Entity Resolution with Iterative Blocking

Steven Euijong Whang, David Menestrina, Georgia Koutrika, Martin Theobald,
Hector Garcia-Molina

Computer Science Department, Stanford University
353 Serra Mall, Stanford, CA 94305, USA
{swhang, dmenest, koutrika, theobald, hector}@cs.stanford.edu

ABSTRACT

Entity Resolution (ER) is the problem of identifying which records in a database refer to the same real-world entity. An exhaustive ER process involves computing the similarities between pairs of records, which can be very expensive for large datasets. Various blocking techniques can be used to enhance the performance of ER by dividing the records into blocks in multiple ways and only comparing records within the same block. However, most blocking techniques process blocks separately and do not exploit the results of other blocks. In this paper, we propose an *iterative blocking framework* where the ER results of blocks are reflected to subsequently processed blocks. Blocks are now iteratively processed until no block contains any more matching records. Compared to simple blocking, iterative blocking may achieve higher accuracy because reflecting the ER results of blocks to other blocks may generate additional record matches. Iterative blocking may also be more efficient because processing a block now saves the processing time for other blocks. We implement a scalable iterative blocking system and demonstrate that iterative blocking can be more accurate and efficient than blocking for large datasets.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Miscellaneous—*data cleaning*; D.2.8 [**Information Storage and Retrieval**]: Information Search and Retrieval—*clustering*; H.2.8 [**Database Management**]: Database Applications—*data mining*

General Terms

Algorithms

Keywords

entity resolution, blocking, iterative blocking

1. INTRODUCTION

Entity resolution (ER) is the problem of matching records that represent the same real-world entity and then merging the matching records. ER is a well known problem that arises in many applications. For example, mailing lists may contain multiple entries representing the same physical address, but each record may be slightly different, e.g., containing different spellings or missing some information. As a second example, two companies that merge may want to combine their customer records: for a given customer that dealt with the two companies they create a composite record that combines the known information. An exhaustive ER process involves comparing all the pairs of records, which can be very expensive for large datasets.

Various blocking techniques [19, 24, 5, 16, 13, 11, 15, 2, 10] have been proposed to make ER scalable. Blocking divides the data into (possibly overlapping) blocks and only compares records within the same block, assuming that records in different blocks are unlikely to match. For example, we might partition a set of people records according to the zip codes in address fields. We then only need to compare the records with the same zip code. Since a single blocking criterion may miss matches (e.g., a person may have moved to places with different zip codes), several blocking criteria (i.e., dividing the data in several ways) are typically used to ensure that all the likely matching records are compared, improving the accuracy of the result.

Although the previous works above focus on finding the best blocking criteria, most of them assume that all the blocks are processed separately one at a time. In many cases, however, it is useful to exploit an ER result of a previously processed block. First, when two records match and merge in one block, their composite may match with records in other blocks. Second, an ER result of a block can be used to reduce the time of processing another block. That is, the same pair of records may occur in multiple blocks, so once the pair is compared in one block, we can avoid comparing it in other blocks. To address these two points, we propose an iterative blocking framework where the ER result of a block is immediately reflected to other blocks. Unlike previous blocking techniques, there is an additional stage where newly created records of a block are distributed to other blocks. Since the propagation of ER results can generate new record matches in other blocks, the entire operation becomes iterative in the sense that we are processing blocks (possibly the same block multiple times) until we arrive at a state where we cannot find any more matching records. Our work is thus focused on effectively processing the blocks given a blocking criteria.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

Record	Name	Address(zip)	Email
r	John Doe	02139	jdoe@yahoo
s	John Doe	94305	
t	J. Foe	94305	jdoe@yahoo
u	Bobbie Brown	12345	bob@google
v	Bobbie Brown	12345	bob@google

Figure 1: Customer records

Criterion	Partitions by	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	zip code	r	s, t	u, v
SC_2	1^{st} char of last name	r, s	t	u, v

Figure 2: Multiple blocking

Motivating Example: Consider the four people records shown in Figure 1, that are to be resolved. We would like to merge records that actually refer to the same person. Suppose that records r and s match with each other because their names are the same, but do not match with t because the strings differ too much. However, once r and s are merged into a new record $\langle r, s \rangle$, the combination of the address and email information of r and s may lead us to discover a new match with t, therefore yielding an initially unforeseen merge $\langle r, s, t \rangle$. Notice that, in order to find this new merge, we need to compare the merged result of r and s with all the other records again.

In reality, our dataset can be very large, and it may not be feasible to compare all pairs of the dataset. Hence, we divide the customer records in Figure 1 into blocks. We start by dividing the records by their zip codes. As a result, we only need to compare customers that are in the same geographical region. In Figure 2, the first blocking criterion SC_1 uses zip codes to divide the records. Records s and t have the same zip code and are assigned to the block 2 (denoted as $b_{1,2}$) and records u and v are assigned to $b_{1,3}$ while r is assigned to $b_{1,1}$. Since we may miss matches for people who have moved to several places with different zip codes, say we also divide the customer records according to the first characters of their last names. Hence, even if two records referring to the same person have different zip codes, we will have a better chance of comparing them because their last names might be similar. In Figure 2, the matching records r and s can be compared because, although they have different zip codes, they have the same last name. After processing all the blocks, the final result of blocking is $\{\langle r, s \rangle, t, \langle u, v \rangle\}$.

Although the blocking of Figure 2 reduces the number of records to compare, it misses the iterative match between $\langle r, s \rangle$ and t. Iterative blocking can find this match by distributing the newly created $\langle r, s \rangle$ (found in block $b_{2,1}$) to the other blocks. Assuming that $\langle r, s \rangle$ contains the zip codes of both r and s (i.e., 02139 and 94305), $\langle r, s \rangle$ is then assigned to both blocks of SC_1. In $b_{1,2}$, $\langle r, s \rangle$ can then be compared with t, generating the record $\langle r, s, t \rangle$. Eventually, the final iterative blocking solution becomes $\{\langle r, s, t \rangle, \langle u, v \rangle\}$ (see Section 2.4.3 for details). Thus, the iterative blocking framework helps find more record matches compared to simple blocking.

Iterative blocking also provides fast convergence. For example, once the records u and v are merged in $b_{1,3}$, they do not have to be compared in $b_{2,3}$. While the blocks in Figure 2 are too small to show any improvements in runtime, we will later demonstrate in our experiments that iterative blocking can actually run faster than blocking for large datasets when the runtime savings exceed the overhead of iterative blocking. Intuitively, the more work we do for each block, the runtime savings for other blocks become significant.

Our example has illustrated the two potential advantages of iterative blocking. The first is improved accuracy, as each iteration may find additional record matches. The second potential advantage is improved runtime performance. By using the ER result of a previous block, we can reduce the time to process other blocks, by avoiding comparisons that were already made.

In summary, we make the following contributions.

- We formalize the iterative blocking model (I-BlockER). Unlike blocking, the ER results of blocks are now reflected to subsequent blocks. The blocks are iteratively processed until no blocks contain any more matching records. I-BlockER can accommodate any "core" ER algorithm that resolves records within a single block.
- We present I-BlockER algorithms for two scenarios:
  - *Lego:* An in-memory algorithm that efficiently processes blocks within memory.
  - *Duplo:* A scalable disk-based algorithm that processes blocks from the disk.
- We experimentally evaluate Lego and Duplo using actual comparison shopping data from Yahoo! Shopping and hotel information data from Yahoo! Travel. Our results show that iterative blocking can improve over blocking both in accuracy and runtime. We evaluate our algorithms using two different core ER algorithms, demonstrating the generality of our iterative blocking framework.

2. FRAMEWORK

In this section, we define the framework for iterative blocking. We first define a general model for entity resolution, enabling any core ER algorithm that resolves one block of records to fit in the framework. Next, we formalize blocking by defining single and multiple blocking criteria. Finally, we define the iterative blocking process and discuss the design choices that arise.

2.1 ER Model

Intuitively, an ER algorithm takes as input a set of records R and groups together records that represent the same real world entity. We represent the output of the ER process as a partition of the input. In our motivating example, the input was a set of records R=$\{r, s, t, u, v\}$ and the output was $\{\langle r, s, t \rangle, \langle u, v \rangle\}$, where the angle brackets denote the clusters in the partition.

Since records in an output cluster represent some real world entity, the cluster can be considered a "composite" new record. In some cases we may apply a merge function to actually generate the composite records, but in other cases ER just outputs the cluster. Even if the records are merged, the lineage of the new record will most likely identify the original records in the cluster. In our model, we do not explicitly represent the merging of records, but instead leave the ER output as a set of clusters.

Since we may want to run ER on the output of a previous resolution (as we shall see in Section 2.3), we generalize our model so that the input itself may be a partition. We refer to the original records in R as the *base records*, and say that both the ER input and output are partitions of these base records. In our example, we then view the initial input as the partition $\{\langle r\rangle, \langle s\rangle, \langle t\rangle, \langle u\rangle, \langle v\rangle\}$. If we run a second ER on the first output $\{\langle r, s, t\rangle, \langle u, v\rangle\}$, we may obtain the partition $\{\langle r, s, t, u, v\rangle\}$. In this case, $\langle r, s, t\rangle$ contains enough information to merge with $\langle u, v\rangle$.

Since the clusters in an input partition are analogous to records, we will refer to clusters of records as records. Thus, $\langle r, s, t\rangle$ is an input record to the second ER above. For simplicity, we omit the cluster brackets for singleton clusters. For example, we write $\{\langle r, s, t\rangle, \langle u\rangle\}$ as $\{\langle r, s, t\rangle, u\}$.

We assume that ER never undoes the groupings done earlier. Hence, the output partition P_o always *dominates* the input partition P_i.

Definition 2.1. *A partition P_o* dominates *another partition P_i (denoted as $P_i \leqslant P_o$) when the following condition holds:*

- *$\forall r \in P_i$, $\exists s \in P_o$ s.t. $r \subseteq s$.*

We now formally define an ER algorithm.

Definition 2.2. *We say an ER algorithm is* valid *if, given an input partition P_i of R, the algorithm returns an output partition P_o of R such that $P_i \leqslant P_o$.*

We say that two input records in P_i *match* if they represent the same entity and are placed in the same cluster in P_o.

2.2 Blocking Model

Consider an ER algorithm whose input is a set of records P_i. In general, a given $r \in P_i$ may be included with any other $s \in P_i$ in an output partition. Thus, the ER algorithm must somehow "compare" r and s to see if they belong in the same partition, for every r, s pair. This quadratic process is expensive in general.

A single *blocking criterion* is a heuristic that prunes the number of records that must be compared with r, i.e., it reduces the number of candidates that may join r in an output cluster.

Records are placed in blocks and only the records that share a block are candidates for placement in an output cluster. Let $b_{j,k}$ be a block where subscript j identifies the single blocking criterion used for record placement, and k identifies the block within the single blocking criterion. For a single blocking criterion j, function SC_j maps a record r to one or more of the available $b_{j,k}$ blocks. Two records r and s should be compared if and only if $SC_j(r) \cap SC_j(s) \neq \emptyset$. We refer to the contents of block $b_{j,k}$ by $IN(b_{j,k})$. Initially for an input partition P_i, $IN(b_{j,k}) = \{r | r \in P_i, b_{j,k} \in SC_j(r)\}$. When it is clear from context, we will refer to the contents of block $b_{j,k}$ simply by $b_{j,k}$.

A multiple blocking criteria MC uses a set of single criteria $SC_1, SC_2, \ldots, SC_N$. A record is mapped to all blocks mapped to by the individual single blocking criteria, i.e., $MC(r)= \bigcup_{j=1,2,\ldots,N} SC_j(r)$. Hence, records r and s are compared only if $MC(r) \cap MC(s) \neq \emptyset$.

Notice that $SC_j(r)$ may place r in more than one block, enabling blocks to overlap. This added flexibility is useful when a record has several values for an attribute. For example, consider a base record r with zip code 94305 and a record s with zip code 12345. Say in a first ER pass the records are merged into one record $\langle r, s\rangle$, and $\langle r, s\rangle$ contains both zip codes. Hence, in a second pass, it is natural for a zip code criterion to map $\langle r, s\rangle$ to two zip code blocks, i.e., the block for zip code 94305 and the block for zip code 12345.

Our example illustrates how many blocking criteria operate on records that are clusters of base records. That is, a blocking criterion determines the blocks by simply seeing where each base record in the cluster would be placed. We say that a multiple blocking criterion MC has the *coverage property* if all of its constituent single blocking criteria operate in this way.

Definition 2.3. *A multiple blocking criterion MC satisfies the coverage property if $\forall j = 1, 2, \ldots, N, SC_j(r) = \bigcup_{z\in r} SC_j(\langle z\rangle)$.*

2.3 Iterative Blocking Model

Given an ER algorithm and a multiple blocking criteria function, an *iterative blocking* process identifies matching records by running a core ER algorithm (which satisfies Definition 2.2) on each block and reflecting the resolution results to other blocks, possibly generating more record matches. The above process is repeated until no blocks contain any more matching records. The final "fixed-point state" produces the solution for iterative blocking.

We now formally define iterative blocking.

Definition 2.4. *(I-BlockER Model) Given a set of records R, a core entity resolution algorithm CER, and a multiple blocking criteria function MC, a* valid *iterative blocking result J = I-BlockER(R) satisfies the following conditions.*

1. *$\bigcup_{r\in J} r = R$*
2. *$\forall r, s \in J$ s.t. $r \neq s$, $r \cap s = \emptyset$*
3. *$\forall$ block $b, IN(b) = CER(IN(b))$ where $IN(b)$=$\{r | r \in J, b \in MC(r)\}$*

Conditions 1) and 2) guarantee that J partitions R, making any algorithm that satisfies the I-BlockER model also a valid ER algorithm. Condition 3) states that no records match within any block in the final state. Returning to our example in Figure 2, the iterative blocking solution J=$\{\langle r, s, t\rangle$, $\langle u, v\rangle\}$ satisfies Definition 2.4 because J is a partition of R, and applying the CER algorithm to any of the blocks (which contain either $\{\langle r, s, t\rangle\}$ or $\{\langle u, v\rangle\}$; Figure 6 in Section 2.4.3 shows the final state of the blocks for J) results in the same block. However, the blocking solution J=$\{\langle r, s\rangle$,t,$\langle u, v\rangle\}$, does *not* satisfy condition 3) because $IN(b_{1,2}) \neq$ CER$(IN(b_{1,2}))$ where $IN(b_{1,2}) = \{\langle r, s\rangle, t\}$. (We assume the coverage property holds and that $\langle r, s\rangle$ is assigned to $b_{1,2}$.)

2.4 Iterative Blocking Algorithm

Algorithm 1 shows how an iterative blocking solution can be generated to satisfy Definition 2.4. For each block, we preprocess the block and run the CER algorithm (Steps 9∼10). In Steps 13∼17, we distribute newly created records to other blocks. We repeat until no blocks have any more matching records. Finally, in Step 22, we gather the records of all the blocks to derive our final solution. We discuss the issues that arise in Algorithm 1 and propose reasonable design choices that guarantee the algorithm terminates and satisfies Definition 2.4.

```
 1: input: a partition $P_i$ of $R$ and a core entity resolution algorithm $CER$
 2: output: a partition $P_o$ of $R$ such that $P_i \leqslant P_o$
 3: for each block $b_{j,k}$ do
 4:   $IN(b_{j,k}) \leftarrow \{r | r \in P_i, b_{j,k} \in SC_j(r)\}$
 5: end for
 6: repeat
 7:   $NewRec \leftarrow$ false
 8:   for each block $b_{j,k}$ do
 9:     $R_i \leftarrow$ Preprocess $IN(b_{j,k})$ into a partition of base records
10:     $R_o \leftarrow$ CER($R_i$)
11:     if $R_o - IN(b_{j,k}) \neq \emptyset$ then
12:       $NewRec \leftarrow$ true
13:       for each $r \in R_o - IN(b_{j,k})$ do
14:         for each $b \in MC(r)$ do
15:           $IN(b) \leftarrow IN(b) \cup \{r\}$  /* Distribute $r$ to $b$ */
16:         end for
17:       end for
18:     end if
19:     $IN(b_{j,k}) \leftarrow R_o$
20:   end for
21: until $NewRec =$ false  /* No new records created */
22: return Union of all blocks
```

Algorithm 1: Iterative Blocking

2.4.1 Processing blocks

Processing the blocks in Algorithm 1 (Steps 9∼10) involves preprocessing and then running the CER algorithm. Before the CER algorithm is applied, a block needs to be preprocessed to ensure that the CER input R_i is a partition of base records. A block can contain records that overlap in base records because of the distribution stage (Steps 13∼17) where newly created records are distributed to other blocks. For example, a block that contains the record $\langle r,s \rangle$ may receive an overlapping record $\langle s,t \rangle$ from another block. Records that overlap in base records can be viewed as "conflicting" advice from different heuristics (i.e., blocking criteria). For example, suppose that one blocking criterion placed r, s in one block and t in another. In the first block, suppose that r, s were merged into $\langle r, s \rangle$. Also, suppose that the second blocking criterion placed r apart from s, t, and the latter records were merged into $\langle s, t \rangle$. Record $\langle r, s \rangle$ can be viewed as an advice that r and s match, but not with t. Similarly $\langle s, t \rangle$ implies that s and t match, but not with r. Preprocessing a block into a partition of base records is thus a process of resolving the conflicts. One may argue that the CER algorithm can simply be run on the conflicting records. However, if R_i is not a partition of base records, then the CER algorithm is not guaranteed to return a partition of base records for the output either.

There are two approaches for resolving conflicts in Step 9:

- *Unmerge Conflicting Records*: Unmerge all the conflicting records into base records and perform a union. For example, if we have the records $\{\langle r, s \rangle, \langle s, t \rangle, \langle u, v \rangle\}$, we return $\{r, s, t, \langle u, v \rangle\}$ because $\langle r, s \rangle$ and $\langle s, t \rangle$ conflict.
- *Connected Component*: Merge all the records that conflict. In the above example, we return $\{\langle r, s, t \rangle, \langle u, v \rangle\}$. The connected component operation is equivalent to a transitive closure of matching records.

The first strategy, unmerging conflicting records, is a safe way to resolve conflicts by unmerging any record that conflicts. However, unmerging records may also be inefficient because we might need to redo many redundant record comparisons as a result. In the worst case, Algorithm 1 may not terminate if the same records repeatedly merge in one block and unmerge in another. Hence, we may need to enforce certain properties on the preprocessing stage or CER algorithm to guarantee the termination of Algorithm 1 (see our extended report [22] for a full discussion on supporting the unmerge conflicting records strategy). On the other hand, the connected component strategy considers the conflicts to be a result of the "incompleteness" of the heuristics to identify more record matches and thus resolves the conflicts by preserving all the record matches that were found.

In our work, we choose the connected component strategy to resolve conflicts for two reasons. First, merging conflicting records is a common technique used to generate groups of matching records (e.g., most previous works [7, 11, 17] assume that their matching is precise and run a transitive closure on all the matching pairs of records to produce an ER result). Second, the management of merged records can be done efficiently (see Section 2.5).

2.4.2 Distributing records

Distributing newly created records of a block to other blocks (Steps 13∼17) can help find additional record matches and reduce the processing time for the other blocks. We can invoke MC to distribute the records to blocks. In Section 2.5, we will see that, if the coverage property holds for MC and the connected component strategy is used, the distribution of records does not have to be done on the blocks directly, but can be managed efficiently using a single data structure (e.g., a hash table in memory or a log structure on disk) without modifying the blocks.

2.4.3 Gathering results

Once there are no blocks that have matching records, we can union the records of all the blocks to arrive at a final solution (Step 22). In fact, if the coverage property and connected component strategy are used, we only need to union the blocks of any single blocking criterion because any of them contains all the final records in their blocks at the end of the algorithm. (See Lemma 2.7 below.)

Example. We illustrate Algorithm 1 assuming the coverage property and connected component strategy. We use our motivating example in Figure 1. The first step is to distribute all the base records into blocks as shown in Figure 3. Again, SC denotes the blocking criterion while b denotes the block. For example, the records s and t are located together in $b_{1,2}$, which is the second block of the first blocking criterion SC_1.

Criterion	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	r	s, t	u, v
SC_2	r, s	t	u, v

Figure 3: After initial distribution

Next, we start processing all the blocks. Suppose that we repeatedly process blocks in the order of increasing subscript order (i.e., $b_{1,1}$,$b_{1,2}$,…,$b_{2,3}$). The first block that has matching records is $b_{1,3}$ where we merge u and v into $\langle u, v \rangle$. Since we assume the coverage property, $\langle u, v \rangle$ is then distributed to all the blocks containing either u or v and is thus assigned to $b_{2,3}$. We next process $b_{2,1}$ which contains the matching records r and s. This time, we distribute the merged record

$\langle r, s\rangle$ to $b_{1,1}$ and $b_{1,2}$. The next block $b_{2,2}$ does not produce any record matches. Next, block $b_{2,3}$, which at this point contains $\{u, v, \langle u, v\rangle\}$, is preprocessed into $\{\langle u, v\rangle\}$. Then CER is run on $b_{2,3}$, but it does not generate any record matches. The intermediate result after the first iteration is shown in Figure 4.

Criterion	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	r, $\langle r, s\rangle$	s, t, $\langle r, s\rangle$	$\langle u, v\rangle$
SC_2	$\langle r, s\rangle$	t	$\langle u, v\rangle$

Figure 4: Intermediate result after first iteration

We now repeat the entire loop again. Block $b_{1,1}$ is preprocessed into $\{\langle r, s\rangle\}$, but does not generate any new records after the CER algorithm is applied. Block $b_{1,2}$, however, is preprocessed into $\{\langle r, s\rangle, t\}$ and generates the new record $\langle r, s, t\rangle$ by merging $\langle r, s\rangle$ and t. Record $\langle r, s, t\rangle$ is then distributed to $b_{1,1}$, $b_{2,1}$, and $b_{2,2}$. Block $b_{1,3}$ does not generate any record merges while blocks $b_{2,1}$ and $b_{2,2}$ are both preprocessed to $\{\langle r, s, t\rangle\}$, but do not generate record matches. Finally, block $b_{2,3}$ also does not generate record merges. Hence, the intermediate result after the second iteration is shown in Figure 5.

Criterion	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	$\langle r, s\rangle$,$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\langle u, v\rangle$
SC_2	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\langle u, v\rangle$

Figure 5: Intermediate result after second iteration

After one more iteration, we arrive at the final state in Figure 6. We then union the records of any blocking criterion (SC_1 or SC_2) to get the final answer $\{\langle r, s, t\rangle, \langle u, v\rangle\}$. In total, we have processed 18 blocks (6 for each of the three iterations) to derive the final solution.

Criterion	$b_{-,1}$	$b_{-,2}$	$b_{-,3}$
SC_1	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\langle u, v\rangle$
SC_2	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\langle u, v\rangle$

Figure 6: Final blocking result

We show that, given the coverage property and connected component strategy for MC, Algorithm 1 both terminates and returns a partition of R (see our extended paper [22] for all the proofs in this paper).

Lemma 2.5. *If the coverage property holds for MC and the connected component strategy is used, Algorithm 1 terminates and outputs a partition of R.*

Proposition 2.6. *If the coverage property holds for MC and the connected component strategy is used, the result of Algorithm 1 satisfies Definition 2.4.*

Lemma 2.7. *If the coverage property holds for MC and the connected component strategy is used, Step 22 of Algorithm 1 can be replaced by* 22: return $\bigcup_k IN(b_{j,k})$ for any j.

Although not presented here, one can also show that, if the connected component strategy is not used, Algorithm 1 may not terminate. In addition, if either the connected component strategy is not used or the coverage property does not hold for MC, Algorithm 1 may not output a partition of R.

The result of Algorithm 1 is unique as long as the CER algorithm is a "monotonic" process where merged records never unmerge, and the CER result is independent of the order in which records are processed. (Benjelloun et al. [3] shows how these properties can be achieved.) However, if the CER algorithm is order dependent, the result of Algorithm 1 also depends on the order in which blocks are processed.

To derive the worst case complexity of Algorithm 1, assume that each of N blocking criteria generates exactly B blocks. Then the time to process our set R of initial records is $O(N \times B \times \frac{|R|^2}{B^2} \times (|R| - 1)) = O(\frac{N \times |R|^3}{B})$. Here we assume that the CER algorithm does pairwise comparisons between $\frac{|R|}{B}$ records for each block processed. The complexity is based on the worst-case scenario where exactly one record merge occurs per iteration (the maximum number of record merges possible being $|R| - 1$). In practice, however, most of the record merges occur during the first iteration and decrease in number for subsequent iterations at an exponential rate. In Section 4, we show that iterative blocking may be faster than blocking because the work that is saved by exploiting the ER results of previously processed blocks exceeds the additional work done by multiple iterations.

2.5 The Lego Algorithm

The Lego algorithm (Algorithm 2) improves Algorithm 1 by efficiently managing merged records using the "maximal" records of base records. A maximal record of a base record r is denoted as $max(r)$ and is defined as the "largest" record containing r (see Definition 2.8 below). To illustrate, suppose there are two blocks b_1 and b_2 where b_1 contains two matching base records r and s while b_2 contains the two base records s and t. If b_1 is processed first, records r and s merge into $\langle r, s\rangle$, and we have $max(r) = max(s) = \langle r, s\rangle$. Before we process the second block, we now replace s by $max(s)$ before comparing the record with t.

Definition 2.8. *The* maximal record $max(r)$ *of base record r is a record where $max(r) \in \bigcup_{j,k} b_{j,k}$ and $\forall r' \in \bigcup_{j,k} b_{j,k}$ s.t. $r \subseteq r'$, then $r' \subseteq max(r)$.*

Lemma 2.9. *The maximal records in the Lego Algorithm satisfy Definition 2.8*

The second improvement is that the blocks are no longer processed sequentially, but are managed by the *block queue* Q. Initially, all the blocks are inserted into Q (Step 10) because they need to be processed. However, for each newly merged record r, we only re-insert the blocks in $MC(r)$ that are not already in Q (Steps 19∼23). We say that the blocks in $MC(r)$ are "hit" by r. Hence, only the blocks that have a possibility of generating new record merges are processed again. In addition, Q does not necessarily have to process the blocks in the order they were inserted, and can use any policy to choose which blocks to process first. (Our default policy is to process the blocks in the order of insertion.) When the same block is processed several times in Step 14, we can also save processing time if the CER algorithm is "incremental" and previously resolved records do not have to be processed again. In this case, the input R_i can then be given in two parts: R_a, which are the records that have already been compared, and R_b, which are the new records

```
1: input: A partition $P_i$ of $R$ and a core entity resolution algorithm CER
2: output: A partition $P_o$ of $R$ such that $P_i \leqslant P_o$
3: $Q \leftarrow \emptyset$
4: for each $r \in P_i$ do
5:    for each $r_b \in$ {Base records of $r$} do
6:       $max(r_b) = r$
7:    end for
8: end for
9: Create blocks
10: Push all blocks onto $Q$
11: while $Q \neq \emptyset$ do
12:    $b_{j,k} \leftarrow Q.pop()$
13:    $R_i \leftarrow Update(b_{j,k})$
14:    $R_o \leftarrow$ CER($R_i$)
15:    for each $r \in R_o - R_i$ do
16:       for each $r_b \in$ {Base records of $r$} do
17:          $max(r_b) = r$
18:       end for
19:       for each $b \in MC(r)$ do
20:          if $b \notin Q$ then
21:             $Q.push(b)$
22:          end if
23:       end for
24:    end for
25: end while
26: return $\bigcup_k Update(b_{0,k})$
27:
28: function Update($b_{j,k}$):
29: $b \leftarrow \emptyset$
30: for each $r \in IN(b_{j,k})$ do
31:    for each $r_b \in$ {Base records of $r$} do
32:       $b \leftarrow b \cup max(r_b)$
33:    end for
34: end for
35: return $b$
```

Algorithm 2: The Lego Algorithm

that need to be compared with all the records. (Note that $R_a \cup R_b = R_i$.) In this case, Step 14 can be replaced by "$R_o \leftarrow$ CER(R_a, R_b)."

To illustrate the operation of Lego, we revisit the example of Figure 3. We show how each block is processed and how the contents of Q changes in Figure 7. After the initialization in Steps 9∼10, $Q = \{b_{1,1}$,$b_{1,2}$,$b_{1,3}$,$b_{2,1}$,$b_{2,2}$,$b_{2,3}\}$ (we assume Q processes the blocks in the order they came in). We then process each block as in Algorithm 1, except that after we process block $b_{1,3}$, we do not distribute $\langle u, v\rangle$ to the other blocks but instead update $max(u)$ and $max(v)$ to $\langle u, v\rangle$ (Step 17). The maximal record information can be stored in a hash table that maps each base record to its maximal record. Since $b_{2,3}$ is already in Q, we do not push any block into Q. After we process block $b_{2,1}$, $max(r)$ and $max(s)$ are updated to $\langle r, s\rangle$. This time, we push the blocks $b_{1,1}$ and $b_{1,2}$ back onto the queue Q (Step 21) because $MC(\langle r, s\rangle) = \{b_{1,1}$,$b_{1,2}$,$b_{2,1}\}$, and $b_{2,1}$ is currently being processed. The queue Q then becomes $\{b_{2,2}$,$b_{2,3}$,$b_{1,1}$,$b_{1,2}\}$. After we process $b_{2,2}$ and $b_{2,3}$, we process $b_{1,1}$ again. In Step 13, we now directly update r with $max(r)$=$\langle r, s\rangle$, which is equivalent to the distribution and preprocessing steps of Algorithm 1. When preprocessing $b_{1,2}$, we update the block to $\{\langle r, s\rangle$,$t\}$. Once $\langle r, s\rangle$ and t merge to $\langle r, s, t\rangle$, we update $max(r)$, $max(s)$, and $max(t)$ to $\langle r, s, t\rangle$. Since $MC(\langle r, s, t\rangle)$ = $\{b_{1,1}, b_{1,2}$, $b_{2,1}, b_{2,2}\}$, blocks $b_{1,1}$,$b_{2,1}$,$b_{2,2}$ are pushed back onto Q. While processing the remaining blocks in Q, there are no new record matches, and Q becomes empty. Hence, we arrive at the final result shown in Figure 6.

Lego processes fewer blocks than Algorithm 1. While Algorithm 1 processes 18 blocks (6 blocks processed for each of the 3 iterations), Lego only processes 11 blocks (see Figure 7). Also, Lego does not directly update the blocks with new merged records but instead manages the maximal record of each base record.

The following proposition establishes the correctness of the Lego algorithm.

Proposition 2.10. *If the coverage property holds for MC and the connected component strategy is used, the Lego algorithm returns a valid I-BlockER result of R.*

3. DISK-BASED ITERATIVE BLOCKING

An important requirement for iterative blocking is to scale ER to large datasets. In real applications, we may have millions of records that do not necessarily fit in the memory and need to be stored on disk. The Lego algorithm (Algorithm 2) is not scalable because it assumes that the blocks are not stored on the disk. Moreover, for large datasets, even managing the maximal records using a hash table could exceed the memory.

In this section we introduce a disk-based iterative blocking system that efficiently manages blocks on the disk. Our system improves Lego in two ways. First, the blocks are now saved in fixed-sized extents called *segments* on the disk. A segment is the unit of transfer to memory and has the advantage of evening out blocks of different sizes. Intuitively, we are now processing "N blocks at at time" in memory, assuming each segment contains N records on average. Second, we use a *merge log* for managing maximal records. A merge log keeps track of record merges and can be sequentially accessed from the disk to update the blocks. The merge log has the advantages of using a small amount of memory and of requiring sequential I/O. We discuss segments and the merge log in detail in the following sections. We then propose a disk-based algorithm (called Duplo) that uses segments and the merge log, and satisfies the I-BlockER model.

3.1 Segments

We use fixed-sized extents called segments to store the blocks on disk. A segment acts as a unit of transfer for reading blocks into memory and thus cannot exceed the memory size. For each blocking criterion, we allocate a fixed number of segments consecutively on disk. We then allocate the segments for the next blocking criterion consecutively and so on. Hence, each segment belongs to one blocking criterion and contains the blocks of that criterion. For each blocking criterion, the blocks are randomly assigned to segments for even distribution. Figure 8 shows a possible assignment of blocks to segments for Figure 2. For example, segment $s_{1,1}$ contains the blocks $b_{1,1}$ and $b_{1,3}$. The segments $s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}$ are stored on the disk consecutively. Internally, a segment stores the union of the records of its blocks. This strategy saves space for storing the blocks because the blocks may contain overlapping records. Each block can later be extracted from the segment by applying MC to each record. A segment is thus a set of records, and we denote the set of records inside segment s as $IN(s)$.

The advantage of using segments is that the different block sizes are evened out when they are randomly assigned to the segments. As a result, we can approximately do the same amount of work for each segment processed. One problem

Nth block	Block processed	Records before CER	Records after CER	Q after CER
1	$b_{1,1}$	r	r	$\{b_{1,2}, b_{1,3}, b_{2,1}, b_{2,2}, b_{2,3}\}$
2	$b_{1,2}$	s, t	s, t	$\{b_{1,3}, b_{2,1}, b_{2,2}, b_{2,3}\}$
3	$b_{1,3}$	u, v	$\langle u, v\rangle$	$\{b_{2,1}, b_{2,2}, b_{2,3}\}$
4	$b_{2,1}$	r, s	$\langle r, s\rangle$	$\{b_{2,2}, b_{2,3}, b_{1,1}, b_{1,2}\}$
5	$b_{2,2}$	t	t	$\{b_{2,3}, b_{1,1}, b_{1,2}\}$
6	$b_{2,3}$	$\langle u, v\rangle$	$\langle u, v\rangle$	$\{b_{1,1}, b_{1,2}\}$
7	$b_{1,1}$	$\langle r, s\rangle$	$\langle r, s\rangle$	$\{b_{1,2}\}$
8	$b_{1,2}$	$\langle r, s\rangle$,t	$\langle r, s, t\rangle$	$\{b_{1,1}, b_{2,1}, b_{2,2}\}$
9	$b_{1,1}$	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\{b_{2,1}, b_{2,2}\}$
10	$b_{2,1}$	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\{b_{2,2}\}$
11	$b_{2,2}$	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\{\}$

Figure 7: Processing blocks with the Lego algorithm

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	$b_{1,1}, b_{1,3}$	$b_{1,2}$
SC_2	$b_{2,1}$	$b_{2,2}, b_{2,3}$

Figure 8: Assigning blocks to segments for Figure 2

with fixed-sized segments is that they may overflow during the iterative blocking process. For example, segment $s_{1,1}$ of Figure 8 starts with the records $\{r,u,v\}$ and ends with $\{\langle r, s, t\rangle, \langle u, v\rangle\}$ where $\langle r, s, t\rangle$ could be larger than r while $\langle u, v\rangle$ has a similar size as u and v combined. To prevent overflows, we leave some extra space for each segment in case the blocks grow in size. Although we could dynamically extend the segment size during runtime, a fixed extension of space is also reasonable and works well in practice.

3.2 Merge Log

The merge log is used to update the records in a block to their maximal records. The merge log is required when we cannot manage the maximal records in memory for large datasets. While we could also implement a hash table on disk, updating a block would clearly involve many random I/Os for looking up the maximal records. Instead, the merge log keeps track of record merges and can be read sequentially to update a block. For example, if two records r and s merge into $\langle r, s\rangle$, then we add to the merge log the entries $r \rightarrow \langle r, s\rangle$ and $s \rightarrow \langle r, s\rangle$. Later on when we process a segment containing r, we seek the "appropriate" region of the log to find the entry $r \rightarrow \langle r, s\rangle$ and update r to $\langle r, s\rangle$.

The advantage of the merge log is that a single sequential scan of the merge log is sufficient to update a block. Although the number of entries in the merge log could grow to 2*$|R|$-2 (in the case where all the records merge into a single record), the number of records that merge is typically much smaller than $|R|$, making the merge log size reasonably small. Algorithm 3 shows how to update a block by scanning the merge log once. Notice that each block b has a "timestamp," that can be used to skip L entries that have already been applied. Hence, we only need to read the merge log from the entry with the next largest timestamp. In the next section, we also use a function called *UpdateSegment*, which is identical to *UpdateBlock* except that a segment is updated instead of a block.

3.3 The Duplo Algorithm

The Duplo algorithm (Algorithm 4) uses segments and the merge log to scale iterative blocking. The idea of Duplo is to process "N blocks at a time," assuming a segment contains N blocks on average. Hence, the iterative blocking is now done in two levels: processing a segment and then processing the blocks within that segment.

```
$UpdateBlock(b,L)$:
Skip $L$ entries with timestamps less than or equal to the timestamp of $b$
while $L$ has more entries do
    $\langle r \rightarrow s\rangle \leftarrow L.nextEntry()$
    if $r \in IN(b)$ then
        $IN(b) \leftarrow IN(b) - \{r\} + \{s\}$
    end if
end while
return $IN(b)$
```

Algorithm 3: Updating a block using the merge log

We maintain two queues and two merge logs for managing the segments and the blocks within each segment. In order to process segments, we use a *segment queue* Q_1 that determines which segment to process next and a "global" merge log L_1 that keeps track of all the record merges done until now. Next, in order to process the blocks of a single segment, we use a *block queue* Q_2 that determines which block to process next (just like Q in Lego) and a "local" merge log L_2 that keeps track of the record merges done within the current segment. Both segments and blocks have timestamps indicating which point of the log they are up-to-date for L_1 and L_2, respectively. Merge log L_1 is located on disk and is accessed once for each segment processed while L_2 is in memory and is accessed once for each block processed.

To illustrate the Duplo algorithm, suppose that we create the segments of Figure 8. The actual contents of the segments are shown in Figure 9 (i.e., each segment contains a union of records of its blocks). We show how each segment is processed in Figure 10.

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	r, u, v	s, t
SC_2	r, s	t, u, v

Figure 9: Initial segments of Duplo

Initially, all four segments are placed in the segment queue Q_1 (i.e., Q_1 =$\{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}\}$). When we first process segment $s_{1,1}$, we extract the two blocks $b_{1,1}$ =$\{r\}$ and $b_{1,3}$ =$\{u, v\}$, and process them with the CER algorithm. Since u and v merge into $\langle u, v\rangle$, the merge logs L_1 and L_2 are both set to $\{u \rightarrow \langle u, v\rangle, v \rightarrow \langle u, v\rangle\}$. Although $s_{2,2}$ is hit by $\langle u, v\rangle$, we do not insert the segment into Q_1 because Q_1 already contains $s_{2,2}$. We then continue with processing segment $s_{1,2}$, which does not have matching records. When

Nth segment	Segment processed	Records before CER	Records after CER	Q_1 after CER
1	$s_{1,1}$	r, u, v	$r, \langle u, v\rangle$	$\{s_{1,2}, s_{2,1}, s_{2,2}\}$
2	$s_{1,2}$	s, t	s, t	$\{s_{2,1}, s_{2,2}\}$
3	$s_{2,1}$	r, s	$\langle r, s\rangle$	$\{s_{2,2}, s_{1,1}, s_{1,2}\}$
4	$s_{2,2}$	$t, \langle u, v\rangle$	$t, \langle u, v\rangle$	$\{s_{1,1}, s_{1,2}\}$
5	$s_{1,1}$	$\langle r, s\rangle, \langle u, v\rangle$	$\langle r, s\rangle, \langle u, v\rangle$	$\{s_{1,2}\}$
6	$s_{1,2}$	$\langle r, s\rangle, t$	$\langle r, s, t\rangle$	$\{s_{1,1}, s_{2,1}, s_{2,2}\}$
7	$s_{1,1}$	$\langle r, s, t\rangle, \langle u, v\rangle$	$\langle r, s, t\rangle, \langle u, v\rangle$	$\{s_{2,1}, s_{2,2}\}$
8	$s_{2,1}$	$\langle r, s, t\rangle$	$\langle r, s, t\rangle$	$\{s_{2,2}\}$
9	$s_{2,2}$	$\langle r, s, t\rangle, \langle u, v\rangle$	$\langle r, s, t\rangle, \langle u, v\rangle$	$\{\}$

Figure 10: Processing segments with the Duplo algorithm

```
input: a partition $P_i$ of $R$ and a core entity resolution algorithm CER
output: a partition $P_o$ of $R$ such that $P_i \leqslant P_o$
$L_1 \leftarrow \emptyset$  /* Disk merge log for segments */
$Q_1 \leftarrow \emptyset$  /* Segment queue */
Create segments
Push all segments into $Q_1$
while $Q_1$ is not empty do
    $s \leftarrow Q_1.Pop()$
    $IN(s) \leftarrow UpdateSegment(s, L_1)$
    $L_2 \leftarrow \emptyset$  /* In-memory merge log for blocks */
    $Q_2 \leftarrow \emptyset$
    Push all blocks in $s$ into $Q_2$
    while $Q_2$ is not empty do
        $b \leftarrow Q_2.Pop()$
        $R_i \leftarrow UpdateBlock(b, L_2)$
        $R_o \leftarrow$ CER($R_i$)
        Add to $L_2$ the new record merges in $b$
        Add to $L_1$ the new record merges in $b$
        for each record $r \in R_o - R_i$ do
            for each block $b' \in MC(r)$ do
                $s' \leftarrow BlockToSegment(b')$  /* Returns the segment where $b'$ is */
                if $s = s'$ then  /* Hit the same segment */
                    if $b \neq b'$ and $b' \notin Q_2$ then
                        $Q_2.Push(b')$
                    end if
                else  /* Hit a different segment */
                    if $s' \notin Q_1$ then
                        $Q_1.Push(s')$
                    end if
                end if
            end for
        end for
    end while
    Write $s$ back to disk
end while
$J \leftarrow \emptyset$
$J \leftarrow$ {Records in $R$ that were never merged}
$J \leftarrow J \cup$ {Records in $L_1$ that are not contained by any other record in $L_1$}
return $J$
```

Algorithm 4: The Duplo algorithm

we process segment $s_{2,1}$, we extract $b_{2,2} = \{r, s\}$ and merge r and s into $\langle r, s\rangle$. The global merge log L_1 is then updated to $\{u \rightarrow \langle u, v\rangle,\ v \rightarrow \langle u, v\rangle,\ r \rightarrow \langle r, s\rangle,\ s \rightarrow \langle r, s\rangle\}$ while L_2 is updated to $\{r \rightarrow \langle r, s\rangle,\ s \rightarrow \langle r, s\rangle\}$. (Notice that L_1 and L_2 are now different because L_2 only keeps track of the local merges within $s_{1,2}$.) Since $\langle r, s\rangle$ hits the segments $s_{1,1}$ and $s_{1,2}$, we push $s_{1,1}$ and $s_{1,2}$ back into Q_1 (i.e., $Q_1 = \{s_{2,2}, s_{1,1}, s_{1,2}\}$). The next segment $s_{2,2}$ is updated to $\{t, \langle u, v\rangle\}$, but does not generate any record matches. Similarly, the next segment $s_{1,1}$ is updated to $\{\langle r, s\rangle, \langle u, v\rangle\}$, but does not generate record matches. We then process $s_{1,2}$. This time, after we update the segment to $\{\langle r, s\rangle, t\}$ and extract the block $b_{1,2} = \{\langle r, s\rangle, t\}$, we merge $\langle r, s\rangle$ and t into $\langle r, s, t\rangle$. The global merge log L_1 is updated to $\{u \rightarrow \langle u, v\rangle, v \rightarrow \langle u, v\rangle, r \rightarrow \langle r, s\rangle, s \rightarrow \langle r, s\rangle, \langle r, s\rangle \rightarrow \langle r, s, t\rangle, t \rightarrow \langle r, s, t\rangle\}$ while L_2 is updated to $\{\langle r, s\rangle \rightarrow \langle r, s, t\rangle, t \rightarrow \langle r, s, t\rangle\}$. Since $\langle r, s, t\rangle$ hits all four segments, we insert $s_{1,1}, s_{1,2}$, and $s_{2,1}$ back into Q_1. After processing all the segments in Q_1, we arrive at the final state of Figure 11.

Criterion	$s_{-,1}$	$s_{-,2}$
SC_1	$\langle r, s, t\rangle, \langle u, v\rangle$	$\langle r, s, t\rangle$
SC_2	$\langle r, s, t\rangle$	$\langle r, s, t\rangle, \langle u, v\rangle$

Figure 11: Final state of Duplo

The correctness of the Duplo algorithm is shown in Proposition 3.1.

PROPOSITION 3.1. *If the coverage property holds for MC and the connected component strategy is used, the Duplo algorithm returns a valid I-BlockER result of R.*

Segment queue policy. The policy for determining which segment to process from Q_1 significantly affects the runtime of Duplo. The following list shows various policies Q_1 might have.

- *Hits*: Sorts the segments in decreasing number of hits they receive from newly merged records of other segments. Segments with the highest number of hits are processed first. Initially, all segments have an infinite number of hits. The hit count for a segment is set to zero after the segment is processed.
- *First-Come-First-Serve (FCFS)*: Processes the segments in the order they were pushed into the segment queue.
- *Random*: Randomly processes segments.
- *Inverse Hits*: Sorts the segments in increasing number of hits, so that the segments with the fewest hits are processed first.

Later in Section 4.3, we compare the policies and show which policy makes Duplo efficient.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate iterative blocking on real datasets and show how iterative blocking outperforms blocking both in accuracy and runtime. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM. We also used a raw disk without a file system for storing the blocks in order to avoid caching and accurately measure the I/O cost.

Real Dataset. The comparison shopping dataset we use was provided by Yahoo! Shopping and contains millions of records that arrive on a regular basis from different online stores and must be resolved before they are used to answer customer queries. Each record contains various attributes including the title, price, and category of an item. We experimented on a set of 2 million records randomly chosen from the entire dataset. We also experimented on a hotel dataset provided by Yahoo! Travel (see Section 4.4) where tens of thousands of records arrive from different travel sources (e.g., Orbitz.com), and must be resolved before they are shown to the users.

CER Algorithm. For our evaluation we use two different CER algorithms. Our primary CER algorithm is the R-Swoosh algorithm of Benjelloun et al. [3], and the main body of our results are generated with this algorithm. However, to illustrate that our Lego and Duplo algorithms can be used with any CER algorithm, in Section 4.5 we study a different algorithm, and show how the accuracy-performance tradeoffs vary. For both algorithms we use the products and travel data sets.

R-Swoosh uses a Boolean pairwise match function to compare records and a pairwise merge function to merge two records that match into a composite record. The match function for the shopping data compares the title, price, and category values of two records. (Details on the match and merge functions can be found in Benjelloun et al. [3].) For the hotel dataset, we compared the names and addresses of hotels. R-Swoosh returns a partition of R by merging records and thus satisfies our ER model.

Blocking Criteria. We use minhash signatures [14] for distributing the records into blocks. A minhash signature is used to estimate the Jaccard similarity between two strings (i.e., the portion of n-grams of the strings shared). Records with the same minhash signature are assigned to the same block. Also, merged records are assigned to all of its base record blocks, satisfying the coverage property. For our datasets, we extract 3-grams from the titles of the shopping records. Similarly, we extracted 3-grams from the names of the hotel records. Throughout our experiments, we do not vary the n-gram length and fix it to 3. We then generate a minhash signature that is an array of integers where each integer is generated by applying a random hash function to the 3-gram set of the record. We can produce several minhash signatures of a record for each blocking criterion using different sets of random hash functions.

The advantage of using minhash signatures is that we can easily adjust the number of blocking criteria and signature length to produce reasonable accuracy and performance. Although there are many other ways to produce blocking criteria (e.g., manually creating blocking criteria instead of using minhash signatures), we believe our approach is ideal in showing the trends of accuracy and performance against different "qualities" of blocking criteria.

Metrics. We used accuracy and runtime metrics to evaluate iterative blocking. To evaluate accuracy, we compare our algorithm results with a "Gold Standard," which is the result of running CER on the entire dataset (i.e., CER(R)). Notice that we are *not* measuring the correctness of the CER algorithm itself, but instead determining how "close" the blocking or iterative blocking results are to the exhaustive result. We consider all the input records that merged into an output record to be identical to each other. For instance, if records r and s merged into $\langle r, s\rangle$ and then merged with t, all three records r, s, t are considered to be the same. Suppose that the Gold Standard G contains the set of record pairs that match for the exhaustive solution while set S contains the matching pairs for our algorithm. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy metric. For runtime, we measured the wall-clock runtime for each algorithm.

4.1 Accuracy

We compare the accuracy (F_1-measure) of the Lego algorithm with the following techniques:

- *Blocking*: Runs CER on each block separately. The answer is produced by simply collecting the final records from all the blocks.
- *Blocking-CC*: Runs CER on each block, and then performs a connected component operation on all the records. For example, if the final records $\langle r, s\rangle$ and $\langle s, t\rangle$ are in different blocks, they are merged to $\langle r, s, t\rangle$.

We use a relatively small dataset of 50,000 records in order to be able to compare our results with the Gold Standard, which takes a long time to produce. The accuracy and runtime values are the average of five test results on distinct random subsets with the same 50,000 base records.

Varying the Average Block Size. In Figure 12, we compare the accuracy results of Lego, Blocking, and Blocking-CC using different average block sizes. Varying the minhash signature length affects the average block size. In our experiments, we varied the minhash signature length from 14 to 1 integers (the shorter the minhash signature, the larger the average block size becomes) and fixed the number of blocking criteria to 5. However, instead of plotting all graphs with the signature length as the horizontal axis, we plotted against the average block size (i.e., the average number of records in a block), which is the average size of all the blocks produced by MC. We believe that the average block size is a more intuitive parameter, one that captures the number of records that are compared with each other. If the signature length decreases, more records are likely to be assigned to the same block, increasing the average block size. If the average block size increases, we find more record matches, but also perform more record comparisons.

As the average block size increases, Lego shows the most rapid increase in accuracy (see Figure 12). For example, when the average block size is about 13, Lego has 88% accuracy while Blocking has 76% accuracy. Blocking-CC has a higher accuracy than Blocking, but still has a lower accuracy than Lego because, although it finds more matching records, it also finds many false positives by merging records unnecessarily, which lowers the precision. Notice that, for the largest average block size 32, Blocking has almost the same accuracy as Lego because enough records are compared with each other. However, increasing the average block also increases the runtime as we shall see in Section 4.2.

Varying the Number of Blocking Criteria. We also compared the accuracies of Lego, Blocking, and Blocking-CC

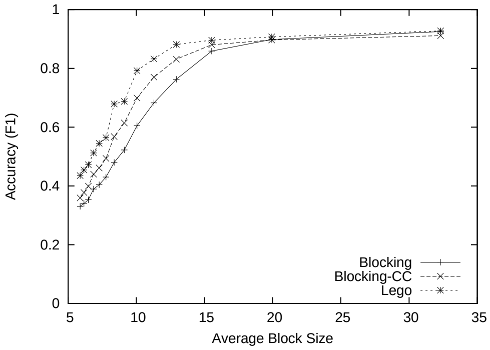

Figure 12: Average block size impact on accuracy, 50K records

while fixing the minhash signature length to 10 (making the average block size 7.3, the fifth smallest block size in Figure 12) and increasing the number of blocking criteria from 1 to 15 as shown in Figure 13. As the number of blocking criteria increases, the accuracy of Lego increases faster than the other approaches. When the number of blocking criteria is 15, Lego achieves 78% accuracy while Blocking achieves 55% accuracy. All the algorithms will eventually achieve high accuracy as the number of blocking criteria increases further, but Lego can achieve a high accuracy using many fewer blocking criteria than the other approaches.

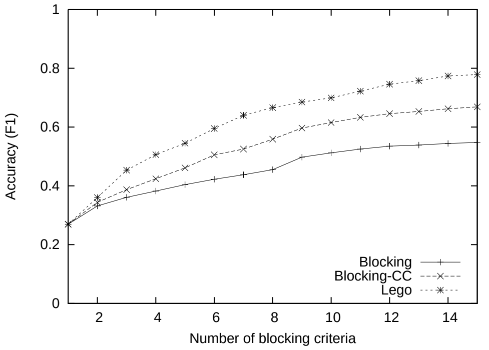

Figure 13: Number of Blocking Criteria impact on accuracy, 50K records

4.2 Lego Runtime

We compare the runtimes of Lego, Blocking, and Blocking-CC with the Gold Standard (i.e., running ER on the entire dataset). In all cases, all data was memory resident. Figure 14 shows the runtimes (on a log scale) while fixing the number of blocking criteria to 5 and decreasing the minhash signature length from 14 to 1 integers to generate different average block sizes. The points of Figure 14 that have an average block size of 7.3 (using a signature length of 10) correspond to the points of Figure 13 that use 5 blocking criteria. The runtime for the Gold Standard is 1641 seconds while the runtime for Lego ranges from 10 to 289 seconds.

Lego also has a comparable performance with Blocking. Lego is actually faster than Blocking for the average block sizes of 16 and 20 because the time saved by reflecting the CER results of blocks to other blocks is larger than the additional time needed to iteratively process blocks. To illustrate how Lego can be faster than Blocking, suppose that two blocks b_1 and b_2 contain the same two matching records r and s. While Blocking compares r and s twice, Lego can process b_1 and reflect the merged record $\langle r, s\rangle$ on b_2, saving the next record comparison. Finally, Blocking-CC is slower than Lego and Blocking because of the additional overhead of connecting components.

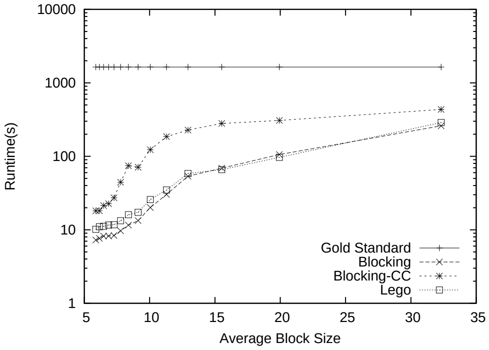

Figure 14: Average block size impact on runtime, 50K records

We omit the plot for the runtime versus number of blocking criteria due to space and because it shows a similar trend as Figure 14.

4.3 Duplo Performance

We now compare the performances of the Duplo algorithm and Blocking on large datasets that may not fit in memory. For Blocking, we simply read all the blocks from disk and run ER on each block. For the Duplo algorithm, we now use segments and the merge log to iteratively process blocks from the disk. We also test on a variant of Duplo (called Duplo Memory), which uses segments but not the global merge log L_1 (i.e., Duplo Memory keeps an in-memory hash table for managing maximal records just like in Lego). While having a hash table in memory enhances the performance of Duplo, we must tradeoff that gain against the requested larger amount of memory.

Before comparing the three techniques, we first evaluate several segment queue policies that determine which segment to process from Q_1 first. Once we have found the best segment queue policy, we then compare the scalability results of Duplo, Duplo Memory, and Blocking. We show that both Duplo and Duplo Memory outperform Blocking when producing accurate ER results of large datasets. Throughout the experiments for Duplo and Duplo Memory, we use 160 segments per blocking criterion for 5 blocking criteria and allocate 30MB [1] of disk space per segment.

Segment Queue Policy. The segment queue policy significantly affects the number of segments that need to be processed as well as the overall runtime. We compare the four policies mentioned in Section 3.3: Hits, FCFS, Random, and Inverse Hits. Figure 15 shows the runtime results of using the four policies on 1 million random shopping records where the minhash signature length is 10.

Both Hits and FCFS have excellent performance where Hits is slightly faster than FCFS. The Hits policy works well

[1] While a segment size of 30MB may seem small, additional memory is needed to process the records in the segment by the CER algorithm and the local merge log L_2.

because segments that have many hits are likely to generate many record merges each time they are processed. The FCFS policy also works well because segments stay in the queue as long as possible and thus tend to generate more record merges when they are processed. The Inverse Hits policy is the worst strategy and is even slower than the Random policy because segments that are least likely to generate record matches are processed first, resulting in repeated processing of the same segments. Having a good segment queue strategy is thus very important for efficiency. For the rest of our experiments for Duplo and Duplo Memory, we use Hits as our default policy.

Strategy	Runtime (hrs)
Hits	2.0
FCFS	2.1
Random	7.5
Inverse Hits	11.7

Figure 15: Runtimes for different segment queue strategies, 1M records

Scalability. Using the Hits policy, we compare the scalability results of Duplo, Duplo Memory, and Blocking. Figure 16 shows the result of running the three algorithms on 0.5 to 2 million random shopping records where the minhash signature length is 4 (the average block sizes being 75, 130, 232 for 0.5, 1, 2 million records, respectively). We set the maximum java heap size to 3.5G for Duplo Memory and 1.5G for Duplo in order to demonstrate that Duplo uses a small amount of memory (Duplo Memory actually slows down significantly when using 1.5G of memory). Both Duplo and Duplo Memory scale better than Blocking (by 15% and 43% for 2 million records, respectively) because Duplo and Duplo Memory save processing time by reflecting the ER results of blocks to other blocks. Duplo Memory is 33% faster than Duplo for 2 million records. The runtime improvements will of course depend on the number of blocking criteria and average block size used.

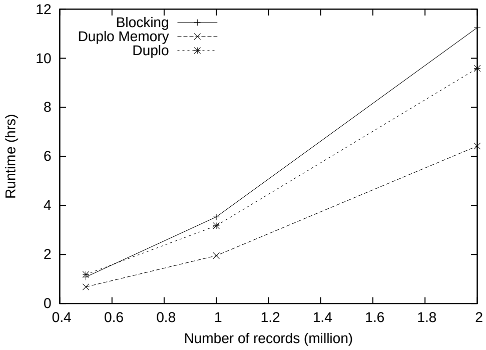

Figure 16: Scalability, 2M records

Figure 17 shows the runtimes needed for Duplo, Duplo Memory, and Blocking to achieve certain accuracy results on 2 million records. We generated the figure by running a series of scenarios with different minhash signatures. For each scenario we obtained an accuracy and runtime pair, and plotted it in Figure 17. We could not directly calculate the accuracy because the dataset was too large to compute the Gold Standard using the CER algorithm. Instead, we exploited the fact that the accuracy is only dependent on the number of blocking criteria and minhash signature length, and not on the size of the random dataset. Hence, by using the same number of blocking criteria and minhash signature length, we could use the accuracy results on 50,000 records in Figure 12 to estimate the accuracy results on 2 million records. We can see in the figure that both Duplo and Duplo Memory significantly outperform Blocking for highly accurate results. For example, Duplo Memory and Duplo takes 14 and 17 hours to achieve 91% accuracy while Blocking takes 26 hours to achieve 90% accuracy.

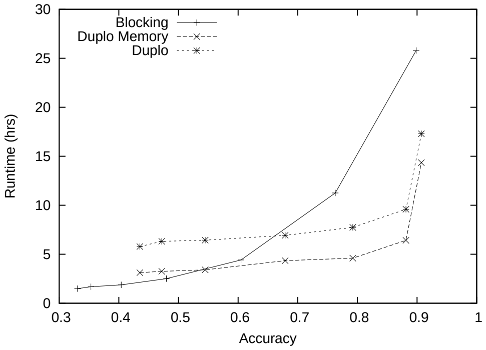

Figure 17: Runtime needed to achieve given accuracy, 2M records

In summary, Duplo and Duplo Memory outperform Blocking when producing accurate ER results on large datasets. The key idea is that ER results of blocks are reflected on other blocks, saving a lot of processing time. Although Duplo Memory is faster than Duplo, it requires a large amount of memory to use the in-memory hash table. Duplo should be used instead of Duplo Memory when the main memory size is too small to hold the hash table.

4.4 Other Datasets

In our shopping application, there are many iterative record matches where merged records generate additional record matches. However, in other applications there may be fewer iterative matches. For example, suppose that there are two blocks b_1 $=\{r, s\}$ and b_2 $=\{s, t\}$. If r and s merge into $\langle r, s\rangle$ in b_1, but never match with t, then we cannot generate any new record matches by reflecting $\langle r, s\rangle$ to b_2. In order to investigate how iterative blocking works for different types of datasets, we also tested on a hotel dataset provided by Yahoo! Travel. The records in the hotel dataset mostly come from two datasources that do not have duplicates within themselves. As a result, we rarely have more than two records matching with each other.

In this case, iterative blocking does not significantly improve accuracy over blocking. However, its blocks still have a performance gain because ER processing in one block saves processing time for subsequent blocks. Figure 18 shows the runtimes for Duplo Memory and Blocking on hotel records. The hotel dataset was only 27,049 records, so we generated larger datasets by simply replicating the same dataset 16, 32, and 64 times (resulting in 0.43, 0.86, 1.73 million records, respectively). Duplo Memory improves Blocking by 48% for 1.73 million records. We omit the data for Duplo because replicating the data was producing too many record merges, significantly increasing the time to read the merge log for each segment. In practice, the number of merging records is much smaller.

Algorithm	Runtimes for 0.43M / 0.86M / 1.73M records (hrs)
Blocking	0.27 / 0.73 / 2.27
Duplo Memory	0.18 / 0.41 / 1.18

Figure 18: Runtimes on the hotel dataset, 1.7M records

4.5 Other CER Algorithms

In this section, we experiment with a CER algorithm based on Monge and Elkan [17] (we call it the ME algorithm) where records are sorted using an application-specific key and then clustered with a sequential scan. Each cluster has a representative record that can be compared with other records. The representative record contains values of recently added records to the cluster, but only those that were "far away enough" from the representative record when added to the cluster. During the sequential scan, a priority queue is used to contain the most recently updated clusters. (While the size of the priority queue in [17] is constant, we use a size proportional to the number of records processed.) Each new record is compared to all the clusters in the priority queue. If there is a matching cluster, the record is combined with the cluster, and the cluster is moved to the head of the priority queue. If there is no match, a new singleton cluster is created and pushed into the head of the priority queue. The final result is a set of clusters. The ME algorithm can also start from a set of clusters (using the representative records) and thus satisfies Definition 2.2.

We used the Yahoo! Shopping dataset and compared the records with the same match function used in R-Swoosh. We also used the same blocking criteria that uses minhash signatures generated from the titles of the records. Throughout our experiments, we used 5 blocking criteria and a minhash signature length of 3.

Figure 19 shows that Lego has a higher accuracy than Blocking or Blocking-CC for 50,000 shopping records. To correctly measure the accuracy values, we constructed the Gold Standard by repeatedly running the ME algorithm on the entire dataset until the clusters no longer merged. Although both Lego and Blocking-CC have lower precision values than Blocking, they have much higher recall and accuracy values.

Algorithm	Precision	Recall	Accuracy
Lego	0.93	0.74	0.82
Blocking	0.99	0.38	0.55
Blocking-CC	0.93	0.69	0.79

Figure 19: Accuracy for the ME algorithm, 50K records

Figure 20 shows that Duplo Memory is 42∼76% slower than Blocking while Duplo Disk is 101∼109% slower than Blocking for 0.5 to 2 million shopping records (the average block sizes being 165, 309, 577 for 0.5, 1, 2 million records, respectively). This result contrasts with that of Figure 16 where Blocking is the slowest algorithm. The reason is that ME was not finding all the matching records for each block (notice the low recall for Blocking in Figure 19) because the shopping data required iterative comparisons to be completely resolved. As a result, we could not exploit the ER results of previously processed blocks as much as we did before in Figure 16. In addition, more blocks had to be iteratively processed until there were no matching records in any of the blocks. Hence, the ME algorithm illustrates the case where we are trading off runtime (i.e., the overhead of managing the blocks and merge logs) for better accuracy.

Algorithm	Runtimes for 0.5M / 1M / 2M records (hrs)
Blocking	0.83 / 2.95 / 10.46
Duplo Memory	1.18 / 4.75 / 18.37
Duplo Disk	1.67 / 6.02 / 21.9

Figure 20: Scalability for the ME algorithm, 2M records

In summary, both Duplo Memory and Duplo Disk are better than Blocking in accuracy, but are worse in runtime when the ME algorithm is used as the CER algorithm. It is important to understand that we are not comparing the performances of R-Swoosh and ME through these experiments. ME is designed to be an efficient algorithm for data that can easily be clustered (which is not the case here) while R-Swoosh is a more exhaustive algorithm that can find subtle record matches. The key observations here are that any CER algorithm can be plugged into the iterative blocking framework and that there is a different runtime/accuracy tradeoff when using the ME algorithm.

5. RELATED WORK

Entity Resolution has been studied under various names including record linkage [19], merge/purge [12], deduplication [20], reference reconciliation [7], object identification [21], and others (see [8, 23, 9] for recent surveys). Entity Resolution involves comparing records and determining if they refer to the same entity or not. Many exhaustive ER algorithms [17, 1, 4, 6, 3] can be used as the core ER algorithm of our iterative blocking framework.

Various blocking techniques that focus on ER accuracy have been proposed. Early works [19, 18] manually generate blocking criteria based on the characteristics of the data. Learning techniques [5, 16] produce blocking criteria based on training data so that the number of matching record pairs found is maximized while the number of non-matching record pairs in the same block is minimized. Estimation techniques [24, 13] can be used to estimate the number of matching record pairs missed by each blocking criterion. Most of the works above use several blocking criteria to increase the chance of similar records to be compared within the same block. Our iterative blocking framework can use any blocking criteria produced from the works above as its MC function.

Another line of research is blocking techniques that focus on ER performance [2]. The Canopy Clustering [15] technique uses a cheap, approximate distance measure to efficiently divide the data into overlapping subsets called canopies and then processes each canopy using exact distances. The notion of canopies fits into our model of overlapping blocks. In addition, iterative blocking can use several blocking criteria instead of one. Most importantly, iterative blocking exploits the ER results of previously processed blocks. The Sorted Neighborhood method [11] sorts records based on a sorting key and moves a fixed-sized window sequentially over the sorted records. Several passes can be done using different sorting keys. A transitive closure is then performed on all the matching record pairs. In comparison, the iterative blocking framework takes a generic approach and does not assume sorting keys. Nevertheless, the Sorted Neighborhood technique also fits into our iterative blocking framework where windows of records can be viewed as blocks. While the Sorted Neighborhood enhances the processing of blocks within the same blocking criterion, it does not exploit the ER results of blocks in different blocking criteria. Hence, although the works above fit into our iterative blocking framework, they do not fully exploit the ER results of different blocks, as we do.

To the best of our knowledge, the only work that takes a generic approach in efficiently processing blocks is the BigMatch [25] algorithm. BigMatch compares a large file of records on disk with a smaller file of records in memory without having to sort the larger file. For each record from the larger file, BigMatch uses the blocking criteria to search all the records in the smaller file that are in the same block for at least one blocking criteria. In contrast, iterative blocking is designed to resolve a single large set of records and has the additional feature of iteratively processing blocks.

6. CONCLUSION

We have proposed a novel framework for iterative blocking where the ER results of blocks are reflected to other blocks, possibly generating new record matches. Blocks are processed in an iterative fashion (possibly more than once) until no blocks contain any matching records. Our approach can be used with any "core" ER algorithm that processes a block of records. The potential advantage of iterative blocking is twofold. First, we achieve high accuracy because reflecting the ER results can generate new record matches in subsequent blocks. Second, we have fast convergence because the ER results of blocks save the processing time for other blocks. We have implemented an in-memory algorithm called Lego and a disk-based algorithm called Duplo, which uses segments and a merge log to process blocks from the disk. We experimentally showed that Duplo can largely outperform blocking while producing highly accurate ER results on large datasets.

7. REFERENCES

[1] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. In *FOCS*, pages 238–, 2002.

[2] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Identification*, 2003.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, S. E. Whang, Q. Su, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 2008.

[4] I. Bhattacharya and L. Getoor. Iterative record linkage for cleaning and integration. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, 2004.

[5] M. Bilenko, B. Kamath, and R. Mooney. Adaptive blocking: Learning to scale up record linkage. In *ICDM*, 2006.

[6] S. Chaudhuri, V. Ganti, and R. Motwani. Robust identification of fuzzy duplicates. In *Proc. of ICDE*, Tokyo, Japan, 2005.

[7] X. Dong, A. Halevy, and J. Madhavan. Reference reconciliation in complex information spaces. In *SIGMOD*, 2005.

[8] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[9] L. Gu, R. Baxter, D. Vickers, and C. Rainsford. Record linkage: Current practice and future directions. Technical Report 03/83, CSIRO Mathematical and Information Sciences, 2003.

[10] L. Gu and R. A. Baxter. Adaptive filtering for efficient record linkage. In *SDM*, 2004.

[11] M. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[12] M. A. Hernández and S. J. Stolfo. The merge/purge problem for large databases. In *Proc. of ACM SIGMOD*, pages 127–138, 1995.

[13] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data Quality and Record Linkage Techniques*. Springer, July 2007.

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

[15] A. K. McCallum, K. Nigam, and L. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proc. of KDD*, pages 169–178, Boston, MA, 2000.

[16] M. Michelson and C. Knoblock. Learning blocking schemes for record linkage. In *AAAI*, 2006.

[17] A. E. Monge and C. P. Elkan. An efficient domain independent algorithm for detecting approximately duplicate database records. In *SIGMOD DMKD*, 1997.

[18] H. B. Newcombe. *Handbook of record linkage: methods for health and statistical studies, administration, and business*. Oxford University Press, Inc., New York, NY, USA, 1988.

[19] H. B. Newcombe and J. M. Kennedy. Record linkage: making maximum use of the discriminating power of identifying information. *Commun. ACM*, 5(11):563–566, 1962.

[20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIGKDD*, Edmonton, Alberta, 2002.

[21] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8):635–656, 2001.

[22] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina. Entity resolution with iterative blocking. Technical report, Stanford University, 2008.

[23] W. Winkler. Overview of record linkage and current research directions. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 2006.

[24] W. E. Winkler. Approximate string comparator search strategies for very large administrative lists. Technical report, US Bureau of the Census, 2005.

[25] W. Yancey. Bigmatch: A program for extracting probable matches from a large file for record linkage. Technical report, US Bureau of the Census, 2002.