Scalable Load Balancing for MapReduce-based Record Linkage

Wei Yan, Yuan Xue
Dept. of Electrical Engineering and Computer Science
Vanderbilt University
Nashville, TN, USA
Email: {wei.yan, yuan.xue}@vanderbilt.edu

Bradley Malin
Dept. of Biomedical Informatics
Vanderbilt University
Nashville, TN, USA
Email: b.malin@vanderbilt.edu

Abstract—Recent research has introduced load balancing schemes that are aware of the input data distribution (i.e., data profile) to mitigate data skew and fully exploit the parallel capability of the MapReduce framework to support record linkage. However, existing solutions face a significant scalability issue when applied to massive data sets with millions or billions of blocks (a basic unit in record linkage) because their data profiles can not be maintained precisely in an efficient manner. The goal of this paper is to introduce a profiling method based on the notion of a sketch, which allows for a compact scalable solution for maintaining block size statistics. In addition, we propose two load balancing algorithms to work over sketch-based profiles, and solve the data skew problem associated with record linkage. We provide an analytical analysis and extensive experiments (using Hadoop), with real and controlled synthetic data sets, to illustrate effectiveness of our solution.

Keywords—Record Linkage; MapReduce; Scalability; Load Balance

I. INTRODUCTION

The integration of data from multiple sources (i.e., record linkage) [7]), where duplicates are merged or removed, is critical to ensure big data repositories are managed efficiently and effectively. Traditional sequential architectures for record linkage hit significant performance barriers when applied to big data, leading to extremely long running time and high resource consumption [13]. Recently, several techniques [12], [18], [21] have been proposed to parallelize the record linkage process and reduce its processing time to within an acceptable limit based on the MapReduce platform [6]. In these techniques, the data sets are partitioned into several blocks using blocking keys by the map tasks and assigned to parallel reduce tasks, where the record pairs are constructed for comparison.

A common issue faced by these approaches is reduce-phase data skew, which occurs when the block workload is non-uniformly distributed. Figure 1(a) illustrates an example of block workload ranking for DBLP-1 data set (Table I for details). When this data skew occurs, the reducer that receives the highest workload requires a significantly longer amount of time to complete its task compared to its peers. As a result, an overall prolonged reduce-phase running time is introduced, which diminishes the benefits realized through parallelization. Figure 1(b) shows an example of the reducer workload ranking when we deploy the DBLP-1 data set in MapReduce using 80 reducers. Here the most loaded reducer takes 44x more workload than average. To address this problem, it has been suggested that the size of blocks can be profiled and leveraged for skew avoidance. More specifically, block size profiles have been used to design subkey schemes [16], block division strategies [11], and record pair allocation methods [11] to balance the load of the reducers.

However, establishing an accurate profile for block sizes is extremely challenging for massive data sets with millions or billions of block keys. For example, when one attempts to establish a per-block size profile for large data sets, such as DBLP-10 and DBLP-20 (we refer the readers to Table I for details), the profile needs to record the size for over 5 million blocks. As a result, the program crashes when the profile is loaded onto a virtual machine that assigns 1 GB memory for each task. In general, when blocking keys are linked to the input data, the blocking key’s value domain can be extremely large and hard to predict a priori. This clearly limits the applicability of precise profiling which establishes accurate per-block size information, and its associated load balancing approaches to big data sets where millions to billions of blocks could be the norm.

In this paper, we introduce scalable reduce-phase load balancing solutions for record linkage over the MapReduce framework. To do so, we address two specific problems: 1) how to design an efficient data structure that summarizes the block-related load information and 2) how to leverage information recorded in this structure to assign records to reducers so that their loads are balanced. To address the first

1Record linkage is also known as deduplication, entity resolution, merge-purge and name matching.

2In current generation of MapReduce/Hadoop (i.e., YARN [20]), 1 GB is the default allocated memory for each task.
problem, we introduce a sketch-based data profiling method [4] for capturing block size statistics. For context, informally, a sketch is a two-dimensional array of cells, each indexed by a set of pairwise independent hash functions. For the second problem, we present two load balancing algorithms – cell block division and cell range division – that directly operate on sketch-based data profiles to achieve reducer load balancing.

[23] also introduced a sketch-based solution to capture key group size statistics and presents sketch packing algorithm which assigns the key groups to the reducers in a load balancing manner. However, the sketch packing algorithm cannot be deployed with record linkage application, due to the following reasons. The sketch packing algorithm assumes the key group workload is proportional to its size, which cannot work here as record linkage is a join-like operation. Furthermore, the sketch packing algorithm works on the granularity of key groups and treats each key group as a indivisible unit. This characteristic makes the sketch packing algorithm as a general load balancing solution for several applications (e.g., PageRank, Inverted Indexing, etc.), but also limits its performance with highly skewed data sets (as demonstrated in Figures 3(d) and 4(d) in [23]). Data sets used in record linkage application are always highly skewed. In the example shown in Figure 1(a), where the maximum block takes 55% of total workload. We name this type block (key group) as expensive block (key group). Wherever the sketch packing algorithm assigns this expensive block, that reducer would be the straggler. These two reasons limit the sketch packing algorithm performance on record linkage application.

The main contributions of our work are summarized as follows. First, our sketch-based data profiling method is 1) scalable with the size of the input data and the number of blocks and 2) efficient for construction, while each update takes constant time. Second, our proposed cell block division and cell range division algorithms can efficiently divide expensive blocks without breaking the original record linkage semantics. Third, our theoretical analysis shows that our load balancing algorithms have a bounded load balancing performance, as well as computational complexity. Last, we perform an empirical study using several real-world and synthetic data sets and demonstrate that our algorithms can achieve near-optimal load balancing performance in comparison to precise profiling, with fixed memory and incurs only very small running time overhead.

The remainder of this paper is organized as follows. Section II illustrates the MapReduce-based record linkage implementation, and discusses the reduce-phase skew problem and the scalability issue involved in existing skew avoidance approaches. In Section III, we present our sketch-based data profiling method and the load balancing algorithms, as well as analytical results. Section IV provides experimental results. Finally, we discuss related work in Section V and conclude the paper in Section VI.

II. PROBLEM DESCRIPTION

A. Record Linkage

Record linkage is the process of matching records on specific entities (e.g., “John Smith” and “John S.”) may refer to the same person.) in disparate sources. Large-scale record linkage frameworks involve three fundamental data-intensive steps [7]. The first step is blocking, which uses a quick coarse-grained similarity filtering strategy to produce subsets (i.e., blocks) of the record pair set that contain pairs that likely correspond to the same entity (i.e., candidate record pairs). As a commonly used blocking strategy, each input record is assigned with a blocking key; records with the same key are grouped together into a block\(^3\). Only records within the same block are compared with each other in the ensuing comparison step. This step involves the assessment of multiple fields between a pair of records to produce a similarity vector. The third step is classification, which determines the match status of each record pair based on their similarity vectors and outputs the set of matches and non-matches.

As in previous work [11], [12], we focus on optimization for the first two steps because the final classification step can utilize several existing statistical strategies that are independent of the scale of the problem [8]. Thus, the whole process can be described as: given two sets of records \(R\) and \(S\), computing similarity vector for each pair of records from different data sets with the same blocking key. We refer to the similarity vector for records \(r_a\) and \(r_b\) as \(x = [x_1, x_2, ..., x_t]\) with \(t\) components that correspond to the \(t\) comparable fields. Each \(x_i\) shows the level of agreement of the \(i\)th field of the records \(r_a\) and \(r_b\). \(x_i\) is computed using a similarity function \(\text{sim}\), such as the edit distance or Q-gram distance.

B. MapReduce Programming Model

MapReduce [6] was proposed to simplify large-scale data processing on distributed and parallel architectures, particularly clusters of commodity hardware. The main idea of this programming model is to hide details of the data distribution and the load balancing and let users focus on the data processing aspects. A MapReduce program consists of two primitives, map and reduce, as shown below:

\[
\text{map} :: (K_1, V_1) \rightarrow \text{list}(K_2, V_2)
\]

\[
\text{reduce} :: (K_2, \text{list}(V_2)) \rightarrow \text{list}(V_3)
\]

In the map phase, input data is processed by map tasks in parallel; the intermediate outputs of the map tasks are collected locally and grouped based on their key values. Based on a (default hashing or user-defined) partition function, these groups are allocated to the appropriate reducers depending on their keys. Once the map phase is completed and the intermediate results have been transferred to the reducers, the reduce phase begins. In this phase, the reduce function is applied in parallel to each key group and produces the final results.

C. Blocking-based Record Linkage in MapReduce

Recently proposed approaches [12], [18], [21] have favored a common design to support the blocking-based record linkage process over MapReduce platform. In this design, the data sets are partitioned into several blocks using the blocking keys by map tasks and assigned to parallel reduce tasks, where record pairs are constructed for comparison. Figure 2 illustrates an example of blocking-based record linkage using the MapReduce framework, where field \(a\) acts as the blocking

\(^3\)For various blocking mechanisms, we refer readers to [2].
key. Records with the same blocking key are sent to the same reducer, where the similarity vectors are built.

A common issue faced by the current design of record linkage protocols over the MapReduce framework is reduce-phase data skew. When the block size distribution is highly skewed, the default MapReduce hash-based key-group-assignment mechanism can assign some reducers with much higher workload than others. This results in a prolonged reduce-phase running time. For example, in Figure 2, the first reducer needs to compare 7 record pairs, while the other two reducers only have to compare 2 and 4. Data skew happens because of the imbalance distribution of block sizes and the MapReduce default hashing partition mechanism.

To achieve reduce-phase load balancing, existing solutions introduce an additional MapReduce job to establish a block size profile [11, 16] which records the number of records within each block. This block size profile can then be used to design subkey [16], block division [11], or record pair allocation [11] schemes so that the load at the reducers can be balanced. For instance, in [11], a MapReduce job is included to build a block size matrix for the number of records within each block. A global index is then reported for each record pair and each reducer is assigned with an index range of equal length, thus achieving load balancing across reducers. A major limitation of the aforementioned block-based profiling method is its scalability. In reality, data sets to be linked can be extremely large – on the order of millions or billions. Thus, given current computing architectures, it is impossible to maintain a precise block size matrix in a space-limited environment.

In this paper, we seek an approximate data profiling method that is both memory and time-efficient. For such a method to scale to massively sized data sets, the memory cost should be independent of the number of blocks and the processing time should be linear (or sublinear) in the input data set. Moreover, the data profile should introduce a bounded approximation error and yield a highly accurate load balancing strategy, regardless of the data skew.

### III. OUR APPROACH

Figure 3 illustrates the overall design of our system with an illustration of linking two data sets $R$ and $S$. Similar to the approaches adopted in [11], our design is based on two rounds of MapReduce jobs. The profiling job analyzes the input data sets and provides the load estimation in terms of sketch for input data sets $R$ and $S$. The load information is loaded by map tasks into the second MapReduce job (comparison), where load balancing strategies are applied to perform the actual record linkage task.

#### A. Sketch-based Data Profiling

We first describe the centerpiece of our approach – how to build, and achieve load balancing based upon, the sketch. A sketch [4, 5] is a data structure that provide space-efficient summaries for massive, rapid-rate data streams. Here, we use the sketch data structure to estimate block sizes for the input data. Specifically, we use the FastAGMS sketch [4] because it provides the most accurate estimation for the size of join operation4 [19], regardless of data skew. We also evaluate other type of sketches in Section IV, such as Count-Min sketch [5].

To estimate the workload in terms of record-pair comparisons within each block, we maintain two FastAGMS sketches for data sets $R$ and $S$, which we refer to as $C_R$ and $C_S$, respectively. Each FastAGMS sketch $C$ ($C \in \{C_R, C_S\}$) maintains a two-dimensional array of cells with $d$ rows of width $w$, which are indexed by a set of pairwise independent hash functions $H = \{h_i, i = 1, \ldots, d\}$. Each hash function $h_i$ maps a blocking key $k$ into a hashing space of size $w$ (i.e., $h_i(k) \in \{0, 1, \ldots, w - 1\}$)$^5$. The FastAGMS sketch also maintains a family of $\pm 1$ four-wise independent hash functions $G = \{g_i, i = 1, \ldots, d\}$6 The unique family $G$ preserves the dependencies across the counters.

Each cell of the sketch carries a counter. Initially, all of the counters in the array are set to zero:

$$C[i, j] = 0, \text{ for all } i \in \{1, \ldots, d\}, j \in \{1, \ldots, w\}$$

---

4Record linkage can be treated as a join operation where the blocking key act as the join key.

5$h_i(k) = (a_i k + b_i) \mod w$.

6$g_i(k) = \left\{ \begin{array}{ll} 1 & \text{if } (a_i k^3 + b_i k^2 + c_i k + d_i) \mod 2 = 0 \\ -1 & \text{otherwise} \end{array} \right.$
Algorithm 1 Update operation for sketches

1: function UPDATE(r, C_R, C_S, H, G)
2: \hspace{1em} // calculate blocking key
3: \hspace{2em} k \leftarrow \text{calculateBKV}(r)
4: \hspace{1em} // update sketch
5: \hspace{2em} if r \in R then
6: \hspace{3em} for i = 1 \rightarrow d do
7: \hspace{4em} C_R[i, h_i(k)] \leftarrow C_R[i, h_i(k)] + g_i(k)
8: \hspace{3em} end for
9: \hspace{2em} else if r \in S then
10: \hspace{3em} for i = 1 \rightarrow d do
11: \hspace{4em} C_S[i, h_i(k)] \leftarrow C_S[i, h_i(k)] + g_i(k)
12: \hspace{3em} end for
13: \hspace{2em} end if
14: end function

When a new blocking key \( k \) is emitted, the counters are updated as shown in Algorithm 1.

\[ C[i, h_i(k)] = C[i, h_i(k)] + g_i(k), \text{ for all } i \in \{1, ..., d\} \]

Specifically, for each row \( i \), \( h_i(k) \) determines the cell to be updated, and \( g_i(k) \) decides whether to increment or decrement the counter in the corresponding cell. Figure 4 illustrates this update process with \( w = 9 \) and \( d = 4 \). Here, a key-value pair with key \( k \) is mapped to a counter in each row \( i \) \( (i \in \{1, 2, ..., d\}) \) by the hash function \( h_i \) and increments the counter by \( g_i(k) \).

The sketches \( C_R \) and \( C_S \) are constructed based upon the same parameters (i.e., \( d, w, H, G \)). Each sketch provides an approximate summary of block sizes \( L(k) \) for each data set, where the sizes of multiple blocks are compressed into one cell. Recall that the record linkage workload in terms of record pair comparisons is the product of the block sizes from these two data sets \( L_R(k) \times L_S(k) \). To estimate this workload, we consider the inner product of the sketches, which is accomplished through two steps. In the first step, we choose the median value of the row inner products. Formally, the row inner product is provided by

\[ C^i = \sum_{j=1}^{w} C_R[i, j] \times C_S[i, j], i \in \{1, 2, ..., d\} \]

Let the value of row \( i = \theta \) be the median value among these \( d \) row inner products. In the second step, we use this row to build a counter array \( C^\theta \) with width \( w \) as follows.

\[ C^\theta[j] = C_R[\theta, j] \times C_S[\theta, j], \text{ for all } j \in \{1, ..., w\} \]

The array \( C^\theta \) provides the estimation of the record-pair comparison workload within blocks and will be applied for load balancing algorithms.

**Implementation in MapReduce.** The map tasks of the profiling job build local sketches \((C^\theta_R \text{ and } C^\theta_S)\) based on the input records from the two data sets. Once completed, the local sketches are sent to one reducer, where all local sketches are combined to build the final sketch \((C_R \text{ and } C_S)\). Sketch combination is straightforward: local sketches with same sizes are combined by summing them up, entry-wise.

The outputs of the profiling job are the final inner product sketch \( C^\theta \), as well as the corresponding row vectors from the sketches of the two data sets \((C^\theta_R \text{ and } C^\theta_S)\). Our load balancing algorithms work directly on these values.

**B. Cell Block Division Algorithm**

The \( C^\theta \) from the profiling job is loaded by each map task in the comparison job before processing the input records. As noted earlier, \( C^\theta \) provides an estimation of the record comparison workload, where each cell carries the estimated workload for multiple blocks. Let \( L \) represent the estimated overall workload, then \( \hat{L} = \sum_{j=1}^{w} C^\theta[j] \). The average reducer workload can then be estimated as \( \hat{L}/n \), where \( n \) is the number of reducers.

Since \( C^\theta \) contains \( w \) cells, a simple idea that might come to mind is to pack these cells into \( n \) partitions and assign each reducer to one partition. However, some cells may have a workload larger than the average \( \hat{L}/n \), so a division procedure is required for large cells (i.e., the function DivideCell in Algorithm 2). For each cell in \( C^\theta \), if its estimated workload is larger than average, we divide it into several subcells; otherwise, we keep the cell as a single subcell. Lines 4 – 9 in Algorithm 2 illustrate the process of calculating subcells. All subcells are maintained in a set \( S \). Finally, a packing operation is performed on the set \( S \). The result \( \Phi \) is a mapping from subcells to reducers.

Figure 5 presents an example of cell block division with \( w = 4 \) and an estimated workload 10. \( C^\theta \) contains four cells, and their workload is \( \{1 \times 1, 2 \times 3, 1 \times 1, 2 \times 1\} \). Now assume there are three reducers and the estimated average workload is 4. Notice, cell \( C^\theta[2] \) is larger than the average, so it is divided into two subcells. In our implementation, we always follow row-based division, such that cells are divided along the axis correspond to \( R \). After division, \( S \) has five (sub)cells. Next, we perform the packing operation, where each (sub)cell is assigned to the reducer with the minimum workload.

Based on the mappings \( \Phi \), the map tasks in the comparison round identify the corresponding reducers for each record \( r \), to which \( r \) needs to be sent as in function GetReducer in Algorithm 2. First, the corresponding cell \( j \) for the given record \( r \) is calculated (line 19). For each record from data set \( R \), as we performs row-based division, we only need to send \( r \) to one reducer. Since each cell is divided into \( D[j] \) subcells with same size, we can randomly select a subcell and obtain the matching reducer for this subcell (lines 22 – 24). For each record from \( S \), we need to send it to all reducers that map to current cell (lines 26 – 27).

For example, in Figure 5, assume record \( r \) is hashed to \( C^\theta[2] \) which has two subcells. If \( r \) comes from data set \( R \), we randomly select one subcell and send \( r \) to reducer \( T_1 \) or \( T_2 \). If \( r \) is from \( S \), we send \( r \) to both reducers \( T_1 \) and \( T_2 \).
Algorithm 2 Cell Block Division Algorithm

1: function DIVIDECELL($C^0$, $n$)
2:  // calculate estimated workload
3:  $L \leftarrow \sum_{j=1}^{\infty} C^0[j]$
4:  // calculate division number for each cell
5:  for $j = 1 \rightarrow w$ do
6:    $D[j] \leftarrow Math.cell(C^0[j] \times n/L)$
7:    subcells[$i$] $\leftarrow createSubCells$(row, j, D[j])
8:    $S \leftarrow S \cup subcells$
9:  end for
10:  // perform packing operation on (sub)cells
11:  sort(S)
12:  for all subcell $\in S$ do
13:    reducerID $\leftarrow$ selectMinLoadedReducer()
14:    $\Phi(subcell) \leftarrow reducerID$
15:  end for
16: end function

Algorithm 3 Cell Range Division Algorithm

1: function DIVIDRANGE($C^0$, $n$)
2:  // calculate estimated workload
3:  $L \leftarrow \sum_{j=1}^{\infty} C^0[j]$
4:  // assign each range to one reducer
5:  $avg \leftarrow L/n$
6:  for $i = 1 \rightarrow n$ do
7:    $\Phi \leftarrow \Phi \cup \{(i-1) \times avg + 1, i \times avg, i\}$
8:  end for
9:  // calculate offset of each cell
10:  $O[1] \leftarrow 0$
11:  for $j = 2 \rightarrow w$ do
12:    $O[j] \leftarrow O[j-1] + C^0[j-1]$
13:  end for
14: end function

15: function GETREDUCER($r, \mathcal{H}, \theta, C^0, C_R, C_S, \Theta, O, \Phi$)
16:  // cell index for current record $r$ in row $\theta$
17:  cell $\leftarrow h_\theta(r)$
18:  // calculate the start and end indexes
19:  if $r \in R$ then
20:    rand $\leftarrow$ random($C_R[\theta, cell]$)
21:    $o_1 \leftarrow O[cell] + C_S[\theta, cell] \times rand$
22:    $o_2 \leftarrow o_1 + C_S[\theta, cell]$
23:  else if $r \in S$ then
24:    $o_1 \leftarrow O[cell]$
25:    $o_2 \leftarrow O[cell] + C^0[cell]$
26:  end if
27:  // assign record to reducers
28:  reducerIDs $\leftarrow$ getReducerIDs($\Phi, o_1, o_2$)
29: end function

Through such a cell block division approach, we can limit the maximum cell workload and achieve better load balancing performance. In Figure 5, the maximum reducer workload is 4 (reducer $T_2$). If no division introduced, the reducer that is assigned with $C^0[2]$ would become the straggler and have workload of 6.

C. Cell Range Division Algorithm

The cell block division algorithm divides large cells may still lead to unbalanced reducer workloads due to variation in the size of the subcells. To account for this problem, we now present a more sophisticated pair-based load balancing strategy that strives to generate a uniform number of pairs for all reduce tasks.

Each map task processes $C^0$ and can therefore enumerate workload per cell. We label each record pair in $C^0$ with a global index, and divide the whole record pairs into $n$ equal-length ranges (lines 4 – 8 in Algorithm 3). For a given cell $C^0[j]$, the overall number of record pairs in all preceding cells has to be added as index offset, and we maintain the offset in an array $O$ (lines 9 – 13). To make it simpler, our cell range division mechanism treats each row in a cell $C^0[j]$ as a unit and would not divide it.

Figure 6 presents an example of the cell range division with $w = 4$ and an estimated workload 10. Each record pair is labeled with an index, from 1 to 10. Let the reducer number is 3, and the total set of record pairs is divided into three ranges. Reducers $T_1$, $T_2$, and $T_3$ will process the record pairs with indexes in the ranges [1,4], [5,8] and [9,10], respectively.

Function GetReducer in Algorithm 3 calculates the reducers that record $r$ is sent to. First, we find the corresponding cell for $r$. After that, we calculate the start and end indexes for record pairs that are related to $r$. If $r$ comes from data set $R$, we randomly select a row for $r$ in its corresponding cell. The start index $o_1$ is calculated as all preceding record pairs (line 21), and the end index $o_2$ is the end of the selected row (line 22). For each record from $S$, we need to send it to all reducers that map to current cell (lines 24 – 25). Finally, we find out the reducers by giving the start and end record pair indexes (line 28).

For example, in Figure 6, assume a record $r$ is hashed to $C^0[2]$. If $r$ comes from $R$, we randomly select a row in $C^0[2]$, and would not divide it.
and send $r$ to reducer $T_1$ or $T_2$. If $r$ comes from $S$, we need to send $r$ to both reducers $T_1$ and $T_2$.

D. Performance Analysis

Here we analyze the memory and computational complexities of our proposed algorithms, and the load balancing performance in terms of the reduce-phase imbalance ratio.

Proposition 1: The memory complexity of our profiling method is $O(d \times w)$, where $d$ is the sketch depth and $w$ is the width. The computational complexity of sketch update is $O(d)$.

Proof: In the profiling job, each map/reduce task maintains two sketches with size $d \times w$, thus its memory complexity is $O(d \times w)$. The outputs of the profiling job are $C^0_R$, $C^0_S$, and $C^i_R[i]$ and $C^i_S[i]$, each of which is an array with width $w$. The map tasks in the comparison job load them into memory, thus introduce memory cost of $O(w)$. So the total memory complexity is $O(d \times w)$.

As each update process involves with $d$ counters, thus the computational complexity is $O(d)$.

Same to previous work [11], [18], we measure the load balancing performance bound on the reduce-phase imbalance ratio.

Definition 1: Reduce-phase imbalance ratio $\rho$: Let $L_i$ represent the workload of reducer $T_i$, the imbalance ratio $\rho$ can be calculated as normalizing the maximum reducer workload by the average workload.

$$\rho = \frac{\max_{i=1}^{n} L_i}{\sum_{i=1}^{n} L_i/n}$$

To analyze the maximum reducer workload in our algorithms, we firstly bound the estimated workload $\hat{L}$.

Lemma 1: According to the analysis in [4], the workload estimation $\hat{L}$ for two given FastAGMS sketches $C_R$ and $C_S$ with size $d \times w$, has the following guarantee: $\hat{L} \in (L \pm \varepsilon ||R||_2 ||S||_2)$, with probability at least $1 - \delta$. Here $\varepsilon = \varepsilon/w$, $\delta = 1/e^\beta$, $\beta$ is the base of the natural logarithm, $L$ is the accurate workload, and $||.||_2$ is the $L_2$-norm.

Next we analyze the load balancing performance bound on the reduce-phase imbalance ratio of our proposed algorithms in the following theorems.

Theorem 1: The reduce-phase imbalance ratio of the cell block division algorithm is at most $(2 - \frac{1}{n})(1 + \frac{1}{n})$, with a probability of at least $1 - \delta$, where $\Delta = \varepsilon ||R||_2 ||S||_2$.

Proof: Suppose reducer $s$ receives the maximum workload and subcell $sc$ is the last one it has been assigned. Let $L_s$ be the workload of reducer $s$ before it receives $sc$. When subcell $sc$ was assigned, the workload of its reducer was no larger than the other reducers, so every reducer at that time has a larger workload than $L_s$. Thus, the maximum reducer workload is

$$L_{\text{max}} = L_s + L_{sc} \leq \frac{\hat{L} - L_{sc}}{n} + L_{sc} = \frac{\hat{L}}{n} + (1 - \frac{1}{n})L_{sc}$$

Since each subcell has a workload that is less than $\hat{L}/n$, it can be stated

$$L_{\text{max}} \leq \frac{\hat{L}}{n} + (1 - \frac{1}{n})\frac{\hat{L}}{n} = (2 - \frac{1}{n})\frac{\hat{L}}{n}$$

Now, let $\Delta = \varepsilon ||R||_2 ||S||_2$. It is the case that

$$L - \Delta \leq \hat{L} \leq L + \Delta$$

and the imbalance ratio is at most

$$\rho = \frac{(2 - 1/n)\hat{L}/n}{L/n} \leq \frac{(2 - 1/n)(L + \Delta)}{L} = (2 - 1/n)(1 + \Delta/L).$$

Theorem 2: The reduce-phase imbalance ratio of the cell range division algorithm is at most $1 + \frac{2}{n}$ with a probability of at least $1 - \delta$, where $\Delta = \varepsilon ||R||_2 ||S||_2$.

Proof: Let $\Delta = \varepsilon ||R||_2 ||S||_2$. Then, it is case that

$$L - \Delta \leq \hat{L} \leq L + \Delta,$$

with a probability at least $1 - \delta$. As each reducer has been assigned with $\hat{L}/n$ estimated workload, the maximum reducer workload $L_{\text{max}} = (L + \Delta)/n$. The imbalance ratio is at most

$$\rho = \frac{(L + \Delta)/n}{L/n} = 1 + \frac{\Delta}{L}.$$
TABLE I. DATA SET SUMMARY

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Records (million)</th>
<th>Blocks (million)</th>
<th>Pairs (billion)</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-1</td>
<td>2.5</td>
<td>0.6</td>
<td>0.5</td>
</tr>
<tr>
<td>DBLP-5</td>
<td>12.6</td>
<td>3.3</td>
<td>1.8</td>
</tr>
<tr>
<td>DBLP-10</td>
<td>25.2</td>
<td>8.7</td>
<td>4.0</td>
</tr>
<tr>
<td>DBLP-20</td>
<td>50.4</td>
<td>16.9</td>
<td>10.0</td>
</tr>
<tr>
<td>Synth-0.5</td>
<td>416</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>Synth-1.0</td>
<td>226</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>Synth-1.5</td>
<td>32.6</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>Synth-2.0</td>
<td>21.6</td>
<td>5.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>

master node. Each node has one 2.4 GHz Intel Core2 CPU with 2 GB of memory. The HDFS block size was set to 64 MB and each node was configured to run at most two map tasks and two reduce tasks concurrently. We disabled the speculative task execution feature to better analyze the running time of each task. By default, each MapReduce job is configured with 80 reducers.

**Baseline Algorithms.** We evaluate four algorithms: 1) the Hadoop default (HD) algorithm, which uses the hash-based partition function for key group assignment; 2) the pair-based (PR) algorithm [11], which utilizes the precise block profile; 3) our cell block division (CB) algorithm (Algorithm 2); and 4) cell range division (CR) algorithm (Algorithm 3).

The performance of the algorithms is measured in terms of 1) job running time, which is the entire running time including both profiling and comparison rounds; 2) imbalance ratio of reducer workload in the comparison job, which is calculated by normalizing the maximum reducer workload by its average. Here the reducer workload is measured as the number of record pairs received by each reducer.

**Sketch and Profile Settings.** In most experiments, the sketch structure is set with size $w = 10000$ and $d = 10$, which requires less than 1 MB memory.

**B. Performance of CB and CR algorithms**

We first evaluate the performance of the CB and CR algorithms using the DBLP data sets. Figure 7 shows the running time of each phase. No results are shown for the PR algorithm in the experiments with the DBLP-10 and DBLP-20 data sets because the program crashed on account of a OutOfMemory exception. In other words, the precise block size profile could not be maintained in memory.

In comparison to HD, the other three algorithms introduce another MapReduce job (profiling) to build a data profile. As such, their overall job running time also includes the time spent in the profiling job. It can be seen that PR, CB and CR, which balance the load among reducers, all significantly reduce the job running time compared with the HD algorithm.

Figure 7(a) and 7(b) show that PR, CB and CR have similar running time. Further, CB and CR require less time in the ReduceProfiling and MapComparison (which is particularly noticeable in Figure 7(b) with 34.1% reduction). This is because our algorithms operate on sketch (which are time-efficient for construction and retrieval), while PR algorithm normally maintains a hash table in memory (whose operation time increases dramatically with its size when collision exists).

In comparison to the HD algorithm, the profiling job in the CB and CR algorithms introduces 3% extra running overhead in average. However, the entire running time is reduced by 71.56% and 70.73%, respectively.

We also record the number of record pairs processed by each reducer and calculate the reduce-phase imbalance ratio. Both CB and CR can achieve nearly optimal reducer-side load balancing with an imbalance ratio of reducer running time around 1.1 as shown in Figure 8. Moreover, in comparison to the PR algorithm, our algorithms increase the imbalance ratio by only 2.5%, indicating very small load balancing performance penalty introduced by approximate data profiles. The imbalance ratio of HD highly depends on the input data and is always much higher than the other algorithms.
C. Performance under various data skew

We further evaluate the CB and CR algorithms under various data skew scenarios using the Synth-α data sets. In this study, we mainly focus on the reduce-phase imbalance ratio in the comparison MapReduce job, which has 80 reducers. As shown in Figure 9, the CB and CR algorithms have similar performance. The imbalance ratio remains around 1.25. Since both CB and CR divide large cells, their performance is highly stable with respect to the amount of skew of the data set.

D. Performance under number of reduce tasks

We now study the influence of the number of reducers \( n \) in a fixed cloud environment of 40 nodes. We evaluate the CB and CR algorithms with the DBLP-20 data set, and vary the reducer number from 80 to 400. As shown in Figure 10, both CB and CR maintain stable performance as the number of reducers increases.

E. Experiments with sketch size

Here we fix the sketch depth \( d = 10 \), and vary the width \( w \) from 100 to 10000. Figure 11 shows that the imbalance ratio is inversely correlated with the sketch width. This phenomenon is also reflected in our theoretical analysis – when the number of reducers increases, the workload estimation error decreases. Our empirical study shows that the best performance is achieved when the sketch width is set to 50 to 100 times that of the number of reducers.

F. Experiments with various types of sketches

In this paper, we use the FastAGMS sketch [4] as the default implementation of sketch data structure as it provides the most accurate workload estimation [19]. There are also other sketch implementations, such as Count-Min sketch [5].

In this study, we deploy Count-Min sketch in our CB and CR algorithms and compare the results with the default FastAGMS implementation. We use the DBLP-20 data set, and calculate the reduce-phase imbalance ratio for each group of experiment. Figure 12 illustrates that FastAGMS sketch performs much better than Count-Min sketch.

G. Comparison with Sketch Packing algorithm

As we illustrated in Section III-B, the \( C^θ \) provides an estimation of the record comparison workload, where each cell carries the estimated workload for multiple blocks. Since \( C^θ \) contains \( w \) cells, a simple idea is to pack these cells into \( n \) partitions and assign each reducer to one partition. This idea is very similar to the sketch packing algorithm proposed in [23].

Here we evaluate the sketch packing approach (SP), in comparison to our CB and CR algorithms that further divide each cell. Data set DBLP-20 is used here, with 80 reducers. Figure 13 shows the reducer workload imbalance ratio. As the SP algorithm doesn’t divide cells, its performance is quite limited by the data skew.

V. RELATED WORK

Record linkage was first proposed by the public health community in [17], which matched marriage and birth records
In general, record linkage can be treated as a join operation, and the blocking key acts as the join key. There are several algorithms that try to optimize join operations in the MapReduce framework, including reduce join, fragment-replication join and map-merge join. [1] presented a detailed analysis of these mechanisms. A more general case is the computation of theta-join with MapReduce [18]. Similar to our approach, [18] employed a pre-analysis phase to determine the datasets characteristics (using sampling) and thereby avoids the evaluation of the Cartesian product.

Besides these application-aware solutions that focus on a particular application, there are also some load balancing approaches that work with general applications in MapReduce. [14] provided a taxonomy of MapReduce skew, which includes map-phase skew and reduce-phase skew. [9] presented a solution for reduce-phase skew, which assigns key groups to reducers based on a locality/fairness model. This approach requires loading all key group size information into memory to perform the packing operation, which is impractical when massive datasets are processed.

To build scalable load balancing solution, [23] introduced sketch-based profiling to capture all key group size information in a sketch structure. Instead of performing packing operation on key groups directly, [23] presented optimal sketch packing algorithm that works on the sketch directly, thus providing a scalable load balancing solution.

All aforementioned approaches are skew-avoidance solutions, which always introduce a profiling MapReduce job before the real job running. Another imbalance-handling mechanism (e.g., SkewTune [15]) deploys an online approach, which does not require a key group size profiling step. SkewTune dynamically monitors the task execution and estimates the remaining time for each task. Whenever a machine becomes idle, SkewTune mitigates workload from overloaded machines to the idle machine. SkewTune may incur significant communication overhead when moving workload across different machines, especially with massive dataset.

All of these approaches work well in mitigating reduce-phase data skew. However, they work on the granularity of key groups, thus cannot be deployed with record linkage application directly. In record linkage, the reduce-phase data skew is always caused by one or several expensive key groups. The size of expensive key groups is significantly larger than other key groups and exceeds the average workload. Even if one such group is assigned to a particular reduce task, that task will still be a straggler in the reduce phase.

**VI. CONCLUSION**

In this paper, we presented a scalable solution to achieve load balancing record linkage over the MapReduce framework. The solution contains two low-memory load balancing algorithms that work with the sketch-based approximate data profiles. We performed a theoretical and an empirical analysis on both real-world and synthetic data sets to demonstrate that, compared with the state-of-the-art solution, our algorithms have nearly same load-balancing performance while requiring much less memory. We plan to deploy such a sketch-based approach to resolve data skew problem with other applications over MapReduce.

---

Fig. 12. Reduce-phase imbalance ratio under various sketches

Fig. 13. Reduce-phase imbalance ratio in comparison to sketch packing algorithm

Parallellism is another complementary approach to reduce the running time of record linkage application, especially the development of MapReduce framework. [21] presented an in-depth study of parallel set-similarity join in MapReduce. [11] discussed the reduce-phase data skew problem in MapReduce-based record linkage and presented two load balancing solutions. To cope with data skew, these solutions first profile the input data sets to collect block sizes and then utilize these information to achieve reduce-phase load balancing. However, the scalability issue associated with the profiling process was not considered, and their approaches would fail when handling massive data sets (e.g., DBLP-10 and DBLP-20 in Table I).

to better understand relationships within a human population. Then, Fellegi and Sunter [8] formalized the intuition and showed the optimality of probabilistic decision rule by introducing formal mathematical model. The next major development came from Winkler’s adaptation of the Expectation-Maximization algorithm to estimate parameters which tightened probabilistic decision rules [22]. All of these approaches mainly focused on how to make deduplication more accurate, while another research direction was how to reduce the time cost of deduplication process.

To reduce the record comparison space, [10] introduced a blocking mechanism. This mechanism used a quick coarse-grained similarity filtering strategy to produce subsets (i.e., blocks) of the record pair set that contain likely matching pairs (i.e., candidate record pairs). Only records in the same block need to be compared with each other. [2] surveyed several different blocking implementations.

Parallelism is another complementary approach to reduce the running time of record linkage application, especially the development of MapReduce framework. [21] presented an in-depth study of parallel set-similarity join in MapReduce. [11] discussed the reduce-phase data skew problem in MapReduce-based record linkage and presented two load balancing solutions. To cope with data skew, these solutions first profile the input data sets to collect block sizes and then utilize these information to achieve reduce-phase load balancing. However, the scalability issue associated with the profiling process was not considered, and their approaches would fail when handling massive data sets (e.g., DBLP-10 and DBLP-20 in Table I).
ACKNOWLEDGMENT

We would like to thank the anonymous referees for their useful reviews that helped us to improve the quality of the paper. This research is supported in part by the National Institutes of Health (R01LM009989) and the National Science Foundation (CCF-0424422).

REFERENCES


