A Dynamic Indexing for Incremental Entity Resolution over Query Results

Priscilla Vieira, Ana Carolina Salgado, Bernadette Farias Lóscio
Federal University of Pernambuco
Brazil
priscillakmv@gmail.com

ABSTRACT: Entity Resolution (ER) is the problem of identifying groups of tuples from one or multiple data sources that represent the same real-world entity. This is a crucial stage of data integration processes, which often need to integrate data at query time. This task becomes more challenging in scenarios with dynamic data sources or with a large volume of data. As most ER techniques deal with all tuples at once, new solutions have been proposed to deal with large volumes of data. One possible approach consists in performing the ER process on query results rather than the whole data. In this case, previous results of ER tasks are reused in order to reduce the number of comparisons between pairs of tuples at query time. In a similar way, indexing techniques can also be employed to help the identification of equivalent tuples and to reduce the number of comparisons between pairs of tuples. In this context, this work proposes an indexing technique for incremental Entity Resolution processes. The expected contributions of this work are the specification, the implementation and the evaluation of the proposed indexes. We evaluated the reuse of previous results of the ER process and highlighted its impact. The time spent for storing, accessing and updating the indexes was measured. We concluded that the reuse is more efficient than the reprocessing of tuples comparison.

Keywords: Data Mining, Data Integration; Entity Resolution; Data Matching; Duplicate Detection

Received: 24 May 2016, Revised 29 June 2016, Accepted 5 July 2016

© 2016 DLINE. All Rights Reserved

1. Introduction

In the last years, companies and government organizations around the world increased their production of digital data. In general, these data are stored in multiple data sources, which can be heterogeneous and dynamic. To access and analyze these data in a uniform and integrated fashion, data integration strategies are needed. The aim of data integration is to combine heterogeneous and autonomous data sources for providing a single view to the user [1].

Entity Resolution (ER) is an important step of the data integration process [2]. The ER goal is to identify tuples referring to the same real-world entity (in this paper, tuple is a synonym of instance or record). This problem has been the focus of several works
One of the main tasks of the ER is the tuple pair comparison. Tuple pair comparison calculates the similarity value between each pair of tuple in order to identify the corresponding ones. This is the most onerous task of the ER, especially in scenarios with a large volume of data. However, in some cases, additional strategies are still necessary to reduce the costs of the ER task. Indexing strategies are one of the possible solutions to reduce the number of tuples that need to be compared [2].

In this paper, we focus on ER over query results in scenarios with a large volume of data. The main challenge of this scenario is the performance of ER at query time. In this case, previous results of ER tasks are reused in order to reduce the number of comparisons between pairs of tuples at query time.

To allow the reuse an ER incremental approach is suitable. Incremental approaches process each iteration a set of tuples (increment). The new tuples (not previously processed) are compared and added to the tuples of previous iterations. In each iteration, the volume of classified tuples increases.

Incremental approaches are more suitable when performing ER on large volumes of data. However, in some cases, additional strategies of indexing are still necessary to reduce the costs of ER. For this purpose, we propose a dynamic indexing technique for incremental ER over query results. Our proposal consists of creating dynamic indexes that will be available in main memory. Our strategy reduces the costs of disk access when possible. The indexes are dynamic because they can be easily updated to reflect the results of the incremental ER.

We present the following three contributions:

- To the best of our knowledge, this is the first work that proposes and formalizes indexing techniques to incremental ER over query results.
- We propose two indexes [4] to incremental ER: Similarity Index and Cluster Index. The first one is used to index the similarity values between each pair of tuples being compared. The second one indexes a set of clusters of tuple identifiers.
- We show that reusing ER previous results in future ER process, runs significantly faster than the traditional one.

The rest of the paper is organized as follows. Section 2 describes some important concepts related to entity resolution and indexing. Section 3 discusses related work. Section 4 formally defines our research problem. Sections 5 and 6 describe our proposal for incremental indexes. Section 7 presents our experimental results. Finally, Section 8 concludes the paper.

2. Background

This section introduces important concepts related to this paper: entity resolution and indexing.

2.1 Entity Resolution

Entity resolution (ER) is the problem of identifying tuples referring to the same real-world entity. The ER problem may arise in scenarios where the data being analyzed is stored in a single data source as well as when the data is distributed in multiple data sources. It is more challenging in the second case, because several data sources contain redundant data with different representations. In this case, to allow an accurate and consistent data access, it is necessary to consolidate the different representations, detect and eliminate possible duplicates [2, 9, 10, 11].

ER can be seen as a cluster problem, such that each cluster corresponds to a single distinct real-world entity (e.g., a business, a person). A natural thought for incremental ER is that for each new tuple, we compare it with existing clusters, then either put it into an existing cluster or create a new cluster for it.

The incremental ER over query results has similar steps than the traditional ER [4]. However, it has additional steps to allow the reuse of previous classifications [11, 12, 13, 14]. The incremental ER over query results considers only what is relevant for a query [6, 8, 9, 10]. This characteristic is particularly important when considering a large volume of data.
Fig. 1 shows the basic steps of the ER approach over query results. In this process, for each new query result, we create blocks of tuples. Each block has a list of tuple pairs that are very unlikely to correspond to the same real-world entity. The blocks are accessed by a blocking key (or search key) [4]. After that, we reuse previous indexes. In the next step, the tuples do not processed previously, are processed in the tuple pair comparison step. In this step, the similarity value between each pair of tuples is calculated. The next step is the incremental clustering. In the end of the process the duplicated tuples are indicated and the dynamic indexes are updated. In Fig. 1, we highlight the steps of indexing.

![Figure 1. Entity resolution over query results](image)

### 2.2 Dynamic Index

The Indexing step in ER, creates candidate records that potentially correspond to matches. Different indexing techniques are summarized in [4]. Standard blocking [11], for example, segregates tuples into blocks according to a certain criteria, and subsequently compares only tuples that are in the same block. Usually, this criteria, called blocking key, is based on one or more attributes.

Most of these techniques aimed at traditional ER. However, not so much research has concentrated on entity resolution over query results. The dynamic index proposed in this paper aims at providing incremental ER over query results. This is flexible and values are added whenever a new query is being processed.

The main idea behind this approach is to pre-calculate similarities between tuples that are in the same block and previous indication of duplicate tuples. This information is stored in main memory to be used later in the ER process. Avoiding similarity calculations and classifications at query time significantly reduces the time needed for matching tuples.

### 3. Related Work

Recent researches have focused on the use of queries, indexing techniques or both to reduce the volume of data to be processed [7, 8, 16, 17, 18, 19]. Different indexing techniques are summarized in [15]. However, most of these techniques are focused on traditional ER process, with batch algorithms. There have been a few researches on incremental ER over query results [7, 8].

The first work on query-time ER is based on a collective classification approach [7]. The idea is to identify and process only database tuples that answer a query. In this approach, indexing to reuse previous classifications was not evaluated.
In [12], a pay-as-you-go approach to ER is proposed, constructing hints, which give information on tuples that are likely to refer to the same real-world entity. A hint can be represented in various formats (e.g., a grouping of tuples based on their likelihood of matching), and the ER can use this hint as a guideline to select which tuples should be compared first. Additionally, it utilizes blocking, but the data source is static and processed at once.

In [8], a query-driven approach to ER is proposed, exploiting the semantics of the given SQL queries. The semantic of a query can be exploited to reduce the number of comparisons between tuples. Additionally, it uses blocking along with the query, showing to be an effective combination to reduce the time of processing. The blocking is not incremental and the ER is traditional.

The solution proposed in [5] uses incremental clustering. For each inserted tuple, it compares the new tuple with previous tuples in clusters. Then, it either put the new tuple into an existing cluster, or create a new cluster for it. Additionally, it can use extra information from the data updates to fix previous problems of clusters. A blocking strategy is used, but the strategy is applied in a single data source and the ER is not query-driven.

Other researches close to ours are about the dynamic indexing. In [16] an indexing technique stores the most frequent attribute values in a data source. In [17] a tree-based dynamic indexing is proposed. The purpose is to maintain multiple trees with ordered attribute value. For each tree a different blocking key is used to create the key access of index. The index proposed facilitates the matching on a stream of query tuples, considering a large and dynamic database in real-time. Both papers focused on information retrieval [4]. Besides that, attribute values are indexed. This type of indexing is less likely to succeed when multiple attributes are used.

Our indexes are different in three aspects. First, our focus is the data integration problem and an incremental ER over query results. Second, we index tuples identifiers, and not only attribute values. Third, we index similarity measures and previous classifications of tuples from multiple data sources.

4. Problem Statement

Given a set of tuples, the ER process is essentially a clustering problem, in which each cluster contains tuples identifier that represent a single real-world entity (Definition 1). If we consider the ER problem in multiple data sources, each tuple can be from a different source.

<table>
<thead>
<tr>
<th>Tuples Id</th>
<th>Source Id</th>
<th>Name</th>
<th>Affiliation</th>
<th>Blocking Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Id₁</td>
<td>S₁</td>
<td>Alice</td>
<td>University Pernambuco</td>
<td>als</td>
</tr>
<tr>
<td>Id₂</td>
<td>S₁</td>
<td>Carlos</td>
<td>University Paraiba</td>
<td>krls</td>
</tr>
<tr>
<td>Id₁</td>
<td>S₂</td>
<td>Nunez</td>
<td>University Paraiba</td>
<td>als</td>
</tr>
<tr>
<td>Id₂</td>
<td>S₂</td>
<td>Carlos</td>
<td>University Pernambuco</td>
<td>krls</td>
</tr>
<tr>
<td>Id₃</td>
<td>S₂</td>
<td>Penelope</td>
<td>null</td>
<td>pnilp</td>
</tr>
<tr>
<td>Id₄</td>
<td>S₂</td>
<td>Alyce</td>
<td>University Pernambuco</td>
<td>als</td>
</tr>
</tbody>
</table>

Figure 2. Tuples with Blocking keys
We denote \( S = \{S_1, S_2, ..., S_n\} \) a set of data sources and \( Q = \{Q_1, Q_2, ..., Q_n\} \) a set of queries running on \( S \). Each source has a set of entities \( S_i.E, E = \{E_1, E_2, ..., E_n\} \). Each entity \( E_i \) from \( S_i \) has a set of tuples \( S_i.E_i.T_i = \{t_1, t_2, ..., t_n\} \) (instances of an entity \( E_i \)). A tuple \( t_i \) is defined as follows.

**Definition 1** Each tuple \( t_i \) belonging to \( T_i \), is represented by a set of pairs of attributes \( (A_k) \) and values \( (v_k) \), \( t_i = \{ (S_j.E_i.A_1, v_1), (S_j.E_i.A_2, v_2), ..., (S_j.E_i.A_n, v_n) \} \). Each attribute belongs to an entity \( E_i \) of a data source \( S_j \). A tuple \( t_i \) has a pair \( (S_j.E_i.A_k, v_k) \) which represents a single identifier to the tuple (Id) (See Fig. 2). The result of query \( i \), \( Q'_i.R \), is defined as a set of tuples.

Each tuple has a blocking key. The blocking key values are generated based on the values of either a single or several attributes. In Fig. 2 the function used to generate the blocking keys was Double-Metaphone [4] over name attribute.

For each new query result, the ER process reuses the previous generated clusters [18]. For this purpose we proposed two indexes: Cluster Index and Similarity Index. The first incrementally indexes a set of clusters of tuples identifiers. The second one incrementally indexes the similarity measure between each pair of tuples. To facilitate the understanding of these indexes, we use as an example the tuples of Fig. 2.

5. **Cluster Index**

The Cluster Index (CI) is defined as follows.

**Definition 2 (Cluster Index)** A Cluster Index, represented by CI, indexes a set of clusters of tuples identifiers. It is defined as, \( CI = \{(\text{ClusterId}, S_j.E_i, S_j.E_i.T_k, \text{Id}, \text{Key})\} \), where ClusterId is an identifier of a cluster, \( S_j.E_i \) is the entity and the data source of the tuple, \( S_j.E_i.T_k.\text{Id} \) is the tuple identifier and \( \text{Key} \) is the blocking key of the index [4].

Fig. 3 shows the CI generated from the tuples of the Fig. 2. We start with an empty CI. For each new \( Q'_i.R \) we search the ClusterId of the tuple in the CI by the blocking key.

The process of searching in the Cluster Index is illustrated in algorithm of Fig. 4. The input is a query result, \( Q'_R \), the entity queried, \( E_k \), the encoding function and the previous CI to reuse. The process starts generating a blocking key of each tuple (line 9). Second, we search in previous CI the ClusterId of the tuple (line 10). If the ClusterId exists, we retrieve the value of the CI (line 11). If the value of ClusterId does not exist, we will classify and update de CI at the end of the ER process.

**Figure 3. The Cluster Index created from the tuples in the Figure 2**
1. ClusterIndex ($Q_i'.R, E_v, Encoding Function, CI$)
2. Input: $Q_i'.R$: Query result
3. $E_v$: Entity
4. Encoding Function: Function to encode attribute value
5. CI: previous cluster index
6. Output: ClustersId: Clusters Id recovered from CI
7. Begin
8. ClustersID ← $\emptyset$
9. For each $t_i \in Q_i'.R$ do
10. key ← Encoding function ($T_i$);
11. If(exists (CI(key, $T_i$))) then
12. ClustersID.insert ($T_i.Id$, $T_i.S$, key);
13. return ClustersID;
14. End.

Figure 4. The Algorithm illustrates the overall process to a search in the Cluster Index

6. Similarity Index

1. SimilarityIndex($Q_i'.R$ with Blocking key, $E_v$, SimilarityFunction, SI)
2. Input: $Q_i'.R$ with Blocking key: Query result with the blocking key (key) of each tuple
3. $E_v$: Entity
4. SimilarityFunction: List of Similarity Functions - $S_{fi}$, $i = 1,\ldots, n$
5. SI: previous Similarity Index
6. Output: Graph: Similarity Graph
7. Begin
8. Graph ← $\emptyset$;
9. For each $T_i \in Q_i'.R$ do
10. For $T_j \in Q_j'.R$ do
11. If ($i\neq j \& T_i.key = T_j.key$) then
12. SIAux ← SI.get($T_i$, $T_j$)
13. If SIAux! = null then
14. Graph.insert(SIAux);
15. Else
16. SimilarityAux ← Similarity ($T_i.v$, $T_j.v$, $Sfi$)
17. Graph.insert (SimilarityAux, $T_i$, $T_j$);
18. SI.update ($T_i.v$, $T_j.v$, SimilarityAux);
19. return Graph;
20. End.

Figure 5. The Algorithm illustrates the overall process to search a similarity measure in a Similarity Index
To identify if a tuple is duplicated regarding a set of tuples (if they belong to the same cluster), it is necessary to make comparisons between this tuple and other tuples [5, 18]. For this purpose, similarity functions are commonly used in ER processes [4].

The second index, called Similarity Index (SI), indexes incrementally the similarity measure between each pair of tuples. At each new query result, the similarity measure is retrieved from the SI or inserted into the SI. Avoiding similarity calculations at query time significantly reduces the time needed for entity resolution over query results (see Section 8).

The Similarity Index is defined as follows.

**Definition 3** A Similarity Index (SI), indexes a set of similarities values. The SI is defined by a set of pairs, $SI = \{ (Key, AdjacencyList) \}$. AdjacencyList is a list of similarity values between a pair of tuples. Key is a blocking key [4] to access the index.

The process of searching a similarity measure in the Similarity Index is illustrated in algorithm from Fig. 5. The process requires the query result, $Q_i'.R$, with a blocking key of each tuple, the entity queried, $E_k$, the similarity function to be used and the previous SI to be reused. The process starts accessing the previous SI using the blocking key of each tuple (line 11). If the similarity measure exists, the value of SI is retrieved (lines 12-13). If the value of the similarity measure does not exist in the SI, it is calculates and the SI is updated at the end of the ER process (lines 14-17).

Fig. 6 shows the SI generated from the $Q_i'.R$ (Fig. 2). We start with an empty SI. For each new $Q_i'.R$, we search the similarity measure between tuples in the same block. The function used in the example to generate the blocking keys was Double-Metaphone [4] over name attribute.

![Figure 6. The Similarity Index created from the tuples in the Figure 2](image)

7. **Experimental Evaluation**

We present experimental results on real-world dataset. The results show that our incremental indexing is likely to succeed in incremental ER. Additionally, we show that the incremental ER has a better performance than traditional ER, without diminishing the quality of results.

7.1 **Experiment Setup**

Dataset: We performed experiments with CDDB dataset. This dataset includes 9763 CDs randomly extracted from freeDB, with 298 duplicates [20]. The CDDB was extracted by Hasson Pattner Institut and available in [20]. We created a set of random
samples of tuples from CDDB to simulate a set of query results. The samples size varies according to the purpose of each experiment. We indexed the tuples using the Double-Metaphone function [4]. Then, we applied the Levenshtein string similarity function [4], for pairwise similarity computation and ignored edges with a similarity below 0.9.

**Implementation:** To determine the effectiveness of our incremental indexing, we implemented the following incremental cluster algorithms to entity resolution:

- **Hill – Climbing [21]:** An ideal clustering should have a high cohesion within each cluster and a low correlation between different clusters. Several objective functions have been proposed for clustering [22]. The choice of this function is orthogonal to our techniques; here we adopt the cohesion, where the high values are better.
- **Single-Link [23]:** hierarchical clustering, we merge in each step the two clusters whose two closest members have the smallest distance.

We focus on these algorithms because they was previously used to entity resolution and evaluated as good algorithms for scenarios with a large volume of data [22, 5].

The algorithms were implemented in Java. We performed the experiments on a Windows machine with Intel Core i5 (2.2GHz).

**Measures:** We measured efficiency and quality of incremental algorithms using the proposed indexes. For efficiency, we repeated the experiments 100 times and reported the average execution time. For quality, we report the F-measure [4], given that we have the gold standard. Precision measures, among the pairs of records that are clustered together, how many are correct; recall measures among the pairs of records that refer to the same real-world entity, how many are clustered together; and the F-measure is computed as $\frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$.

**Objective:** The goal of these experiments is two-fold. First, we want to establish that the indexes proposed are desirable in a dynamic environment because of performance improvement. Second, we will show that incremental Entity Resolution using the proposed indexes has the quality similar to traditional Entity Resolution with batch algorithms.

### 7.2 Experiments to measure efficiency

To measure the efficiency, we created a set of random samples from CDDB to simulate a set of query results. For example, in Fig 7, we start with 70% of query result tuples indexed and 30% new tuples. The percentage is decreased from 70% to 10% (see Fig. 7). For each situation, we repeated the experiment 100 times. The result was the average of values in all runs. The same interpretation should be used to Fig. 8 - Fig. 10.
Figure 8. Time execution process for modified SI using Hill–Climbing Algorithm

Figure 9. Time execution process for naïve SI using Single-Link Algorithm
We considered four cases in each experiment (Fig. 7 - Fig.10): (i) Traditional uses batch algorithm. (ii) Best-Case uses incremental algorithm, assuming that all the query result tuples were indexed. (iii) Average-Case uses incremental algorithm, assuming that a percentage of query result tuples were indexed and another is new. (iv) Worst-Case assumes that all the query result tuples are new and they were not indexed.

We considered two scenarios: i) all the similarity measures calculated are indexed in the SI, independently of a threshold, named naive SI (Fig. 7 and Fig.9). ii) Only the similarity measures calculated above the threshold are indexed in the SI, named SI modified (Fig. 8 and Fig.10). Each scenario is executed with Hill-Climbing (Fig.7 and Fig.8) and Single-Link (Fig.9 and Fig.10) algorithms.

We observed that the size of the SI influences the performance of ER. The control of the Similarity Index is important to the efficiency of the incremental ER process. Because of trade the off, we considered that a SI modified is more efficient to the incremental ER over query results.

7.3 Experiments to measure quality
For measuring the quality, we calculated the average of F-measure over a set of runs of the previous experiment. We considered two cases: i) The quality of batch algorithm, without indexes for all dataset. ii) The quality of incremental algorithm, using the proposed indexes. We measure the result of average execution 100 times. Each time, we considered the same configuration of previous experiments. A query result has 70% of tuple indexed, after that 60%, and so on.

We evaluated the F-measure of ER with Hill – Climbing (Fig.11) and Single – Link (Fig.12) algorithms. In both cases, we observed that the F-measure from incremental algorithm is very close to batch algorithm.

![Figure 12. F-measure of Single-Link algorithm](image)

8. Conclusions

In this paper, two indexes for incremental ER over query results was presented, Similarity Index and Cluster Index. The quality and the efficiency of the ER process were evaluated, as well as the influence of Similarity Index size was investigated. We showed, on real dataset, that our indexes are applied in the incremental ER over query results to entity resolution. The incremental ER had the same quality of traditional processes, without indexes, but is more efficient. Our future work aims to analyse the indexes with other datasets, as well as to evaluate other incremental algorithms to ER.

References


[18] Blind Review


