Graph-Parallel Entity Resolution using LSH & IMM

Pankaj Malhotra
TCS Innovation Labs, Delhi
Tata Consultancy Services
Ltd., Sector 63
Noida, Uttar Pradesh, India
malhotra.pankaj@tcs.com

Puneet Agarwal
TCS Innovation Labs, Delhi
Tata Consultancy Services
Ltd., Sector 63
Noida, Uttar Pradesh, India
puneet.a@tcs.com

Gautam Shroff
TCS Innovation Labs, Delhi
Tata Consultancy Services
Ltd., Sector 63
Noida, Uttar Pradesh, India
gautam.shroff@tcs.com

ABSTRACT

In this paper we describe graph-based parallel algorithms for entity resolution that improve over the map-reduce approach. We compare two approaches to parallelize a Locality Sensitive Hashing (LSH) accelerated, **I**terative **M**atch-**M**erge (IMM) entity resolution technique: BCP, where records hashed together are compared at a single node/reducer, vs an alternative mechanism (RCP) where comparison load is better distributed across processors especially in the presence of severely skewed bucket sizes. We analyze the BCP and RCP approaches analytically as well as empirically using a large synthetically generated dataset. We generalize the lessons learned from our experience and submit that the RCP approach is also applicable in many similar applications that rely on LSH or related grouping strategies to minimize pair-wise comparisons.

1. MOTIVATION AND INTRODUCTION

The map-reduce (MR) parallel programming paradigm [9] and its implementations such as Hadoop [24] have become popular platforms for expressing and exploiting parallelism due to the ease with which parallelism can be abstracted to a higher-level. However, it has become increasingly apparent that there are classes of algorithms for which MR may not be well suited, such as those involving iterative or recursive computation. Graph-based parallelism via the Pregel programming paradigm [19] and its various implementations such as Giraph [2], Graphlab [18], or GPS [22] is an alternative approach that has been shown to perform better in such scenarios, for example in sparse-matrix multiplications, page-rank calculation, or shortest-paths in graphs etc.

In this paper we focus on the entity resolution (ER) problem and submit that graph-based parallelism is better suited for it. We consider the iterative match-merge (IMM) approach to ER [4], accelerated by locality-sensitive hashing (LSH) [1] to avoid unnecessary comparisons. In this context we found that the strategy that is natural if using MR, i.e., where records hashed to the same bucket are compared at a single reducer / node, need not be the most efficient approach. Instead, the graph-parallel model offers an alternative mechanism that is better at distributing the computational load across processors.

(c) 2014, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2014 Joint Conference (March 28, 2014, Athens, Greece) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

We introduce the IMM+LSH approach for ER later in this section. Next, in Section 2, we discuss how this entity-resolution algorithm should be parallelized, via MR as well as its natural translation to the graph-parallel model, which we call bucket-centric parallelization (BCP). We then describe our alternative technique for record-centric parallelization (RCP). In Section 3, we analyze the BCP and RCP approaches analytically as well as empirically using a large synthetically generated dataset.

We believe the lessons learned from this exercise of parallelizing ER are more general. The RCP approach appears better suited at dealing with the skewed work-loads that naturally arise when items need to be compared efficiently using probabilistic hashing or executing similarity joins. Further, when records containing large attributes (such as documents or images) need to be matched, albeit even exactly as in standard multi-way joins, our experience here leads us to suggest mechanisms to avoid unnecessary communication of large attributes that do not figure in the final matching result. We present these learnings in Section 5, after describing related work in Section 4.

1.1 Entity Resolution via LSH and IMM

We assume that information about real-world entities is available from disparate data sources in the form of records, with each record (such as a passport, or driving license) belonging to a unique real-world entity. Two records are said to *match* if a suitable match function returns 'true' indicating that the records belong to the same entity. Match functions can be implemented in different ways, such as rules, or even binary classifiers derived via machine learning. Under certain conditions [4] matching records may be *merged* to produce a new record that derives its attributes and values (e.g. via a union) from the matching input records.

Given a collection of records where each record belongs to a unique entity, **ER** seeks to determine disjoint subsets of records where the records in a subset match under some match function and form an entity by merging these records. For example, different records belonging to the same person, such as voter card, passport, and driving-licence should get merged to form one entity.

The R-Swoosh IMM algorithm as described in [4], performs ER as follows: Initialise the set I to contain all records and the set I' to be empty. R-Swoosh iterates over the records in set I, removing a record r from I and comparing it to records in I'. As soon as a matching record r' is found in I', it is removed from I', and a new record obtained by merging r and r' is added to I. On the other hand, if no matching record is found in I', the record r is added to I'. The procedure continues until the set I is empty and I' contains a set of merged records representing the resolved entities.

The time complexity of IMM is quadratic in the number of records to be resolved, so it makes sense to attempt to pre-group records so that records in different groups are highly unlikely to

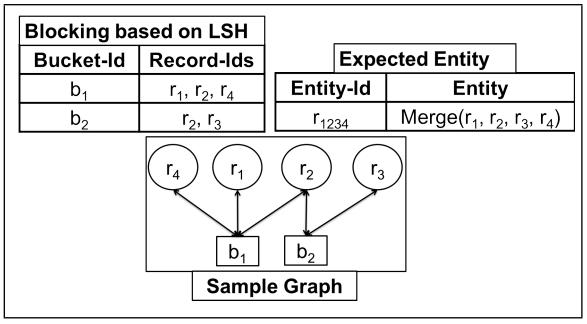

Figure 1: Sample Data and Corresponding Graph

match with each other. One way to achieve such grouping is via LSH [1]. LSH hashes each record to *one or more buckets* so that records that do not share a bucket are highy unlikely to match [20]. Therefore, instead of performing IMM on the entire set of records, only records belonging to the same bucket are considered for IMM.

2. PARALLEL ENTITY RESOLUTION

2.1 Bucket-centric Approach

2.1.1 BCP using Map-Reduce

We use standard LSH via minhashing [5] to create buckets of potentially similar records. A Match function implementation may not use all the attributes in records to compare them. In other words, not all attributes in records may be relevant for the purpose of comparison of records. We use only the words occurring in the values of the relevant attributes for the purpose of hashing also. Each such relevant word occurring in any of the records is mapped to a unique integer. For each record, we consider the set of integers S corresponding to the set of relevant words it contains. The minhash for the set S is calculated as the $min((sx_i + z) \bmod Q)$, $\forall x_i \in S$ (where s and z are random integers, and Q is a large prime). Different combinations of s and z determine different hash functions. For each record, we compute $a \times b$ hash values using $a \times b$ minhash functions. Out of these hash-values of a record, a are concatenated to get a *bucket-id*, we therefore get a total of b such bucket-ids for each record. Finally, a bucket consists of the record-ids of all the records that get hashed to the corresponding bucket-id.

A natural way to execute the above procedure in parallel using MR is to generate b key-value pairs [*bucket-id*, *record*] for every record in the map phase [8]. Records mapped to the same bucket-id are sent to a common reducer where IMM is run for each bucket. The result of IMM in each bucket is a set of 'partially' resolved entities since the possibility remains for records belonging to a single entity to get mapped to more than one bucket.

Consider the example shown in Figure 1, we have a collection $\Re$ of four records: $\Re = \{r_1, r_2, r_3, r_4\}$ such that all the records belong to the same entity r_{1234}. The match function applied to any pair of records in $\Re$ gives `true`. Assuming $a = 1$ and $b = 3$, each of the 4 records is hashed to 3 buckets using LSH. Of the buckets generated by LSH on the records in $\Re$, only two buckets b_1 and b_2 end up with more than one record being hashed to them. The *singleton buckets*, i.e., the buckets having only one record hashed to them are not shown in the Figure 1. Therefore, bucket to record mapping becomes $\{b_1, \{r_1, r_2, r_4\}\}$, $\{b_2, \{r_2, r_3\}\}$. As a result of IMM on b_1, we get a partial-entity e_{b1} consisting of $\{r_1, r_2, r_4\}$. Similarly, IMM on b_2 gives another partial-entity e_{b2} consisting of $\{r_2, r_3\}$. The 'partial-entities' e_{b1} and e_{b2} belong to the same entity since they contain an overlapping record r_2. We consolidate the *partial-entities* emerging from all the buckets by computing connected components in a graph of records (similar to [23]), where an edge exists between two records if they belong to the same partial-entity, as shall be explained later. If a pair of partial-entities e_a and e_b happen to share at least one of the original records r_i, they end up being in the same connected component and all the records in them get merged.

The above approach using MR has two potential problems: First, a large number of buckets are singletons, so many reduce keys get only one record as a value. Such records do not need to be compared with any other record, so sending them to the reducers is unnecessary and causes significant communication overhead especially when records are large in size. Secondly, we need to find connected components after the first MR phase, which is itself an iterative procedure and is likely to perform better using the Pregel model.

2.1.2 BCP using Pregel

Consider using MR to generate LSH buckets and discard singletons, and a graph-parallel approach using the Pregel paradigm thereafter. A high-level block diagram of how this works is shown in Figure 2. We perform LSH using MR as earlier except that instead of passing the records themselves we ensure that mappers only send record-ids, thus significantly reducing communication costs. In the reduce phase, instead of running IMM we merely generate the adjacency-list of a graph with two types of vertices, a *record-vertex* for each record in the collection and a *bucket-vertex* for each bucket obtained through LSH for collection of records, along with edges between them as follows: A *record-vertex* has outgoing edges to all the bucket-vertices corresponding to the buckets it gets hashed to. A *bucket-vertex* has outgoing edges to all the record-vertices which are hashed to it.

Note that since reducers know the size of every bucket they are responsible for, singletons are easily removed at this stage itself, i.e., *no* vertices are created for singleton buckets. As a result, only buckets containing record-ids that need to be compared are passed on to subsequent stages, eliminating the need to ship record contents for records in singleton buckets.

Note also that edges in the resulting graph are bi-directional, i.e., if there is an edge from v to v' then there is also an edge between v' and v. The adjacency-list files created by MR and also files mapping record-ids to records are inputs to a graph-parallel Pregel-based platform (such as Apache Giraph [2]), so that record-vertices are initialised with both record-ids as well as the full content of each record. Thereafter graph-parallel computations in the Pregel-model proceed via a number of supersteps as follows:

SS1: Each bucket-vertex is to perform IMM on all records that are hashed to it. Initially, only the IDs of the records hashed to a bucket are available at bucket-vertex via its outgoing edge-list. So in this superstep each record-vertex sends its value (which includes the record's content) to the bucket-vertices in its outgoing edge-list (which is the list of all the non-singleton buckets the record is hashed to). For example, for the graph in Figure 1, record-vertex r_2 sends its value to b_1 and b_2.

SS2: Bucket-vertices receive the contents of all the records hashed to them, after which records at each bucket-vertex are compared using IMM. The result is a set of merged records or *partial-entities* at each bucket-vertex.

A bucket-vertex randomly selects one of the records in each partial-entity (merged record) as a *central record* for that partial-entity. Next the bucket-vertex sends a *graph-mutation* request so

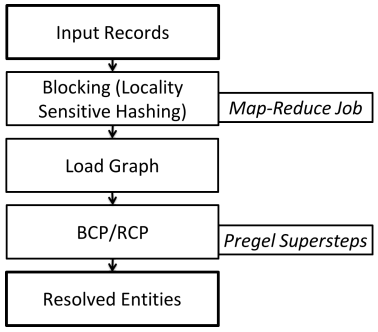

Figure 2: Overall flow across Map-reduce and Pregel

as to create a bi-directional edge between vertices corresponding to the central record and each of the remaining records of the partial-entity. As a result, all the record-vertices involved in a partial-entity get *connected* to each other through the vertex corresponding to the central record.

For the example in Figure 1, the records r_1, r_2, and r_4 are merged to give a partial-entity r_{124} at bucket-vertex b_1. Assuming r_1 to be the central record, graph-mutation requests are sent by b_1 to create bi-directional edges between r_1 and r_2, and r_1 and r_4. Similarly, at bucket-vertex b_2, we get a partial-entity r_{23} where a graph-mutation request is sent to create edges between r_2 and r_3.

SS3cc: As a result of the previous superstep we know that there is a path connecting records belonging to a single entity even though they may have been part of different partial-entities resolved at different buckets. So we now find the connected components in the part of the graph consisting only of the record-vertices and the edges between them, ignoring the bucket-vertices and their edges. Note that it may take more than one superstep to find connected-components in this graph. Finding connected-components in a graph using the Pregel model is straightforward, so we omit its details for brevity.

Finally every record-vertex gets a connected-component id which corresponds to the entity it is resolved to. For the example in Figure 1, we will get one connected-component containing all the 4 records r_1, r_2, r_3, and r_4. It remains a simple matter to exchange component information between record-vertices to complete the resolution process: Record-vertices mutate the graph to create entity-vertices corresponding to their component-id along with edges to these; the platform ensures that only one instance of each distinct entity-vertex is created, with the result that all records for that entity are now connected to it. Records send themselves to their entity-vertex where final merging takes place to produce a resolved entity.

While the above approach using both MR and Pregel avoids shipping the records mapped to singleton buckets, this approach potentially suffers because of load imbalance: As we shall see in the next section, LSH naturally results in buckets of widely varying size. As a result, some bucket-vertices have to perform heavy IMM computations, which are of quadratic time complexity, whereas others are lightly loaded. We shall see that even with careful distribution of bucket-vertices to processors, this still results in significant load-imbalance, or *skew*, causing inefficiency.

2.2 Record-centric Approach in Pregel (RCP)

The motivation for record-centric parallelization (RCP) is to overcome the problems of skew by distributing the IMM computations for records mapped to the same bucket *back* to the record-vertices themselves. The load for large IMM jobs at bucket-vertices is thus further parallelized. Record-vertices end up with work assigned to them from at most b buckets where they are mapped to; as a result the computations are better balanced even when record-vertices are randomly distributed across processors. Of course, the cost for such further re-distribution of load is increased communication cost. (A detailed cost analysis of both approaches is presented in the next section.)

RCP comprises of seven supersteps using the graph-parallel Pregel paradigm as explained below and summarised in Figure 3. Note however, that this sequence of seven supersteps will run iteratively many times until all vertices halt and no further messages are generated. As before, we assume that the initial LSH phase is performed using MR to instatiate a graph comprising of *all* records but only non-singleton buckets.

SS1: Every bucket-vertex sends messages to the record-vertices that are hashed to it in order to *schedule* their comparisons as per the IMM algorithm: For each pair of record-ids $\{r_i, r_j\}$ from the set of record-ids hashed to the bucket-vertex, a message is sent to one of the two record-vertices: Say $\{r_j\}$ is sent to r_i if $i < j$, otherwise the message $\{r_i\}$ is sent to r_j. After sending all such messages, the bucket-vertex votes to halt. Also, a record-vertex does nothing in this superstep, and votes to halt. (Unless it is a lone record that is not present in any non-singleton bucket, it will get woken up via messages in the next step; in general, vertices perform their work in a superstep and vote to halt, only to be woken up via messages later.)

If the outgoing edge-list of a bucket-vertex consists of k records $\{r_1, r_2, ..., r_k\}$, then r_1 will be sent $\{r_2, ..., r_k\}$ messages, r_2 will be sent $\{r_3, ...,r_k\}$ messages, and so on. This message-sending scheme ensures that if a pair of records co-exists in more than one bucket, then the same record-vertex (the record-vertex with lower id) will receive the messages from all such buckets. For the graph in Figure 1, bucket-vertex b_1 has $\{r_1, r_2, r_4\}$ in its neighborhood and it sends messages to r_1: $\{r_2, r_4\}$, and to r_2: $\{r_4\}$. Similarly, b_2 sends a message r_3 to r_2.

SS2: Before this superstep, every record-vertex is inactive, and gets activated when it receives IDs of record-vertices that it needs to be compared with. A record-vertex now sends its *value* (containing the full record) to the record-vertices whose IDs were received in messages.

Note that if two record-ids r_i and r_j (with $i < j$) co-occur in edge-lists of k bucket-vertices, r_i will receive k messages (one from each of the k bucket-vertices), all containing the record-id r_j. The record-vertex r_i will send its value to r_j only once. For the graph in Figure 1, record-vertex r_1 sends its value to r_2 and r_4 based on the messages received from b_1. Similarly, r_2 sends its value to r_3 and r_4, based on the messages received from bucket-vertices b_2 and b_1, respectively.

SS3: Now actual comparisons between records takes place via the match function. A record-vertex r receives messages containing the values of the record-vertices it has to be compared with. Note that the message sending scheme in SS2 ensures that even if a pair of record-ids co-occur in more than one bucket they get compared only once.

If the value of the record-vertex r matches the value of an incoming message r', a message $\{r, r'\}$ containing the IDs of the two matched vertices is sent to r and r'. For the graph in Figure 1, in SS3, record-vertex r_2 receives the value of r_1, r_3 receives the value of r_2, and r_4 receives the values of r_1 and r_2. At vertex r_2, values of r_1 and r_2 are compared and are found to be matching, and the message $\{r_1, r_2\}$ (containing only the IDs of the two vertices) is sent by r_2 to both r_1 and r_2. Similarly the message $\{r_2, r_3\}$ is generated at r_3, and is sent to both r_2 and r_3. The messages $\{r_1,$

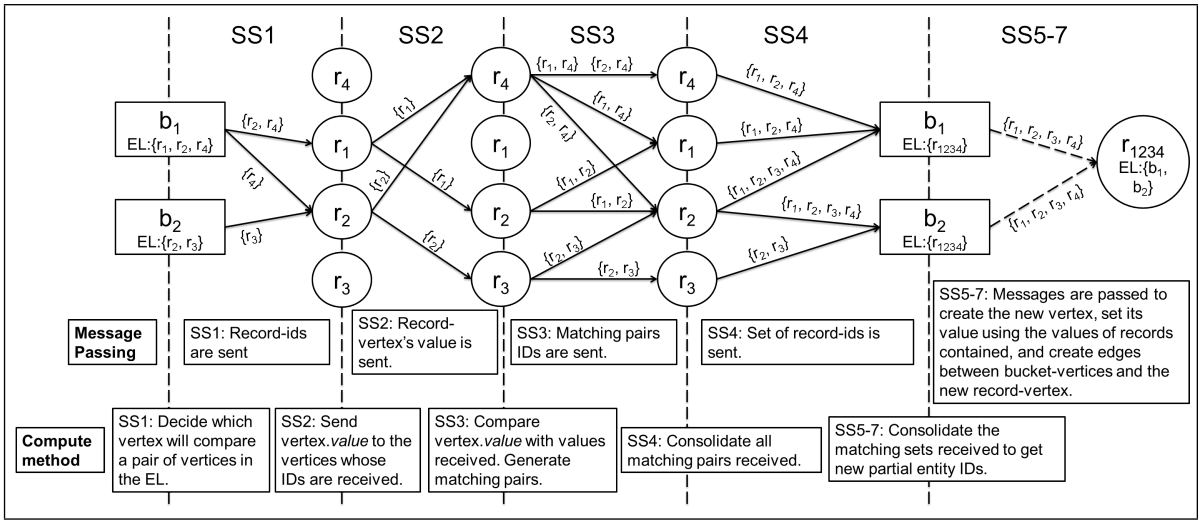

Figure 3: RCP: Supersteps SS1 to SS7 for the example in Figure 1. Here, EL stands for outgoing edge-list.

$r_4\}$ and $\{r_2, r_4\}$ are generated at record-vertex r_4. The message $\{r_1, r_4\}$ is sent to r_1 and r_4, and the message $\{r_2, r_4\}$ is sent to r_2 and r_4.

SS4: If a record r_i matched with any record r_j in the previous superstep, it will receive a message $\{r_i, r_j\}$. If r_i matched m records in the previous superstep, it receives m messages in this superstep: one from each of the m matches. Since r_i matches all the m records, all the $m + 1$ records (including r_i) belong to the same entity. The record-vertex consolidates all the pairs of IDs received as messages into a set containing the $m + 1$ IDs all belonging to the same entity. This set of IDs is sent to all the bucket-vertices in the outgoing edge-list of the record-vertex r_i.

For the graph in Figure 1, r_1 receives $\{r_1, r_2\}$ and $\{r_1, r_4\}$, and consolidates them to create a message $\{r_1, r_2, r_4\}$ which is sent to b_1. Record-vertex r_2 consolidates $\{r_1, r_2\}$, $\{r_2, r_3\}$ and $\{r_2, r_4\}$ to get $\{r_1, r_2, r_3, r_4\}$ which is sent to both b_1 and b_2. Similarly, r_3 sends $\{r_2, r_3\}$ to b_2, and r_4 sends $\{r_1, r_2, r_4\}$ to b_1.

SS5: Similar to the previous superstep, where a record-vertex consolidates the matching information about itself, a bucket-vertex consolidates all the sets received as messages and creates new record-ids as follows: If any two sets s_i and s_j received by a bucket-vertex have an element (a record-id) in common, i.e. $s_i \cap s_j \neq \emptyset$, all the record-ids in the two sets belong to the same entity. A new set $s_{ij} = s_i \cup s_j$ is created, and the sets s_i and s_j are deleted. This is done iteratively till all the sets remaining are disjoint.

Now new record-vertex needs to be created for each of the disjoint sets, so the bucket-vertex sends a graph-mutation request to create these new vertices, which we shall call *partial-entity-vertices*. Mutation requests are also sent to create bi-directional edges between the partial-entity-vertex and the bucket-vertex. The IDs of these vertices are based on the final consolidated sets s_{ij}, so if the same partial-entity-vertex is created by multiple buckets, this does not lead to duplicate vertices. The bucket-vertices also send a message to each record-vertex they are connected with, informing the record-vertices about their corresponding partial-entity-id.

For the graph in Figure 1, bucket-vertex b_1 receives 3 messages $\{r_1, r_2, r_4\}$, $\{r_1, r_2, r_3, r_4\}$ and $\{r_1, r_2, r_4\}$ which are consolidated to get a set $\{r_1, r_2, r_3, r_4\}$. Bucket-vertex b_1 sends a message to create a partial-entity-vertex with id r_{1234}. Similarly, bucket-vertex b_2 receives 2 messages $\{r_1, r_2, r_3, r_4\}$ and $\{r_2, r_3\}$ which are consolidated to get $\{r_1, r_2, r_3, r_4\}$, and a message is sent by b_2 to create a new vertex r_{1234}. Here, both the bucket-vertices b_1 and b_2 ask for the creation of the same vertex since both have computed the same consolidated set: $\{r_1, r_2, r_3, r_4\}$. Also, b_1 sends a message to create a bi-directional edge between b_1 and r_{1234}. Similarly, b_2 sends a message to create a bi-directional edge between b_2 and r_{1234}. Also, b_1 and b_2 send a message containing the id r_{1234} to all their record-vertices.

SS6: New partial-entity-vertices get created before start of this superstep. A record-vertex r receives messages containing the id of a new partial-entity-vertex r'. The record-vertex r sends its value (i.e., record content) and its outgoing edge-list as a message of the form $\{v_i, e_i\}$ (where v_i is the value of vertex r_i and e_i its outgoing edge-list) to r'. For the graph in Figure 1, the record-vertices r_1, r_2, r_3 and r_4 send $\{v_1, e_1\}$, $\{v_2, e_2\}$, $\{v_3, e_3\}$, and $\{v_4, e_4\}$ respectively to r_{1234}.

SS7: In this superstep, a partial-entity-vertex r receives messages of the form $\{v_i, e_i\}$. The value (i.e., content) of r is initialised by merging all the record content values v_is received. The vertex r also sends messages to create outgoing edges with every bucket-vertex whose ID is present in the outgoing edge-lists e_is received. For every bucket-vertex b_i, to which a partial-entity-vertex p_i is added, p_i needs to be compared with the other records and partial-entities in b_i's edge-list. So a message is also sent to these bucket-vertices to activate them before the beginning of next iteration. The partial-entity-vertices are treated like record-vertices for next iteration of supersteps SS1 to SS7.

Finally, each partial-entity-vertex r sends graph-mutation requests to delete every vertex r_i (corresponding to $\{v_i, e_i\}$) that led to r's creation (also for deleting all incoming and outgoing edges of r_i). For the graph in Figure 1, r_{1234} receives the values of record-vertices r_1, r_2, r_3, and r_4 as messages, and uses them to update its value. Also, messages are sent to create bi-directional edges between r_{1234} and b_1, and r_{1234} and b_2. Messages are also sent to delete r_1, r_2, r_3, and r_4, and activate b_1 and b_2.

Iterations in RCP: In the first iteration of the algorithm, i.e., at the beginning of SS1, all bucket-vertices are active. In the beginning (i.e., SS1) of each subsequent iteration, only those bucket-vertices are active that receive messages from SS7 of the previous iteration. Iterations continue until no more messages are generated.

A bucket-vertex can have both old and new record-ids in its out-

going edge-list at the end of any iteration of supersteps SS1 to SS7. Record-id pairs for a bucket-vertex which have already been compared need not be compared again. To avoid such comparisons, a set P is maintained for each bucket-vertex which contains the pairs which have already been compared in previous iterations. For example, suppose a bucket b has 4 records $\{r_1, r_2, r_3, r_4\}$ in its outgoing edge-list. After the first iteration of supersteps SS1 to SS7, suppose r_1 and r_2 get merged to form a new record r_{12}. Then the outgoing edge-list of b will be $\{r_{12}, r_3, r_4\}$, and the set $P = \{\{r_1, r_2\}, \{r_1, r_3\}, \{r_1, r_4\}, \{r_2, r_4\}, \{r_3, r_4\}\}$. So, in the next iteration only the following record pairs need to be compared: $\{\{r_{12}, r_3\}, \{r_{12}, r_4\}\}$. For the graph in Figure 1, buckets b_1 and b_2 will have only one record-id $\{r_{1234}\}$ in their respective outgoing edge-lists. So, no further comparisons are done and no further messages are sent, terminating the algorithm.

3. COMPARISON OF BCP AND RCP

Some of the notations used for the analysis in this section have been presented in Table 1. To compare the parallel execution times of the two approaches presented in Section 2 we first note that total execution time comprises of (a) total computation cost (T_n) and (b) total communication cost (T_c) [10]. In an ideal situation, the parallel computation time on p processors should be $T_n(p) = T_n/p$. Further, assuming a fully interconnected inter-processor network, communication between two disjoint pairs of processors can take place in parallel; if all communication is parallel in this manner then the parallel communication time on p processors should be $T_c(p) = T_c/p$.

Therefore, the ideal parallel execution time $T(p)$ should be the sum of $T_n(p)$ and $T_c(p)$. However, this is not the case in practice. First, computation load is often unevenly distributed across processors; say the most heavily loaded processor having s_1 times more work than the average processor, which we refer to as *computation skew*. For example, in case of LSH, commonly used words lead to high textual similarity between a lot of records, and all these records may get hashed to the same bucket. So some buckets end up with a very large number of records. Similarly, communication need not be evenly balanced either: For example, a source processor sending a message to all other processors takes $p - 1$ steps in spite of the fact that each receiving processor gets only a single message; the source processor becomes the bottleneck. *Communication skew* s_2 is similarly defined as the ratio between the heaviest and average communication time expended by processors. Taking skew into account, we note that the parallel execution time on p processors satisfies:

$$T(p) = (s_1 T_n + s_2 T_c)/p \quad (1)$$

In the context of entity resolution via BCP and RCP, messages between processors are of two types: a) messages containing only vertex-ids, b) messages containing an entire record. In our analysis, we assume that latter dominates and so in our analysis we ignore messages carrying only ids. Also, we ignore the cost of sending the values of record-vertices to the entity-vertices in the final merging process (when values of entity-vertices are updated) in both BCP and RCP, assuming it to be of the same order in both cases. Further, assuming uniform distribution of vertices across processors, if a message has to be sent from one vertex to another, the probability that the message will be sent to another processor is $(p-1)/p$. For large enough p we can assume that the message is almost always sent to another processor.

We assume that computation cost is directly proportional to the number of pairs of records to be compared. Therefore, total computation cost $T_n = w.\alpha$, assuming α pairs of records were compared, and w is the time taken for comparison of a pair of records. Similarly, communication cost is assumed to be directly proportional to the number of record-carrying messages exchanged between processors. Therefore, $T_c = R.\beta$, assuming β messages were exchanged between various processors, and R is the time taken for communication of one record between a pair of processors. Using (1) the parallel execution cost with p processors is as in (2) below, using which we proceed to compare the BCP and RCP approaches.

$$T(p) = (s_1.w.\alpha + s_2.R.\beta)/p \quad (2)$$

Parameter	Description
n	Total number of records
R	Cost to send a record to another processor
w	Cost of one comparison using *Match* function
b	Number of buckets each record is hashed to
t	Total number of pairs across all buckets
u	Number of unique pairs of records
f	Replication factor $(t-u)/t$
N	Maximum number of records in a bucket
d	Maximum number of records in an entity
p	Total number of processors

Table 1: Parameters used

3.1 BCP: Parallel Analysis

The BCP approach distributes LSH buckets across processors and runs IMM within each bucket. If a bucket has x records, in the best case, IMM performs $x - 1$ comparisons: This happens when all the records belong to the same entity and each successive match operation succeeds. For example, if there are 4 records d_1, d_2, d_3, and d_4 in a bucket, then there will be a minimum of 3 match operations on (d_1, d_2), (d_{12}, d_3), and (d_{123}, d_4). However, according to [4] the worst case computation cost of IMM on a block with x records and x' entities is $(x-1)^2 - (x'-1)(x'-2)/2$. This will be maximum when $x' = 1$. Therefore, in the worst case a bucket with x records will execute $(x-1)^2$ comparisons.

For the average-case analysis, we assume $\binom{x}{2}$ comparisons in a bucket of size x, i.e., the number of pairs in the block. So, the total number of comparisons is just t, the number of pairs of records across all blocks, i.e., $\alpha_{BCP} = t$.

The maximum total communication is $n.b$ since every record is sent to at most b buckets. However, a record need not be sent to a singleton bucket containing only itself. If k_1 is the expected fraction of non-singleton buckets per record, with $0 < k_1 \leqslant 1$, the total communication cost for BCP is $\beta_{BCP} = k_1.n.b$. Using the above in (2) the total computation and communication costs for BCP are summarized in Table 2.

3.2 RCP: Parallel Analysis

In RCP a bucket with x records schedules $\binom{x}{2}$ comparisons that are executed at processors holding records, rather than at the processors holding the bucket itself. Further, unlike BCP, RCP involves multiple iterations: Nevertheless, RCP will involve at most $d - 1$ iterations, if d is the maximum number of records an entity can have.

For an average-case analysis we assume that if there are x_i records in a bucket in the ith iteration, then $x_{i+1} = \lceil x_i/2 \rceil$. Consequently, if there are t_i comparisons in the ith iteration, $t_{i+1} \approx t_i/4$. Therefore, the total number of comparisons across $d-1$ iterations will be $[t + t/4 + t/4^2 + ... + t/4^{(d-2)}] = \frac{4}{3}t(1 - 4^{-(d-1)})$.

Further, in RCP, a pair of records is compared only once (see SS3 in Section 2.2) even though it may occur in multiple buckets,

Approach	pT(p)
BCP	$s_1.w.t + s_2.k_1.R.n.b$
RCP	$s'_1.(1-f).k_2.w.t + s'_2.(1-f).k_2.R.t$

Table 2: Average-case parallel execution times

with replication factor f (see Table 1). Therefore, the total number of comparisons (in first iteration) is $(1-f)t$ rather than t. As a result, $\alpha_{RCP} = (1-f).k_2.t$, where $k_2 = \frac{4}{3}(1-4^{-(d-1)})$.

Further, for each pair of records that need to be compared, one of the records in the pair is sent to the other's processor. Therefore, the number of messages is same as the number of comparisons, so $\beta_{RCP} = (1-f).k_2.t$. Using the above in (2) the parallel execution time for RCP is also summarized in Table 2.

3.3 Skew Analysis and Inferences

Computation skew s_1 and communication skew s_2 depend on the distribution of records to the buckets. Let the number of buckets having x records be $f(x)$. We analyze the case when $f(x)$ follows a power-law distribution (which is what we observed empirically as shown in the next section), i.e., $f(x) = c/x^r$, where c and r are positive constants.

Further, $\binom{x}{2} \geqslant x$, for $x \geqslant 3$; and $\binom{x}{2}.f(x) \geqslant x.f(x)$, for $f(x) \geqslant 0$. Therefore, the total number of pairs $t = \sum_{x=2}^{N} \binom{x}{2}.f(x)$ is much more than $nb = \sum_{x=1}^{N} xf(x)$, the total number of records in all buckets. As a result, we can assume that $t >> n.b$. So when the computation and communication skews for BCP and RCP are comparable, i.e., $s_1 = s'_1$, and $s_2 = s'_2$, and the replication factor f is small, we observe from Table 2 that the computation cost of both approaches is of the same order, whereas the communication cost of BCP ($s_2.k_1.R.n.b$) is clearly lower than that of RCP ($s'_2.k_2.R.t$). As a results, it appears that BCP will perform much better than RCP.

However, in most of the situations, computation skew s_1 of BCP, will be so high that it will become the dominant component of the parallel execution cost. To explain this, let us compare the effect of a large bucket in the two approaches. In BCP, a large bucket with size x, will result in $\binom{x}{2}$ computations at the processor responsible for this bucket; this processor can become slow and can increase the parallel execution cost. However, in RCP, a large bucket results in less computation skew, since the $\binom{x}{2}$ comparisons are distributed to $x-1$ record-vertices that, on the average, will reside on different processors for large enough p. Thus the computation load gets distributed.

Further, even though the large bucket causes $\binom{x}{2}$ messages to schedule comparisons across processors in case of RCP, these messages only contain the IDs of the record-vertices and as we have assumed earlier, this cost is small enough to be ignored. With this assumption, the number of record-bearing communications arising from a bucket of size x are those where the records send themselves to each other, i.e., at most $x-1$ messages for one of the records in the bucket and $x/2$ on the average. This is comparable to the communications required in BCP where x records need to be received at the bucket of size x. Therefore, the communication skews in both BCP and RCP are comparable whereas the computation skew caused by the largest bucket in case of RCP is likely to be less than the computation skew caused in case of BCP. In the next section we demonstrate this empirically with realistic data.

3.4 Empirical Comparison of BCP and RCP

We generated synthetic data of 1.2 million 'residents' (n = 1.2M) as follows: We began with 100,000 seed records, where a seed record has all the information about an entity. For each such entity, a maximum of 5 records ($d = 5$) are created corresponding to the following 5 domains: 'Election Card', 'Income-Tax Card', 'Driving-Licence', 'Bank Account Details', and 'Phone Connections'. An entity can have a maximum of one record per domain. We control the creation of a record of a particular type for an entity using a random-number generator. To create a record from the seed record, values of some of the attributes of the person are omitted. For example, e-mail id of a person may be present in 2 of his records and absent in others. The values of the attributes of an entity across records are varied by using different methods. For example, the address of the different records for the same entity need not be same. Variation in values of attributes for different records for an entity are inserted by introducing typographical errors, swapping the first and last name, or omitting the middle name. To add further ambiguity after creation of the records for an entity, additional records which share considerable textual similarity with each other are generated for related entities, such as parent, child, spouse, neighbour, etc.

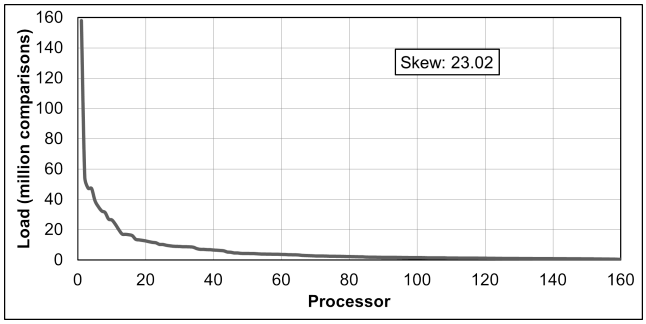

Figure 4: Average-case Computation Load for BCP

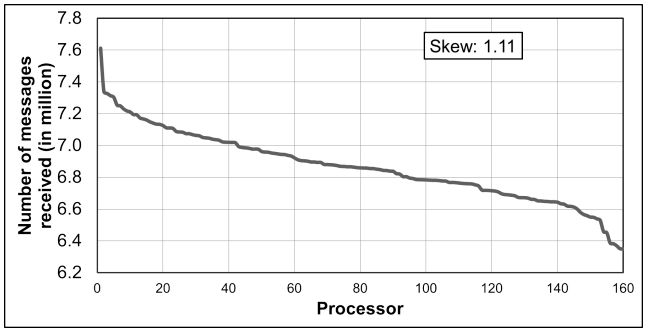

Figure 5: Average-Case Communication Load for BCP

We applied 90 hash functions on each record and used $a = 3$ and $b = 30$ in the LSH formulation (refer Section 2.1.1). We get $\approx$ 17.29M buckets, 59.34% of the buckets (10.26M) were singletons, 99.89% (17.27M) of the buckets had $\leqslant 30$ records, 0.02% (696) buckets had > 100 records, and 0.00092% (163) buckets had $> 1,000$ records. (With this generated data, the value of r in power law distribution of records to buckets, i.e., $f(x)$ as defined earlier in Section 3.3 was found to be ≈ 2.8.) The distribution of $f(x)$ vs x is shown in Figure 7. The 163 (0.00092%) large buckets are the bottlenecks in terms of the execution-time taken by the processors which have them. The size of the largest and second largest buckets are $17,668$ and $9,662$, respectively.

To estimate the values of skew factors, s_1, s_2, s'_1, and s'_2, we assume the number of processors $p = 160$ (corresponding to our

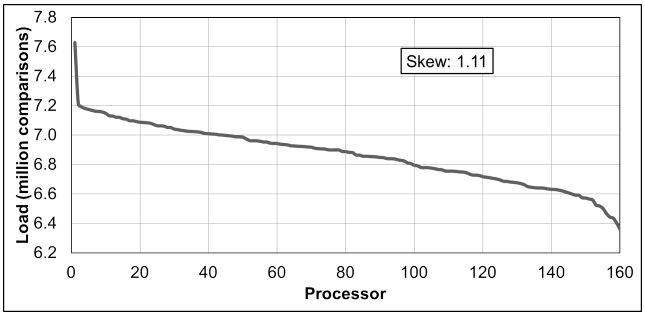

Figure 6: Average-Case Computation Load for RCP

Approach	Comparisons (millions)			Communications (millions)		
	Max.	Avg.	s_1	Max.	Avg.	s_2
BCP	158.039	6.864	**23.02**	0.171	0.154	1.11
RCP	7.629	6.864	1.11	7.612	6.864	1.11

Table 3: Skew factors for BCP and RCP (average-case)

physical cluster of five 4 CPU nodes with 4 cores and two virtual processors per core). The records and buckets (1.2M+17.29M vertices) were randomly distributed to these 160 processors. Assuming $w = R = 1$, the computation and communication loads per processor are shown in Figures 4, 5, and 6.

The skew factors as estimated from the above distributions are shown in the Table 3: It can be observed that the computation skew s_2 in BCP is 23.02, whereas the communication skew of BCP s_1 and the computation and communication skews s'_1 and s'_2 are almost the same ($= 1.11$). Based on the formulations in Table 2, it is clear that assuming $w = R = 1$, replication factor $f = 0$, $d = 5$ so that $k_2 = \frac{4}{3}(1 - 4^{-(d-1)}) \approx \frac{4}{3}$, $k_1 = 1$, the parallel execution-time of BCP,

$$\begin{aligned} p.T^b(p) &= s_1.w.t + s_2.k_1.R.n.b \\ &= 23.02 \times t + 1.11 \times n.b \\ &\geqslant 23.02 \times t \end{aligned}$$

will be much larger than that of RCP,

$$\begin{aligned} p.T^r(p) &= s'_1.(1-f).k_2.w.t + s'_2.(1-f).k_2.R.t \\ &= 2 \times 1.11 \times \frac{4}{3}t \\ &= 2.96 \times t \end{aligned}$$

since s_1 is much higher than both s'_1 and s'_2.

It may appear surprising that computation skew in BCP is so much higher that communication skew; after all the number of pairs at a bucket (which determines computation cost) is at most the square of the bucket's size (which determines communication cost). Computation skew, on the other hand, is much worse than the square of communication skew.

However, consider the following example taken from our synthetic data: Suppose we have 100 processors, and 150 buckets have size $\geqslant 1000$. Assuming uniform distribution, each processor gets 0.16M (16M total buckets / 100 processors). Suppose the sizes of the largest buckets b_1 and b_2 at processors P_1 and P_2 be $18,000$ and 300 respectively. There are 0.16M - 1 other buckets each at P_1 and P_2. Assuming uniform distribution of load on these two processors based on the 0.16M - 1 buckets (not considering the two buckets b_1 and b_2), the ratio of execution-times on P_1 and P_2 would be close to 1. Suppose the execution-time of these 0.16M - 1 buckets (all buckets except b_1 in case of P_1 and b_2 in case of P_2) is T on both the processors. Then the execution-time T_1 on processor P_1 is $T + \binom{18,000}{2} = T + 161,991,000$, and T_2 on processor P_2 is $T + \binom{300}{2} = T + 44,850$. To compute T, suppose the 60% of the singleton buckets get uniformly distributed. So, 96k (60% of 0.16M) of the buckets at P_1 and P_2 each have no contribution to T. Suppose the remaining 64k buckets get 30 records each (which gives an approximate upper bound on T assuming 99.9% of the buckets have less than 30 records). Then, $T = 64,000 \times \binom{30}{2}$, i.e., $27,840,000$, so that $T_1 = 189,831,000$ and $T_2 = 27,884,850$. So the computation skew $\frac{T_1}{T_2} \approx 6.8$. Empirically, even with uniform distribution of buckets across processors, we still find $s^b_1 \approx 20$.

Thus, even if the distribution of 150 large-sized buckets is uniform across the 100 processors, still any processor getting two large-sized buckets results in significant computation skew for BCP. On the other hand, since RCP distributes computations by records, with the number of computations at a record directly proportional to the sum of its bucket *sizes* (rather than the square of bucket size as in BCP), the resulting computation skew is relatively small.

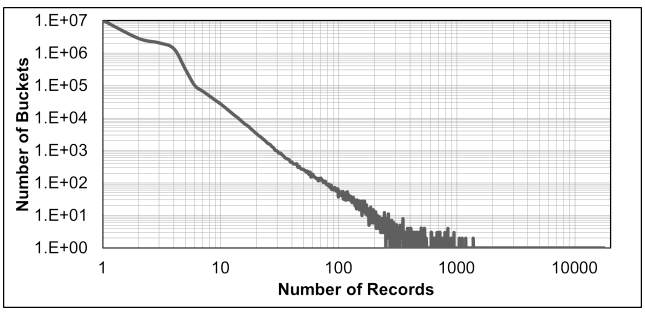

Figure 7: Number of Buckets ($f(x)$) vs. Number of Records (x)

4. RELATED WORK

Parallel entity resolution (ER) using three distributed computing paradigms has been described in [23]: distributed key-value stores, MR, and bulk synchronous parallellism. Their strategy is to perform pair-wise comparisons followed by connected components. The bulk synchronous parallel (BSP) algorithm in [23] is related to the graph-parallel model. However, they do not use any acceleration strategy (such as LSH) to group candidate pairs; neither do they deal with the issue of load-imbalance (skew).

Other parallel implementations for entity resolution have been proposed in [3, 11, 7, 13], but none of these address the issue of skew. The D-Swoosh family of algorithms in [3] implements R-Swoosh based IMM [4] by distributing the task of comparing records to multiple processors by using *scope* and *responsibility* functions, where the *scope* function decides the processors where a record will be sent to, and the *responsibility* function decides which processor will compare a given pair of records. In one particular domain-dependent strategy called *Groups*, records are assigned to groups, and two records are compared only when they belong to the same group. This is similar to the idea of buckets in the context of BCP algorithm. However, in *Groups*-based D-Swoosh, each group is assigned to one processor, whereas in BCP, multiple buckets end up being assigned to the same processor.

The Dedoop tool described in [14] borrowing ideas from another work by the same authors in [15], is a MR-based entity resolution tool that includes different blocking techniques, provides strategies to achieve balanced workloads, and also provides redundant-free

comparisons when multiple blocking keys are used. Load balancing in Dedoop is achieved through an additional MR step before the actual matching job to analyze the input data and create a block distribution matrix (BDM). This BDM is then used by, for example, the BlockSplit strategy to split the match tasks for large-sized blocks into smaller match tasks for sub-blocks, ensuring all comparisons for the large-sized block gets done. Dedoop considers only pair-wise comparisons and does not perform IMM for which their load-balancing strategy may not work. Similar to our RCP approach, where a pair of records co-occurring in multiple buckets is compared only once, Dedoop also has provision for avoiding redundant comparisons when multiple blocking keys are used by comparing record-pairs only for their smallest common blocking key.

The stragglers problem in the context of MR, in general, has been discussed in [17, 21, 16]. The basic idea to avoid load imbalance, due to large number of values for a particular reduce-key, is to somehow split the keys with large loads into multiple sub-keys that can then be assigned to different reducers. Using such solutions in the context of Iterative Match-Merge for the values (records) at a reduce-key is not straight-forward and can be a direction for future research.

In [17], it is shown that when the distribution $f(x)$ of the number of reduce-keys receiving x values follows a Zipf distribution, i.e $f(x) \propto \frac{1}{x}$, and each reduce task performs $O(x)$ work, then the maximum speedup is around 14. This suggests that the situation will be worse in our case when the amount of work done for the task with x elements is $O(x^2)$, thereby justifying our RCP approach further.

Various grouping techniques such as sorted-neighborhood indexing and Q-gram-based indexing have been proposed to reduce the set of candidate pairs, as surveyed in [6]. Locality Sensitive Hashing (LSH) for entity resolution has been discussed in [12].

5. CONCLUSIONS AND LEARNINGS

We set out to develop a parallel implementation of entity resolution so as to handle with cases involving billions of records and hundreds of millions of entities. The bucket-parallel approach, which is natural using MR, results in significant skew. The record-parallel approach emerged as a natural alternative and turned out to have better load-balancing properties especially in the presence of severe skew, which arises naturally in hashing where some buckets corresponding, say, to very common names, end up with a large number of records.

Many problems involving evaluating pair-wise similarity of large collections of objects can be efficiently accelerated using probabilistic hashing techniques such as LSH, in much the same manner as we have accelerated IMM-based entity resolution. Clustering using canopies, similarity joins, feature-based object search as well as duplicate-detection in object databases (such as for biometric applications) are some such examples. In all such cases parallel implementation can be done bucket-wise or record-wise, and the advantages of the record-parallel approach can be derived whenever hashing is expected to result in skew due to some features being much more heavily shared across objects than others.

Last but not least, note that even in the BCP approach, we avoid sending records to singleton buckets; i.e., we first form buckets using just record-ids and then send the more voluminous actual records only to non-singleton buckets. Since we find that over 60% buckets are singletons in our sample data, this optimisation is fruitful even though it costs an additional communication step. Upon reflection we realised that other scenarios present a similar opportunity for optimisation: Consider a multi-way join implemented in MR, but where some attributes are large objects such as images or videos. Even for normal joins (on say, object-id, i.e., not similarity joins), traditional MR implementations would ship entire records to reducers, including those that never match with others and are therefore absent in the final result. In such cases, eliminating singletons via an initial phase is an important optimisation that applies both for MR as well as graph-based parallelization.

6. ACKNOWLEDGEMENT

The authors would like to thank Jeffrey D. Ullman for suggesting that while the RCP approach may have some advantages, the BCP approach would have lower communication costs; so a thorough analysis is in fact required.

7. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pages 459–468, 2006.

[2] C. Avery. Giraph: Large-scale graph processing infrastructure on hadoop. *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[3] O. Benjelloun, H. Garcia-Molina, H. Gong, H. Kawai, T. Larson, D. Menestrina, and S. Thavisomboon. D-swoosh: A family of algorithms for generic, distributed entity resolution. In *Distributed Computing Systems, 2007. ICDCS '07. 27th International Conference on*, pages 37–37, 2007.

[4] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *The VLDB Journal*, 18(1):255–276, 2009.

[5] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[6] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *Knowledge and Data Engineering, IEEE Transactions on*, 24(9):1537–1555, 2012.

[7] P. Christen, T. Churches, and M. Hegland. Febrl âĂŞ a parallel open source data linkage system. In H. Dai, R. Srikant, and C. Zhang, editors, *Advances in Knowledge Discovery and Data Mining*, volume 3056 of *Lecture Notes in Computer Science*, pages 638–647. Springer Berlin Heidelberg, 2004.

[8] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: Scalable online collaborative filtering. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 271–280, New York, NY, USA, 2007. ACM.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, Jan. 2008.

[10] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors. Vol. 1: General Techniques and Regular Problems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[11] H.-s. Kim and D. Lee. Parallel linkage. In *Proceedings of the Sixteenth ACM Conference on Conference on Information and Knowledge Management*, CIKM '07, pages 283–292, New York, NY, USA, 2007. ACM.

[12] H.-s. Kim and D. Lee. Harra: Fast iterative hashed record linkage for large-scale data collections. In *Proceedings of the*

13th International Conference on Extending Database Technology, EDBT '10, pages 525–536, New York, NY, USA, 2010. ACM.

[13] L. Kolb, H. Köpcke, A. Thor, and E. Rahm. Learning-based entity resolution with mapreduce. In *Proceedings of the Third International Workshop on Cloud Data Management*, CloudDB '11, pages 1–6, New York, NY, USA, 2011. ACM.

[14] L. Kolb, A. Thor, and E. Rahm. Dedoop: efficient deduplication with hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[15] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 618–629, 2012.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. A study of skew in mapreduce applications. *Open Cirrus Summit*, 2011.

[17] J. Lin. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce. *Proceedings of 7th Workshop on Large-Scale Distributed Systems for Information Retrieval*, pages 57 – 62, 2009.

[18] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[20] A. Rajaraman and J. Ullman. *Mining of Massive Datasets*. April 2010.

[21] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 16:1–16:14, New York, NY, USA, 2012. ACM.

[22] S. Salihoglu and J. Widom. Gps: A graph processing system. *Technical Report, Stanford University*, 2013.

[23] C. Sidló, A. Garzó, A. Molnár, and A. Benczúr. Infrastructures and bounds for distributed entity resolution. In *9th International Workshop on Quality in Databases*, 2011.

[24] T. White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.