return false positives and never give false negatives.

Assume that the hash functions are perfectly random,  $k$  is the number of hash functions, and  $m$  is the size of the Bloom filter  $BF(S)$ , which is the number of bits in the array. After inserting all  $n$  elements of  $S$  into the Bloom filter, the probability that a particular bit is still 0, is

$$p = \left(1 - \frac{1}{m}\right)^{kn} \stackrel{i=kn}{\approx} \lim_{i \rightarrow \infty} \left(1 - \frac{kn}{mi}\right)^i = e^{-\frac{kn}{m}} \quad (2.1)$$

Hence, the probability of a false positive for an element not in the set can be calculated as the following expression (the probability that all  $k$  bits have been previously set).

$$f = (1 - p)^k = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-\frac{kn}{m}}\right)^k \quad (2.2)$$

From equation (2.2) **Error! Reference source not found.**, setting the partial derivative of  $f$  with respect to  $k$  to zero, the false positive probability  $f$  is minimized for

$$k = \left(\ln 2 * \frac{m}{n}\right) \quad (2.3)$$

This will lead to the false positive probability is:

$$f_{\min} = \left(\frac{1}{2}\right)^k \approx 0.6185^{m/n} \quad (2.4)$$

In practice, however, we should use only a small number of hash functions because the computational overhead of each hash additional function is constant while the incremental benefit of adding a new hash function decreases after a certain threshold as shown in Figure 2.12.

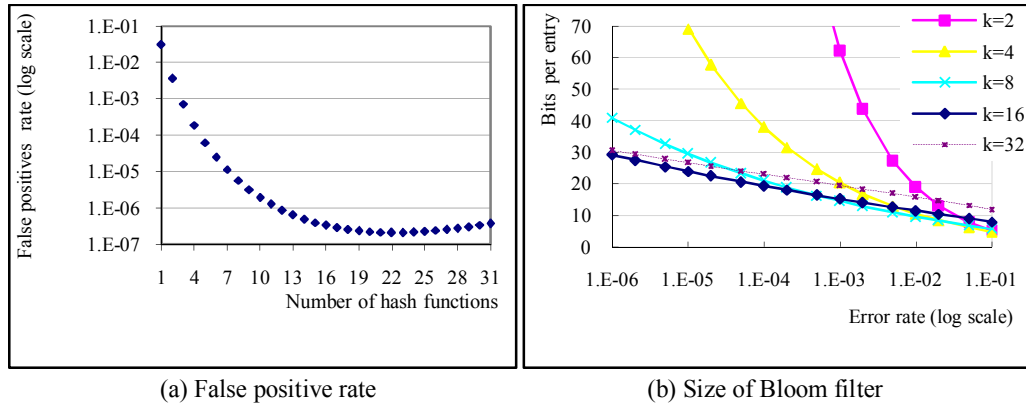


Figure 2.12: False positive rate and number of hash functions

Figure 2.12(a) depicts the false positive rate as a function of the number of hash functions used. In this illustration, the size of the Bloom filter is 32 bits per entry ( $m/n=32$ ) and thus the false positive rate is minimized for  $k=22$  hash functions. However, we can see that adding a hash function does not considerably reduce the false positive rate when more than 10 hashes are already used.

Figure 2.12(b) describes the size of the Bloom filter (bits/entry) as a function of the error rate desired. Various lines represent different numbers of hash keys used. It is found that, for the false positive rate considered, using 32 keys does not bring considerable benefits over using only eight keys.

One prominent feature of the Bloom filter is that the size of the filter is space-efficient and fixed regardless of the number of the elements of the set  $S$ , but there is a clear tradeoff between the size of the filter and the false positive probability. From equation (2.4), if the number of elements in  $S$  does not change, the error probability decreases as  $m$  increases (i.e., more memory usage). In other words, Bloom filters become more effective if we can minimize the percentage of the false positive of Bloom filters through tuning two important parameters such as  $m$  and  $k$ . There would be a good result when larger sized filters were used, but they are less efficient in memory and transmission over the network.

### 2.3.1.3 Constructing Bloom Filters

We consider a set  $S = \{s_1, s_2, \dots, s_n\}$  of  $n$  elements. A Bloom filter  $BF(S)$  represents membership information of  $S$  using a bit vector  $V$  of length  $m$  bits. The Bloom filter uses  $k$  hash functions,  $h_1, h_2, \dots, h_k$  with  $h_i : X \rightarrow \{1..m\}$ .

The following algorithm describes building the Bloom filter  $BF(S)$ .

---

#### Algorithm 5 - Building Bloom filter

---

```

BloomFilter( $S$ : a set,  $H$ : hash functions,  $m$ : size of Bloom filter)
   $bloomFilter \leftarrow$  allocate  $m$  bits initialized to 0;
  for each  $s_i$  in  $S$  do
    for each  $h_j$  in  $H$  do
       $bloomFilter[h_j(s_i)] = 1$ ;
  return  $bloomFilter$ ;

```

---

Listing 2.5: Pseudo code for building Bloom filter

From Listing 2.6, each element  $s_i$  of  $S$  is inserted into the Bloom filter  $BF(S)$  by setting  $k$  positions that correspond to hashed values of  $x$  to one.

Listing 2.6 shows the pseudo code for the algorithm of testing an element  $x$  in the Bloom filter  $BF(S)$ .

---

**Algorithm 6** - Checking an element in Bloom filter

---

```

MembershipTest( $x$ : an element,  $filter$ : Bloom filter,  $H$ : hash functions)
  for each  $h_j$  in  $H$  do
    if  $filter[h_j(x)] \neq 1$  return No
  return Yes;

```

---

Listing 2.6: Pseudo code for testing an element in Bloom filter

To check an element  $x$  in the set  $S$ , the algorithm tests the  $k$  positions ( $h_i(x)$ ) in the filter. If all the  $k$  positions are set to 1,  $x$  may be a member of  $S$ . Otherwise,  $x$  is not a member of  $S$ . In other words,  $x$  may be an element of  $S$  if and only if all hashed values of  $x$  are set to one in  $BF(S)$ .

### 2.3.2 Bloomjoin algorithm description

*Bloomjoin* is mentioned as the Reduce-side join using Bloom filter in [11][4]. This algorithm is also originated from the Bloomjoin approach in DBMSs [61][56][62] which is an improvement of the semi-join approach by using the well-known Bloom filter [15], a space-efficient data structure, to represent the  $L.uid$  instead of using a hash table.

We present the Bloomjoin algorithm without any modifications to the basic MapReduce framework.

We still use the two datasets *user*  $R(uname, uid)$  and *log*  $L(uid, event)$  as an illustration of the Bloomjoin. The algorithm is presented in two computing stages:

- *Stage 1*: projects all tuples of the input dataset  $L$  on the join key column  $uid$ , hashes these keys into a Bloom filter, and stores the filter into a file  $BF\_L.uid$ . The filter is not dependent on the number of the keys as well as key duplication and the size of the filter is small.
- *Stage 2*: distributes the file  $BF\_L.uid$  to all the compute nodes, uses this filter to eliminate non-joining tuples in  $R$ , then performs the join of the input dataset  $L$  and the filtered version of  $R$ .

The Bloomjoin algorithm can be implemented by using two MapReduce jobs as shown in Figure 2.13.

- *Job 1*: builds a Bloom filter  $BF\_L.uid$  storing all join keys of  $L$  by projecting tuples of  $L$  on the join key column ( $uid$ ). It is a full MapReduce job with only one reducer. The mappers scan splits of  $L$ , extract the join key column for each tuple, insert these join keys into local Bloom filters without considering key duplication because of characteristics of the Bloom filter, and then emit the local filters to the reducer. The reducer receives all the local filters from

all the mappers, merges these filters into a global filter by using the bit-wise **OR**. The global filter is stored into a file  $BF\_L.uid$  on DFS.

- **Job 2:** filters out non-joining tuples in  $R$  and joins filtered version  $R'$  with  $L$ . This job is a typical MapReduce job. We use a *distributed cache* to distribute the file  $BF\_L.uid$  to all the compute nodes and the mappers load it into an in-memory Bloom filter structure using an initialization function. The mappers scan splits of  $R$  and  $L$ , and extract the join key column for each tuple (*joinKey*). For the tuples of  $R$ , the mappers check their join keys in the filter  $BF\_L.uid$ , and emit the tuples whose keys are in the filter. The tuples of  $L$  are not filtered. Each the tuple is transformed into a pair in form of  $((joinKey, tag), tuple)$  and emitted to the reducers. The join processing is executed in the reduce phase similar to the Reduce-side join algorithm.

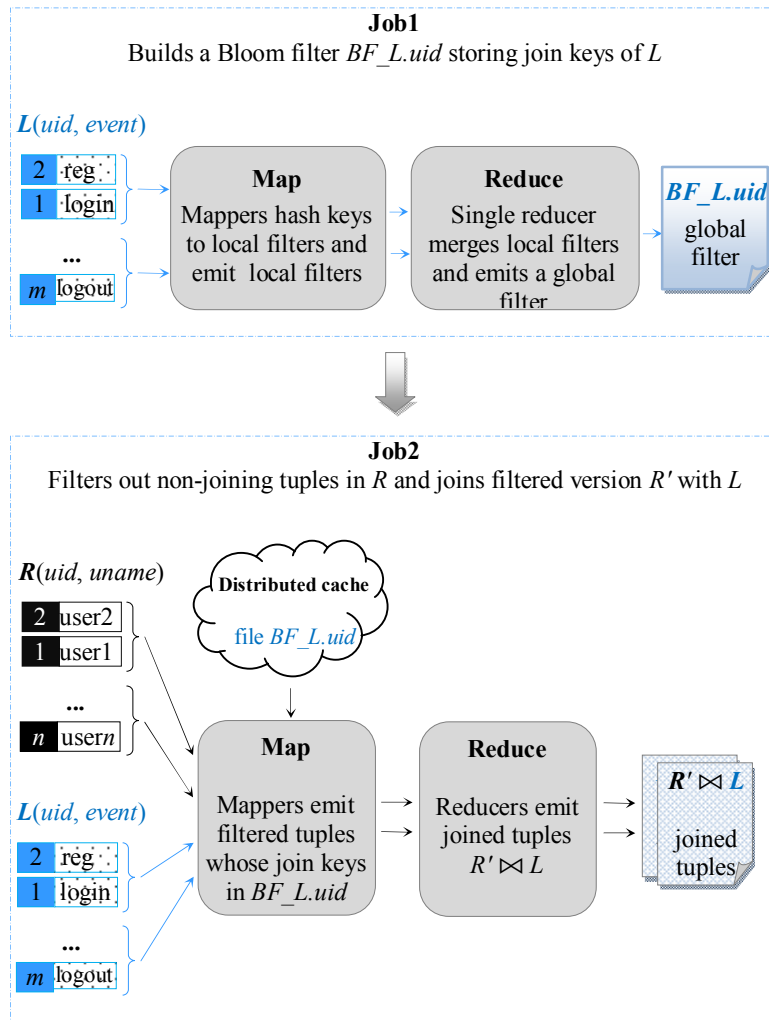


Figure 2.13: Bloomjoin in MapReduce

Listing 2.7 presents the pseudo code of the Bloomjoin algorithm.

---

**Algorithm 7** - Bloomjoin algorithm

---

**Job 1:** builds a Bloom filter  $BF_{L.uid}$  storing all join keys of  $L$

**Init\_Map()** // init function for map phase

$bfilter \leftarrow \text{empty};$  //storing keys of  $L$

**Map**( $k$ : null,  $v$ : a tuple from an  $L$  split)

$joinKey \leftarrow \text{extract the join column from } v$

add  $joinKey$  to  $bfilter$ ;

**Close\_Map()** // close function for map phase

**emit**(null,  $bfilter$ );

**Init\_Reduce()** // init function for reduce phase

$globalBF \leftarrow \text{empty};$  //merging local filters

**Reduce**( $k'$ : null,  $v'$ : a list of *local bloom filters*)

**for** each  $bfilter$  in  $v'$  **do**

**OR**( $bfilter$ ,  $globalBF$ );

**Close\_Reduce()** // close function for reduce phase

Save the filter structure  $globalBF$  into a file  $BF_{L.uid}$  on DFS

**Job 2:** filters out non-joining tuples in  $R$  and joins  $R'$  with  $L$

**Init\_Map()** // init function for map phase

$globalBF \leftarrow \text{load}(BF_{L.uid});$  //loading filter

**Map**( $k$ : null,  $v$ : a tuple from an  $R$  or  $L$  split)

$tag \leftarrow \text{a bit 0 or 1 corresponding to name of } R \text{ or } L;$

$key \leftarrow \text{extract the join key from } v;$

**if** ( $tag == '1' \parallel (tag == '0' \ \& \ key \text{ in } globalBF)$ ) **then**

**emit**(pair( $key$ ,  $tag$ ),  $v$ );

**Partitioner**( $k'$ : taggedkey,  $v$ : value,  $p$ : the number of reducers)

**return** hash\_func( $k'.key$ ) **mod**  $p$ ;

**Init\_Reduce()** // init function for reduce phase

$currentKey \leftarrow '0';$  //for storing current key

$buff \leftarrow \text{empty};$  //for storing tuples with same key of  $R$

**Reduce**( $k'$ : taggedkey,  $v'$ : list of *values*  $v$  with key  $k'$ )

**if**  $k'.key \neq currentKey$  **then**

clear( $buff$ );

$currentkey = k'.key;$

**endif**

---

---

```

if  $k'.tag == '0'$  then
  for each  $l$  in  $v'$  do
    add tuple  $l$  to  $buff$ ;
else if  $k'.tag == '1'$  then
  for each  $l$  in  $v'$  do
    for each  $r$  in  $buff$  do
      emit(null, pair( $r, l$ ));
end if

```

---

Listing 2.7: Pseudo code for Bloomjoin algorithm

Advantages and disadvantages of the Bloomjoin algorithm are listed below:

- Advantages:
  - Like the semi-join, the Bloomjoin is an approach that can be used to reduce the amount of data transferred and perform efficient join processing without any restrictions on input datasets. The algorithm uses a compact join key representation that is a bit vector for distributing to all the compute nodes rather than transferring values of the join keys as the semi-join done. It should be noted that the size of the filter does not depend on the number of join keys. In addition, the algorithm uses only two MapReduce jobs instead of the three jobs, thus it is more efficient than the semi-join.
  - There is now an effort to improve the Bloomjoin algorithm by Lee et al [4]. The join only includes a MapReduce job. However, Lee have made two changes to the typical MapReduce framework by assigning map tasks in the order of the dataset and building the filter with the heartbeat technique.
- Disadvantages:
  - The algorithm uses an additional job for building the filter, which represents extra costs as scanning the input dataset  $L$  two times, intermediate filters, communication cost, etc.
  - Broadcasting the filter becomes inefficient if the size of the filter is large. Additionally, this approach also accepts a small false positive rate in filtering the non-joining tuples.

## 2.4 Summary

This chapter reviews the two of popular and important techniques for handling large-scale datasets, the MapReduce framework and the Bloom filter. The MapReduce programming model enables easy development of scalable parallel applications to process vast amounts of data. The Bloom filter based on space efficiency has found applications in many fields, especially databases [63][64][65][59]. An illustration of using the Bloom filter is an optimization for the join processing.



Furthermore, we provide a state of the art on the status of studies on joins with MapReduce and the recent research. We present an overview of the prominent join algorithms and categorize them with respect to their strategies. The two main existing approaches in literature for the MapReduce join operation are: (1) the Map-side join approach, and (2) the Reduce-side join approach. In particular, we have introduced the semi-join and Bloomjoin algorithms that allow reducing the amount of redundant data transferred over the network, and the communication costs. In addition, we also show the advantages and disadvantages of the join algorithms.

Through the survey, we realize that there remain a lot of non-joining data sent to the reducers in the existing join algorithms. The Bloomjoin can only remove redundant data in one input dataset. Therefore, we need to look for a type of filter that has the ability to eliminate all tuples whose join keys are not common keys in input datasets. There are some important variations of the Bloom filter such as compressed Bloom filter [66], spectral Bloom filter [67], Bloomier filter [68], space-code Bloom filter [69], distance-sensitive Bloom filter [70], etc. A variant called *Counting Bloom Filters* (CBF) [71] allows deletion of elements from the Bloom filter by using counters instead of a single bit at every position. Furthermore, another version of the Bloom filter is *Invertible Bloom Filters* (InvBF) [72] that supports not only the insertion, deletion, and lookup of elements, but also enables a listing of its contents with a probability. However, all the filters are not designed for our purposes. As a result, an intersection filter for optimization of joins should be proposed.

Multi-way joins can get benefits from the above idea because all their intermediate join results contain only actual joining data.

Nevertheless, some of problems of recursive joins still exist. As mentioned, our recursive join is computed as an iteration of the join and difference operations with the loop-invariant input relation  $K_0$ . We face two problems for manually chaining multiple MapReduce jobs. The first one is the loop-invariant data  $K_0$  that must be rescanned, retransformed, and reshuffled on each iteration. It incurs significant overheads such as I/O, CPU, and communication. The second problem is the termination condition involving a fixpoint, i.e., the output of the current iteration and the previous iteration is the same. This condition requires an additional MapReduce job on each iteration to specify the fixpoint. It must once again incur substantial overheads such as scheduling the extra job, rescanning the output of two last iterations, and transferring large amounts of data via network.

To overcome the existing limitations of the Hadoop MapReduce framework for iterative applications, we mention the HaLoop framework to deploy our recursive joins. HaLoop caches the loop-invariant input dataset  $K_0$  and the output of each iteration on the physical node's local disk for later reuse. If a cache becomes unavailable, it is automatically reloaded, either from map task physical nodes, or from HDFS. More importantly, we extend the Bloom filter to be able to specify difference elements between datasets. Our difference filter is then used in the recursive join without using an additional job. This improvement reduces many associated overheads.

All these elements help us devise better optimizations for the joins that are the main subject of this research.

## **Part II**

# **Contributions**



---

## OPTIMIZATION FOR TWO-WAY JOINS AND IMPORTANT MULTI-WAY JOINS

MapReduce has become an attractive and dominant model for processing large-scale datasets. However, this model is not designed to directly support operations with multiple inputs as joins. Many current studies on join algorithms including both Bloomjoins in MapReduce have been conducted but they still have much redundant data generated and transmitted over the network. This research will help us address the problem by providing a new type of filter called *Intersection Bloom filter* using a probabilistic model to remove most non-joining elements between input datasets. Namely, three ways are proposed on the intersection filter. We then consider two-way joins and important multi-way joins using the intersection filter, and analyze their costs. As a result, thanks to the high accuracy intersection filter, the join processing can minimize disk I/O and communication costs. Finally, the research is proved to be more efficient than existing solutions through a cost-based comparison and experiments of joins using different approaches.

This chapter is formed as follows. Section 3.1 provides a short description of previous work as well as points out its limitation. We then introduce an overview of our contributions, definitions and notations. The remainder of the chapter therefore presents our proposals in detail. Section 3.2 describes three approaches to building the intersection filter with a small false intersection probability. Section 3.3 uses the intersection filter for optimizing two-way joins. A cost model and a cost comparison of two-way join algorithms are given. Next, we show advantages of an extended intersection filter (EIF) for optimizing multi-way joins in Section 3.4, namely, three-way joins and chain joins. The Lagrangian method is used to help us choose a three-way join cascade and a two-way join cascade. In addition, two optimized solutions using the EIF for chain joins are suggested in this section. Moreover, two cost models for three-way joins and chain joins are also provided. Thanks to the cost models, we can make comparisons of different join algorithms for multi-way joins more convincing. The evaluation environments, experimental protocols and experiments are reported in Section 3.5. Finally, Section 3.6 includes conclusions on our work.

### 3.1 Introduction

#### 3.1.1 Previous work

Bloomjoins in DBMSs [61][56][62] with the necessary modifications have been deployed to handle large datasets in MapReduce. However, the standard Bloom filter in Bloomjoins only have the ability to remove non-joining tuples from one of input

### 3.1 Introduction

datasets instead of both. As a result, there remains a large amount of non-joining data from another input dataset sent to the reducers for the join processing. Figure 3.1 describes an illustration of a basic join operation using the Bloom filter in MapReduce.

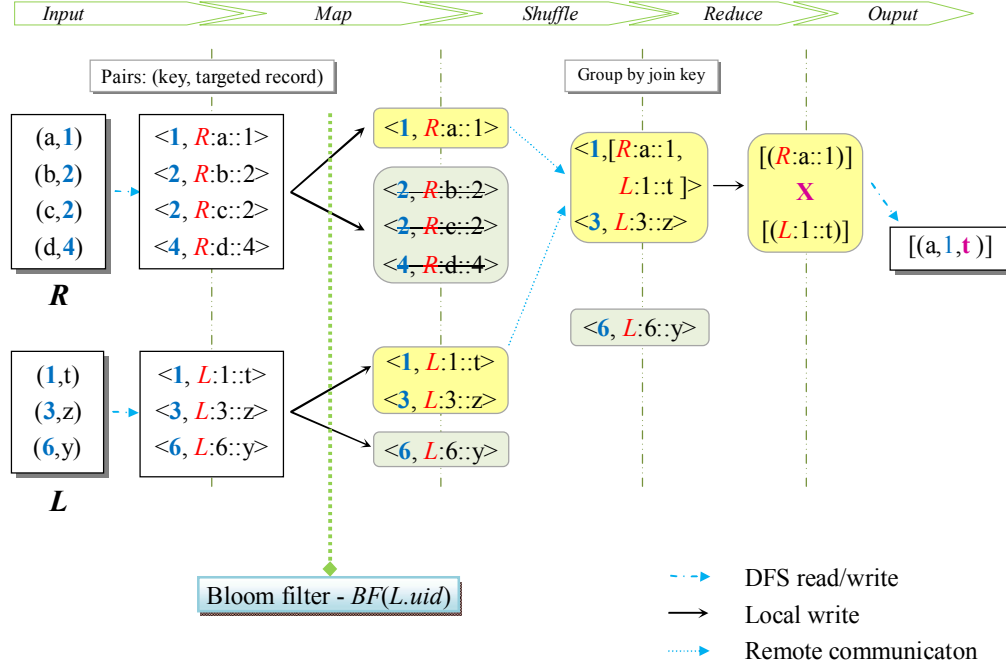


Figure 3.1: Basic join operation using  $BF$  in MapReduce

A Bloom filter  $BF(L.uid)$  is first built for an input dataset  $L$  on a join key column  $uid$  and is delivered across all the mappers. Each the mapper receives tuples from  $R$  or  $L$ , it eliminates tuples whose join keys are not in  $BF(L.uid)$  and emits key/value pairs for the remaining tuples. Then, the pairs are passed to corresponding reducers to be joined. However, the filter efficiency of the Bloomjoin algorithm and even with the recent extended researches [11][4] has not really been taken into consideration yet. Namely, with using only the filter  $BF(L.uid)$  for both the two inputs  $R$  and  $L$ , the algorithm can only eliminate non-joining tuples of the dataset  $R$  (e.g., tuples with the join key values of 2 and 4) without eliminating non-joining tuples of the dataset  $L$  (e.g., tuples with the join key values of 3 and 6). This redundancy considerably increases associated overheads in cases of multi-way joins and iterative joins.

We can see that the actual results of the inner join operation only contain tuples whose join keys belong to the intersection of the two input datasets projected on the join key column. As shown in Figure 3.1, the output is tuples whose join keys have the same value as 1 and belong to the intersection of  $R.uid$  and  $L.uid$ :

$$\{1\} = \{1, 2, 4\} \cap \{1, 3, 6\}.$$

For this reason, we need to build a new filter type representing the intersection of the input datasets to be able to filter out non-joining data in both of these datasets. The complex joins can take advantages from our proposed filter. This chapter, therefore, makes the following main contributions: (a) the intersection filter with

three approaches that approximates the intersection of datasets; (b) optimization for two-way joins and multi-way joins using the intersection filter in MapReduce; (c) comparison among the joins using different approaches through cost models and experiments.

It should be noted that major research subjects of the chapter are the queries  $Q_1$  and  $Q_2$  introduced in Chapter 2, which are inner join queries.

### 3.1.2 Definitions and notations

We supply definitions and notations used in this research as follows.

**Definition 3.1.** The *intersection* (denoted  $\cap$ ) of two or more sets is the set of elements that are common in all the sets.

**Definition 3.2:** An *Intersection Bloom Filter* (IBF) is a probabilistic data structure designed to represent the intersection of sets. It is used to recognize common elements of the sets with a false positive probability.

Notations are given by Table 3.1.

Table 3.1: List of notations

Notation	Explanation
$ S $	The cardinality of a set $S$ , which is the number of elements in set $S$
$\setminus$	The difference operator
$\cup$	The union operator
$\cap$	The intersection operator
$S_1 \cap S_2$	The intersection of two sets $S_1$ and $S_2$
$IF(S_1 \cap S_2)$	The general intersection filter representing the intersection of $S_1$ and $S_2$
$BF(S)$	The Bloom filter built for a set $S$
$BF(S_1) \cap BF(S_2)$	Intersecting two Bloom filters $BF(S_1)$ and $BF(S_2)$
$IBF(S_1 \cap S_2)$	The Intersection Bloom filter representing the intersection of $S_1$ and $S_2$
$f_{\cap BF}$	The false intersection probability of Bloom filters
$f_{\cap PBF}$	The false intersection probability of partitioned Bloom filters
$EIF$	The extended intersection filter

### 3.2 Modeling intersection filter

We propose a filter type called *Intersection Filter* (IF) as follows:

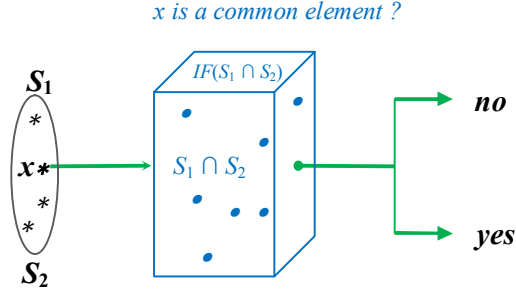


Figure 3.2: Intersection Filter returning an output with two possibilities

As illustrated in Figure 3.2, an intersection filter  $IF(S_1 \cap S_2)$  represents an approximation of the set intersection,  $(S_1 \cap S_2)$ . It is used to check whether an element  $x$  is a common element (i.e.  $x \in S_1 \cap S_2$ ). The intersection filter accepts an input and returns an output that is one of two possibilities:

- "no" :  $x$  is NOT a common element of the sets  $S_1$  and  $S_2$ .
- "yes" :  $x$  may be a common element of the sets  $S_1$  and  $S_2$ .

With this design, when the intersection filter returns an answer "no", the answer is always the correct response. An answer "yes" may be the wrong response because  $x$  may be NOT a common element. It also means that the intersection filter returns "yes" answers with a false positive probability. As a result, the intersection filter enables us to specify a superset of common elements including the "yes" elements, and eliminate disjoint elements that are the "no" elements. Accordingly, we should minimize false positives for the intersection filter.

This section shows three approaches based on Bloom filters to build the intersection filter, known as *intersection Bloom filter* (IBF). For convenience, two Bloom filters  $BF(S_1)$  and  $BF(S_2)$  are used as the concise representation of two input datasets  $R$  and  $L$  projected on the join key column, respectively. Since each of the Bloom filters has the false positive probability, there exist "false" common elements discovered by the intersection Bloom filter.

#### 3.2.1 Approach 1: a pair of Bloom filters

First, we observe the following expression for set intersection representation of two sets  $S_1$  and  $S_2$ .

$$S_1 \cap S_2 = (S_1 \cup S_2) \setminus (S_1 \Delta S_2) = (S_1 \cup S_2) \setminus ((S_1 \setminus S_2) \cup (S_2 \setminus S_1))$$

From the above expression, we can specify the set intersection by eliminating all elements of the difference between the sets. Precisely, the intersection filter

recognizes common elements in the set  $S_1$  by  $BF(S_2)$  and common elements in the set  $S_2$  by  $BF(S_1)$ . To achieve this work, we use a pair of Bloom filters as follows.

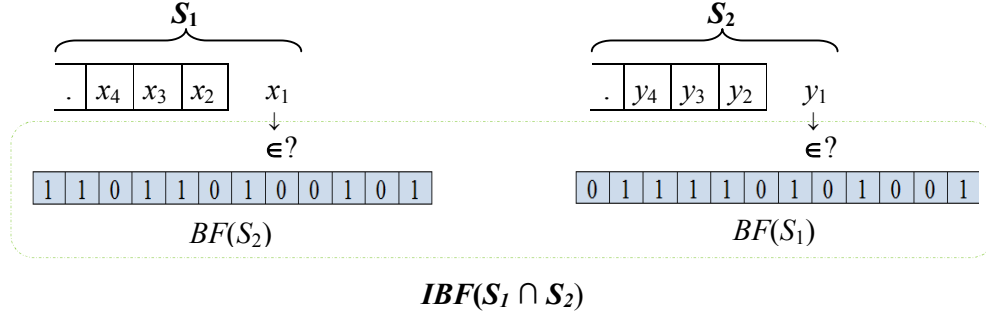


Figure 3.3: Intersection filter using a pair of Bloom filters

Each element in one set is queried into a Bloom filter of another set by  $k$  hash functions. If the element is a member of the filter, the intersection filter returns a "yes" answer because the element may be a common member of the sets. Otherwise, the intersection filter returns a "no" answer because the element belongs to the set difference. As Figure 3.3, for instance, an element  $x_1$  of the set  $S_1$  is queried into  $BF(S_2)$  and an element  $y_1$  of the set  $S_2$  is queried into  $BF(S_1)$ . If  $x_1$  is not a member of  $BF(S_2)$ , the output of the  $IBF$  is "no" answer. If  $y_1$  is in  $BF(S_1)$ , the output of the  $IBF$  is "yes" answer. After all elements of  $S_1$  and  $S_2$  are respectively queried into  $BF(S_2)$  and  $BF(S_1)$ , we get all common elements that are "yes" answers. This corresponds to the operation of  $(S_1 \cup S_2) \setminus ((S_1 \setminus S_2) \cup (S_2 \setminus S_1))$ . In other words, the intersection filter  $IBF(S_1 \cap S_2)$  can be obtained through the pair of the Bloom filters.

This approach does not require the filters to have the same size  $m$  and  $k$  hash functions.

### 3.2.2 Approach 2: intersecting unpartitioned Bloom filters

The second approach is based on the idea that intersecting Bloom filters will produce a result filter called the intersection filter.

There is little difference between the intersection filter and the intersection of Bloom filters as shown in [73] then  $IBF(S_1 \cap S_2) = BF(S_1) \cap BF(S_2)$  with probability  $(1 - 1/m)^{k \cdot |S_1 - S_1 \cap S_2| \cdot k \cdot |S_2 - S_1 \cap S_2|}$ .

The intersection of filters is not sufficient to accurately calculate the intersection filter  $IBF(S_1 \cap S_2)$ . However, we can get an approximation of the  $IBF$  by joining  $BF(S_1)$  and  $BF(S_2)$  with the bit-wise **AND**, and the intersection Bloom filter still maintains the inherent querying features [74][73]. This means that if all  $k$  positions hashed for a join key  $x$  are set 1 in the intersection filter,  $x$  belongs to  $S_1 \cap S_2$  with high probability.

In this approach, we use standard (unpartitioned) Bloom filters with the same size  $m$  and  $k$  hash functions. Building the intersection filter is shown in Figure 3.4.



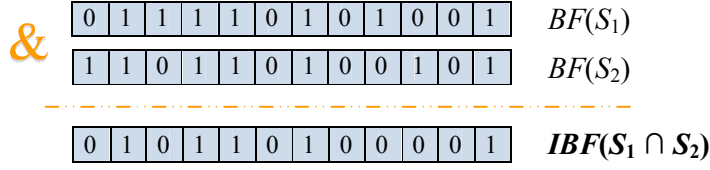


Figure 3.4: Intersection filter based on intersecting *unpartitioned* Bloom filters

It illustrates that the intersection filter  $IBF(S_1 \cap S_2)$  is formed by intersecting two standard Bloom filters  $BF(S_1)$  and  $BF(S_2)$  with the bitwise AND operator. This can be expressed by the following form:

$$IBF(S_1 \cap S_2) = BF(S_1) \& BF(S_2)$$

It performs the bitwise AND operation between the two bit arrays  $BF(S_1)$  and  $BF(S_2)$  with the same size, and place it in the third array  $IBF$ . The  $IBF$  is now an approximate representation of the set intersection  $(S_1 \cap S_2)$ .

Querying an element  $x$  into the intersection filter  $IBF$  is similar to the standard Bloom filter. If  $x$  is in  $IBF(S_1 \cap S_2)$ , it returns a "yes" answer because  $x$  may be a common element. Otherwise, it returns a "no" answer because  $x$  is an element of the difference.

With this approach, we only maintain one intersection Bloom filter to remove most non-joining tuples from both input datasets instead of using two filters as the first approach.

### 3.2.3 Approach 3: intersecting partitioned Bloom filters

Our last approach begins with the same idea as the second approach to create the intersection filter but we use *partitioned* Bloom filters.

#### 3.2.3.1 Partitioned Bloom filter

A partitioned Bloom filter [75], a variant of the standard Bloom filter, is defined by an array of  $m$  bits that is partitioned into  $k$  disjoint arrays of size  $m_p = m/k$  bits. Figure 3.5 suggests that  $BF(S)$  consists of three 4-bit partitions,  $k=3$  hash functions and size  $m=12$  bits.

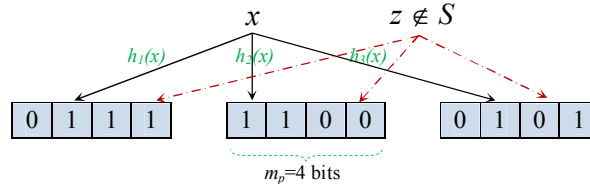


Figure 3.5: Partitioned Bloom filter  $BF(S)$

We insert an element  $x \in S$  into  $BF(S)$  by computing  $h_i(x)$  and setting the corresponding position in the  $i^{\text{th}}$  partition to 1 ( $i=1 \dots k$ ). Similarly, we test if the element  $z$  is in  $S$  by checking the position corresponding to  $h_i(z)$  in the  $i^{\text{th}}$  partition.

The probability that a bit has remained 0 after inserting  $n$  elements for the standard filter  $p$  and the partitioned filter  $p_p$  is asymptotically equivalent. Precisely, the standard Bloom filter tends to perform slightly better than the partitioned Bloom

filter since when  $k > 1$  the standard filter tends to have more 0's than the partitioned filter as shown in the following expression. From expression (2.1), we derive the following inequality:

$$p = \left(1 - \frac{1}{m}\right)^{kn} > p_p = \left(1 - \frac{k}{m}\right)^n \quad (3.1)$$

However, the partitioned filter is more flexible than the standard Bloom filter. After building the partitioned filter, its false positive probability can still be changed by increasing or reducing its partitions without rehashing. Consequently, intersecting (merging) partitioned filters with different sizes can be performed by using the bit-wise **AND** (**OR**) of two bit-arrays. Obviously, the resolution of one of the partitioned Bloom filters may be adjusted. For example, the partitioned filter  $BF(S_1)$  has three 4bit-partitions (the filter size  $m_1=12$  bits),  $BF(S_2)$  has two 4bit-partitions (size  $m_2=8$  bits). To intersect these filters, we only eliminate the third partition of  $BF(S_1)$ , then AND two remaining partitions of  $BF(S_1)$  and  $BF(S_2)$ . This affects the resolution of  $BF(S_1)$ . In contrast, we cannot reduce the size of the unpartitioned filter because we have to completely rehash the bit array of the filter.

### 3.2.3.2 Intersection filter design

We describe the intersection filter  $IBF(S_1 \cap S_2)$  built from two partitioned Bloom filters  $BF(S_1)$  and  $BF(S_2)$  in Figure 3.6.

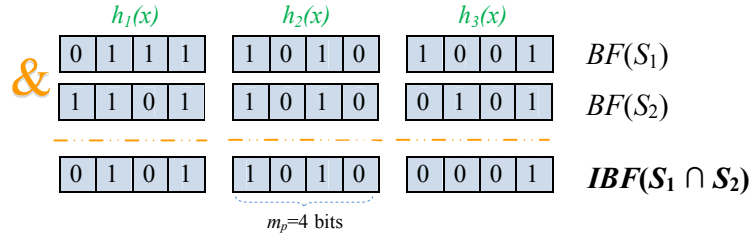


Figure 3.6: Intersection filter based on intersecting *partitioned* Bloom filters

The intersection filter  $IBF(S_1 \cap S_2)$  is generated by intersecting pairs of partitions of the two partitioned filters. As shown in Figure 3.6,  $BF(S_1)$  and  $BF(S_2)$  including three partitions are pairwise intersected with the bitwise **AND** to produce  $IBF(S_1 \cap S_2)$  including three 4-bit partitions. The filter  $IBF(S_1 \cap S_2)$  represents the approximate intersection of the sets  $S_1$  and  $S_2$ .

An interesting characteristic of this approach is that if the intersection filter  $IBF(S_1 \cap S_2)$  has at least one partition with all  $m/k$  bits set to 0, the sets  $S_1$  and  $S_2$  are disjoint. Consequently, the join processing can be finished without doing anything. This characteristic is really useful for joins and is not present in the first approach. Even for the second approach, it would be rare for all  $m$  bits to be equal to 0.

In addition, the third approach does not require the filters  $BF(S_1)$  and  $BF(S_2)$  to have the same size. In this case, the filters may have different sizes but their partitions should have the same size and the same hash functions. We can adjust the partitioned filters  $BF(S_1)$ ,  $BF(S_2)$  and  $IBF(S_1 \cap S_2)$  by reducing or adding partition(s) without rehashing. Similar to the second approach, we also maintain one intersection filter for filtering non-joining data in both input datasets.

### 3.2.4 The false intersection probability

We can represent the intersection of two sets with false positives as follows.

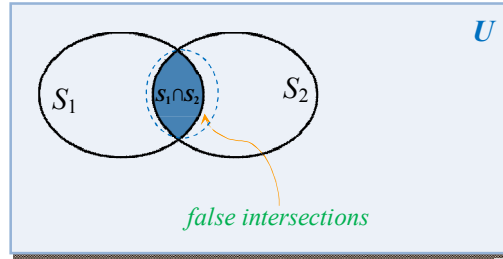


Figure 3.7: Set intersection representation using Bloom filters

The dark area (blue) in Figure 3.7 shows the actual intersection of the sets  $S_1$  and  $S_2$ . The bright area inside the dashed oval represents false positives of the set intersection, also known as *false intersections*. The false intersections result from intersecting the Bloom filters  $BF(S_1)$  and  $BF(S_2)$ .

The false intersection probability for each of the approaches corresponds to each theorem below.

**Theorem 3.1.** *A false intersection by a pair of Bloom filters is identified with one of the two probabilities*

a. For  $BF(S_1)$ ,

$$f_{\cap pair(S_1)} = \left( 1 - \left( 1 - \frac{1}{m_1} \right)^{k_1 |S_1|} \right)^{k_1}$$

b. For  $BF(S_2)$ ,

$$f_{\cap pair(S_2)} = \left( 1 - \left( 1 - \frac{1}{m_2} \right)^{k_2 |S_2|} \right)^{k_2} \quad (3.2)$$

where  $m_1$  and  $m_2$  correspond to the sizes of  $BF(S_1)$  and  $BF(S_2)$ ;  $k_1$  and  $k_2$  are the numbers of hash functions of  $BF(S_1)$  and  $BF(S_2)$ , respectively.

**Proof.** As shown in Figure 3.3, we obtain the approximate intersection of the sets thanks to the pair of Bloom filters  $BF(S_1)$  and  $BF(S_2)$ . From equation (2.2), it is easy to show that the false intersection probability of  $BF(S_1)$  for the set  $S_2$  is  $f_{\cap pair(S_1)}$  as equation (3.2) and the false intersection probability of  $BF(S_2)$  for the dataset  $S_1$  is  $f_{\cap pair(S_2)}$  as equation (3.2)  $\square$ .

**Theorem 3.2.** *A false intersection by intersecting unpartitioned Bloom filters is identified with probability*

$$f_{\cap BF} = \left( 1 - \left( 1 - \frac{1}{m} \right)^{k |S_1|} \right)^k \left( 1 - \left( 1 - \frac{1}{m} \right)^{k |S_2|} \right)^k \quad (3.3)$$

where  $BF(S_1)$ ,  $BF(S_2)$  and  $IBF(S_1 \cap S_2)$  have the same size  $m$  and  $k$  hash functions.

**Proof.** It argues that the intersection of the unpartitioned Bloom filters causes false intersections when all  $k$  bits in the resulting bit array is set to 1 from two different join keys. From equation (2.2), the probability for  $k$  bits to be set in both  $BF(S_1)$  and  $BF(S_2)$  from two different keys is the product of

$$f_1 = \left(1 - \left(1 - \frac{1}{m}\right)^{k|S_1|}\right)^k \text{ and } f_2 = \left(1 - \left(1 - \frac{1}{m}\right)^{k|S_2|}\right)^k$$

It is also the false positive probability of  $IBF(S_1 \cap S_2)$ , and thus the theorem has been demonstrated  $\square$ .

**Theorem 3.3.** *A false intersection by intersecting partitioned filters is identified with probability*

$$f_{\cap PBF} = \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1|}\right)^k \left(1 - \left(1 - \frac{k}{m}\right)^{|S_2|}\right)^k \quad (3.4)$$

where  $BF(S_1)$ ,  $BF(S_2)$  and  $IBF(S_1 \cap S_2)$  have the same size  $m$  and  $k$  hash functions,  $k$  partitions are the same size  $m_p = m/k$ .

**Proof.** Similar to Theorem 3.2, the probability for  $k$  bits to be set in  $k$  partitions of  $BF(S_1)$  and  $BF(S_2)$  with two different keys, thus also falsely be in  $IBF(S_1 \cap S_2)$  is the product of

$$f_{p_1} = \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1|}\right)^k \text{ and } f_{p_2} = \left(1 - \left(1 - \frac{k}{m}\right)^{|S_2|}\right)^k$$

It completes the proof of the theorem  $\square$ .

**Theorem 3.4.** *The false intersection probability of the unpartitioned filter intersection is less than the false intersection probability of the partitioned filter intersection  $f_{\cap BF} < f_{\cap PBF}$*  (3.5)

**Proof.** If Bloom filters have more bits set to 1, the probability for a bit collision can be higher. And thus partitioned filters tend to have more 1's than unpartitioned filters. As equation (3.1), we get

$$\left(1 - \frac{1}{m}\right)^{k|S_1|} > \left(1 - \frac{k}{m}\right)^{|S_1|} \Rightarrow \left(1 - \left(1 - \frac{1}{m}\right)^{k|S_1|}\right)^k < \left(1 - \left(1 - \frac{k}{m}\right)^{|S_1|}\right)^k$$

From the above equation, we can simply compute the theorem  $\square$ .

It can be seen that the intersection filter by a pair of the filters is the most flexible approach because the hash functions and the size of the filters can be different while other approaches require the same size that affects the resolution of filters. In contrast, the first approach needs to maintain two filters while others only maintain one filter on nodes. Besides, the second and third approaches enable us to discover disjoint sets and early stop the join processing. This characteristic is very important for evaluating multi-way joins and recursive joins.

### 3.3 Optimization for two-way joins using intersection filters in MapReduce

We take a closer look at the semi-join and Bloom-join algorithms in Sections 2.2.4 and 2.3.2 of Chapter 2. The algorithms remove the non-joining tuples from only one input dataset  $R$ . Consequently, the non-joining tuples in another input dataset  $L$  have still not been filtered. For instance from Facebook, a largest online social network with 1.23 billion monthly active users as of December 31, 2013 [60], the log dataset  $L$  contains user's activities that many of them can deactivate their personal profiles. Hence, the join operation of the user dataset  $R$  and the log dataset  $L$  leads to many non-joining tuples of the dataset  $L$  transferred across the network. Our join optimization takes the data redundancy in both  $R$  and  $L$  sent to the join processing into consideration.

#### 3.3.1 Implementation overview

We implement a two-way join of the two datasets  $user\ R(uid, uname)$  and  $log\ L(uid, event)$  to evaluate the query  $Q_1$  in Chapter 2. Because  $R$  and  $L$  are two arbitrary input datasets, we will discuss and evaluate the join operation in general by using Reduce-side join type. However, it can still use our intersection filter for the Map-side join type.

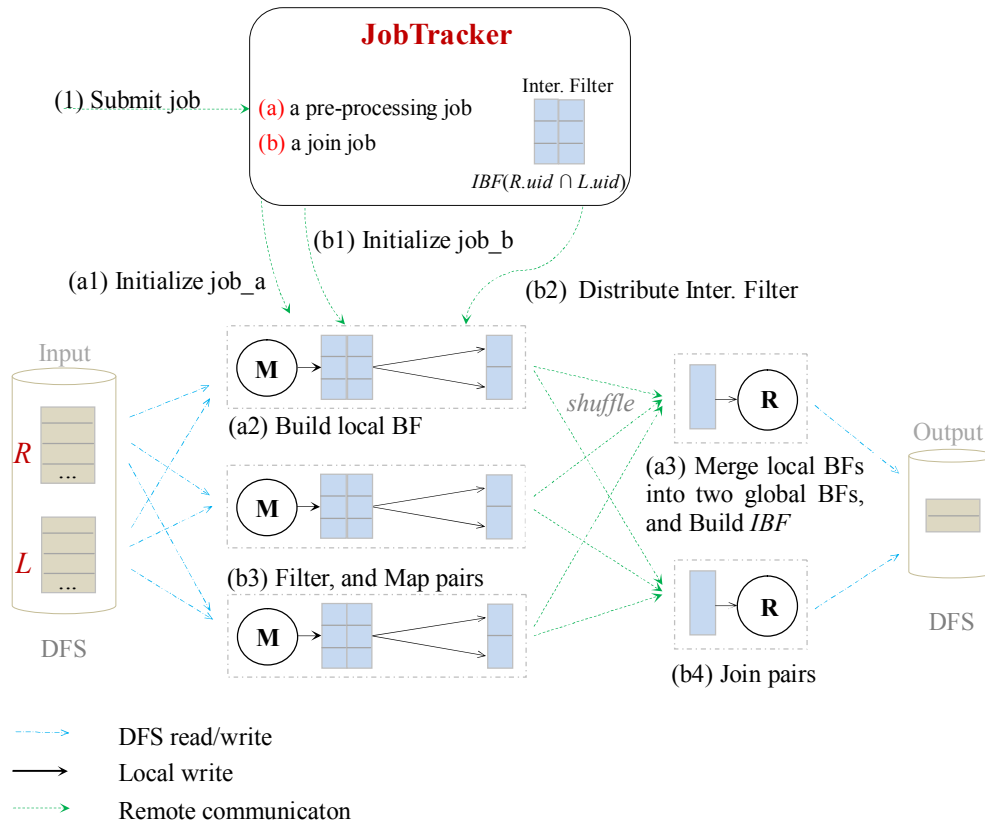


Figure 3.8: Join implementation using intersection filter in MapReduce

The intersection filter-based join algorithm involves the following two computing stages:

- *Stage 1*: projects all tuples of both the input datasets  $L$  and  $R$  on the join key column  $uid$ , hashes the keys into two Bloom filters, computes the intersection filter, and stores the filter into a file  $IBF.uid$ .
- *Stage 2*: distributes the file  $IBF.uid$  to all compute nodes, uses this intersection filter to remove non-joining tuples in both  $R$  and  $L$ , and then computes the join of two filtered datasets of  $R$  and  $L$ .

Figure 3.8 illustrates the algorithm with two MapReduce jobs corresponding to the two stages. The first job, known as *pre-processing job*, is to build the intersection filter  $IBF(R.uid \cap L.uid)$ . The second one performs a *join job* using the intersection filter generated earlier to remove redundant data in the Map phase. The details of implementing the MapReduce jobs are described as follows.

### 3.3.1.1 Pre-processing job

The join operation ( $R \bowtie L$ ) together with its input datasets  $R$  and  $L$  is configured and submitted by a client to the jobtracker. At this moment, the operation is compiled into two jobs in which the join job is blocked until the end of the pre-processing job.

The pre-processing job includes two groups of parallel map tasks (*mtgroup1* and *mtgroup2*), and one reduce task for computing the intersection filter. The *mtgroup1* processes the dataset  $R$  for creating *local* filters  $BF(R.uid)$  while the *mtgroup2* independently processes the dataset  $L$  for creating *local* filters  $BF(L.uid)$ . The mappers scan splits of  $R$  and  $L$ , extract the join key column for each tuple, and insert the join keys of  $R$  and  $L$  into local Bloom filters  $BF(R.uid)$  and  $BF(L.uid)$ , respectively, on tasktrackers. The mappers emit the local filters to the reducer. The reducer receives all the local filters from all the mappers, merges these filters into two respectively *global* filters  $BF(R.uid)$  and  $BF(L.uid)$  using the bit-wise **OR** (see Section 3.3.1.3). Based on the three approaches proposed in Section 3.2, the reducer computes the intersection filter  $IBF(R.uid \cap L.uid)$  from the global filters. For the first approach, the intersection filter is a pair of the global filters and thus it does nothing. The intersection filter is then stored into a file  $IBF.uid$  on Distributed File System (DFS).

As an option, the pre-processing job can detect to not rebuild the existing filters, even this job will be omitted if the intersection filter exists. Consequently, it enables a dramatic reduction in I/O and computational overhead when joins are recalculated. In addition, when the size of filters is large, the filter files will be compressed in formats such as gzip, bzip2, LZO and Snappy. This compression is really efficient for delivering filters to all nodes.

Notably, the implementation of the pre-processing job can detect the empty intersection filter to early complete the join operation. This interesting characteristic, which is very useful for multi-way joins and recursive joins, is not present in the existing studies.

At the end of the pre-processing job, if the intersection filter is empty, the join job will be omitted and the entire join operation will be finished. Otherwise, the intersection filter will be written into a file on DFS. This pre-processing job

corresponds to activities (a1) to (a3) in Figure 3.8. The activities are the job initiation and map-reduce functions presented in Section 3.3.2.

#### 3.3.1.2 Join job

This job will use the intersection filter generated earlier to remove redundant data from both the input datasets  $R$  and  $L$ , and perform a cross join. Activities (b1) to (b4) in Figure 3.8 describe the join job in detail. In order to start the job, the intersection filter file  $IBF.uid$  is distributed to all the compute nodes in a cluster using a *distributed cache*. Then, the jobtracker will create  $mp_1$  and  $mp_2$  map tasks for the inputs  $R$  and  $L$  respectively,  $r$  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 intersection filter:

Each mapper uses an initialization function to load the file  $IBF.uid$  into memory  $IBF(R.uid \cap L.uid)$ . A tag '0' or '1' is used to tick a split of  $R$  or  $L$ , respectively. The mapper reads each tuple from its split, produces a  $\langle key, tuple \rangle$  pair, and then calls a map function to process the pair. The map function queries the join key ( $uid$ ) of the tuple into the intersection filter  $IBF$ . If the key is in the  $IBF$ , the tuple is mapped into a pair in form of  $((uid, tag), tuple)$  and emitted to the reducers. Otherwise, the tuple is omitted.

A partition function,  $partitioner()$ , ensures that partitioning the tagged tuples takes into consideration only the join key part ( $uid$ ) and ignores the tag part ( $tag$ ). The tag attached to the join key  $uid$  is used to do a secondary sort that ensures all tuples from one input dataset are processed before the other. This is implemented by overriding the default grouping function.

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:

This reduce phase is the same as the one of the basic Reduce-side join. The reducer receives the tagged tuples of the form  $((uid, tag), tuple)$  with the same  $uid$ , and calls the reduce function for each join key  $uid$ . The reduce function performs the cross product of the tuples of  $R$  that are buffered and each incoming tuple of  $L$ . It is completed by writing the output to DFS.

#### 3.3.1.3 Merging Bloom filters

We introduce the way to merge the local filters that it takes place in the Reduce phase of the pre-processing job. Merging the Bloom filters is simpler than intersecting these filters. The merging operation corresponds to the construction of the union filter  $BF(S_1 \cup S_2)$ . We can build the union Bloom filter by the union of the Bloom filters as shown in Lemma 1 and 2 below.

**Lemma 1.** [73] *Assuming  $BF(S_1)$ ,  $BF(S_2)$  and  $BF(S_1 \cup S_2)$  use the same size  $m$  and  $k$  hash functions, then*

$$BF(S_1 \cup S_2) = BF(S_1) \cup BF(S_2)$$

We can easily extend Lemma 2 out to the following fact.

**Lemma 2.** *Assuming  $BF(S_1), BF(S_2), \dots, BF(S_q)$  and  $BF(S_1 \cup S_2 \dots \cup S_q)$  use the same size  $m$  and  $k$  hash functions, then  $BF(S_1 \cup S_2 \dots \cup S_q) = BF(S_1) \cup BF(S_2) \dots \cup BF(S_q)$ .*

The union of Bloom filters with the same size and hash functions is implemented by bitwise **OR**. In the pre-processing job, therefore, the reducer collects all the local Bloom filters from tasktrackers, merges the local filters by using the bitwise OR of the bit arrays, and generates the global Bloom filter. This global filter is then intersected with another global filter to create the intersection filter.

### 3.3.2 Optimized two-way join algorithm

Listing 3.1 shows the pseudo code of the intersection filter-based join algorithm.

---

**Algorithm 1** - Two-way join algorithm using Intersection filter

---

**Job1\_2Way:** builds intersection filter *IBF.uid* storing common join keys between *R* and *L*

```

Init_Map() // init function for map phase
    bfilter_R  $\leftarrow$  empty; //storing keys of R
    bfilter_L  $\leftarrow$  empty; //storing keys of L

Map(k: null, v: a tuple from an R or L split)
    joinKey  $\leftarrow$  extract the join column from v
    add joinKey to bfilter_R or bfilter_L;

Close_Map() // close function for map phase
    emit('R', bfilter_R);
    emit('L', bfilter_L);

Init_Reduce() // init function for reduce phase
    globalBF_R  $\leftarrow$  empty; //merging local filters of R
    globalBF_L  $\leftarrow$  empty; //merging local filters of L

Reduce(k: 'R' or 'L', v: a list of local bloom filters)
    filterPointer  $\leftarrow$  null; // a pointer
    if k == 'R' then
        filterPointer = &globalBF_R;
    else
        filterPointer = &globalBF_L;
    endif

    for each bfilter in v do
        OR(bfilter, filterPointer);
    emit(null, null);

Close_Reduce() // close function for reduce phase
    IBF.uid  $\leftarrow$  empty; //intersection filter
    compute IBF.uid from globalBF_R and globalBF_L
    save IBF.uid into a file IBF.uid on DFS

```

---



### 3.3 Optimization for two-way joins using intersection filters in MapReduce

---

**Job2\_2Way:** filters out non-joining tuples in  $R$  and  $L$ , and joins filtered datasets  $R'$  and  $L'$

```
Init_Map() // init function for map phase
   $IBF.uid \leftarrow \text{load}(IBF.uid)$ ; //loading intersection filter

Map( $k$ : null,  $v$ : a tuple from an  $R$  or  $L$  split)
   $tag \leftarrow$  a bit 0 or 1 corresponding to name of  $R$  or  $L$ ;
   $key \leftarrow$  extract the join key from  $v$ ;
  if ( $key$  in  $IBF.uid$ ) then
    emit( $\text{pair}(key, tag)$ ,  $v$ );
  endif

GroupComparator( $taggedKey1$ : taggedkey,  $taggedKey2$ : taggedkey)
   $res = \text{compare}(taggedKey1.key, taggedKey2.key)$ ;
  if ( $res == 0$ ) then
     $res = \text{compare}(taggedKey1.tag, taggedKey2.tag)$ ;
  endif
  return  $res$ ;

Partitioner( $k'$ : taggedkey,  $v$ : value,  $p$ : the number of reducers)
  return  $\text{hash\_func}(k'.key) \bmod p$ ;

Init_Reduce() // init function for reduce phase
   $currentKey \leftarrow '0'$ ; //for storing current key
   $buff \leftarrow \text{empty}$ ; //for storing tuples with same key of  $R$ 

Reduce( $k'$ : taggedKey,  $v'$ : list of values  $v$  with key  $k'$ )
  if  $k'.key \neq currentKey$  then
     $\text{clear}(buff)$ ;
     $currentkey = k'.key$ ;
  endif

  if  $k'.tag == '0'$  then
    for each  $l$  in  $v'$  do
      add tuple  $l$  to  $buff$ ;

  else if  $k'.tag == '1'$  then
    for each  $l$  in  $v'$  do
      for each  $r$  in  $buff$  do
        emit(null,  $\text{pair}(r, l)$ );
  end if
```

---

Listing 3.1. Pseudo code for two-way join algorithm using intersection filter

### 3.3.3 Cost analysis for two-way joins in MapReduce

#### 3.3.3.1 Cost model

We adapt the cost model presented in [76] to suit our cost model for two-way joins. Suppose that  $R$  and  $L$  are two input datasets. Table 3.2 summarizes parameters within our cost model for two-way joins.

Table 3.2: Cost model parameters for two-way joins

Parameter	Explanation
$ R $	The size of the input dataset $R$
$ L $	The size of the input dataset $L$
$c_l$	The cost of reading or writing data locally
$c_r$	The cost of reading/writing data remotely
$c_t$	The cost of transferring data from one node to another
$B+1$	The size of the sort buffer is $B+1$ pages
$mp_1$	The number of map tasks of the dataset $R$
$mp_2$	The number of map tasks of the dataset $L$
$mp = mp_1 + mp_2$	The total number of map tasks
$t$	The number of tasktrackers
$m$	The size of the Bloom filter (bits)
$\phi$	The compression ratio for the filter file
$ O $	The size of the join processing output
$C_{pre}$	The total cost to perform the pre-processing job
$C_{read}$	The total cost to read the data
$C_{sort}$	The total cost to perform the sorting and copying at the map and reduce nodes
$C_{tr}$	The total cost to transfer intermediate data among the nodes
$C_{write}$	The total cost to write the data on DFS

### 3.3 Optimization for two-way joins using intersection filters in MapReduce

Accordingly, we get the total cost of a two-way join using various algorithms as follows:

$$C = C_{pre} + C_{read} + C_{sort} + C_{tr} + C_{write} \quad (3.6)$$

where

- $C_{read} = c_r \cdot |R| + c_r \cdot |L|$
  - $C_{sort} = c_l \cdot |D| \cdot 2 \cdot (\lceil \log_B |D| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$  [76]
  - $C_{tr} = c_t \cdot |D|$
  - $C_{write} = c_r \cdot |O|$
  - $C_{pre} = C' + c_r \cdot m \cdot \phi \cdot t$
- $C' = \begin{cases} C_{read} + (c_l + c_t) \cdot m \cdot \phi \cdot mp + a, & \text{for the intersection filter approaches} \\ c_r \cdot |L| + (c_l + c_t) \cdot m \cdot \phi \cdot mp_2, & \text{for the Bloomjoin} \end{cases}$
- $a = c_r \cdot m \cdot \phi \cdot t$  for the first intersection filter approach, otherwise  $a = 0$
- $C_{pre} = 0$  for approaches without using the filters. Besides, assume that the filters are the same size  $m$ . If  $m$  is small, we will not compress the filter files, and so  $\phi = 1$ .

In equation (3.6), an additional cost  $C_{pre}$  should be added to the cost model in [76]. Intuitively, we can see that  $|D|$ , the size of the intermediate data, decides the total cost of the join operation. Thus, we should focus on analyzing this parameter for our different approaches in order to have a more complete assessment.

#### 3.3.3.2 Cost comparison of approaches

From the pros and cons of the join algorithms mentioned in Sections 2.2 and 2.3 of Chapter 2, we consider three prominent join algorithms such as the Reduce-side join, the Bloomjoin, and our intersection filter-based join with the three filter approaches. These algorithms implement a general join model and have no restrictions on input datasets. More importantly, the Bloomjoin and the intersection filter-based join are good algorithms for optimizing the join performance. They are more efficient than the semi-join because they execute lesser jobs, and use a smaller data structure that is a bit vector for minimizing the amount of data transferred over the network. Besides, these algorithms do not have to distribute the filtered input dataset that can be large to all compute nodes.

In order to estimate  $|D|$ , it is assumed that  $\partial_L$  is the ratio of the joined records of  $R$  with  $L$ , and  $\partial_R$  is the ratio of the joined records of  $L$  with  $R$ . The size of intermediate data with the false intersection probability is:

$$|D| = \begin{cases} \partial_L|R| + f_{\cap pair(L)} \cdot (1 - \partial_L)|R| + \partial_R|L| + f_{\cap pair(R)} \cdot (1 - \partial_R)|L| & (3.7) \\ \partial_L|R| + f_{\cap BF} \cdot (1 - \partial_L)|R| + \partial_R|L| + f_{\cap BF} \cdot (1 - \partial_R)|L| & (3.8) \\ \partial_L|R| + f_{\cap PBF} \cdot (1 - \partial_L)|R| + \partial_R|L| + f_{\cap PBF} \cdot (1 - \partial_R)|L| & (3.9) \\ \partial_L|R| + f_{\cap pair(L)} \cdot (1 - \partial_L)|R| + |L| & (3.10) \\ |R| + |L| & (3.11) \end{cases}$$

where

equation (3.7) for the IBF-based join with the pair of the filters (approach 1),

equation (3.8) for the IBF-based join with the unpartitioned IBF (approach 2),

equation (3.9) for the IBF-based join with the partitioned IBF (approach 3),

equation (3.10) for the Bloomjoin with one filter  $BF(L.uid)$ ,

equation (3.11) for the Reduce-side join,

and  $f_{\cap pair(L)}$ ,  $f_{\cap pair(R)}$ ,  $f_{\cap BF}$  and  $f_{\cap PBF}$  refer to Section 3.2.4.

From the equation of the intermediate data size  $|D|$  above, we can point out the following important evaluation.

**Theorem 3.5.** *The join operation using the intersection filter is more efficient than using a basic Bloom filter because it produces less redundant and intermediate data than the latter. Additionally, we can derive comparing equation for  $|D|$ :*

$$|D|_{3.7} \approx |D|_{3.8} < |D|_{3.9} < |D|_{3.10} < |D|_{3.11} \quad (3.12)$$

where  $|D|_i$  is the intermediate data size for equation  $i^{th}$  ( $i = 3.7, \dots, 3.11$ ).

**Proof.** We can see that querying tuples of  $R$  into  $BF(L.uid)$  and tuples of  $L$  into  $BF(R.uid)$  corresponds to finding common elements between the filters. It is also the intersection operation of filters as presented in Section 3.2.2. Thus the intermediate data generated by the first two approaches is equivalent  $|D|_{3.7} \approx |D|_{3.8}$ . (3.13)

From Theorem 3.4, we get  $0 < f_{\cap BF} < f_{\cap PBF} \ll 1$ . So we can deduce:

$$\partial_R|L| + f_{\cap BF} \cdot (1 - \partial_R)|L| < \partial_R|L| + f_{\cap PBF} \cdot (1 - \partial_R)|L| < |L| \quad (3.14)$$

and

$$\partial_L|R| + f_{\cap BF} \cdot (1 - \partial_L)|R| < \partial_L|R| + f_{\cap PBF} \cdot (1 - \partial_L)|R| < |R| \quad (3.15)$$

Additionally, we also have  $0 < f_{\cap PBF} < f_{\cap pair(L)} \ll 1$ . Thus it simply shows:

$$\partial_L|R| + f_{\cap PBF} \cdot (1 - \partial_L)|R| < \partial_L|R| + f_{\cap pair(L)} \cdot (1 - \partial_L)|R| < |R| \quad (3.16)$$

Combining inequalities (3.13), (3.14), (3.15), and (3.16) into equations (3.7), (3.8), (3.9), (3.10), and (3.11), Theorem 3.5 is proved  $\square$ .

### 3.3 Optimization for two-way joins using intersection filters in MapReduce

From equations (3.6) and (3.12), we can evaluate the total cost of the join operation for the different approaches by the following theorem.

**Theorem 3.6.** *The join operation using the intersection filter has the lowest cost. In addition, we can derive comparing equation for the total cost of the algorithms:*

$$C_{3.7} \approx C_{3.8} < C_{3.9} < C_{3.10} < C_{3.11} \quad (3.17)$$

where  $C_i$  is the total cost in case of equation  $i^{th}$  ( $i = 3.7, \dots, 3.11$ ).

It should be noted the total cost to perform the pre-processing job

$$C_{pre} = \begin{cases} C_{read} + (c_l + c_t) \cdot m \cdot \phi \cdot mp + 2 \cdot c_r \cdot m \cdot \phi \cdot t; & \text{in case of (3.7)} \\ C_{read} + (c_l + c_t) \cdot m \cdot \phi \cdot mp + c_r \cdot m \cdot \phi \cdot t; & \text{in cases of (3.8) and (3.9)} \\ c_r \cdot |L| + (c_l + c_t) \cdot m \cdot \phi \cdot mp_2 + c_r \cdot m \cdot \phi \cdot t; & \text{in case (3.10)} \\ 0; & \text{in case of (3.11)} \end{cases}$$

For the data locality optimization, the MapReduce framework runs the map task on a node where the input data resides in DFS and the data is directly fetched. Thus the read cost of this phase is low. As a result, the total cost  $C_{pre}$  is negligible compared to the creation and transfer of redundant data over the network.

However, the join algorithm using the different intersection filters will become inefficient when there is a large number of map tasks ( $mp$ ), and very little redundant data in the join operation. In the case of so many map tasks, a tasktracker running multiple map tasks will merge the local filters of each task and will maintain only two local filters  $BF(R)$  and  $BF(L)$ . This is not difficult to be solved in our future work. In the case of little redundant data, we will not need to use the filter as well as the pre-processing job. For this reason, we should estimate the threshold of redundant data so that the cost of the pre-processing job is less than the cost associated with redundant data and thus the intersection filter becomes more useful.

Let  $|D^*|$  be the size of redundant data or removed data,  $C_{sort}^*$  be the total cost to perform the sorting and copying redundant data at the map and reduce nodes, and  $C_{tr}^*$  be the total cost to transfer redundant data among the nodes. Accordingly, the cost associated with redundant data is the sum of  $C_{sort}^*$  and  $C_{tr}^*$ . We show the threshold of the size of redundant data that the join optimization should use the intersection filter as follows:

$$C_{pre} < C_{sort}^* + C_{tr}^*$$

where

- $|D^*| = |R| + |L| - |D|$ ,
- $C_{tr}^* = c_t \cdot |D^*|$ ,
- $C_{sort}^* = c_l \cdot |D^*| \cdot 2 \cdot (\lceil \log_B |D^*| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil) [76]$

Based on the size of intermediate data  $|D|$ , the threshold depends on  $\partial_L$  (the ratio of the joined records of  $R$  with  $L$ ) and  $\partial_R$  (the ratio of the joined records of  $L$  with  $R$ ).

### 3.4 Optimization for multi-way joins using intersection filters in MapReduce

Multi-way join algorithms are still an open issue and their existing solutions from traditional distributed and parallel databases cannot be easily extended to adapt a shared-nothing distributed computing paradigm as MapReduce. In this section, therefore, we propose several intersection filter-based approaches to computing important multi-way joins as the query  $Q_2$ .

#### 3.4.1 Extended intersection filter

In order to evaluate multi-way joins, we introduce an *extended intersection filter* (EIF) as in Figure 3.9.

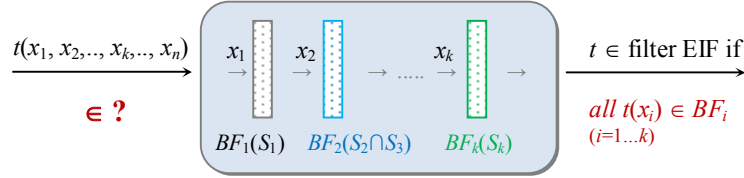


Figure 3.9: Extended intersection filter -  $EIF(BF_1, BF_2, \dots, BF_k)$

The EIF includes an array of standard Bloom and intersection filters hashed on different join key columns  $x_1, x_2, \dots, x_k$ . Each tuple  $t(x_1, x_2, \dots, x_k, x_{k+1}, \dots, x_n)$  may contain  $k$  join keys linking to others. The extended intersection filter accepts an input and returns a "yes" or "no" answer. If one of the join keys of the tuple  $t$ ,  $t(x_i)$ , is not a member of a component filter  $BF_i$  of the EIF, the output is "no" answer. Otherwise, the output of the EIF is "yes" answer, i.e., every  $t(x_i)$  is in the component filter  $BF_i$  of the EIF.

#### 3.4.2 Three-way join using intersection filter

In this section, we consider a different join way including three input datasets at once instead of two datasets as the two-way join. This operation is called three-way join and represented under the form of

$$R \bowtie_{uid=uid1} K \bowtie_{uid2=uid} L$$

It corresponds to the query  $Q_2$  in Chapter 2, which is a simple kind of multi-joins. There are several ways to compute the three-way join as follows.

$$\begin{aligned} & R \bowtie_{uid=uid1} K \bowtie_{uid2=uid} L \\ &= (R \bowtie_{uid=uid1} K) \bowtie_{uid2=uid} L \\ &= R \bowtie_{uid=uid1} (K \bowtie_{uid2=uid} L) \end{aligned}$$

The execution plans show that we can implement the three-way join by a sequence of 2 two-way joins mentioned earlier. The first way consists in joining two datasets  $R$  and  $K$ , and then joins the intermediate output with  $L$ . Another way performs joining two datasets  $K$  and  $L$ , and then joins  $R$  with the intermediate output.

Both the two ways must use at least two MapReduce jobs to execute the three-way join operation. However, there is an alternative way which joins all the three datasets at once in a single MapReduce job. The algorithm is proposed by Afrati and Ullman [6] for optimizing multi-way joins. It begins with the idea of a matrix of reduce processes (reducers) as shown in Figure 3.10.

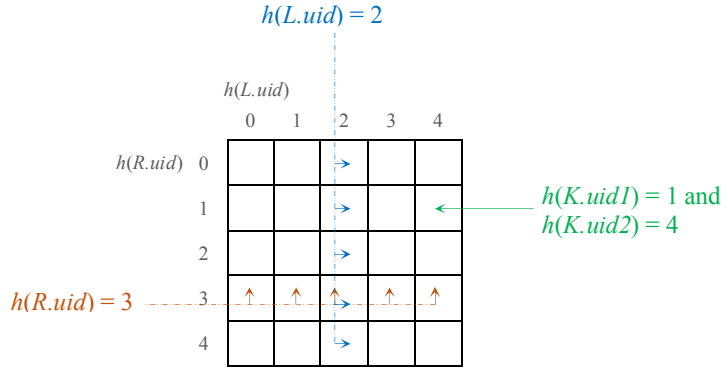


Figure 3.10: Distributing tuples of  $R$ ,  $K$ , and  $L$  among  $r = m^2$  reducers

Given the number of reducers  $r = m^2$  forming a reducer matrix  $m \times m$ , and a hash function  $h$  generating a random number within range  $0, 1, 2, \dots, m - 1$ . Each reducer is associated with a cell  $(i, j)$  in the reducer matrix, where  $i$  and  $j$  are integers within the range of  $m - 1$ . Namely, a cell  $(3, 2)$  associates with the reducer  $(i * m + j) = 17$ .

To compute  $R(\text{uname}, \text{uid}) \bowtie K(\text{uid1}, \text{uid2}) \bowtie L(\text{uid}, \text{event})$  in a MapReduce job, the mappers distribute tuples of  $R$ ,  $K$ , and  $L$  to the reducer matrix as follows. The mappers send each tuple of  $K$  to only one reducer, while each tuple of  $R$  and  $L$  are sent to many different reducers. Specifically, each tuple of  $K(\text{uid1}, \text{uid2})$  is sent to the reducer numbered  $(h(K.\text{uid1}), h(K.\text{uid2}))$ . Each tuple  $R(\text{uname}, \text{uid})$  is sent to all the reducers numbered  $(h(R.\text{uid}), x)$ , for any  $x$ . Each tuple  $L(\text{uid}, \text{event})$  is sent to all the reducers numbered  $(y, h(L.\text{uid}))$ , for any  $y$ .

As illustrated in Figure 3.10, we have a reducer matrix  $5 \times 5$  with 25 reducers ( $m=5$ ). A tuple of  $R(\text{uname}, \text{uid})$  with  $h(R.\text{uid}) = 3$  is sent to all the reducers 15 to 19 (numbered  $(3, x)$ ). A tuple of  $L(\text{uid}, \text{event})$  with  $h(L.\text{uid}) = 2$  is sent to all the reducers 2, 7, 12, 17, and 22 (numbered  $(y, 2)$ ). A tuple of  $K(\text{uid1}, \text{uid2})$  with  $h(K.\text{uid1}) = 1$  and  $h(K.\text{uid2}) = 4$  is sent to only one the reducers 9 (numbered  $(1, 4)$ ). In the example, the output of joining these tuples is empty. If there is a tuple of  $K(\text{uid1}, \text{uid2})$  with  $h(K.\text{uid1}) = 3$  and  $h(K.\text{uid2}) = 2$  is sent to the reducer 17, we get a result of joining the tuples. Another example, we can easily see that if there are three tuples  $R(\text{'Laurent dOrazio'}, \text{'b'})$ ,  $K(\text{'b'}, \text{'c'})$ , and  $L(\text{'c'}, \text{'login'})$ , they will all be sent to the reducer numbered  $(h(\text{'b'}), h(\text{'c'}))$  and then the reducer computes the join of these tuples correctly.

We use the method of Lagrangian multipliers in order to present how to choose the parameter  $r$  for minimizing the communication cost. For simplicity, it is assumed  $|R|=|K|=|L|$ . The total communication cost for the optimal three-way join is  $O(|R|\sqrt{r})$  and the total communication cost for the cascade of 2 two-way joins is  $O(|R|^2 \cdot \alpha)$ , where  $\alpha$  is the probability of two tuples from different datasets agreeing on their common column. This analysis shows that the three-way join is better than the cascade of the two-way joins when  $r < (|R| \cdot \alpha)^2$ , and becomes a good choice.

However, the use of the reducer matrix for distributing tuples of the input datasets leads to tuple duplications. For each tuple of the dataset  $R$  or  $L$ , a mapper generates  $m$  duplicates of the tuple because the mapper cannot ensure the join key of a tuple in the dataset  $K$ . Consequently, the communication and I/O overheads are large. This situation can be improved significantly if we can discover and remove non-joining tuples of the input datasets  $R$  and  $L$  without replicating them. The improvement is shown in Figure 3.11.

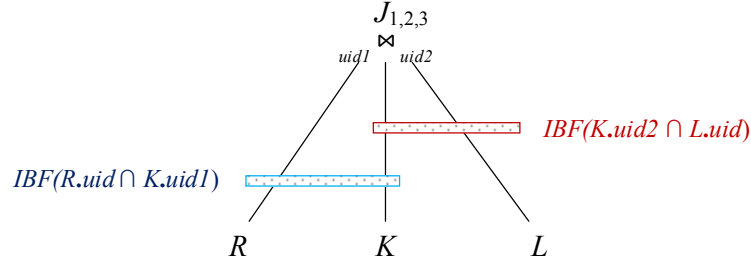


Figure 3.11: Three-way join operation using intersection filter

The input datasets  $R$  and  $L$  are filtered by intersection filters  $IBF(R.uid \cap K.uid1)$  and  $IBF(K.uid2 \cap L.uid)$ , respectively. The input dataset  $K(uid1, uid2)$  is filtered by an extended intersection filter including the two filters  $IBF(R.uid \cap K.uid1)$  and  $IBF(K.uid2 \cap L.uid)$ . In other words, the dataset  $K$  is filtered two times by  $IBF(R.uid \cap K.uid1)$  and  $IBF(K.uid2 \cap L.uid)$  on the distinct join key columns  $uid1$  and  $uid2$ , respectively. Obviously, the solution considerably reduces the amount of data transferred to the reducers.

Similarly to the two-way join, implementing the three-way join with the (extended) intersection filter also uses two MapReduce jobs. However, this implementation needs to be changed to comply with its three inputs. The *pre-processing job* now generates two intersection filters  $IBF_1(R.uid \cap K.uid1)$  and  $IBF_2(K.uid2 \cap L.uid)$  on distinct join key columns. An extended intersection filter  $EIF(IBF_1, IBF_2)$  consists of the two filters  $IBF_1$  and  $IBF_2$ , which is used to filter the dataset  $K$ . The *join job* is executed like the reduce-side join using  $BF_1$ ,  $BF_2$  and  $EIF$  to remove redundant data from its inputs  $R$ ,  $L$  and  $K$ , respectively, in the Map phase. Additionally, map and reduce functions of the join job are modified more complexly than the two-way join job. These are described in the following details.

### Pre-processing job

A join operation with three input datasets  $R$ ,  $K$  and  $L$  is submitted and compiled into two MapReduce jobs, in which a pre-processing job is followed by a join job. The pre-processing job has three groups of parallel map tasks ( $mp_1$ ,  $mp_2$  and  $mp_3$ ) to build local Bloom filters and one reduce task to produce two intersection filters  $IBF_1(R.uid \cap K.uid1)$  and  $IBF_2(K.uid2 \cap L.uid)$ . The  $mp_1$  processing  $R$  creates local Bloom filters  $BF(R.uid)$ , the  $mp_2$  handling  $K$  produces local filters  $BF(K.uid1)$  and  $BF(K.uid2)$ , while the  $mp_3$  processes  $L$  to generate local filters  $BF(L.uid)$ . All the local filters are then sent to the reducer. The reducer merges the corresponding local filters to generate four global filters  $BF(R.uid)$ ,  $BF(K.uid1)$ ,  $BF(K.uid2)$  and  $BF(L.uid)$ . Based on the proposals for the intersection filter, the reducer calculates on the global filters and generates two intersection filters  $IBF_1(R.uid \cap K.uid1)$  and



$IBF_2(K.uid2 \cap L.uid)$ . Then, the two intersection filters  $IBF_1$  and  $IBF_2$  will be saved to DFS.

It should be noted that the join operation will return the empty output immediately without executing the join job if one of the intersection filters  $IBF_1$  and  $IBF_2$  is empty. This feature is necessary to compute multi-way joins.

#### Join job

The job begins with distributing the two intersection filters  $BF_1$  and  $BF_2$  to all tasktrackers. Next, the jobtracker will create  $mp_1$ ,  $mp_2$  and  $mp_3$  map tasks for inputs  $R$ ,  $K$  and  $L$  respectively,  $r$  reduce tasks and assign each split to one map task. Two phases to implement this job are described by the following phases.

- *Map phase with filtering*: the mapper reads each tuple from its split and calls a map function to process. The map function queries the join key of the tuple into the corresponding filter. Specifically, the tuple of  $R$  or  $L$  is queried into  $IBF_1$  or  $IBF_2$  on the join key column  $uid$ , respectively. The tuple of  $K$  is queried into  $EIF(IBF_1, IBF_2)$  on the join key columns  $uid1$  and  $uid2$ . If the tuple is not present in the filter, it is eliminated. Otherwise, the tuple is replicated into tagged pair(s)  $((uid, tag), tuple)$  that are then sent to the reducers. The tuple replication is executed as shown in Figure 3.10.
- *Reduce phase*: the reduce function takes its input and does a full cross-product of tuples from the different input datasets for each join key pair of  $K$  to create the joined output. The reducer buffers the tuples of  $R$  and  $L$ , and performs the cross product of  $R$ ,  $L$  and  $K$  for each incoming tuple of  $K$ . It is completed by writing the output to DFS.

The following pseudo code presents a three-way join algorithm using the intersection filter.

---

#### Algorithm 2 - Three-way join algorithm using Intersection filter

---

**Job1\_3Way**: builds two intersection filters  $IBF(R.uid \cap K.uid1)$  and  $IBF(K.uid2 \cap L.uid)$

```

Init_Map() // init function for map phase
    bfilter_R ← empty; //storing keys R.uid
    bfilter_K1 ← empty; //storing keys K.uid1
    bfilter_K2 ← empty; //storing keys K.uid2
    bfilter_L ← empty; //storing keys L.uid
    tag = null; //storing name of input dataset
    localFilterPointer ← null; // a pointer

Map(k: null, v: a tuple from an R, K or L split)
    if (tag == null) then
        tag = name of input dataset 'R', 'K', or 'L';
        switch (tag)
            case 'R': localfilterPointer = &bfilter_R;
            case 'K': localfilterPointer = &bfilter_K1;
            case 'L': localfilterPointer = &bfilter_L;
        endswitch
    endif

    joinKey ← extract the join column from v

```

---

---

```

add joinKey to localfilterPointer;
if (tag == 'K') then
    joinKey2 ← extract the join column uid2 from v
    add joinKey2 to bfilter_K2
endif

```

```

Close_Map() // close function for map phase
emit(tag, localfilterPointer);
if (tag == 'K') then
    emit('K2', bfilter_K2);
endif

```

```

Init_Reduce() // init function for reduce phase
globalBF_R ← empty; //merging local filters BF(R.uid)
globalBF_K1 ← empty; //merging local filters BF(K.uid1)
globalBF_K2 ← empty; //merging local filters BF(K.uid2)
globalBF_L ← empty; //merging local filters of BF(L.uid)

```

```

Reduce(k: 'R', 'K', 'K2' or 'L', v: a list of local bloom filters)
globalFilterPointer ← null; // a pointer
switch (k)
    case 'R': globalFilterPointer = &globalBF_R;
    case 'K': globalFilterPointer = &globalBF_K1;
    case 'K2': globalFilterPointer = &globalBF_K2;
    case 'L': globalFilterPointer = &globalBF_L;
endswitch

for each bfilter in v' do
    OR(bfilter, globalFilterPointer);
emit(null, null);

```

```

Close_Reduce() // close function for reduce phase
IBF_R_K ← globalBF_R; //intersection filter IBF(R.uid ∩ K.uid1)
IBF_K_L ← globalBF_L; //intersection filter IBF(K.uid2 ∩ L.uid)
AND(globalBF_K1, IBF_R_K);
AND(globalBF_K2, IBF_K_L);
save IBF_R_K and IBF_K_L into two files IBF_R_K and IBF_K_L on DFS

```

**Job2\_3Way:** filters out non-joining tuples in *R*, *K* and *L*, and joins filtered datasets *R'*, *K'* and *L'*

```

Init_Map() // init function for map phase
IBF_R_K ← load(IBF_R_K); //loading intersection filter IBF(R.uid ∩ K.uid1)
IBF_K_L ← load(IBF_K_L); //loading intersection filter IBF(K.uid2 ∩ L.uid)
reducerMatrixSize ← sqrt(the number of reducers);

```

```

Map(k: null, v: a tuple from an R or L split)
tag ← 1, 2, or 3 corresponding to name of R, L or K;
key ← extract the join key from v; //R.uid, L.uid or K.uid1
p = h(key) mod reducerMatrixSize;
if (tag == 1 && key in IBF_R_K) then
    //sending v(uname, uid) of R to reducers: (h(uid), j)
    for (j=0; j<reducerMatrixSize; j++) do
        partition = p*reducerMatrixSize + j;
        emit(pair(key, tag:partition), v);
    else
        if (tag == 2 && key in IBF_K_L) then
            //sending v(uid,event) of L to reducers: (i, h(uid))
            for (i=0; i<reducerMatrixSize; i++) do
                partition = i*reducerMatrixSize + p;

```

---

---

```

        emit(pair(key, tag:partition), v);
    else
        key2 = ← extract the join key uid2 from v; // K.uid2
        if (tag == 3 && key in IBF_R_K && key2 in IBF_K_L) then
            //sending v(uid1, uid2) of K to reducer (h(uid1), h(uid2))
            col = h(key2) mod reducerMatrixSize;
            partition = p*reducerMatrixSize + col;
            emit(pair(key, tag:partition), v);
        endif
    endif
endif

GroupComparator(taggedKey1: taggedkey, taggedKey2: taggedkey)
    res = compare(taggedKey1.tag, taggedKey2.tag);
    if (res == 0) then
        res = compare(taggedKey1.key, taggedKey2.key);
    endif
    return res;

Partitioner(k': taggedkey, v: value, p: the number of reducers)
    return k'.tag.getPart();

Init_Reduce() // init function for reduce phase
    multiMap_R ← empty; //storing <key, values> of R
    multiMap_L ← empty; //storing <key, values> of L

Reduce(k': taggedKey, v': list of values v with key k')
    if (k'.tag == 0) then
        add (k'.key, v') into multiMap_R
    else
        if (k'.tag == 2) then
            add (k'.key, v') into multiMap_L
        else
            //for tuples of K
            if (k'.key in multiMap_R) then
                for each k in v' do
                    if (k.uid2 in multiMap_L) then
                        for each r in multiMap_R[k'.key] do
                            for each l in multiMap_L[k.uid2] do
                                emit(r, pair(k, l));
                            endif
                        endif
                    endif
                endif
            endif
        endif
    endif
endif

```

---

Listing 3.2. Pseudo code for three-way join algorithm using intersection filter

### 3.4.3 Chain join using intersection filter

We consider a *chain join*, which is a cascading join of relations so that each relation is linked to the following one by a single or multiple attributes. This join case has the form of  $R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie \dots \bowtie R_n(x_n, x_{n+1})$ , and is shown by:

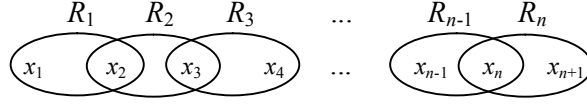


Figure 3.12: A chain join

The query  $Q_2$  in Chapter 2 is an illustration of the chain join. We begin with an implementation of the chain join using a cascade of Bloomjoins in MapReduce. It is an iterative implementation of two-way Bloomjoins as presented in Figure 3.13.

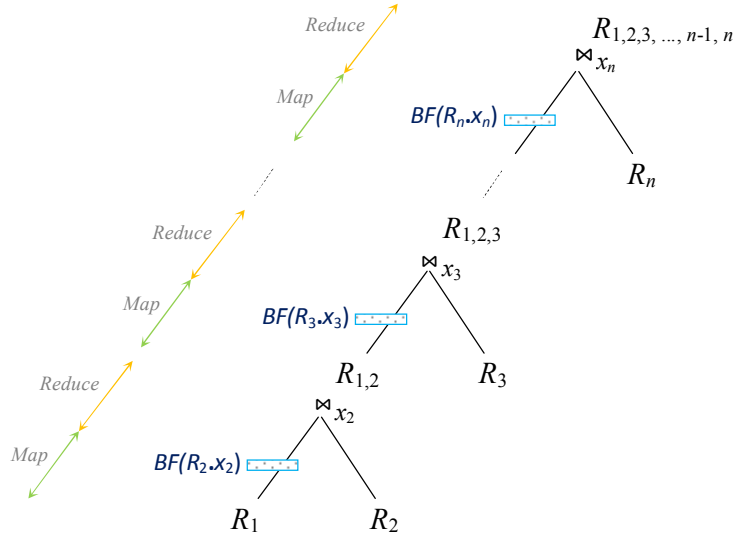


Figure 3.13: Implementation of a chain join using a Bloomjoin cascade

The implementation of the two-way Bloomjoin is earlier mentioned in Section 2.3.2. The chain join operation includes multiple Bloomjoin jobs for joining datasets, two datasets at a time. Considering  $n$  datasets  $R_1 \dots R_n$ ,  $R_1$  is joined with  $R_2$  on the key  $x_2$  as one job. The result of this join,  $R_{1,2}$ , is joined with  $R_3$  and so on.

In the cascade of the Bloomjoins, we can see that the dataset  $R_1$  and intermediate join results  $R_{1,2}$ ,  $R_{1,2,3}$ , ...,  $R_{1,2,\dots,n-1}$ , are filtered by  $BF(R_2.x_2)$ ,  $BF(R_3.x_3)$ ,  $BF(R_4.x_4)$ , ...,  $BF(R_n.x_n)$ , respectively. Meanwhile, the input datasets  $R_2$ ,  $R_3$ , ...,  $R_n$  are not filtered and thus there remain a lot of non-joining data transferred over the network. This situation will be considerably improved by using intersection filters as follows.

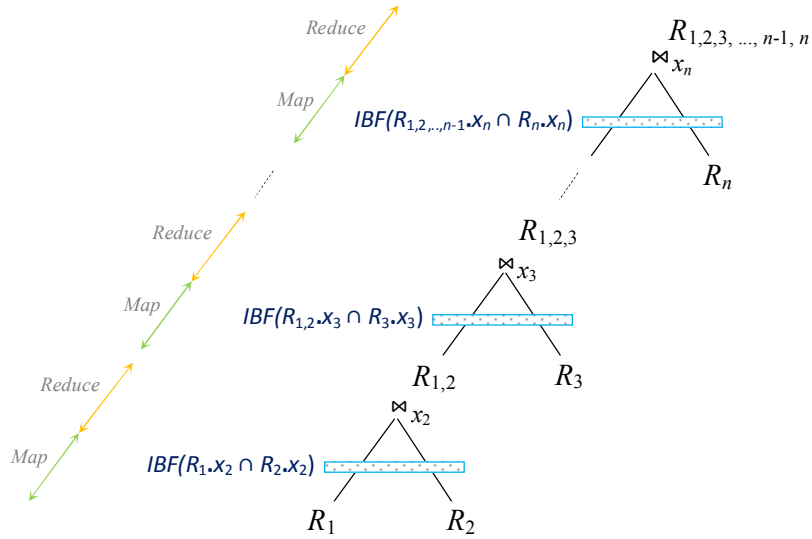


Figure 3.14: Implementation of a chain join using a cascade of two-way joins using intersection filters

The intersection filter  $IBF(R_i.a \cap R_j.b)$  is formed from two basic filters  $BF(R_i.a)$  and  $BF(R_j.b)$  with using the approaches in Section 3.2. It is an approximate representation of the set intersection  $R_i \cap R_j$ .

Figure 3.14 illustrates the implementation of the chain join as a cascade of two-way joins using intersection filters. All the input datasets and the intermediate join results are filtered by their corresponding intersection filters. For instance, the intersection filter  $IBF(R_{1,2}.x_3 \cap R_3.x_3)$  is used to eliminate most of non-joining data in both the datasets  $R_{1,2}$  and  $R_3$ . Based on Theorem 3.5, it is easy to deduce that intermediate data sent to the reducers in the case of the intersection filter-based join cascade is less than in case of the Bloomjoin cascade.

For all the above implementations, however, the intermediate join results  $R_{1,2}$ ,  $R_{1,2,3}$ , ...,  $R_{1,2,\dots,n-1}$  still contains redundant tuples passed to the next join. This is because the join processing  $i$  generates result tuples that some of them do not participate the next join processing  $i+1$ . We therefore discover two improvements for chain joins using extended intersection filters as follows.

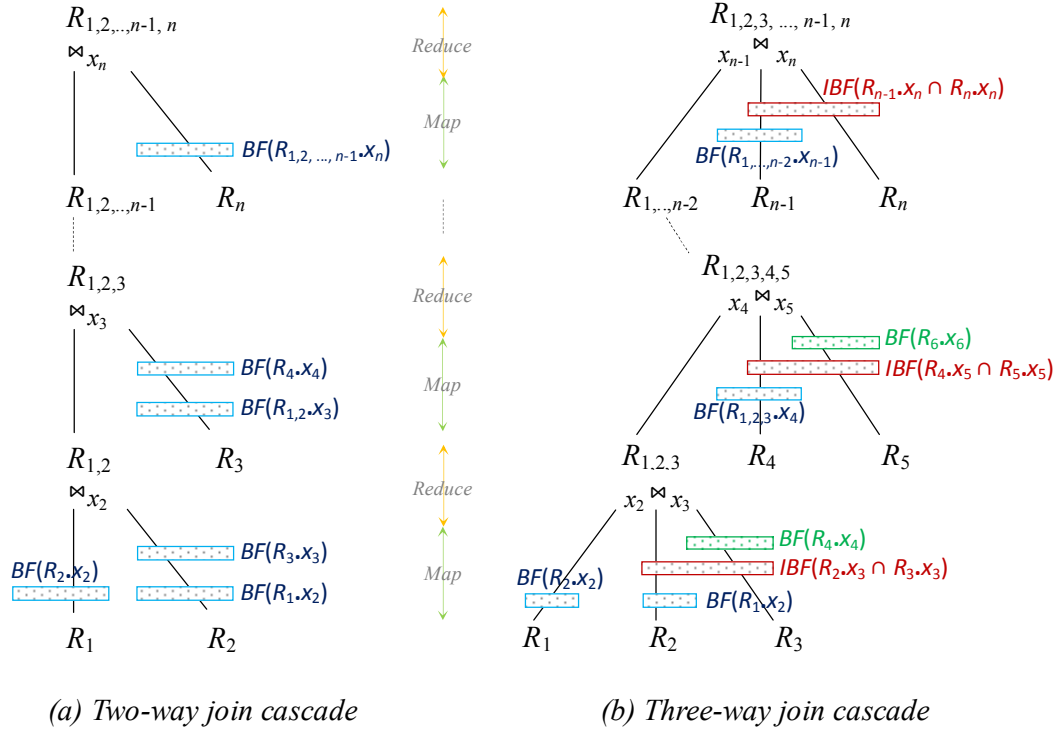


Figure 3.15: Optimization of a chain join using extended intersection filters

In the first solution as Figure 3.15 (a), the chain join is implemented by a two-way join cascade. We have to filter out redundant data from an intermediate join result. Instead, we should move this filtering operation into the previous join job. Hence, the input datasets  $R_2, \dots, R_n$  are filtered by extended intersection filters  $EIF$ . The extended filter  $EIF_i$  includes a Bloom filter  $BF(R_1, \dots, i-1.x_i)$  built from the intermediate join result and a filter  $BF(R_{i+1}.x_{i+1})$  from the next input dataset. Particularly,  $EIF_2$  contains  $BF(R_1.x_2)$  and  $BF(R_3.x_3)$ . Besides,  $R_1$  and  $R_n$  are filtered by standard Bloom filters  $BF(R_2.x_2)$  and  $BF(R_{1,2,\dots,n-1}.x_n)$ , respectively. Obviously, we now do not need to perform any extra filtering operations for the intermediate join results. In other words, the intermediate results generated by the two-way joins of the chain join only contain actual joining data and can be sent to the next join without filtering. This is an important special characteristic while other solutions need to use the complement filters to check the intermediate join results.

To execute this solution, we first use a pre-processing job to build the Bloom filters  $BF(R_i.x_i)$  from the input datasets  $R_i$  ( $i = 2, \dots, n$ ), and  $BF(R_1.x_2)$  from  $R_1$ . Next, we implement the chain join as an iteration of one two-way join job with changing the inputs. The first input of the two-way join,  $R_1, \dots, i-1$ , do not need to be filtered, exceptionally for  $R_1$  filtered by  $BF(R_2.x_2)$ . The second input of the join,  $R_i$ , is filtered by the filter  $EIF_i$  that is formed by  $BF(R_1, \dots, i-1.x_i)$  and  $BF(R_{i+1}.x_{i+1})$ . Initially, for  $i = 2$ ,  $BF(R_1, \dots, i-1.x_i)$  is the filter  $BF(R_1.x_2)$ . Then, for  $3 \leq i \leq n$ , the filter  $BF(R_1, \dots, i-1.x_i)$  is generated in the reduce phase of the join processing between  $R_1, \dots, i-2$  and  $R_{i-1}$ . As a result, building the filters from the intermediate join results do not have any

additional overheads. The iteration stops when one of the two input datasets is null. The output of the chain join is the join result  $R_{1, \dots, i}$  that is then written to DFS.

As in Figure 3.15 (b), the second solution suggests that the chain join includes a cascade of the three-way joins using the extended intersection filters. Assume that  $i$  is an even number and greater than 1. The first input dataset of each the three-way join,  $R_{1, \dots, i-1}$ , does not need to be filtered, exceptionally for  $R_1$  filtered by  $BF(R_2.x_2)$ . The second input of the join is filtered by the extended intersection filter  $EIF'_i$  that consists of a filter  $BF(R_{1, \dots, i-1}.x_i)$ , and a filter  $IBF(R_i.x_{i+1} \cap R_{i+1}.x_{i+1})$ . The last input of the join needs the extended filter  $EIF'_i$  to remove redundant data. The  $EIF'_i$  includes a filter  $IBF(R_i.x_{i+1} \cap R_{i+1}.x_{i+1})$  and a filter  $BF(R_{i+2}.x_{i+2})$ .

The execution of the second solution is similar to the first solution. The solution runs a pre-processing job to produce the Bloom filters  $BF(R_j.x_j)$  from the input datasets  $R_j$  ( $j = 2, \dots, n$ ), and  $BF(R_1.x_2)$  from  $R_1$ . Next, the chain join is implemented as an iteration of one three-way join job  $(R_{1, \dots, i-1} \bowtie R_i \bowtie R_{i+1})$  using their corresponding intersection filters. It is noted that each the filter  $BF(R_{1, \dots, i-1}.x_i)$  is generated in the reduce phase of emitting the intermediate join result  $R_{1, \dots, i-1}$ . Besides, the last join may be one two-way join. Initially, for  $i = 2$ , the dataset  $R_{1, \dots, i-1}$  is also the dataset  $R_1$ . The evaluation of the three-way join with the corresponding inputs is repeated until the first input or the second input is null. The final output is the intermediate join result  $R_{1, \dots, i}$  stored on DFS.

The solution (a) is designed to use less memory than the solution (b) because the former only buffers one input for each two-way join, whereas the second one must buffer two inputs for each three-way join. However, the first solution uses more jobs than the second one. Assume that  $n$  is the number of the input datasets, the number of the two-way join jobs of the first solution is  $(n-1)$ , while the second one has  $(n-1)/2$  jobs for the three-way joins.

#### 3.4.4 Star join using intersection filter

We examine a star join including a set of joins in which a fact table (a large central table) is joined with several dimension tables (smaller tables containing descriptions for keys in the fact table). The star join is shown in Figure 3.16.

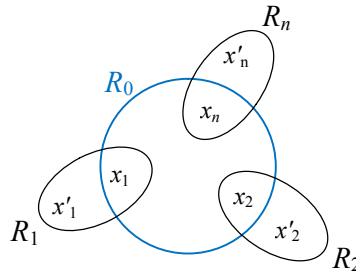


Figure 3.16: A star join

The fact table is a dataset  $R_0$  and the dimension tables are  $R_1, R_2, \dots, R_n$ . The star join query is a popular query in data warehouses that are also a target domain of data-parallel frameworks. Evaluating the star join query in data warehouses is expensive because the fact table participates in every join operation.

The implementation of the star join using the extended intersection filters in MapReduce is suggested by Figure 3.17.

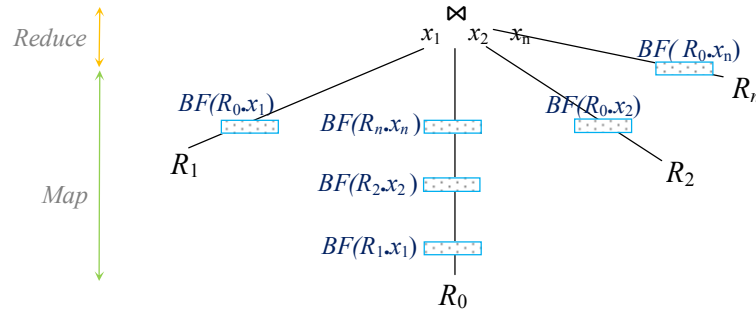


Figure 3.17: Implementation of a star join

We build an extended intersection filter that is an array of  $n$  filters  $EIF_i(R_i.x_i)$ , and  $n$  filters  $BF(R_0.x_i)$ ,  $i=1...n$ . The size of the extended filter is not too big because dimension tables are small. As shown in Figure 3.17, a star join is executed by joining all the datasets in one, in which the large central dataset  $R_0$  is filtered by the extended filter and the other datasets are filtered by the filters  $BF(R_0.x_i)$ , respectively. Consequently, there is no redundant data when the datasets are sent to the join processing. This implementation is more efficient than Bloomjoins because the extended filter can eliminate non-joining tuples from the central dataset at the map phase of one job and it reduces the number of intermediate join jobs to zero.

### 3.4.5 Cost analysis of three-way joins in MapReduce

#### 3.4.5.1 Cost model

The cost model of the three-way join is constructed similarly to the one of the two-way join. We also use the parameters in Table 3.2 for this model. Assume that  $R$ ,  $K$ , and  $L$  are three input datasets. We obtain the total cost of the three-way join as follows:

$$C_{3wJoin} = C_{pre} + C_{read} + C_{sort} + C_{tr} + C_{write} \quad (3.18)$$

where

- $C_{read} = c_r \cdot |R| + c_r \cdot |K| + c_r \cdot |L|$
- $C_{sort} = c_l \cdot |D| \cdot 2 \cdot (\lceil \log_B |D| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil) [76]$
- $mp = mp_1 + mp_2 + mp_3$ , the total number of map tasks for the three inputs
- $C_{tr} = c_t \cdot |D|$
- $C_{write} = c_r \cdot |O|$
- $C_{pre} = C_{read} + (c_l + c_t) \cdot m \cdot \phi \cdot mp + 2 \cdot c_r \cdot m \cdot \phi \cdot t$
- $C_{pre} = 0$  for the approach without using the filters. Besides, assume that the filters are the same size  $m$ .



Because the parameter  $|D|$  (the size of intermediate data) decides the total cost, we should consider it to indicate the efficiency of the three-way join.

### 3.4.5.2 Comparison with cascade of 2 two-way joins

We make a comparison of the intermediate data size between the three-way join and the cascade of 2 two-way joins for evaluating the query  $Q_2$ . It is noted that the size of the intermediate data is also the amount of communication in MapReduce. To simplify the computation, we suppose that the input datasets  $R$ ,  $K$  and  $L$  are the same size.

The implementation model described in Section 3.4.2 shows that the three-way join increases the communication cost because each tuple of  $R$  and  $L$  is sent to many different reducers. However, in compensation, this data replication helps us avoid incremental costs of the two-way join cascade such as incurring an additional job, scanning and shuffling the intermediate join result. Multi-way joins can therefore take the benefits of the three-way join, especially if a typical tuple of one dataset joins with many tuples of another dataset. For instance, we multiply or join copies of the Web matrix.

The optimal three-way join raises two problems that need to be considered. They include choosing the number of reducers and the size of the reducer matrix.

**Theorem 3.7.** *A three-way join  $R(A, B) \bowtie K(B, C) \bowtie L(C, D)$  is more efficient than a cascade of 2 two-way joins  $(R(A, B) \bowtie K(B, C)) \bowtie L(C, D)$  or  $R(A, B) \bowtie (K(B, C) \bowtie L(C, D))$  when  $r < (|R| \cdot \alpha)^2$ . Additionally, the size of the intermediate data is specified by*

$$|D| = \begin{cases} 2 \cdot |R| \cdot \sqrt{r}, & \text{for the optimal three-way join.} \\ |R|^2 \cdot \alpha, & \text{for the cascade of the 2 two-way joins.} \end{cases}$$

where  $r$  is the number of reducers,  $|R| = |K| = |L|$ , and  $\alpha$  is the probability of two tuples from different datasets agreeing on their common column.

**Proof.** First, we consider the three-way join. Two attributes  $B$  and  $C$  of the join query are join key columns. Thus, we use hash functions to map values of  $B$  to  $b$  different buckets, and values of  $C$  to  $c$  buckets, as long as  $b \cdot c = r$ .

The intermediate data size of the three-way join is

$$|R|.c + |K| + |L|.b \tag{3.19}$$

We must find optimal values  $b$  and  $c$  to minimize the above expression subject to the constraint that  $b \cdot c = r$  with  $b$  and  $c$  are positive integers. In this case, the Lagrangian multiplier method is used to present the solution.

Here  $L = |R|.c + |K| + |L|.b - \lambda(b \cdot c - r)$ . We consider the problem

$$\min_{b, c \geq 0} [|R|.c + |K| + |L|.b - \lambda(b \cdot c - r)]$$

We make derivatives of  $L$  with respect to variables  $b$  and  $c$ .

$$\frac{\partial L}{\partial b} = |L| - \lambda \cdot c = 0 \Rightarrow |L| = \lambda \cdot c$$

$$\frac{\partial L}{\partial c} = |R| - \lambda \cdot b = 0 \Rightarrow |R| = \lambda \cdot b$$

We obtain the Lagrangian equations:

$$|L|.b = \lambda.r$$

$$|R|.c = \lambda.r$$

We can multiply these two equations together to get  $|L|.|R| = \lambda^2.r$ . From here, we deduce  $\lambda = \sqrt{|R|.|L|/r}$ . Applying the value of  $\lambda$  into the Lagrangian equations, we get  $b = \sqrt{|R|.r/|L|}$  and  $c = \sqrt{|L|.r/|R|}$

Then, substituting these values in expression (3.19) to be optimized, we get the minimum communication amount of the three-way join:

$$|R|.\sqrt{|L|.r/|R|} + |K| + |L|.\sqrt{|R|.r/|L|} \approx 2.|R|.\sqrt{r}$$

Next, we specify the intermediate data size of the cascade of 2 two-way joins:

$$|R|.|K|. \alpha + |L| \approx |R|^2. \alpha \text{ (where } |R|. \alpha > 1)$$

The cost of the three-way join  $O(|R|\sqrt{r})$  is thus compared with the cost of the two-way join cascade  $O(|R|^2. \alpha)$ . We can conclude that the three-way join will be better the cascade when  $\sqrt{r} < |R|. \alpha$ . In other words, for the optimal three-way join, there is a limit on the number of reducers  $r < (|R|. \alpha)^2$ . The theorem is proved  $\square$ .

We can easily extend Theorem 3.7 for a general three-way join with  $n$  join key columns using an  $n$ -dimensional reducer matrix. For example, a three-way join  $R(A, B) \bowtie K(B, C) \bowtie L(C, A)$  with three join attributes  $A$ ,  $B$ , and  $C$ . This three-way join needs a three-dimensional reducer matrix. The optimal three-way join will become more efficient than the cascade of 2 two-way joins when  $r < (|R|. \alpha)^3$  and its amount of communication is  $3.|R|.\sqrt[3]{r}$ . In fact, choosing the number of reducers is not difficult to satisfy this condition. For example, if  $|R|. \alpha = 15$ , as might be the case for the Web incidence matrix, we can use the number of reducers  $r$  up to 3375.

Similarly, the intermediate data size of the three-way join using the intersection filters is shown by the following theorem.

**Theorem 3.8.** *A three-way join  $R(A, B) \bowtie K(B, C) \bowtie L(C, D)$  is more efficient with the intersection filters than without the intersection filters. Besides, the three-way join using the filters is also more efficient than the two-way join cascade using the filters when  $r < (|R'|. \alpha)^2$ . In the cases of using the intersection filters, the size of the intermediate data is defined by*

$$|D| = \begin{cases} 2.|R'|.\sqrt{r}, & \text{for the optimal three-way join.} \\ |R'|^2. \alpha, & \text{for the cascade of 2 two-way joins.} \end{cases}$$

$$|R'| = \partial. |R| + f_{\cap BF}. (1 - \partial) |R|, \text{ } R' \text{ is the filtered dataset of one input.}$$

where  $r$  is the number of reducers,  $\alpha$  is the probability of two tuples from different datasets agreeing on their common column,  $|R| = |K| = |L|$ ,  $\partial$  is the ratio of the joined records of one input dataset with another, and  $f_{\cap BF}$  is the false intersection probability between the datasets.

**Proof.** We have the following inequalities:

$$0 < \partial < 1 \text{ and } 0 < f_{\cap BF} < 1$$

$$\Rightarrow \partial \cdot |R| + f_{\cap BF} \cdot (1 - \partial) |R| < |R| \Rightarrow |R'| < |R|$$

Combining this equality with Theorem 3.7, we can easily prove Theorem 3.8  $\square$ .

### 3.4.6 Cost analysis of chain joins in MapReduce

#### 3.4.6.1 Cost model

Given a chain join of  $n$  input datasets  $R_1, R_2, \dots, R_n$ . We use the optimized solution (b) to evaluate the chain join as an repetition of one three-way join job with changing inputs,  $\vec{J} = \{J_2, J_4, J_6, \dots, J_{(n-1)/2}\}$ . Initially,  $J_1$  scans  $n$  inputs for building the filters, the initial join result  $R_1, \dots, i-1$  is the dataset  $R_1$ . On each iteration,  $J_i$  performs the join of three inputs including  $R_1, \dots, i-1$ ,  $R_i$ , and  $R_{i+1}$ . The output of the job  $J_i$  is the intermediate join result  $R_1, \dots, i+1$  that becomes the input of the next join job  $J_{i+2}$ . The final output is written to DFS. Based on the cost model of the three-way join, we can extend to compute the total cost of the chain join as follows:

$$C(\vec{J}) = C_{pre} + \sum_{i=2}^{i=i+2 \leq (n-1)/2} C_{distCache} + C_{read}(J_i) + C_{sort}(J_i) + C_{tr}(J_i) + C_{write}(J_i) \quad (3.20)$$

where

- $C_{pre} = \left( \sum_{i=1}^n c_r \cdot |R_i| \right) + (c_l + c_t) \cdot m \cdot \phi \cdot mp$
- $C_{distCache} = 3 \cdot c_r \cdot m \cdot \phi \cdot t$ 
  - $C_{distCache} = 0$  for the approach without using the filters.
- $C_{read}(J_i) = c_r \cdot |R_{1, \dots, i-1}| + c_r \cdot |R_i| + c_r \cdot |R_{i+1}|$
- $C_{sort}(J_i) = c_l \cdot |D_i| \cdot 2 \cdot (\lceil \log_B |D_i| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$ 
  - $mp = mp_1 + mp_2 + mp_3$ , the total number of map tasks for the three inputs
  - $|D_i|$  is the size of the intermediate data in the  $i$ th iteration
- $C_{tr}(J_i) = c_t \cdot |D_i|$
- $C_{write}(J_i) = c_r \cdot |R_{1, \dots, i+1}| + a$ 
  - $a = 2 \cdot c_r \cdot m \cdot \phi \cdot t$ , for building the filter  $BF(R_{1, \dots, i+1})$  in the  $i$ th iteration
- $C_{pre} = 0$  and  $m = 0$  for the approach without using the filters. Besides, assume that the filters are the same size  $m$ .

### 3.4.6.2 Comparison between three-way and two-way join cascades

We can see that the computation of a chain join using the optimized solution (a) can be considered as an iteration of one three-way join job, in which the three-way join job is compiled into 2 two-way join jobs. Therefore, the total cost of the chain join using the solution (a) is determined by the sum of  $C(\vec{J})$  and the extra costs of writing and re-reading the intermediate results of the two-way joins on DFS. In other words, the total cost of the solution (a) is the total cost of the solution (b) added the extra costs of writing and re-reading the intermediate results. The problem arises that we should consider the intermediate data generated by each the solution.

From Theorem 3.8, we can easily show that the three-way join cascade using the intersection filters is more efficient than the two-way join cascade using the filters when  $r < (|R| \cdot \alpha)^2$ . Associating with Theorem 3.6, we deduce that a chain join using the three-way join cascade with the intersection filters becomes a better choice than a chain join using the Bloomjoin cascade and the two-way join cascade.

## 3.5 Experimental evaluation

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

### 3.5.1 Two-way joins

#### 3.5.1.1 Cluster environment and datasets

All experiments were run on a computer cluster of 15 virtual machines using Virtualbox [77]. Each machine has two 2.4Ghz AMD Opteron CPUs with 2MB cache, 10GB RAM and 100GB SATA disks. The operating system is 64-bit Ubuntu server 12.04.2 LTS, and the java version is 1.7.0.21. We installed Hadoop [46] version 1.0.4 on all nodes in which one of the nodes was selected to act as master and ran the namenode and the jobtracker processes; the remaining nodes were the tasktrackers that acted as both storage and CPU. Each tasktracker node was configured to run up to two simultaneous map tasks and two reduce tasks. Some non-default hadoop configuration parameters used to run our experiments. The HDFS block size was set to 128MB, size of read/write buffer was 128KB, heap-size for child jvms of maps/reduces was set to 2048M, and the number of reduce tasks is set to 28.

All test datasets were produced by a data generation script of the Purdue MapReduce Benchmarks Suite [78], called “PUMA” which represents a broad range of MapReduce applications exhibiting application characteristics with high/low computation and high/low shuffle volumes. The maximum number of columns in the datasets is 39 and string length in each column is set 19 characters. The dataset *dataset1* contains the first column as a foreign key that refers to the fifth column of the dataset *dataset2*. Table 3.3 summarizes the various dataset sizes used in our experiments.

Table 3.3: Input datasets used in three tests

Inputs	Test 1		Test 2		Test 3	
	<i>size</i>	<i>records</i>	<i>size</i>	<i>records</i>	<i>size</i>	<i>records</i>
dataset1	15GB	40,259,163	35GB	92,681,333	55GB	145,099,559
dataset2	15GB	40,108,215	35GB	92,524,495	55GB	139,573,823
Total	30GB	80,367,378	70GB	185,205,828	110GB	284,673,382

We used three sets of the test datasets such as Test 1, Test 2, and Test 3. These tests have the different sizes, namely, 30GB, 70GB, and 110GB. Each the test includes the two inputs *dataset1* and *dataset2*. For the test1, *dataset1* and *dataset2* contain 40,259,163 and 40,108,215 records, respectively. All the datasets are saved in the same text file format.

### 3.5.1.2 Experimental protocol

We evaluated our experiments by executing the different algorithms for a join query on the datasets of each the test. The following join query is used.

```
SELECT *
FROM dataset1(c0..c20) d1, dataset2(c0..c20) d2
WHERE      d1.column0 = d2.column5 AND
           d1.ROWNUM <= $number1 AND
           d2.ROWNUM <= $number2
ORDER BY d1.column0
```

The query is executed by changing *\$number1* and *\$number2* to the number of records of the *dataset1* and the *dataset2*, respectively. An output tuple of the experiments *t* is defined by the concatenation of the pair of tuples of the first 21 columns that joined to produce the output. Furthermore, for general joins, we set up a many-to-many relationship between the datasets. Accordingly, a parent tuple in *dataset1* contains several child tuples in *dataset2*, and vice versa.

For comparing the efficiency of the join algorithms, we are especially interested in four main aspects for each the algorithm evaluation. They include the number of intermediate tuples generated (i.e. Map output), the total execution time, the task timeline of the implementation, and large-scale input data (e.g. applying the algorithms to different data amounts).

### 3.5.1.3 Evaluation of approaches

First, it is important to focus on comparing the amount of intermediate data (Map output) listed in Table 3.4. The intermediate data is a decisive factor that affects the total execution time of the two-way join.

Table 3.4: The number of intermediate tuples (Map output)

Join algorithms	30GB. Test 1	70GB. Test 2	110GB. Test 3
Pair-Filters-Join	43,453	106,116	179,091
Intersect-Filter-Join	43,453	106,116	179,091
PartIntersect-Filter-Join	59,986	220,214	357,336
BloomJoin	40,276,915	92,747,151	145,206,430
Reduce-Side-Join	80,320,684	185,098,062	284,510,488

We consider the Reduce-side join where no pre-processing job is done. As shown in Table 3.4, it is the most inefficient solution compared to the other approaches although it only runs a single join job. This is mainly due to its intermediate results containing a large amount of non-joining data. The number of intermediate tuples generated in this case is nearly equal to the number of Map input records (see Table 3.3 and Table 3.4). This slight difference is because some records of *dataset2* do not contain *column5*.

We then consider the Bloomjoin and the intersection filter-based joins where the pre-processing job and the filtering operation are done to improve the join performance. To efficiently execute these algorithms, we specified the size of filters according to the cardinality of the join keys of datasets and chose the largest filter. There is a tradeoff between  $m$  and the probability of a false positive. Hence the probability of a false positive  $f$  is approximated by:

$$f \approx (1 - e^{-k \cdot n / m})^k$$

For a given false positive probability  $f$ , the length of the Bloom filter  $m$  is proportionate to the number of elements being filtered  $n$  as Table 3.5.

Table 3.5: Parameters of filters used in experiments

Tests	$f$	$k$	$n$	$m/n$	$m$ (bit)	$m_k=m/k$ (bit)
Test 1	0.001	7	14866	15	222990	31856
Test 2	0.0001	8	15790	21	331590	41449
Test 3	0.0001	8	15790	21	331590	41449

where  $m/n$  is the number of bits allocated for each join key and  $m_k$  is the size of a partition of the partitioned filter.

### 3.5 Experimental evaluation

We can determine optimal parameters for the filter (e.g.  $f$ ,  $k$  and  $m$ ). In practice, however, we should choose values less than optimal value to reduce computational overhead. As shown in Table 3.5, we deliberately select various values of  $f$ ,  $k$  and  $m/n$  for the experiments to consider if they might affect our join performance. In addition, the filter files generated in the tests are compressed in the gzip format.

For the Bloomjoin, the number of intermediate tuples is considerably reduced and so it is better than the Reduce-side join. However, when we compare the amount intermediate data of the Bloomjoin to the intersection filter-based join in each the test (see in Table 3.4), it still produced much more redundant data because the filtering operation is only executed on one input dataset (*dataset1*). This situation is overcome by the intersection filter which has the ability to filter out redundant data from both the input datasets.

Although the intersection filter-based joins have the additional cost for the pre-processing job, they are still the most efficient solutions because most unnecessary data has been removed from both the inputs. As a result, the amount of intermediate data is very small compared to the Bloomjoin (see in Table 3.4).

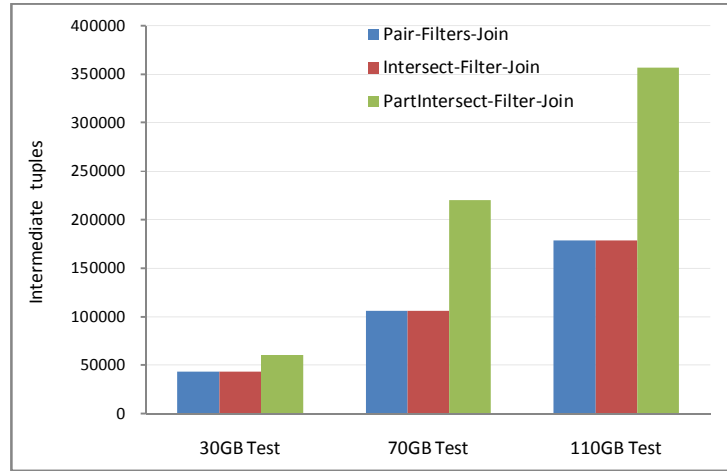


Figure 3.18: Comparison of Map output among the intersection filter-based joins

It is noted that the joins using the pair of filters and the intersection filter (known as our approach 1 and 2) generate the same amount of intermediate data. This data amount is smaller than the amount of intermediate data of the join using the partitioned intersection filter (the approach 3). These arguments have been verified by our experiments and presented in Figure 3.18. The results show that the performance of the intersection filter proposed by the approach 1 or 2 is better than the approach 3.

Next, we evaluate the efficiency of these join algorithms by comparing their total execution time. Generally, the join algorithms generating lesser intermediate data are executed faster.

Table 3.6: Execution of pre-processing job and join job

Join algorithms	30GB. Test 1		70GB. Test 2		110GB. Test 3	
	Pre-processing job time(min)	Join job time(min)	Pre-processing job time(min)	Join job time(min)	Pre-processing job time(min)	Join job time(min)
Pair-Filters-Join	3.08	6.32	6.45	24.67	11.22	94.67
Intersect-Filter-Join	3.17	6.15	6.45	24.25	10.00	92.12
PartIntersect-Filter-Join	3.40	6.95	7.28	24.65	11.50	95.70
BloomJoin	2.12	17.07	3.63	43.63	5.22	139.58
Reduce-Side-Join	0	28.25	0	70.13	0	150.00

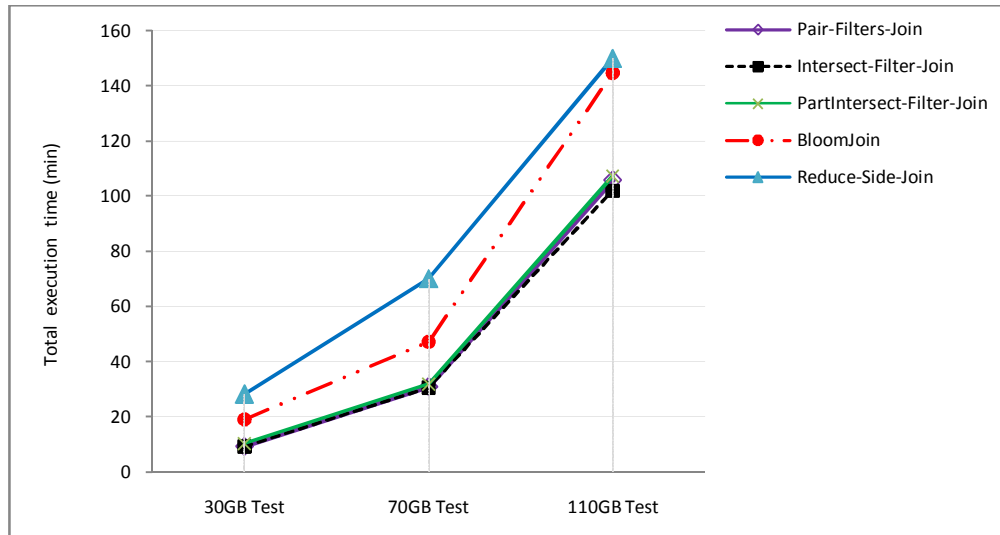


Figure 3.19: Total execution time

Table 3.6 identifies in detail the execution time of the pre-processing job and the join job for the join algorithms. The execution time of the pre-processing job for the IF-based joins is greater than the time for the Bloomjoin and the Reduce-side join because the IF-based joins have to scan two input datasets for building the intersection filter. In contrast, the execution time of the join job for the IF-based joins is much less than the others because they filter out redundant data in both the input datasets.

Figure 3.19 demonstrates that the best execution is the join using the intersection filter. Its total execution time is significantly reduced compared to the Bloomjoin even if the execution time of its pre-processing job is greater. The IF-based join using a pair of filters or the unpartitioned intersection filter runs faster than the join using the partitioned intersection filter. This is because the filtering performance of redundant data of the approaches 1 and 2 is better than the one of the approach 3. The worst performance is the standard Reduce-side join because there are much



### 3.5 Experimental evaluation

redundant data generated. All these results have been shown through the experiments of 30GB, 70GB and 110GB inputs.

Finally, we should analyze their task timelines during the execution of the join job as presented in Figure 3.20. This helps us thoroughly evaluate the performance of the join algorithms. We will not refer to the task timelines of the pre-processing job because it is negligible when we run the join query with the large input datasets (see in Table 3.6).

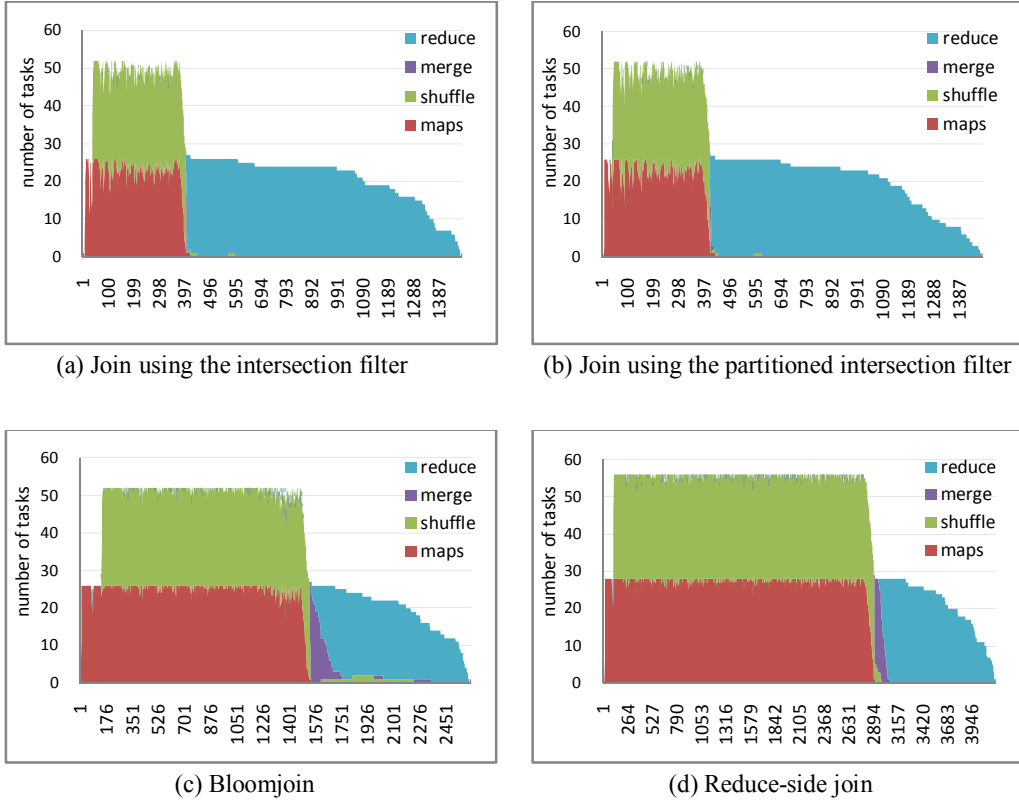


Figure 3.20: 70GB Task timelines during the execution of the join job

Figure 3.20 represents the task timelines of 70GB join jobs using the various algorithms. The task timeline of the join using the pair of filters is omitted because it is quite similar to Figure 3.20 (a). These graphs are created by parsing log files that were generated by Hadoop when we ran the join jobs including 555 map tasks and 28 reduce tasks to process 185,205,828 input records and produce 26,062,967 output records. For each graph, it will start off mostly running map tasks, and by the end, only reduce tasks will be running. The maximum number of simultaneous map or reduce tasks is 28. It may be observed that the peak number of tasks running of the filter-based joins at once is about 52 while the Reduce-side join requires 56.

For the join using the intersection filter as shown in Figure 3.20 (a), the execution time of all map tasks and reduce tasks is significantly reduced versus the Bloomjoin and the Reduce-side join as in Figure 3.20 (b) and Figure 3.20 (c). Besides, the map and reduce phases of the intersection filter-based joins are finished earlier than the Bloomjoin and the Reduce-side join because they have lesser

intermediate data and, as a consequence, the total cost of the local I/O, sort, and remote data copy is also smaller. The joins using the intersection filter are the most efficient solutions because their data filtering efficiency is the best and thus the amount of intermediate data is at least.

However, the two-way join algorithms using the filter(s) are just really efficient when there is a minimum amount of redundant data in the input datasets. The minimum amount, also called threshold, is defined by the two parameters  $\partial_{dataset2}$  and  $\partial_{dataset1}$ . These parameters are the ratios of the joined records between the datasets. We conducted a survey of the ratios of the joined records for the join algorithms with 2GB input; results are shown in Figure 3.21 below.

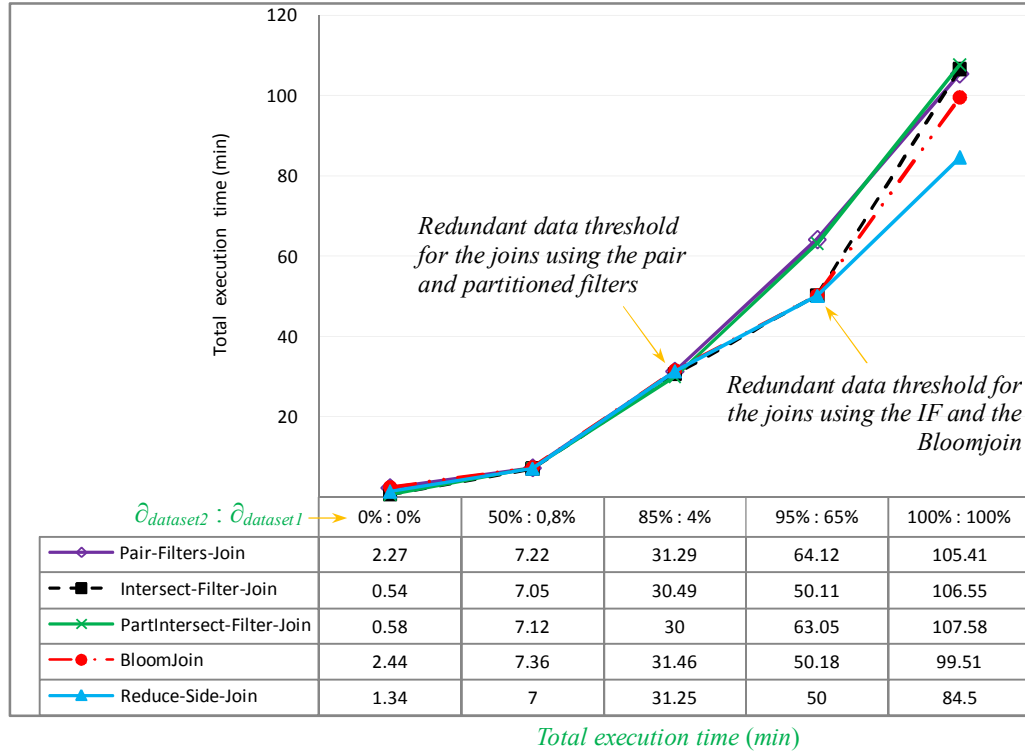


Figure 3.21: Threshold of redundant data amount for the joins with 2GB inputs

In the first case, *dataset1* and *dataset2* are disjoint (i.e. the joined ratios are 0% : 0%). The joins using the intersection filter (exceptionally, the pair of filters) are the best performances because they only run the pre-processing job and discover the empty intersection to omit the join job. Meanwhile, the other joins cannot discover the empty intersection and, as a result, they continue running the join job. Because we use the small input datasets 2GB, the performance of the joins using the filter is not better than the Reduce-side join. This is because they have to incur the additional overhead of the pre-processing job. It means that the filter-based joins should not be used for small input datasets.

We consider two cases of the joined ratios (85% : 4%) and (95% : 65%). There is little redundant data in these cases. Consequently, the total execution time for the joins using the filter increases rapidly than the Reduce-side joins. These are the redundant data thresholds for the filter-based joins. In other words, we should use the

### 3.5 Experimental evaluation

filter-based join algorithms when the amount of redundant data is greater than the threshold (i.e. the joined ratios,  $\hat{\rho}_{dataset2} : \hat{\rho}_{dataset1}$ , are smaller).

In the last case, there is no redundant data in the input datasets with 2GB (i.e. the joined ratios are 100% : 100%). The Reduce-side join is better than the others because the filtering operation is not necessary here.

#### 3.5.2 Chain joins

##### 3.5.2.1 Cluster environment and datasets

We run experiments for the chain join on another computer cluster of 15 virtual machines using KVM (Kernel-based Virtual Machine) [79]. Each machine has two 1.4Ghz AMD Opteron CPUs with 512KB cache, 5GB RAM and 100GB SATA disks. The operating system is 64-bit Ubuntu server 12.04.2 LTS, and the java version is 1.7.0.21. We installed Hadoop [46] version 1.0.4 on all nodes. The other configurations of this cluster are similar to the ones of the cluster running the experiments of the two-way join. The number of reduce tasks is set to 25.

All test datasets were also produced by the data generation script of the PUMA. The maximum number of columns in the datasets is 39 and string length in each column is set 19 characters. The datasets *dataset1*, *dataset2*, *dataset3*, and *dataset4* contain the join key columns such as *column1* ( $c_1$ ), *column2* ( $c_2$ ), *column3* ( $c_3$ ), and *column4* ( $c_4$ ). Table 3.7 summarizes the different dataset sizes used in our experiments.

Table 3.7: Input datasets used in three tests

Inputs	Test 1		Test 2		Test 3	
	<i>size</i>	<i>records</i>	<i>size</i>	<i>records</i>	<i>size</i>	<i>records</i>
dataset1	10GB	26,836,497	20GB	53,675,946	20GB	53,682,929
dataset2	3GB	8,051,454	10GB	26,838,960	30GB	73,881,305
dataset3	10GB	26,836,497	20GB	53,675,946	20GB	53,682,929
dataset4	3GB	8,051,454	10GB	26,838,960	30GB	73,881,305
Total	26GB	69,775,902	60GB	161,029,812	100GB	255,128,468

We used three sets of the test datasets such as Test 1, Test 2, and Test 3. These tests correspond to 26GB, 60GB, and 100GB. Each the test includes the two inputs *dataset1* and *dataset2*. Each the test includes the four inputs *dataset1*, *dataset2*, *dataset3*, and *dataset4*. All the datasets are saved in the same text file format.

##### 3.5.2.2 Experimental protocol

Seven chain join algorithms developed in our experiments are the Reduce-side join cascade, the Bloomjoin cascade, the intersection filter-based join cascade (using three filtering approaches: the pair of the filters, the IBF, and the partitioned IBF), the optimal two-way join cascade (the solution (a) of optimization for the chain join), and the optimal three-way join cascade (the solution (b)).

We run our experiments by executing the different algorithms for a chain join query on the datasets of each the test. The following chain join query is used.

```
SELECT *
FROM dataset1(c1..c10) d1, dataset2(c1..c10) d2,
     dataset3(c1..c10) d3, dataset4(c1..c10) d4
WHERE   d1.column2 = d2.column2 AND
        d2.column3 = d3.column3 AND
        d3.column4 = d4.column4 AND
        d1.ROWNUM <= $number1 AND
        d2.ROWNUM <= $number2 AND
        d3.ROWNUM <= $number3 AND
        d4.ROWNUM <= $number4
ORDER BY d4.column4
```

The query is executed by changing *\$number1*, *\$number2*, *\$number3*, and *\$number4* to the number of records of the *dataset1*, *dataset2*, *dataset3*, and *dataset4*, respectively. A quadruple (*\$number1*; *\$number2*; *\$number3*; *\$number4*) corresponds to one test. For example, (26,836,497; 8,051,454; 26,836,497; 8,051,454) is the Test 1 of 26GB.

An output tuple of the experiments *t* is defined by the concatenation of four tuples of the first 11 columns that joined to produce the output. Furthermore, for general joins, we set up a many-to-many relationship between the datasets.

For each of the tests, we compare the seven chain join algorithms on three main aspects such as the total intermediate data amount, the total output data amount, and the total execution time.

### 3.5.2.3 Evaluation of approaches

A given false positive probability *f*, the length of the Bloom filter *m* is proportionate to the number of elements being filtered *n*, *m/n* is the number of bits allocated for each join key and *m<sub>k</sub>* is the size of a partition of the partitioned filter. The experiments use the parameters of the Boom filters as calculated in Table 3.8.

Table 3.8: Parameters of filters used in experiments

Tests	<i>f</i>	<i>k</i>	<i>n</i>	<i>m/n</i>	<i>m</i> (bit)	<i>m<sub>k</sub></i> = <i>m/k</i> (bit)
Test 1	0.000101	8	13147	21	276087	34511
Test 2	0.000101	8	13840	21	290640	36330
Test 3	0.000101	8	15295	21	321195	40150

First, we consider the total amount of intermediate data generated by each the chain join algorithm as in Table 3.9.

### 3.5 Experimental evaluation

Table 3.9: The total number of intermediate tuples (all map outputs)

Chain join algorithms	26GB. Test 1	60GB. Test 2	100GB. Test 3
Intersect-Filter-Joins	1,309,349	1,469,048	1,497,692
Pair-Filters-Joins	1,309,349	1,469,048	1,497,692
PartIntersect-Filter-Joins	1,309,349	1,475,849	1,497,692
BloomJoins	45,402,907	89,201,979	89,248,190
Reduce-Side-Joins	88,296,034	196,465,292	290,582,143
Chain-Optimal-2-WayJoin	1,281,036	1,417,684	1,445,428
Chain-Optimal-3-WayJoin	1,221,769	1,359,575	1,385,053

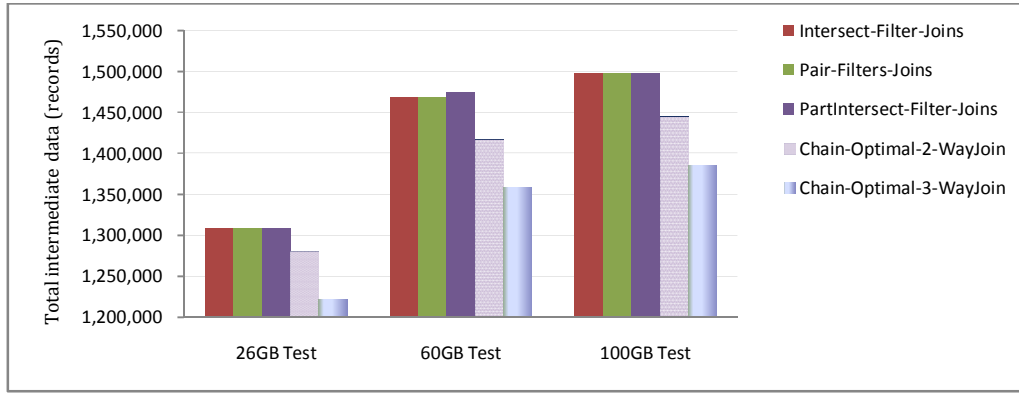


Figure 3.22: Total intermediate data

As we can see in Table 3.9, the cascades of Bloomjoins and Reduce-side joins generate much more intermediate data than any other chain join algorithms because of the existence of much redundant data. We take a look at Figure 3.22 to have a visual look for a comparison of the others using the intersection filters. The chain optimal three-way join has the least amount of intermediate data because it has less intermediate join jobs than the two-way join cascades, and has no redundant data in the intermediate join result(s). The intermediate data amount of the chain optimal two-way join is slightly larger than the chain optimal three-way join due to more intermediate join jobs. However, it is still better than the typical intersection filter-based join cascades. The typical intersection filter-based join cascades cannot prevent redundant data included in intermediate join results. Overall, the intermediate data amounts of these typical intersection filter-based methods are almost the same. The chain two-way join using the partitioned intersection filters tends to generate more intermediate data than using the different intersection filters.

Next, we examine the total output of the chain join algorithms. The total output consists of all the intermediate data and the intermediate join results. In other words, it includes all map output records and reduce output records of the chain join. This output has significant overheads involving I/O and communication overheads. The results of the total output are presented in Figure 3.23.

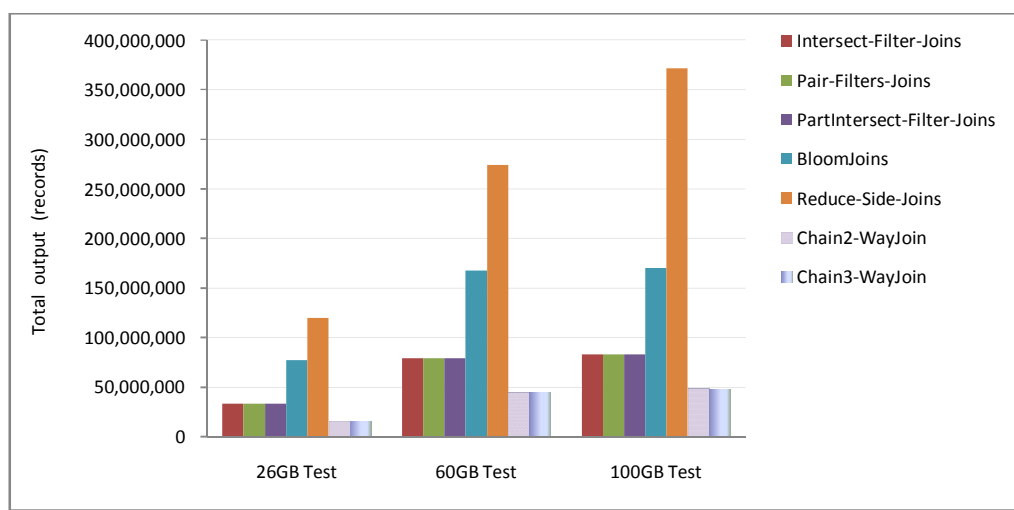


Figure 3.23: Total output data (Map output + Reduce output)

Figure 3.23 illustrates the amount of the total output of the seven chain join algorithms over the three tests 26GB, 60GB and 100GB. It can clearly be seen that the Reduce-side join cascade and the Bloomjoin cascade generate the largest outputs, whilst the chain optimal two-way join and the chain optimal three-way join using the intersection filters (Chain2-WayJoin and Chain3-WayJoin) have the least outputs of the seven. The chain intersection filter-based joins generally produce a little more output than the optimal chain joins. The main reason is that the optimal chain joins have the ability to filter out much more redundant data than the others.

To begin, the Reduce-side join cascade and the Bloomjoin cascade show a similar pattern, with both significantly increasing for the tests from 26GB to 100GB. Obviously, the Reduce-side join cascade is the highest over all the tests. In the Test 1, the Reduce-side join cascade outputs around 119,928,957 records, while the Bloomjoin cascade about 77,035,830 records and the chain intersection filter-based joins about 32,942,272 records lower. With the similarity in the Test 3, the Reduce-side join cascade produces around 371,782,345 records, whereas the Bloomjoin cascade about 170,448,392 records and the chain intersection filter-based joins about 82,697,894 records much lower.

The outputs that are generated the least are the Chain2-WayJoin and the Chain3-WayJoin. In the Test 1, the Chain2-WayJoin emits around 15,577,281 records, while the Chain3-WayJoin about 15,255,188 records lower. Observing the Test 3, the Chain2-WayJoin produces around 48,436,677 records and the Chain3-WayJoin about 48,097,527 records lower.

Lastly, we make a performance comparison among the seven chain join algorithms. Overall, the optimal chain joins have the total execution time the lowest because they produce the total output the least. This is demonstrated in the following.

### 3.6 Summary

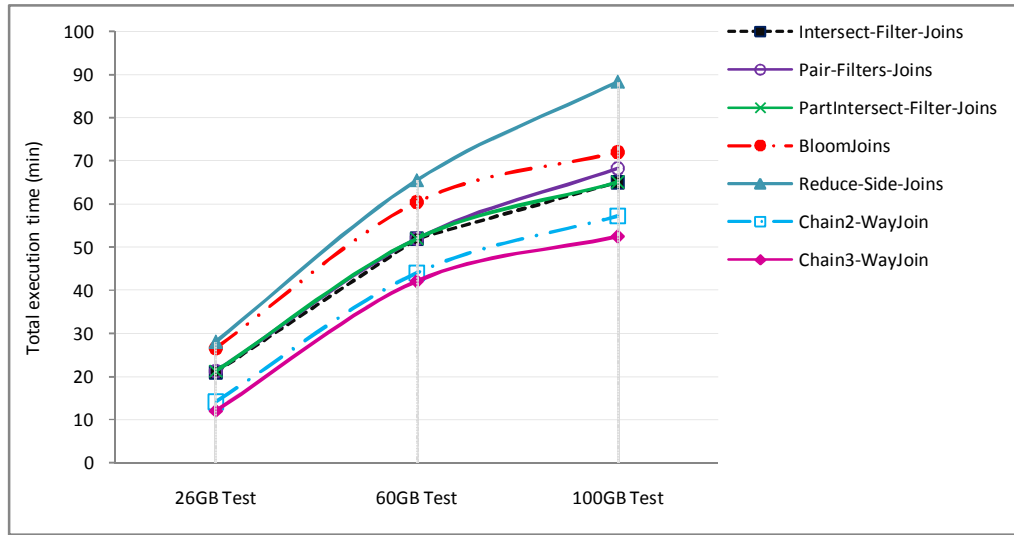


Figure 3.24: Total execution time

Figure 3.24 presents the total execution time of the chain join using the different algorithms from 26GB to 100GB. There are seven graphs in the chart. The bottom two graphs show the total execution time of the optimal chain joins, the next three ones deal with the chain intersection filter-based joins and the top two graphs show the Bloomjoin cascade and the Reduce-side join cascade. For the test 100GB, the Chain2-WayJoin and the Chain3-WayJoin run about 52.57 and 57.22 minutes respectively, while the chain intersection filter-based joins about 65.13 minutes. The Bloomjoin cascade and the Reduce-side join cascade execute about 72.09 and 88.34 minutes much longer. It is similar to compare the total execution time of the algorithms for the remaining tests. From the chart, we can conclude that the optimal chain joins have the total execution time the least, although they additionally run the pre-processing job. This is logical since they have less the output than as analyzed above. It is observed that the Chain3-WayJoin tends to perform better than the Chain2-WayJoin.

### 3.6 Summary

In this research, we consider the problem of computing the intersection Bloom filter to optimize two-way joins and important multi-way joins in MapReduce. Based on the probabilistic model, three ways are proposed on the intersection filter such as the pair of Bloom filters, the intersection of Bloom filters, and the intersection of partitioned Bloom filters. The intersection filter is then applied to two-way join operations to eliminate most of the non-joining tuples in the input datasets before sending the intermediate pairs to actual join processing. Additionally, we make an extension of the intersection filter to improve the performance of three-way joins and chain joins including both cyclic chain joins with many shared join keys. The two optimized solutions for chain joins are proposed in this research. We use the Lagrangian multiplier method to indicate a good choice between the two solutions. Remarkably, we build the general cost models for two-way joins and multi-way joins. Thanks to these cost models, we can make comparisons of the join algorithms more persuasive. As a result, with using the intersection filters, the join operations

can minimize disk I/O and communication costs. Finally, the intersection filter-based join operations are demonstrated to be more efficient than existing solutions through the experimental evaluations. The joins using a pair of filters and the unpartitioned intersection filter are more efficient than the joins using the partitioned intersection filter because of their filtering performance. However, the partitioned intersection filter is easy to discover disjoint datasets on a join key column and stop the join processing.

This work leads to one publication [5] in Proceedings of the 2Nd International Workshop on Cloud Intelligence (Cloud-I@VLDB).



## OPTIMIZATION FOR RECURSIVE JOINS AND SEMI-NAIVE ALGORITHM

Implementing a recursive join is considered as the calculation of the transitive closure to evaluate a recursive query with fixpoint semantics. It is a complex and expensive operation because it involves repeating the join operation. In the MapReduce environment, the issue becomes even more complicated when we implement a recursive join as an iteration of a join job and a deduplication-difference job (*dedup-diff job*). In this chapter, we present a simple and efficient solution for recursive join evaluation. It folds the join job and the dedup-diff job into one single job using a *Difference filter*. The evaluation, based on alternating sequences of *Join*  $\rightarrow$  *Deduplication-Difference* operations, is now replaced by an iteration of one combined operation. This improvement will significantly reduce the number of executed jobs by half, and especially the overheads of data rescanning, intermediate data, and communication for the deduplication and difference operations. Therefore, the difference filter-based optimization for recursive joins as well as the general semi-naive algorithm is thoroughly considered in this research.

We discuss previous works and propose our solution for optimizing recursive joins in Section 4.1. Some definitions and notations are also introduced. Each remaining section of this chapter therefore highlights a contribution of our work. Section 4.2 provides a *difference filter* to check whether an element is not in a set. The existing problems, concepts, and design details for the difference filter are described. Specially, the false difference probability that affects to the filtering performance is also considered and analyzed thoroughly. Next, Section 4.3 presents an optimization for the recursive join as well as the semi-naive algorithm. The processing phases and the general algorithm 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 4.4. Finally, we conclude our contributions in Section 4.5.

### 4.1 Introduction

#### 4.1.1 Previous work

We consider the query  $Q_3$  in Chapter 2 that is a recursive join query also known as a typical transitive closure query. We can point out its similar form expressed in Datalog as follows.

$$\begin{aligned} \text{Friend}(x, y) &\leftarrow \text{Know}(x, y); \\ \text{Friend}(x, y) &\leftarrow \text{Friend}(x, z) \bowtie \text{Know}(z, y); \\ \text{“friend”} &\text{ depends on Know and itself; recursive} \end{aligned}$$

There are many algorithms designed to compute the transitive closure of a database relation in the literature [80][23][81][82]. However, they are not always well suited for implementing in the MapReduce environment. Several recent studies have found solutions for evaluating this query type in the environment. Afrati et al [7][12] propose an implementation of recursion on a cluster with addressing the transitive closure as a starting point. The authors show how to significantly reduce the number of needed rounds for evaluating nonlinear transitive closures. Namely, the solution decreases the number of rounds to  $O(\log_2 n)$  rather than  $O(n)$  on a  $n$ -node graph.

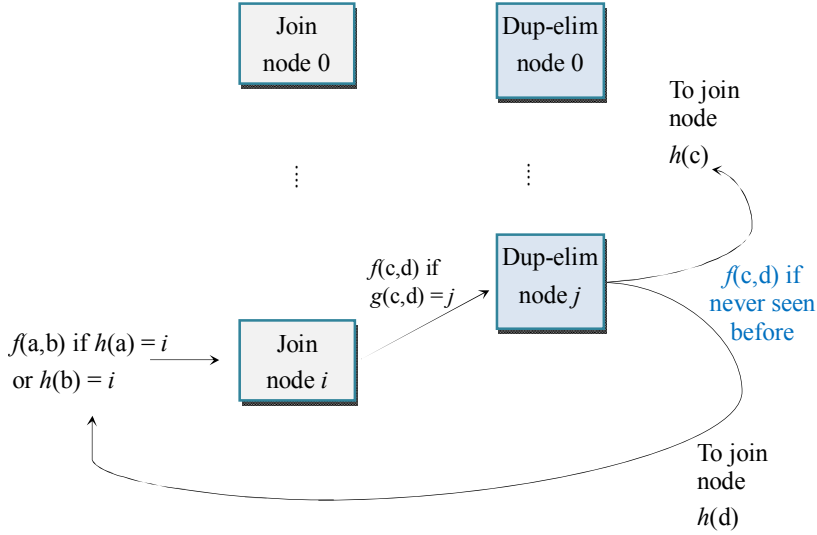


Figure 4.1: Relationship between join and dup-elim tasks

As shown in Figure 4.1 [12], the solution uses two groups of tasks consisting of *Join* tasks and *Dup-elim* tasks. The join tasks compute the join of tuples. The dup-elim tasks is to remove duplicate  $f$ -tuples before they can be delivered. Each join task  $i$  receives and stores no duplicate tuples  $f(a, b)$  such that  $h(a)$  or  $h(b)$  is  $i$ . It then searches its store for tuples  $(b, c)$  and  $(c, a)$ , and sends corresponding tuples  $(a, c)$  and  $(c, b)$  to dup-elim tasks numbered  $g(a, c)$  and  $g(c, b)$ , respectively. The dup-elim task checks each received tuple in its store. If the tuple exists, it is omitted. Otherwise, the tuple is stored and sent to join tasks  $h(a)$  and  $h(b)$ .

A major obstacle of this solution is due to long-running recursive tasks that may increase risk for failures. In addition, there are modifications to the typical MapReduce framework such as *blocking property* and failure recovery methods. The blocking property aims to deals with compute node failures by controlling tasks in a way that each task does not deliver output to any other task until it has completely finished its work. For this solution, however, the tasks cannot have the blocking property in which the tasks can deliver some output before finishing. As a result, it uses alternative failure recovery mechanisms such as *idempotence* and *checkpointing* [12] which are complex and are not directly supported in Hadoop. Moreover, this solution is used to calculate nonlinear transitive closures and its communication cost

## 4.1 Introduction

is typically much greater than that of linear transitive closures due to the output replication of the dup-elim tasks.

Pregel [83] executes true recursion on a graph using the Bulk Synchronous Parallel (BSP) model, but checkpoints all tasks at intervals. If there is a failed task, all tasks are rolled back to the previous checkpoint.

HaLoop [84] has modified version of Hadoop to support efficient iterative data processing on clusters. This system implements recursion by repetition of MapReduce jobs and minimizes communication by caching the Mapper Input (MIC) and the Reducer Input/Output (RIC/ROC). This solution can avoid re-scanning and re-shuffling data on every iteration, of course it still must rescan the caches. A limitation is that tasks should operate in synchronous rounds and the output of one task must be passed to the next MapReduce phase. In addition, a drawback of the cache implementation in the current HaLoop comes from completely rewriting the cache on every iteration. Moreover, HaLoop still uses an old version (0.20.2), and it is not updated to the latest versions of Hadoop.

We look at another algorithm for evaluating the recursive join query. The well-known semi-naive algorithm [85] is used to find the fixpoint of the evaluation. It replaces recursion by a repetition of MapReduce job(s). In this algorithm, incremental relations are used to avoid recomputing the same facts.

Assuming  $F$  and  $K$  denote the relations *Friend* and *Know*, respectively. Let  $F_{i,i}$  between 0 and  $n$ , be the temporary value of the relation *Friend* at iteration step  $i^{th}$ , and  $K$  be the relation *Know* at all iterations. The differential of  $F_i$  between step  $i$  and step  $i-1$  is defined as follows.

$$\Delta F_i = F_i - F_{i-1} = \prod_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$$

$\Delta F_i$  is also called an incremental relation of the relation  $F_i$  at iteration step  $i$ . The details of the algorithm is shown in Listing 4.1.

---

**Algorithm 1** - Semi-Naive evaluation for recursive joins

---

```

 $F_0 = \emptyset, \Delta F_0 = K(x,y), i=1$ 
While  $\Delta F_{i-1}$  not empty do
     $F_{i-1} = (\Delta F_0 \cup \dots \cup \Delta F_{i-1})$ 
     $\Delta F_i = \prod_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$ 
     $i++$ 

```

---

Listing 4.1: Pseudo code for Semi-naive algorithm

At each iteration step  $i$ , some new facts are inferred and stored in  $\Delta F_i$ . To infer a new fact at step  $i$ , one must use at least one fact derived at step  $i-1$ . The loop is repeated until no new fact is inferred ( $\Delta F_i = \emptyset$ ), i.e., the fixpoint is reached. The result is that the union of all the incremental relations,  $(\Delta F_0 \cup \dots \cup \Delta F_{i-1})$ , is a least fixed point of the query.

The advantage of the semi-naive method is that at each iteration a differential term  $\Delta F_{i-1}$  is used in each join computation instead of the whole  $F_{i-1}$ . For this way, the time complexity of a computation is decreased significantly.

Shaw et al [14] have proposed an optimization for implementing this semi-naive algorithm in MapReduce as follows. On each iteration of the evaluation, the command line  $\Delta F_i = \prod_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$  is compiled into two MapReduce jobs, namely, one for join job and one for deduplication and difference (*dedup-diff*) job. Their implementation is described in MapReduce framework as Figure 4.2.

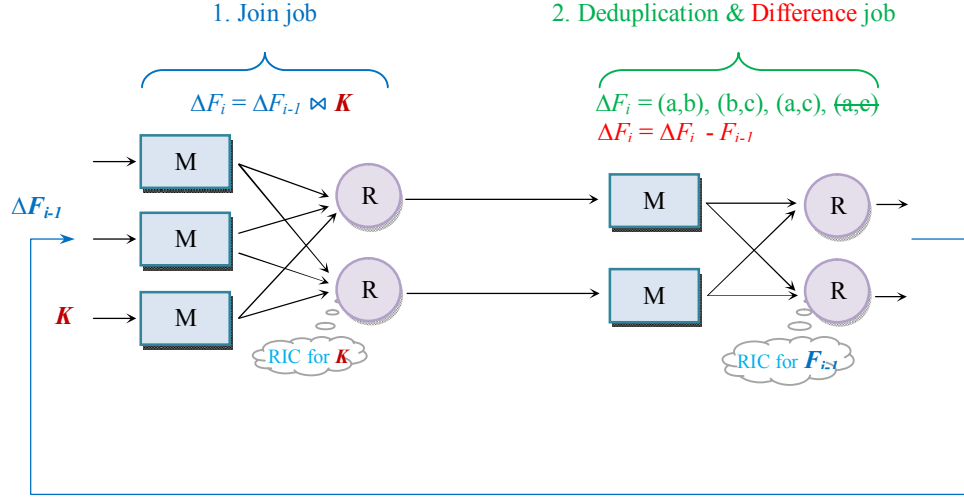


Figure 4.2: Semi-naive implementation of recursive joins in MapReduce

The evaluation of a recursive join first executes the join job ( $\Delta F_{i-1} \bowtie K$ ) that joins the incremental relation of *Friend* with the relation *Know* to produce new tuples of the result. The second job aims to eliminate duplicate tuples in the result, compute the difference of the new result and all previous results, and then generate next incremental relation  $\Delta F_i$ . This execution plan is then iterated until the  $\Delta F_i$  is empty.

The difference job using the RIC cache is described as follows. Each tuple is stored in the cache as a key/value pair  $(t, i)$ , where the key is the tuple  $t$  discovered by the previous join job and the value is the iteration number  $i$  for which that tuple was discovered. On each iteration, the map phase of the difference job hashes the incoming tuples as keys with values indicating the current iteration number. During the reduce phase, for each incoming tuple (from the map phase), the cache is probed to find all instances of the tuples previously discovered across all iterations. Both the incoming and cached data are passed to the user-defined reduce function. Any tuples that were previously discovered are omitted from the output. If the tuple had never before been seen, this tuple should be included in the  $\Delta F_i$  and emit the tuple.

#### 4.1.2 Proposal for recursive join using filters

Our research focuses on the general semi-naive algorithm for computing the recursive join as well as the transitive closure of a relation. More importantly, this algorithm can be translated to the MapReduce distributed computing environment. The main idea behind the algorithm is a loop containing operations such as join,

projection and difference to calculate the transitive closure breadth-first. However, the MapReduce model is not the convenient model for the iterative computation and the join operation. As illustrated in Figure 4.2, it turns out that the semi-naive algorithm has some problems that need to be considered.

- (1) With the join job ( $\Delta F_{i-1} \bowtie K$ ), the relation  $K$  is always re-scanned and re-shuffled on every iteration even though it is invariant.
- (2) With the difference job ( $\Delta F_i - F_{i-1}$ ), all the incremental relations ( $F_{i-1}$ ) are also re-scanned and re-shuffled on every iteration.
- (3) On each loop, there are the two jobs consisting of the join job and the difference job; this makes implementing the recursive join quite expensive.

Shaw et al have addressed the problems (1) and (2) in the HaLoop system by using the RIC cache. To avoid re-scanning and re-shuffling the datasets on each loop, the solution uses the RIC cache for the datasets  $K$  and  $F_{i-1}$  in the job join and the dedup-diff job, respectively, as described in Figure 4.2. However, the solution from Shaw cannot overcome the problem (3) because it still requires the additional difference job to calculate the incremental relation  $\Delta F_i$ . This job takes expensive overheads such as rescanning the output of the join job on DFS, incurring a new job, generating intermediate data from the output of the join job, and shuffling the intermediate data. Besides, the overhead of implementing the cache is significant because all discovered results are cached, indexed and probed during the evaluation. In addition, the cache is rewritten completely on every iteration during which new results are discovered. Therefore, folding the difference operator into the join job would considerably save the overheads for the recursive join implementation. This also improves the semi-naive algorithm in MapReduce such that the number of computation steps as well as the jobs can be reduced to  $l$  instead of  $2 \times l$ , where  $l$  is the longest path length in the relation graph - 1.

For this reason, we propose a solution to optimize the recursive join and the semi-naive algorithm in MapReduce as follows.

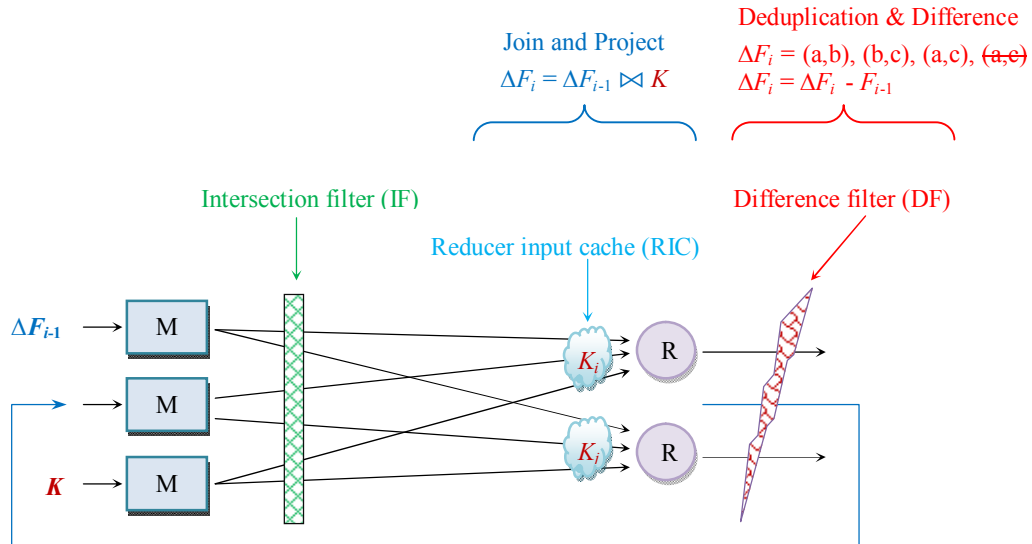


Figure 4.3: Filter-based optimization for the semi-naive algorithm and recursive joins in MapReduce

As shown in Figure 4.3, a typical MapReduce job performs the join operation between two input datasets  $K$  and  $\Delta F_{i-1}$  (the incremental relation at iteration step  $i-1$ ). The dataset  $K$  is scanned only one time at the first loop.  $K_i$  and  $K_j$  are splits of  $K$ , which are cached at the reducer input caches  $i$  and  $j$ , respectively. An *intersection filter* (IF) [5] contains common join keys between the input datasets. The reducer input cache is used to cache the loop-invariant relation  $K$ . A *difference filter* (DF) stores distinct tuples discovered. The join job uses the IF to remove non-joining data at the map phase. The RIC is used to avoid rescanning the input dataset  $K$  on each iteration. The DF is to eliminate duplicate results and computes actual new results for the next iteration.

For our solution, the recursive join is implemented as an iteration of the join job. Initially, the IF and the DF are empty. The dataset  $\Delta F_0$  is assigned to  $K$ . The map phase hashes the tuples of both the datasets by the join key. The tuples whose join keys are not in the IF are eliminated. This checking is not conducted in the first loop since the BF is nothing. The tuples then are passed to the reducers. At the reducer, the RIC caches the tuples of  $K$  to avoid re-scanning and re-shuffling the loop-invariant dataset  $K$  on the next iterations (the problem (1)). The reduce phase performs the join of  $\Delta F_{i-1}$  and  $K$  for each unique key. Each result of the join is queried into the difference filter DF. If the result is not in the DF, it is the actual new result and thus it is hashed to the DF and emitted to the output  $O_i$ . Otherwise, it is the duplicate result and omitted. The output  $O_i$  is the incremental relation  $\Delta F_i$  that is used for the next iteration. In addition, the intersection filter IF is recomputed by the common join keys from  $\Delta F_i$  and  $K$ . The iteration of this job ends when the IF is empty because of no common join key between  $\Delta F_i$  and  $K$ . The final output of the recursive join includes all the outputs  $O_i$ . It is very important to note that using the difference filter DF avoids the overheads of re-scanning and re-shuffling all the incremental relations ( $F_{i-1}$ ) and incurring the additional difference job (the problem (2) and (3)).

We address the problems of the general semi-naive algorithm for evaluating the recursive join query in the MapReduce environment, and propose the difference filter to compute the incremental relation without using the expensive difference job. Two key aspects of the difference filter need to be considered including (1) approach to modeling the difference filter, (2) probability of a false difference.

### 4.1.3 Definitions and notations

We introduce definitions and notations used in this research.

**Definition 4.1.** A *disjoint element* of a set is an element NOT in the set. Given a set  $S$ ,  $x$  is a disjoint element of  $S$  if  $x \notin S$ .

**Definition 4.2.** *Disjoint elements* of sets  $S_1$  and  $S_2$  are elements of  $(S_1 \setminus S_2) \cup (S_2 \setminus S_1)$

**Definition 4.3:** A *difference filter* (DF) is a probabilistic data structure designed to represent a set and examine whether an element is **NOT** present in the set. In other words, the difference filter of a set is to recognize the disjoint elements of the set. This is contrary to a Bloom filter used for membership queries.

## 4.2 Modeling difference filter

For formality, suppose that  $R$  and  $S$  are two sets. The difference filter of  $S$ ,  $DF(S)$  that represents the set  $S$ , is used to test whether an element  $x$  of  $R$  is NOT in  $S$  ( $x \in R \setminus S$ ).

**Definition 4.4:** A *recursive join* of a relation is an operation to compute the transitive closure of the relation. It is a compound operation, which involves repeating the join operation until no further result is produced (“fixpoint”).

Table 4.1: List of notations

Notation	Explanation
$K$	A dataset <i>Know</i> that is a loop-invariant dataset
$\Delta F_i$	An incremental relation <i>Friend</i> in iteration $i$
$F_i$	All incremental relations on iterations 0 to $i$ ( $\Delta F_0 \cup \dots \cup \Delta F_i$ )
$DF(S)$	A difference filter built for a set $S$
$BF(S)$	A Bloom filter built for a set $S$
$DBF(S)$	A dynamic Bloom filter built for a set $S$
$LHT(S)$	A Lossy Hash Table built for a set $S$
$T$	A Hash table
$DFS$	Distributed File System
$\setminus$	The difference operator, e.g. $R \setminus S$ is the difference of $R$ with $S$

## 4.2 Modeling difference filter

### 4.2.1 Existing solutions

First, we should consider existing solutions that are relevant to this issue, e.g., data reconciliation, deduplication, error-correction, etc. Data reconciliation and deduplication are important tasks in distributed systems and have been carried out in a few different ways. These tasks can be efficiently carried out thanks to accurately identifying disjoint elements of two sets.

The easiest way to recognize disjoint elements of two sets is based on hash tables. The hash table  $T$  contains fingerprints of all elements belonging to one set and checks whether elements of another set are present. For instance, we perform reconciliation in a distributed environment for a set of records  $A$ . Assume that  $A_1$  and  $A_2$  of  $A$  are distributed over two various data sites  $S_1$  and  $S_2$ , respectively.  $S_1$  sends the hash table  $T_1$  of  $A_1$  to  $S_2$  and receives  $T_2$  of  $A_2$  from  $S_2$ . We can now specify disjoint elements through querying the records of  $A_1$  into the table  $T_2$  at  $S_1$ , and the records of  $A_2$  into  $T_1$  at  $S_2$ . Using a perfect hash function for the sets, this approach needs  $O_c(|A_1| + |A_2|)$  communication overhead for exchanging these two hash tables  $T_1$  of size  $|A_1|$  and  $T_2$  of size  $|A_2|$ . It also requires  $O_t(|A_1| + |A_2|)$  run time to query  $|A_1|$  and  $|A_2|$

elements into the two hash tables. However, the hash tables are space-inefficient on large data sets because the number of buckets in the hash table grows at the same rate as the cardinality of the set so that it remains nearly the same size.

A better solution involves an approximate membership query data structure called a Bloom filter [15], which excels at determining if an item is a member of a set. Since we exchange the filters containing only bits that represent elements of one set, the communication overhead can be reduced and is proportional to the size of the filter. Although, the run time for checking elements in the two sets is also  $O_i(|A_1| + |A_2|)$ , the approach is very space-efficient for both large data sets because Bloom filter uses a bit array and its size is fixed regardless of the cardinality of the set. Of course, there is a clear tradeoff between the size of the filter and the false positive probability. As shown in [74], if the number of elements in the set does not change, the error probability decreases as the size increases (i.e., more memory usage).

Several recent approaches have extended Bloom filter such as Stable Bloom Filters [86], Time Interval Bloom Filters [87], Approximate Reconciliation Trees [29], horizontal and vertical Bloom filters [88], etc. They minimize the amount of memory assigned to the filters and the number of errors, including *false positives* and *false negatives*. It is noted that a *false positive* is a disjoint element wrongly reported as duplicate, and a *false negative* is a duplicate element wrongly reported as difference.

Unfortunately, these solutions work on a shortcoming because of accepting false positives. Based on the standard Bloom filter, the extended filters always generate false positives [86][87][29][88] and sometimes additional false negatives [86][88]. As a result, the filters can indicate a superset of duplicate elements, consisting of all actually duplicate elements and a few disjoint elements that are false positives. In contrast, they cannot identify a set of all disjoint elements because the missing disjoint elements of the set belong to the false positives approximated by a probability. This is an obstacle for applying the Bloom filters to deduplication and reconciliation. For an instance of deduplication, if an element  $x$  exists in the filter;  $x$  will be omitted because it is a duplicate element. However, a disjoint element  $y$  will be still omitted if the filter reports  $y$  as an existing element due to a false positive error. The solutions [86][88] can reduce the false positive errors without removing them.

That is unacceptable for our difference filter in which it requires the ability to recognize a superset of all disjoint elements. To achieve this goal, the difference filters should only generate false negatives without false positives. The false negatives are duplicate elements wrongly reported as disjoint elements. Precisely, the difference filters allow us to specify a superset, which consists of all disjoint elements and a few duplicate elements with a small rate.

Another good alternative was proposed by Eppstein, Goodrich et al. [89]. Actually, the authors have solved the set difference problem that is related to our problem. This solution supplies a data structure *Difference Digest* to compute the set difference with communication proportional to the size of the difference. The data structure is based on an *Invertible Bloom filter* [31], [72] (InvBF). The InvBF is a variant version of the Bloom Filter that uses a three-component data structure to supports not only the insertion, deletion, and lookup of key-value pairs, but also allows a listing of its contents with high probability. Figure 4.4 below describes the data structure.



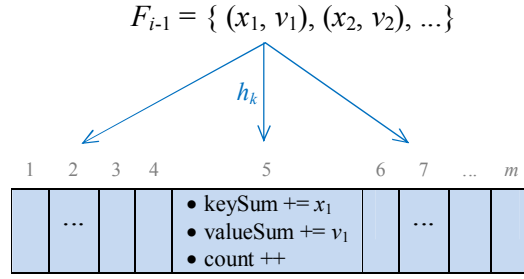


Figure 4.4: Invertible Bloom filter

Each bucket of the InvBF contains three fields, which are initially set to 0:

- A *keySum* field, which is the sum of all the keys that have been mapped to this bucket.
- A *valueSum* field, which is the sum of all the values that have been mapped to this bucket.
- A *count* field, which counts the number of entries that have been mapped to this bucket.

The invertible Bloom filter supplies fourth operations:

- INSERT( $x, v$ ): insert a key-value pair, ( $x, v$ ), into the InvBF.
- GET( $x$ ): return the value  $v$  such that there is a key-value pair, ( $x, v$ ), in the InvBF.
- DELETE( $x, v$ ): delete the key-value pair, ( $x, v$ ), from the InvBF.
- LIST\_ENTRIES(): list all the key-value pairs being stored in the InvBF. With low probability, it may return a partial list along with an “*list-incomplete*” error condition.

The INSERT and DELETE operations never fail, whereas the GET and LIST\_ENTRIES operations may fail with low probability.

Similarly to the standard Bloom filter, the InvBF uses  $k$  hash functions to compute locations for storing an element in an array of buckets. The hash functions  $h_k()$  will result in storing multiple data elements in the same location. Therefore, it cannot prevent collisions and a form of collision resolution is needed.

We can see that we cannot store  $x$  and then store  $z$  in the same location because this would wipe out all trace of  $x$ . The solution is to use a reversible storage function. For example, if we stored  $x$  in an *InvBF* at location  $i$  and then we insert  $z$  into the same location of  $x$ , we add it to the key sum:

$$\text{InvBF}[i].\text{keySum} = \text{InvBF}[i].\text{keySum} + z \quad // \text{ i.e., } = x + z$$

We can remove  $z$  by subtracting it:

$$\text{InvBF}[i].\text{keySum} = \text{InvBF}[i].\text{keySum} - z \quad // \text{ i.e., } = (x + z) - z = x$$

It means that if the invertible Bloom filter already stored  $x$  before and we added  $z$  then we subtract  $z$ , we can get the value  $x$  back again. For this reason, the

INSERT() function uses the addition operator and the DELETE() function uses the subtraction operator.

Applying the invertible Bloom filter to compute the set difference includes three steps such as encode, subtraction and decode. Suppose  $R\{X, Y, V, W\}$  and  $S\{Y, Z, W\}$  are two sets,  $k$  hash functions  $h_k()$  are used to locate positions ( $k=3$ ),  $h_c()$  is a cryptographic hash function that maps an element to a fixed-size bit string. The set difference between the sets is conducted as follows.

- *Encode*: constructs two invertible Bloom filters,  $InvBF_R$  and  $InvBF_S$  initialized to zero, by inserting each element  $x$  in  $R$  or  $S$  to  $InvBF_R$  or  $InvBF_S$ , respectively. For each index  $i$  returned from  $h_k()$ , we XOR  $x$  into  $InvBF[i].keySum$ , XOR  $h_c(x)$  into  $InvBF[i].hashSum$ , and increase  $InvBF[i].count$ .
- *Subtraction*: subtracts  $InvBF_R$  from  $InvBF_S$  cell by cell. To subtract cells, the *keySum* and *hashSum* fields are XOR'ed, and *count* fields are subtracted. The results of the subtracting are written to a new invertible Bloom filter,  $InvDiff$  of the same size. This subtracting process is illustrated in Figure 4.5.

$InvBF_R = \{X, Y, V, W\}$

keySum	V+X+ <b>Y</b>	V+ <b>W</b> +X	X	V+ <b>W</b> + <b>Y</b>	<b>W</b> + <b>Y</b>
hashSum	$h_c(V)+h_c(X)+h_c(Y)$	$h_c(V)+h_c(W)+h_c(X)$	$h_c(X)$	$h_c(V)+h_c(W)+h_c(Y)$	$h_c(W)+h_c(Y)$
count	3	3	1	3	2

$InvBF_S = \{Y, Z, W\}$

keySum	<b>Y</b>	<b>W</b> +Z	Z	<b>W</b> + <b>Y</b>	<b>W</b> + <b>Y</b> +Z
hashSum	$h_c(Y)$	$h_c(W)+h_c(Z)$	$h_c(Z)$	$h_c(W)+h_c(Y)$	$h_c(W)+h_c(Y)+h_c(Z)$
count	1	2	1	2	3

$InvBF_R \oplus InvBF_S$

$InvDiff = InvBF_R - InvBF_S$

keySum	V+X	V+X+Z	X+Z	V	Z
hashSum	$h_c(V)+h_c(X)$	$h_c(V)+h_c(X)+h_c(Z)$	$h_c(X)+h_c(Z)$	$h_c(V)$	$h_c(Z)$
count	2	1	0	1	-1

Figure 4.5: InvBF Subtraction.  $InvDiff$  results from subtracting  $InvBF_R$  from  $InvBF_S$  cell by cell.

For each index  $i$ , we XOR  $InvBF_R[i].keySum$  and  $InvBF_S[i].keySum$  into  $InvDiff[i].keySum$ , XOR  $InvBF_R[i].hashSum$  and  $InvBF_S[i].hashSum$  into  $InvDiff[i].hashSum$ , and subtract  $InvBF_R[i].count$  from  $InvBF_S[i].count$  into  $InvDiff[i].count$ .

It is very important to note that the elements common to  $InvBF_R$  and  $InvBF_S$  (shown blue bolded) are cancelled during the XOR operation. Therefore, the result filter  $InvDiff$  only contains disjoint elements of the sets.

- *Decode*: recovers “pure” cells from the *InvDiff*'s table. Pure cells are those whose *keySum* matches the value of an element  $x$  in the set difference. In order to verify that a cell is pure, it must satisfy two conditions: the *count* field must be either 1 or -1, and the *hashSum* field must equal  $h_c(\text{keySum})$ . If the *InvDiff* is the result of subtracting the  $\text{InvBF}_R$  from the  $\text{InvBF}_S$ , then a positive *count* indicates  $x \in (R - S)$ , while a negative count indicates  $x \in (S - R)$ .

The decoding process begins by scanning the *InvDiff*'s table and creating a list of all pure cells. For each pure cell in the list, we add the value  $x=\text{keySum}$  to the appropriate output set ( $R - S$  or  $S - R$ ) and remove  $x$  from the table. The process of removal is similar to that of insertion. We compute the list of distinct indices where  $x$  is present, then decrement *count* and XOR the *keySum* and *hashSum* by  $x$  and  $h_c(x)$ , respectively. If any of these cells becomes pure after  $x$  is removed, we add its index to the list of pure cells. The process continues until no indices remain in the list of pure cells. At this point, if all cells in the table have been cleared (i.e. all fields have value equal to zero), then the decoding process has successfully recovered all elements in the set difference. Otherwise, some encoded elements remain in the table, but insufficient information is available to recover them. This problem is considered as “list-incomplete” error.

We should consider the following features of the solution mentioned by Eppstein:

- (1) Two sets  $R$  and  $S$  must be defined before performing subtraction.
- (2) It can list the disjoint elements encoded in the *InvDiff*.
- (3) The decoding process may fail because some encoded elements may not be recovered.
- (4) It requires an additional job for computing the set difference.

We realize that this solution is not suitable for our work. For the feature 1, we cannot know in advance the datasets to avoid rescanning the data multiple times. Besides, we need a checking for membership (e.g. whether an element  $x$  is a disjoint element) instead of a listing of disjoint elements in the filter (the feature 2). Moreover, the decoding process may only output a partial list of the disjoint elements due to some cells in the *InvDiff* with non-zero counts (the feature 3). This leads to a disadvantage that we cannot specify a superset of the disjoint elements that is requested in our work. Finally, for the feature 4, using an additional job to compute the set difference is expensive because it has to rescan the datasets, generate intermediate filters, and pass the filters to the reducers. The job can be folded into our join job.

### 4.2.2 Problem definition

From the limitations of the existing solutions, we propose a new filter type called *Difference Filter* as follows:

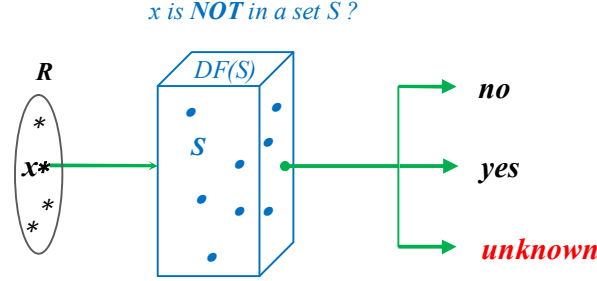


Figure 4.6: Difference filter returning an output with three possibilities

As illustrated in Figure 3.2, a difference filter  $DF(S)$  represents a set  $S$  and checks whether an element  $x$  of a set  $R$  is **NOT** in the set  $S$ . It accepts an input and returns an output that is one of three possibilities:

- **"no"** :  $x$  is NOT a disjoint element of  $S$  if  $x$  is in the set  $S$ .
- **"yes"** :  $x$  is a disjoint element of  $S$  if  $x$  is NOT in the set  $S$ .
- **"unknown"** :  $x$  "may be or may be NOT" a disjoint element of  $S$ .

With this assumption, when the difference filter returns an answer "no" or "yes", the answer is always the correct response and it is considered as an answer "known". An answer "unknown" may be the wrong response because  $x$  may be in the set  $S$ . This means that the difference filter only generates false negatives without false positives. As a result, the difference filter allows us to identify a superset of disjoint elements consisting of the "yes" and "unknown" elements, and eliminate duplicate elements that are the "no" elements. Accordingly, minimizing the number of the "unknown" elements will be the key for an effective solution to build the difference filter. In addition, it is noted that we cannot know elements of a set in advance to avoid reading data multiple times. Thus, the filter should be designed to update dynamically new elements according to the incoming data of a certain job, e.g., the join job. This feature aims to respond to the continuous updating of the incremental relations into the difference filter in the recursive join evaluation without using an additional job.

In our context, the difference filter is better than the prior solutions because it has the ability to dynamically filter out duplicate elements and retain disjoint elements with a specified false negative rate. The next sections therefore present a method to build the difference filter.



---

## CONCLUSIONS AND FUTURE WORK

In this final chapter, we will conclude by describing the results of the optimization for the two-way joins using the intersection filter and its extensions to the problems of the multi-way joins in MapReduce. Another important result is the improvement of the recursive joins using the difference filter. We will also suggest some future research directions which would further extend its applicability.

### 5.1 Thesis conclusions

Nowadays, more and more applications have encountered difficulties to handle large-scale data using traditional data processing methods. We can easily find such kind in applications of social networks, bibliographies, bioinformatics, databases, etc. The MapReduce programming model has become very popular recently for processing, analyzing and generating such large data in a massively parallel manner. However, this model has its own limitations. Complex operations in MapReduce are used extensively and expensively, especially the join operation. The research efforts have markedly expanded to address this problem and given some solutions surveyed in Chapter 2, in which a join operation will be compiled to MapReduce job(s). For these solutions, however, it is realized that much redundant data is involved in the join operation. Therefore, this dissertation is dedicated to solving the problems of the joins in the efficient ways. It focuses not only on the two-way joins, but also the complex joins such as the multi-way joins and the recursive joins. The main contributions of our research are the following:

#### (1) Intersection filter

Based on the probabilistic model, we propose three approaches to compute the intersection filter that approximates the intersection of sets. The approaches include the pair of Bloom filters, the intersection of Bloom filters, and the intersection of partitioned Bloom filters. The intersection filter is used to remove most of the disjoint elements between the sets.

#### (2) Difference filter

We define a new filter type, the difference filter, to represent a set and test whether an element is **NOT** present in the set. It is contrary to a Bloom filter used for membership queries. Notably, the false difference probability that affects to the performance of the filter is also considered and analyzed thoroughly. It can be applied to a wide range of popular problems such as recursive join operation, reconciliation and deduplication, error-correction, etc.

### (3) Optimization for two-way and multi-way joins and cost models

- We provide a survey on the prominent join algorithms in MapReduce recently.
- We minimize the amount of intermediate data generated in two-way joins and three-way joins using the different approaches of the intersection filter. The three-way join is then compared to the cascade of 2 two-way joins by the Lagrangian method.
- We point out two optimized solutions of chain joins using the intersection filter, the two-way join cascade and the three-way join cascade. Our analysis shows that the three-way join cascade is better than the two-way join cascade when the number of reducers is smaller than the output size of the join of two sets ( $r < (|R| \cdot \alpha)^2$ ) and becomes a good choice.
- We give the optimized join algorithms of two-way joins, three-way joins, and chain joins.
- We supply cost models for comparisons of our join algorithms and the existing algorithms.
- We also specify a threshold of the amount of redundant data that the join optimization using the intersection filter becomes a good choice.

### (4) Optimization for recursive join and cost model

- We propose an optimization for recursive joins using the difference filter in MapReduce. A recursive join is implemented as an iteration of one join job instead of two jobs including a join job and a difference job. Thanks to the difference filter, we can compute the join of two datasets and the incremental relation in the join job, and thus eliminate the significant overheads of the difference job on each iteration. These overheads consist of re-scanning and re-shuffling all the incremental relations. As a result, our recursive join is processed in the fixed number of jobs (or iterations),  $l$  rather than of  $2 \times l$ . The recursive join implementation is then illustrated by an algorithm in form of pseudo code.
- We provide a cost model for the recursive join. Our recursive join is then proved more efficient than the existing solution through the cost model-based comparison.

### (5) Experimental evaluation

- We deploy MapReduce Hadoop over two different computer clusters to utilize the computing effectively. The clusters are built with virtual machines using Virtualbox and KVM virtualization techniques, namely, first cluster of 15 virtual machines using Virtualbox and second cluster of 15 virtual machines using KVM.
- Experiments of two-way joins and multi-way joins implemented by the different algorithms are supplied.

- Experimental comparisons of the different algorithms for each the join are examined with respect to the intermediate data amount, the total output amount, the total execution time, and especially task timelines.

Our filter designs bring vital benefits that they can be applied to solve popular problems in various fields such as join operation, reconciliation and deduplication, error-correction, etc.

Both the cost models and the experiments show that a join operation using the intersection filter is more efficient than using the other solutions since it significantly reduces redundant data, and thus produces less intermediate data. Moreover, the intersection filter provides an extremely important characteristic for the join cascade in which intermediate join results generated from component joins only contain actual joining data without filtering. These significantly reduce I/O and communication overheads. Although the intersection filter has false positives and an extra cost for the pre-processing job, its efficiency in space-saving and filtering often outweighs these drawbacks.

Besides, this research indicates that joins using the pair of filters or the unpartitioned intersection filter are more efficient than using the partitioned intersection filter. But the partitioned filter-based joins are still much better than the existing join algorithms.

More importantly, this research also improves the general semi-naive algorithm, as well as the evaluation of recursive queries in MapReduce.

Finally, all these contribute to the global scene of optimizing data management for MapReduce applications on large-scale distributed infrastructures.

## 5.2 Discussion and future work

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

### 5.2.1 Two-way and multi-way joins

Together with the popularity of MapReduce for processing large-scale datasets, join algorithms using MapReduce has also received much attention during the past few years. However, most two-way join algorithms are sensitive to data skew that may be due to the bad partitioning function or a large number of tuples with the same join key. They need to be improved to overcome this problem.

Overall, studies of the join operation have largely concentrated on two-way join algorithms. Hence, there are many challenges in evaluating multi-way joins through the existing join algorithms in a shared-nothing environment.

To implement join query, we need to specify an execution plan. In this section, therefore, we discuss some scheduling strategies for the  $n$ -way join query evaluation in MapReduce. First, we should take a simple multi-way join query as follows:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$$

We can implement this query by the following possible execution plans.

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 = (((R_1 \bowtie R_2) \bowtie R_3) \bowtie R_4) \quad (5.1)$$

$$= ((R_1 \bowtie R_2 \bowtie R_3) \bowtie R_4) \quad (5.2)$$



$$= (R_1 \bowtie R_2) \bowtie (R_3 \bowtie R_4) \quad (5.3)$$

$$= (R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4) \quad (5.4)$$

A typical approach is to use a sequence of two-way joins, called a cascade of two-way joins (**C2**), as shown in equation (5.1). In the C2 approach, each join operation needs at least one separate MapReduce job and depends on the output of a previous join operation. In other words, these join operations have to run sequentially. As a result, the C2 approach does not implement the join operations in parallel but each its join operation includes parallel tasks. This form of parallelism is termed intra-operator parallelism.

Similarly, we also have an approach using a cascade of three-way joins (**C3**), as represented in equation (5.2). In this approach, the join operations perform joining three input datasets at the same time that is mentioned in Chapter 3.

Another approach to the above query evaluation is to through parallel two-way join operations (**P2**), as represented in equation (5.3). The join  $R_{12}=(R_1 \bowtie R_2)$  and the join  $R_{34}=(R_3 \bowtie R_4)$  are run simultaneously. The output of the query is the join of  $R_{12}$  and  $R_{34}$ . Hence, the P2 approach implements the join operations in parallel and each join operation has concurrent tasks. This form of parallelism is known as inter-operator parallelism.

The last approach is to employs joining all the relations at the same time (**All-in-One**), as shown in equation (5.4). Joining all relations in one requires only one MapReduce job rather than using multiple jobs as others approaches. Consequently, there are no intermediate results which may save physical space and communication overhead. However, buffering tuples to process the join can easily lead to memory overflow error.

We need to specify the number of MapReduce iterations of each approach. The C2 approach uses three iterations, the C3 approach uses two iterations, the P2 approach uses two iterations, and the All-in-One approach uses one iteration.

From the illustration of the 4-way join, we can generally compute the number of iterations for the typical multi-way join as follow. The approaches 1 and 2 are summarized as a cascade of  $m$ -way joins (**CM**), where  $m$  is the number of inputs of a component join operation. The approach 3 is generalized as parallel  $m$ -way join operations (**PM**).

For the general  $n$ -way join, the number of iterations of the CM is  $(n-1)/(m-1)$ , where  $n$  is the total number of the relations of the multi-way join ( $n \geq 2$ ),  $m$  is the number of the relations of the component join ( $m \geq 2$  and  $m \leq n$ ). In the case of the PM algorithm, the number of iterations is  $(\log_m n)$ .

We can encounter the following challenges when implementing the approaches:

- Choice of join order:

The  $n$ -way join can be processed by combining  $m$ -way joins ( $m \leq n$ ). As a result, there are some combination ways of the component joins for evaluating the query. Each way can have different join tree representations.

For the PM approach, we have Bushy join tree representation [94] as shown in Figure 5.1.

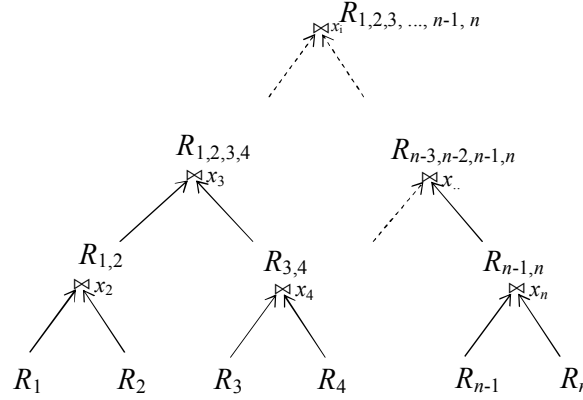
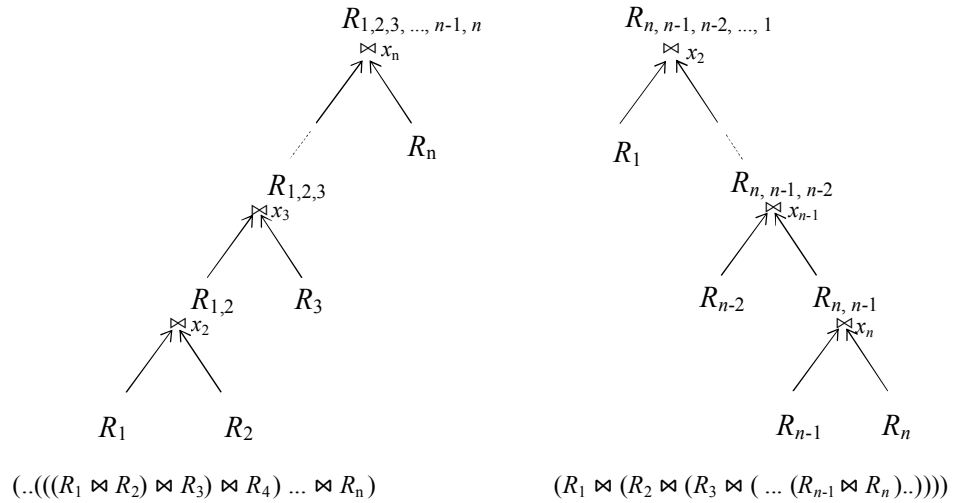


Figure 5.1. A kind of Bushy join trees in PM approach

The Bushy join tree includes a set of both left-deep and right-deep trees. This tree kind performs parallelism of join operations in the query.

For the CM approach, we have two kinds of join tree representations including left-deep trees and right-deep trees [94] as depicted in Figure 5.2.



(a) Left-Deep join trees

(b) Right-Deep join trees

Figure 5.2. Two kinds of join trees in CM approach

The left-deep join trees (Figure 5.2 (a)) are executions of the CM approach in which the combination of the component joins (the  $m$ -way joins) are executed from left to right. Meanwhile, another execution kind of the CM approach that chains the component joins from right to left is called the right-deep join trees (Figure 5.2 (b)). The join trees of the CM approach are also known as linear trees or linear processing trees.

It is very important to note that the number of possible join orders determine the number of join trees. For the  $n$ -way join query, therefore, the number of

left-deep or right-deep trees is calculated as  $n$  factorial ( $n!$ ) and the number of bushy trees is specified by  $(2n - 2)!/(n - 1)!$ .

The problem of choosing an optimal join order that is expected to result in a minimum cost for a query in MapReduce is difficult and even impossible because analyzing all the possible join orders could take a long time and especially many communication overheads. Consequently, a query optimizer cannot perform an exhaustive search and instead uses some heuristics to support the search process. This problem should be considered with respect to sort, merge, and communication overheads.

- Choice of join algorithm:

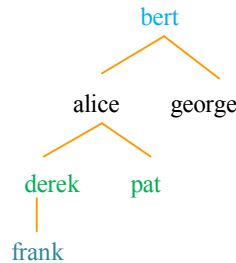
Besides specifying the join order, we need to choose a MapReduce join algorithm for each of the component joins as well. When selecting a join algorithm, the optimizer should take into account factors such as the size of each input relation, indexes and partitions available on each relation, a particular join order, the number of rows to be scanned for each relation in each join order, non-joining data rate, cost model, etc.

For example, we can use the star join algorithm for queries that have one large input dataset and other small datasets. As another instance of the PM approach, we may apply the intersection filter-based join algorithm for the initial joins  $(R_1 \bowtie R_2)$ ,  $(R_3 \bowtie R_4)$ , ...,  $(R_{n-1} \bowtie R_n)$ . The intermediate results, which are indexed and partitioned by the join key, are joined together using the Map-side join algorithm.

The selection of MapReduce join algorithms for a query can even decide to a join order (a join tree). Therefore, the choice of a join order should also take the choice of join algorithms and the communication overheads into consideration. In the future work, we need to further discuss on the query optimizer which uses a good heuristic algorithm for choosing an efficient query execution plan in a MapReduce environment.

### 5.2.2 Recursive joins

Assume that we have the relation  $Know\ K = \{(bert, alice), (bert, george), (alice, derek), (alice, pat), (derek, frank)\}$  and it can be represented as a directed graph which we refer to as the following relation graph.



The tuples correspond to edges in the graph and the unique attribute values correspond to the nodes. A tuple  $(x, y)$  in relation  $K$  becomes an edge from node  $x$  to node  $y$  in the directed graph.

The relation *Friend* ( $F$ ) is built by a recursive join  $F(x, y) = F(x, z) \bowtie K(z, y)$ . A tuple  $(x, y)$  in  $F$  means that there is a non-zero length path from node  $x$  to node  $y$  (the transitive closure (TC) of the relation  $K$ ). In general, any linearly recursive query can be expressed as a transitive closure, preceded or followed by relational algebra operations [95]. Therefore, the number of iterations to evaluate a recursive join is the longest path length in the graph-1, called the depth of the transitive closure ( $l$ ). For the example, the number of iterations or the number of MapReduce jobs is 2 ( $l=3-1$ ).

We can consider two main performance aspects for a MapReduce operation, namely the amount of data in each job and the number of jobs done. We solved reducing the amount of data in each job and the number of jobs on each round. There remains an opportunity for us to reduce the number of jobs by reducing the number of iterations. With the same input data and algorithm in MapReduce environment, computation of an operation becomes better if it has less jobs.

From the above arguments, we find here other challenges that need to be considered to further improvement of the recursive join evaluation in MapReduce. Four ideas make these possible: (1) *minimize the number of iterations*, (2) *solve a recursive join in an unbalanced graph*, (3) *handle data skew in recursive joins*, and (4) *handle small increment relations in every join performed*. For clarity, it is necessary to go into some detail here.

### 5.2.2.1 Minimize the number of iterations

We try to reduce the number of iterations as much as possible. This can reduce the number of MapReduce jobs for the evaluation. As a result, we can avoid rescanning input data and generating much intermediate data as well as transferring the data over the network.

In fact, this problem is not new in databases. Several algorithms have been presented in the literature to efficiently process the transitive closure (TC) of a relation [80][23][24][96]. Afrati and Ullman [19] have also made a comparison of Smart TC and related algorithms, where they examine the relative efficiency, in terms of data-volume cost. However, these algorithms are not suitable in MapReduce environment because they can reduce the number of round iterations but they do not actually reduce the number of join jobs in MapReduce. For instance, Smart and Logarithmic TC algorithms [23][24] reduces the number of iterations by computing more of the transitive closure in each iteration and so it also requires to execute more of join jobs in each iteration. They use  $(l/n)$  iterations, where  $l$  is the longest path length in the relation graph - 1 and  $n$  is the number of join jobs in each iteration. Indeed the total of join jobs in these algorithms is still  $l$  jobs. Consequently, their improvement is useless in MapReduce.

**Example 5.1:** compute a transitive closure  $(\Delta F_{i-1} \bullet K \bullet K)$

We use an iteration of two-way join jobs to evaluate the transitive closure:  $((\Delta F_{i-1} \bowtie K) \bowtie K)$

Input  $(a, b) \bullet (b, c) \bullet (c, d)$  is compiled as:

$(a, b) \bowtie (b, c) \Rightarrow$  the join result:  $r_1 = (a, b), (b, c), (a, c)$

$r_1 \bowtie (c, d) \Rightarrow$  the final join result:  $r_2 = (a, b), (b, c), (a, c), (c, d), (b, d), (a, d)$ .

For each iteration, we have to use the two join jobs to compute the transitive closure.

The recent results of Afrati and Ullman [12][7] showed a interesting way to compute the transitive closure by recursive doubling. To implement the way, there will be four groups of tasks consisting of two groups of join tasks and two groups of deduplication tasks. A limitation of this solution is long-running recursive tasks that may increase risk for failures. In addition, its failure recovery mechanisms are complex and have not been directly supported in Hadoop. Moreover, large amounts of data are transferred back and forth among the groups.

In this case, our solution is proposed as the following. We will adapt the solution of our three-way join based on the intersection filters [5] in each iteration to fit evaluating recursive joins in Mapreduce. Hence our algorithm only requires  $\lceil \log_2(l+1) \rceil$  iterations as well as  $\lceil \log_2(l+1) \rceil$  join jobs. The modification of the three-way join is defined by discovering three transitive closures on each round iteration instead of only one as the three-way join. That requires recomputing the amount of data of each input tuple distributed to corresponding reducers, deriving all the possible transitive closures as well as showing its cost model as compared with existing solutions.

### 5.2.2.2 Solve a recursive join in an unbalanced graph

We address the problem of a recursive join in an *unbalanced graph*. This recursive join will be executed with a large amount of MapReduce join jobs. It is interesting to think about an algorithm to partition the graph into a set of subgraphs and evaluate them in parallel. The final result of the recursive join is transitive closures among the subgraphs.

### 5.2.2.3 Handle data skew in recursive joins

We consider the problem of a large number of tuples that are sent to the same reducer.

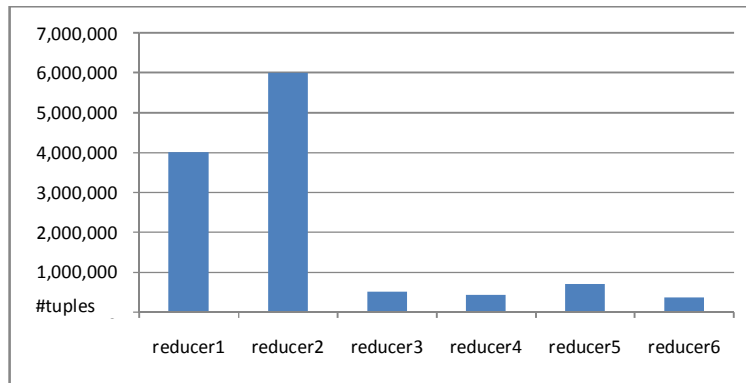


Figure 5.3. Data skew in recursive join

Data skew [97] is an asymmetry in the distribution of tuples to reducers. This leads to some nodes working in overcrowding situation. It is noted that data skew in the recursive join includes not only data skew with the different join key but also data skew with the same join key. For example, we have 5,000,000 tuples in which 4,000,000 tuples are sent to the same *reducer1* even though they may have the different join keys. Another example, we have 7,000,000 tuples including 6,000,000 tuples with the same join key that must be sent the same reducer 2. There exist

several solutions [98][99][100] to address data skew for the first example but not the second example. Therefore, we have to address the problem of efficiently processing MapReduce join jobs with join reducer tasks over skewed data with tuples having the same join key. Without any optimization, the actual task completion time for each join reducer differs significantly and the overall system performance is largely affected by the long running tasks.

In this case, we will implement a hybrid join algorithm which is a combination of the reduce-side join, the hash join, and Reducer Input Cache (RIC) with the ability to handle skew. In general, we consider two arbitrary big datasets that have not been sorted/partitioned yet. For the first job, one dataset  $S$  will be partitioned and these partitions will be cached at reducers. For the second job, another dataset  $R$  will be mapped into pairs and sent to the reducers. Each pair of  $R$  will be joined on fly with one corresponding partition of  $S$ .

#### 5.2.2.4 Handle small increment relations in joins performed

We focus on the problem of intermediate results of a recursive join. They are incremental relations in each iteration that may be small relations. In this case, we should find a good choice for storing the intermediate results to avoid rescanning and reshuffling them many times over the network. In this case, we may re-implement cache technology to be able to merge/append small cache outputs into one bigger file without rewrite all small caches as the existing solution.

Solving all these challenges aims to optimize recursive joins. It becomes more important when the recursive joins is a decisive factor for evaluating recursive queries in MapReduce.

### 5.2.3 Query language for NoSQL databases

For over forty years, the relational database (RDBMS), in which data are structured into tables or relations that are easily restructured for accessing data in different ways, has been the dominant model for database management. Together with RDBMS, SQL has become a standard language supported by most relational database systems. SQL provides a complete data-definition language, including the ability to create relations with the specified attribute types, and the ability to define integrity constraints on the data [101].

However, as information technology becomes ever more prevalent in nearly every aspect of our lives, the vast amount of data generated and stored continues to grow at an astounding rate, especially with social network applications today. This arises new challenges for data management, most notably scalability of storage, flexible data model, non-relational, and dynamic and implicit schema for collections of documents with varying structure. The relational database technologies have not kept pace with these changes. Although there have also been many attempts to extend the technologies (horizontal and vertical sharding, distributed caching and data denormalization), these tactics nullify key benefits of the relational model while increasing total system cost and complexity.

A major trend over the last few years has seen that NoSQL, Non-relational, "Cloud", or "Document" databases is an alternative model for data management in order to match the new challenges. In a NoSQL database, there are no a fixed schema and may be no joins. An RDBMS "scales up" by getting faster hardware and adding

memory. NoSQL, on the other hand, can take advantage of "scaling out". Scaling out refers to spreading the load over many commodity systems. This is the component of NoSQL that makes it an inexpensive solution for large datasets. NoSQL is designed for distributed data stores where very large scale data needs.

NoSQL databases have grown up and prove themselves worthy, such as Google's BigTable, Amazon's Dynamo, Facebook's Cassandra, 10gen's MongoDB, Apache's CouchDB, etc. that are now mature. Even Oracle, a company which is known by its RDBMS, also launched a product called Oracle NoSQL Database, a multi-terabyte distributed key/value pair storage [102].

NoSQL's initial success and the explosion of related modern applications have led to an increase in the amount of new document databases and data formats. However, each such database system creates a new query language in a framework or an API that only works with a single document store. At the moment the new database systems have no query language (completely depending on the highly specialized map/reduce approach), while others like (e.g. MongoDB or Cassandra) have rudimentary and proprietary query languages. The query methods tend to be very low-level and must now be manually coded into the application by the programmer instead of being handled automatically by the database engine. In other words, document databases have their own proprietary and incompatible query methods, meaning that it is hard to move an application from one database engine to another. This caused many difficulties for developers in implementing or integrating systems because it still lacks a common query tool that attempts to query multiple document databases. As a result, this has hindered the popularity of document databases.

For all reasons, developers generally agree that a standard language will be good for the NoSQL space and not too early. And this problem becomes an attractive topic that has been discussed so much in order to give out a feasible solution. We are therefore aiming to a logic-based abstract approach to a high-level standard query language for document databases. It is called Datalog-based Document Query Language (DLogQL) including the following features:

- Provide a standard abstract query language for document databases. It is a powerful expressive language supporting relational operations and recursion.
- Give an abstract-level logical query language whose semantics is a subset of Datalog and let syntax specification at an abstract level. As a result, many sub-languages may be derived from DLogQL. Each sub-language will be represented in a popular data format (e.g. key-value data model). This aims to adapt to the continuous change of modern application model and data format and close the gap among query language, programming language and the database.
- Extend the well-known Datalog language, including abstract-level language, ability to query scalable document databases and scalable Datalog over Hadoop (an efficient parallel implementation of Datalog on the MapReduce framework).
- A mediation solution for document databases and ability to model documents in a document database as a deductive data model.

DLogQL towards a general logical query language for databases aims to adapt to the continuous change of modern application model and data format.

# Bibliography

- [1] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, pp. 137–150.
- [2] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel data processing with MapReduce: a survey,” *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012.
- [3] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, “A comparison of join algorithms for log processing in MaPReduce,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, New York, NY, USA, 2010, pp. 975–986.
- [4] T. Lee, K. Kim, and H.-J. Kim, “Join processing using Bloom filter in MapReduce,” in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, New York, NY, USA, 2012, pp. 100–105.
- [5] T.-C. Phan, L. d’ Orazio, and P. Rigaux, “Toward Intersection Filter-based Optimization for Joins in MapReduce,” in *Proceedings of the 2Nd International Workshop on Cloud Intelligence*, New York, NY, USA, 2013, pp. 2:1–2:2.
- [6] F. N. Afrati and J. D. Ullman, “Optimizing joins in a map-reduce environment,” in *Proceedings of the 13th International Conference on Extending Database Technology*, New York, NY, USA, 2010, pp. 99–110.
- [7] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, “Map-reduce extensions and recursive queries,” in *Proceedings of the 14th International Conference on Extending Database Technology*, New York, NY, USA, 2011, pp. 1–8.
- [8] S. Blanas, Y. Li, and J. 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*, New York, NY, USA, 2011, pp. 37–48.
- [9] M. A. H. Hassan and M. Bamha, “Semi-join computation on distributed file systems using map-reduce-merge model,” in *Proceedings of the 2010 ACM Symposium on Applied Computing*, New York, NY, USA, 2010, pp. 406–413.
- [10] T. White, *Hadoop: the definitive guide 2012*. Farnham: O’Reilly, 2012.
- [11] C. Lam, *Hadoop in Action*, 1st ed. Manning Publications, 2010.
- [12] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman, “Cluster Computing, Recursion and Datalog,” in *Datalog Reloaded*, O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds. Springer Berlin Heidelberg, 2011, pp. 120–144.
- [13] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient Iterative Data Processing on Large Clusters,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 285–296, Sep. 2010.



- 
- [14] M. Shaw, P. Koutris, B. Howe, and D. Suciu, "Optimizing Large-Scale Semi-Naïve Datalog Evaluation in Hadoop," in *Datalog in Academia and Industry*, P. Barceló and R. Pichler, Eds. Springer Berlin Heidelberg, 2012, pp. 165–176.
- [15] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.
- [16] T. Bolognesi and S. A. Smolka, "Fundamental Results for the Verification of Observational Equivalence: A Survey," in *Protocol Specification, Testing and Verification VII, Proceedings of the IFIP WG6.1 Seventh International Conference on Protocol Specification, Testing and Verification, Zurich, Switzerland, 5-8 May, 1987*, 1987, pp. 165–179.
- [17] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd edition. Boston: Addison Wesley, 2006.
- [18] S. Ceri, G. Gottlob, and L. Tanca, "What You Always Wanted to Know About Datalog (And Never Dared to Ask)," *IEEE Trans. on Knowl. and Data Eng.*, vol. 1, no. 1, pp. 146–166, Mar. 1989.
- [19] F. N. Afrati and J. D. Ullman, "Transitive Closure and Recursive Datalog Implemented on Clusters," in *Proceedings of the 15th International Conference on Extending Database Technology*, New York, NY, USA, 2012, pp. 132–143.
- [20] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," in *On Knowledge Base Management Systems*, M. L. Brodie and J. Mylopoulos, Eds. Springer New York, 1986, pp. 165–178.
- [21] I. Balbin and K. Ramamohanarao, "A generalization of the differential approach to recursive query evaluation," *The Journal of Logic Programming*, vol. 4, no. 3, pp. 259–262, Sep. 1987.
- [22] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract)," in *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, New York, NY, USA, 1986, pp. 1–15.
- [23] Y. E. Ioannidis, "On the Computation of the Transitive Closure of Relational Operators," in *Proceedings of the 12th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1986, pp. 403–411.
- [24] P. Valduriez and H. Boral, "Evaluation of Recursive Queries Using Join Indices," presented at the Expert Database Conf., 1986.
- [25] S. Warshall, "A Theorem on Boolean Matrices," *J. ACM*, vol. 9, no. 1, pp. 11–12, Jan. 1962.
- [26] H. S. Warren, Jr., "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Commun. ACM*, vol. 18, no. 4, pp. 218–220, Apr. 1975.
- [27] S. Ramesh, O. Papapetrou, and W. Siberski, "Optimizing Distributed Joins with Bloom Filters," in *Distributed Computing and Internet Technology*, M. Parashar and S. K. Aggarwal, Eds. Springer Berlin Heidelberg, 2009, pp. 145–156.
- [28] Z. Sun, J. Shen, and J. Yong, "A Novel Approach to Data Deduplication over the Engineering-oriented Cloud Systems," *Integr. Comput.-Aided Eng.*, vol. 20, no. 1, pp. 45–57, Jan. 2013.
- [29] J. W. Byers, J. Considine, M. Mitzenmacher, and S. Rost, "Informed Content Delivery Across Adaptive Overlay Networks," *IEEE/ACM Trans. Netw.*, vol. 12, no. 5, pp. 767–780, Oct. 2004.

- 
- [30] G. Cormode and S. Muthukrishnan, "What's New: Finding Significant Differences in Network Data Streams," *IEEE/ACM Trans. Netw.*, vol. 13, no. 6, pp. 1219–1232, Dec. 2005.
- [31] D. Eppstein and M. T. Goodrich, "Straggler Identification in Round-Trip Data Streams via Newton's Identities and Invertible Bloom Filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 2, pp. 297–306, 2011.
- [32] L. Y. Rusin, E. V. Lyubetskaya, K. Y. Gorbunov, and V. A. Lyubetsky, "Reconciliation of Gene and Species Trees," *BioMed Research International*, vol. 2014, p. e642089, Mar. 2014.
- [33] Y. Zheng, T. Wu, and L. Zhang, "Reconciliation of Gene and Species Trees With Polytomies," *arXiv:1201.3995 [q-bio]*, Jan. 2012.
- [34] O. Hassanzadeh and R. J. Miller, "Creating Probabilistic Databases from Duplicated Data," *The VLDB Journal*, vol. 18, no. 5, pp. 1141–1166, Oct. 2009.
- [35] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate Record Detection: A Survey," *IEEE Trans. on Knowl. and Data Eng.*, vol. 19, no. 1, pp. 1–16, Jan. 2007.
- [36] F. Panse, M. van Keulen, and N. Ritter, "Indeterministic Handling of Uncertain Decisions in Deduplication," *J. Data and Information Quality*, vol. 4, no. 2, pp. 9:1–9:25, Mar. 2013.
- [37] T. Mandel and J. Mache, "Practical Error Correction for Resource-constrained Wireless Networks: Unlocking the Full Power of the CRC," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, New York, NY, USA, 2013, pp. 3:1–3:14.
- [38] N. Chilamkurti, J. H. Park, and N. Kumar, "Concurrent Multipath Transmission with Forward Error Correction Mechanism to Overcome Burst Packet Losses for Delay-sensitive Video Streaming in Wireless Home Networks," *Multimedia Tools Appl.*, vol. 65, no. 2, pp. 201–220, Jul. 2013.
- [39] M. Burke, B. Amento, and P. Isenhour, "Error Correction of Voicemail Transcripts in SCANMail," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2006, pp. 339–348.
- [40] E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Commun. ACM*, vol. 13, no. 6, pp. 377–387, Jun. 1970.
- [41] E. F. Codd, "Relational Completeness of Data Base Sublanguages," In: R. Rustin (ed.): *Database Systems: 65-98*, Prentice Hall and IBM Research Report RJ 987, San Jose, California, 1972.
- [42] K.-L. Tan and H. Lu, "A Note on the Strategy Space of Multiway Join Query Optimization Problem in Parallel Systems," *SIGMOD Rec.*, vol. 20, no. 4, pp. 81–82, Dec. 1991.
- [43] S. Idreos, E. Liarou, and M. Koubarakis, "Continuous Multi-way Joins over Distributed Hash Tables," in *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, New York, NY, USA, 2008, pp. 594–605.
- [44] C. Ordonez, "Optimizing Recursive Queries in SQL," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2005, pp. 834–839.

- 
- [45] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. New York: Springer Science, 2011.
  - [46] “Apache Hadoop.” [Online]. Available: <http://hadoop.apache.org/docs/stable/>. [Accessed: 30-Jan-2013].
  - [47] “The Disco Project.” [Online]. Available: <http://discoproject.org/>. [Accessed: 30-Jan-2013].
  - [48] “Sector/Sphere: High Performance Distributed Data Storage and Processing.” [Online]. Available: <http://sector.sourceforge.net/>. [Accessed: 30-Jan-2013].
  - [49] A. Holmes, *Hadoop in Practice*, 1 edition. Manning Publications, 2012.
  - [50] J. Lin, S. Konda, and S. Mahindrakar, *Low-Latency, High-Throughput Access to Static Global Resources within the Hadoop Framework*. 2009.
  - [51] D. DeWitt and J. Gray, “Parallel Database Systems: The Future of High Performance Database Systems,” *Commun. ACM*, vol. 35, no. 6, pp. 85–98, Jun. 1992.
  - [52] P. Valduriez and G. Gardarin, “Join and Semijoin Algorithms for a Multiprocessor Database Machine,” *ACM Trans. Database Syst.*, vol. 9, no. 1, pp. 133–161, Mar. 1984.
  - [53] E. Babb, “Implementing a Relational Database by Means of Specialized Hardware,” *ACM Trans. Database Syst.*, vol. 4, no. 1, pp. 1–29, Mar. 1979.
  - [54] M. Bamha and G. Hains, “An Efficient Equi-semi-join Algorithm for Distributed Architectures,” in *Computational Science – ICCS 2005*, V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. J. Dongarra, Eds. Springer Berlin Heidelberg, 2005, pp. 755–763.
  - [55] P. A. Bernstein, N. Goodman, E. Wong, C. L. Reeve, and J. B. Rothnie, Jr., “Query Processing in a System for Distributed Databases (SDD-1),” *ACM Trans. Database Syst.*, vol. 6, no. 4, pp. 602–625, Dec. 1981.
  - [56] L. F. Mackert and G. M. Lohman, “R\* Optimizer Validation and Performance Evaluation for Distributed Queries,” in *Proceedings of the 12th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1986, pp. 149–159.
  - [57] M. W. Blasgen and K. P. Eswaran, “Storage and access in relational data bases,” *IBM Syst. J.*, vol. 16, no. 4, pp. 363–377, Dec. 1977.
  - [58] M. G. Gouda and U. Dayal, “Optimal Semijoin Schedules for Query Processing in Local Distributed Database Systems,” in *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1981, pp. 164–175.
  - [59] J. K. Mullin, “Optimal semijoins for distributed database systems,” *IEEE Transactions on Software Engineering*, vol. 16, no. 5, pp. 558–560, May 1990.
  - [60] “Facebook Reports Fourth Quarter and Full Year 2013 Results - Facebook.” [Online]. Available: <http://investor.fb.com/releasedetail.cfm?ReleaseID=821954>. [Accessed: 10-Feb-2014].
  - [61] K. Bratbergsengen, “Hashing Methods and Relational Algebra Operations,” in *Proceedings of the 10th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1984, pp. 323–333.
  - [62] L. Michael, W. Nejd, O. Papapetrou, and W. Siberski, “Improving distributed join efficiency with extended bloom filter operations,” in *21st International*

- Conference on Advanced Information Networking and Applications, 2007. AINA '07*, 2007, pp. 187–194.
- [63] L. L. Gremillion, “Designing a Bloom Filter for Differential File Access,” *Commun. ACM*, vol. 25, no. 9, pp. 600–604, Sep. 1982.
- [64] M. V. Ramakrishna, “Practical performance of Bloom filters and parallel free-text searching,” *Commun. ACM*, vol. 32, no. 10, pp. 1237–1239, Oct. 1989.
- [65] J. K. Mullin, “A Second Look at Bloom Filters,” *Commun. ACM*, vol. 26, no. 8, pp. 570–571, Aug. 1983.
- [66] M. Mitzenmacher, “Compressed Bloom Filters,” *IEEE/ACM Trans. Netw.*, vol. 10, no. 5, pp. 604–612, Oct. 2002.
- [67] S. Cohen and Y. Matias, “Spectral Bloom Filters,” in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2003, pp. 241–252.
- [68] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal, “The Bloomier Filter: An Efficient Data Structure for Static Support Lookup Tables,” in *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, PA, USA, 2004, pp. 30–39.
- [69] A. Kumar, J. (Jim) Xu, L. Li, and J. Wang, “Space-code Bloom Filter for Efficient Traffic Flow Measurement,” in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, New York, NY, USA, 2003, pp. 167–172.
- [70] A. Kirsch and M. Mitzenmacher, “Distance-Sensitive Bloom Filters,” in *Proc. Eighth Workshop Algorithm Eng. and Experiments (ALENEX '06)*, 2006.
- [71] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, “Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol,” *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jun. 2000.
- [72] M. T. Goodrich and M. Mitzenmacher, “Invertible Bloom Lookup Tables,” *arXiv:1101.2245 [cs]*, Jan. 2011.
- [73] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, “The Dynamic Bloom Filters,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, Jan. 2010.
- [74] A. Broder and M. Mitzenmacher, “Network Applications of Bloom Filters: A Survey,” *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.
- [75] A. Kirsch and M. Mitzenmacher, “Less hashing, same performance: building a better bloom filter,” in *Proceedings of the 14th conference on Annual European Symposium - Volume 14*, London, UK, UK, 2006, pp. 456–467.
- [76] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, “MRShare: sharing across multiple queries in MapReduce,” *Proc. VLDB Endow.*, vol. 3, no. 1–2, pp. 494–505, Sep. 2010.
- [77] “Oracle VM VirtualBox.” [Online]. Available: <https://www.virtualbox.org/>. [Accessed: 05-Jun-2013].
- [78] “pumadatasets - Faraz Ahmad.” [Online]. Available: <https://sites.google.com/site/farazahmad/pumadatasets>. [Accessed: 25-May-2014].
- [79] “Main Page - KVM.” [Online]. Available: [http://www.linux-kvm.org/page/Main\\_Page](http://www.linux-kvm.org/page/Main_Page). [Accessed: 27-May-2014].

- 
- [80] R. Agrawal and H. V. Jagadish, "Direct Algorithms for Computing the Transitive Closure of Database Relations," in *Proceedings of the 13th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1987, pp. 255–266.
  - [81] H. Lu, "New Strategies for Computing the Transitive Closure of a Database Relation," in *Proceedings of the 13th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1987, pp. 267–274.
  - [82] H. Lu, K. P. Mikkilineni, and J. P. Richardson, "Design and Evaluation of Algorithms to Compute the Transitive Closure of a Database Relation," in *Proceedings of the Third International Conference on Data Engineering*, Washington, DC, USA, 1987, pp. 112–119.
  - [83] G. Malewicz, M. H. Austern, A. J. . Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, pp. 135–146.
  - [84] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "The HaLoop Approach to Large-scale Iterative Data Analysis," *The VLDB Journal*, vol. 21, no. 2, pp. 169–190, Apr. 2012.
  - [85] J. D. Ullman, *Principles of Database and Knowledge-base Systems, Vol. I*. New York, NY, USA: Computer Science Press, Inc., 1988.
  - [86] F. Deng and D. Rafiei, "Approximately Detecting Duplicates for Streaming Data Using Stable Bloom Filters," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2006, pp. 25–36.
  - [87] C.-H. Lee and C.-W. Chung, "An approximate duplicate elimination in RFID data streams," *Data & Knowledge Engineering*, vol. 70, no. 12, pp. 1070–1087, Dec. 2011.
  - [88] G. Koloniari, N. Ntarmos, E. Pitoura, and D. Souravlias, "One is Enough: Distributed Filtering for Duplicate Elimination," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, New York, NY, USA, 2011, pp. 433–442.
  - [89] D. Eppstein, M. T. Goodrich, F. Uyeda, and G. Varghese, "What's the Difference?: Efficient Set Reconciliation Without Prior Context," *SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 4, pp. 218–229, Aug. 2011.
  - [90] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.
  - [91] "SHA-3," *Wikipedia, the free encyclopedia*. 15-Jun-2014.
  - [92] "Cryptographic hash function," *Wikipedia, the free encyclopedia*. 15-Jun-2014.
  - [93] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and Network Applications of Dynamic Bloom Filters," in *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, 2006, pp. 1–12.
  - [94] Y. E. Ioannidis and Y. C. Kang, "Left-deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1991, pp. 168–177.

- 
- [95] H. V. Jagadish, R. Agrawal, and L. Ness, "A Study of Transitive Closure As a Recursion Mechanism," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1987, pp. 331–344.
- [96] Y. E. Ioannidis and R. Ramakrishnan, "Efficient Transitive Closure Algorithms," in *Proceedings of the 14th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1988, pp. 382–394.
- [97] Y. Kwon, K. Ren, M. Balazinska, and B. Howe, "Managing Skew in Hadoop," *IEEE Data Eng. Bull.*, vol. 36, no. 1, pp. 24–33, 2013.
- [98] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handling Data Skew in MapReduce," presented at the CLOSER, 2011, pp. 574–583.
- [99] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing Reducer Skew in MapReduce Workloads Using Progressive Sampling," in *Proceedings of the Third ACM Symposium on Cloud Computing*, New York, NY, USA, 2012, pp. 16:1–16:14.
- [100] Y. Gan, X. Meng, and Y. Shi, "Processing Online Aggregation on Skewed Data in Mapreduce," in *Proceedings of the Fifth International Workshop on Cloud Data Management*, New York, NY, USA, 2013, pp. 3–10.
- [101] S. Abiteboul, R. Hull, and V. Vianu, Eds., *Foundations of Databases: The Logical Level*, 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.
- [102] "Oracle NoSQL Database (Version: 12cR1.3.0.9 Enterprise Edition, 2014-05-02 11:52:34 UTC." [Online]. Available: <http://docs.oracle.com/cd/NOSQL/html/index.html>. [Accessed: 09-Jun-2014].