Scaling up the ALIAS Duplicate Elimination System: A Demonstration

Sunita Sarawagi Alok Kirpal
{sunita,alok}@it.iitb.ac.in
Indian Institute of Technology Bombay

Abstract

Duplicate elimination is an important stage in integrating data from multiple sources. The challenges involved are finding a robust deduplication function that can identify when two records are duplicates and efficiently applying the function on very large lists of records. In ALIAS *the task of designing a deduplication function is eased by learning the function from examples of duplicates and non-duplicates and by using active learning to spot such examples effectively [1]. Here we investigate the issues involved in efficiently applying the learnt deduplication system on large lists of records. We demonstrate the working of the* ALIAS *evaluation engine and highlight the optimizations it uses to significantly cut down the number of record pairs that need to be explicitly materialized.*

1 Introduction

The goal of the ALIAS deduplication system is to automate the manual, time-consuming process of removing duplicates in large semi-structured lists. There are two main challenges of the duplicate elimination task that ALIAS addresses:

The first challenge is to define a robust deduplication function that can capture when two records refer to the same entity in spite of the various inconsistencies and errors in data. ALIAS automates this task by learning the function from examples of duplicates and non-duplicates. The success of the learning approach critically hinges on being able to provide a large *covering and challenging* set of examples that bring out the subtlety of the deduplication function. ALIAS interactively discovers such challenging training pairs through the use of **active learning**.

The second challenge is to efficiently evaluate the learnt deduplication function on large lists of records. If the function is treated as a black box, then the only method of evaluating it is to take a cartesian product of the entries. Then for each pair of entries, invoke the function to determine if the pair is a duplicate or not. This method could be intolerably expensive when the number of records is large. ALIAS views the function as a general AND/OR predicate on simpler similarity functions and applies a number of novel optimization techniques to defer materialization of pairs.

In Section 2 we summarize the process of learning the duplicate elimination function using active learning (details in [1, 2]). Our focus in this demonstration is the evaluation engine of ALIAS. We describe its novel cluster-based evaluation model and optimization strategies in Section 3.

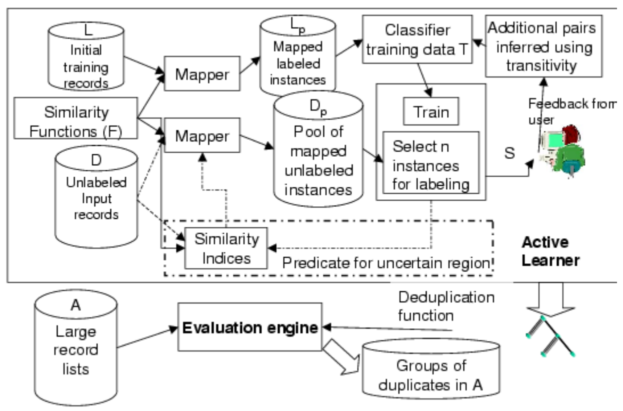
2 Learning the deduplication function

Figure 1 shows the overall design of ALIAS.

The input at this stage is a large list of unlabeled records D in which duplicates need to be found and a small seed set L of records labelled as duplicates and non-duplicates. Another kind of input to this stage is a collection of simple, easy-to-define similarity functions on attribute fields of the data. Examples of such functions are edit-distance, soundex, abbreviation-match on text fields, and absolute difference for integer fields. Many of the common functions can be inbuilt and added by default based on the data type. However, it is impossible to totally obviate an expert's domain knowledge in designing specific matching functions. The deduplication function is constructed by combining these similarity functions in the best possible way. We use $\mathcal{F}$ to denote the set of these input similarity function and n_f to denote the total number of such function.

An outline of the main steps is given in Figure 1. The first step is to convert each pair of records in $D \times D$ and $L \times L$ respectively into a similarity vector after applying the n_f similarity functions on them. The cartesian product on D could be expensive to compute. We can defer its materialization using the evaluation strategies discussed in the next section in conjunction with proper indices (as discussed in [1]).

The initial training data L_p is used to train an initial learner. Then the interactive process of active learning ensues on the unlabeled data D_p. The active learner chooses from the unlabeled pairs, a small list of record pairs and prompts the user for their label. The additional labeled data is used to train a revised classifier and the next set of pairs chosen for labeling. This process continues until the user



```
1. Input: $L$, $D$, $\mathcal{F}$.
2. Create pairs $L_p$ from the labeled data $L$ and $\mathcal{F}$.
3. Create pairs $D_p$ from the unlabeled data $D$ and $\mathcal{F}$.
4. Initial training set $T = L_p$
5. Loop until user satisfaction
       • Train classifier $C$ using $T$.
       • Invoke the active learner to select a set $S$ of $n$ instances from $D_p$
         for labeling.
       • If $S$ is empty, exit loop.
       • Collect user feedback on the labels of $S$.
       • Augment $S$ with pairs inferred using transitivity of the “dupli-
         cates” relation.
       • Add augmented $S$ to $T$ and remove $S$ from $D_p$.
6. Output classifier $C$
```

Figure 1. Overall design and working of the ALIAS **interactive deduplication system.**

is satisfied with the learnt function. The algorithm used by the active learner for choosing the most informative pairs for labeling appear in [1].

The output of this process is a deduplication function $\mathcal{I}$ that when given a new list of records A can identify which subset of pairs in the cross-product $A \times A$ are duplicates.

3 Optimization and Evaluation

In this section we discuss how we optimize the evaluation of a learnt deduplication function on very large lists of records. Although, ALIAS is general enough to train any kind of classifier (Naive Bayes, Support Vector Machines, Decision trees and so on), we recommend using decision trees for their ease of interpretation and high accuracy. A decision tree can be interpreted as a set of AND/OR predicates on similarity functions. For example, for deduplicating a list of citation entries the learner might return a tree-like predicate of the form below that all duplicate pairs need to satisfy:

```
TitleWordMatch > 0.41
AND  YearMatch > 0.3
     AND (PageMatch > 0.5
      OR  AuthorEditDistance < 5 )
```

A straightforward way of evaluating this expression is to first take a cartesian product of the citation entries. Then for each pair of entries, calculate the first similarity function that measures the overlap in the words common between the Title fields, if the overlap is greater than 0.41, check the year match and so on. ALIAS uses a number of optimizations to improve this simple-minded scheme requiring quadratic time. We describe these next.

3.1 Optimization 1: Grouped evaluation

Our goal is to design an execution engine that can efficiently evaluate an AND/OR combination of predicates on a *general* class of similarity functions while delaying the explicit materialization of pairs. While there has been prior work on optimizing the evaluation of a specific similarity function, there is little prior work on evaluating a combination of them efficiently. We propose an operator-based evaluation model where each operator takes as input a stream of *groups* of records and outputs another stream of *groups* of record. A *group* consisting of a set of n records actually represents the set of $n(n-1)/2$ pairs of records but the pairs are not materialized. The idea is to pipeline operators involved in the complete or partial evaluation of the atomic similarity functions such that they work on these linear groups as far as possible.

The component similarity functions so far were required to just work on pairs of records and return a real value denoting some similarity measure. Now they need to support an optional API that returns an evaluation plan for the function in terms of these inbuilt operators. The evaluation tree may not be an exact reproduction of the function's logic, as long as it is able to put all record pairs that satisfies the similarity predicate in at least one group.

The challenge is to design operators that adhere to the linearity of the groups and at the same time are useful for expressing a large class of similarity functions. The set of operators that are currently supported in ALIAS are described below. In the discussion that follows, we use the term *Partition* to denote a set of groups.

1. Create groups: create finer groups by breaking each group in an input partition into smaller, possibly overlapping groups. We have currently three methods of creating groups:
    - **Equal** match: create groups by putting all records with equal values of an attribute set in the same group. This is easily implemented by sorting the records in a group on the attributes and outputing all records with matching values as a new group.
    - **Range** match: create groups by grouping together all records with values of an ordered attribute within some range. This can be executed

by sorting the records on the attribute and using a sliding window to output overlapping groups.

- **Hash** match: Use a hash function to map each record into a set of values. All records mapped to a hash bucket define a group. The difference between Hash and Equal match is that while the hash match may put a record into multiple groups, the equal match puts a record in only one.

2. **Select** condition on a group: output only those records in each group that satisfy some condition.
3. **Split** an input partition into two partition streams by applying a condition on each record of a group and outputting it into the first stream if it satisfies the condition and in the second stream if it does not.
4. **Merge** a group G with a partition P: Union each group in P with G.
5. **Union** two partitions: the resultant partition is a union of the groups in each partition.
6. **Join** two partitions P_1, P_2: intersect each possible group pair (g_1^i, g_2^j) where g_1^i belongs to P_1 and g_2^j belongs to P_2.
7. **Aggregate:** Given a partition with each group attached with a weight and a threshold, output all possible minimal subsets of groups whose aggregate weight is greater than equal to the threshold.
8. **Compact**: collapse all groups in a partition with overlap in records greater than a threshold into a single group.

In Figure 2 we show how the similarity function predicate "YearMatch > 0.3" can be expressed using these operators. The YearMatch function returns a score of 1 when two years are equal, 0.35 when year is null in one or both, and 0 in all other cases. The first "Split" operator splits each group into records based on Year=NULL and feeds the non-null records to the Equal node that creates a group corresponding to each distinct value of year. The final Merge operator merges the Null-group(output by the Split operator) with each group output by Equal. Thus, all record pairs with YearMatch > 0.3 will appear in at least one group.

3.2 Optimization 2: Prefix expensive predicates with simpler canopies

Sometimes, it might be possible to come up with simpler functions (called **canopies**) that bound a more expensive function i.e. they might produce some false matches but will not drop any true matches.

For example, as shown in Figure 2, the predicate AuthorEditDistance < 5 can be prefixed by a simple canopy LengthDifference < 5, since two records with their length differing by more than 5 cannot have an edit distance less than 5.

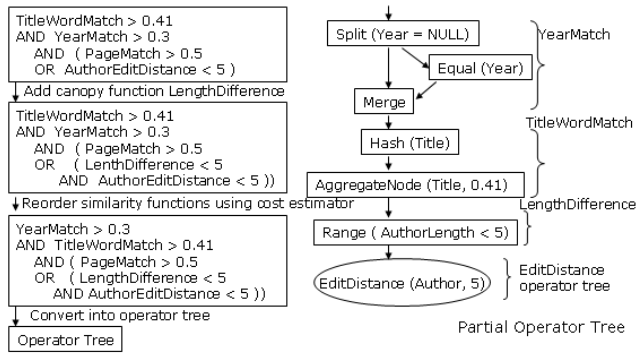


Figure 2. Example demonstrating the evaluation process and partial operator tree

3.3 Optimization 3: Reorder predicates to evaluate cheaper ones first

Some similarity functions are more expensive to evaluate than others. Also, they might materialize pairs earlier than others. For example, the YearMatch function is significantly easier to evaluate than the WordMatch function. Unlike normal expensive function optimization (as in predicate migration), the cost of each function is not easily modeled as a constant value per tuple. The cost depends on the size and number of input partitions. Also, once a function generates groups of size 2, all subsequent functions after it get little scope for optimization. Figure 2 shows the reordering of YearMatch function before the expensive WordMatch function. This avoids early application of the expensive functions on large lists by deferring them until partitions with smaller groups are obtained.

Figure 2 demonstrates the step by step application of these three optimizations for our example predicate. The operator tree does not show the ORed PageMatch due to lack of space. This operator plan provides a significant improvement over evaluating the function via cartesian products.

Acknowledgments This project was funded by the Ministry of Information Technology, India under the project "Mobile agents for collaborative distributed applications", 2001-2002. We would like to acknowledge the contribution of B. Anuradha and C. Mouli in the initial version of the software [2] and S. Sudarshan for helpful discussions.

References

[1] S. Sarawagi and A. Bhamidipaty. Interactive deduplication using active learning. In *Proc. of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining(KDD-2002)*, Edmonton, Canada, July 2002.

[2] S. Sarawagi, A. Bhamidipaty, A. Kirpal, and C. Mouli. Alias: An active learning led interactive deduplication system. In *Proc. of the 28th Int'l Conference on Very Large Databases (VLDB) (Demonstration session)*, Hongkong, August 2002.