Bufoosh: Buffering Algorithms for Generic Entity Resolution

Hideki Kawai Hector Garcia-Molina Omar Benjelloun
Tait E. Larson David Menestrina Suttipong Thavisomboon
Stanford University
{hkawai, hector, benjello, telarson, dmenest, sthavis}@stanford.edu

Abstract

Entity Resolution (ER) is a problem that arises in many information integration applications. ER process identifies duplicated records that refer to the same real-world entity (match process), and derives composite information about the entity (merge process). Even though the cost of the match process is high, the cost of disk I/O is likely to be the dominant problem for a limited memory or very large record set. In this paper, we proposed buffering algorithms for ER, Bufoosh, based on lazy disk update and locality-aware match scheduling. Our evaluation results using Yahoo! shopping data show that the algorithms can reduce disk I/O dramatically.

1. Introduction

Entity Resolution (ER) is a problem that arises in many information integration applications. ER (also known as deduplication or record linkage) uses two functions, *match* and *merge*. The match function identifies duplicated records that refer to the same real-world entity, e.g. the same person, the same product, or the same organization. The merge function derives composite information about the matched entities. For example, customer databases from different business divisions may contain multiple records representing the same person, but each record may be slightly different, e.g., with different spellings of the name or address, or missing some information.

There are many potential applications of ER, not only data cleansing for customer relationship management [13], but also price comparison search for online shopping [4], epidemiological statistics for biomedical research [19] and data mining for counter terrorism [12]. The match and merge functions in ER are often application dependent. For example, consider matching a person's name: some users may think two names that sound alike, such as "Quadaffi" and "Kadafi", should be merged, others may want to use canonical dictionaries for example to map "Bobby" to "Robert". When matching organization names, acronym correspondence is important in some cases, e.g. "Defense Advanced Research Projects Agency" would be considered the same as "DARPA". Similarly, one may want to include the department name of the organization in the matching, and the other may not. In the first case, "Stanford School of Engineering" and "Stanford School of Law" shouldn't match, but in the latter case, the two names should match as "Stanford". The same is true about merge function. Some users may want to combine different values into one value, but others may want to keep every possible value from the source records.

A majority of the traditional work on ER has focused on a specific application, e.g., resolving bibliographic data or customer records. So it is often hard to separate the application-specific techniques from the ideas that could be used in other contexts. To clearly separate the application details from generic ER processing, our ER framework approach is to treat the match and merge processes as black-box functions of a platform system. For example, the match function takes as input two records, and decides if they represent the same real-world entity. The match function may compute similarities between the two records. If two records match, the merge function is called to generate a new composite record.

To illustrate our approach, and to illustrate some of the challenges faced, consider the following simple example: Figure 1 shows one possible generic ER outcome for this example. Suppose that our black-box match function returns a boolean value for the input record pair, e.g.., it returns *true* if name and address of two records are similar enough or email address of the records are exactly the same. For merging records, the black-box merge function combines the names and addresses into a "normalized" representative, and performs a set-union on the emails, telephone numbers, and social security numbers.

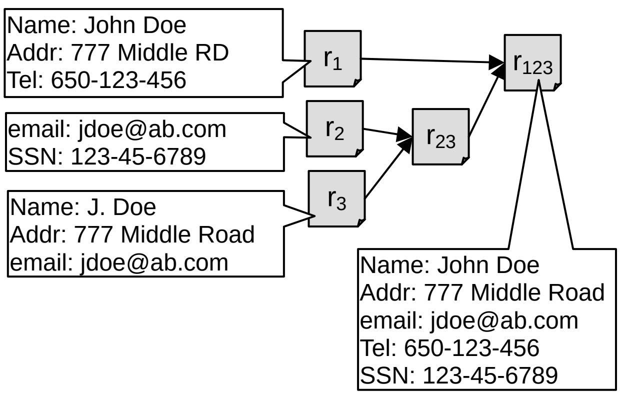

Figure 1: A sample of ER process

For our example, the match function compares r_1 and r_2 and returns *false* because the records do not share information with each other. However the match function returns *true* for the pair of r_2 and r_3 because they have exactly the same e-mail addresses. Thus the merge function combines these records into one record r_{23}. Then r_1 can match with r_{23} because r_{23} has similar name and address with r_1. Finally, we can get integrated information about John Doe by merging r_1 and r_{23} into r_{123}. Note that every time two records are merged, the combined record needs to be re-compared with "everything else", because a merged record has more information, and it can yield an initially unforeseen merge like r_1 and r_{23}, as shown in Figure 1. Of course, the final results can be inconsistent depending on the order of match and merge if we chose inappropriate match and merge functions. Therefore, we will discuss properties for match and merge functions which lead to a consistent result (Section 3).

Treating very large data or processing in the limited memory on the single processor is also an important problem. If we have semantic knowledge about a record set, we can split an ER process into small pieces, e.g., partitioning records by category and then only try comparison within in the same category. Of course, in some applications we may not have such knowledge, or we may not assume that the categorization is perfect. And even with the perfect semantic knowledge, there may still be too many records because the resulting categories are still relatively too large to fit within the limited memory of single processor. In this context, disk I/O cost becomes more serious than record comparison cost. Thus, we have to develop buffering algorithms for a wide range of applications. In this paper we propose buffering algorithms for ER, Bufoosh, based on lazy disk update and locality-aware match scheduling. Our results of evaluation using Yahoo! shopping data show that the algorithms can reduce disk I/O dramatically.

In summary, the contributions we make are:

- We present a definition of generic entity resolution, and show how we model the ER process.(Section 3)
- We define the problem of buffered entity resolution for the case where black-box match and merge functions are used, and we design a memory layout for buffering algorithms. (Section 4.1)
- We present a generic, buffering algorithm, Bufoosh that runs ER on the limited memory of a single processor using lazy update and locality-aware match scheduling. (Section 4.2)
- We evaluate the algorithms using real data and quantify the performance gains. (Section 5)

2. Related Work

Originally introduced by Newcombe et al. [18] as “record linkage” and formalized by Fellegi and Sunter [9], the ER problem was then studied under various names, such as Merge/Purge [11], deduplication [20], reference reconciliation [8], object identification [23], and others. Most traditional works focused on only “matching” problems, i.e. accurately finding the records that represent the same entities [10, 27]. There are mainly two approaches, accuracy oriented and performance oriented.

For an accuracy oriented approach, the matching function can be split into two phases: distance measurement and classification model. Distance measurement calculates the similarity between the atomic values of two records. Examples of distance measurement algorithms include edit distances [22], Jaccard similarity [15], soundex code [18], TF-IDF [7], q-grams [5], and so on. The classification model receives results from distance measurement functions and makes decisions for records. A simple example of a classification model is just to use a threshold for a certain distance value. For more sophisticated techniques, machine learning approaches are often exploited such as Bayesian networks [24], decision trees [14], SVMs [3], conditional random fields [21], and active learning [16]. These models would need a training set labeled by the user. However, unsupervised models are also available such as the EM algorithm [25] and clustering [17,6].

Performance oriented approaches have been focused on how to effectively eliminate record comparisons using blocking techniques by sorting or indexing. Blocking methods efficiently select a subset of record pairs for subsequent similarity comparison, ignoring the remaining pairs as highly dissimilar. A number of blocking algorithms have been proposed [1, 15, 11]. However, in most cases, these methods are based on strong assumptions for a specific application and provide only an approximate result. So even for a performance oriented approach, one must treat accuracy as an inseparable problem.

In contrast, our generic approach focuses on optimizing match and merges which can provide complete results for arbitrary conditions designated by users. A benefit of our approach is that we can treat accuracy and performance issues separately, i.e., users can choose any match and merge functions (and blocking functions if needed) based on their purpose and desired accuracy, while the system can provide the common optimization method which doesn't violate the user's conditions.

In the data base management system literature, there are several buffering algorithms to execute join effectively. Our basic idea departs from existing techniques in this area but we studied memory layout for merged records that traditional join algorithms didn't need to consider.

3. Generic Entity Resolution

We first define our generic setting for ER; (additional details can be found in [2]). We are given a set of records $R = \{r_1, ..., r_n\}$ to be resolved. The *match function* $M(r_i, r_j)$ returns *true* if records r_i and r_j are deemed to represent the same real-world entity. As shorthand we write $r_i \approx r_j$. If $r_i \approx r_j$, a *merge function* generates $<r_i, r_j>$, the composite of r_i and r_j.

As discussed in [2], if match and merge functions have the following four properties, we can discard the source pair of records as redundant information after a merge. Furthermore, if the properties hold, then ER process is *consistent*, in the sense that it is finite and does not depend on the order in which matching records are combined and redundant records discarded.

- *Commutativity*: $\forall\ r_1, r_2,\ r_1 \approx r_2$ iff $r_2 \approx r_1$, and if $r_1 \approx r_2$ then $<r_1, r_2> = <r_2, r_1>$.
- *Idempotence*: $\forall r,\ r \approx r$ and $<r, r> = r$.
- *Merge representativity*: If $r_{12} = <r_1, r_2>$ then for any r_3 such that $r_1 \approx r_3$, we also have $r_{12} \approx r_3$.
- *Merge associativity*: $\forall\ r_1, r_2, r_3$ such that both $<r_1,<r_2, r_3>>$ and $<<r_1,r_2>,r_3>$ exist, $<r_1,<r_2,r_3>> = <<r_1,r_2>,r_3>$

A basic procedure for generic ER, R-Swoosh [2], is illustrated in Figure 2. R-Swoosh uses two record sets *R* and *R*' which represent the initial record set and the non-matching record set, respectively. The basic processing flow is as follows:

(1) Pick one record from *R* as a target record, and remove it from *R*.
(2) Compare the target record with every record in *R*'.
(3) If there are matching records, merge them and generate a new record, and put it in *R*. At the same time, discard the two parent records.
(4) If there are no matching records, add the target record to *R*'.
(5) Go back to (1) and repeat the above processes until *R* becomes empty.
(6) Return *R*' as a result of ER(*R*).

In Figure 2(a), in the initial state, there are four records in *R* and no records in *R′*. We now walk through the execution of R-Swoosh. First, we pick up r_1 as a target record, and move it to *R′* because there are no matching records in *R*'. Second, we pick up r_2 as a target record, and say, r_1 in *R*' does not match with r_2; then r_2 can move to *R*'. Next, r_3 becomes a target record, and say, it does not match with r_1 but does match with r_2 (Figure 2 (b)). Then r_2 and r_3 are merged into new record r_{23}; the resulting r_{23} is put in *R*. Immediately, we can eliminate r_2 and r_3 as redundant information. Next, we pick up r_{23} as a target record and compare it with r_1 in *R*'. If the match function returns true (Figure 2 (c)), we merge them into r_{123} and put the new record in *R*, and eliminate r_{23} and r_1. If r_{123} and r_4 do not match, these records go to *R*' and *R* becomes empty (Figure 2 (d)). The algorithm returns the non-matching record set *R*' as the result of the ER process.

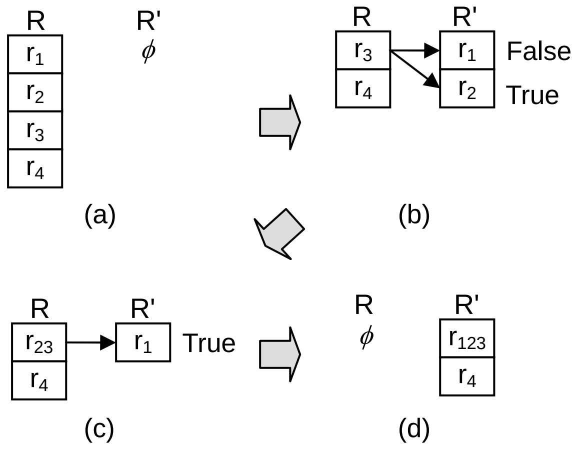

Figure 2: Generic ER process

A benefit of our generic approach is that we can discuss accuracy and performance issues separately. In our framework, users can choose match and merge functions based on their desired accuracy, because our system provides an optimized execution method which does not violate user-given accuracy. Thus our research goal is developing an optimization method which can provide complete results for given black-box match and merge functions.

In this paper we mainly focus on buffering optimization to handle very large data sets or for processing with the limited memory on the single processor. In this context, the disk I/O cost may dominate the record comparison cost. Thus, it is important to reduce the number of disk I/Os, e.g., by considering what kind of disk access can be avoided or how to increase the buffer hit ratio.

4. Buffering for Generic Entity Resolution

In this section, we extend the R-Swoosh to be a buffered algorithm. First, we will discuss a memory layout considering a basic buffering method and its challenges. Second we will propose a buffering algorithm Bufoosh which can reduce the number of disk I/Os. Finally we will discuss some options to tune the proposed algorithms.

4.1 Memory Layout

We assume that the initial record set *R* and the non-matching record set *R′* can be too big to hold all of them in the memory. So we use *R* and *R′* files in disk to keep all record instances of *R* and *R′* respectively. And these files are divided logically into fixed-size blocks that contain instance information of multiple records.

To execute ER process, the system reads some blocks from *R* and *R′* files to compare each other. At the same time, we have to decide which record pair should be compared next. So we need two kinds of spaces in the memory, one is record instance space and the other is utility space.

Record instance space holds the instance information of records from *R* and *R′* files. The record information is moved between disk and memory in entire blocks. The utility space holds record index and comparison queue to manage the order of comparisons. (The details of the index and the queue will be given in the next section.) We also assume that the memory has enough space to hold the utility data because these data do not need whole record instance but only record IDs, and the size of a record ID is much smaller than that of the record instance.

The simplest way for buffering in R-Swoosh is just reading/writing records when needed. To illustrate the challenge of a buffering algorithm, consider an intermediate state as shown in Figure 3. In the figure, all records in *R* and *R*' are on disk. Suppose that memory has two blocks for instance space and that each block can hold two records. Now the memory holds r_1 and r_2 from *R* and s_1 and s_2 from *R*'. First, pick up r_1 as a target record and compare it with s_1 and s_2. If r_1 does not match either one, the system flushes out s_1 and s_2 from memory and reads s_3 and s_4 from the disk to compare with the target record r_1. Again, if r_1 does not match either, r_1 can become non-matching record. In this case, r_1 is removed from the *R* file, and added to the *R′* file. However if r_1 matches with s_3, a new record <r_1, s_3> is generated by the merge and added to the *R* file. In this case, r_1 and s_3 are removed from the *R* and *R′* file respectively, and <r_1, s_3> is added to the *R* file. In the same way, the remaining in *R* can be processed by buffering.

This basic process is very simple but needs many disk I/Os. The main reason is that for every single target record both *R* and *R′* files need to be updated by record operations. As shown above example, there are three kinds of record operations for the *R* and *R′* files, *read*, *remove*, and *add*. Our main idea of memory layout to reduce disk I/Os is putting these operations together by buffers. The relation between record operations and buffers is summarized in Table 1.

For records in the *R* file, we need the *read* operation when picking up a target record. The *remove* operation can be done at the same time, because the target record should be removed independently of that the target record matches a record in *R′* or not. Repeating these operations for each target record one by one takes many disk I/Os, but multiple records can be read and removed from *R* file at one time by an *R* buffer. The *add* operation for the *R* file occur when the target record matches a record in *R*', because the two records should be merged and put into *R*. In this context, we don't need to update the *R* file because memory can keep the merged record in the space that was occupied by the parent records, and the merged record can be next target record as soon as they generated.

For the *R*' file, *remove* operation occurs at the same time as the *add* operation for the *R* file. But removing *R′* records one by one also needs a huge number of disk I/Os. So we solved this problem by making an *R*' index which holds an ID list of a non-matching record set in the utility space, and updating only this index instead of *R′* file. When *remove* operation occurs, the ID of the *R′* record is removed from *R′* index, but the instance of the *R′* record is not removed from *R′* file. So *R′* file can contain duplicate records. However we can get the correct non-matching records by filtering *R′* file with the ID list of *R*' index at the end of the process. The *add* operation for the *R*' file occurs when a target record did not match any of the records in *R*' and it becomes a new *R*' record. The buffer for this new *R*' record has two advantages. One is multiple records in new *R′* buffer can be added to *R′* file at one time. The other is the new *R*' records are likely to match with the next target record especially if *R* records are sorted by a key attribute. The *read* operation for the *R*' record occurs when comparing a target record *r* with record *r*' in *R*' but the instance of *r*' is not in the memory. In this case, a buffer for the *R*' record can reduce disk I/O by reading multiple records from *R*' file at one time.

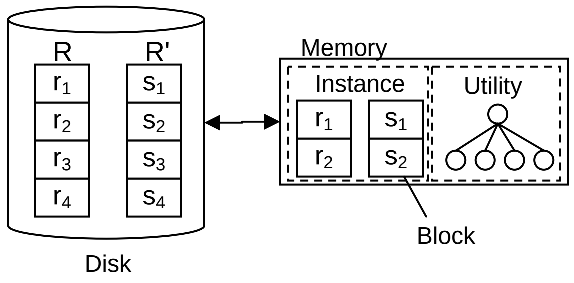

Figure 3: An example of basic buffering

Table 1: Record operation and buffers

	Operation	When	Buffer
R	Read and Remove	Picking up a target record	R buffer
	Add	Merging records	No need
R'	Remove	Merging records	No need
	Add	No r' in R' match with a target record	new R' buffer
	Read	Trying M(r, r')	R' buffer

Another issue of a buffering algorithm is that actual record size is not fixed and it will grow depending on a merge function. In Figure 3, we assumed that one block of memory can contain two records to simplify the illustration. But in reality, we have to consider that even block size is fixed but the number of records which can be kept in one block will change depending on the size of the records. If the size of a record becomes bigger than the size of a block, we trim the information of the record so that the record size does not exceed the block size.

4.2 Bufoosh

Based on the memory layout discussed above, Bufoosh exploits the *R*, new *R'*, *R'* buffers and *R'* index to reduce disk I/O. In this section, we propose two algorithms, a lazy update and a locality-aware lazy update. We will describe these algorithms by giving pseudo code, and show how these buffers and index work. However we do not provide a detailed buffer management policy for each buffer here. We will discuss buffer policies as options for the algorithms in Section 4.3.

(A) Lazy update algorithm

In the lazy update algorithm, the order of record comparison is basically the same as in the generic ER process explained in Section 3. The pseudo code for the lazy update algorithm is given in Figure 4 and the summary of procedures in Figure 4 is given in Table 2. Initially, input record set *R* is given as a file on disk. The system makes an empty file, *tempR*', to temporarily keep instances of *R*' records. As mentioned in the previous section, *tempR*' is a not-updated record set of *R*' to which new *R*' records will be added but no records will be removed even if a merge occurs. The system also initializes *R'Index* as empty.

Procedure *Rbuffer.read*() reads records from the *R* file on disk until the total size of records fill *Rbuffer*. Then procedure *Rbuffer.remove*() removes a target record *r* from *Rbuffer*. The *buddy* which holds a record that matches to the target record *r* is set to null at first. Next, *R'Index.getCand*(*r*) returns an ID list of candidate records which may match with the target record *r*. In the case that we can not exploit any domain knowledge, *R'Index.getCand*(*r*) returns all record IDs included in *R'Index*. However if users have designed sub match functions like 'exact match' or 'range value', *R'index.getCand*(*r*) can limit the number of candidates. Next, the system picks up one record ID *r'ID* from the list of candidates, and *Buffers.have*(*r'ID*) searches the instance of *r'ID* in the memory. If *Buffers.have*(*r'ID*) returns *false*, *R'buffer.read*(*r'ID*) reads a block from the *tempR'* file which contains a record whose ID is *r'ID*. Then the procedure *Buffers.get*(*r'ID*) gets the instance of *r'ID* and assigns it to a candidate record *r'*. Next, the system compares the target record *r* and the candidate record *r*' by M(*r*, *r*'). If these two records do not match, try

```
input: Initial record set R
output: Non-matching record set R'

Initialization:
make an empty file as tempR';
R'Index ← $\phi$;

// main loop
while(Rbuffer.read()){
  while(Rbuffer.recordNum() > 0){
    r = Rbuffer.remove();
    buddy = null;
    candIDs = R'Index.getCand(r);

    // match try loop
    foreach(r'ID in candIDs){
      if(Buffers.have(r'ID) = false){
        R'buffer.read(r'ID);
      }
      r' = Buffers.get(r'ID);
      if(M(r, r') = true){
        buddy = r;
        break;
      }
    }
    if(buddy = null){
      newR'buffer.add(r);
      R'index.add(r);
    }else{
      newR'buffer.remove(buddy);
      R'buffer.remove(buddy);
      R'Index.remove(buddy);
      mr = <r, buddy>;
      Rbuffer.add(mr);
    }
  }
}

Termination:
outputInstances(R'Index);
```

Figure 4: The lazy update algorithm

another candidate record. If these records match, *buddy* holds the candidate *r*' and breaks "match try loop". After the loop, if *buddy* is still null, it means that the target record *r* does not find any matching records in *R*'. So *r* is added to new *R*' buffer and *R'Index* by *newR'buffer.add*(*r*) and *R'Index.add*(*r*), respectively. But if *buddy* holds an instance of record, it means that the target record *r* matched *buddy*. So the system removes *buddy* from *R*' buffer, new *R*' buffer and *R'* index by *R'buffer.remove*(*buddy*), *newR'buffer.remove*(*buddy*) and *R'Index.remove*(*buddy*), respectively. In this case, *R'buffer.remove*(*buddy*) and *newR'buffer.remove*(*buddy*) do nothing if *buddy* is not in each buffer. And the system generates merged record *mr* by <*r*, *buddy*>. Then, *Rbuffer.add*(*mr*) adds *mr* to *R* buffer.

If *R* buffer becomes empty by iterating match try loop for every target record in *R* buffer, read the next subset of

R records, and iterate the above process. After the main loop, *R*' index holds all *R*' record IDs. Finally, we can get all instances of *R*' records by *outputInstances*(*R'Index*) which reads *tempR'* file and output record instance if its ID is included in *R'Index*.

Table 2: A summary of Procedures in the lazy update algorithm

Rbuffer	
add(r)	Add record r to Rbuffer
read()	Read records from top of the R file until Rbuffer become full
recordNum()	Return the number of records in Rbuffer
remove()	Remove a record from Rbuffer
R'buffer	
read(rID)	Search tempR' file and read a block containing an instance of record ID
remove(r)	Remove record r from R'buffer
newR'buffer	
add(r)	Add record r to newR'buffer
remove(r)	Remove record r from newR'buffer
Buffers	
get(rID)	Return the instance of record ID in any of three buffers
have(rID)	Return true, if the instance of record ID is kept in any of three buffers
R'Index	
add(r)	Add record r to R'Index
getCand(r)	Return record IDs that should be compare with the target record
remove(r)	Remove record r from R'Index

(B) Locality-aware lazy update algorithm

The locality-aware lazy update algorithm also uses the same memory layout as the lazy update algorithm. This algorithm does not mind the order of target record, but tries to compare as many record pairs as there are in memory. So the algorithm uses a queue of record pairs to keep which record pairs have already compared. To illustrate this algorithm and how it differs from the previous algorithm, an example of a record-pair queue is shown in Table 3. Suppose that the *R* , *R*' and new *R*' buffer consists of one block each, and a block can hold up to two records. Let's assume that, all buffers are empty. Next, *R* buffer reads r_1 and r_2 from the *R* file. And say, there are six records s_1 to s_6 in the *R*' index. If say we do not use any domain knowledge, then in Figure 3, all s_1 to s_6 are listed as candidate records for both r_1 and r_2. And in Table 3, if *R*' records are grouped together by parentheses, it means they are located in the same block in the *tempR*' file.

Table 3: An example of record pair queue

R	R'
r_1	(s_1, s_2), (s_3, s_4), (s_5, s_6)
r_2	(s_1, s_2), (s_3, s_4), (s_5, s_6)

The locality-aware lazy update algorithm tries to compare record pairs in a vertical direction in this queue. For example, the *R*' buffer reads a block which contains the instance of s_1 and s_2 from the *tempR'* file. Then, the system executes the match function for four record pairs: M(r_1, s_1), M(r_2, s_1), M(r_1, s_2) and M(r_2, s_2). If they do not match, the *R*' buffer reads a block that contains s_3 and s_4 from the *tempR'* file, and the system tries four more pairs: M(r_1, s_3), M(r_2, s_3), M(r_1, s_4) and M(r_2, s_4). If these pairs do not match either, the *R*' buffer reads s_5 and s_6, and the system tries four more pairs: M(r_1, s_5), M(r_2, s_5), M(r_1, s_6) and M(r_2, s_6). If these pairs do not match, r_1 can go to the new *R*' buffer, at the same time, r_1 should be added to the record pair queue. Next, the system tries M(r_2, r_1), if these pairs do not match, r_2 can go to the new *R*' buffer. Now the new *R*' buffer is full, so r_1 and r_2 should be flushed out of the buffer and be written into the *tempR*' file. No pairs remain in the queue, so the *R* buffer reads the next block from the *R* file. In this case, the number of disk I/Os for *tempR'* file is 4 (3 reads and 1 write).

Comparing this locality-aware lazy update algorithm with the lazy update algorithm mentioned above tries to compare record pairs horizontally in the queue, i.e. it first tries all pairs with a target record r_1, M(r_1, s_1), M(r_1, s_2), M(r_1, s_3), M(r_1, s_4), M(r_1, s_5) and M(r_1, s_6). Then it tries all pairs with r_2, M(r_2, s_1), M(r_2, s_2), M(r_2, s_3), M(r_2, s_4), M(r_2, s_5) and M(r_2, s_6). In this case, the number of disk I/Os for *tempR*' is 7 (6 reads and 1 write).

One may wonder what happens if a target record finds a matching record in the queue for the locality-aware lazy update algorithm. For example, say, record r_1 matches s_3. Then, after trying M(r_1, s_3), the system removes the row of r_1, and removes all s_3 from the *R'* column in the queue, and adds <r_1, s_3> plus candidates of the merged record to the queue. In this case, the record pair queue becomes as shown in Table 4, and the system tries M(r_2, s_4) and M(<r_1, s_3>, s_4) because the instance of record s_4 is already in memory.

Table 4: An example of record pair queue when a match has happen

R	R'
r_2	(s_4), (s_5, s_6)
< r_1, s_3>	(s_1, s_2), (s_4), (s_5, s_6)

The pseudo code for the locality-aware lazy update algorithm is shown in Figure 5 and the summary of additional procedures for queue is given in Table 5. In this algorithm, the system maintains *Queue*, in addition to the three buffers and the *R'* index that are also used in the lazy update algorithms. In the main loop, after *Rbuffer* reads the records from *R* file, *R'Index.getCand*(*r*) gets a list of candidate record IDs of the *R'* record set. Now, the procedure *Queue.addRow*(*r.ID*, *candIDs*) makes a new row in *Queue*, puts the ID of target record *r* in the *R* column and puts the ID list of candidates in the *R*' column. In the queue loop, if the memory holds no *R'* record instances which are needed to compare record pairs in *Queue*, *Queue.getOneCand*() picks up one *R'* record ID in the *Queue* randomly and *R'buffer* reads a block which contains the ID in the *tempR'* file. Next, for all records in the *R* column of *Queue*, the system tries comparisons with candidates whose instance is in the memory. At the same time *Queue* removes candidate record ID which is compared with target record *r*. If any candidates did not match with a target record *r*, and no candidates for *r* remain in *Queue*, *r* can become a new *R*' record. Then *r* is added to the new *R'* buffer and *R'Index* by *newR'buffer.add*(*r*) and *R'Index.add*(*r*), respectively. The ID of *r* is added in the *R'* column in *Queue* by *Queue.addCand*(*r*). But if a match occurred, *buddy* holds the matching *R'* record. Then *buddy* is removed from the *R*' buffer, the new *R*' buffer and the *R'* index in the same way as that of the lazy update algorithm. *Buddy* is also removed from the *R'* column of *Queue*. Additionally, the system generates merged record *mr* by <*r*, *buddy*> and adds *mr* to the *R* buffer by *Rbuffer.add*(*mr*). It also makes a new row in *Queue* for *mr* and its candidates by *Queue.addRow*(*mr*, *candIDs*). After the main loop, the system can output the *R'* record set in the same way as that of the lazy update algorithm.

Table 5: A summary of Procedures for Queue in the locality-aware lazy update algorithm

Queue	
addCand(r)	Add an ID of record r to the R' column for all entries
addRow(rID, r'IDs)	Create a new entry, and set rID and r'IDs to the R and the R' column, respectively
isEmpty()	Return true if the queue has no entries.
getOneCand()	Return a record ID in R' column
getCands(rID)	Return a record ID list of the R' column for the entry rID
remove(rID, r'ID)	Remove r'ID from the R' column of the entry rID
removeCand(r')	Remove an ID of r' from the R' column of all entries

```
input: Initial record set R
output: Non-matching record set R'

Initialization:
make an empty file as tempR';
R'Index ← ϕ;
Queue ← ϕ;
// main loop
while(Rbuffer.read()){
  // make a queue
   foreach(r in Rbuffer){
      candIDs = R'Index.getCand(r);
      Queue.addRow(r.ID, candIDs);
   }
   // queue loop
   while(Queue.isEmpty() = false){
      if(no candidates in memory){
         r'ID = Queue.getOneCand();
         R'buffer.read(r'ID);
      }
      //match try loop
      foreach(rID in Queue.keys){
         r = Buffers.get(rID);
         buddy = null;
         candIDs = Queue.getCands(rID);
         // match try loop
         foreach(r'ID in candIDs){
            if(Buffers.have(r'ID) = false){
               continue;
            }
            r' = Buffers.get(r'ID);
            Queue.remove(rID, r'ID);
            if(M(r, r') = true){
               buddy = r';
               break;
            }
         }
         if(buddy = null){
            if(Queue.getCands(rID)= null){
               newR'buffer.add(r);
               R'index.add(r);
               Queue.addCand(r);
            }
         }else{
            newR'buffer.remove(buddy);
            R'buffer.remove(buddy);
            R'Index.remove(buddy);
            Queue.removeCand(buddy);
            mr = <r, buddy>;
            candIDs = R'Index.getCand(mr)
            Rbuffer.add(mr);
            Queue.addRow(mr, candIDs);
         }
      }
   }
}
Termination:
outputInstances(R'Index);
```

Figure 5: The locality-aware lazy update algorithm

4.3 Options for Bufoosh

In this section, we will discuss two options for the previous algorithms. One is initial record sorting, and the other is buffer management policy.

The initial record sorting is a kind of optimization. If the match function designated by the user includes an exact match or range query for a certain attribute, sorting initial records by the attribute can localize potentially matching records in the neighborhood. If the initial record set is stored in disk statically, we can sort completely the initial record set before executing the ER. Even if the initial record set is dynamic such as in a data stream, we can exploit a partial sort within a certain length of window.

Bufoosh uses three buffers, *R*, *R'* and new *R'*. These buffers have different functions, so the management policies should be different. Management policies for these buffers are summarized in Table 6. *R* buffer reads a subset of records from *R* file repeatedly, and records in *R* buffer are removed almost monotonically in the main loop. Therefore a simple management policy such as first in first out (FIFO) is enough for an *R* buffer.

Table 6: Management policies for three buffers

Buffer	Management Policies
R	FIFO
R'	LRU / SORT
new R'	LRU / SORT

R' buffer reads a record block which includes an instance of candidate record in the *tempR'* file. If *R'* buffer is full when we need a new block not in memory, we have to replace an old block with the new one. So the least recent used (LRU) policy is good for *R'* buffer management. In another respect, if the initial records are sorted, sorting the records in the *R'* buffer in the same order as that of the initial records may keep the locality of initial records even in *tempR'* file. SORT is based on this idea, and it flushes out a block that has a record whose attribute value is minimum in the *R'* buffer if the initial records are sorted ascending order.

A record in the *R* buffer goes to the new *R'* buffer when it does not have any matching records in *R'*. If the new *R'* buffer is full, we have to flush out one of blocks and write it into *tempR'* file in the disk. Thus LRU and SORT can be good for new *R'* buffer as well as *R'* buffer.

5. Experiment

We implemented the Bufoosh algorithm, and conducted extensive experiments on subsets of a comparison shopping dataset from Yahoo!. In this section, we describe our implementation and experimental setting, and report the main findings of our experiments.

5.1 Experimental Setting

We ran our experiments on subsets of a comparison shopping dataset provided by Yahoo!. The dataset consists of product records sent by merchants to appear on the Yahoo! shopping site. It is about 6GB in size, and contains a very large array of products from many different vendors. Such a large dataset must be (and is in practice) divided into groups for processing. For our experiments we extracted subsets of records of varying size (from 5,000 to 20,000), representing records with related keywords in their titles (e.g., iPod, mp3, book, DVD). Each of these smaller datasets represents a possible set of records that must be resolved. If we are modeling a scenario with no semantic knowledge, then all records in the dataset must be compared. If we are considering a scenario where semantic knowledge is available, the database can be further divided by category. In this experiment, we assumed that users divide an input data set based on semantic knowledge and our system executes ER on the devided set.

Our base case scenario is summarized in Table 7. In each experiment, some of these conditions are modified to study the impact of these conditions. We will notice modified conditions in following sections, but do not notice explicitly the same condition as this base case.

Our match function considers two attributes of product records, the title and the price. We used the Jaro-Winkler similarity measure [26] to compute the edit distance *EditDist*(r_i.*title*, r_j.*title*). The function returns a higher score if two strings are closer to each other. *FracDist*(r_i.*price*. r_j.*price*) returns a numeric fraction between two prices which is computed as |r_i.*price* - r_j.*price*| / max(r_i.price, r_j.price). Note that if r_i is a target record and threshold for *FracDist* is τ_f ($0<\tau_f<1$), r_j such that r_i.price × τ_f < r_j.price < r_i.price / τ_f would match in the latter sub match function. As shown in Table 7, our match function is expressed as a conjunction of distance functions for titles and prices with thresholds. To merge pairs of records, we simply took the union of distinct values for each attribute. And if a merged record has multiple values for titles and prices, we used all values in the record for match function.

We fixed the buffer block size as 50KB. A record instance of input data is about 1 KB in size, so one block can hold 50 records on average. The number of blocks for R, R' and new R' buffer are 3, 4 and 3 respectively, and the buffer management policies are FIFO, LRU and LRU respectively. We also build an *R'* index using price as a key attribute. Window size to limit candidates for comparison is the same value as the threshold of *FracDist* in our match function. Thus if the price of the target record is 100 dollars, records in *R'* whose price is between 80 to 125 dollars are returned as candidates by the *R'Index.getCands*() function in Figure 4 and 5.

We used price and record ID as the key and value of the *R′* Index. The 6GB Yahoo! shopping data contains about 1 million records, and the size of one entry in *R′* Index is at most 128 bit (16 byte). Thus, we can estimate the size of *R′* index to be about 16 MB for the whole Yahoo! data. Note that many products will have the same price, so actual size for *R′* index key could be much smaller. Therefore, it is reasonable to assume the size of *R*' index fits in memory.

Table 7: Base case for experiment

Match function	EditDist(r_i.title, r_j.title) > 0.9 AND FracDist(r_i.price, r_j.price) < 0.8
Merge function	Union
Block size	50KB
# of blocks for each buffer (R,R',new R')	(3, 4, 3)
R buffer policy	FIFO
R' buffer policy	LRU
new R' buffer policy	LRU
R'Index key	price
R'Index window	0.8

Our experimental system is implemented in Java and the system was run on 2.8GHz Pentium processor and 1GB of RAM, and the *R* and *tempR'* files are stored on an EIDE hard disk drive (120GB, 100MB/s, 8MB cache, 7200RPM). To evaluate the performance of Bufoosh, we measured the following metrics:

- Running time: We measured the total wall-clock running time of the main loop in Figures 4 and 5. We also evaluated the details of total running time as follows:
  - The record comparison time to execute the match function *M*(*r*, *r'*)
  - The disk I/O time as the total execution time for *Rbuffer.read*(), *R'buffer.read*(), and *newR'buffer.add*().
  - The overhead time computed as (total time) - (comparison time) - (disk I/O time).
- Number of comparisons: We counted the overall number of comparisons (invocations to match function) performed during main loop. We can derive an average comparison time for one record pair by dividing the total comparison time by the number of comparisons.
- Number of disk I/Os: We counted the number of blocks read or written when *Rbuffer.read*(), *R'buffer.read*(), and *newR'buffer.add*() are executed. Assuming that reading and writing cost is almost the same, we can derive the disk I/O cost per block by dividing disk I/O time by the total number of disk I/Os. Additionally, a buffer hit ratio ρ can be derived by following equation:

$$\rho = 1 - \frac{(\#\,of\ blocks\,read\ by\ R'\,buffer)}{(\#\,of\ comparisons)} \quad ...(5)$$

5.2 Initial record sorting

We started by comparing the performance of the lazy update algorithm and the locality-aware lazy update algorithm, considering the impact of initial record sorting. We ran Bufoosh on 5000 records all of which contain the string "iPod" in their title. Both algorithms of course yield the same ER result, and found 2379 matches.

Figure 6 gives the total running time of the lazy update algorithm (LU) and the locality-aware lazy update algorithm (LALU) for two cases: the initial record set is randomly ordered (RANDOM), or sorted by price attribute (SORT). Table 8 gives a breakdown of the total running time. In Figure 6, we can see significant differences between LU and LALU, RANDOM and SORT. According to Table 8, the differences in comparison time and overhead among the four cases are slight, but disk I/O time is improved dramatically by using the locality-aware algorithm and initial record sorting. For the RANDOM input case, the disk I/O time for LALU is 155 times less than that of LU. Even comparing the SORT case, the disk I/O time of LALU is 95 times less than that of LU. Speaking of the difference between SORT and RANDOM, the disk I/O time became about 9.5 times shorter in LU, and 5.8 times shorter in LALU. While for comparison time, we can see some effect in LALU and SORT, but the difference is relatively small, at most 14%. Note that for LU, the disk I/O time is dominant in total running time, but for LALU the disk I/O time is not dominant any more and comparison time becomes the main cost of ER. Overhead time is always limited and very small.

As explained in Section 4.2, LU reads blocks in the *R'temp* file over and over again to compare all matching candidates for a fixed target record. However, LALU tries the record comparison using the records in three buffers as many as possible. Thus, LALU needs fewer reads from the *R'temp* file than LU. This is the main reason for the big performance improvement in LALU.

As mentioned in Section 4.3, sorting initial records by the attribute can localize potentially matching records in the neighborhood. So in the SORT case, a target record more likely to find the matching candidates in the new *R′* buffer because records in the new *R′* buffer may have similar values to the target record. Therefore SORT can save reads from the *R'temp* file compared to RANDOM.

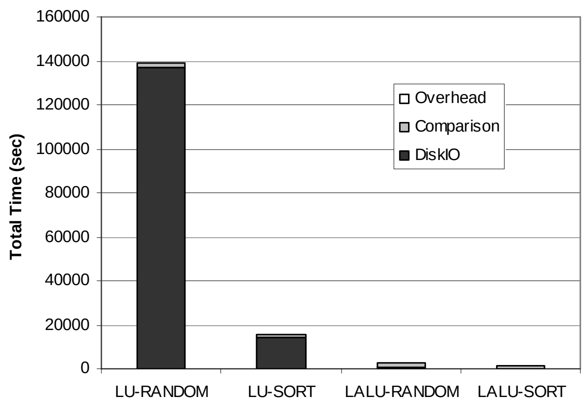

Figure 6: Total running time of Bufoosh for unsorted and sorted records.

Table 8: Breakdown of total running time

	Disk I/O (sec)	Comparison (sec)	Overhead (sec)
LU-RANDOM	137,268	1,405	54
LU-SORT	14,475	1,378	39
LALU-RANDOM	881	1,531	75
LALU-SORT	152	1,320	43

Table 9 gives the number of disk I/Os for *R*, *R'* and new *R'* buffers. The number of *R* buffer reads was the same for all four conditions, because the *R* buffer reads *R* files monotonically. 5000 records can be divided into about 100 blocks because the block size is 50KB and record size is about 1KB on average. The number of new R' buffer writes was also almost the same among the four schemes. This number depends on the order of match and merge, but the actual effect is so slight that we can ignore the difference. The main difference is caused by *R'* buffer reads, and it reflects the tendency of disk I/O time shown in Table 8.

Table 9: Number of disk I/Os for three buffers

	R buffer	R' buffer	new R' buffer
LU-RANDOM	96	712,229	72
LU-SORT	96	74,031	69
LALU-RANDOM	96	5,787	69
LALU-SORT	96	1,098	69

Table 10 gives disk I/O per block and comparison time per record pair under the four schemes. Disk I/O time per record is 150 to 200 times longer than comparison time for per record on average. We can see a tendency that disk I/O become shorter in LALU. This could be due to the effect of file cache controlled by a file system. For LU, the system scans the *R'temp* file over and over again, so if the buffer does not hold an *R*' candidate record for a target record, it is more likely that the file cache does not hold the candidate record, either. However LALU save reads from the *R'temp* file, so even if the buffer does not hold the *R'* candidate record, the file cache can hold the candidate record. We also can see another tendency that RANDOM takes more time for comparison per record than SORT. This can be explained by the following example: Consider there are *p* records, say r_1, r_2,..., r_6, which can match each other and become one merged record, mr^* = <r_1, r_2,..., r_6>. And say, these records are ordered by their price. For a SORT record set, the final process to generate mr^* can be merging r_6 and <r_1, r_2,..., r_5>. If every element of title attributes of the *p* records are identical, this process needs at most 5 executions of distance function, because r_6 has one title, and <r_1, r_2,..., r_5> has five titles. While, for a RANDOM record set, the final process to generate the mr^* can be accomplished by merging 3 merged records and 3 other merged records in the worst case. In this case, distance function for title attribute has to be called 3×3 times in the worst case. Thus, one record pair comparison tends to need a longer time in RANDOM than in SORT.

Table 10: Disk I/O and comparison time per record

	Disk I/O (msec)	Comparison (msec)
LU-RANDOM	193	0.890
LU-SORT	195	0.798
LALU-RANDOM	148	0.909
LALU-SORT	121	0.797

Figure 7 gives the buffer hit ratio under four conditions. The buffer hit ratio is about 0.55 in LU-RANDOM, and it becomes about 0.95 by initial record sort. For the LALU algorithm, the buffer hit ratio was more than 0.99 even for RANDOM records, and it became more than 0.999 for SORT records. This fact revealed that the effectiveness of memory usage in LALU is amazing. Note that the difference between buffer hit ratio of LU-SORT and LALU is very small, however its impact for disk I/O is huge as shown in Table 8 and 9.

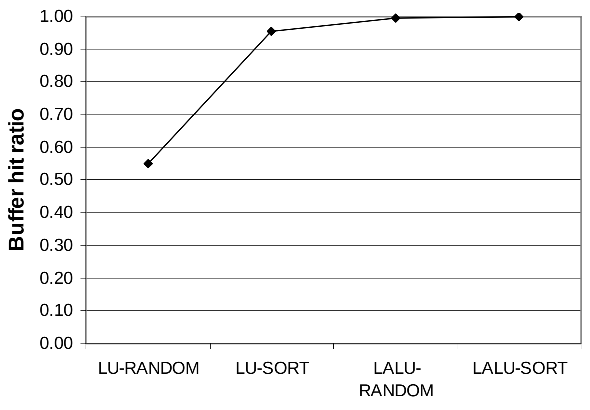

Figure 7: Buffer hit ratio

If the initial record set is statically stored on disk, we can sort all of them before running ER on it. If the initial record set is a continuous data stream, we can use a window to sort initial records partially. Figure 8 gives the running time of LALU changing window size, which is the number of records sorted partially in the initial record set. In Figure 8, RANDOM indicates that initial record is not sorted, and Rbuf-SORT indicates that window size for initial record sorting is the same as the size of *R* buffer. And 1000-SORT, 2000-SORT, and 5000-SORT indicate that window size for initial record sorting is 1000, 2000, 5000 records, respectively. Note that the results of RANDOM and 5000-SORT in Figure 8 are the same as LALU-RANDOM and LALU-SORT in table 8 because the number of records is 5000. In Figure 8, RANDOM and Rbuf-SORT are the almost same. Because LALU compares as many record pairs as in memory, so it does not depend on the record order in the *R* buffer. A partial sort such as the 1000-SORT and the 2500-SORT can save disk I/O time. Even sorting 1/5 of the initial records, disk I/O time becomes less than half of RANDOM.

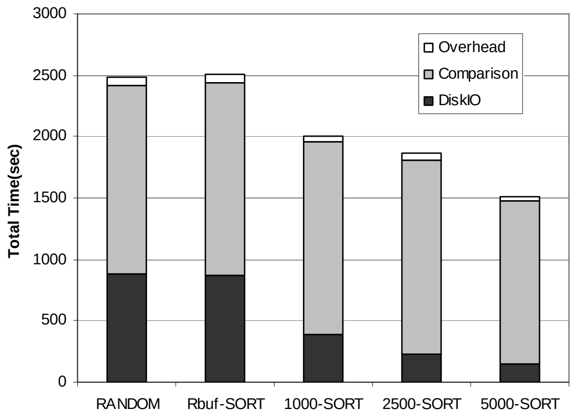

Figure 8: Effect of partial sorting for running time

Figure 9 shows the effect of partial sort for the number of disk I/Os and comparisons. We can see the similar behavior as the Figure 8. The drop of the number of comparison at 5000-SORT is remarkable in Figure 9 however the same tendency is also shown as the shorter comparison time in Figure 8. This fact indicates that the benefit of the initial record sorting does not appear for the record comparison unless the initial records are completely sorted, but we can get the benefit of the initial record sorting for the disk I/O even if the partial sort.

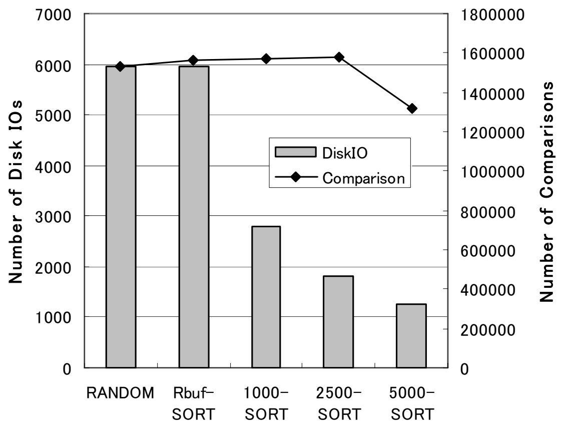

Figure 9: Effect of partial sort for the number of disk I/Os and comparisons

5.3 Buffer management policy

Now we consider the impact of a buffer management policy. Figure 10 shows the running time of LALU for SORT record set changing buffer management policies for *R′* and new *R′* buffer. And Figure 11 shows the number of disk I/Os and comparisons for different buffer management policy. In Figure 10 and 11, LRU-SORT means that management policy of *R′* buffer is LRU and that of new *R′* buffer is SORT, and SORT-LRU means the opposite. Note that the total time of LRU-LRU is the same result as LALU-SORT in Table 8. We can see that the impact of buffer management policies for comparison and overhead time is very small. However the *R′* buffer management policy has a mach bigger impact for disk I/O than the new *R′* buffer policy has. For R' buffer, LRU needs 10 times less disk I/O time than SORT. This can be explained by the following example. Consider that initial records are sorted in ascending order of a key attribute and records in *R′* buffer are sorted in the same way, and *R′* buffer replaces records whose key attribute value is the lowest in the buffer. In this context, records whose key attribute value is higher are replaced rarely, and they tend to stay in the memory for a very long time once they are read in. While records whose key attribute value is lower tend to be replaced easily. Thus, as ER progressed, the most part of the *R′* buffer is occupied by higher records, and needs more replacement to handle lower records. The same thing may happen in the new *R′* buffer if its buffer management policy is SORT. However the number of disk I/Os related to the new R' buffer is very small as mentioned in the previous section. Therefore the impact of the new *R′* buffer management policy is limited. The tendency for the number of disk I/Os and comparisons in Figure 11 are similar to that of disk I/O and comparison time in Figure 10.

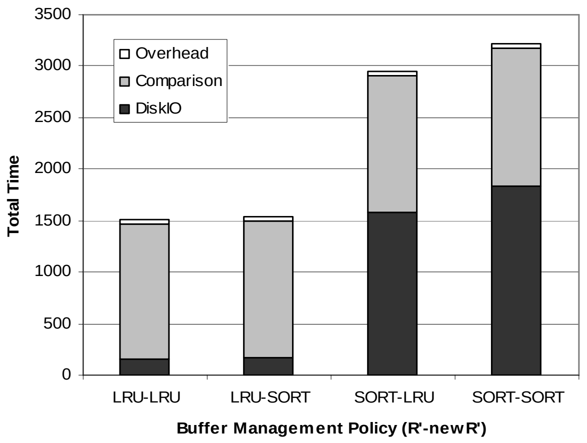

Figure 10: Impact of buffer management policy for running time

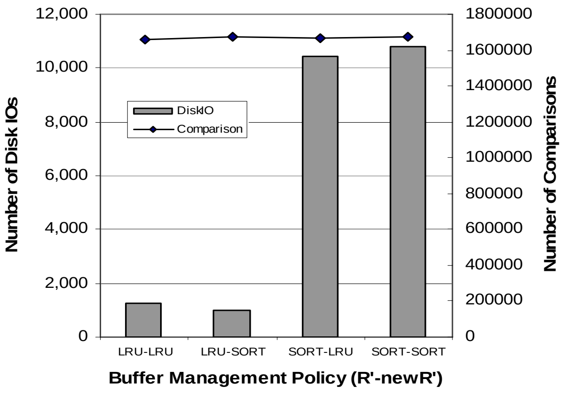

Figure 11: Impact of buffer management policy for the number of disk I/Os and comparisons

5.4 Buffer size and management cost

For a fixed memory size, it is important to find good allocation of three buffers. We studied what allocation of three buffers are the best by fixing the number of one of three buffers and changing the rest of them. We fixed the total number of blocks to ten, and fixed one of three buffers to three blocks. The result shows an important trade-off between the size of buffer and its management cost.

Figure 12 gives the total running time of LALU for SORT record set fixing a new R' buffer to three blocks. And Figure 13 gives the number of disk I/Os and comparisons for the same scenario. Note that the x-axis in Figure 12 and 13 is *R′* buffer size, but it also means that the number of *R* buffer can be calculated by (7 - *R′* buffer size). In Figure 12 and 13, we can not see a clear tendency of comparison time and count among different *R′* buffer sizes. The tendency of disk I/O time among different *R′* buffer sizes is also not clear. However, the number of disk I/Os is obviously reduced as the *R′* buffer size increased. When *R′* buffer is one block, the number of disk I/Os is almost 3000. But when *R′* buffer is six blocks, the number of disk I/Os becomes less than half (1200 disk I/Os). This fact indicates that the larger the size of the *R′* buffer, the smaller the number of disk I/Os, but this also means that the larger the size of the *R′* buffer, the longer the disk I/O time per block. This indicates that there is a trade-off between the size and the management cost of *R′* buffer. Because of this trade-off, running time for a two-block *R′* buffer was the shortest at 1450(sec) when the R' buffer size was two, and the worst running time was 1677(sec) when the *R′* buffer size was six and the *R* buffer size was one.

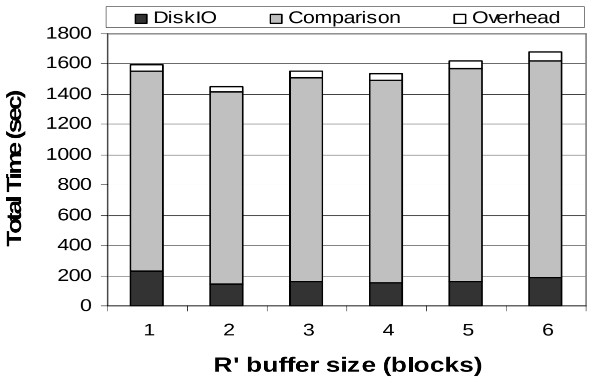

Figure 12: Running time when new R' buffer is fixed to three blocks.

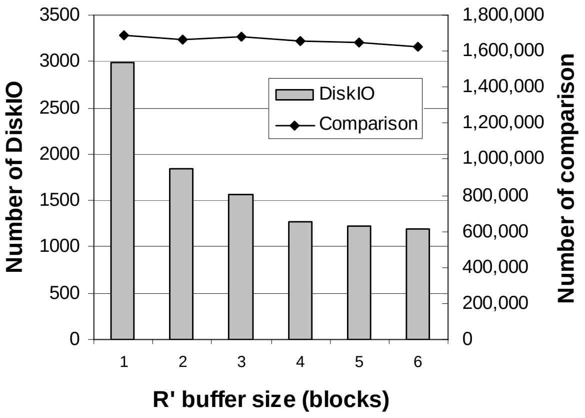

Figure 13: Number of disk I/Os and comparisons when new R' buffer is fixed to three blocks

Figure 14 and 15 give the result of LALU for SORT record set fixing the *R* buffer to three blocks. Note that the number of new *R′* buffer can be calculated by (7 - *R′* buffer size) in Figure 14 and 15. These graphs also show the same behavior as Figure 12 and 13. Comparison cost is almost independent from *R′* buffer size, and even the number of disk I/Os for a one-block *R′* buffer decreases by half for six-block *R′* buffer. So in these graphs, we can see the trade-off between the number of disk I/Os and buffer management cost. In this case, the best running time was 1509 (sec) when *R′* buffer size was two blocks. And the worst running time was 1576 (sec) when R' buffer size was one block.

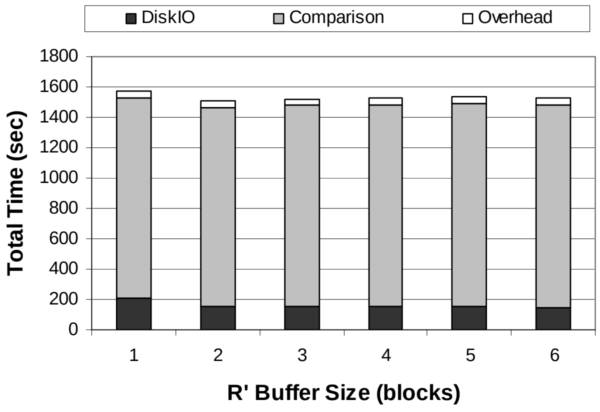

Figure 14: Running time when R buffer is fixed to three blocks.

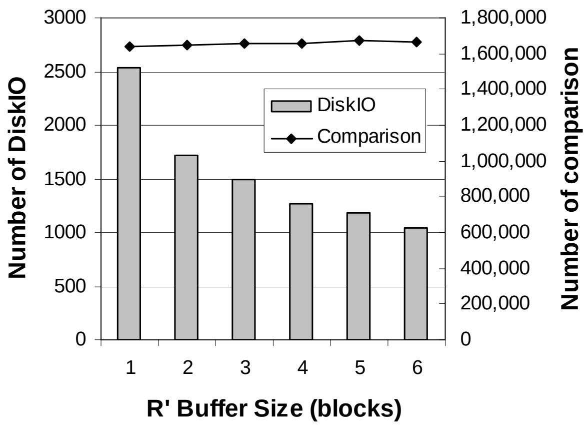

Figure 15: Number of disk I/Os and comparisons when R buffer is fixed to three blocks

Figure 16 and 17 give the result of the LALU for SORT record set fixing the *R* buffer to three blocks. Note that the number of new *R′* buffer can be calculated by (7 - *R* buffer size) in Figure 16 and 17. We can see little behavior difference from previous graphs. In Figure 16, disk I/O time slightly decreased as *R* buffer size increased. But in Figure 17, the number of disk I/O is almost independent from *R* buffer size. This indicates that if new *R′* buffer size becomes larger, the cost of disk I/O increases. In this case, the best running time was 1445 (sec) when the *R* buffer size was five blocks. This was the fastest result in this section. The worst running time was 1709 (sec) when R buffer size was one. So the difference between the best and worst runtime in this section is about 15%.

We can summarize behaviors in different buffer sizes as follows: The differences among the above scenarios are not so large in total run time because of the trade-off between the *R′* or new *R′* buffer size and buffer management cost. And it is good to avoid a memory allocation such of just one block for the *R* or *R′* buffers because in above results the worst case was when the *R′* or *R* buffer was just one block.

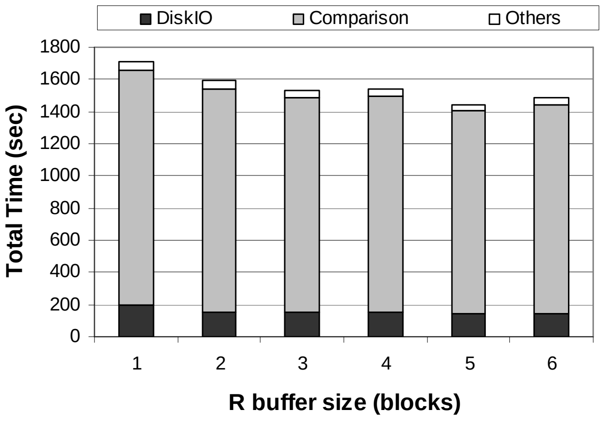

Figure 16: Running time when R' buffer is fixed to three blocks

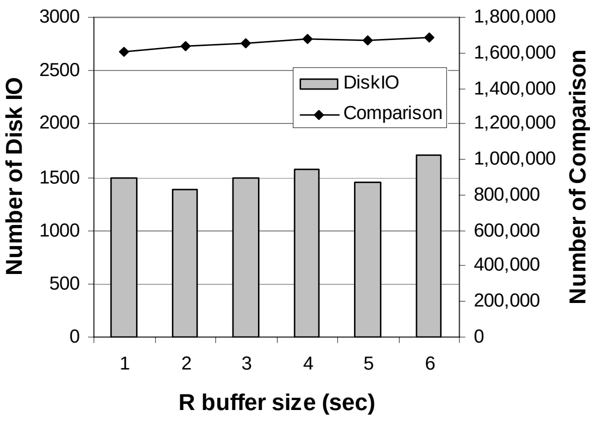

Figure 17: Number of disk I/Os and comparisons when R' buffer is fixed to three blocks

5.5 Index window size

The procedure *R'Index.getCand*(*r*) in Figure 4 and 5 returns candidate records for the target record *r*. We can limit the number of candidate records with semantic knowledge. For example, if the price of the target record is 100 dollars and window size is 0.8, it is not necessary to compare all records in *R′* index with the target record, but it is enough to compare records whose price is between 80 to 125 dollars. Therefore the window size of *R′* index can affect disk I/O time. We conducted some experiments changing *R′* index window size from 0.5 to 0.99 to investigate the impact for disk I/O.

Figure 18 shows changes in the number of matches and comparisons of LALU for SORT record set. In this experiment, we made the threshold of sub match function for price equal to the *R′* index window. Note that larger price threshold indicates fewer candidate records. In Figure 16, both matches and comparisons decrease with an increase in price threshold. The number of matches declined 25% from 2657 to 1982, while the number of comparisons dropped 91% from 4.3 million to 377 thousand. Users can choose *R′* index window size based on their intended accuracy and running time. Note that price threshold 0.8 corresponds to LALU-SORT in Section 5.2.

Figure 19 shows changes in disk I/O and comparison time. Disk I/O dropped 92% from 402 (sec) to 32 (sec), and comparison time also dropped 91% from 3453 (sec) to 302 (sec) by changing the price threshold from 0.5 to 0.99. So the ratio of disk I/O in total time is almost constant to 10% for all window sizes.

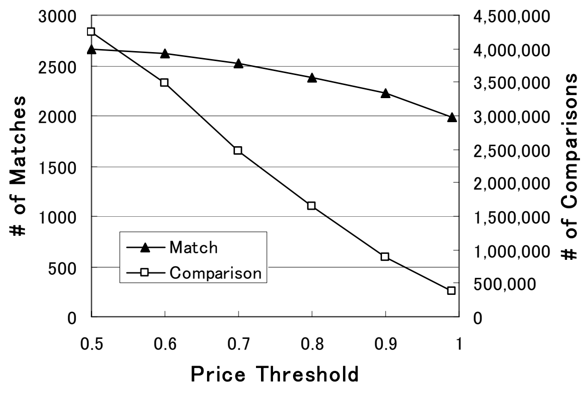

Figure 18: Number of matches and comparisons for different window size

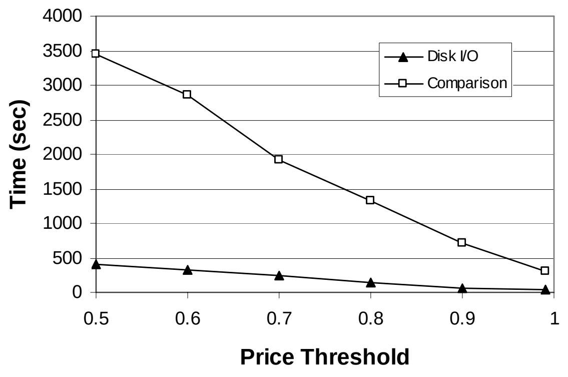

Figure 19: Running time of disk I/O and comparison for different window size (SORT)

For a RANDOM record set, the result shows quite different and interesting behavior. Figure 20 gives the result of a RANDOM record set. In this experiment, the number of matches and comparisons were almost the same as Figure 19 However, as shown in Figure 20, disk I/O time was almost stable and independent from *R′* index window size. The reason for this can be explained as follows. For a RANDOM record set, candidates for a target record can be randomly-dispersed in *tempR′* file. So every block in *tempR′* can contain at least one candidate record for every target record if the number of candidates is greater than the number of blocks in *tempR′* file. Thus, even if the *R′* index window size becomes large, disk I/O time does not increase. And when *R′* index window size is very small (at price threshold 0.99), disk I/O time decrease slightly.

Figure 21 shows the number of disk I/Os for different window size. In this graph, we can see similar tendency to the disk I/O time, i.e., the number of disk I/Os of SORT decreases when the price threshold increases but that of RANDOM is almost stable for the price threshold range from 0.5 to 0.9.

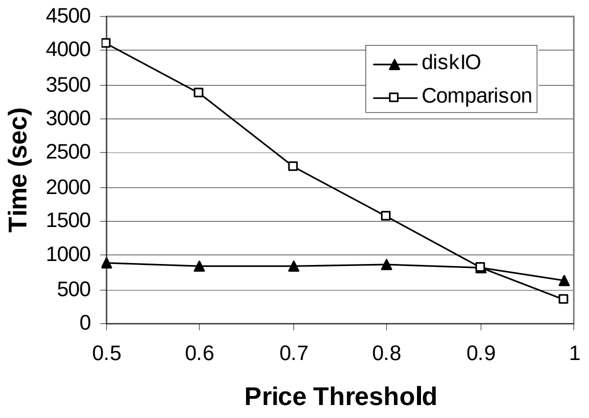

Figure 20: Running time of disk I/O and comparison for different window size (RANDOM)

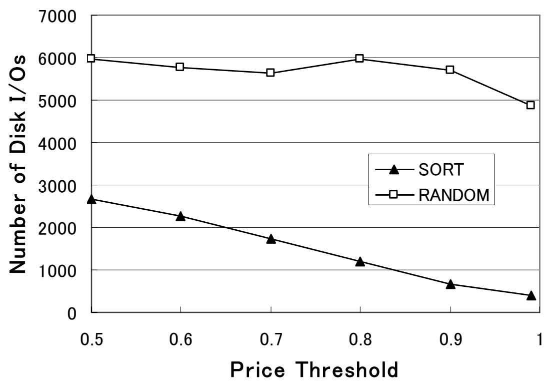

Figure 21: The number of disk I/Os for different window size

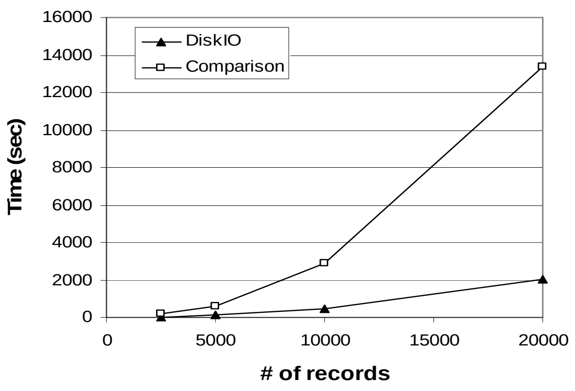

Figure 22: Running time of disk I/O and comparison for different number of records

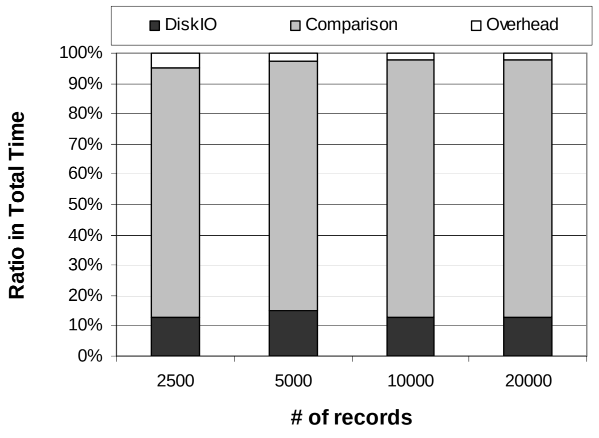

Figure 23: Break down of total running time for different size of record sets

5.6 Scalability

To evaluate the scalability of our algorithm, we compared performance on input datasets of different sizes. In this experiment, we extracted 20,000 records which contain either "iPod", "mp3", "dvd", or "book". We then generated 10,000, 5,000 and 2,500 record sets by randomly removing records from the first set. The main reason why we used not only the word "iPod" but also other words is that there are not enough records whose title contains "iPod". Since the record set used in this experiment consists of products belonging to different categories, the number of matches for 5000 records is fewer than that for 5000 "iPod" records, which are used in previous experiments. But we believe that the basic tendency for scalability is similar, even in the different record sets.

In Figure 22 we show the running time of disk I/O and comparison of LALU for SORT record set as a function of the size of the initial record set. And Figure 23 gives the breakdown of total running time for different sizes of record sets. The cost evolves quadratically with the number of records, and this is an inherent characteristic of ER. However, as shown in Figure 23, this evolution is the same as for the disk I/O time as for the comparison time. Ratio of disk I/O in total running time was always 12-15%.

6. Conclusion and Future Work

Entity resolution is an important problem that arises in many information integration scenarios. A lot of the work to date has focused on how to achieve semantically accurate results, e.g., how to ensure that the resulting records truly represent real-world entities in some application domain. Instead, our generic ER approach treats with accuracy and performance issues separately, and focuses on problems of performance. Given match and merge functions, how can we buffer a large record set in a memory limited single processor? We proposed buffering algorithms for ER, based on lazy disk update and locality-aware match scheduling. These algorithms also exploit initial record sorting to localize matching candidates in memory. Our evaluation results, using Yahoo! shopping data, showed that our locality-aware algorithm can achieve a very high hit ratio and reduce disk I/O dramatically. Our future work will be oriented in three directions: developing distributed algorithms to divide huge numbers of comparisons into multiple processors, designing a platform system integrating various optimization methods, and defining a declarative query language for entity resolution which can be run on the platform system.

References

[1] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *Proc. of ACM SIGKDD'03 Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[2] O. Benjelloun, H. Garcia-Molina, J. Jonas, Q. Su, and J. Widom. Swoosh: a generic approach to entity resolution. Stanford University Technical Report, 2005. Available from http://dbpubs.stanford.edu/pub/2005-5.

[3] M. Bilenko and M. Mooney. Adaptive detection using learnable string similarity measures, In *Porc. of ACM SIGKDD*, 2003.

[4] M. Bilenko, S. Basu, and M. Sahami. Adaptive product normalization: Using online learning for record linkage in comparison shopping. In *Proc. of ICDM-2005*, pages 58-65, 2005.

[5] S. Chaudhuri, K. Ganjam, V. Ganti, and R. Motwani. Robust and efficient fuzzy match for online data cleaning. In *Proc. of ACM SIG-MOD*, pages 313-324. ACM Press, 2003.

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

[7] W. Cohen. Data integration using similarity joins and a word-based information representation language. ACM Transactions on Information Systems, 18, pages 288-321, 2000.

[8] X. Dong, A. Y. Halevy, J. Madhavan, and E. Nemes. Reference reconciliation in complex information spaces. In *Proc. of ACM SIGMOD*, 2005.

[9] I. P. Fellegi and A. B. Sunter. A theory for record linkage. Journal of the American Statistical Association, 64(328), pages 1183-1210, 1969.

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

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

[12] P. Hsiung, A. Moore, D. Neill, and J. Schneider. Alias detection in link data sets. In *Proc. of the International Conference on Intelligent Analysis*, 2005.

[13] IBM. DB2 Entity Analytic Solutions. http://www-306.ibm.com/software/data/db2/eas/.

[14] L. Jin, C. Li, and S. Mehrotra. Efficient record linkage in large data sets, In *Proc. of DASFAA*, 2003.

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

[16] M. Michalowski, S. Thakkar, and C. A. Knoblock. Exploiting secondary sources for unsupervised record linkage. *VLDB Workshop on Information Integration on the Web (IIWeb)*, 2004.

[17] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proc. of SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery*, pages 23-29, 1997.

[18] H. B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130(3381), pages 954-959, 1959.

[19] C. Quantin, H. Bouzelat, F. A. A. Allaert, A. M. Benhamiche, J. Faivre, and L. Dusserre, International Journal of Medical Informatics, 49, pages 117-122, 1998.

[20] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of ACM SIG-KDD*, 2002.

[21] P. Singla, P. Domingos, Object identification with attribute-mediated dependences. In *Proc. of PKDD*, pages 297-308, 2005.

[22] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147, pages 195-197, 1981.

[23] S. Tejada, C. A. Knoblock, and S. Minton. Learning object identification rules for information integration. *Information Systems Journal*, 26(8), pages 607-633, 2001.

[24] V. Verykios, G. Moustakides, and M. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12(1), pages 28-40, 2003.

[25] W. Winkler. Using the EM algorithm for weight computation in the fellegi-sunter model of record linkage. American Statistical Association, In *Proc. of Section on Survey Research Methods*, pages 667-671, 1988.

[26] W. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC, 1999.

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