Record Linkage Data Structures

Evan Harris, ETSU, 2025

I. Background

Record linkage is the process of de-duplicating records as they come into a system. For example, in credit card processing, a record linkage system identifies if two transactions were from the same person. It can be performed "online", where each record is matched in real time and the algorithm cannot know what the future data will be, or "offline" where the algorithm can see an entire batch of records and act accordingly. A record is typically a person, and has fields for First Name, Last Name, etc. which can be compared using string similarities.

I will go over two data structures, disjoint sets and self organizing lists, for solving this problem and analyze how they scale with many records. The Potential Method and amortized costs will be helpful to see the average cost of an operation and whether or not the total cost is within a constant factor of the "optimal cost". There are other approaches for record linkage, such as probabilistic matching, that may be used instead of or in combination with these two data structures. The *online version* can be described as follows.

- **Given**: A sequence of n records $R = [r_1 \dots r_n]$ and database of existing records L
- For each r_i, find all pairs $\{(r_i, x) : r_i \in R, x \in L\}$ where *similarity*$(r_i, x) \geq t$. Where t is a manually provided threshold number based on the data requirements and *similarity* function.
- Insert r_i into L only if there are no pairs found for r_i.

The *offline version* is similar.

- **Given**: A batch B of records and database L
- For each r_i in B, find all pairs in $\{(r_i, x) : r_i \in B, x \in B \cup L\}$ where *similarity*$(r_i, x) \geq t$.
- Only insert r_i into L only when none of the pairs for r_i contain any items from L. (For example $\{(r_i, x) : x \in B\}$ should be inserted, but not $\{(r_i, x) : x \in L\}$. We still need to find all pairs strictly within B though so they aren't duplicated.)

In both cases, it may be important to track which records were merged. Essentially, each pair (r_i, x) will have one representative, *representative*(r_i, x) that will end up in the final database L. The algorithm must be able to also see the non-representative items from each set also so they don't get "lost".

II. Disjoint Sets

Disjoint sets are a collection of $S = \{S_1, S_2 \dots S_k\}$ dynamic disjoint sets. The overall structure S supports three operations.

- `Make-set`(x): create a new set containing just x.
- `Union`(x, y): absorb set containing x into set containing y or vice versa, deleting the absorbed set.
- `Find-set`(x): find a pointer to the representative or "leader" of x's set.

This can be represented using **linked lists** or a **forest of directed graphs**. With linked lists, each disjoint set is one list of objects, where each of the objects have a pointer to the next item as well as back to the list head. The slowest operation is `Union`(x, y), which requires updating all of y's objects to point to the new head (the head of x). This means a sequence of $O(n)$ operations takes $\Theta(n^2)$ time [1].

Forests can also be used instead of linked lists. Each tree in the forest is one disjoint set. Each node in a tree has a pointer to its parent, and roots point to themselves. By using heuristics that estimate the tree heights and flatten the tall and narrow trees, this data structure is able to achieve closer to linear time, $O(m \times \lg^* n)$ where m is the total number of operations, and n is the number of `Make-set`(x) calls specifically, ie: the size of the input [1].

Both of these data structures can be used for record linkage. Let R be the superset over all the disjoint sets, $R = (S1 \cup S2 \cup \dots Sk)$. Also define a new operation `All-string-matches`(x) that given some $x \in R$, returns all items $x' \in R$ that have *similarity*$(x, x') \geq t$.

```
1: for x in R do
2:     Make-set(x)
3:     matches ← All-string-matches(x)
4:     for m in matches do
5:         Union(x, m)
6:     end for
7: end for
```

Let $n = |R|$. Line 2 is always $O(1)$, line 3 is $O(n)$. In [1], the authors prove `Union` called n times is $O(\lg n)$. The total running time is $n \times (n + \lg n) = O(n^2 + n \lg n)$. With a good `All-string-matches` function, line 3 will only produce a few matches, but this naive approach using disjoint sets still has $O(n^2)$ running time.

III. Self-organizing lists

Another approach is to use an in-memory self-organizing array. It is in-memory instead of connected to an external database. Self-organizing means it changes its order over time so that the total number of comparisons between two records are limited. For example, if a record was just seen, then it

can be moved to the front of this array. Then, if the same person/record appeared immediately after, then the algorithm would only have to do one lookup to find the duplicate. This is similar to a LRU cache. The **Analysis** section will have more on the running time of this solution.

This data structure can also support the offline or batched version of the problem. The in-memory list only contains the B items that are most similar to the current batch. Let L_b any items from the database that could match anything in the current batch. The size will be $<= B$.

1: **for** B in batches **do**
2: Fetch L_b using current batch B
3: Process current batch using $B \cup L_b$
4: Write B to database
5: **end for**

IV. Analysis

To analyze the cost of online or sequential algorithms, one method is to use probability and look at the expected total cost. **Amortized analysis** instead considers how often the expensive operations occur compared to the cheap operations in order to guarantee an average cost [1].

A simple example is allocating an empty array, adding items until it is full, then doubling the array to allow more space. When adding an item, the cost is just $O(1)$. But doubling the size of the array at sequence i means all i previous items have to be copied into a new array that is twice the size. Inserting the sequence $3, 4, 5$ into $[1, 2]$ would look like this.

1	2

1	2		
1	2	3	
1	2	3	4

1	2	3	4	5			

The cost of operation i is 1 when the list has an empty slot, but i when a new array needs to be allocated.

$$c_i = \{i \text{ if } (i-1) \text{ a power of 2; 1 otherwise}\} \quad (1)$$

In other words, the algorithm can "store up" energy on the cheap operations to later spend on the expensive operations. The potential function Φ measures the amount of stored energy on a dataset D after each operation. The **amortized cost** is then the base cost of an operation plus the change in potential.

$$\hat{c_i} = c_i + (\Phi(D_i) - \Phi(D_{i-1)}) \quad (2)$$

Consider charging 3 for each operation in the previous example. Spend 1 to insert the item, then save up the remaining cost, increasing Φ by 2.

0	2

0	0		
0	0	2	
0	0	2	2

0	0	0	0	2			

The left-over costs are spent when copying the 4 items to the new array. It is important that Φ never goes below 0.

$$\hat{c_i} = c_i + (\Phi(D_i) - \Phi(D_{i-1)})$$
$$\hat{c_i} = 1 + (2)$$
$$\hat{c_i} = 3$$

In the self-organizing array approach for record linkage, an array L is maintained that contains all items previously seen. The LRU algorithm then searches the array one-by-one to find a duplicate for each new record in the sequence $[x_1, ...x_n]$. If a match is found, it then must swap each adjacent pair to move that match to the front, costing $2 * rank_i$, where $rank_i$ is the position of the match in L.

The actual cost of string matching between the two pairs is constant, even though it might want to be handled differently in some situations. The potential function is then used to compare LRU's array with an oracle list that always minimizes total cost be seeing which items come next. The cost for the oracle algorithm at step i is $rank_i$ + some number of transposes to move x somewhere else in the list. (LRU always moves x to front after accessing, which is why $ci = 2rank_i$). Before analyzing the amortized cost, establish some variables.

- $x =$ item being accessed at step i
- $L_i =$ LRU's list at step i
- $L_i^* =$ oracle's list at step i
- $r = rank(x)$ in L_i
- $r^* = rank(x)$ in L_i^*
- $t_i =$ number of transposes (or swaps) done by oracle after step i
- $c_i = 2\dot{r}$
- $c_i^* = r^* + ti$

Then compare L_i and L_i^* using the following sets at step i.

- $A = \{$ everything before x in both L_i and L_i^* $\}$
- $B = \{$ everything before x in L_i, but after x in L_i^* $\}$
- $C = \{$ everything after x in L_i, before x in L_i^* $\}$
- $D = \{$ everything after x in both L_i and L_i^* $\}$

It should be apparent that $r = |A| + |B| + 1$ and $r^* = |A| + |C| + 1$. Note a few other facts based on these variables.

- $|B| = r - |A| - 1$
- $r^* \geq |A| + 1$

Next define the potential function for LRU. In practice, designing potential functions is a key step, but assume we already have one defined.

$$\Phi(L_i) = 2(\text{number of inversions between } L_i \text{ and } L_i^*) \quad (3)$$

Now solve for the **amortized cost**. When LRU moves x to the front it creates $|A|$ inversions and removes $|B|$ inversions. The oracle algorithm moves x also, creating at most t_i

inversions between L_i and L_i^*. This is the change in potential, $\Phi(L_i) - \Phi(L_{i-1})$.

$$\begin{aligned}
\hat{c_i} &= c_i + (\Phi(L_i) - \Phi(L_{i-1})) \\
&\leq 2r + 2(|A| - |B| + t_i) \\
&= 2r + 2(|A| - (r - |A| - 1) + t_i) \\
&= 2r + 2|A| - 2r + 2|A| + 2 + t_i \\
&= 4|A| + 2 + t_i \\
&\leq 4|A| + 4 + 4t_i \\
&= 4(|A| + 1 + t_i) \\
&\leq 4(c_i^*)
\end{aligned}$$

What this means is that the self-organizing array approach for record linkage is within a constant factor of the `oracle` cost, even though the self-organizing algorithm has no idea what order the records will arrive in. This problem setup and proof comes directly from [1].

For disjoint sets using the graph data structure, [1] defines the following potential function of a given node after i operations. The potential function again measures how much work is stored up in the data structure.

$$\phi(x) = \begin{cases} \alpha(n) \times x.rank \text{ if x is a root or } x.rank = 0 \\ (\alpha(n) - \text{level}(x)) \times x.rank - \text{iter}(x) \text{ otherwise} \end{cases}$$

Some of the details are beyond the scope here, but notice that $\phi(x)$ is defined in terms of $\alpha(n)$, which is the inverse **Ackermann function**. This grows very slowly, $\alpha(16^{514}) = 4$. It can then be shown that each operation `Make-set`(x), `Find-set`(x) causes a change in potential of at most $\alpha(n)$ [1]. Again refer to the definition of amortized cost ((1)).

$$\begin{aligned}
\hat{c_i} &= c_i + (\Phi(D_i) - \Phi(D_{i-1)}) \\
&= O(1) + O(\alpha(n))
\end{aligned}$$

This means each operation has an amortized cost of $O(a(n))$. So the total cost of m operations is $O(m \times a(n))$. This essentially means that the graph data structure for disjoint sets maintains all the necessary relationships for record linkage with minimal (linear) overhead, since $\alpha(n)$ is effectively a constant. The expensive part may come in the string comparisons when comparing items, but there are several other ways to handle that piece.

V. Practical Recommendations

Both of these data structures support the basic operations needed to de-duplicate items, but in practice their usage depends on several other factors. To start, it will be useful to estimate how many unique items the input is expected to have. If most of the items are unique, then the `LRU` algorithm might not be efficient since recent items are never re-used.

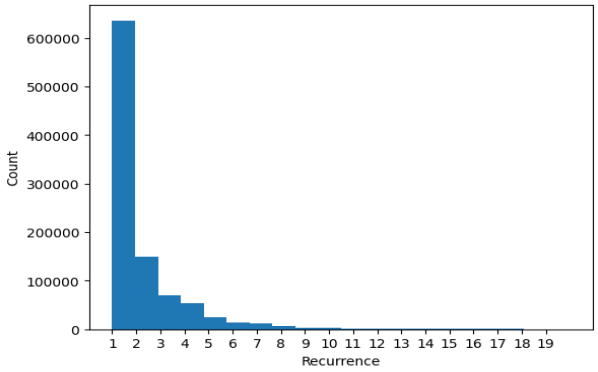
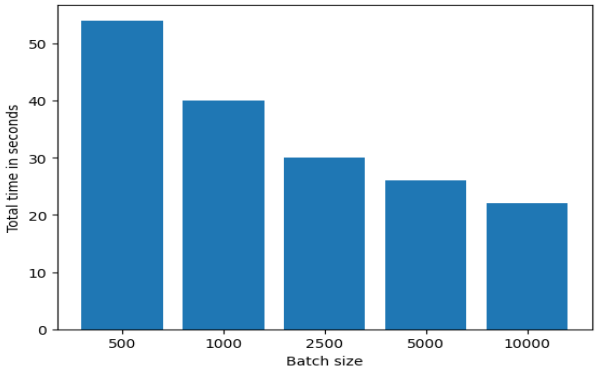


Fig. 1. Transaction recurrences

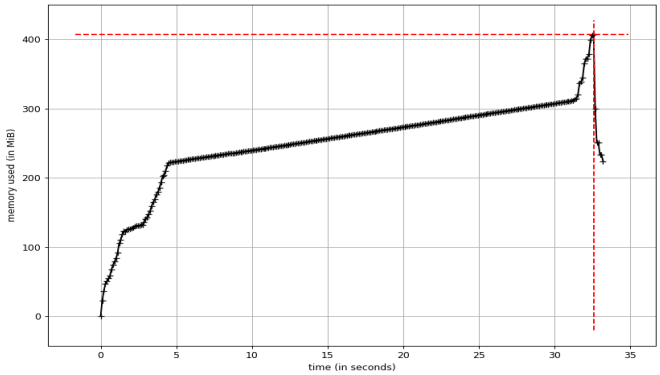
Using a dataset of 1 million unique people, this shows how many transactions were found for each person. Most people only had 1-3 transactions, meaning `LRU` might not be helpful. An approach that has worked well in practice for this dataset is to use a Lookup table based on some prefix or hashing function.

```
1: lookup ← new LookupTable()
2: for r in [r_1 ... r_n] do
3:     h ← hash(r)
4:     if h in lookup then
5:         Compare each match in lookup to r
6:     else
7:         Add r to lookup
8:     end if
9: end for
```

While there could be up to n matches to compare in line 5, a good prefix or hash strategy will limit this to a small number m. This means the total time of the lookup method is $O(n \times m)$ and $m \ll n$. The lookup method works with batching also. The following chart shows the total time taken to process 20,000 records using the lookup method with different batch sizes.



Note that the lookup table will always grow over time, so should only contain the items that are potential matches for the current batch. The following chart shows memory usage while processing 100,000 records with a lookup table.



The following shows a basic implementation of the LookupTable.

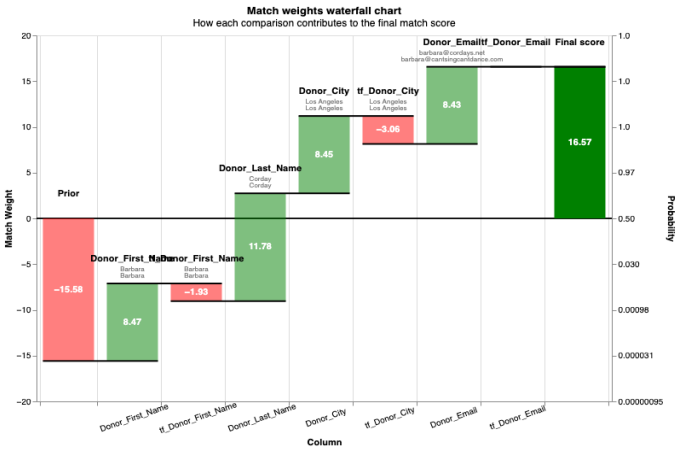
```
class LookupTable:
    by_email = {}
    by_first_last = {}
    def add_item(x):
        by_email[x.email].add(x)
        by_first_last[(x.first, x.last)].add(x)
    def get_matches(x):
        return by_email.get(x) | by_first_last.get((
    x.first, x.last))
```

VI. Other techniques

There are several other ways to handle record linkage, especially with the increased power of LLMs.

- Probabilistic matching considers the frequency of values for First Name, Last Name, etc to identify duplicates. Splink is a good library for this technique [2].
- Use transformers to get embeddings for each record and then use classification or clustering [3], [4].
- On-demand record linkage initially does not pre-process the records at all. Only when a user queries for a certain item, it finds duplicates for that specific item [5].

Probabilistic record linkage is based on the idea that different values for First Name, Last Name, etc. have different probabilities of appearing. A mismatch between two rows is weighted by how common the mismatched values are. For example, two Mary Smiths with different emails are more likely to be two different people than someone with a very unique first and last name, who happens to use two different emails. The following chart using Splink is an example. The matches in green increase the score, while the term frequency of Los Angeles and Barbara both decrease the score.



The Splink library applies to the offline version of record linkage with blocking or batching. Rows are filtered based on certain matching rules to prevent $O(n^2)$ string comparisons.

Another method for record linkage is deep learning and language models. In [4], the authors describe how records can be represented using attribute embeddings, where each record is a set of attributes. The model then classifies two records as a match or not a match. They discuss some different design decisions, and find that character-level embeddings work better than word embeddings and pre-trained embedding models work better than learning the embeddings for each dataset.

A few years later another solution named Ditto was introduced [3]. Instead of using pairs of embeddings, the model relies even more on large language models like BERT to do the classification. Each pair is serialized with special tokens that delimit the columns (attributes) and values.

$$serialize(r) = \text{[COL]}\ attr_1\ \text{[VAL]}\ val_1 \ldots$$
$$\text{[COL]}\ attr_k\ \text{[VAL]}\ val_k$$

The BERT model is then trained to identify matches and non-matches using this data.

BrewER is a popular approach for the on-demand version of the problem. In [5], the authors describe the limitations of the batch processing approach and the potential for quadratic complexity. Instead, BrewER "evaluates SQL SP (Selection and Projection) queries with ordering on the dirty data and returns the results as if they were issued on the cleaned data" [5]. There are two key characteristic of BrewER.

- "when a query is run on the dirty dataset, ER has to be performed only on the portion of the dataset needed for answering the query
- the resulting entities have to be returned in a progressive fashion as soon as they are obtained" [5].

This means results are available much sooner, which suits the online version of record linkage. It essentially allows this by maintaining a queue and only doing expensive operations in order of decreasing probability that two records are a match.

VII. Limitations

Disjoint sets and linked lists did not perform well for my specific dataset, but do still have applications in record linkage. One requirement specific to my problem was the ability to see which records were merged retro-actively. Essentially, a user should be able to know where every item in the original sequence went. Without this requirement, there may be faster algorithms that do not explicitly keep track of the merged items. Disjoint sets are suited for this requirement as long as there is a faster matching procedure. String matching is often very expensive, and dependent on the dataset. Most of the analysis assumes matches are all the same cost, but this can be a bottle neck in practice. A more complete analysis would account for string matching costs more carefully.

VIII. Review and conclusions

In practice, the lookup table method solves the primary $O(n^2)$ matching problem effectively enough in many situations, especially with low recurrences (Fig 1). There are several more advanced techniques depending on the specific problem requirements however. Using BERT models to predict matches can be efficient as well as accurate, but requires a match/non-match training set, which may not be available. The online version of record linkage is still a difficult problem to handle efficiently, but the BrewER method of limiting expensive queries until run-time may be useful in that area.

Disjoint sets fundamentally represent the problem in a very precise way, so it is possible that record linkage databases could leverage some of these ideas to efficiently store the records once processed. Self-organizing lists are another useful data structure, especially if the dataset has many co-located duplicates.

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2022). Introduction to algorithms (4th ed.). The MIT Press.

[2] Linacre, R., Lindsay, S., Manassis, T., Slade, Z., Hepworth, T., Kennedy, R., and Bond, A. (2022). Splink: Free software for probabilistic record linkage at scale. International Journal of Population Data Science, 7(3), 1794. https://doi.org/10.23889/ijpds.v7i3.1794

[3] Li, Y., Li, J., Suhara, Y., Doan, A., and Tan, W. C. (2020). Deep entity matching with pre-trained language models. Proceedings of the VLDB Endowment, 14(1), 50–60. https://doi.org/10.14778/3421424.3421431

[4] Mudgal, S., Li, H., Rekatsinas, T., Doan, A., Park, Y., Krishnan, G., Deep, R., Arcaute, E., and Raghavendra, V. (2018). Deep learning for entity matching: A design space exploration. Proceedings of the ACM SIGMOD International Conference on Management of Data, 19–34. https://doi.org/10.1145/3183713.3196926

[5] Zecchini, L., Simonini, G., Bergamaschi, S., and Naumann, F. (2023). BrewER: Entity Resolution On-Demand. Proceedings of the VLDB Endowment, 16(12), 4026–4029. https://doi.org/10.14778/3611540.3611612

[6] Maciejewski, J., Nikoletos, K., Papadakis, G., and Velegrakis, Y. (2025). Progressive Entity Matching: A Design Space Exploration. Proceedings of the ACM on Management of Data, 3(1), 1–25. https://doi.org/10.1145/3709715

[7] Kannangara, S., Abrahamyan, A., Elias, D., Kilby, T., Dar, N., Pizzato, L., Leontjeva, A., and Jermyn, D. (2025). A Robust and Efficient Pipeline for Enterprise-Level Large-Scale Entity Resolution. http://arxiv.org/abs/2508.03767

[8] Menestrina, D., Whang, S. E., and Garcia-Molina, H. (2150). Evaluating Entity Resolution Results.