Distributed Data Deduplication

Xu Chu
University of Waterloo
x4chu@uwaterloo.ca

Ihab F. Ilyas
University of Waterloo
ilyas@uwaterloo.ca

Paraschos Koutris
University of
Wisconsin-Madison
paris@cs.wisc.edu

ABSTRACT

Data deduplication refers to the process of identifying tuples in a relation that refer to the same real world entity. The complexity of the problem is inherently quadratic with respect to the number of tuples, since a similarity value must be computed for every pair of tuples. To avoid comparing tuple pairs that are obviously non-duplicates, blocking techniques are used to divide the tuples into blocks and only tuples within the same block are compared. However, even with the use of blocking, data deduplication remains a costly problem for large datasets. In this paper, we show how to further speed up data deduplication by leveraging parallelism in a shared-nothing computing environment. Our main contribution is a distribution strategy, called Dis-Dedup, that minimizes the maximum workload across all worker nodes and provides strong theoretical guarantees. We demonstrate the effectiveness of our proposed strategy by performing extensive experiments on both synthetic datasets with varying block size distributions, as well as real world datasets.

1. INTRODUCTION

Data deduplication, also known as record linkage, or entity resolution, refers to the process of identifying tuples in a relation that refer to the same real world entity. Data deduplication is a pervasive problem, and is extremely important for data quality and data integration [12]. For example, finding duplicate customers in enterprise databases is essential in almost all levels of business. In our collaboration with Thomson Reuters, we observed that a data deduplication project takes 3-6 months to complete, mainly due to the scale and variety of data sources.

Data deduplication techniques usually require computing a similarity score of each tuple pair. For a dataset with n tuples, naïvely comparing every tuple pair requires $O(n^2)$ comparisons, a prohibitive cost when n is large. A commonly used technique to avoid the quadratic complexity is blocking [4, 6, 16], which avoids comparing tuple pairs that are obviously not duplicates. Blocking methods first partition all records into blocks and then only records within the same block are compared. A simple way to perform blocking is to scan all records and compute a hash value for each record based on a subset of its attributes, commonly referred to as *blocking key attributes*. The computed hash values are called *blocking key values*. Records with the same blocking key values are grouped into the same block. For example, blocking key attributes can be the zipcode, or the first three characters of the last name. Since one blocking function might miss placing duplicate tuples in the same block, thus resulting in a false negative (for example, zipcode can be wrong or obsolete), multiple blocking functions [12] are often employed to reduce the number of false negatives.

Despite the use of blocking techniques, data deduplication remains a costly process that can take hours to days to finish for real world datasets on a single machine [21]. Most of the previous work on data deduplication is situated in a centralized setting [4, 6, 7], and does not leverage the capabilities of a distributed environment to scale out computation; hence, it does not scale to large distributed data. Big data often resides on a cluster of machines interconnected by a fast network, commonly referred to as a "data lake". Therefore, it is natural to leverage this scale-out environment to develop efficient data distribution strategies that parallelize data deduplication. Multiple challenges need to be addressed to achieve this goal. First, unlike centralized settings, where the dominating cost is almost always computing the similarity scores of all tuple pairs, multiple factors contribute to the elapsed time in a distributed computing environment, including network transfer time, local disk I/O time, and CPU time for pair-wise comparisons. These costs also vary across different deployments. Second, as it is typical in a distributed setting, any algorithm has to be aware of data skew, and achieve load-balancing [5, 11]. Every machine must perform a roughly equal amount of work in order to avoid situations where some machines take much longer than others to finish, a scenario that greatly affects the overall running time. We show the effect of data skew when we discuss distribution strategies in Section 4. Third, the distribution strategy must be able to handle effectively multiple blocking functions; as we show in this paper, the use of multiple blocking functions impacts the number of times each tuple is sent across nodes, and also induces redundant comparisons when a tuple pair belongs to the same block according to multiple blocking functions.

A recent work Dedoop [19, 20] uses MapReduce [10] for data deduplication. However, it only optimizes for compu-

Proceedings of the VLDB Endowment, Vol. 9, No. 11
Copyright 2016 VLDB Endowment 2150-8097/16/07.

tation cost, and requires a large memory footprint to keep the necessary statistics for its distribution strategy, thus limiting its performance and applicability, as our experiments show in Section 6. The problem of data deduplication is also related to distributed join computation, which will be discussed in detail in Section 7. However, parallel join algorithms are not directly applicable to our setting: (1) most of the work on parallel join processing [22, 2] is for two-table joins, so the techniques are not directly applicable to self-join without wasting almost half of the available workers, as shown in Section 3; (2) even with an efficient self-join implementation, applying it to every block directly without considering the block sizes yields a sub-optimal strategy, as shown in Section 4.2; and (3) to the best of our knowledge, there is no existing work on processing a disjunction of join queries, a problem we have to tackle in dealing with multiple blocking functions.

In this paper, we propose a distribution strategy with optimality guarantees for distributed data deduplication in a shared-nothing environment. Our proposed strategy aims at minimizing elapsed time by minimizing the maximum cost across all machines. Note that while blocking affects the quality of results (by introducing false negatives), we do not introduce a new blocking criteria, rather we show how to execute a given set of blocking functions in a distributed environment. In other words, our technique does not change the quality but tackles the performance of the deduplication process. We make the following contributions:

- We introduce a cost model that consists of the maximum number of input tuples any machine receives (X), and the maximum number of tuple pair comparisons any machine performs (Y) (Section 2). We provide a lower bound analysis for X and Y that is independent of the actual dominating cost in a cluster.
- We propose a distribution strategy for distributing the workload of comparing tuples in a single block (Section 3). Both X and Y of our strategy are guaranteed to be within a small constant factor from the lower bound X_{low} and Y_{low}.
- We propose Dis-Dedup for distributing a set of blocks produced by a single blocking function. The X and Y of Dis-Dedup are both within a small constant factor from X_{low} and Y_{low}, regardless of block-size skew (Section 4). Dis-Dedup also handles multiple blocking functions effectively, and avoids producing the same tuple pair more than once even if that tuple pair is in the same block according to multiple blocking functions (Section 5).

We perform extensive experiments on synthetic datasets with varying block-size skew, and on real datasets. Our experiments demonstrate the effectiveness of our proposed distribution strategy (Section 6).

2. PROBLEM DEFINITION AND SOLUTION OVERVIEW

In this section, we present the parallel computation model we will use in this paper. We formally introduce the problem definition, and provide an overview of our solution.

2.1 Parallel Computation Model

We focus on scale-out environments, where data is usually stored in what is called a *data lake*. Such an environment usually adopts a shared-nothing architecture, where multiple machines, or nodes, communicate via a high-speed interconnect network, and each node has its own private memory and disk. In every node there are typically multiple virtual processors running together, so as to take advantage of the multiple CPUs and disks available on each machine and thus increase parallelism. These virtual processors that run in parallel are called *workers* in this paper.

In a shared-nothing system, there is usually a trade-off between the *communication cost* and the *computation cost* [26]. For a particular data processing task, it is often hard to predict which type of cost is dominating, let alone constructing an objective function that combines these two costs. In addition, the influence of each cost on the running time is dependent on many parameters of the cluster configuration. For example, there are more than 250 parameters that are tunable in a Hadoop cluster[1]. In this paper, we follow a similar strategy used in parallel join processing [22], and seek to minimize both costs simultaneously.

Since all workers are running in parallel, to minimize the overall elapsed time, we focus on minimizing the largest cost across all workers. For worker i, let X_i be the communication cost, and Y_i be the computation cost. Assume that there are k workers available. We define X (resp. Y) to be the maximum X_i (resp. Y_i) at any worker:

$$X = \max_{i \in [1,k]} X_i \qquad\qquad Y = \max_{i \in [1,k]} Y_i \qquad (1)$$

A typical example of a parallel shared-nothing system is MapReduce [10]. MapReduce has two types of workers: the *mapper* and the *reducer*. A mapper takes a key-value pair, and generates a list of key-value pairs; while a reducer takes a key associated with a list of values, and generates another list of key-value pairs. The input keys of the reducers are the output keys of the mappers. Users have the option of implementing a customized *partitioner*. Partitioners decide which key-value pairs are sent to which reducers, based on the key. MapReduce balances the load between mappers very well, however, it is the programmers' responsibility to ensure that the workload across different reducers is balanced.

2.2 Formal Problem Definition

We are given a dataset I with n tuples, s blocking functions $h_1, \ldots, h_s$, and a tuple-pair similarity function f. Every blocking function $h \in \{h_1, \ldots, h_s\}$ is applied to every tuple t, and returns a blocking key value $h(t)$. A blocking function h divides all n tuples into a set of m blocks $\{B_1, B_2, \ldots, B_m\}$, where tuples in the same block have the same blocking key value. Tuple pairs in the same block are compared using f to obtain a similarity score. Based on the similarity scores, a clustering algorithm is then applied to group tuples together.

We perform data deduplication using the computational model described in Section 2.1. In this case, X_i represents the number of tuples that Worker i receives; and Y_i represents the number of tuple pair comparisons that Worker i performs. We aim at designing a distribution strategy that minimizes (a) the maximum number of tuples any worker receives, namely, X, and (b) the maximum number of tuple pair comparisons any worker performs, namely, Y, at the same time. This may not be possible for a given data deduplication task, but we show that we can always achieve optimality for both X and Y within constant factors. Hence, our algorithm will perform optimally independent of how the runtime is as a function of X and Y.

[1] `http://tinyurl.com/puqduqj`

Example 1. *Consider a scenario where a single blocking function produces few large blocks and many smaller blocks. To keep the example simple, suppose that a blocking function partitions a relation of* $n = 100$ *tuples into* 5 *blocks of size* 10 *and* 25 *blocks of size* 2*. The total number of comparisons* W *in this case is* $W = 5 \cdot \binom{10}{2} + 25 \cdot \binom{2}{2} = 250$ *comparisons.*

Assume $k = 10$ *workers. Consider first a strategy that sends all tuples to every worker. In this case,* $X_i = 100$ *for every worker* i*, which results in* $X = 100$ *according to Equation* (1)*. We then assign* $Y_i = \frac{W}{k} = 25$ *comparisons to worker* i *(for example by assigning to worker* i *tuple pairs numbered* $[(i-1)\frac{W}{k}, i\frac{W}{k}]$*). Therefore,* $Y = 25$ *according to Equation* (1)*.This strategy achieves the optimal* Y*, since* W *is evenly distributed to all workers. However, it has a poor* X*, since every tuple is replicated* 10 *times.*

Consider a second strategy that assigns one block entirely to one worker. For example, we could assign each of the 5 *blocks of size* 10 *to the first* 5 *workers, and to each of the remaining* 5 *workers we assign* 5 *blocks of size* 2*. In this case,* $X = 10$*, since each worker receives exactly the same number of tuples; moreover, each tuple is replicated exactly once. However, even though the input is evenly distributed across workers, the number of comparisons is not. Indeed, the first* 5 *workers perform* $\binom{10}{2} = 45$ *comparisons, while the last* 5 *workers perform only* $5 \cdot \binom{2}{2} = 5$ *comparisons. Therefore,* $Y = 45$ *according to Equation* (1)*.*

The above example demonstrates that the distribution strategy has significant impact on both X and Y, even in the case of a single blocking function. In the next three sections, we show how we can construct a distribution strategy that achieves an optimal behavior for both X and Y for any distribution of block sizes, and outperforms in practice any alternative strategies.

2.3 Solution Overview

Consider a blocking function h that produces m blocks $B_1, B_2, \ldots, B_m$. A distribution strategy would have to assign, for every block B_i a subset of the k workers of size $k_i \leq k$ to handle B_i.

A straightforward strategy assigns one block entirely to one worker, i.e., $k_i = 1, \forall i \in [1, m]$, hence, parallelism happens only across blocks. Another straightforward strategy uses all the available workers to handle every block, i.e., $k_i = k, \forall i \in [1, m]$, hence, parallelism is maximized for every block, and uses an existing parallel join algorithms [2, 22] to handle every block. However, both strategies are not optimal, as we will show in Section 4.2.

In light of these two straightforward strategies, we first study how to distribute the workload of one block B_i to k_i workers to minimize X and Y (Section 3). Given the distribution strategy for a single block, we then show how to assign workers to blocks $B_1, \ldots, B_m$ generated by a single blocking function h, so as to minimize both X and Y across all blocks (Section 4). Given the distribution strategy for a single blocking function h, we will finally present how to assign workers given multiple blocking functions $h_1, \ldots, h_s$, so that the overall X and Y are minimized (Section 5).

The optimality results we will show next for X and Y hold as long as the distribution strategy can be implemented in a shared-nothing system, where the distribution strategy specifies (1) which tuples are sent to which workers; and (2) which tuple pairs are compared inside each worker. We provide in the Appendix of the full paper[2] [9] the general description of the distribution strategy. However, whether our distribution strategy can be implemented in a particular shared-nothing system depends on the APIs of the system. For simplicity, we describe and implement our distribution strategy using MapReduce, which uses mappers and partitioners to specify (1) and uses reducers to specify (2). Other example platforms where our distribution strategies could be implemented include Spark [28], Apache Flink (previously known as Stratosphere) [3], and Myria [15].

3. SINGLE BLOCK DEDUPLICATION

In this section, we study the problem of data deduplication for tuples in a single block that is produced by one blocking function; in other words, we need to compare every tuple with every other tuple in the block. The distribution strategy presented in this section will serve as a building block when discussing distribution strategies in Sections 4 and 5. Assume that there are n tuples in the block, and k available reducers to compute the pair-wise similarities.

3.1 Lower Bounds

We first analyze the lower bounds X_{low} and Y_{low} for X and Y, respectively. The lower bounds are necessary to reason about the optimality of our distribution strategies.

Theorem 1. *For any distribution strategy that performs data deduplication on a block of size* n *using* k *reducers, the maximum input is* $X > X_{low} = \frac{n}{\sqrt{k}}$ *and the maximum number of comparisons is* $Y \geq Y_{low} = \frac{n(n-1)}{2k}$*.*

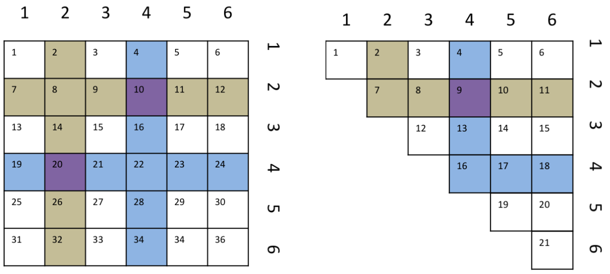
Proof. To show the lower bound on the number of comparisons Y, observe that the total amount of comparisons required is $\binom{n}{2} = \frac{n(n-1)}{2}$. Since there are k available reducers, there must exist at least one reducer j with $Y_j \geq \frac{n(n-1)}{2k}$.

For the lower bound on the input, suppose for the sake of contradiction that the maximum input is $n' \leq n/\sqrt{k}$. Then, each reducer will perform at most $\binom{n'}{2}$ comparisons, which means that the total number of comparisons will be at most $k\binom{n'}{2} = n(n - \sqrt{k})/2 < n(n-1)/2$, a contradiction (since the comparisons must be at least $\binom{n}{2}$). □

As we show in Section 3.2, our algorithm matches the lower bound Y_{low}, but not X_{low}. The problem of designing a strategy that matches X_{low} is tightly related to an extensively studied problem in combinatorics called *covering design* [25]. A (n, ℓ, t)-covering design is a family $\mathcal{F}$ of subsets of size ℓ from the universe $\{1, \ldots, n\}$, such that every subset of size t from the universe is a subset of a set in $\mathcal{F}$. The task in hand is to compute the minimum size $C(n, \ell, t)$ of such a family. To see the connection with our distribution problem, consider a $(n, X, 2)$-covering design $\mathcal{F}$ of size k. Then, we can assign to each of the k reducers a set from the family (that will be of size X); but now, we can perform every comparison in some reducer, since the covering design guarantees that every subset of size 2 (i.e., every pair) will be in some set (i.e., in some reducer). Thus, designing a strategy that achieves X_{low} means finding a $(n, X_{low}, 2)$-covering design, such that $C(n, X_{low}, 2) \leq k$.

The lower bound for X presented in Theorem 1 is called the Schönheim bound [25], but the only constructions that match it are explicit constructions for fixed values of n, ℓ. There exists a large literature of such constructions [14],

[2] https://cs.uwaterloo.ca/~x4chu/CS-2016-02.pdf



(a) $R \times S$ join (b) Self-join

Figure 1: Reducer arrangement. (The number in the upper left corner of each cell is the reducer id.)

and it is an open problem to find tight upper and lower bounds. Hence, instead of looking for an optimal solution, our algorithm provides a constant-factor approximation of the lower bound.

3.2 Triangle Distribution Strategy

We present here a distribution strategy, called *triangle distribution strategy*, which guarantees with high probability a small constant-factor approximation of the lower bounds.

The name of the distribution strategy comes from the fact that we arrange the k reducers in a triangle whose two sides have size l (thus $k = l(l+1)/2$ for some integer l). To explain why we organize the reducers in such a fashion, consider the scenario studied in [2, 5] where we compute the cartesian product $R \times S$ of two relations of size n: in this case, the reducers are organized in a $\sqrt{k} \times \sqrt{k}$ square, as shown in Figure 1(a) for $k = 36$. Each tuple from R is sent to the reducers of a random row, and each tuple from S is sent to all the reducers of a random column; the reducer function then computes all pairs it receives. However, if we apply this idea directly to a self-join (where $R = S$), the comparison of each pair would be repeated twice, since if a tuple pair ends up together in the reducer (i, j), it will also be in the reducer (j, i). For example, in Figure 1(a) tuple t_1 is sent to all reducers in row 2 and column 2, and tuple t_2 is sent to all reducers in row 4 and column 4. Therefore, the joining of t_1 and t_2 is duplicated at reducers $(2, 4)$ and $(4, 2)$. Because of the symmetry, the lower left half of the reducers in the square are doing redundant work. Arranging the reducers in a triangle circumvents this problem and allows us to use all available reducers.

Figure 1(b) gives an example of such an arrangement for $k = 21$ reducers with $l = 6$. Every reducer is identified by a two dimensional index (p, q), where p is the row index, and q is the column index, and $1 \leq p \leq q \leq l$. Each reducer (p, q) has a unique reducer ID, which is calculated as $(2l - p + 2)(p - 1)/2 + (q - p + 1)$. For example, Reducer $(2, 4)$ marked purple in Figure 1(b) is Reducer 9.

Algorithm 1 describes the distribution strategy given the arrangement for the k reducers. For any tuple t, the mapper randomly chooses an integer a, called an *anchor*, between $[1, l]$, and distributes t to all reducers whose row or column index $= a$ (Lines 3-11). By replicating each tuple l times, we can ensure that for every tuple pair, there exists at least one reducer that receives both tuples. In fact, if two tuples have different anchor points, there is *exactly one* reducer that receives both tuples; while if two tuples have the same anchor point a, both tuples will be replicated on the same set of reducers, but we only compare the tuple pair on reducer (a, a). The key of the key-value pair of the mapper output is the reducer id, and the value of the key-value pair of the mapper output is the tuple augmented with a flag L, S or R to avoid comparing tuple pairs that have the same anchor points a in reducers other than (a, a). Within each reducer, tuples with flag L are compared with tuples with flag R (Lines 28-31), and tuples with flag S are compared only with each other (Lines 33-35).

Algorithm 1 Triangle distribution strategy

```
1:  class Mapper
2:      method Map(Tuple $t$, null)
3:          Int $a \leftarrow$ a random value from [1,l]
4:          for all $p \in [1, a)$ do
5:              $rid \leftarrow$ rid of Reducer $(p, a)$
6:              Emit(Int $rid$, $L\#$Tuple $t$)
7:          $rid \leftarrow$ rid of Reducer $(a, a)$
8:          Emit(Int $rid$, $S\#$Tuple $t$)
9:          for all $q \in (a, l]$ do
10:             $rid \leftarrow$ rid of Reducer $(a, q)$
11:             Emit(Int $rid$, $R\#$Tuple $t$)
12: class Partitioner
13:     method Partition(Key $rid$, Value $v$, $k$)
14:         Return $rid$
15: class Reducer
16:     method Reduce(Key $rid$, Values $[v_1, v_2 \ldots]$)
17:         $Left \leftarrow \emptyset$
18:         $Right \leftarrow \emptyset$
19:         $Self \leftarrow \emptyset$
20:         for all Value $(v) \in$ Values $[v_1, v_2 \ldots]$ do
21:             $t \leftarrow v.subString(2)$
22:             if $v$ starts with L then
23:                 $Left \leftarrow Left + t$
24:             else if $v$ starts with R then
25:                 $Right \leftarrow Right + t$
26:             else if $v$ starts with S then
27:                 $Self \leftarrow Self + t$
28:         if $Left \neq \emptyset$ and $Right \neq \emptyset$ then
29:             for all $(t_1) \in Left$ do
30:                 for all $(t_2) \in Right$ do
31:                     compare $t_1$ and $t_2$
32:         else
33:             for all $(t_1) \in Self$ do
34:                 for all $(t_2) \in Self$ do
35:                     compare $t_1$ and $t_2$
```

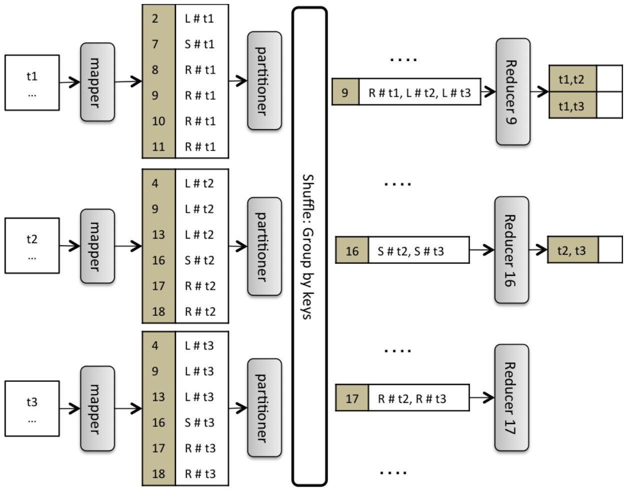


Figure 2: Single block distribution example using three tuples, given reducers in Figure 1(b)

Example 2. *Figure 2 gives an example for three tuples t_1, t_2, t_3 given the arrangement of the reducers in Figure 1(b). Suppose that tuple t_1 has anchor point $a = 2$, and tuples*

t_2, t_3 have the same anchor point $a = 4$. The mapping function takes t_1 and generates the key-value pairs $(2, L\#t_1)$, $(7, S\#t_1)$,$(8, R\#t_1)$,$(9, R\#t_1)$,$(10, R\#t_1)$,$(11, R\#t_1)$. Note the different tags L, S, R for different key-value pairs. Reducer 9 receives a list of values $R\#t_1, L\#t_2, L\#t_3$ associated with key 9, and compares tuples marked with R with tuples marked with L, but not tuples marked with the same tag. Reducer 16 receives a list of values $S\#t_2, S\#t_3$, and performs comparisons among all tuples marked with S.

Theorem 2. *The distribution of Algorithm 1 achieves with high probability*[3] *maximum input $X \leq (1+o(1))\sqrt{2}X_{low}$ and maximum number of comparisons $Y \leq (1 + o(1))Y_{low}$.*

We present the detailed proof in the full paper [9], and we provide the intuition here. Fix a reducer $i = (p, q)$. If $p \neq q$, the reducer will receive in expectation n/l tuples with flag L (the ones with anchor p) and n/l tuples with flag R (the ones with anchor R), therefore in expectation $X_i = 2n/l$ and $Y_i = n^2/l^2$. If $p = q$, the reducer will receive in expectation n/l tuples with flag S, therefore in expectation $X_i = n/l$ and $Y_i = n^2/2l^2$. We can show that the X_i and Y_i will also be concentrated around the expectation, and since $k = l(l+1)/2$, we have $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ and $Y \approx \frac{n^2}{2k}$. Comparing X, Y with the lower bounds in Theorem 1, we have Theorem 2.

What happens if the k reducers cannot be arranged in a triangle? Following the same idea that applies when k reducers cannot be arranged in a square for an $R\times S$ join [8], we choose the largest possible integer l', such that $l'(l' + 1)/2 \leq k$. We have $l'(l' + 1)/2 = k' \leq k$ and $(l' + 1)(l' + 2)/2 > k$. Since both $(l' + 1)(l' + 2)/2$ and k are integers, we have $(l' + 1)(l' + 2)/2 - 1 \geq k$. Therefore, the reducer utilization rate is $u = \frac{k'}{k} \geq \frac{l'(l'+1)/2}{(l'+1)(l'+2)/2-1} = 1- \frac{2}{l'+3} \geq 0.5$. Even for $k = 50$ reducers, we have $l' = 9$, and $u = 0.83$. Observe also that the utilization rate u increases as l' grows.

4. DEDUPLICATION USING SINGLE BLOCKING FUNCTION

In this section, we study distribution strategies to handle a set of disjoint blocks $\{B_1, \ldots, B_m\}$ produced by a single blocking function h. Let m denote the number of blocks, and for each block B_i, where $i \in [1, m]$, we denote by $W_i = \binom{|B_i|}{2}$ the number of comparisons needed. Thus, the total number of comparisons across all blocks is $W = \sum_{i=1}^{m} W_i$.

We will discuss the case where a single blocking function produces a set of overlapping blocks when we discuss multiple blocking functions in Section 5.

4.1 Lower Bounds

We first prove a lower bound on the maximum input size X and maximum number of comparisons Y for any reducer.

Theorem 3. *For any distribution strategy that performs data deduplication for n tuples and W total comparisons resulting from a set of disjoint blocks using k reducers, we have $X \geq X_{low} \geq \max(\frac{n}{k}, \frac{\sqrt{2W}}{\sqrt{k}})$ and $Y \geq Y_{low} = \frac{W}{k}$.*

Proof. Since the total amount of comparisons required is W, there must exist at least one reducer j such that $Y_j \geq \frac{W}{k}$. To prove a lower bound for the maximum input, consider the input size X_j of the reducer j. The maximum number of comparisons that can be performed will then be $X_j(X_j - 1)/2$, which happens when all tuples of the input belong in the same block. Hence, $Y_j \leq X_j(X_j - 1)/2 < X_j^2/2$. Since $Y_j \geq \frac{W}{k}$, we obtain that $X_j^2 > 2W/k$. The $X \geq n/k$ bound comes from the fact that every tuple will have to be sent to at least one reducer, and thus the total size of the inputs must be at least n. □

Notice that Theorem 1 can be viewed as a simple corollary of the above lower bound, since in the case of a self-join we have a single block of size n, so $W = \binom{n}{2}$.

4.2 Baseline Distribution Strategies

Assume we have k reducers to handle a set of blocks $B_1, B_2, \ldots, B_m$ produced by a single blocking function. We analyze the baseline strategies Naive-Dedup and PJ-Dedup.

The first baseline strategy Naive-Dedup assigns every block B_i entirely to one reducer. Consider the scenario where there exists a single block B_1 with $|B_1| = n$; then, Naive-Dedup assigns B_1 to one reducer, resulting in $X = n$ and $Y = W$, which is k times worse than X_{low} and Y_{low} (cf. Section 4.1), completely defeating the purpose of having k reducers. However, there are scenarios where Naive-Dedup behaves optimally as we show in Example 3.

The second baseline strategy PJ-Dedup uses all k reducers to handle every block B_i, and it uses the triangle distribution strategy discussed in Section 3 to perform self-join for every block. However, instead of invoking Algorithm 1 m times for every block $B_i, \forall i \in [1, m]$, which includes the overhead of initializing m MapReduce jobs, we design PJ-Dedup to distribute the tuples as if there was a single block (hence using the triangle distribution strategy of a self-join), and then perform grouping into the smaller blocks inside the reducers. The mapper of PJ-Dedup is similar to that of Algorithm 1, except that the key of the mapper output is a *composite key*, which includes both the reducer ID (as in Algorithm 1) and the blocking key value. The partition function of PJ-Dedup simply takes the composite key and returns the reducer ID part. The reduce function of PJ-Dedup is exactly the same as that of Algorithm 1, since the MapReduce framework automatically groups by blocking key values within each reducer. Regardless of the block sizes, PJ-Dedup has $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ because the tuples are sent by the mappers in the same way as Algorithm 1, and $Y \approx \frac{W}{k}$ because the workload W is roughly evenly distributed amongst k reducers. The formal proof for the behavior of PJ-Dedup and an example of PJ-Dedup are provided in the full paper [9].

Example 3. *We consider three blocking functions that generate blocks of different sizes. For each of them, we show the lower bounds X_{low} and Y_{low}, and analyze how* Naive-*Dedup, and* P*J-Dedup perform w.r.t. those lower bounds. For* P*J-Dedup, regardless of the blocking function, $X \approx \frac{\sqrt{2}n}{\sqrt{k}}$ and $Y \approx \frac{W}{k}$, as explained previously.*

(1) The first blocking function h_1 produces βk blocks of equal size, for an integer $\beta > 1$, that is, $|B_i| = \frac{n}{\beta k}$ for all $i \in [1, \beta k]$. In this case, $W = \frac{\frac{n}{\beta k}(\frac{n}{\beta k}-1)}{2}\beta k$, and thus Theorem 3 gives us $X_{low} = \frac{n}{k}$ and $Y_{low} = \frac{W}{k}$. For Naive-*Dedup, every reducer receives $\frac{\beta k}{k} = \beta$ blocks. Therefore, $X = \frac{n}{\beta k}\beta = \frac{n}{k}$ and $Y = \frac{\frac{n}{\beta k}(\frac{n}{\beta k}-1)}{2}\beta = \frac{W}{k}$, which is optimal.*

[3] The term "with high probability" means that the probability of success is of the form $1 - 1/f(n)$, where $f(n)$ is some polynomial function of the size of the dataset n.

(2) The second blocking function h_2 produces only one block of size n. In this case, $W = \frac{n(n-1)}{2}$, and thus Theorem 3 gives us $X_{low} \approx \frac{n}{\sqrt{k}}$ and $Y_{low} = \frac{W}{k}$. For N*aive-Dedup, one reducer does all the work, and thus $X = n$ and $Y = W$.*

(3) The third blocking function h_3 produces $\frac{k}{\beta}$ blocks of equal size for some $1 \leq \beta < k$, that is, $|B_i| = \frac{\beta n}{k}$ for all $i \in [1, \frac{k}{\beta}]$. In this case, $W = \frac{\frac{\beta n}{k}(\frac{\beta n}{k}-1)}{2}\frac{k}{\beta}$, and thus Theorem 3 gives us $X_{low} = \frac{\sqrt{\beta}n}{k}$ and $Y_{low} = \frac{W}{k}$. For N*aive-Dedup, since the number of blocks is less than the number of reducers, one block is assigned to one reducer, leading to $X = \frac{\beta n}{k}$ and $Y = \frac{\frac{\beta n}{k}(\frac{\beta n}{k}-1)}{2} = \frac{\beta W}{k}$, both of which are not bounded. For* P*J-Dedup Y is optimal, but X is a factor $\sqrt{\frac{2k}{\beta}}$ away from the lower bound.*

The comparison is summarized in Table 1. For h_1, N*aive-Dedup matches the lower bounds for both X and Y; for h_2,* P*J-Dedup matches the lower bounds; and for h_3, neither matches the lower bounds.*

	h_1		h_2		h_3	
	X	Y	X	Y	X	Y
Lower bounds	$\frac{n}{k}$	$\frac{W}{k}$	$\frac{n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{\beta}n}{k}$	$\frac{W}{k}$
Naive-Dedup	$\frac{n}{k}$	$\frac{W}{k}$	n	W	$\frac{\beta n}{k}$	$\frac{\beta W}{k}$
PJ-Dedup	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$	$\frac{\sqrt{2}n}{\sqrt{k}}$	$\frac{W}{k}$

Table 1: Three example blocking functions

Example 3 demonstrates that (1) when the block sizes are small and uniform, such as the ones produced by h_1, we should use one reducer to handle each block, as in Naive-Dedup; (2) when there are dominating blocks, such as the ones produced by h_2, we should use multiple reducers to divide the workload, as in PJ-Dedup; and (3) when there are multiple relatively large blocks, we should use multiple reducers to handle every large block to avoid unbalanced computation. However, using all k reducers for every large block sends more tuples than necessary, since tuples from different blocks might be sent to same reducer, even though they will not be compared, as in PJ-Dedup.

4.3 The Proposed Strategy

Dis-Dedup adopts a distribution strategy which guarantees that both X and Y are always within a constant factor from X_{low} and Y_{low}, by assigning reducers to blocks in proportion to the workload of every block.

Intuitively, since we want to balance computation, a block of a larger size needs more reducers than a block of a smaller size. Since the blocks are independent, we allocate the reducers to blocks in proportion to their workload, namely, block B_i will be assigned to $k_i = \frac{W_i}{W}k$ reducers. However, k_i might not be an integer, and it is meaningless to allocate a fraction of reducers. Thus, k_i needs to be rounded to an integer. If $k_i > 1$, we can assign $\lfloor k_i \rfloor \geq 1$ reducers to B_i. On the other hand, if $k_i \leq 1$, which means $\lfloor k_i \rfloor = 0$, we must still assign at least one reducer to B_i. The total number of reducers after rounding might be greater than k, in which case reducers have to be responsible for more than one block. Therefore, we need an effective way of assigning reducers to blocks, such that both X and Y are minimized.

If $k_i \leq 1$, we call B_i a *single-reducer block*; otherwise, B_i is a *multi-reducer block*. Let $\mathcal{B}_s$ and $\mathcal{B}_l$ be the set of single-reducer blocks and multi-reducer blocks respectively. Next, we show how to handle single-reducer blocks and multi-reducer blocks separately, such that X and Y are bounded by a constant factor.

$$\mathcal{B}_s = \{B_i \mid W_i \leq \frac{W}{k}\}, \quad \mathcal{B}_l = \{B_i \mid W_i > \frac{W}{k}\}$$

For the sake of convenience, assume that we have ordered the blocks in increasing order of their workload: $W_1 \leq W_2 \leq \ldots \leq W_c \leq \frac{W}{k} < W_{c+1} \leq \ldots \leq W_m$. Let $W_l = \sum_{i=c+1}^{m} B_i$ be the total amount of workload for multi-reducer blocks, and $W_s = \sum_{i=1}^{c} B_i$ be the total amount of workload for single-reducer blocks. Also, let X_s (resp. X_l) be the maximum number of tuples from single-reducer blocks (resp. multi-reducer blocks) received by any reducer; and let Y_s (resp. Y_l) be the maximum number of comparisons from single-reducer blocks (resp. multi-reducer blocks) performed by any reducer. Therefore, $X \leq X_s + X_l$ and $Y \leq Y_s + Y_l$.

4.3.1 Handling multi-reducer blocks

Every block $B_i \in \mathcal{B}_l$ has $k_i \geq 1$ reducers assigned to it, and we will use k_i reducers to distribute B_i via the triangle distribution strategy in Section 3. If k_i is fractional, such as $k_i = 3.1$, we will simply use $\lfloor k_i \rfloor$ reducers to handle B_i. Since $\sum_{i=c+1}^{m} k_i \leq k$, every reducer will exclusively handle at most one multi-reducer block.

EXAMPLE 4. *Recall the blocking function h_3 in Example 3: every block is large, since $W_i > \frac{W}{k}$. Instead of using all k reducers to handle every block, we now use $k_i = \frac{W_i}{W}k = \beta$ reducers to handle block B_i. Thus, we have $X = X_l = \frac{\sqrt{2}}{\sqrt{k_i}}|B_i| = \frac{\sqrt{2\beta}}{k}n$, and $Y = Y_l = \frac{W_i}{k_i} = \frac{W}{k}$. Compared to the lower bound, we see that Y is optimal, and X is only $\sqrt{2}$ away from optimal.*

In fact, we can be even more aggressive in assigning reducers to big blocks, by assigning $k_i = \frac{W_i}{W_l}$ (instead of $k_i = \frac{W_i}{W}$) reducers to B_i. This still guarantees that there is at least one reducer for every multi-reducer block, and one reducer handles at most one multi-reducer block, since $\sum_{i=c+1}^{m} k_i = k$. By handling multi-reducer blocks this way, Dis-Dedup achieves the following bounds for X_l and Y_l:

THEOREM 4. D*is-Dedup has with high probability $Y_l \leq (1 + o(1))2Y_{low}$ and $X_l \leq (1 + o(1))2X_{low}$*

We present the detailed proof in the full paper [9].

4.3.2 Handling single-reducer blocks

For every block $B_i \in \mathcal{B}_s$, since we assign $k_i \leq 1$ reducers to it, we can use one reducer to handle every single-reducer block, just like Naive-Dedup does. However, we must assign single-reducer blocks to reducers to ensure that every reducer has about the same amount of workload.

We first present a *deterministic distribution* strategy for single-reducer blocks, which achieves a constant bound for X_s and Y_s. However, the deterministic strategy requires the mappers to keep the ordering of the sizes of single-reducer blocks in memory, which is very costly. We next consider a *randomized distribution*, which is cheaper to implement, but whose bound for X is dependent on k. Finally, we introduce a *hybrid distribution* strategy that uses the randomized distribution for most of the single-reducer blocks, and the deterministic distribution for only a small subset of the single-reducer blocks. The hybrid distribution requires a small memory footprint, and in the same time achieves constant bounds for both X_s and Y_s.

Deterministic Distribution. In order to allocate the single-reducer blocks evenly, we first order the c single-reducer blocks according to their block sizes, and divide them into $g = \frac{c}{k}$ groups[4], where each group consists of consecutive k blocks in the ordering. Then, we assign to each reducer g blocks, one from each group.

THEOREM 5. *The deterministic distribution strategy for single-reducer blocks achieves $Y_s \leq 2Y_{low}$ and $X_s \leq 2X_{low}$.*

We present the detailed proof in the full paper [9].

The problem with implementing the deterministic distribution is that there can be a large number of single-reducer blocks, and keeping track of the ordering of all block sizes is an expensive task within each mapper, not to mention the need to actually order all the single-reducer blocks.

Randomized Distribution. This algorithm simply distributes each single-reducer block to a reducer by using a random hash function. In order to analyze the randomized distribution, it suffices to consider the worst-case scenario, which is when we have k single-reducer blocks, each with $|B_i| = n/k$. In this case, the problem becomes a *balls-into-bins* scenario, where we have k balls (blocks) that we distribute independently and uniformly at random into k bins (reducers). It then holds [23] that with high probability each reducer will receive a maximum of $O(\ln(k))$ blocks. Thus:

THEOREM 6. *The randomized distribution strategy for single-reducer blocks achieves $Y_s \leq \ln(k)Y_{low}$ and $X_s \leq \ln(k)X_{low}$.*

We present the detailed proof in the full paper [9].

The randomized method is efficient in practice, since we do not have to keep track of the single-reducer blocks, but we are adding (in the worst case) an additional $\ln(k)$ factor.

Hybrid Distribution. The hybrid algorithm combines the randomized and deterministic distribution to achieve efficiency and almost optimal distribution. To start, we set a threshold $\tau = \frac{W}{3k\ln(k)}$.[5] For the blocks where $W_i \geq \tau$ (but still $W_i \leq \frac{W}{k}$), we use the deterministic distribution, while for the blocks where $W_i < \tau$ we use the randomized distribution. Observe that now the deterministic option is much cheaper, since we have to keep track of at most $3k\ln(k)$ blocks (which number depends only on the number of reducers, and not n).

THEOREM 7. *The hybrid distribution strategy for single-reducer blocks has with high probability $Y_s \leq (1 + o(1))3Y_{low}$ and $X_s \leq (1 + o(1))3X_{low}$.*

We present the detailed proof in the full paper [9].

4.3.3 Implementation

Dis-Dedup combines the triangle distribution strategy for multi-reducer blocks and the hybrid distribution for single-reducer blocks. However, given a new tuple ingested by a mapper, how does the mapper know whether the tuple belongs to a multi-reducer block or a single-reducer block? In order to make this decision, we need some preprocessing to collect statistics to be loaded into each mapper. In particular, we need to compute the blocking key values of every multi-reducer block $B_i \in \mathcal{B}_l$, and the associated workload W_i. Let HM_1 denote the HashMap data structure that stores the mapping from a blocking key value in the multi-reducer blocks to the size of the block. In addition, we need the blocking key values of those single-reducer blocks $B_i \in \mathcal{B}_l$ that have $W_i \geq \tau$, and the associated workload W_i, in order to implement the hybrid distribution strategy for the single-reducer blocks. Let HM_2 denote the HashMap data structure that stores the mapping from a blocking key value to the size of the block.

Algorithm 2 Dis-Dedup

```
1: class MAPPER
2:     $BKV2RIDs \leftarrow$ empty dictionary
3:     method MAPPERSETUP($HM_1$, $HM_2$)
4:         $W_l \leftarrow \sum_{bkv \in HM_1.keySet()} HM_1[bkv]$
5:         $S \leftarrow \{1, 2, \ldots, k\}$
6:         for all $bkv \in HM_1.keySet()$ do
7:             $k_i \leftarrow \lfloor \frac{HM_1[bkv]}{W_l} k \rfloor$
8:             $RIDS_i \leftarrow$ select $k_i$ elements from $S$
9:             $BKV2RIDs[bkv] \leftarrow RIDS_i$
10:            $S \leftarrow S - RIDS_i$
11:        $SortedBKVs \leftarrow$ sort all keys in $HM_2$
12:        $RID \leftarrow 0$
13:        for all $bkv \in SortedBKVs$ do
14:            $BKV2RIDs[bkv] \leftarrow RID\%k$
15:            $RID \leftarrow RID + 1$
16:    method MAP(Tuple $t$, null)
17:        $bkv \leftarrow h(t)$
18:        if $BKV2RIDs[bkv] \neq \emptyset$ then
19:            $rid \leftarrow$ a random number from $[1, k]$
20:            EMIT(Key $rid\#bkv$, $S\#$Tuple $t$)
21:        else
22:            $RIDs \leftarrow BKV2RIDs[bkv]$
23:            $k_i \leftarrow RIDs.size()$
24:            $l_i \leftarrow$ the large integer s.t. $l_i(l_i - 1)/2 < k_i$
25:            Int $a \leftarrow$ a random value from $[1, l_i]$
26:            for all $p \in [1, a)$ do
27:                $ridIndex \leftarrow$ rid of Reducer $[p, a]$
28:                $rid \leftarrow RIDs[ridIndex]$
29:                EMIT(Key $rid\#bkv$, $L\#$Tuple $t$)
30:            $ridIndex \leftarrow$ rid of Reducer $[a, a]$
31:            $rid \leftarrow RIDs[ridIndex]$
32:            EMIT(Key $rid\#bkv$, $S\#$Tuple $t$)
33:            for all $q \in (a, l]$ do
34:                $ridIndex \leftarrow$ rid of Reducer $[a, q]$
35:                $rid \leftarrow RIDs[ridIndex]$
36:                EMIT(Key $rid\#bkv$, $R\#$Tuple $t$)
37: class PARTITIONER
38:    method PARTITION(Key $rid\#bkv$, Value $v$, $k$)
39:        RETURN $rid$
40: class REDUCER
41:    method REDUCE(Key $rid\#bkv$, Values $[v_1, v_2 \ldots]$)
42:        same as the reduce function in Algorithm 1
```

To compute HM_1 and HM_2, we use three simple word-count alike MapReduce jobs. The first job takes the original dataset as input, and counts the size of every block B_i. The second job takes as input the result of the first job, and counts the total workload W. The third job takes as input the result of the first job and W, and outputs the blocks for which $W_i > \frac{W}{k}$, namely HM_1, and also the blocks where $\frac{W}{3k\ln(k)} < W_i \leq \frac{W}{k}$, namely HM_2.

Algorithm 2 describes in detail Dis-Dedup. The mapper now has a setup method that needs to be executed, which allocates reducers to blocks in HM_1 and HM_2 (Lines 3-15).

[4] We assume that $\frac{c}{k}$ is an integer; otherwise, we can conceptually add less than k empty blocks to $\mathcal{B}_s$ to make $\frac{c}{k}$ an integer, and the analysis remains intact.

[5] The threshold value is chosen as a result of the connection to the weighted balls-into-bins problem. When we throw k balls of weight W/k into k bins uniformly at random, the expected maximum number of balls is $O(\ln(k))$, and so the maximum weight is $O(W\ln(k)/k)$. If we instead throw $k\ln(k)$ balls of weight $W/k\ln(k)$, the expected maximum number remains $O(\ln(k))$, but since the balls are of smaller weight the maximum weight decreases to $O(W/k)$.

To allocate reducers to blocks in HM_1, we first compute the sum of workload of all the multi-reducer blocks, i.e., W_l (Line 4). For every blocking key value in HM_1, we calculate the number of reducers k_i allocated to it, and select k_i reducers from the set of k reducers (Lines 5-10). For every blocking key value in HM_2, we first sort them based on their workload (Line 11), and allocate one reducer to the sorted blocks in a round-robin manner (Lines 12-15). In the actual mapping function, given a new tuple t and its blocking key value *bkv*, we check if we have a fixed set of reducers allocated to it. If there is no such fixed allocation, *bkv* must the block that is randomly distributed, and thus we randomly choose a reducer to send the tuple (Lines 18-20). If there is a fixed set of reducer *RIDs* allocated to it, we distribute t according to PJ-Dedup, by arranging reducer *RIDs* in a triangle (Lines 22-36). The partitioner sends a key-value pair according to the *rid* part in the key (Lines 37-39). The reducer is the same as that of Algorithm 1 (Lines 40-42).

Theorem 8. D*is-Dedup with hybrid distribution for single-reducer blocks achieves with high probability* $X \leq c_x X_{low}$, *and* $Y \leq c_y Y_{low}$, *where* $c_x = 5 + o(1)$ *and* $c_y = 5 + o(1)$.

Theorem 8 is obtained by combining Theorems 4 and 7.

5. DEDUPLICATION USING MULTIPLE BLOCKING FUNCTIONS

Since a single blocking function might result in false negatives by failing to assign duplicate tuples to the same block, multiple blocking functions are often used to decrease the likelihood of a false negative. In this section, we study how to distribute the blocks produced by s different blocking functions $h_1, h_2, \ldots, h_s$.

A straightforward strategy to handle s blocking functions would be to apply Dis-Dedup s times (possibly simultaneously). However, this straightforward strategy has two problems: (1) it fails to leverage the independence of the blocking functions, and the tuples from blocks generated by different blocking functions might be sent to same reducer, where they will not be compared. This is similar to PJ-Dedup's failure to leverage the independence of the set of blocks generated by a single blocking function, as shown in Example 3; and (2) a tuple pair might be compared multiple times, if that tuple pair belongs to multiple blocks, each from a different blocking function. We address these two problems by two principles: (1) allocating reducers to blocking functions proportional to their workload; and (2) imposing an ordering of the blocking functions.

Reducer Allocation. We allocate one or more reducers to a blocking function in proportion to the workload of that blocking function, similar to Dis-Dedup discussed in Section 4. Let m_j denote the number of blocks generated by a blocking function h_j, B_i^j denote the i-th block generated by h_j, and $W_i^j = \binom{|B_i^j|}{2}$ denote the workload of B_i^j. Let $W^j = \sum_{i=1}^{m_j} W_i^j$ be the total amount of workload generated by h_j, and $W = \sum_{j=1}^{t} W^j$ be the total workload generated by all t blocking functions. Therefore, the number of reducers B_i^j gets assigned is $\frac{W_i^j}{W^j} \cdot \frac{W^j}{W} k$, where $\frac{W^j}{W} k$ is the number of reducers for handling the blocking function h_j. Thus, the number of reducers assigned to B_i^j is $\frac{W_i^j}{W}$, regardless of which blocking function it originates from.

This means that we can view the blocks from multiple blocking functions as a set of (possibly overlapping) blocks produced by one blocking function, and apply Dis-Dedup as-is. The only modification needed is that in the mapper of Dis-Dedup, instead of applying one hash function to obtain one blocking key value, we apply s hash functions to obtain s blocking key values (we also need to apply the body of the mapping function for every blocking key value). We call the slightly modified version of Dis-Dedup to handle multiple blocking functions Dis-Dedup$^+$.

Theorem 9. *For* t *blocking functions,* D*is-Dedup*$^+$ *achieves* $X < (5s + o(1))X_{low}$, *and* $Y = (5 + o(1))Y_{low}$.

Proof. The lower bound for X and Y for s blocking functions is the same lower bound for single blocking function, as stated in Theorem 8, except W now denotes the total number of comparisons for all s blocking functions. The proof of Theorem 9 follows the same line as the proof of Theorem 8. The only difference is when we are dealing with $\mathcal{B}_s$: instead of having the inequality $\sum_{i=1}^{c} |B_i| < n$ for a single blocking function, we now have $\sum_{i=1}^{c} |B_i| < sn$, which leads to an additional factor s in the bound for X. □

The above analysis tells us that the number of comparisons will be optimal, but the input may have to be replicated as many as s times. The reason for this increase is that the blocks may overlap, in which case tuples that belong in multiple blocks may be replicated. The results from Theorem 9 can be carried to a single blocking function that produces a set of overlapping blocks, where $s = \frac{\sum_{i=1}^{m} |B_i|}{n}$.

Blocking Function Ordering. Since a tuple pair can have the same blocking key values according to multiple blocking functions, a tuple pair can occur in multiple blocks. To avoid producing the same tuple pair more than once, we impose an ordering of the blocking functions, from h_1 to h_s. Every reducer has knowledge of all s blocking functions, and their fixed ordering. Inside every reducer, before a tuple pair t_1, t_2 is compared according to the j^{th} blocking function h_j, it applies all of the the lower-numbered blocking function $h_z, \forall z \in [1, j-1]$, to see if $h_z(t_1) = h_z(t_2)$. If such h_z exists, then the tuple pair comparison according to h_j is skipped in that reducer, since we are sure that there must exist a reducer that can see the same tuple pair according to h_z. In this way, every tuple pair is only compared according to the *lowest numbered blocking function* that puts them in the same block. The blocking function ordering technique assumes that applying blocking functions is much cheaper than applying the comparison function. If this is not true (e.g., the pairs comparison is cheap) the ordering benefit will not be obvious. Based on our discussion with Thomson Reuters and Tamr, which are engaged in large-scale deduplication projects, the assumption of expensive comparison is often true in practice, since it is usually performed by applying multiple classifiers and machine learning models.

Example 5. *Suppose two tuples* t_1 *and* t_2 *are in the same block according to the blocking functions* h_2, h_3, h_5*, namely,* $h_2(t_1) = h_2(t_2), h_3(t_1) = h_3(t_2)$ *and* $h_5(t_1) = h_5(t_2)$*. In this case, we only compare* t_1 *and* t_2 *in the block generated by* h_2*, and omit the comparison of those two tuples in the blocks generated by* h_3 *and* h_5*.*

To recognize which blocking function generated a tuple pair comparison, we augment the key of the mapper from *rid#bkv* to *rid#bkv#hIndex*, where *hIndex* is the blocking function number that generated this *bkv*. In the reduce function, before a tuple pair is compared, we check if there is another blocking function whose index is smaller than the *hIndex* that puts that tuple pair in the same block. If such a blocking function exists, we skip the comparison.

6. EXPERIMENTAL STUDY

In this section, we evaluate the effectiveness of our distribution strategies. All experiments are performed on a cluster of eight machines, running Hadoop 2.6.0 [1]. Each machine has 12 Intel Xeon CPUs at 1.6 GHz, 16MB cache size, 64GB memory, and 4TB hard disk. All machines are connected to the same gigabit switch. One machine is dedicated to serve as the resource manager for YARN and as the HDFS namenode, while the other seven are slave nodes. The HDFS block size is the default 64MB, and all machines serve as data nodes for the DFS. Every node is configured to be able to run 7 reduce tasks at the same time, and thus in total, 49 reduce tasks can run in parallel. Our cluster resembles the cluster that is used in production by Thomson Reuters to perform data deduplication. We also use a real dataset from that company, as described next.

Datasets. We design a *synthetic data* generator, which takes as input the number of tuples n, the number of blocks m, and a parameter θ controlling the distribution of block sizes, following the zipfian distribution. Every generated tuple has two columns $\{A, B\}$. Column A is the blocking key attribute, whose values range from 1 to m. The blocking function h for a tuple t in the synthetic dataset is $h(t) = t[A]$. Thus, tuples with the same value for A belong to the same block. Column B is a randomly generated string of 1000 characters. The comparison between two tuples is the edit distance between the two values for B.

We use two real datasets. The first one is a list of publication records from CITESEERX [6], called CSX. The second one, called OA, is a private dataset from Thomson Reuters, containing a list of organization names and their associated information, such as addresses and phone numbers, extracted from news articles. CSX uses the publication title as the blocking key attribute, while OA uses the organization name. The blocking function for both datasets uses minHash [18]. A minHash signature is generated for each tuple, and tuples with the same signature belong in the same block. For each tuple in CSX or OA, the blocking function extracts 3-grams from the blocking key attribute, and applies a random hash function to the set of 3-grams; the minimum hash value for the set of 3-grams is used as the minHash signature for that tuple. We can generate multiple blocking functions by using different random hash functions. The comparison between two tuples is the edit distance between the two values of the blocking key attributes.

Dataset	# Tuples	AVG block size	MAX block size
CSX	1.4M	1.75	35021
OA	7.4M	1.31	7969
SYNTHETIC	20M	4	3967

Table 2: Datasets Statistics

Table 2 summarizes the statistics for each dataset we used for performing data deduplication using a single blocking function in Section 6.2. We will use up to 20 blocking functions for performing data deduplication using multiple blocking functions in Section 6.3.

Algorithms. Dedoop [19, 20] is a state-of-the-art distribution strategy for data deduplication, which has two main drawbacks: (1) it only aims at optimizing Y, with no consideration for X; and (2) it has large memory requirements for mappers and reducers. Dedoop starts by assigning an index to every tuple pair that needs to be compared. For example, for a block with four tuples t_1, t_2, t_3, t_4, tuple pair $\langle t_1, t_2 \rangle$ has index 1, tuple pair $\langle t_1, t_3 \rangle$ has index 2, and so on, and tuple pair $\langle t_3, t_4 \rangle$ has index 6. Dedoop then assigns each reducer an equal number of tuple pairs to compare. If W denotes the total number of tuple pairs, the i^{th} reducer will take care of tuple pairs whose indexes are in $[(i-1)\frac{W}{k}, i\frac{W}{k}]$. All the tuples necessary for a reducer to compare the tuple pairs assigned to it are sent to that reducer. Dedoop always achieves the optimal Y, but could be arbitrarily bad for X. For example, consider a single block of size n. The first reducer is responsible for tuple pairs numbered $[0, \frac{n(n-1)}{2k}]$. Since usually $n \gg k$, the first reducer is responsible for at least n tuple pairs. Since the first n tuple pairs contain all n tuples, the first reducer will have to receive all n tuples. In addition, in order for each mapper to decide which tuples to send to which reducer, and for each reducer to decide which tuple pairs to compare, Dedoop loads into the memory of each mapper and reducer a pre-computed data structure, called *Block Distribution Matrix*, which specifies the number of entities for every block every mapper processes. The size of the block distribution matrix is linear w.r.t. the number of blocks. Indeed, the largest number of blocks reported in evaluating Dedoop is less than 15,000 [20].

We compare the following distribution strategies: (1) Naive-Dedup is the naïve distribution strategy, where every block is assigned to one reducer; (2) PJ-Dedup is the proposed strategy where every block is distributed using the triangle distribution strategy, which is strictly better than applying existing parallel join algorithms [2, 22]; (3) Dis-Dedup is the proposed distribution strategy for a single blocking function, which is theoretically optimal; (4) Dedoop; and (5) Dis-Dedup$^+$, Naive-Dedup$^+$, PJ-Dedup$^+$, Dedoop$^+$ are extensions of Dis-Dedup, Naive-Dedup, PJ-Dedup, and Dedoop$^+$, respectively for multiple blocking functions. Dis-Dedup$^+$ is described in Section 5. Naive-Dedup$^+$, PJ-Dedup$^+$, and Dedoop$^+$ employ the same blocking function ordering technique to avoid comparing a tuple pair multiple times.

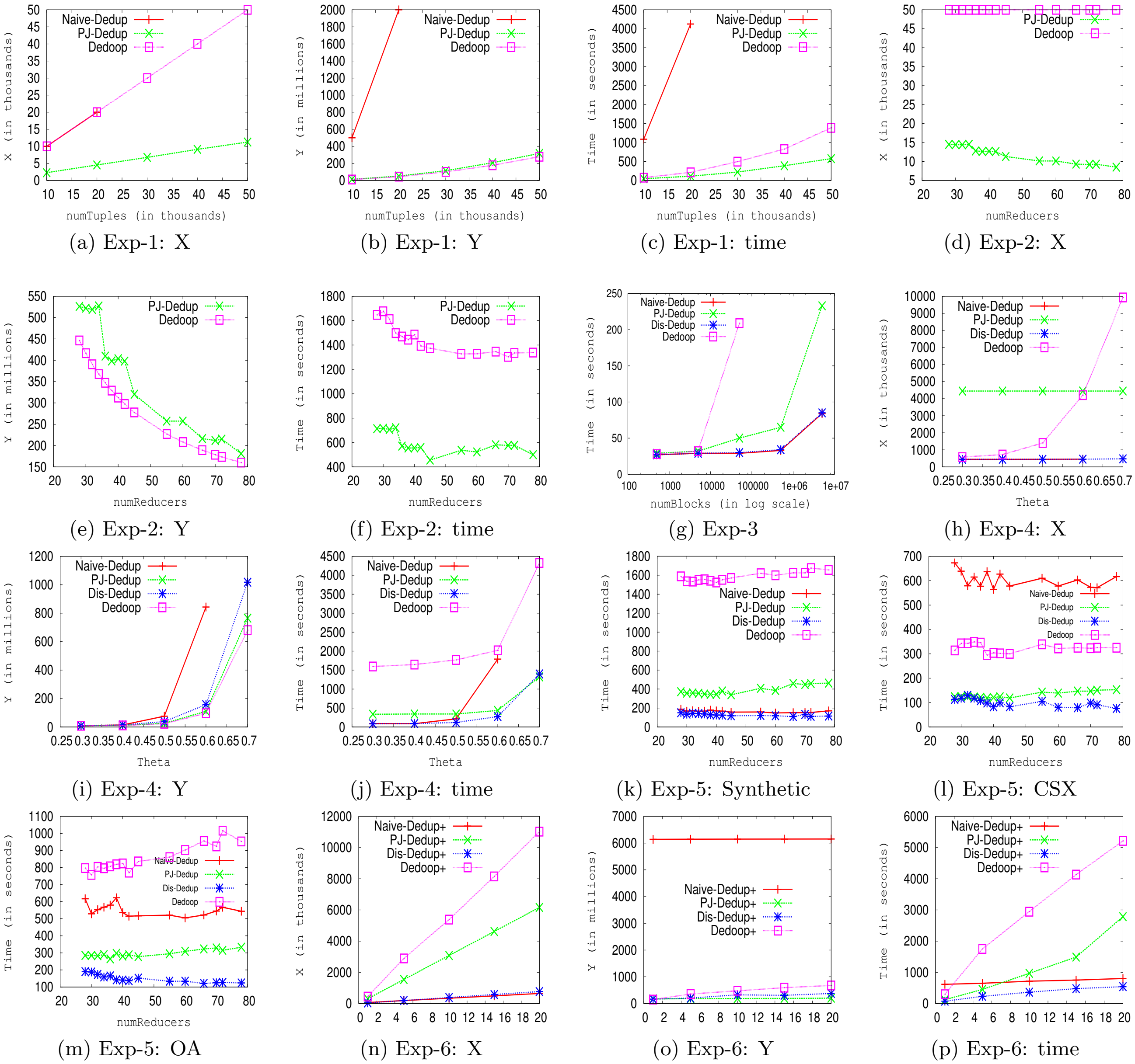
Since the distribution strategies are random, we run each experiment three times, and report the average. The time to compute the statistics for Dis-Dedup and DEDOOP is included in the time of Dis-Dedup and DEDOOP, respectively.

6.1 Single Block Deduplication Evaluation

In this section, we compare Naive-Dedup, PJ-Dedup, and DEDOOP when performing a self-join on the synthetic dataset. We omit Dis-Dedup, since it is identical to PJ-Dedup when there is only one block.

Exp-1: Varying number of tuples. Figure 3(a-c) shows the parameters X, Y, and *time* for Naive-Dedup, PJ-Dedup, and DEDOOP for $k = 45$ reducers. We terminate a job after 6000 seconds. As we can see in Figure 3(c), Naive-Dedup exceeds this time limit after $30K$ tuples, since the computation occurs only in a single reducer. In terms of X, Figure 3(a) shows that PJ-Dedup achieves the best behavior, while for Naive-Dedup and DEDOOP X is equal to the number of tuples n. Indeed, Naive-Dedup uses only one reducer, and DEDOOP has one reducer that gets assigned the first $\frac{n(n-1)}{2k}$ tuple pairs, which need to access all n tuples. In terms of Y, as depicted in Figure 3(b), Naive-Dedup performs the worst. DEDOOP is slightly better than PJ-Dedup, since it distributes the comparisons evenly amongst all reducers, while PJ-Dedup has more work for reducers indexed (p, q) than reducers indexed (p, p).

[6] http://csxstatic.ist.psu.edu/about



(a) Exp-1: X

(b) Exp-1: Y

(c) Exp-1: time

(d) Exp-2: X

(e) Exp-2: Y

(f) Exp-2: time

(g) Exp-3

(h) Exp-4: X

(i) Exp-4: Y

(j) Exp-4: time

(k) Exp-5: Synthetic

(l) Exp-5: CSX

(m) Exp-5: OA

(n) Exp-6: X

(o) Exp-6: Y

(p) Exp-6: time

Figure 3: Evaluating Distribution Strategies

The running time of all three algorithms is depicted in Figure 3(c), and it shows that PJ-Dedup achieves the best performance. In fact, comparing Figure 3(a) with Figure 3(c), we can see that as the number of tuples increases, the gap between PJ-Dedup and DEDOOP in terms of the input size X grows, and so does the running time. This indicates that when Y is similar for PJ-Dedup and DEDOOP, the input size X becomes the differentiating factor.

Exp-2: Varying number of reducers. Figure 3(d-f) shows the effect of the number of reducers on the parameters X, Y, and *time* for both PJ-Dedup and DEDOOP. We fix the number of tuples to be $n = 50K$. The first observation is that PJ-Dedup outperforms DEDOOP for any k, as shown in Figure 3(f). In fact, PJ-Dedup runs 2X faster than DEDOOP. Second, both X and Y are decreasing for PJ-Dedup as k increases. However, the decrease is not smooth across the values of k and we can observe a few dips for $k = 36, 45, 55, 66, 78$. This behavior occurs because PJ-Dedup arranges the k reducers in a triangle. Since for certain values of k this is not possible, we choose then the largest subset of reducers that can be arranged into a triangle, and thus waste a small fraction of reducers.

Finally, as we can see from Figure 3(f), the running time improves as k increases for $k < 50$, and fluctuates as k increases when $k > 50$. This is because our cluster has a maximum number of 49 reducers being able to run in parallel. If $k > 50$, not all reducers can start at once; some reducers can only start after reducers in the first round finish.

Memory. An important requirement for any MapReduce job is to have sufficient memory for each reducer. The maximum memory requirements for all algorithms are linear with respect to X, since X represents the maximum number of tuples that need to be stored in the heap space of a reducer. As we can see from Figure 3(a), PJ-Dedup requires much less memory than Dedoop. Indeed, for Dedoop at least one reducer needs to store all n tuples, while the heap space required for PJ-Dedup is $O(\frac{n}{\sqrt{k}})$, which not only is much smaller than $O(n)$, but also decreases as k increases (this can be seen in Figure 3(d)).

6.2 Single Blocking Function Evaluation

In this section, we compare Naive-Dedup, PJ-Dedup, Dis-Dedup, and Dedoop for deduplication using a single blocking function on all three datasets.

Exp-3: Varying number of blocks. In this experiment, we fix the number of tuples per block to be 4, and then vary the number of blocks, using the synthetic dataset. As shown in Figure 3(g), Dedoop fails (heap space error) after 500,000 blocks. This is because Dedoop keeps in memory of every mapper the blocking distribution matrix, which grows as the number of blocks increases.

Dis-Dedup is identical to Naive-Dedup in this experiment, since all block sizes are the same, and hence no multi-reducer blocks exist. Dis-Dedup is better than PJ-Dedup, since the input size X for Dis-Dedup, which is $\frac{n}{k}$, is smaller than that for PJ-Dedup, which is $\frac{\sqrt{2}n}{\sqrt{k}}$.

Exp-4: Varying block size distribution. In order to test how different algorithms handle block-size skew, we vary the distribution of the block sizes by varying the parameter θ for the synthetic dataset, using $n = 20M$ tuples, and $5M$ blocks. Figure 3(h) shows that Dis-Dedup and Naive-Dedup send less tuples than PJ-Dedup, and Figure 3(i) shows that the number of comparisons Y increases as the data becomes more skewed across all three algorithms. However, Y for PJ-Dedup grows at the lowest rate, and Y for Dis-Dedup is just a little worse than PJ-Dedup. In terms of running time, Figure 3(j) shows that when the data is not skewed ($\theta = 0.3, 0.4$), Naive-Dedup and Dis-Dedup perform the best, since X is now the differentiating factor. When the data becomes more skewed ($\theta = 0.5$), Dis-Dedup starts performing better than Naive-Dedup, which is still better than PJ-Dedup. As θ further increases ($\theta = 0.6, 0.7$), Y becomes the dominating factor in terms of running time. Thus, the running time for Naive-Dedup degrades fast, while Dis-Dedup and PJ-Dedup have similar performance. This observation supports our theoretical analysis that Dis-Dedup can adapt to all levels of skew, while Naive-Dedup and PJ-Dedup perform well only at one end of the spectrum. Note that Dedoop reports heap space error for this synthetic under default memory settings; nevertheless, we increase memory allocation for every mapper and reducer to 6G for Dedoop to compare with other distribution strategies. Observe that Dedoop has the worst running time in Figure 3(j) even though the X and Y of Dedoop are not the worst; this is mainly because (1) the number of concurrent mappers and reducers of Dedoop is less than that under the default setting due to the increased memory requirement of each mapper and reducer and (2) each mapper and reducer of Dedoop has an additional initializing cost of processing the block distribution matrix, whose size is linear w.r.t. the number of blocks.

Exp-5: Varying number of reducers. In this experiment, we vary the number of reducers k, and compare Naive-Dedup, PJ-Dedup, Dis-Dedup and Dedoop using all three datasets. For the synthetic dataset, we used $n = 20M$, $m = 5M$, and $\theta = 0.5$. Similar to the previous experiment, for Dedoop to run, we increase the memory allocation for every mapper and reducer to 6G for the OA and the synthetic datasets, and to 3G for CSX.

Dataset	1	5	10	15	20
CSX	667.45	710.64	741.52	768.51	784.40
OA	66.16	67.24	67.92	68.00	68.06

Table 3: Total number of comparisons W (in millions), for different number of blocking functions

Figure 3(k-m) shows that Dis-Dedup is consistently the best algorithm for all three datasets, and any number of reducers. Dedoop only performs better than Naive-Dedup on CSX due to its small number of blocks, thus making the initializing cost of processing the block distribution matrix relatively cheap; but Dedoop has the worst performance in other two datasets for the same reasons as explained in Exp-4. For the synthetic dataset, Naive-Dedup performs better than PJ-Dedup, while for the two real datasets the opposite behavior occurs, since the number of multi-reducer blocks in CSX and OA is bigger than that of the synthetic dataset. Another interesting trend to note here is that as the number of reducers k increases, the difference in running time between Dis-Dedup and PJ-Dedup also grows. The reason for this behavior is that X for PJ-Dedup is a $\sqrt{2k}$ factor away from the bound X_{low}, and thus dependent on k, while X for Dis-Dedup is only a constant factor 2 away.

6.3 Multiple Blocking Functions Evaluation

In this section, we compare the algorithms Naive-Dedup$^+$, PJ-Dedup$^+$, Dis-Dedup$^+$ and Dedoop$^+$, in the case of multiple blocking functions, using the two real datasets.

Exp-6: Varying the number of blocking functions. Figure 3(n-p) shows the comparison using CSX. The input size X of all four algorithms increases linearly w.r.t. the number of blocking functions, as shown in Figure 3(n), while the output Y of all three algorithms increases very little, as shown in Figure 3(o). This behavior is observed because many tuple pairs generated by a blocking function have already been compared in previous blocking functions, and are thus skipped. Table 3 shows the total number of comparisons W for various numbers of blocking functions, indicating that the new tuple pair comparisons generated by 20 blocking functions is not much larger than the comparisons generated by 1 blocking function.

Figure 3(p) shows the running time comparison. Dedoop$^+$ performs the worst for multiple blocking functions due to multiple reasons: (1) Dedoop$^+$ has the worst X as shown in Figure 3(n); (2) Dedoop$^+$ has higher memory requirement, which limits the number of concurrent mappers and reducers, as explained in Exp-4; and (3) Dedoop$^+$ pays the extra cost of initiating s (≥ 1) MapReduce job to handle s (≥ 1) blocking functions, instead of using one MapReduce job to handle s blocking functions as Naive-Dedup$^+$, PJ-Dedup$^+$, and Dis-Dedup$^+$ do. The reason is again the high memory requirement of Dedoop$^+$; keeping the block distribution matrix produced by all s blocking functions in memory exceeds the memory limit for $s \geq 5$ even after we increase the memory allocation of mappers and reducers to 6G.

Dis-Dedup$^+$ achieves the best performance across any number of blocking functions. As the the number of blocking functions increases, the gap between Naive-Dedup$^+$ and Dis-Dedup$^+$ becomes smaller, while the gap between PJ-Dedup$^+$ and Dis-Dedup$^+$ becomes larger. The reason is that the multi-reducer blocks for s_1 blocking functions may become single-reducer blocks for $s_2 > s_1$ blocking functions as W becomes larger. Therefore, distributing those blocks to one reducer, as Naive-Dedup$^+$ does, instead of distributing them to multiple reducers, as PJ-Dedup$^+$ does, becomes more efficient. Varying number of blocking functions using the OA dataset shows similar results, and is reported in the full paper [9].

7. RELATED WORK

Data deduplication has been extensively covered in many surveys [12, 13, 17]. To avoid n^2 comparisons, blocking methods are often employed [4, 6]. While blocking techniques speed up computation by avoiding the comparison between certain tuple pairs, performing deduplication using a parallel framework can obtain even further speed ups. In this context, Dedoop [19, 20] targets deduplication on MapReduce. As discussed in Section 6, Dedoop evenly distributes the comparisons Y at the cost of an increased input size X, and has a very large memory footprint, rendering it infeasible to run for large datasets.

Parallel join processing, such as similarity joins [24, 27], theta-joins [22], and multi-way natural joins [2], is another line of relevant work, since one can view the distributed blocking problem as a self-join, where the joining attribute is the blocking key attribute. However, most of the work on parallel join processing [22, 2] is for two-table joins, so the techniques are not directly applicable to self-join without wasting almost half of the available workers, as shown in Section 3. Our proposed triangle distribution strategy is a non-trivial adaptation of the existing *Shares* distribution strategy [2] for the $R \times S$ join with theoretical guarantees on both the communication cost X and the computation cost Y. Applying even the adapted self-join strategy on blocks directly without considering the block sizes yields a strategy without constant bound guarantee on X, as shown in Section 4.2. Our proposed Dis-Dedup, however, assigns workers in proportion to the workload of every block to achieve a constant-bound X and Y. Furthermore, dealing with multiple blocking functions basically means to process a disjunction of conjunctive queries, a problem, to the best of our knowledge, is never considered in parallel join processing.

8. CONCLUSION AND FUTURE WORK

In this paper, we study the problem of performing blocking for data deduplication using a shared-nothing distributed computing framework. We develop a cost model that captures both the communication cost X and computation cost Y. Under this cost model, we propose the Dis-Dedup algorithm, which guarantees both X and Y are within a small constant factor from the lower bounds. Through extensive experiments, we show that Dis-Dedup adapts to data skew, and significantly outperforms any alternative strategy we consider, including the state-of-the-art algorithm.

When there are multiple blocking functions, a large percentage of compared tuple pairs belong to the same block according to more than one blocking functions. While our proposed strategy avoids producing the same tuple pair more than once, a tuple still needs to be sent as many times as the number of blocking functions. Future work includes gathering statistics about the overlapping of blocks, so that we can merge largely overlapping blocks to further reduce the communication cost.

9. REFERENCES

[1] Apache hadoop. `http://hadoop.apache.org`.

[2] F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *EDBT*, pages 99–110, 2010.

[3] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[4] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pages 586–597, 2002.

[5] P. Beame, P. Koutris, and D. Suciu. Skew in parallel query processing. In R. Hull and M. Grohe, editors, *PODS*, pages 212–223. ACM, 2014.

[6] M. Bilenko, B. Kamath, and R. J. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, pages 87–96, 2006.

[7] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Trans. on Knowl. and Data Eng.*, 24(9):1537–1555, Sept. 2012.

[8] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *SIGMOD*, pages 63–78, 2015.

[9] X. Chu, I. F. Ilyas, and P. Koutris. Distributed Data Deduplication. Technical Report CS-2016-02, University of Waterloo, 2016.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[11] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, pages 27–40, 1992.

[12] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[13] L. Getoor and A. Machanavajjhala. Entity resolution: theory, practice & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

[14] D. M. Gordon, G. Kuperberg, and O. Patashnik. New constructions for covering designs. *J. COMBIN. DESIGNS*, 3(269–284), 1995.

[15] D. Halperin, V. Teixeira de Almeida, L. L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, et al. Demonstration of the myria big data management service. In *SIGMOD*, pages 881–884. ACM, 2014.

[16] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. *ACM SIGMOD Record*, 24(2):127–138, 1995.

[17] I. F. Ilyas and X. Chu. Trends in cleaning relational data: Consistency and deduplication. *Foundations and Trends in Databases*, 5(4):281–393, 2015.

[18] P. Indyk. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, 38(1):84–90, 2001.

[19] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *PVLDB*, 5(12):1878–1881, 2012.

[20] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *ICDE*, pages 618–629, 2012.

[21] H. Köpcke, A. Thor, and E. Rahm. Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493, 2010.

[22] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *SIGMOD*, pages 949–960. ACM, 2011.

[23] M. Raab and A. Steger. "balls into bins" - A simple and tight analysis. In *RANDOM*, pages 159–170, 1998.

[24] A. D. Sarma, Y. He, and S. Chaudhuri. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB*, 7(12):1059–1070, 2014.

[25] J. Schönheim. On coverings. *Pacific Journal of Mathematics*, 14:1405–1411, 1964.

[26] J. D. Ullman. Designing good mapreduce algorithms. *XRDS*, 19(1):30–34, Sept. 2012.

[27] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506. ACM, 2010.

[28] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *HotCloud*, volume 10, page 10, 2010.