A Theoretical and Experimental Comparison of Filter-Based Equijoins in MapReduce

Thuong-Cang Phan[1(✉)], Laurent d'Orazio[1], and Philippe Rigaux[2]

[1] Blaise Pascal University, CNRS-UMR 6158-LIMOS, Clermont-Ferrand, France
{ThuongCang.Phan,Laurent.Dorazio}@isima.fr
[2] CNAM, CEDRIC, Paris, France
Philippe.Rigaux@cnam.fr

Abstract. MapReduce has become an increasingly popular framework for large-scale data processing. However, complex operations such as *joins* are quite expensive and require sophisticated techniques. In this paper, we review state-of-the-art strategies for joining several relations in a MapReduce environment and study their extension with *filter-based approaches*. The general objective of filters is to eliminate non-matching data as early as possible in order to reduce the I/O, communication and CPU costs. We examine the impact of systematically adding filters as early as possible in MapReduce join algorithms, both analytically with cost models and practically with evaluations. The study covers binary joins, multi-way joins and recursive joins, and addresses the case of large inputs that gives rise to the most intricate challenges.

Keywords: Big data · Cloud computing · Big data analysis · MapReduce · Equijoin · Bloom filter · Intersection Bloom filter

1 Introduction

Since the advent of applications that propose 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 and Facebook, representative of those two distinct trends, have developed for their own needs large-scale data processing platforms. These platforms combine an infrastructure based on millions of servers, data repositories where the least collection size is measured in Petabytes, and finally data processing software products that massively exploit distributed computing and batch processing to scale at the required level of magnitude. Although the Web is a primary source of information production, Big Data issues can now be generalized to other areas that continuously collect data and attempt to make sense of it. Sensors incorporated in electronic devices, satellite images, web server logs, bioinformatics, are considered as gold mines of information that just wait for the processing power to be available, reliable, and apt at evaluating complex analytic algorithms.

A. Hameurlain et al. (Eds.): TLDKS XXV, LNCS 9620, pp. 33–70, 2016.
DOI: 10.1007/978-3-662-49534-6_2

The MapReduce programming model [13] has become a standard for processing and analyzing large datasets in a massively parallel manner. Its success comes from both its simplicity and 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 over 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 *a priori* not suited for operations with multiple inputs.

One of the most representative such operations are *joins*. A join combines related tuples from datasets on different column schemes and thus raises at a generic level the problem of combining several data sources with a programming framework initially designed for scanning, processing and grouping a single input. Join is a basic building block used in many sophisticated data mining algorithms, and its optimization is essential to ensure efficient data processing at scale.

In the present paper we provide a systematic study of joins with *filters* for early removal of non-participating tuples from the input datasets. As known for a long time in the classical context of relational databases, early elimination of useless data is a quite effective technique to reduce the IO, CPU and communication costs of data processing algorithms. The approach can be transposed in distributed systems in general, and to MapReduce frameworks in particular.

We focus on equijoins, and examine state-of-the-art algorithms for two-way joins, multi-way joins and recursive joins. We compare, analytically and experimentally, the benefit that can be expected by introducing filters as early as possible in the data processing workflow. Our result put the research contributions in this field in a coherent setting and clarifies the stakes of combining several inputs with MapReduce.

The rest of the paper is organized as follows. Section 2 summarizes the background of the basic join operation, recalls the essentials of the MapReduce framework and intersection filters, and positions our paper with respect to related work. Section 3 presents filter-based equijoins in MapReduce. We examine two-way joins, multi-way joins, and recursive joins. Section 4 analyzes the algorithms and introduces cost models. The evaluation environment and the results are reported in Sect. 5. Finally, Sect. 6 concludes and discusses future work.

Table 1 provides a quick reference to the algorithms abbreviations used throughout the text.

2 Background and Related Work

2.1 Join Operation

A join combines tuples from one or several relations according to some join condition[1]. A tuple that participates to the result (and therefore satisfies the join condition) is called a *matching tuple* in the following. Non-matching tuples can simply be ignored from the join processing workflow, a fact that calls for their early elimination. We distinguish the following types of joins:

[1] Our study only considers conditions is based an equality operator (=), or *equijoins*.

Table 1. List of abbreviations

Abbreviation	Algorithm
IFBJ	Intersection filter-based join
BJ	Bloom join
RSJ	Reduce-side join
3WJ	Three-way join proposed by Afrati and Ullman [3]
CJ-IFBJ	Chain join using an intersection filter-based join cascade
CJ-BJ	Chain join using a Bloom join (BJ) cascade
CJ-RSJ	Chain join using a reduce-side join (RSJ) cascade
OCJ-2WJ	Optimized chain join using a two-way join cascade
OCJ-3WJ	Optimized chain join using a three-way join (3WJ) cascade
REJ-SHAW	Recursive join using Shaw's approach
REJ-FB	Recursive join using a filter-based approach

- **Two-way join.** Given two datasets R and L, a two-way join denotes the pairs of tuples $r \in R$ and $l \in L$, such that $r.k_1 = l.k_2$ where k_1 and k_2 are join columns in R and L, respectively. The standard notation is:

$$R \bowtie_{k_1=k_2} L$$

  Notation: In order to simplify notations, we will often assume that join keys are known from the context, and will use the abbreviated form $R \bowtie L$.
- **Multi-way join** [35]**.** Given n datasets $R_1, R_2, \ldots, R_n$, we define a multi-way join as a pairwise combination of two-way joins:

$$R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots \bowtie R_n$$

  Considering only pairwise combination is a restriction: this subclass is sometimes called a *chain join* in the literature.
- **Recursive join** [17,29]**.** Given a relation $K(x, y)$ encoding a graph, a recursive join computes the transitive closure of K. It requires an initialization, and an iteration (until a fixpoint occurs):

$$\begin{cases} \text{(Initialization)} & A(x,y) = K(x,y) \\ \text{(Iteration)} & A(x,y) = A(x,z) \bowtie K(z,y) \end{cases}$$

We use the following running example: a *user* dataset $R(uid, uname, location)$, a *log* dataset $L(uid, event, logtime)$ and an *acquaintance* dataset $K(uid1, uid2)$. These datasets illustrate the following searches.

- Q_1 - Two-way join. Find the names and events of all users who logged an event before 19/06/2015.

$$A_1(uname, event) = \pi_{uname,event}(R \bowtie \sigma_{logtime<19/06/2015}(L))$$

- Q_2 - Multi-way join. Find the log events of all users known by Cang

$$A_2(uid, event, logtime) =$$

$$\pi_L(\sigma_{uname='Cang'}(R) \bowtie_{uid=uid1} K \bowtie_{uid2=uid} L)$$

- Q_3 - Recursive join. List the ids of all connected to Philippe.

$$\begin{cases} \text{(Initialization)} & A_3(id) = \pi_{uid}(\sigma_{uname='Philippe'}(R)) \\ \text{(Iteration)} & A_3(id) = \pi_{uid2}(K \bowtie_{uid1=id} A_3) \end{cases}$$

2.2 MapReduce

MapReduce [13] is a parallel and distributed programming model apt at running on computer clusters that scale to thousands of nodes in a fault-tolerant manner. MapReduce usage has become widespread since Google first introduced it in 2004. It allows users to concentrate only on designing their data operations regardless of the distributed aspects of the execution.

A MapReduce job consists of two distinct phases, namely, the *map phase* and the *reduce phase*. Each phase executes a user-defined function on a key-value pair. The user-defined map function **(M)** takes an input pair (k_1, v_1) and outputs a list of intermediate key/value pairs $\langle (k_2, v_2) \rangle$.

$$(k_1, v_1) \xrightarrow{map} list(k2, v2)$$

The intermediate values associated with the same key k_2 are grouped by the framework and then passed to the reduce function which aggregates the values.

$$(k_2, list(v_2)) \xrightarrow{reduce} v_3$$

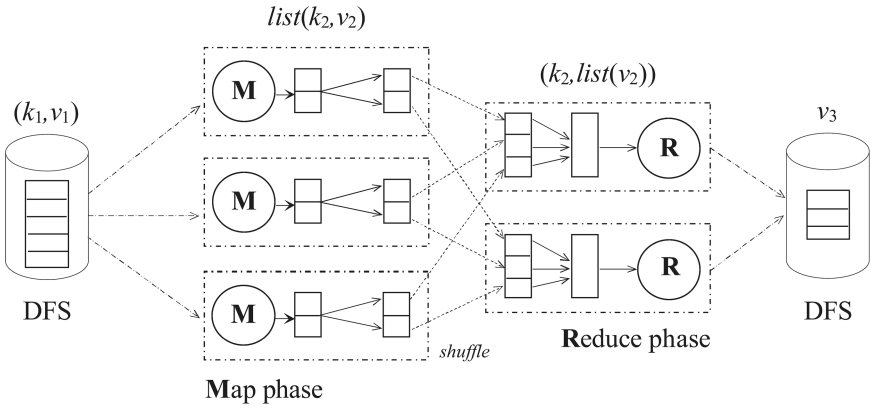


Fig. 1. MapReduce distributed execution

As illustrated by Fig. 1, a typical MapReduce job is executed across multiple nodes. During the map phase, each map task reads a subset (called "*split*") of one input dataset, and applies the map function for each key/value pair of the split. The framework takes care of grouping intermediate data and sends them to the reducer nodes, a communication-intensive process called *shuffling*. Each reduce task collects the intermediate key/value pairs from all the map tasks, sorts/merges the data with the same key, and calls the reduce function to generate the final results.

MapReduce is designed to process a single dataset. Combining several inputs with a MapReduce framework is intricate. The problem has mostly been studied for joins.

2.3 Bloom Filters

A *Bloom filter* (BF) [9] is a space-efficient randomized data structure used for testing membership in a set with a small rate of false positives.

A variant of a Bloom filter is *Intersection Bloom filter* [30], denoted $IBF(S_1, S_2)$, is a probabilistic data structure designed to represent the intersection of sets S_1 and S_2, and check membership in the intersection set. To achieve this, it computes the intersection of the Bloom filters $BF(S_1)$ and $BF(S_2)$. In join processing, matching a join key v against the intersection filter allows to decide (up to the false positive probability) whether it belongs to the shared join keys. The false positive probability of the intersection filter is estimated as f_I representing one of the probabilities of different approaches to the filter [30].

Extended Intersection filter [30] (EIF) is developed from the intersection Bloom filter. The EIF is a filter built on join key columns $k_1, k_2, \ldots, k_m$ of datasets $R_1, R_2, \ldots, R_m$. It consists of Bloom filters hashed on the key columns, $BF_1(R_1.k_1), BF_2(R_2.k_2 \cap R_3.k_2), \ldots, BF_m(R_m.k_m)$. The membership test takes a tuple $t(k_1, k_2, \ldots, k_m, \ldots, k_n)$ and returns a "*yes*" or "*no*" answer indicating whether t is/is not in the filter. If one of the join keys of the tuple t, $t(k_i)_{i=1\ldots m}$, is not a member of the component filter BF_i of the EIF, the output is "*no*" answer. Otherwise, the output is "*yes*" answer. Figure 2 depicts its structure.

For example, consider the three-way join $R(uname, uid) \bowtie K(uid1, uid2) \bowtie L(uid, event)$. K can be filtered by an EIF composed of $BF_1(R.uid \cap K.uid1)$ and $BF_2(K.uid2 \cap L.uid)$, i.e., $IBF_1(R.uid, K.uid1)$ and $IBF_2(K.uid2, L.uid)$.

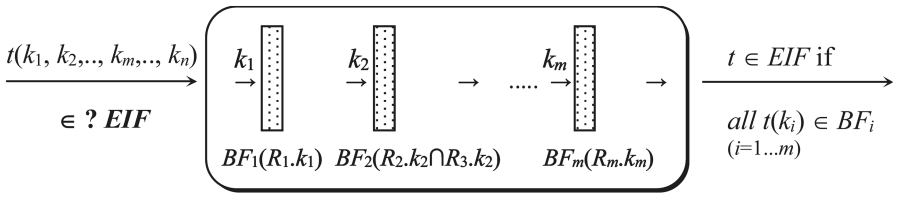


Fig. 2. Extended intersection filter - $EIF(BF_1, BF_2, \ldots, BF_m)$

Each tuple $t(k_1, k_2) \in K$ is checked against the two filters. If k_1 and k_2 are in IBF_1 and IBF_2, respectively, t is accepted, else it is eliminated.

2.4 Joins with MapReduce

Join processing in MapReduce has become a hot research topic in recent years [2,3,8,11,16,22,30]. Many studies have been carried out to evaluate join queries and analyze large datasets in a MapReduce environment. 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, data constraints, and joining rates. Map-side joins [8,20,37] 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 limited in running extra MapReduce jobs to repartition the data sources to be usable. Meanwhile, Reduce-side joins [8,20,25,37] 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-matching data right in the map phase. This problem can be solved by Semi-join [8]. It uses 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, Bloom join [19,22,23,39,40] is a worthy replacement for Semi-join because it benefits from a Bloom filter [9] to do existence tests in less memory than a full list of keys from the hashmap. Another restriction on these solutions is derived from their filtering efficiency, even for recent research efforts [3,22,40]. There remain a lot of non-matching data after filtering because the solutions can only filter on one of input datasets instead of both. Thus, Intersection filter-based join [30] may become a better solution to address this problem by eliminating non-matching data from both input datasets. However, it is necessary to have a complete evaluation of the solutions that indicates their benefits and limitations.

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 latest approaches, computing multi-way joins [3,8,21,40] and recursive joins [1,2,12,33] also often generates intermediate results that may be inputs of component joins of the

joins. These intermediate results contain a lot of non-matching 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-matching data involved in the intermediate results. Besides, minimizing the intermediate data amount sent to the reducers should be calculated appropriately.

The purpose of the present paper is to provide a consistent review of filter-based join processing techniques in a MapReduce environment. It not only (a) covers the recently various techniques for computing two-way joins, multi-way joins and recursive joins, but also (b) qualifies these techniques with cost models and (c) evaluates them with experimental studies to both validate the proposed cost model and investigate their practical behavior. Overall, our goal is to provide a clear, robust and comparative assessment of join processing solutions to guide the choice of practitioners confronted to the need to perform join at scale in a specific context. By founding this assessment on both an analytic and empirical study, we hope to provide a material that puts the research contributions in this field in a coherent setting and clarifies the stakes of combining several inputs with MapReduce.

For the sake of consistency, we focus on join algorithms that share some common features. First, we only consider equijoins. Second, we investigate algorithms that exploit filters to reduce the network communication. Filtering is a strategy that can be combined with all kind of approaches, and turns out to be (almost) always beneficial in a context where I/O and network exchanges constitute the major bottleneck. Third, our work complements a few other surveys recently published [14,20,24,31,32,40] which, on the one hand, explore a larger scope (e.g., non-equi joins [14,27,38,41]), but on the other hand do not propose an in-depth coverage as we do, and a comparison methods uniformly applied to the range of proposals published so far.

3 Extending Equijoins with Filters in MapReduce

The most straightforward way to join datasets with MapReduce is the Reduce-side join algorithm [8,20,25,37], denoted RSJ. It groups tuples from both datasets on their respective join key value during the map phase, and merges/joins them during the reduce phase. Tuples are processed regardless of their actual contribution to the final result, and thus the join algorithm has to pay an overhead for processing and shipping useless data.

Consider for instance the Facebook user dataset R containing more than 1.23 billion users [15]. We would like to obtain users' activities in a certain period of time (e.g., one hour) by joining R and the log dataset L. Since L, *over this period*, contains the activities of only a few million unique users, most of the users in R are not represented, and RSJ spends useless resources to access, process and transfer over the network the non-matching tuples of R.

Several *filter-based* extensions have been proposed to tackle the problem. Their common idea is to filter out the non-matching tuples from the input

datasets during the map phase. A *filter* in this context is a compact data structure that supports fast membership tests. Filter-based joins require two stages:

- *Stage1* (pre-processing). A filter F is built on a join key value set of one input dataset. For the intersection filter, F represents the intersection of the key value sets. A membership test for some key value k on F tells whether k participates or not to the join result.
- *Stage 2* (join). F is distributed to all the computing nodes, and used to eliminate non-matching tuples during the map phase. The join then proceeds as explained above.

A filter is a compact representation of a set. It accepts a rate of false positives (i.e., positive answer for non-matching tuples in our case) but no false negatives. Filtering avoids the communication overhead of shipping tuples from the mappers to the reducers, and the storage and CPU overhead of processing such tuples during the reduce phase. The join strategy remains unchanged, and exploits the MapReduce paradigm: the input datasets are partitioned and grouped during the map phase, in order to solve locally the problem during the reduce phase. Filtering presents some advantages and disadvantages:

- *Advantages*: the strategy does not impose any restrictions on input datasets, nor modifications to the MapReduce framework. Besides, it removes non-matching data to reduce the communication overhead.
- *Disadvantages*: building the filters represents a significant cost, since it requires scanning the input, and transferring the filters.

In the rest of this section, we examine in detail the application of filter-based techniques to the following join variants: two-way joins, multi-way joins and recursive joins. For each variant, we present the state-of-the-art algorithms, along with a discussion on their expected advantages/disadvantages.

3.1 Two-Way Joins

A two-way join $R_1 \bowtie R_2$ involves two relations R_1 and R_2. In the following r_1 (resp. r_2) denote a tuple from R_1 (resp. R_2) and k refers to the join key attribute. We use simplified notations when allowed by the context.

Bloom Joins. *Bloom join* (BJ) [19,22,39] is a specific type of the filter-based join strategy in which the well-known Bloom filter [9] is used. BJ is implemented by two MapReduce jobs as follows:

- *Job 1* (preprocessing) is a job with only one reducer. The mappers scan splits of the input R_2, extract the join key value from each tuple, and produces local Bloom filters. Then, the mappers emit the local filters to the reducer that merges them into a global filter $BF(R_2)$ using the bit-wise **OR**.

- *Job 2* (processing) filters out non-matching tuples in R_1 and joins the filtered result R_1' with R_2. It relies on a distributed cache to store $BF(R_2)$. The mappers scan splits of R_1 and R_2, and eliminate the tuples of R_1 whose keys are not in $BF(R_2)$. Tuples from R_2 are not filtered.
  Each tuple is then ticked with a tag that indicates its dataset name. For our example, mappers emit tagged tuples with composite keys of the form $((r_1.k, `R_1'), r_1)$ or $((r_2.k, `R_2'), r_2)$. The reducers receive tagged tuples grouped on the k value (this requires a small change of the partitioning function). For each group, the reduce function constructs all the pairs (r_1, r_2) to complete the join.

Note that it requires to override the default grouping function in order to ensure that grouping the tagged tuples takes into consideration only the join key part and ignores the tag part. The tag is used for secondary sort which ensures that, for a given key value, all tuples from R_1 are processed before those of R_2. This allows to apply a standard in-memory hash join.

Discussion. BJ benefits from the compacity of the Bloom filter to reduce the amount of data transferred over the network. The size of the filter can be fixed regardless of the number of join keys. However, given a fixed filter size, the probability f of false positives increases with the number of join keys.

A major concern with the filtering approach in general is the need to run a pre-processing job for building the filter. Besides, broadcasting the filter becomes inefficient if its size is large. Finally, it is worth noting that the BJ is asymmetric: non-matching tuples of R_2 have not been filtered, hence the problem is half-solved.

The authors of [22] have proposed an improvement of BJ that avoids the pre-processing job, but requires two internal modifications of the framework. We do not consider in the present study such extensions that necessitate a non-standard environment.

Intersection Filter-Based Joins. We now describe an improvement of the above approach, the *Intersection filter-based join* [30], denoted IFBJ. It relies on the fact that only tuples whose join keys belong to the set of *shared* join keys do participate to the result.

The implementation of IFBJ is done with the following jobs:

- *Job 1* (pre-processing) is a job with only one reducer. The mappers scan splits of R_1 and R_2, extract the join key value for each tuple, and insert them in the local Bloom filters regardless of duplicate keys.
  The mappers then emit the local filters to the reducer which merges them in two global filters $BF(R_1)$ and $BF(R_2)$ using the bit-wise OR. Based on one of three approaches to building the intersection filter [30], the reducer computes the intersection filter $IBF(R_1, R_2)$ from the global filters.
- *Job 2* (join) uses a distributed cache to provide IBF to all the compute nodes. The mappers scan splits of R_1 and R_2, extract the join key for each tuple and

match it against the intersection filter. If the key v belongs to the intersection filter, the tuple is emitted as a pair $((v, tag), tuple)$. The join evaluation in the reduce phase is similar to the Bloom join algorithm.

IFBJ benefits from the standard features of Bloom filters: its small size, its independence from the number of the keys and key duplication, and fast membership test. Join based on the intersection filter is expected to be more efficient than the Bloom join because of its ability to filter out non-matching tuples from both two input datasets. An interesting characteristic of the intersection filter is that if $IBF(R_1, R_2)$ has all bits set to zero, then the sets $R_1.k$ and $R_2.k$ are disjoint and the join evaluation stops without doing anything. However, the algorithm has to pay the additional cost of a MapReduce job for building the intersection filter and requires scanning the two input datasets twice.

3.2 Multi-way Joins

We can extend the above approach to the computation of multi-way joins with an extended intersection filter (EIF) in the following.

Three-Way Joins. We begin our study of multi-way joins by considering the special case of a three-way join $R_1 \bowtie R_2 \bowtie R_3$. For the sake of concreteness, we will discuss the following query on our example relations.

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

There are several possible pairwise combinations to compute this three-way join.

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

We can evaluate three-way joins as a sequence of 2 two-way joins, using two successive jobs. An alternative is to join the three datasets together with a

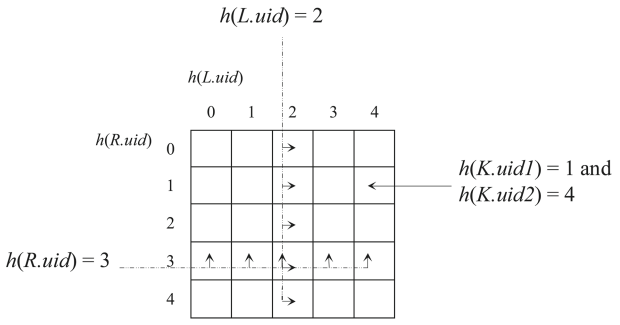


Fig. 3. Distributing tuples of R, K, and L among $r = n^2$ reducers

single job, as recently proposed by Afrati and Ullman [3]. It relies on the idea of a *matrix of reducers* as shown in Fig. 3.

The number of reducers must be the square of some integer n ($r = n^2$) and reducers are mapped (virtually) to a matrix $n \times n$. Each reducer is mapped to a cell (i, j), and identified by $i * n + j$. With $n = 5$, cell (3, 2) is for instance associated with the reducer 17.

The mappers assign tuples of R, K, and L to the reducer matrix as follows. Let h be a hash function with range $[0, n-1]$. Each tuple of K is sent to a single reducer, the one in cell $(h(K.uid1), h(K.uid2))$. Tuples from R and L are sent to all the reducers of, resp. a whole row or column in the matrix. Each tuple $r(uid, uname)$ is sent to all the reducers of the *row* $h(uid)$. Each tuple $l(uid, event)$ is sent to all the reducers of the *column* $h(uid)$.

We can give a perspective: assume three tuples R('Laurent', u_1), $K(u_1, u_2)$, and $L(u_2$, 'login'). They will all be sent to the reducer $h(u_1) * n + h(u_2)$ and the joined tuple will therefore be produced.

Let us assume, for simplicity, that $|R|=|K|=|L|$. The total communication cost for the Afrati's three-way join (denoted 3WJ in the following) is $O(|R|.\sqrt{r})$, whereas the total communication cost for the cascade of 2 two-way joins without filters is $O(|R|^2.\alpha)$, where α is the probability for two tuples from different datasets to match on the join key (Sect. 4.2 for more details). It follows that 3WJ is better than the cascade of the two-way joins when $r < (|R|.\alpha)^2$.

A downside of 3WJ is that it generates n duplicates for each tuple of either R or L. This represents a large communication and I/O overhead. This situation can be improved significantly by removing non-matching tuples prior to the reduce phase. We extend 3WJ with intersection filters as shown in Fig. 4.

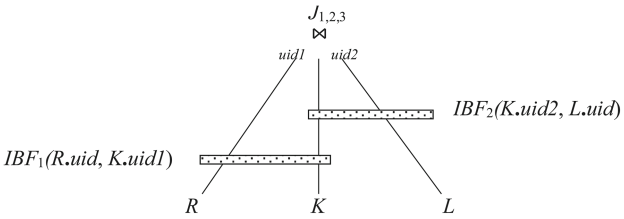


Fig. 4. Three-way join extended with intersection filters

R and L are filtered by $IBF_1(R.uid, K.uid1)$ and $IBF_2(K.uid2, L.uid)$, respectively. K is filtered by an extended intersection filter $EIF(IBF_1, IBF_2)$.

The extension of the three-way join with filters uses two jobs as follows.

- *Job 1* (pre-processing) builds $IBF_1(R.uid, K.uid1)$ and $IBF_2(K.uid2, L.uid)$. Let mp_1, mp_2 and mp_3 be the number of map tasks for R, K and L, respectively. The job consists of $mp_1 + mp_2 + mp_3$ map tasks that build filters and one reduce task that produces two intersection filters. In detail, mp_1 tasks

build local filters on $R.uid$; mp_2 tasks build local filters on $K.uid1$ and $K.uid2$; mp_3 tasks build local filters on $L.uid$. Those filters are shipped to the reducer which produces $BF(R.uid)$, $BF(L.uid)$, $BF(K.uid1)$, $BF(K.uid2)$, as well as IBF_1, IBF_2 and $EIF(IBF_1, IBF_2)$. Note that the join result is known to be empty right away if either IBF_1 or IBF_2 is empty.

- *Job 2* (join) filters out non-matching tuples from R, K and L, and carries out the join evaluation. It distributes the intersection filters to all tasktrackers, creates map tasks for R, K and L and r reduce tasks.
  - ⋆ *Map phase with filtering:* Each mapper matches any tuple of R or L against the relevant filter IBF_1, IBF_2, or $EIF(IBF_1, IBF_2)$. Tuples that pass the filtering process are then sent to the reducers according the 3WJ policy. This involves tuple replication as shown in Fig. 3.
  - ⋆ *Reduce phase:* the reduce function applies a full cross-product of tuples from the different input datasets. Locally, the reducer buffers the tuples of R and L, streams the tuples from K, and performs the cross product.

Chain Joins. We now consider the more general case of multi-way joins, or *chain joins*, a sequence or pair wise joins of the form of $R_1(x_1, x_2) \bowtie R_2(x_2, x_3) \bowtie R_3(x_3, x_4) \bowtie \cdots \bowtie R_n(x_n, x_{n+1})$.

The baseline solution is a cascade of Bloom joins (CJ-BJ). The query plan is a left-deep join tree, and relies on a set of filters $BF_2(R_2.x_2), \ldots, BF_n(R_n.x_n)$ built on the base datasets by a pre-processing job. In this scenario, we can regconize that R_1 and all intermediate results $R_{1,2,\cdots,i}$ are filtered by the filters, whereas the base relations R_i are not, where $i \in [2, n]$.

We propose an improved evaluation that generalizes intersection filters as shown by Fig. 5. In addition to the filters BF on base relations, the extended

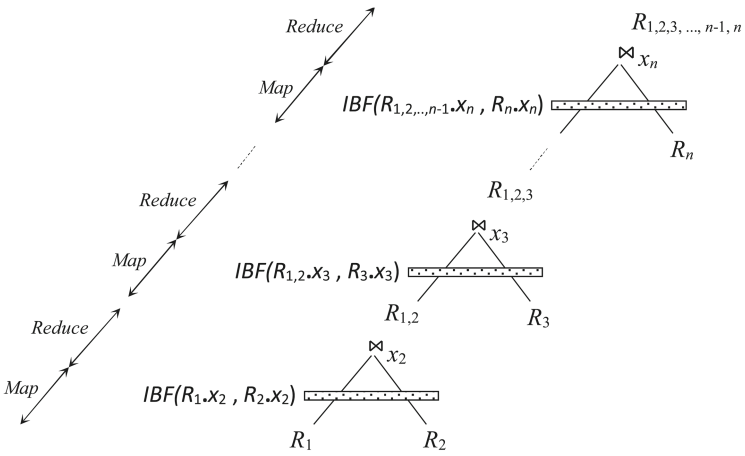


Fig. 5. Implementation of a chain join using a cascade of two-way joins using intersection filters (CJ-IFBJ)

algorithm denoted CJ-IFBJ creates on the fly intersection Bloom Filters on intermediate results, $IBF(R_{1,\cdots,i-1}.x_i, R_i.x_i), i \in [2, n]$ during the reduce phases of intermediate joins

All the input datasets and intermediate join results are filtered by their corresponding intersection filters. For instance, $IBF(R_{1,2}.x_3, R_3.x_3)$ is used to eliminate non-matching tuples in both $R_{1,2}$ and R_3. Intermediate data sent to the reducers with CJ-IFBJ is expected to be much less than in the case of CJ-BJ.

We can even go one step further by noting that intermediate join results still contain non-matching tuples transmitted to the next join. For instance, the join of R_1 and R_2 likely contains tuples that do not match any tuples of R_3 on x_3. We can therefore "push" the filter $BF(R_3.x_3)$, down to the scan of relation R_2. The idea is actually quite reminiscent of the traditional optimization heuristics that pushes selection down the query tree in relational systems.

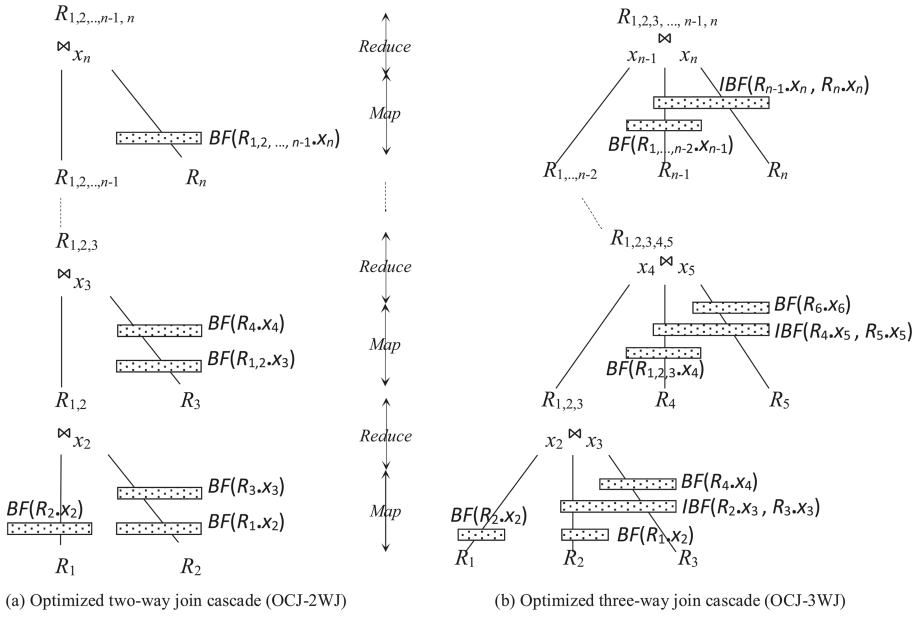


Fig. 6. Optimization of a chain join using extended intersection filters (OCJ)

Figure 6(a) shows a first optimized solution using a cascade of filter-based two-way joins, denoted OCJ-2WJ. The input datasets R_2, ..., R_n are filtered by extended intersection filters EIF. The extended filter EIF_i includes a filter $BF(R_{1,\cdots,i-1}.x_i)$ built from the intermediate join result and a filter $BF(R_{i+1}.x_{i+1})$ from the next input dataset, where $i \in (2, n)$. Specially, EIF_2 contains $BF(R_1.x_2)$ and $BF(R_3.x_3)$, and EIF_n only consists of $BF(R_{1,2,\cdots,n-1}.x_n)$. This solution ensures that intermediate join results only contain (up to false positives) matching data that can be sent to the next join

without filtering. This is an important characteristic which avoids to apply additional filters to intermediate join results.

The implementation first uses a pre-processing job to build the Bloom filters $BF(R_i.x_i)$, $i = 2, \ldots, n$, and $BF(R_1.x_2)$. Next, it evaluates the chain join as a sequence of two-way joins. During the evaluation of $R_{1,\cdots,i-1} \bowtie R_i$, the left input need not be filtered, except R_1 filtered by $BF(R_2.x_2)$. The right input is filtered by the EIF_i built from $BF(R_{1,\cdots,i-1}.x_i)$ and $BF(R_{i+1}.x_{i+1})$. The former is generated in the reduce phase of the previous join processing between $R_{1,\cdots,i-2}$ and R_{i-1}. Building the filters from the intermediate join results does not involve any overhead. The iteration stops if one of the two input datasets is null.

Figure 6(b) illustrates a second optimization, where pairwise joins are replaced by filtered three-way joins (3WJ). We denote this further optimized solution as OCJ-3WJ. Consider the three-way join $R_{1,\cdots,i-1} \bowtie R_i \bowtie R_{i+1}, i \in [2, n-1]$ and i is an even number. The left relation does not need to be filtered, apart from R_1 filtered by $BF(R_2.x_2)$. The middle relation is filtered by the extended intersection filter EIF_i built from $BF(R_{1,\cdots,i-1}.x_i)$ and a filter $IBF(R_i.x_{i+1}, R_{i+1}.x_{i+1})$. The last input is filtered by EIF'_i, built from $IBF(R_i.x_{i+1}, R_{i+1}.x_{i+1})$ and $BF(R_{i+2}.x_{i+2})$. When $(i+2) > n$, the filter EIF'_i does not contain $BF(R_{i+2}.x_{i+2})$ because R_{i+2} does not exist. It is noted that OCJ-3WJ may contain a two-way join of $R_{1,\cdots,n-1}$ and R_n if n is an even number.

The implementation of the second solution is similar to the first one. OCJ-2WJ is expected to use less memory than OCJ-3WJ 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. The downside is that OCJ-2WJ requires more jobs than OCJ-3WJ. If n denotes the number of input datasets, the number of the two-way join jobs of OCJ-2WJ is $(n-1)$, while OCJ-3WJ needs $\lceil \frac{n-1}{2} \rceil$ jobs.

3.3 Recursive Joins

We now turn to another complex type of join. A *recursive join* [17,29] computes the transitive closure of a relation encoding a graph. A typical example, expressed in Datalog, is given below.

$$Friend(x, y) \longleftarrow Know(x, y)$$
$$Friend(x, y) \longleftarrow Friend(x, z) \bowtie Know(z, y)$$

Evaluating a recursive join is tantamount to computing the transitive closure of the graph represented by the relation. This can be done via an iterative process that stops whenever a fixpoint is reached. We examine how the seminaive algorithm [36] can be evaluated in MapReduce.

Let F and K denote the relations *Friend* and *Know*, respectively. Let $F_i, 0 < i \leq n$ be the temporary value of the relation *Friend* at step 0, with $F_0 = \emptyset$. The incremental relation of $F_i, i > 0$, denoted ΔF_i, is defined as:

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

The semi-naive algorithm uses this delta relation to avoid redundant computations (Algorithm 1).

Algorithm 1. Semi-Naive evaluation for recursive joins

Input: A graph encoded as a relation K
Output: The transitive closure of K
1 $F = \emptyset, \Delta F_0 = K(x, y), i = 1$
2 **while** $\Delta F_{i-1} \; != \emptyset$ **do**
3 $\quad \Delta F_i = \Pi_{xy}(\Delta F_{i-1} \bowtie_z K) - F$
4 $\quad F = F \cup \Delta F_i$
5 $\quad i+ = 1$
6 **return** F

At each step i, some new facts are inferred and stored in ΔF_i. The loop is repeated until no new fact is inferred ($\Delta F_i = \emptyset$), i.e., the fixpoint is reached. The union of all incremental relations, $(\Delta F_0 \cup \ldots \cup \Delta F_{i-1})$, is the transitive closure of the graph.

Shaw et al. [33] have proposed the following algorithm to implement the semi-naive algorithm in MapReduce (REJ-SHAW). Each iteration evaluates $\Delta F_i = \Pi_{xy}(\Delta F_{i-1} \bowtie_z K) - F_{i-1}$ with two jobs, namely, one for join and one for deduplication and difference (*dedup-diff*), as shown on Fig. 7.

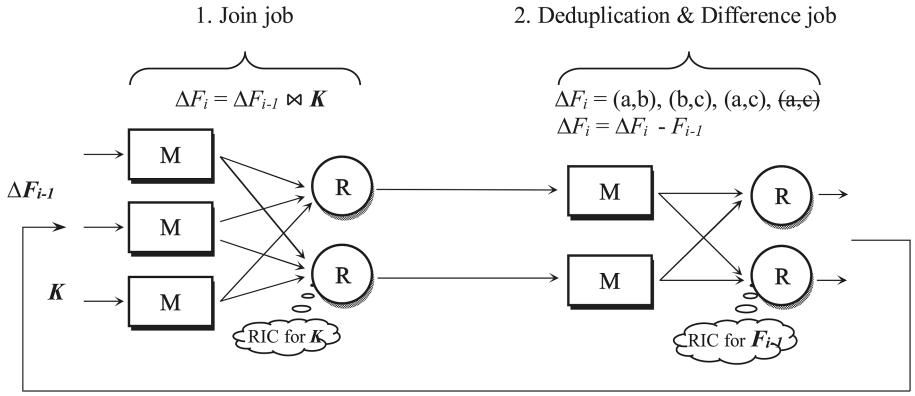


Fig. 7. Semi-naive implementation of recursive joins in MapReduce

The first job computes $(\Delta F_{i-1} \bowtie K)$, the second computes the new delta relation ΔF_i. This 2-jobs execution is iterated until ΔF_i is empty. This means that the invariant relation K and the incremental relation F_{i-1} are re-scanned

and re-shuffled for every iteration. Shaw et al. have tackled this situation in the HaLoop system [12] by using the Reducer Input Cache (RIC). RIC stores and indexes reducer inputs across all reducers. To avoid re-scanning and re-shuffling the same data with the same mapper on iterations, the solution therefore uses RIC for the datasets K and F_{i-1} in the join job and the dedup-diff job, respectively, as shown on Fig. 7. K is scanned only once, at the first loop. K_i and K_j are splits of K, which are cached at the reducer input caches i and j, resp. Note that caching intermediate results during iterative computations is now integrated in modern distributed engines such as Spark [7] and Flink [5].

The dedup-diff job using RIC 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. 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, 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. A tuple previously discovered is omitted from the output, else it is included in ΔF_i.

When evaluating $(\Delta F_{i-1} \bowtie K)$, Shaw's solution (REJ-SHAW) does not discover and eliminate non-matching tuples from ΔF_{i-1} and K. Our extension, REJ-FB in the following, adds an intersection filter $IBF(\Delta F_{i-1}.z, K.z)$ as proposed in Sect. 2.3. Initially, the filter is simply $BF(K.z)$ generated by a pre-processing job. During the i^{th} iteration $(i \geq 1)$, REJ-FB uses $IBF(\Delta F_{i-1}.z, K.z)$ as a filter in the map phase of the join job, and builds $IBF(\Delta F_i.z, K.z)$ in the reduce phase of the dedup-diff job.

A fixpoint of the recursive join is reached when no new tuples are discovered (i.e. ΔF_i is empty) or, equivalently, when the IBF is empty. The latter is a better stop condition because it can save one iteration.

4 Performance Analysis for Filter-Based Equijoins

We now develop an analysis of the algorithms presented so far.

4.1 Two-Way Joins

We note R and L the two input datasets, and analyze the cost for, respectively, the Bloom join (BJ) and the intersection filter-based join (IFBJ). Table 2 summarizes the parameters of our cost model.

Cost Model. We adapt the cost model presented in [26]. We propose the following global formula that captures the cost of a two-way join.

$$\boldsymbol{C_{2wJoin} = C_{pre} + C_{read} + C_{sort} + C_{tr} + C_{write}} \quad (1)$$

where:

- $C_{read} = c_r \cdot |R| + c_r \cdot |L|$
- $C_{sort} = c_l \cdot |D| \cdot 2 \cdot (\lceil \log_B |D| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$ [26]
- $C_{tr} = c_t \cdot |D|$
- $C_{write} = c_r \cdot |O|$
- $C_{pre} = C' + c_r \cdot m \cdot t$
  - $C' = \begin{cases} C_{read} + (c_l + c_t) \cdot m \cdot mp & \text{, for IFBJ} \\ c_r \cdot |L| + (c_l + c_t) \cdot m \cdot mp_2 & \text{, for BJ} \end{cases}$
  - $C_{pre} = 0$, for approaches without filters. In addition, it is assumed that the filters are the same size m. If m is small, we will not compress the filter files and m is therefore the size of the Bloom filter.

An additional component, C_{pre}, is added to the cost model in [26] to form Eq. (1). $|D|$, the size of the intermediate data, strongly influences the total cost of a join operation, and is essential in particular to decide whether the filter-based variant of the algorithm is worth its cost.

Table 2. Parameters of the cost model for two-way joins

Parameter	Explanation
$\|R\|$	The size of R
$\|L\|$	The size of L
$\|D\|$	The size of the intermediate data
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 in pages
mp_1	The number of map tasks for R
mp_2	The number of map tasks for L
mp	The total number of map tasks, $mp = mp_1 + mp_2$
t	The number of tasktrackers
m	The compressed size of the Bloom filter (bits) m = the size of the Bloom filter $\times$ the file compression ratio
$\|O\|$	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

Cost Comparison. In this section, we evaluate $|D|$, for each algorithm mentioned in Sect. 3.1, and provide a cost comparison. Importantly, we identify a threshold that can guide the choice amongst of these algorithms. We add the Reduce-side join (RSJ) to our comparison to highlight the effect of filtering.

We denote as δ_L and δ_R, respectively, the ratio of the joined records of R with L (resp. L with R). The size of intermediate data is:

$$|D| = \begin{cases} \delta_L|R| + f_I(R,L)\cdot(1-\delta_L)|R| + \delta_R|L| + f_I(R,L)\cdot(1-\delta_R)|L| & (2) \\ \delta_L|R| + f(L) \quad \cdot(1-\delta_L)|R| + \quad |L| & (3) \\ \quad |R| \qquad\qquad\qquad\qquad + \quad |L| & (4) \end{cases}$$

where:

- Equation (2) holds for IFBJ, denoted D_{IFBJ}
- Equation (3) holds for BJ, denoted D_{BJ}
- Equation (4) holds for RSJ, denoted D_{RSJ}
- $f_I(R,L)$ is the false positive probability of the intersection filter $IBF(R,L)$ [30],
- and $f(L)$ is the false positive probability of the Bloom filter $BF(L)$.

From these equations, we can infer the following.

Theorem 1. *An* IFBJ *is more efficient than a* BJ *because it produces less intermediate data. Additionally, the following inequality holds:*

$$\boldsymbol{D}_{\text{IFBJ}} \leq \boldsymbol{D}_{\text{BJ}} \leq \boldsymbol{D}_{\text{RSJ}} \tag{5}$$

where D_{IFBJ}, D_{BJ}, and D_{RSJ} are the sizes of intermediate data of IFBJ, BJ, *and* RSJ, *resp.*

Proof. We get $0 < f_I(R,L) < f(L) < 1$ [30]. So we can deduce that:

$$\delta_L \cdot |R| + f_I(R,L)\cdot(1-\delta_L)\cdot|R| \leq \delta_L\cdot|R| + f(L)\cdot(1-\delta_L)\cdot|R| \leq |R| \text{ and} \tag{6}$$

$$\delta_R \cdot |L| + f_I(R,L)\cdot(1-\delta_R)|L| \leq |L| \tag{7}$$

Note that the ratio of the joined records, δ_L or δ_R, could be 1 in the case of a join based on a foreign key.

By combining inequalities (6) and (7) into Eqs. (2), (3) and (4), Theorem 1 is proved. □

From Eqs. (1) and (5), we can evaluate the total cost of the join operation for the different approaches.

Theorem 2. *Once the pre-processing cost C_{pre} is negligible or less than the cost of non-matching data, an* IFBJ *has the lowest cost. In addition, a comparison of the costs is given by:*

$$\boldsymbol{C}_{\text{IFBJ}} \leq \boldsymbol{C}_{\text{BJ}} \leq \boldsymbol{C}_{\text{RSJ}} \tag{8}$$

where C_{IFBJ}, C_{BJ}, *and* C_{RSJ} *are the total costs of* IFBJ, BJ, *and* RSJ, *resp. As a result, the most efficient join approach is typically* IFBJ, *the second one is* BJ, *and the worst one is* RSJ.

The total cost to perform the pre-processing job is given by:

$$C_{pre} = \begin{cases} C_{read} + (c_l + c_t) \cdot m \cdot mp + c_r \cdot m \cdot t & \text{, in case of IFBJ} \\ c_r \cdot |L| + (c_l + c_t) \cdot m \cdot mp_2 + c_r \cdot m \cdot t & \text{, in case of BJ} \\ 0 & \text{, in case of RSJ} \end{cases}$$

Regarding data locality, the MapReduce framework makes its best efforts to run the map task on a node where the input data resides. Although this cannot always be achieved, we can see that the cost of this phase, C_{pre}, is negligible compared to the generation and transfer of non-matching tuples over the network. In general, choosing the filter-based joins relies on the read cost c_r and a threshold of non-matching data shown in Theorem 3.

The filter-based join algorithms will become inefficient when there is a large number of map tasks, and very little non-matching data in the join operation. For large inputs with many map tasks, a tasktracker running multiple map tasks will maintain only two local filters $BF(R)$ and (or) $BF(L)$ thanks to merging the local filters of the tasks. Two hundred map tasks running on a tasktracker, for instance, will produce 200 local filters $BF(R)$. The tasktracker merges all the local filters into one $BF(R)$. Besides, as the number of non-matching tuples decreases, the filters become useless and computing them with an additional job represents a penalty. It hence needs to indicate the dependence of the filter-based joins on the amount of non-matching data through estimating the threshold for this data that determines whether filters should be used.

Let $|D^*|$ be the size of non-matching data, C^*_{sort} be the total cost of sorting and copying it at the map and reduce nodes, and C^*_{tr} be the total cost to transfer it among the nodes. Accordingly, the cost associated with non-matching data is the sum of C^*_{sort} and C^*_{tr}.

Theorem 3. *The filter-based joins become a good choice when:*

$$\boldsymbol{C_{pre} < C^*_{sort} + C^*_{tr}} \tag{9}$$

where:

- $|D^*| = |R| + |L| - |D|$
- $C^*_{tr} = c_t \cdot |D^*|$
- $C^*_{sort} = c_l \cdot |D^*| \cdot 2 \cdot (\lceil \log_B |D^*| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$ *[26]*

Based on the size of $|D|$, *the threshold depends on* δ_L *and* δ_R *(the ratio of the joined records).*

In summary, the best choice of the join approaches is IFBJ, the second one is BJ, and the worst one is RSJ (Theorem 2). However, this would become incorrect when the join has small input datasets and a high ratio of matching tuples that is defined by the threshold of the joined records (Theorem 3). In these cases, RSJ would be the best choice and the filter-based joins should not be used because the cost of building and broadcasting filter(s) becomes relatively significant.

4.2 Multi-way Joins

Three-Way Joins. Let R, K and L be three input datasets. The general formula that estimates the total cost of 3WJ is:

$$\boldsymbol{C_{3wJoin} = C_{pre} + C_{read} + C_{sort} + C_{tr} + C_{write}} \tag{10}$$

where:

- $C_{read} = c_r \cdot |R| + c_r \cdot |K| + c_r \cdot |L|$
- $C_{sort} = c_l \cdot |D| \cdot 2 \cdot (\lceil \log_B |D| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$ [26]
- $mp = mp_1 + mp_2 + mp_3$, the total number of map tasks for the three inputs
- $C_{tr} = c_t \cdot |D|$
- $C_{write} = c_r \cdot |O|$
- $C_{pre} = C_{read} + (c_l + c_t) \cdot m \cdot mp + 2 \cdot c_r \cdot m \cdot t$, for 3WJ using filters; $C_{pre} = 0$ for 3WJ.

To simplify the computation, we suppose that R, K and L have the same size. A 3WJ increases the communication cost because each tuple of R and L is sent to many different reducers. On the other hand, the two-way join cascade must launch an additional job, then scan and shuffle the intermediate result. We characterize the relative costs of the approaches as follows.

Theorem 4. *A* 3WJ, $R(A,B) \bowtie K(B,C) \bowtie L(C,D)$, *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* $r < (|R|.\alpha)^2$. *Additionally, the size of the intermediate data is specified by*

$$|D| = \begin{cases} 2 \cdot |R| \cdot \sqrt{r} & , \textit{ for } 3\text{WJ}. \\ |R|^2 \cdot \alpha & , \textit{ for a cascade of 2 two-way joins.} \end{cases}$$

where r is the number of reducers, $|R| = |K| = |L|$, and α is the probability of two tuples from different datasets to match on the join key column.

Proof. Similar to the proof of Afrati and Ullman in [3]. First, we consider 3WJ. Two attributes B and C are join key columns. 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

$$|R| \cdot c + |K| + |L| \cdot b \tag{11}$$

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

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

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

We make derivatives of $\mathcal{L}$ with respect to variables b and c.

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

We obtain the Lagrangian equations: $|L| \cdot b = \lambda \cdot r$, and $|R| \cdot c = \lambda \cdot r$
We can multiply these two equations together to get $|L| \cdot |R| = \lambda^2 \cdot r$
From here, we deduce $\lambda = \sqrt{|R| \cdot |L|/r}$
By substituting the value of λ in the Lagrangian equations, we get:

$$b = \sqrt{|R| \cdot r / |L|}, \text{ and } c = \sqrt{|L| \cdot r / |R|}$$

Then, from expression (11), we get the minimum communication cost of 3WJ

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

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

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

The cost of 3WJ, $O(|R| \cdot \sqrt{r})$, is compared with the cost of the two-way join cascade $O(|R|^2 \cdot \alpha)$. We can conclude that 3WJ will be better than the cascade when $\sqrt{r} < |R| \cdot \alpha$. In other words, for 3WJ, there is a limit on the number of reducers $r < (|R| \cdot \alpha)^2$ and Theorem 4 is hence proved. □

In general, we can extend Theorem 4 for 3WJ with n join key columns using an n-dimensional reducer matrix. For example, a 3WJ $R(A, B) \bowtie K(B, C) \bowtie L(C, A)$ with three join attributes A, B, and C. This three-way join needs a three-dimensional reducer matrix. The three-way join will become more efficient than a cascade of 2 two-way joins when $r < (|R| . \alpha)^3$ and its amount of communication is $3 \cdot |R| \cdot \sqrt[3]{r}$. In fact, choosing the number of reducers to satisfy this condition is not difficult. For example, if $|R| \cdot \alpha = 15$, as might be the case for the Web incidence matrix, we can choose the number of reducers r up to 3375. We can now characterize the cost of three-way join using filters.

Theorem 5. *A* 3WJ, $R(A, B) \bowtie K(B, C) \bowtie L(C, D)$, *is more efficient with filters than without filters when* C_{pre} *is negligible or less than the cost of processing non-matching data. Moreover, the* 3WJ *using the filters is also more efficient than the two-way join cascade using the filters when* $r < (|R'| \cdot \alpha)^2$. *With using the filters, the size of the intermediate data is defined by*

$$|D'| = \begin{cases} 2 \cdot |R'| \cdot \sqrt{r} & , \textit{ for } 3\text{WJ } \textit{using the filters.} \\ |R'|^2 \cdot \alpha & , \textit{ for a cascade of 2 two-way joins using the filters.} \end{cases}$$

$$|R'| = \delta \cdot |R| + f_I \cdot (1 - \delta) \cdot |R|, R' \textit{ is the filtered dataset of one input.}$$

where r *is the number of reducers,* α *is the probability of two tuples from different datasets to match on the join key,* $|R| = |K| = |L|$, δ *is the ratio of the joined records of one input dataset with another, and* f_I *is the false intersection probability between the datasets.*

Proof. Theorems 2 and 3 show that joins with the filters is more efficient than without the filters if C_{pre} is negligible or less than the cost of non-matching data. The following inequalities hold: $0 < \delta << 1$ and $0 < f_I << 1$

$\Rightarrow\ \delta \cdot |R| + f_I \cdot (1-\delta) \cdot |R| < |R|\ \Rightarrow\ |R'| < |R|$

Combining this equality with Theorem 4, we can easily prove Theorem 5. □

Chain Joins. Consider a chain join over n input datasets R_1, R_2, ..., R_n. We analyze OCJ-3WJ with the EIF filters presented in Sect. 3.2. The chain join is executed as a sequence of 3WJ jobs, $\ddot{J} = \{J_2, J_4, J_6, \ldots, J_{n-1}\}$. J_1 scans $R_1, \ldots, R_n$ inputs for building the filters. Each iteration carries out the join of three inputs, $R_{1,\cdots,2i-1}$, R_{2i}, and R_{2i+1}, where $1 \le i \le \lfloor (n-1)/2 \rfloor$. If n is even, OCJ-3WJ contains an additional two-way join job of $R_{1,\cdots,n-1}$ and R_n. We extend the cost model of 3WJ as follows:

$$C(\ddot{J}) = C_{pre} + \lceil (n-1)/2 \rceil \cdot C_{distCache} + C_{2wJoin} + \sum_{i=1}^{\lfloor (n-1)/2 \rfloor} (C_{read}(J_{2i}) + C_{sort}(J_{2i}) + C_{tr}(J_{2i}) + C_{write}(J_{2i})) \quad (12)$$

where:

- $C_{pre} = (\sum_{i=1}^{n} c_r \cdot |R_i|) + (c_l + c_t) \cdot m \cdot mp$
  - $C_{pre} = 0$ and $m = 0$ for approaches without using filters.
  - mp is the total number of map tasks.
- $C_{distCache} = 3 \cdot c_r \cdot m \cdot t$
  - $C_{distCache} = 0$ for approaches without using filters.
- C_{2wJoin} is specified by Eq. (1), the cost of joining $R_{1,\cdots,n-1}$ and R_n.
  - $C_{2wJoin} = 0$ if n is an odd number and greater than 2.
- $C_{read}(J_{2i}) = c_r \cdot |R_{1,\cdots,2i-1}| + c_r \cdot |R_{2i}| + c_r \cdot |R_{2i+1}|$
- $C_{sort}(J_{2i}) = c_l \cdot |D_i| \cdot 2 \cdot (\lceil \log_B |D_i| - \log_B(mp) \rceil + \lceil \log_B(mp) \rceil)$ [26]
  - $|D_i|$ is the size of the intermediate data in the i^{th} iteration.
- $C_{tr}(J_{2i}) = c_t \cdot |D_i|$
- $C_{write}(J_{2i}) = c_r \cdot |R_{1,\cdots,2i+1}| + a$
  - $a = 2 \cdot c_r \cdot m \cdot t$, for building $BF(R_{1,\cdots,2i+1})$ in the i^{th} iteration.
  - $a = 0$, for $(2i+1) = n$.

The computation of OCJ-2WJ is a sequence of (n-1) two-way join jobs. This computation can be also considered as a sequence of ((n-1)/2) three-way join jobs in which each of them is executed by a cascade of 2 two-way join jobs. As a result, OCJ-2WJ has the extra costs of writing and re-reading the intermediate results of the two-way joins, and initializing additional jobs. On the other hand, OCJ-3WJ incurs the costs of data duplication to the reducers. From Theorem 5, we can show that OCJ-3WJ is more efficient than OCJ-2WJ when $r < (|R'| \cdot \alpha)^2$.

4.3 Recursive Joins

Cost Model. In the semi-naive algorithm, the number of iterations l is the longest path length in the relation graph minus 1, called the depth of the transitive closure. The first job J_1 reads K and $\Delta F_0 = F$, and caches K at the reducers. Each subsequent job J_i reads ΔF_{i-1} and scans partitions of K cached at the reducers (RIC). The dedup-diff job I_i reads the join output O_i containing duplicates, maps and shuffles tuples of O_i to the reducers in order to generate ΔF_i.

We base our analysis on the cost model introduced in [26] and adapt it to the evaluation of the recursive join. Table 3 gives the parameters.

The total cost of the recursive join is specified by:

$$\begin{aligned} C(\hat{J}) = C_K + \textstyle\sum_{i=1}^{l} C_{read}(J_i) + C_{sort}(J_i) + C_{tr}(J_i) + C_{cache}(J_i) + C_{write}(J_i) \\ + \textstyle\sum_{i=1}^{l} C_{read}(I_i) + C_{sort}(I_i) + C_{tr}(I_i) + C_{cache}(I_i) + C_{write}(I_i) \end{aligned} \tag{13}$$

where:

- $C_K = c_r \cdot |K| + c_l \cdot |K| \cdot 2 \cdot (\lceil \log_B |K| - \log_B(mp_K) \rceil + \lceil \log_B(mp_K) \rceil) + (c_t + c_l) \cdot |K|$
- $C_{read}(J_i) = c_r \cdot |\Delta F_{i-1}|$
- $C_{sort}(J_i) = c_l \cdot |D_i| \cdot 2 \cdot (\lceil \log_B |D_i| - \log_B(mp_{\Delta Fi-1}) \rceil + \lceil \log_B(mp_{\Delta Fi-1}) \rceil)$ [26]
- $C_{tr}(J_i) = c_t \cdot |D_i|$
- $C_{cache}(J_i) = c_l \cdot |K|$
- $C_{write}(J_i) = c_r \cdot |O_i|$
- $|D_i| = |\Delta F_{i-1}| = \beta_{i-1} \cdot |O_{i-1}|$
- $C_{read}(I_i) = c_r \cdot |O_i|$
- $C_{sort}(I_i) = c_l \cdot |D^+_i| \cdot 2 \cdot (\lceil \log_B |D^+{}_i| - \log_B(mp_{\mathrm{O}i}) \rceil + \lceil \log_B(mp_{\mathrm{O}i}) \rceil)$ [26]
- $C_{tr}(I_i) = c_t \cdot |D^+{}_i|$
- $C_{cache}(I_i) = c_l \cdot |D^+{}_i| \cdot (|F_{i-1}| \ / \ r) + c_l \cdot |\Delta F_i| \cdot (|F_{i-1}| \ / \ r + 1)$
- $C_{write}(I_i) = c_r \cdot |\Delta F_i|$
- $|D^+{}_i| = |O_i|$
- $|\Delta F_i| = \beta_i \cdot |O_i|$

The average size of the cache at each reducer is $(|F_{i-1}| \ / \ r)$. For each incoming tuple of O_i, the reducer probes the cache to get all tuples previously discovered. For each new tuple discovered, the reducer rewrites its entire cache along with the new tuple. Therefore, the total cost of accessing the cache in the dedup-diff job, $C_{cache}(I_i)$, includes the costs of reading the reducer cache for tuples of O_i and rewriting the reducer cache for new tuples of ΔF_i.

Cost Comparison. The total cost of REJ-FB is smaller than that of REJ-SHAW because the intermediate data of REJ-FB is less than that of REJ-SHAW ($|D'_i| < |D_i|$). The amount of intermediate data of REJ-FB is defined by:

$$\begin{aligned} |D'_i| &= \delta_K^{i-1} \cdot |\Delta F_{i-1}| + f(K) \cdot (1 - \delta_K^{i-1}) \cdot |\Delta F_{i-1}| \\ &= \delta_K^{i-1} \cdot |D_i| + f(K) \cdot (1 - \delta_K^{i-1}) \cdot |D_i| < |D_i| \end{aligned} \tag{14}$$

Table 3. Parameters of our cost model for recursive joins

Parameter	Explanation
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 (all costs are measured in seconds per page)
mp_K	The total number of map tasks of the dataset K
$mp_{\Delta F_{i-1}}$	The total number of map tasks of the incremental relation ΔF_{i-1}
mp_{Oi}	The number of the map tasks of the join output (O_i)
r	The number of reduce tasks
t	The number of tasktrackers
$\|K\|$	The size of the dataset K that is invariant in loops
$\|\Delta F_{i-1}\|$	The size of the incremental relation in the $(i-1)^{th}$ iteration ($\|\Delta F_0\| = \|K\|$)
$\|\Delta F_i\|$	The size of the incremental relation in the i^{th} iteration. The dataset ΔF_i contains only the differences between the join output O_i and F_{i-1}
$\|F_{i-1}\|$	The size of all incremental relations in the iterations 0 to $i-1$ ($\|\Delta F_0 \cup \cdots \cup \Delta F_{i-1}\|$)
$\|D_i\|$	The intermediate data size of the join job J_i in the i^{th} iteration
$\|D^+{}_i\|$	The intermediate data size of the dup-diff job I_i in the i^{th} iteration
$\|O_i\|$	The size of the join processing output O_i. The output O_i may contain duplicate elements with F_{i-1} (previous incremental relations)
β_i	The difference ratio of the output O_i with F_{i-1}
C_K	The total cost to read, map and sort, shuffle, and cache K at the reducers (RIC) in the first iteration
$C_{read}(J_i)$	The total cost to read the incremental relation ΔF_{i-1} from DFS
$C_{sort}(J_i)$	The total cost to perform the sorting and copying of the join job at the map and reduce nodes
$C_{tr}(J_i)$	The total cost to transfer intermediate data of the join job among nodes
$C_{cache}(J_i)$	The total cost to locally read partitions of K cached at the reducers
$C_{write}(J_i)$	The total cost to write O_i to DFS
$C_{read}(I_i)$	The total cost to read the join output O_i from DFS
$C_{sort}(I_i)$	The total cost to perform the sorting and copying of the dup-diff job at the map and reduce nodes
$C_{tr}(I_i)$	The total cost to transfer intermediate data of the dup-diff job (=O_i) among the nodes
$C_{cache}(I_i)$	The total cost to locally read partitions of F_{i-1} cached at the reducers
$C_{write}(I_i)$	The total cost to write ΔF_i to DFS

where:

- δ_K^{i-1} is the ratio of the joined records of ΔF_{i-1} with K
- $f(K)$ is the false positive probability of the Bloom filter $BF(K.z)$

We need a pre-processing job for building the Bloom filter $BF(K.z)$ that is used in all iterations. The additional overhead of building the filter $BF(K.z)$ is:

$$C'_K = C_K + C_{pre} \tag{15}$$

where:

- $C_{pre} = c_r \cdot |K| + (c_l + c_t) \cdot m_k \cdot mp_k + c_r \cdot m_k \cdot t$
- m_k is the compressed size of the Bloom filter of the input dataset K (bits). It is the product of the size of the filter and the file compression ratio. If the size of the filter is small, the file compression ratio should be one.

Besides, on each iteration, the program also re-computes the global filter $BF(\Delta F_i.z)$ generated in the reduce phase of the dedup-diff job. The overhead of creating the filter $BF(\Delta F_i.z)$ is:

$$C'_{write}(I_i) = C_{write}(I_i) + (2 \cdot c_r \cdot m_{\Delta Fi} \cdot r + c_r \cdot m_{\Delta Fi}) \tag{16}$$

where:

- $m_{\Delta Fi}$ is the compressed size of the Bloom filter of the incremental dataset ΔF_i (bits)

Since the size of the filters is small, these extra overheads are negligible compared to the overheads associated with redundant data in the incremental dataset.

5 Experimental Evaluation for Filter-Based Equijoins

In this section, we present experimental results obtained from the execution of two-way joins, chain joins, and recursive joins.

5.1 Two-Way Joins

Cluster Environment and Datasets. All experiments were run on a cluster of 15 virtual machines using Virtualbox [28]. Each machine has two 2.4 Ghz AMD Opteron CPUs with 2 MB cache, 10 GB RAM and 100 GB SATA disks. The operating system is 64-bit Ubuntu server 12.04, and the java version is 1.7.0.21. We installed Hadoop [6] version 1.0.4 on all nodes. One of the nodes was selected to act as Master and ran the NameNode and the JobTracker processes; the remaining nodes host the TaskTrackers in charge of data storage and data processing. Each TaskTracker node was configured to run up to two simultaneous map tasks and two reduce tasks. The HDFS block size was set to 128MB, size of read/write buffer was 128 KB, heap-size for JVMs was set to 2048 M, and the number of reduce tasks set to 28.

All test datasets were produced by a data generation script of the Purdue MapReduce Benchmarks Suite [4], called "PUMA" which represents a broad

Table 4. Input datasets

Inputs	Test 1		Test 2		Test 3	
	size	*records*	*size*	*records*	*size*	*records*
Dataset1	15 GB	40,259,163	35 GB	92,681,333	55 GB	145,099,559
Dataset2	15 GB	40,108,215	35 GB	92,524,495	55 GB	139,573,823
Total	30 GB	80,367,378	70 GB	185,205,828	110 GB	284,673,382

range of MapReduce applications with high/low computing requirements and high/low shuffle volumes. The maximum number of columns in the datasets is 39 and string length in each column is set to 19 characters. The first column of *Dataset1* is a foreign key that refers to the fifth column of *Dataset2*. We used three test sets *Test 1*, *Test 2*, and *Test 3* with respective sizes 30 GB, 70 GB, and 110 GB. Table 4 summarizes the dataset sizes used in our experiments. The ratios of the joined records are 0.054 % (Test 1), 0.057 %(Test 2), 0.063 %(Test 3).

We executed our algorithm for the following join query.

```
SELECT *
FROM dataset1(column0..column20) d1, dataset2(column0..column20) d2
WHERE  d1.column0 = d2.column5
```

We particularly investigate four aspects: the number of intermediate tuples generated, the total execution time, the tasks timeline, and the scalability measured by varying the input size.

Evaluation of Approaches. In order to execute the filter-based algorithms efficiently, we specified the size of filters according to the cardinality of the join key values 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:

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

For a given false positive probability f, the size of the Bloom filter m is proportional to the number of elements n in the filter as shown in Table 5.

Table 5. Parameters for filters

Tests	f	ρ	n	m/n	m (bit)
Test 1	0.001	7	14,866	15	222,990
Test 2	0.0001	8	15,790	21	331,590
Test 3	0.0001	8	15,790	21	331,590

where ρ is the number of hash functions, and m/n is the number of bits allocated to each join key.

We can determine optimized parameters for the filter (e.g. f, ρ and m) [10]. In practice, however, we should choose values less than an optimized value to reduce computational overhead. As shown in Table 5, we deliberately select various values of f, ρ and m/n for the experiments to consider if they might affect our join performance. The filter files generated in the tests are compressed with gzip.

Table 6. Number of intermediate tuples (Map output)

Join algorithms	Test 1 (30 GB)	Test 2 (70 GB)	Test 3 (110 GB)
IFBJ	43,453	106,116	179,091
BJ	40,276,915	92,747,151	145,206,430
RSJ	80,320,684	185,098,062	284,510,488

The intermediate data size (Map output) is given in Table 6. The Reduce-side join (without filter) is the most inefficient solution, although it runs as a single job. This is correlated to the large size of intermediate data. Note that the number of intermediate tuples generated in this case is almost equal to the number of Map input records, see Tables 4 and 6. This slight difference is because a few tuples of *Dataset2* have less than 6 columns, and so they have been eliminated.

Filter-based joins are more efficient in general. BJ and IFBJ include the pre-processing job and the filtering operation to improve the join performance.

The number of intermediate tuples produced by BJ is considerably reduced with respect to RSJ. However, in comparison to IFBJ (see in Table 6), BJ still produces much more intermediate data because the filtering operation is only executed on one input dataset (*Dataset1*). This situation is overcome by IFBJ.

Looking at BJ and IFBJ, Table 6 points out that BJ generates more intermediate data than IFBJ. Namely, for the 110 GB test, BJ produces 145,206,430 intermediate tuples, whereas IFBJ produces 179,091 tuples. The experiments reported above are consistent with our theoretical analysis (Theorem 1).

Next, we evaluate the efficiency of these join algorithms by comparing the total execution time. As a general fact, the join algorithms generating less intermediate data turn out to be faster, even if we sum up the cost of the pre-processing and join jobs.

Table 7 gives the total execution time of the pre-processing job and the join job for each algorithm. Regarding pre-processing, the cost of the filter-based joins is related to the size of the data accessed to build the filter(s). In particular, IFBJ has to scan two input datasets. However, it pays off, since once the filters are available, the cost of join jobs is drastically reduced.

Figure 8 demonstrates that the best execution results from using intersection filters. Their total execution time is significantly reduced compared to BJ in spite of the time spent in the pre-processing job. The total execution time of

Table 7. Execution of pre-processing job and join job (in minutes)

Joins	Test 1 (30 GB)			Test 2 (70 GB)			Test 3 (110 GB)		
	Pre-proc.	*Join job*	*Total time*	*Pre-proc.*	*Join job*	*Total time*	*Pre-proc.*	*Join job*	*Total time*
IFBJ	3.17	6.15	9.32	6.45	24.25	31.10	10.00	92.12	102.12
BJ	2.12	17.07	19.19	3.63	43.63	47.26	5.22	139.58	145.20
RSJ	0	28.25	28.25	0	70.13	70.13	0	150.00	150.00

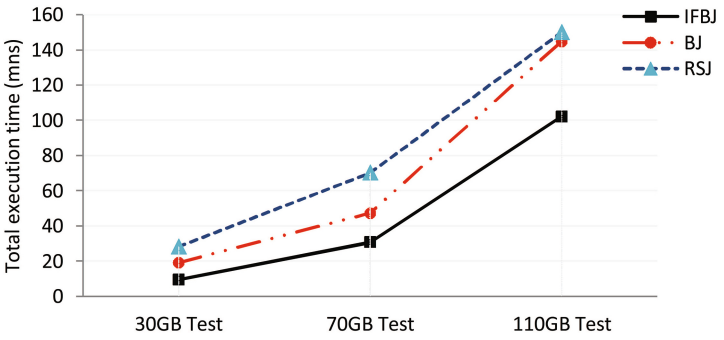


Fig. 8. Total execution time

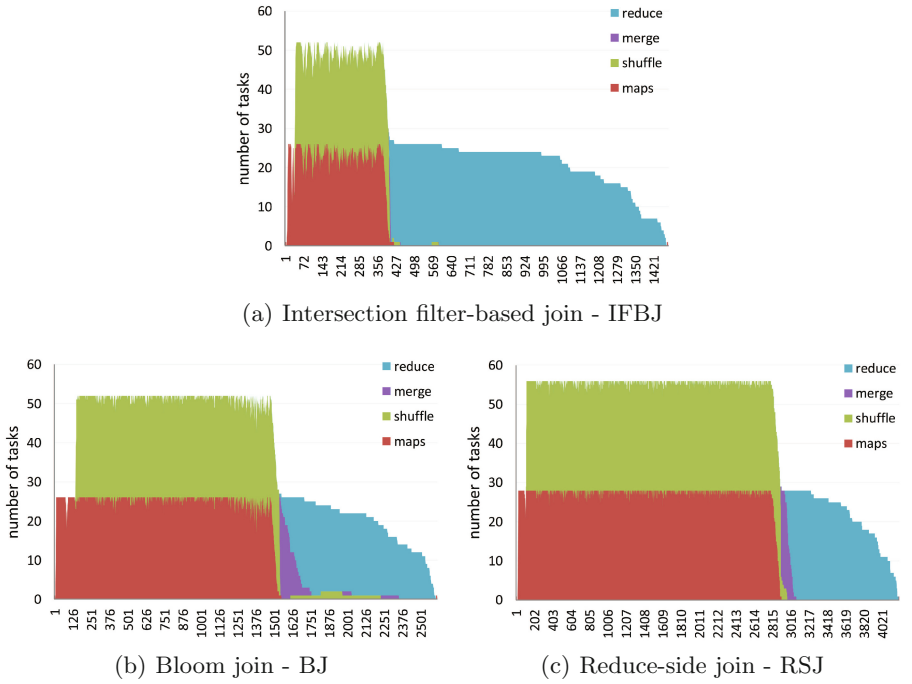
IFBJ increases from about 10 to 105 (mns), whereas that of BJ ranges from 19.19 to 145.20 (mns). The worst execution is RSJ, ranging 28.25 to 150 (mns). The smaller cost of IFBJ compared to the others (Table 7), is analyzed in Theorem 2.

Finally, we analyze the sequence of tasks during job execution (called task timelines). We do not examine the task timelines of the pre-processing job which is negligible compared to the join query over large datasets (see Table 7).

Figure 9 represents the task timelines of 70GB join jobs. These graphs are created by parsing log files generated by Hadoop during the job execution (555 map tasks and 28 reduce task, processing 185,205,828 input records and producing 26,062,967 output records). Each graph shows the respective timelines for map, shuffle and reduce phases.

There is a notable difference between the task timeline of IFBJ and that of other joins. The execution time of all map and reduce tasks of IFBJ, Fig. 9(a), is significantly reduced compared to BJ and RSJ, Fig. 9(b) and (c). Besides, the map and reduce phases of IFBJ finished earlier than BJ and RSJ because they produce less intermediate data and, as a consequence, the total cost of the local I/O, sort, and remote data copy is also smaller. Joins that use the intersection filter are the most efficient solutions because of their better data filtering efficiency.

The efficiency of filter-based joins depends on the ratio of non-matching tuples. The threshold is defined by the two parameters $\delta_{dataset2}$ and $\delta_{dataset1}$, which are the ratios of matching tuples. Figure 10 shows the execution time



(a) Intersection filter-based join - IFBJ

(b) Bloom join - BJ

(c) Reduce-side join - RSJ

Fig. 9. 70GB Task timelines during the execution of the join job

of algorithms for several values of these parameters, in order to identify their impact.

We start with an extreme case (first column) where domain of join attributes in *Dataset1* and *Dataset2* are disjoint. A IFBJ is then able to discover the empty intersection and the join job can therefore be omitted altogether, and their costs represents only that of the pre-processing job. This cost is roughly comparable with that of RSJ because of the small size of the dataset which make the join job fast enough. Filtered joins should not be used for small input datasets because the cost of building and broadcasting filter(s) becomes relatively significant.

We next examine the cases of a high ratio of matching tuples (85 % : 4 %) and (95 % : 65 %). They represent respectively the thresholds for filter-based join resulting from our analysis. Figure 10 clearly shows that this is the point where filters become counter-productive. This can be determined at compile time based on the ratios $\delta_{dataset2} : \delta_{dataset1}$.

The last case shows a join between fully matching datasets (100 % : 100 %), in which case RSJ is the best solution.

5.2 Multi-way Joins

Cluster Environment and Datasets. We run experiments for the chain join on another computer cluster of 15 virtual machines using KVM (Kernel-based

	0% : 0%	50% : 0,8%	85% : 4%	95% : 65%	100% : 100%
IFBJ	0.54	7.05	30.49	50.11	106.55
BJ	2.44	7.36	31.46	50.18	99.51
RSJ	1.34	7	31.25	50	84.5

Total execution time (mns)

Fig. 10. Identification of threshold for non-matching tuples for joins with 2 GB inputs

Virtual Machine) [18]. Each machine has two 1.4 Ghz AMD Opteron CP Us with 512 KB cache, 5 GB RAM and 100 GB SATA disks. We installed Hadoop [6] 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 joins. The number of reduce tasks is set to 25.

All 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 *column1* (c_1), *column2* (c_2), *column3* (c_3), and *column4* (c_4). Tables 8 and 9 summarizes the different input dataset sizes and the joined record ratios, resp.

The chain join algorithms developed in our experiments are the Reduce-side join cascade (CJ-RSJ), the Bloom join cascade (CJ-BJ), the IF-based join cascade (CJ-IFBJ), the optimized two-way join cascade (OCJ-2WJ), and the optimized three-way join cascade (OCJ-3WJ). 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.c2 = d2.c2 AND d2.c3 = d3.c3 AND d3.4 = d4.c4
```

Table 8. Input datasets used in three tests

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

Table 9. The ratios of the joined records of the datasets (%)

Inputs	Test 1	Test 2	Test 3
dataset1	0.721722639	0.304090688	0.123521020
dataset2	0.216530370	0.152050936	0.169996205
dataset3	0.721722639	0.304090688	0.123521020
dataset4	0.216530370	0.152050936	0.169996205

Evaluation. The experiments use the parameters of the Boom filters given in Table 10.

In order to confirm the cost model of chain joins (Sect. 4.2), we first examine the amount of intermediate data (Table 11)

Table 11 shows that CJ-RSJ and CJ-BJ generate much more intermediate data than any algorithms using the (extended) intersection filters. Figure 11 helps us

Table 10. Parameters of Bloom filters

Tests	f	ρ	n	m/n	m (bit)
Test 1	0.000101	8	13,147	21	276,087
Test 2	0.000101	8	13,840	21	290,640
Test 3	0.000101	8	15,295	21	321,195

Table 11. Number of intermediate tuples (all map outputs)

Chain join algorithms	Test 1 (26 GB)	Test 2 (60 GB)	Test 3 (100 GB)
CJ-IFBJ	1,309,349	1,469,048	1,497,692
CJ-BJ	45,402,907	89,201,979	89,248,190
CJ-RSJ	88,296,034	196,465,292	290,582,143
OCJ-2WJ	1,281,036	1,417,684	1,445,428
OCJ-3WJ	1,221,769	1,359,575	1,385,053

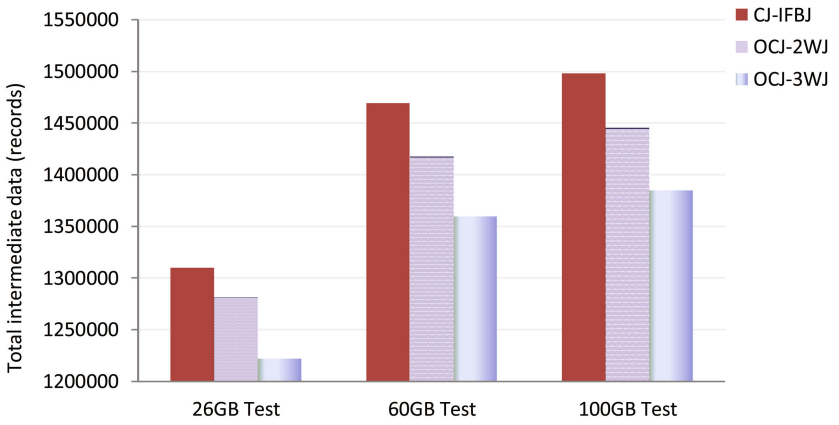


Fig. 11. Total intermediate data of the chain join

to obtain a visual comparison of the intersection filter-based chain joins. OCJ-3WJ has the least amount of intermediate data because it has the least number of jobs, and filters out almost all non-matching tuples in intermediate results. The intermediate data amount of OCJ-2WJ is slightly greater than the intermediate data amount of OCJ-3WJ, as analyzed by Theorem 5. However, OCJ-2WJ is still better than CJ-IFBJ chain joins which do not fully prevent non-matching tuples to propagate throughout the join chain.

Next, we examine the total output of the chain join algorithms (Fig. 12). The total output consists of all the intermediate data generated in the map phase

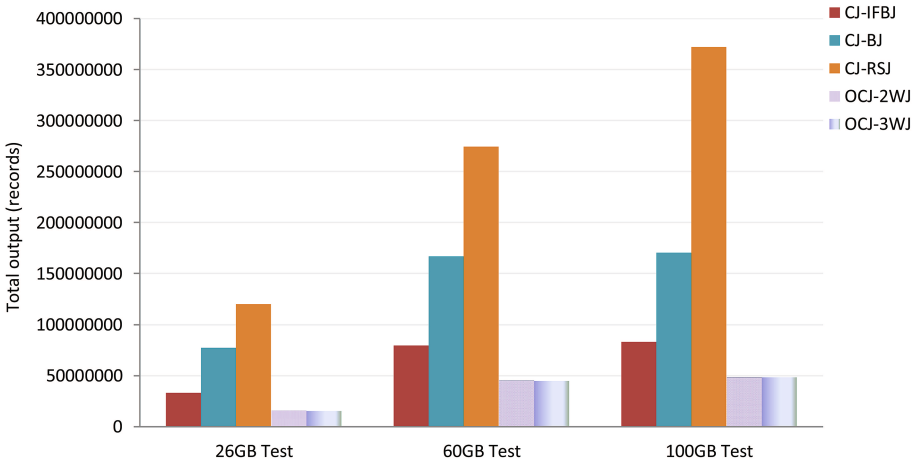


Fig. 12. Total output data (Map output + Reduce output)

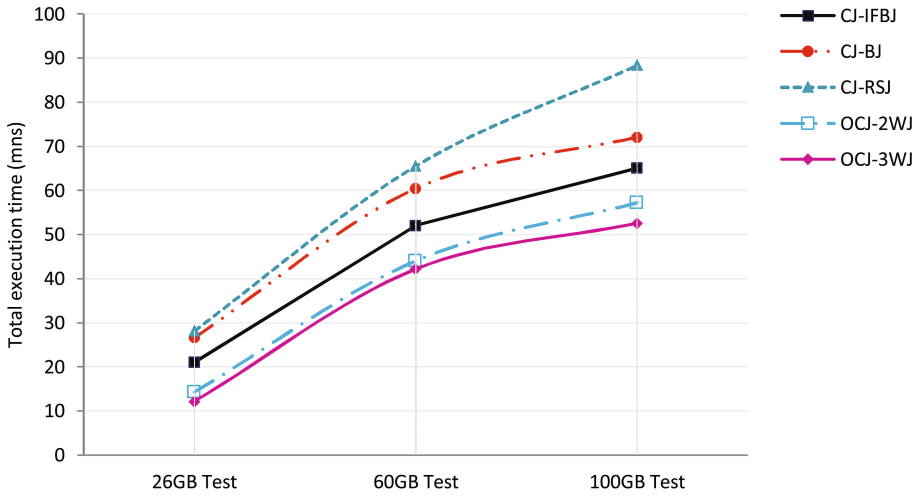


Fig. 13. Total execution time

and the intermediate join results. In other words, it includes all map output tuples and reduce output tuples produced during the chain join.

As shown in Fig. 12, CJ-RSJ and CJ-BJ generate the largest outputs; whilst the OCJ joins (e.g. OCJ-2WJ and OCJ-3WJ) using the extended intersection filters produce the least output. The CJ-IFBJ joins generally produce a little more output than the OCJ joins. The main reason is that the OCJ joins have the ability to filter out much more non-matching tuples than the others.

Both CJ-RSJ and CJ-BJ exhibit a similar pattern, with a significant cost increase from 26GB to 100GB. Obviously, CJ-RSJ has the highest cost with 119,928,957 tuples for Test 1, (77,035,830 for CJ-BJ and 32,942,272 for the CJ-IFBJ joins). This is even worse with Test 3, CJ-RSJ produces 371,782,345 tuples compared to 170,448,392 for CJ-BJ and 82,697,894 for the CJ-IFBJ joins.

Let us finally discuss the performance comparison, summarized by Fig. 13. The run time is clearly correlated to the size of the intermediate data, as confirmed by the comparison of the relative performance of the algorithms and the number of tuples shipped during the execution of joins.

The two bottom graphs show the total execution times of the OCJ joins (OCJ-3WJ and OCJ-2WJ), the next three ones deal with CJ-IFBJ, and the two top graphs show CJ-BJ and CJ-RSJ. For the largest dataset (100GB), OCJ-3WJ and OCJ-2WJ run time is about 52.57 and 57.22 min respectively, while the CJ-IFBJ joins run time is about 65.13 min. CJ-BJ and CJ-RSJ run time is much longer, about 72.09 and 88.34 min resp. This shows the high benefit of filtering out useless data, as carrying this data all over the process constitutes a strong penalty. Note that these costs include the pre-processing step for filter-based joins. In a scenario where filters are built once, and the joins processed many times, the benefit of the approach is even reinforced. The results of these experiments are consistent with our cost analysis presented in Sect. 4.2.

5.3 Recursive Joins

Cluster Environment and Datasets. We performed experiments on a HaLoop cluster running the modified version of Hadoop 0.20.2. The cluster consists of 12 PC computers. Each machine has two 2.53 GHz Intel(R) Core(TM)2 Duo CPUs with 3 MB cache, 3 GB RAM and 80 GB SATA disks. The operating system is 64-bit Ubuntu server 14.04 LTS, and the java version is 1.8.0.20. This cluster has one TaskTracker and one DataNode daemon running on each node. One of the nodes is selected to act as a master and run the NameNode and the JobTracker processes. TaskTracker nodes are configured to run up to two simultaneous map tasks and two reduce tasks. The HDFS block size was set to 128 MB, size of read/write buffer was 128 KB, and the number of reduce tasks is set to 16.

We use test datasets generated by the PUMA to conduct the experiments. The maximum number of columns in the datasets is 31 and string length in each column is set 19 characters. The input dataset *Know* contains two join key columns, namely, *column0*(c_0), and *column1*(c_1). Table 12 lists the different sizes of the dataset *Know* used in our tests.

Table 12. Input dataset *Know* with different sizes

Test	Size	Records
Test 1	10 GB	53,674,078
Test 2	20 GB	107,349,426
Test 3	30 GB	150,000,054

The following recursive join query is used to evaluate our experiments.

$$Friend(c_0, c_1, \ldots, c_{30}) \longleftarrow Know(c_0, c_1, \ldots, c_{30})$$
$$Friend(c_0, c_1, \ldots, c_{30}) \longleftarrow Friend(c_0, c_1, \ldots, c_{30}) \bowtie_{c1=c'0} Know(c'_0, c'_1, \ldots, c'_{30})$$

Evaluation. The filters' parameters used in the filter-based approach are listed in Table 13.

Table 13. Parameters of filters

Tests	f	ρ	n	m/n	m (bit)
Test 1	0.000101	8	7,111	21	149,331
Test 2	0.000101	8	7,123	21	149,583
Test 3	0.000101	8	7,130	21	149,730

Table 14. The total number of intermediate tuples

Recursive join approaches	Test 1 (10 GB)	Test 2 (20 GB)	Test 3 (30 GB)
REJ-SHAW	215,609,705	431,589,879	602,707,978
REJ-FB	188,597,706	377,403,437	527,220,188

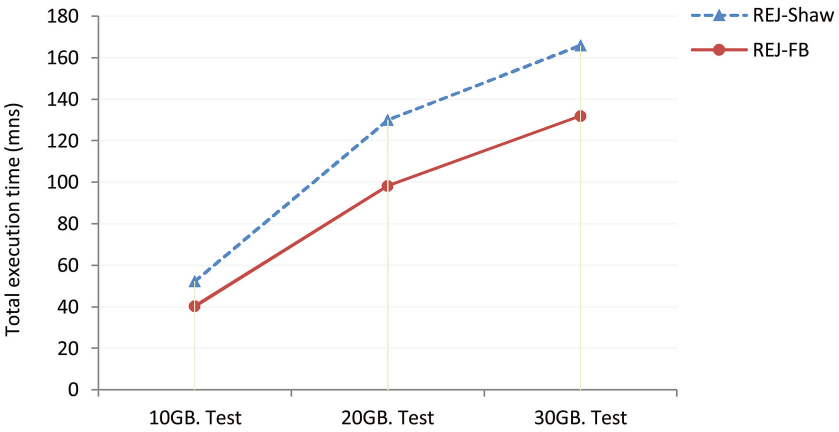


Fig. 14. Total execution time

We first examine the total map output (Table 14). The Shaw's approach (REJ-SHAW) generates more intermediate data than the filter-based approach (REJ-FB). For the tests from 10 GB to 30 GB, REJ-SHAW generates from 215,609,705 to 602,707,978 tuples, whilst REJ-FB has less than from 188,597,706 to 527,220,188 respectively. This is because the intermediate data of the join jobs in REJ-SHAW contains a lot of non-matching tuples, whereas REJ-FB uses the intersection filter to eliminate these non-matching tuples from the intermediate data of the join jobs.

Next, we examine the efficiency of the recursive join approaches. The total execution time of REJ-SHAW is compared to that of REJ-FB. Let us look in Fig. 14 for more details.

Figure 14 presents the total execution time of the pre-processing job and the iterative (join + dedup-diff) jobs for each algorithm. The cost of REJ-FB is considerably reduced in spite of the additional pre-processing job.

With the 10 GB input dataset *Know*, the total execution time of the Shaw's approach is higher than that of REJ-FB. This remains so through the other tests.

6 Conclusions and Future Work

The join operation is one of the essential operations for data analysis. Join evaluation is expensive and not straightforward to implement with MapReduce. This

paper makes three contributions. First, we attempt to gather in a uniform setting some of the main approaches recently proposed for the most common types of joins. In particular, we systematically considered the introduction of filters in execution plans. Filters are known to greatly improve the cost of distributed joins thanks to their ability to avoid network transfer of useless data. We showed how to adapt the join algorithms with filters, on a systematic basis. The second contribution is a modeling of cost that serves as a yardstick to compare the expected efficiency of joins. In particular, we characterize the situations where filters are indeed beneficial. Finally, we conducted a full set of experiments to validate our models, and reported the behavior of the join algorithms in a practical situation.

In general, join evaluation using filters is more efficient than other solutions since it reduces the need for shipping non-matching data. Specific situations may lead to reconsider this general assumption. For instance, in the case of a join between two relations linked by an integrity constraint (primary, foreign key), the system guarantees the inclusion of one key set into the other, and filtering becomes useless. Such structured datasets are arguably not common in the Big Data realms. As another example, small dataset size may reveal the cost of producing and shipping the filters. A direct join approach should be used in that case (in fact using MapReduce for small datasets is probably not a good idea in the first place). Our cost models help to detect those special cases and adopt the proper evaluation strategy.

The present study could be extended in several directions. First, a complete coverage would include star joins, and in general joins amongst n relations linked by complex relationships. Given the complexity of matching such a general setting with a MapReduce framework, we consider that the set of joins cases investigated in what precedes constitute a satisfying set of primitives to start with. Regarding our experimental evaluation, we did our best to use the state-of-the-art MapReduce framework (e.g., HaLoop). We note that some recent distributed engines (e.g., Spark [7], Stratosphere/Flink [5,34]) natively bring some of the features examined here, and notably iterations. At a physical level, they support caching of intermediate result, if possible in main memory. This strengthens our expectation that joins (including recursive joins) as studied here, constitute the basic building block of sophisticated algorithms for machine learning and data mining, which stand as the most promising outcome of Big Data processing in a near future. In this respect, the present study stands as a first step toward the design of an optimizer for distributed query processing, apt at considering complex integration of iterative, recursive and multi-set operators. We plan to investigate in the future the foundations of such an optimizer.

References

1. Afrati, F.N., Borkar, V., Carey, M., Polyzotis, N., Ullman, J.D.: Cluster computing, recursion and datalog. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 120–144. Springer, Heidelberg (2011)

2. Afrati, F.N., Borkar, V.R., Carey, M.J., Polyzotis, N., Ullman, J.D.: Map-reduce extensions and recursive queries. In: Proceedings of the International Conference on Extending Database Technology (EDBT), Uppsala, Sweden, pp. 1–8 (2011)
3. Afrati, F.N., Ullman, J.D.: Optimizing joins in a map-reduce environment. In: Proceedings of the International Conference on Extending Database Technology (EDBT), Lausanne, Switzerland, pp. 99–110 (2010)
4. Ahmad, F.: Puma benchmarks and dataset downloads (2012). https://engineering.purdue.edu/~puma/datasets.htm. Accessed: 18 June 2015
5. Apache: Flink. http://flink.apache.org. Accessed: 18 June 2015
6. Apache: Hadoop. http://hadoop.apache.org/. Accessed: 18 June 2015
7. Apache: Spark. https://spark.apache.org. Accessed: 18 June 2015
8. Blanas, S., Patel, J.M., Ercegovac, V., Rao, J., Shekita, E.J., Tian, Y.: A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD 2010, pp. 975–986. ACM, New York (2010)
9. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
10. Broder, A.Z., Mitzenmacher, M.: Survey: network applications of Bloom filters: a survey. Internet Math. **1**(4), 485–509 (2003)
11. Bruno, N., Kwon, Y., Wu, M.C.: Advanced join strategies for large-scale distributed computation. Proc. VLDB Endow. **7**(13), 1484–1495 (2014)
12. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDBJ **21**(2), 169–190 (2012)
13. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the International Symposium on Operating System Design and Implementation (OSDI), San Francisco, California, pp. 137–150 (2004)
14. Doulkeridis, C., Nrvg, K.: A survey of large-scale analytical query processing in mapreduce. VLDB J. **23**(3), 355–380 (2014)
15. Facebook,: Facebook reports fourth quarter and full year 2013 results - facebook (2014). http://investor.fb.com/releasedetail.cfm?ReleaseID=821954. Accessed: 18 June 2015
16. Hassan, M.A.H., Bamha, M.: Semi-join computation on distributed file systems using map-reduce-merge model. In: Proceedings of the Symposium on Applied Computing (SAC), Sierre, Switzerland, pp. 406–413 (2010)
17. Idreos, S., Liarou, E., Koubarakis, M.: Continuous multi-way joins over distributed hash tables. In: Proceedings of the EDBT, Nantes, France, pp. 594–605 (2008)
18. KVM: Kernel virtual machine. http://www.linux-kvm.org/page/Main_Page. Accessed: 18 June 2015
19. Lam, C.: Hadoop in Action. Manning Publications, Greenwich (2010)
20. Lee, K.H., Lee, Y.J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with mapreduce: a survey. SIGMOD Rec. **40**(4), 11–20 (2012)
21. Lee, T., Im, D.H., Kim, H., Kim, H.J.: Application of filters to multiway joins in MapReduce. Math. Probl. Eng. **2014**, 11 (2014)
22. Lee, T., Kim, K., Kim, H.J.: Join processing using Bloom filter in MapReduce. In: Proceedings of the RACS, San Antonio, TX, USA, pp. 100–105 (2012)
23. Lee, T., Kim, K., Kim, H.J.: Exploiting bloom filters for efficient joins in MapReduce. Inf. Int. Interdisc. J. **16**(8), 5869–5885 (2013)
24. Li, F., Ooi, B.C., Özsu, M.T., Wu, S.: Distributed data management using MapReduce. ACM Comput. Surv. **46**(3), 31:1–31:42 (2014)

25. Liu, L., Yin, J., Gao, L.: Efficient social network data query processing on MapReduce. In: Proceedings of the Workshop on HotPlanet, Hong Kong, China, pp. 27–32 (2013)
26. Nykiel, T., Potamias, M., Mishra, C., Kollios, G., Koudas, N.: MRShare: sharing across multiple queries in MapReduce. Proc. Very Large Data Bases Endowment (PVLDB) **3**(1), 494–505 (2010)
27. Okcan, A., Riedewald, M.: Processing theta-joins using mapreduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, pp. 949–960. ACM, New York (2011)
28. Oracle: Oracle vm virtualbox. https://www.virtualbox.org. Accessed: 18 June 2015
29. Ordonez, C.: Optimizing recursive queries in SQL. In: Proceedings of the SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, pp. 834–839 (2005)
30. Phan, T.C., d'Orazio, L., Rigaux, P.: Toward intersection filter-based optimization for joins in mapreduce. In: Proceedings of the 2nd International Workshop on Cloud Intelligence, Cloud-I 2013, pp. 2:1–2:8. ACM, New York (2013)
31. Sakr, S., Liu, A., Batista, D., Alomari, M.: A survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutorials **13**(3), 311–336 (2011)
32. Sakr, S., Liu, A., Fayoumi, A.G.: The family of mapreduce and large-scale data processing systems. ACM Comput. Surv. **46**(1), 11:1–11:44 (2013)
33. Shaw, M., Koutris, P., Howe, B., Suciu, D.: Optimizing large-scale Semi-Naive Datalog evaluation in Hadoop. In: Proceedings of the International Workshop on Datalog 2.0 (Datalog), Vienna, Austria, pp. 165–176 (2012)
34. Stratosphere: Next generation big data analytics platform. http://stratosphere.eu. Accessed: 18 June 2015
35. Tan, K.L., Lu, H.: a note on the strategy space of multiway join query optimization problem in parallel systems. SIGMOD Rec. **20**(4), 81–82 (1991)
36. Ullman, J.D.: Principles of Database and Knowledge-Base Systems, vol. I. Computer Science Press, Rockville (1988)
37. White, T.: Hadoop: The Definitive Guide. O'Reilly, Sebastopol (2012)
38. Zhang, C., Li, J., Wu, L., Lin, M., Liu, W.: Sej: an even approach to multiway theta-joins using mapreduce. In: CGC 2012, pp. 73–80. IEEE Computer Society (2012)
39. Zhang, C., Wu, L., Li, J.: Optimizing distributed joins with bloom filters using MapReduce. In: Kim, T., Cho, H., Gervasi, O., Yau, S.S. (eds.) GDC, IESH and CGAG 2012. CCIS, vol. 351, pp. 88–95. Springer, Heidelberg (2012)
40. Zhang, C., Wu, L., Li, J.: Efficient processing distributed joins with bloom filter using mapreduce. Int. J. Grid Distrib. Comput. (IJGDC) **6**(3), 43–58 (2013)
41. Zhang, X., Chen, L., Wang, M.: Efficient multi-way theta-join processing using mapreduce. Proc. VLDB Endow. **5**(11), 1184–1195 (2012)