Structure-aware XML Object Identification

Diego Milano, Monica Scannapieco and Tiziana Catarci

Università degli Studi di Roma “La Sapienza”
Dipartimento di Informatica e Sistemistica
Via Salaria 113, 00198 Roma
{milano,monscan,catarci}@dis.uniroma1.it

Abstract

The object identification problem is particularly hard for XML data, due to its structural flexibility. Tree edit distances have been proposed for approximate comparisons among XML trees. However, such distances ignore the semantics implicit in XML data structure, and their use is computationally infeasible for unordered data. In this paper, we define a new distance for XML data, the *structure aware XML distance*, that overcomes these issues, together with a polynomial-time algorithm to calculate it, and we present experimental result that prove its effectiveness and efficiency.

1 Introduction

The *object identification problem* is a central problem arising in data cleaning and data integration, where different objects must be compared to determine if they refer to the same *real-world entity*, even in the presence of errors such as misspellings. As the spread of the XML format as a data model increases, the need to develop effective strategies for XML object identification grows.

XML documents often represent complex, nested data, and schema languages for XML allow great flexibility in how such values are represented inside a document. XML data representations may allow for optional values, and lists of values whose length is not known schema-wise. Functions for approximate XML data comparisons must thus be able to cope both with errors at the level of textual data values and with structural flexibility. The hierarchical nature of XML data has lead to the use of *tree edit distances*([1]) to compare XML documents for various purposes, like detection of differences in versions of XML documents ([2],[3]). Some proposals also address the object identification problem ([4]). Tree edit distances in their original form give great importance to topological features of trees, but are not well suited when node labels and their nesting have semantics and data structure is somewhat regular. Another issue is that, due to the infeasibility of tree edit distance measures for unordered trees ([16]), such proposals are usually based on versions of the tree edit distance for ordered trees. Notice that, while the XML data model is indeed ordered, the presence of unbounded lists of values and optional elements in the data motivates for the adoption of unordered comparisons when looking for approximate matches. As an example, consider an element defined by the following DTD element definition: `<!ELEMENT SHOP (NAME, ADDRESS?, PHONENUM*)>` Here, an object representing a shop may contain zero or more phone numbers. The order in which phone numbers are listed is irrelevant, or however unspecified, so two objects representing the same shop might contain the same set of phone numbers in different order. Requiring that elements correspond to each other in an ordered way may lead to miss some of the similarities among those objects.

We propose a novel distance measure for XML data, the *structure aware XML distance*, that copes with the flexibility which is usual for XML documents, but takes into proper account the semantics implicit in structural information. The structure aware XML distance treats XML data as unordered. Nonetheless, differently from other distances for unordered trees, it can be computed in polynomial time. In this paper, we formally define the structure aware XML distance, we present an algorithm to measure the distance, prove its correctness and its computational cost, and we perform experiments to test the effectiveness and efficiency of our distance measure as a comparison function for XML object identification.

The rest of this paper is organized as follows. In Section 2 we review some related work. In Section 3 we first motivate the introduction of a new distance,

CleanDB, Seoul, Korea, 2006

showing with examples how approaches based on classical tree edit distance fail to respect the semantics of XML data and then define formally the structure aware XML distance. In Section 4 we introduce some theoretical properties of our distance and in Section 5 we present an algorithm to calculate it and we show its correctness and its time complexity. Section 6 describes experimental results. In Section 7 we draw some conclusions and describe some future work.

2 Related Work

The *object identification* problem has been extensively studied for relational data (with the name of record matching or record linkage problem), but the correspondent problem for semi-structured data has only recently drawn some attention. Most proposals for XML object identification are *structure oblivious*, in the sense that they rely on some kind of flattening of document structure to perform comparisons. In [14], XML objects are flattened and compared using string comparison functions. In the DOGMATIX framework([13]), data is extracted from an XML document and stored in relations called *object descriptions*. Tuples of two object descriptors containing data with the same XPath are classified as similar or contradictory using string edit distance, and object descriptions similarity is assessed taking into account the number of similar and contradictory tuples. The approach in [10] is similar, but comparisons of two objects take into account also approximate similarity results of *descendant* objects. *Structure aware* approaches rely on distance measures based on the tree structure of XML, like *tree edit distances* (see [1] and below in this section). In particular, the authors of [4] integrate string comparison functions into the classic tree edit distance for ordered trees to compute approximate joins on XML documents.

The notion of *tree edit distance* for ordered trees is due to Tai ([11]). The problem has also been extended to unordered trees ([16, 9]) and many other variations have been proposed (see e.g. [8, 5, 15]). Most versions of the edit distance problem allow polynomial-time algorithms for the case of ordered trees, but become NP-hard for the unordered case([16]). The *tree alignment distance*([5]) is a restricted version of edit distance. In tree alignment, trees are first made isomorphic (ignoring node labels) with the *insertion* of nodes labelled with *spaces*, and then overlayed. A cost function is defined on pairs of labels and the cost of an alignment is the sum of the costs of opposing labels. An *optimal alignment* is an alignment of minimum cost. Differently from the distance we propose in this paper, tree alignment considers insertions of nodes and overlays nodes with different labels, and it is NP-Hard for unordered trees.

Tree edit distances have been employed also for data and document change detection [2, 3]. The problem has connections with object identification, but in that context XML documents are mostly modelled as ordered trees, edit operations are extended to entire subtrees, and the focus is on efficiently finding an *edit script* to represent the changes.

3 A Structure-Aware Approach to XML Object Identification

Approaches to solve the object identification problem generally make use of some kind of distance function to detect the similarity of two objects. In record matching techniques proposed for the relational model, attribute values are often compared using *string comparison functions* ([7]). XML documents can be modelled as node labelled trees. This hierarchical, tree-like nature justifies the proposal of similarity measures that integrate string comparison functions with *tree edit distances* ([1]). However, tree distances are not fully able to capture the semantics of XML data, as they do not keep into account the semantics and structural relationships among XML elements.

In this section, we first show some weaknesses that classic tree edit distances suffer when used to compare XML data, and then define a new notion of distance for XML data, the *structure-aware XML distance*, as the basis of an approach to XML object identification.

3.1 Tree Distances

Given a set of edit operations on labelled trees (i.e. node insertions, deletions and relabelling) and a function that assigns a cost to each operation, the *tree edit distance* between two trees is defined ([11]) as the minimum cost sequence of tree edit operations required to transform one tree to another.

Comparison of XML data based on tree distances has been proposed for various purposes ([4, 2, 3]). The authors of [4] perform approximate comparison of XML data with the tree edit distance defined above, using a string comparison function to compute the cost of node relabelling. This approach has the advantage of keeping into account the tree structure of XML data. However, the use of tree edit distance for this purpose has some drawbacks. The examples in Figure 1 illustrate two of them. First, consider the XML data trees **a) b)** and **c)**. Tree **c)** represents the same data as tree **a)**, and also contains some additional information. Tree **b)**, instead, represents different data. However, the tree distance between **a)** and **c)** is greater than the distance between **a)** and **b)**. Consider now trees **d)** and **e)**. Here, a person is represented with its parents and an optional list of friends. When measuring the tree edit distance between such two trees, a minimal distance is obtained by deleting from tree **d)** the entire *parent* subtree, relabelling node *friends* into *parents* and matching its leaves to two of the nodes of the *parent* subtree of tree **e)**. This behaviour clearly violates

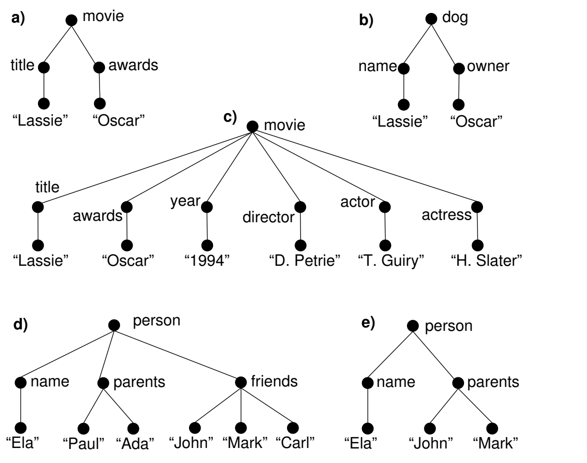


Figure 1: Two issues in classical tree edit distance-based XML comparisons

the semantics implicit in node labels. These problems, in addition to the need of performing unordered comparisons efficiently, motivate the introduction of a new distance measure for XML data.

3.2 XML Structure-aware Distance

In this section, we give an intuitive description of how the issues highlighted in the previous section can be overcome, and then formalize the intuition to define a new distance for XML data. Notice that, throughout this section and in the rest of this paper, we consider XML trees as *unordered*. The above examples show that, when comparing XML trees, a good choice is to match subtrees that have similar structure and that are located under the same path from the root. These can be indeed interpreted as clues of the same semantics. If two trees have exactly the same structure, and only differ by the textual values present on the leaves, we can *overlay* the trees so that nodes with the same path match. When multiple overlays are possible, then we choose one such that the distance among textual values on the leaves is minimal. If the structure of the two trees differ, due to additional information, we can still realize an overlay as above by deleting extra subtrees that do not match well.

The following definitions make the notion of overlay introduce above more formal. We assume a model of XML objects as labelled trees. All leaves are labelled with the same special label τ. Given a leaf l, its textual value (different from its label) is denoted by $text(l)$.

Definition 1 (Overlay) *An* overlay O *of* T_1 *and* T_2 *is a non-empty set of pairs of nodes from* T_1 *and* T_2 *with the following properties:* $\forall v_i, v_i' \in T_i, \forall n_i \in T_i - leaves(T_i), i = 1, 2,$

$$if\ \langle v_1, v_2\rangle, \langle v_1', v_2'\rangle \in O,\ then\ v_1 = v_1'\ iff\ v_2 = v_2' \quad (1)$$

$$if\ \langle v_1, v_2\rangle \in O,\ then\ path(v_1) = path(v_2) \quad (2)$$

$$\langle n_1, n_2\rangle \in O\ iff\ \exists v_1, v_2\ s.t.\ n_1 = parent(v_1) \wedge \quad (3)$$

$$n_2 = parent(v_2) \wedge \langle v_1, v_2\rangle \in O$$

Where $path(v_i)$ *denotes the sequence of node labels* $label(root_i) \ldots label(v_i)$ *encountered when traversing* T_i *from the root to node* v_i.

If $\langle v, w\rangle \in O$ we say that v and w *match*. If a node is not matched with any other node, we say that it is *deleted*. Informally, an overlay matches nodes from T_1 to nodes from T_2 one-to-one, so that nodes or leaves are matched only if they have the same path from the root. Two non-leaf nodes can be matched iff they are ancestors of two leaves that are matched. Notice that this implies that, if a node is deleted, all its descendants are also deleted. It also implies that an overlay of two trees exists only if there exist two leaves $l_1 \in T_1$ and $l_2 \in T_2$ with the same path from the root. We say that two trees are *comparable* if they have at least one overlay.

Definition 2 (Maximal overlay) *An overlay* O *of two trees is* maximal *if there is no other overlay* O' *such that* $O \subset O'$.

Consider the two trees in Figure 2. In the figure, nodes are marked in breadth-first visit order, respectively with numbers and lowercase letters. Uppercase letters beside nodes denote labels, and leaf labels are not shown, while quoted strings denote their textual values. An example of overlay is $O = \{\langle 1, a\rangle, \langle 3, b\rangle, \langle 6, e\rangle\}$. This overlay is not maximal, since there exist other overlays that contain it. An example of maximal overlay that includes O is the overlay shown in the figure using dashed lines $O_c = \{\langle 1, a\rangle, \langle 2, c\rangle, \langle 3, b\rangle, \langle 5, g\rangle, \langle 6, e\rangle\}$. Intuitively, O_c is maximal since it is not possible to add another line from a node of T_1 to a node of T_2 not already touched by a line while maintaining overlay properties. Notice, however, that more than one maximal overlay may exist between two trees. For the trees in Figure 2, another maximal overlay is obtained by matching node 5 with node f rather than node g. Another one is $O_c' = \{\langle 1, a\rangle, \langle 2, c\rangle, \langle 3, d\rangle, \langle 5, g\rangle, \langle 6, i\rangle, \langle 7, h\rangle\}$.

Definition 3 (Cost of a match) *Let* $sdist(s_1, s_2)$ *be a string comparison function. The* cost of match *for two nodes* v, w *is:*

$$\mu_{v,w} = \begin{cases} sdist(text(v), text(w)) & if\ v, w\ are\ leaves \\ 0 & otherwise \end{cases}$$

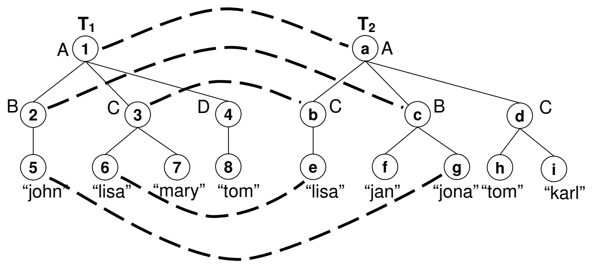


Figure 2: A maximal overlay of two trees

Definition 4 (Cost of an overlay) *The* cost of an overlay O *is defined as* $\Gamma_O = \sum_{\langle v,w\rangle \in O} \mu_{v,w}$.

Definition 5 (Optimal overlay) *An overlay O of two trees is* optimal *if it is maximal and there is no other maximal overlay O' such that $\Gamma_{O'} < \Gamma_O$.*

Consider again the trees in Figure 2. The cost of the overlay O_c showed in the figure is given by the sum of distances of textual values on matching leaves. Using for instance the common string edit distance ([12]), their distance can be calculated as *sdist*("john","jona") + *sdist*("lisa","lisa") $= 2 + 0 = 2$. The cost for overlay O'_c defined above is instead given by *sdist*("john","jona") + *sdist*("lisa","karl") + *sdist*("mary","tom") $= 10$. Actually, O_c is an optimal overlay for the two trees. Notice that more than one optimal overlays may exist for two trees. In this example, the overlay obtained by O_c by matching leaf 5 with leaf f instead of leaf g is still optimal, as the string "john" has the same distance from the strings "jan" and "jona". It is worthwhile to notice that, if two given data trees are comparable, i.e. there is at least an overlay for them, then from the above definitions it follows that there is also a maximal overlay and an optimal overlay for them.

Definition 6 (Structure aware XML distance) *The* structure aware XML distance *of two comparable XML trees T_1 and T_2 is defined as the cost of an optimal overlay of T_1 and T_2.*

Notice that, when applied to the trees in the example given in Figure 1, this distance works as expected. Trees **a)** and **b)** are incomparable, while the distance of trees **a)** and **c)** is zero. In the case of trees **d)** and **e)**, the distance only considers the differences among those leaves that it is meaningful to compare, giving as a result the least distance between names present under the nodes *parents*.

4 Properties of Overlays

In the next section, we present an algorithm to measure the structure aware distance defined in section 3. In this section, we describe some properties of overlays that are useful to prove its correctness. In particular, we show that an optimal overlay of two trees T_1 and T_2 can be found by determining an assignment among the children of their roots such that the sum of the costs of optimal overlays for the corresponding subtrees is minimal (Theorem 4). To prove this result, we first show that the cost of an overlay of two trees can be rewritten in terms of the cost of overlays of the children of their roots (Theorems 1 and 2).

For space reasons, we omit the proofs of some theorems and only sketch other proofs. Throughout this section, we denote with T_1, T_2 two comparable data trees, with r_1, r_2 their roots, and with $v_i, w_j, i \in [1, deg(r_1)], j \in [1, deg(r_2)]$ the children of r_1 and r_2, respectively. Furthermore, given a node v, we denote with $T(v)$ the tree rooted at v. We call the trees $T(v_i)$ and $T(w_j)$ the *first-level subtrees* of T_1 and T_2 respectively.

Theorem 1 *Let O be an overlay of T_1, T_2. If $\langle v,w\rangle \in O$, then the set $O_{v,w} = \{\langle y,z\rangle \in O | y \in T(v), z \in T(w)\}$ is an overlay of $T(v)$ and $T(w)$. Moreover, if O is maximal then also $O_{v,w}$ is maximal. We say that $O_{v,w}$ is the overlay* induced *by O on $T(v)$ and $T(w)$.*

Proof sketch: To show that $O_{v,w}$ is an overlay of $T(v)$ and $T(w)$, we show that properties (1),(2) and (3) of overlays are respected. The fact that $O_{v,w}$ is also maximal can be proved by contradiction. If there exist another overlay of $T(v)$ and $T(w)$ $O'_{v,w} \supset O_{v,w}$, then there exists another overlay $O' \supset O$, and thus O is not maximal. ■

Theorem 2 *Let O be an overlay of two trees T_1, T_2. Then*

$$O = \langle r_1, r_2\rangle \cup \Big(\bigcup_{\{\langle v_i, w_j\rangle\} \in O} O_{v_i,w_j} \Big) \qquad (4)$$

$$\forall \langle v_i, w_j\rangle, \langle v_h, w_k\rangle \in O, O_{v_i,w_j} \cap O_{v_h,w_k} = \varnothing \qquad (5)$$

Proof sketch: (4) By definition, $O_{v_i,w_j} \subset O$ and thus $\langle r_1, r_2\rangle \cup (\bigcup_{\langle v_i,w_j\rangle \in O} O_{v_i,w_j}) \subset O$. It remains to show that the reverse inclusion holds, i.e. that for any match $\langle y,z\rangle \in O, \langle y,z\rangle \in \{\langle r_1,r_2\rangle\} \cup (\bigcup_{\langle v_i,w_j\rangle \in O} O_{v_i,w_j}$. This can be shown reasoning on the depth of $\langle y,z\rangle$.(5) can be proved showing that for any two overlays $O_{v_i,w_j}, \in O_{v_h,w_k}$, if their intersection is not empty, then necessarily $v_i = v_h$ and $w_j = w_k$. ■

In other words, an overlay of T_1 and T_2 is a partition of the overlays it induces on its first-level subtrees and the couple $\langle r_1, r_2\rangle$.

Theorem 3 *The cost of a maximal overlay O is the sum of the costs of all the overlays O_{v_i,w_j} induced on its first level subtrees.*

Proof: Follows immediately from Theorem 2. ■ From this result it follows, trivially, that an overlay is optimal only if the overlays induced on its first level subtrees are all optimal. Notice that the reverse does not hold in general. As an example, consider the overlay shown in Figure 4. It is easy to see that the overlays $O_{3,b}$ and $O_{4,c}$ are both optimal since *sdist*("john","joe")+*sdist*("mary","mark") $<$ *sdist*("john","mark") + *sdist*("mary","joe"). However, the overlay of T_1 and T_2 shown in the picture is not optimal, since another overlay with cost 0 can be obtained by matching node 3 with c and node 4 with b. In order to reach a necessary and sufficient condition for the optimality of an overlay, we introduce a few more definitions. Given two trees T_1 and T_2, a *first-level assignment* of T_1 and T_2 is a set of couples

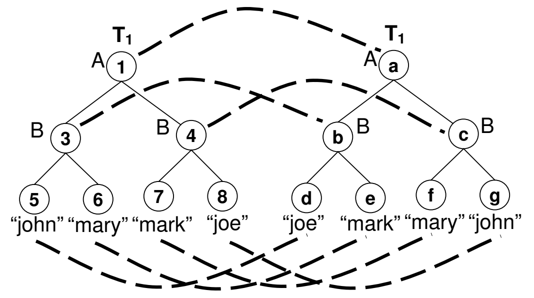


Figure 3: The overlay of T_1 and T_2 is not optimal, even if all the overlays induced on subtrees are optimal

$\langle v_i, w_j \rangle$ such that each node of each tree is coupled with at most another node of the other one, and the trees $T(v_i)$ and $T(v_j)$ are comparable. The concept of maximality defined for overlays can be easily extended to first level assignments. It is easy to prove that an overlay is maximal iff includes a maximal assignment. By defining the *cost* of a maximal first-level assignment as the sum of the costs of the overlays of first level trees whose roots are coupled in the assignment, the concept of optimality can also be extended to first-level assignments. Given this definition, we can now present the main result of this section:

Theorem 4 *An overlay O is optimal iff it contains an optimal first-level assignment A and the overlays induced on its first level subtrees are all optimal.*

Proof sketch: Maximality follows from the maximality of A. From the optimality of A and of the overlays induced of first level subtrees it follows, by Theorem 3, that the overlay is also optimal. ■

The previous result gives an operative way to build an optimal overlay of two trees T_1 and T_2. A maximal assignment can be obtained by first matching r_1 with r_2 and then matching a children of r_1 with a children of r_2 until no more matches are possible. Two nodes v_i and w_j are matched only if $T(v_i)$ and $T(v_j)$ are comparable. In this case, an optimal overlay is built for $T(v_i)$ and $T(w_j)$ by applying the same process, recursively, up to the leaves. All possible maximal assignments must be built and evaluated, and an optimal one must be chosen.

5 Structure Aware XML Distance Measurement

In this section, we introduce an algorithm to measure the structure aware XML distance defined in Section 3, and prove its correctness and its worst case complexity.

Algorithm 1 analyzes two comparable trees recursively, starting from the roots. If the roots are leaf nodes, a distance measure for their associated text values is returned. Such function is denoted by the procedure *sdist*() in the algorithm. Otherwise, the algorithm considers their children, and computes a distance for each couple of subtrees rooted at children

Algorithm 1 $dist(T_1, T_2)$

```
if $isLeaf(r_1)$ and $isLeaf(r_2)$ then
  $return\ sdist(text(r_1), text(r_2))$
else
  $xmldist := \infty$
  for all $l$ in $labels(children(r1) \cup children(r_2))$ do
    for all $v_i \in children_l(r_1)$ do
      for all $w_j \in children_l(r_2)$ do
        $D_l[i,j] := dist(T(v_i), T(w_j))$
      end for
    end for
    $assignment_l := findAssignment(D_l[])$
    for all $\langle h,k \rangle \in assignment_l$ do
      if $xmldist = \infty$ then
        $xmldist := 0$
      end if
      $xmldist := xmldist + D_l[h,k]$
    end for
  end for
  return $xmldist$
end if
```

with the same label, recursively. After all distances have been calculated, the algorithm must assign each node to another node with the same label, minimizing the overall cost. This is an assignment problem and can be solved using a variation of the well-known Hungarian Algorithm ([6]). In the algorithm, this task is performed by a call to procedure *findAssignment*(). In particular, given a matrix of distances, the procedure returns a set of assignments containing couples of indices of assigned nodes. For ease of presentation, in the algorithm we denote the set of all children of node v having label l with $children_l(v)$. Results of distance calculations for a certain set of children having label l are stored in an array named D_l. The distance is initially set to ∞, and reset to 0 only in the case that there is at least one assignment of root children.

From the results given in the previous section and the definition of structure aware XML distance given in section 3 it follows immediately that:

Theorem 5 *Algorithm 1 correctly computes the structure aware XML distance of two comparable trees.*

In order to understand the computational cost of the algorithm, let us consider a case in which all the leaves of the tree have the same path, and the data trees are maximal. We consider distance calculation among two trees T_1 and T_2. We denote with deg_1 and deg_2 their respective degrees and with L_1, L_2 their sets of leaves.

Let T_1', T_2' be two subtrees of T_1 and T_2 rooted at level l, and let r_1', r_2' be their roots. In order to compute their distance, we must choose a match among the children of r_1' and r_2' such that the the sum of distances for corresponding subtrees is minimal. Assuming that we have already calculated all pairwise distances, we need to solve an instance of the linear assignment problem. The Hungarian algorithm gives a solution in cubic time, so the cost of an assignment is $O((deg_1 + deg_2)^3)$.

To compute all distance measurements, we proceed bottom up, starting from the leaves and calculating all pairwise distances among all nodes at each level. At level $depth - 1$, before performing the assignment phase we must compute distances among textual values. These are computed in constant time (w.r.t. the size of the trees). At upper levels, we already know the distances among nodes at lower levels, so we just need to perform the assignment phase. In total, the assignment phase is repeated $(|T_1| - |L_1|) \times (|T_2| - |L_2|)$ times. Thus, the overall cost is $O((|T_1| - |L_1|) \times (|T_2| - |L_2|) \times (deg_1 + deg_2)^3)$. When there is more than one path for leaf nodes, the calculation is less expensive.

6 Experiments

This section presents an experimental evaluation of the structure aware XML distance introduced in the previous sections. We tested both *effectiveness* and *efficiency* of our distance as the basis of a comparison function for XML Object identification.

6.1 Experimental Setting

In the tests, a set of objects contained in an XML document are compared pairwise for similarity. Two objects o_i, o_j are classified as similar if $dist(o_i, o_j) < t$, where dist is the structure aware XML distance and t is a fixed threshold. The string comparison function we use is the string-edit distance([12]). We performed three sets of tests. Tests in the first and second set aim at determining how good is our measure at correctly identifying similar objects under various experimental conditions. The third set was performed to compare our distance with the tree-edit distance used in [4].

6.2 Data Sets

The experiments were performed on a synthetic data set created with ToXGene[1]. The objects compared for similarity represented persons, defined as in the following DTD element declarations:

```
<!ELEMENT PERSON (NAME, MIDDLENAME*, SURNAME, ADDRESS)>
<!ELEMENT ADDRESS (STREET, CITY, COUNTRY, EMAIL*, PHONENUM*)>
```

all elements not defined above contained PCDATA. Textual data values were taken from the XMark benchmark; *middlename*, *email* and *phonenumber* optional elements were limited to a maximum of respectively to 2, 2, and 4 instances for each person. We started from a clean data set of 300 distinct *person* objects. In all experiments, for each *person* object another similar object was created (duplication rate = 100%). In order to create similar objects under controlled conditions, we have developed a small tool[2] that allows to produce objects that differ from the original by a given rate of *internal data deletions*, *text changes*, and *internal data duplications*, and *element swaps*. Internal data deletions are leaf deletion. By internal data duplication we mean that a portion of an object (e.g. an *middlename* element or an *address* subtree) is duplicated *inside the object*. These internal duplicates receive the same amount of changes as other duplicates, except for internal duplication. Text changes consisted of deletions, insertions and character swaps. We inserted a minimum of 2 errors for each text modification. The specific kind of errors were chosen randomly with equal probability (e.g. a character deletion and a swap, or two swaps).

Notice that change rates mentioned in the tests refer to the rate of changes *for each duplicate*. That is, all duplicates differ from their original, and the change rate refers to which percentage of the data present in a duplicated object has received such changes. As an example, a deletion rate of 20% means that in the duplicate of a certain object 20% of the leaves were deleted. A text change of 10% means that 10% of the textual values present on leaves that were not deleted are altered with a certain number of deletions, insertions and character swaps.

[1]www.cs.toronto.edu/tox/toxgene
[2]www.dis.uniroma1.it/milano/duplicatecreator

6.3 Experimental Set 1

A first set of experiments tested how the effectiveness of our measure varies depending on the chosen threshold. Effectiveness is measured in terms of the f-score:

$$Fscore = 2 \cdot recall \cdot precision / (recall + precision)$$

Where *recall* is the percentage of matches identified by the algorithm, and *precision* is the percentage of actual matches among those declared by the algorithm.

We let the threshold vary over a wide range of values, and performed the test for files in which the differences among duplicate object are set as follows:

case	deletion rate	text change rate
1	10%	10%
2	10%	20%
3	20%	20%
4	20%	30%

Observe that the last one is a rather critical situation: a duplicate for an object with 10 leaves has 2 leaves removed and three of the remaining leaves contain errors. The results for this set of experiments are shown in Figure 4. A first observation that must be done is that in all cases the measure achieves an f-score of 100% for some threshold values. Thus, the distance shows high effectiveness as an object identification comparison function. It is also worthwhile to notice that all the curves in Figure 4 show a wide plateau in which the f-score stay at its maximum value. Since, in real-use conditions, determining the right threshold is mainly a matter of experience and it is often infeasible to evaluate various thresholds, the relative insensibility of the distance to threshold variation in a wide range of threshold must be considered as a positive feature. The curves in the graph appear grouped

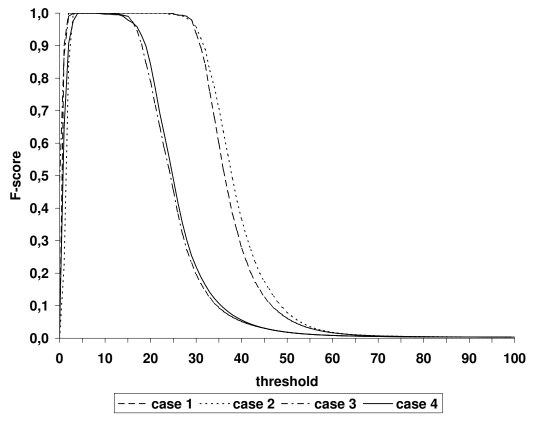


Figure 4: Results for experimental set 1: fscore dependency on similarity threshold

in two sets, corresponding to the two values of deletion rate used in the experiment. This behavior means that the distance is more sensible to a variation in the data available for comparison than to diffused errors over the data itself.

6.4 Experimental Set 2

Our second set of experiments was performed to determine how our distance is influenced by a variation in the kind and amount of differences among similar objects. The threshold was fixed to a value of 10, and the rate differences introduced in duplicate objects were varied independently. Figures 6(b), 6(a) and 6(c) show respectively the trend of the recall, precision and f-score measures for these tests. Each graph contains different curves, each one relative to a specific kind of differences. Different kinds of changes affect in different way the behavior of the measure. From the graphs it appears that a high rate of deletions affects precision but not recall. This is due to the fact that deletions reduce the amount of comparable information available to determine if two objects are similar. On the other hand, high text change rates mainly influence recall, since a high rate of errors may introduce high differences in comparable features of objects. The measure is completely insensible to swaps of elements. This is not surprising, since it compares unordered trees.

6.5 Experimental Set 3

We described in section 3 some issues related to ordered tree edit distances when used in an object identification context, and we claimed that our distance is both more efficient and effective than such class of distances. Our third set of experiments confirms this claim. We tested our distance measure and the ordered tree-edit distance over the same set of files, and compared both execution time and f-score. The algorithm used for the ordered tree edit distance is described in [17], and the cost function was modified to account for textual value comparison as proposed in [4]. More specifically, the cost for relabelling, insertion and deletion for leaves was evaluated based on the same string comparison function used in our measure. For internal nodes, a unit cost function was used. The files used in the tests contained duplicates produced by applying to each object all the four kinds of changes described in section 6.2, with the same change rate for each difference. Thus, a *data change* rate of 10% means that each duplicate has data deletion, text changes, internal duplication, and element swap rates all set to 10%. We let the change rate vary from 10% to 45% at intervals of 5%. The threshold was fixed. The graphs shown in Figure 5 refer to the results obtained for the best possible choice of a threshold for both measures. The differences in execution time for the two distances were dramatic. To perform the comparison over a total of 600 objects the ordered tree edit distance took approximately 7,5 hours while, on the same machine and for the same data, our distance takes something more than a minute. The results shown in Figure 5 highlight how our distance constantly outperforms the ordered tree edit distance from the effectiveness point of view. This differences are partly due to the fact that our distance performs an unordered comparison, and the files contain swapped elements. Notice that only adjacent elements with the same name where swapped in the test files, thus respecting the DTD of the data. Another difference is also due to the fact that our measure ignores internally duplicate data that cannot be matched, while the tree edit distance accounts for the cost of deletion of such data.

7 Conclusions

XML data has tree-like nature and flexible structure. These features have lead to proposals for XML object identification that exploit tree-edit distances to perform approximate comparisons among XML trees.

The tree-like nature and flexible structure of XML

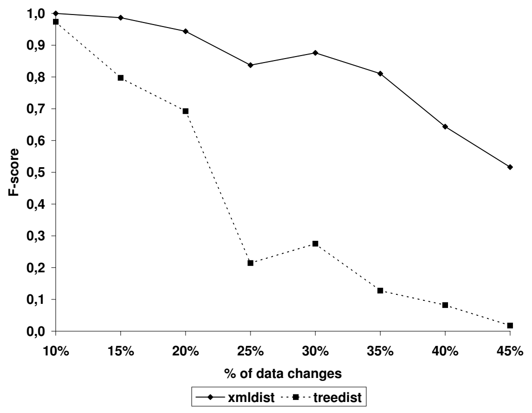
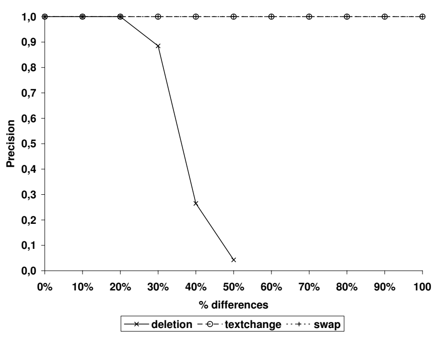
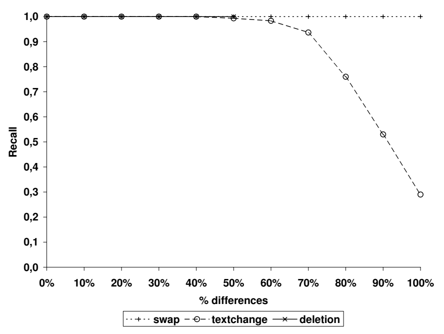


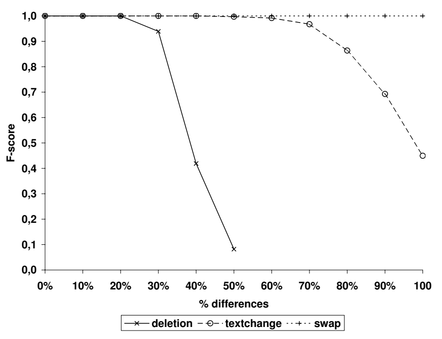
Figure 5: Results for experimental set 3: comparison between tree edit distance and structure aware XML distance



(a)



(b)



(c)

Figure 6: Precision, recall and f-score for experimental set 2

data are serious issues in XML object Identification. Such features have lead to proposals that exploit tree-edit distances to perform approximate comparisons of XML trees. However, tree-edit distances ignore the semantics implicit in element labels and nesting relationships. Furthermore, while tree-distances for unordered trees are better suited to perform approximate comparisons of XML data, their use is computationally infeasible. In this paper, we have defined a new distance for XML data, the *structure aware XML distance*, that overcomes these issues. The distance compares only portions of XML data trees whose structure suggest similar semantics. Furthermore, it performs comparison on unordered trees, without incurring in high computational costs. We have presented an algorithm to measure the distance between two trees, and discussed its complexity, that is polynomial. We also presented an experimental evaluation of our measure as the basis of an object identification approach.

Our measure is suited for detecting similar objects when the scheme of objects is approximately the same, as in the case of a single data source, or several data sources on which a schema integration activity has already been performed. We are investigating how to add more flexibility without sacrificing the gain in efficiency we have obtained. We also plan to investigate other issues related to the use of tree distances for XML comparison. In [4], the authors suggest the use of ontology-based techniques to evaluate the cost of relabelling nodes. How to balance the effects of string-comparison-based and ontology- based cost evaluation seems far from trivial.

References

[1] Philip Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, 337(1-3), 2005.

[2] Sudarshan S. Chawathe, Anand Rajaraman, Hector Garcia-Molina, and Jennifer Widom. Change detection in hierarchically structured information. In *SIGMOD, Montreal, Canada*, 1996.

[3] Gregory Cobena, Serge Abiteboul, and Amélie Marian. Detecting changes in xml documents. In *ICDE, San Jose, CA*, 2002.

[4] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava, and Ting Yu. Approximate xml joins. In *SIGMOD, Madison, Wisconsin*, 2002.

[5] Tao Jiang, Lusheng Wang, and Kaizhong Zhang. Alignment of trees - an alternative to tree edit. *Theor. Comput. Sci.*, 143(1):137–148, 1995.

[6] James Munkres. Algorithms for assignment and transportation problems. *SIAM Journal on Computing*, 5(1), March 1957.

[7] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

[8] Stanley M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[9] Dennis Shasha, Jason Tsong-Li Wang, Kaizhong Zhang, and Frank Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4), 1994.

[10] Felix Naumann Sven Puhlmann, Melanie Weis. Xml duplicate detection using sorted neighborhoods. In *EDBT, Munich, Germany*, 2006.

[11] Kuo-Chung Tai. The tree-to-tree correction problem. *J. ACM*, 26(3), 1979.

[12] Levenshtein V.I. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, 10:707–710, 1966.

[13] Melanie Weis and Felix Naumann. Dogmatix tracks down duplicates in xml. In *SIGMOD, Baltimore, Maryland, USA, June 14-16, 2005*.

[14] Melanie Weis and Felix Naumann. Detecting duplicate objects in xml documents. In *IQIS,Paris, France*, 2004.

[15] Kaizhong Zhang. A constrained edit distance between unordered labeled trees. *Algorithmica*, 15(3), 1996.

[16] Kaizhong Zhang, Richard Statman, and Dennis Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3), 1992.

[17] Kaizong Zhang and Dennis Shasha. Tree pattern matching. In Apostolico and Galil, editors, *Pattern Matching in Strings, Trees, and Arrays*. Oxford University Press, 1997.

QUEST: *QU*ery-driven *E*xploration of *S*emistructured Data with Conflic*T*s and Partial Knowledge[*]

Yan Qi
Comp. Sci. and Eng.
Arizona State Univ.
yan.qi@asu.edu

K. Selçuk Candan
Comp. Sci. and Eng.
Arizona State Univ.
candan@asu.edu

Maria Luisa Sapino
Dip. di Informatica
Univ. of Torino
mlsapino@di.unito.it

Keith W. Kintigh
Sch.of Human Evol.&Social Change
Arizona State Univ.
kintigh@asu.edu

Abstract

An important reality when integrating scientific data is the fact that data may often be "missing", partially specified, or conflicting. Therefore, in this paper, we present an assertion-based data model that captures both *value-based* and *structure-based* "nulls" in data. We also introduce the `QUEST` system, which leverages the proposed model for *Qu*ery-driven *E*xploration of *S*emistructured data with conflic*T*s and partial knowledge. Our approach to integration lies in enabling researchers to observe and resolve conflicts in the data by considering the context provided by the data requirements of a given research question. In particular, we discuss how *path-compatibility* can be leveraged, within the context of a query, to develop a high-level understanding of conflicts and nulls in data.

1 Motivation and Related Work

Through a joint effort of archaeologists and computer scientists, we are developing an integrated framework of knowledge-based collaborative tools that will provide the foundation for a shared information infrastructure for archaeology and contribute substantially to a shared knowledge infrastructure of science [21]. Today, the incapacity to integrate data across projects cripples archaeologists' and other scientists' efforts to recognize phenomena operating on large spatio-temporal scales and to conduct crucial comparative studies [20, 21]. A major challenge with integration of data is that the meaning of an archaeological observation is rarely self-evident.

1.1 Incompleteness and Inconsistencies

An important reality when integrating archaeological data is that entries (archaeological observations and interpretations) may often be "missing" or only partially specified. For example, one may not be able to associate a bone collected at a given site to the species and may use vague terms or references to a hierarchically higher concept in the biological taxonomy. Thus, researchers reach conflicting conclusions, not just because their primary data differ, but because they operationalize interpretive concepts differently [20].

Within the context of our efforts to determine the needs and challenges associated with archaeological information integration, a working group of domain experts selected datasets representing archaeological fauna recovered from two excavations in the western US [28]. The goal of the effort was to integrate these two datasets into one by using ontologies to map data codes to concepts shared by the datasets and to resolve the ambiguities (as much as possible) using ontologies. One outcome of this effort was the understanding that, even after a careful study of the data sets by the domain experts, there were parts of the data that could not be successfully mapped (e.g., while use of the actual taxonomic categories was consistent, investigators differed in how they dealt with bones that could not be fully identified). Nevertheless, in the context of a particular research question, archaeologists could identify reasonable means of addressing these inconsistencies.

Thus, reconciling data and classification schemes entails developing novel data integration techniques to allow query dependent integration, despite inherent inconsistencies. Our goal is to develop a tool to allow a researcher to extract sensibly integrated observations and consistent variables from potentially incomplete and inconsistent data archive. During query processing, the repository needs to integrate data from multiple sources, note and resolve conflicting and missing data. Where there are discrepancies or missing data, the system needs to allow the researcher to interpret results and resolve conflicts as she sees appropriate.

[*]Supported by NSF Grant "*Enabling the Study of Long-Term Human and Social Dynamics: A Cyberinfrastructure for Archaeology*"

1.2 Related Work

In general, there are many different types of null values (e.g., existential, maybe, place holder, and partial), each of which reflects different knowledge or intuitions about why a particular piece of information is missing [8]. An early attempt at modeling semistructured data with missing and partial data is presented in [23]; authors used an object-based model, where *null*, *or-valued*, and *partial set* objects are used to handle partial and missing knowledge in semi-structured data. Although it is richer than standard semistructured data models, such as Object Exchange Model (OEM) [24, 7], and Document Object Model (DOM) [1], this model is more focused on *value* nulls and does not capture inconsistencies and missing knowledge in the structure of the data. In contrast, we propose a new model for semi-structured databases where different types of null values are represented uniformly. Each entry has an associated assertion; intuitively, an entry may be thought of as being in the database iff the corresponding assertion is true. Although the idea of using assertions (constraints) to handle null values in relational databases is not new (Imielinski-Lipski [15, 16], Liu [22], Candan [8]), the use of constraints for a unified way of handling different types of nulls within the context of hierarchical data and metadata is an open problem.

Knowledge integration from diverse sources involves *matching* and *integration*. There is extensive work in the area of matching schemas and data when integrating independent sources. Our focus, in this paper, however is not on the matching, but on dealing with conflicts that arise during integration. Conflict resolution has also been studied in the context of active databases and production rule systems [3, 17]. Most of these study what to do when multiple active production rules with conflicting heads request that an atom be both added and deleted simultaneously. In contrast, we attempt to evaluate queries and resolve conflicts in answers to queries spanning multiple data sources. Furthermore, unlike the related work in this area, we will explore the application of these within semi-structured data and metadata.

In their work on nondeterministic choices in logic programming languages, Zaniolo [31, 32] and his colleagues suggest that in logic database languages, one may wish to express the fact that only one of several possible ways of satisfying an atom is nondeterministically selected. They then use this to define a choice semantics for logic programs with negation. Multiple model semantics, like the *2-valued, stable model semantics*,[12], or the *3-valued finite failure stable model semantics* [13] associate multiple, equally likely, models to the given knowledge base, each one corresponding to a possible context, or a possible consistent scenario described by the knowledge base. Problem solvers interact with *truth maintenance systems* (TMSs) [9], that record and maintain the reasons for the possible context (*belief sets*) under consideration. Sentences are associated with their justifications, which indicate what assumptions need to be changed if they need to be invalidated. In this paper, we show that we can leverage the special hierarchical structure of the data and knowledge taxonomies to develop efficient and specialized algorithms, rather than having to use general purpose truth maintenance solutions. We use path query instances to provide contexts in which conflicts can be resolved. Like us, Piazza [14] and HepToX [5] also recognize that it is unrealistic to expect an independent data source entering information exchange to agree to a global mediated schema or to perform heavyweight operations to map its schema to every other schema in the group. Unlike these, however, we recognize that while collating information from multiple sources, the knowledge that is acquired may be incomplete or inconsistent either in data values, structural relationships between data elements, or both. Yet, since the base data reflect what is currently known, data and interpretations from different sources may be important to keep *as is*, even when they may be conflicting with each other. We argue that an ultimate integrated view of multiple data sets is often not possible, and in fact is often not needed. Therefore, unlike related work [27, 10] in repairing inconsistencies in XML data using available external domain knowledge, such as functional dependencies or DTDs, our aim is to *maintain* the inconsistencies in the data and allow the researcher to resolve conflicts within the context of a given query.

1.3 Contributions of this Paper

Effective use of archaeological data requires on-the-fly data integration, where discrepancies or incomplete information is properly dealt with within the context of the given query. In this paper, we first present a data model which captures not only *value*-based, but also *structure-based* nulls in semistructured data and metadata. In particular, we suggest that it is most effective to reconcile data source observations with data requirements of a query rather than attempting global reconciliation of data sources. We refer to this as *query driven ad hoc data integration and exploration [19].* This enables us to constrain the incompatibilities of the data within the context of the question itself to reduce the complexity of the problem. In this paper, we also present an overview of a system, called `QUEST`, which we are developing to leverage the proposed model for exploratory research on the incomplete and conflicting data, based on the query driven ad hoc data integration and exploration paradigm. We are currently developing efficient algorithms to process queries on (null-valued) semi-structured data in the presence of a multitude of such alternatives, without having to materialize all alternatives.

2 Assertion-based Data Representation and Basic Null Assertions

To provide a uniform treatment to value and structure-nulls, we shred the semistructured data into its object nodes. Shredding is used in relational storage of XML data, where each node is represented as a tuple of the form $\langle node\_id, label, type, value, parent\_id\rangle$ [11, 29]. The model we describe below is reminiscent of well accepted node-labeled semi-structured data models, such as DOM [1] and their shredding into tuples [11, 29].

2.1 Constraint-based Data Representation

Let I denote the set of object node identifiers and let D be the domain of node tags[1]. We represent hierarchical data as a set, N, of object nodes, where each object node $n \in N$ is represented as a 3-tuple (id, tag, pid):

- $n.id \in I$ is the unique id of the object node,
- $n.tag \in D \cup I$ is its tag, and
- $n.pid \in I \cup \{\top\}$ is its parent's identifier.

If $n.pid = \top$, then n is referred to as the root of the data. If $n.tag \in I$, then its value is an object reference. The object nodes in N are constrained such that they collectively form a tree structure:

C1. *No node can be its own parent:* $\forall n_i \in N, n_i.id \neq n_i.pid$.

C2. *No two distinct nodes can have the same ID:* $\forall n_i \neq n_j \in N, n_i.id \neq n_j.id$.

C3. *All non-root nodes have a parent in the document:* $\forall n_i \in N, (n_i.pid = \top) \vee (n_i.pid \in I)$.

C4. *There is only one root:* $\forall n_i \in N, (n_i.pid = \top) \rightarrow (\nexists n_j \neq n_i \;\; n_j.pid = \top)$

C5. *Parent relationship between two nodes is captured by attribute "pid":* $\forall n_i, n_j \in N, parent(n_i, n_j) \leftrightarrow (n_i.id = n_j.pid)$.

C6. *Ancestor relationship between two nodes is defined using the parent relationship:*
$\forall n_i, n_j \in N, ancestor(n_i, n_j) \leftrightarrow$
$\exists m_1, m_2, \ldots, m_K \in N, K \geq 0, \;\; s.t.$
$parent(n_i, m_1) \wedge parent(m_1, m_2) \wedge$
$\ldots \wedge parent(m_K, n_j)$.

C7. *There are no cycles in the data:* $\forall n_i, n_j \in N,$
$ancestor(n_i, n_j) \rightarrow \neg ancestor(n_j, n_i)$.

These constraints describe hierarchically structured data without nulls. Next, we discuss how to extend this constraint model with value- and structure-nulls in a uniform manner.

[1]For simplicity of the presentation, we combine label, type, and value into a single tag.

2.2 Value- and Structure-Nulls

A value-null commonly occurs when the value of a node can not be determined for certain. E.g.,

- *"Node* &5*'s tag can be 4, 6, or 9."*

is a value null. Structure-nulls, on the other hand, occur when the structural relationship between the data nodes can not be determined in certain. For example,

- *"Node* &5 *is a child of node* &3 *or* &4*".*
- *"Either node* &5 *or* &6 *is a child of node* &3*".*

are structure nulls. When nodes suffer from both value- and structural uncertainties or inconsistencies, we refer to these as hybrid-nulls. Naturally, the object node based representation in Subsection 2.1 is not suitable to describe *disjunctions* or *non-existence requirements* that form the basis of various types of nulls [8]. Therefore, we present a basic *choice assertion* construct, which forms the basis of nulls.

2.3 Basic Choice Assertions

We refer to a triple, $\bar{a} = \langle \overline{id}, \overline{tag}, \overline{pid}\rangle$, where $\overline{id} \subseteq I$, $\overline{tag} \subseteq (D \cup I)$, and $\overline{pid} \subseteq (I \cup \{\top\})$, as a *basic choice assertion* (or *assertion* in short). The set of all assertions corresponding to a given data is denoted as A. For example, $\langle\{\&2, \&3\}, \{Cow, Bison\}, \{\&7, \&8\}\rangle$ is a basic choice assertion. Intuitively, each assertion in A declares constraints on id, tag, and pid related to a *single* object node in N.

Informally, a choice assertion states that "one of all possible alternatives described by the $\overline{id}$, $\overline{tag}$, and $\overline{pid}$ sets is true". If all the sets in an assertion are singular valued (e.g. of the form $\langle\{\&2\}, \{Bison\}, \{\&7\}\rangle$), then the assertion corresponds to a single object node, and vice versa: e.g., the object node $(\&2, Bison, \&7)$ could be asserted as $\langle\{\&2\}, \{Bison\}, \{\&7\}\rangle$. These types of assertions are referred to as *singular choice assertions*[2]. We classify the choice assertions into two categories: *positive* and *negative* choice assertions.

2.3.1 Positive Choice Assertions

Positive choice assertions do not contain any empty sets, but contain at least one non-singular set. For example, $\langle\{\&1, \&2\}, \{Bison, Cow\}, \{\&3\}\rangle$ is a positive choice assertion. We define the **semantics** of the positive assertion, $\bar{a}_i = \langle \overline{id}_i, \overline{tag}_i, \overline{pid}_i\rangle$, in terms of a many-to-1 mapping, $\mu : A \rightarrow N \cup \{\bot\}$, from the set, A, of assertions to nodes in N, such that

$$\mu(\bar{a}_i) = n \in N \longrightarrow (n.id \in \overline{id}_i) \wedge (n.tag \in \overline{tag}_i) \wedge (n.pid \in \overline{pid}_i).$$

The fact that the mapping, μ, is many-to-1 implies that

[2]Any data without null-values can be represented as a set of singular assertions.

- each positive assertion describes properties of a *single* object node, while
- properties of a single object node may be described by multiple assertions.

If $\mu(\bar{a}_i) = \perp$, then the assertion $\bar{a}_i$ is ignored.

Example 2.1 *Let* $\langle\{\&1\}, \{Pelvis\}, \{\&2, \&3\}\rangle$ *be a choice assertion. Informally, this assertion means that the value of the object node with id* &1 *is "Pelvis" and its parent is either* &2 *or* &3*. However, the assertion does not mean that* &1 *has two parents. In other words, this assertion is about a* single *node, whose parent we cannot ascertain without other assertions.*

2.3.2 Negative Choice Assertions

Negative choice assertions, on the other hand, contain at least one empty set. For example, $\langle\{\&1, \&2\}, \{Pelvis\}, \emptyset\rangle$ is a negative assertion. We define the **semantics** of a negative assertion in terms of the following non-existence constraints, corresponding to various empty set scenarios:

- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} \neq \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.tag \in \overline{tag}_i) \wedge (n.pid \in \overline{pid}_i)$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} \neq \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.pid \in \overline{pid}_i)$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} = \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.tag \in \overline{tag}_i)$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} = \emptyset]$
  Const.: $\nexists n \in N$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} = \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.id \in \overline{id}_i)$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} = \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.tag \in \overline{tag}_i) \wedge (n.id \in \overline{id}_i)$.
- Scenario: $[\overline{\mathbf{id}}_\mathbf{i} \neq \emptyset,\ \overline{\mathbf{tag}}_\mathbf{i} = \emptyset,\ \overline{\mathbf{pid}}_\mathbf{i} \neq \emptyset]$
  Const.: $\nexists n \in N\ \ s.t.\ \ (n.id \in \overline{id}_i) \wedge (n.pid \in \overline{pid}_i)$.

2.4 Interpretation of a Set of Assertions

A set, A, of basic choice assertions can be thought of being composed of a positive assertion set, A^+, and a negative assertion set, A^-. An interpretation of A is a data instance, which (a) satisfies the structural constraints, describing the hierarchy, in Section 2.1, (b) conforms to a mapping μ, which satisfies the constraints imposed by the positive assertion set, A^+, and (c) satisfies all the non-existence constraints imposed by the negative assertion set, A^-. Given an assertion set, there may be zero, one, or more interpretations. In a sense, the positive assertions *produce* candidate interpretations, while the negative assertions, A^-, prune the space of alternative conforming data instances.

2.5 Compatible Assertions

Assertions that conflict, for example $\langle\{\&1\}, \{Bison\}, \{\&2\}\rangle$ and $\langle\{\&1\}, \{Cow\}, \{\&3\}\rangle$, may coexist in the data. Thus, we introduce the concept of *compatibility* among assertions.

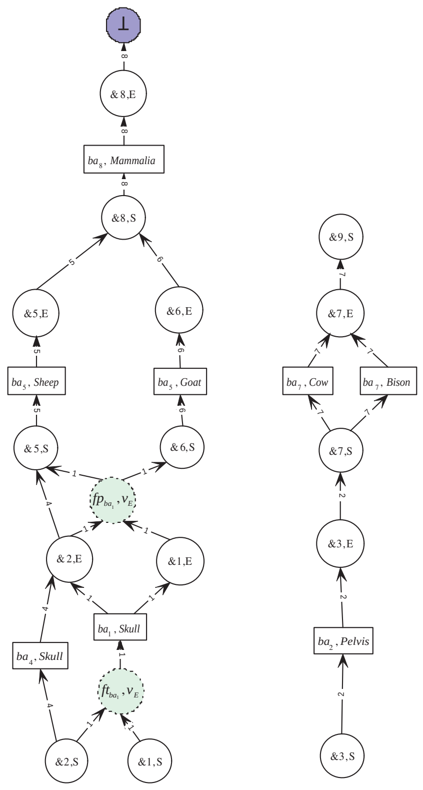


Figure 1: G^+ for a set of positive choice assertions

- A pair of positive assertions are compatible if they do neither lead to the indeterminate tags, nor imply a node with multiple parents or a cycle.
- A positive choice assertion and a negative choice assertion are compatible, if at least one choice in the positive assertion can be accepted without violating the negative constraints.

Note that although it is possible to identify consistent models (i.e., sets which consist of compatible assertions) of a given set of choice assertions, and clean the data (for instance, by choosing a *maximal* model among the alternatives), we argue that (especially in scientific data integration domain, where consistency can not be expected during research, until ultimately one model is shown to be correct) it is more meaningful to refrain from early data cleaning and resolve the conflicts within the context of the user's queries.

3 Integrated Representation of a Set of Positive Assertions

Given a set of assertions, `QUEST` integrates available positive assertions in a graph-based representation, G^+, which captures the intended structural relationships between object nodes as well as the choice semantics underlying the nulls. In this section, we provide an example of this graphical representation. The details of the model are beyond the scope of this paper.

Example 3.1 *Let us consider the assertions,*

$$\begin{aligned}
ba_1 &= \langle\{\&1, \&2\}, \{Skull\}, \{\&5, \&6\}\rangle \\
ba_2 &= \langle\{\&3\}, \{Pelvis\}, \{\&7\}\rangle \\
ba_3 &= \langle\emptyset, \{Deer\}, \emptyset\rangle \\
ba_4 &= \langle\{\&2\}, \{Skull\}, \{\&5\}\rangle \\
ba_5 &= \langle\{\&5\}, \{Sheep\}, \{\&8\}\rangle \\
ba_6 &= \langle\{\&6\}, \{Goat\}, \{\&8\}\rangle \\
ba_7 &= \langle\{\&7\}, \{Cow, Bison\}, \{\&9\}\rangle \\
ba_8 &= \langle\{\&8\}, \{Mammalia\}, \{\top\}\rangle
\end{aligned}$$

which outline hierarchical relationships among bones and taxa. For example, "Skull" belongs to the taxon "Goat", which is a branch of "Mammalia". In detail, ba_1 is a basic choice assertion, informing that there is an object node, whose tag is "Skull", but neither its identifier nor its parent can be exactly determined (i.e., the position of the skull in the hierarchy is not exactly identified). Another poorly identified data involves basic choice assertion, ba_7, where the tag of the object node can have just one of the two alternative values. The negative assertion, ba_3, states that there is no object node in the data with "Deer" as its tag.

The directed graph G^+ based on this set of positive assertions is shown in Figure 1. We use solid-lined circles to denote the graph vertices corresponding to known object ids; for each object node there are two solid vertices (start, S, and end, E). Since each assertion needs to be mapped to a single object node, dashed vertices in the graph act as mutual exclusion constraints. The possible values for the object node tags are shown in rectangular vertices. Below, we describe the salient points of the G^+ using this example.

- *First, note that, ba_3 can not be represented in G^+ as it is not a positive assertion.*
- *Since ba_1 has a non-singleton $\overline{id}$, the mutual exclusion nodes $\langle fp_{ba_1}, v_E\rangle$ and $\langle ft_{ba_1}, v_E\rangle$ (for parent and tag respectively) are introduced. Each mutual exclusion node ensures that only one of the incoming edges supported by a given basic assertion is allowed in a given interpretation of data.*
- *Some nodes, such as &9, do not have any associated assertions; thus only the corresponding start vertices, such as $\langle\&9, S\rangle$, are included; i.e., it is impossible to determine their tags or parent with the available information. In fact, G^+ may be composed of several unconnected sub-graphs.*
- *There are two different assertions, ba_1 and ba_4, describing the parent/child relationship between nodes labeled &2 and &5.*

  These two assertions have to be seen as two non-coordinated statements. Therefore, they neither support each other nor weaken the respective claims. More specifically, the non-choice assertion $ba_4 = \langle\{\&2\}, \{Skull\}, \{\&5\}\rangle$ does not make any of the two alternative choices in the assertion $ba_1 = \langle\{\&1, \&2\}, \{Skull\}, \{\&5, \&6\}\rangle$ any more likely, until interpreted by a researcher within the appropriate context.

4 Beyond Basic Assertions

Each positive basic choice assertion describes a constraint on the relationship between a node, its tag, and its parent. Since by definition of the mapping, μ, each assertion $\bar{a}_i$ is interpreted independently from the others, there is no way to correlate the choice statements that have to hold for more than one node. Thus, any null which requires a constraint on two or more (non parent-child) nodes cannot be described using a single basic choice assertion:

- *Nodes* $\underline{\&5 \textit{ and } \&6}$ *have either* &8 *or* &9 *as their common parent.*

  This statement requires a mapping, μ, where

  $$\begin{aligned}
  &(\mu(\bar{a}_i) \in N \rightarrow (\mu(\bar{a}_i).id \in \{\&5\}) \wedge (\mu(\bar{a}_i).pid \in \{\&8, \&9\})) \wedge \\
  &(\mu(\bar{a}_j) \in N \rightarrow (\mu(\bar{a}_j).id \in \{\&6\}) \wedge (\mu(\bar{a}_j).pid \in \{\&8, \&9\})) \wedge \\
  &(\mu(\bar{a}_i).pid = \mu(\bar{a}_j).pid)\,.
  \end{aligned}$$

  The last conjunct $(\mu(\bar{a}_i).pid = \mu(\bar{a}_j).pid)$ is a coordination requirement that can not be captured using basic choice assertions[3].

- *Node* &2 *has either* &5 *or* &6 *as its child; if the child is* &5 *the tag of the child is "Antelope" and if it is* &6*, the tag of the child is "Deer".*

  This statement requires a mapping μ, where

  $$\begin{aligned}
  &(\mu(\bar{a}_i) \in N \rightarrow (\mu(\bar{a}_i).id \in \{\&5\}) \wedge \\
  &\qquad (\mu(\bar{a}_i).tag \in \{\text{“}Antelope''\}) \wedge \\
  &\qquad (\mu(\bar{a}_i).pid \in \{\&2\})) \wedge \\
  &(\mu(\bar{a}_j) \in N \rightarrow (\mu(\bar{a}_j).id \in \{\&6\}) \wedge \\
  &\qquad (\mu(\bar{a}_j).tag \in \{\text{“}Deer''\}) \wedge \\
  &\qquad (\mu(\bar{a}_j).pid \in \{\&2\})) \wedge \\
  &(\mu(\bar{a}_i).pid \neq \mu(\bar{a}_j).pid).
  \end{aligned}$$

  Once again, last conjunct $(\mu(\bar{a}_i).pid \neq \mu(\bar{a}_j).pid)$ is a coordination requirement[4].

- *Node* &2 *has either the set of nodes* $\underline{\{\&5, \&6\}}$ *as its children or the set* $\underline{\{\&7,\&8\}}$.

  This statement[5] requires a mapping, μ, where

  $$\begin{aligned}
  &(\mu(\bar{a}_i) \in N \rightarrow (\mu(\bar{a}_i).id \in \{\&5\}) \wedge (\mu(\bar{a}_i).pid \in \{2\})) \wedge \\
  &(\mu(\bar{a}_j) \in N \rightarrow (\mu(\bar{a}_j).id \in \{6\}) \wedge (\mu(\bar{a}_j).pid \in \{\&2\})) \wedge \\
  &(\mu(\bar{a}_k) \in N \rightarrow (\mu(\bar{a}_k).id \in \{\&7\}) \wedge (\mu(\bar{a}_k).pid \in \{\&2\})) \wedge \\
  &(\mu(\bar{a}_l) \in N \rightarrow (\mu(\bar{a}_l).id \in \{8\}) \wedge (\mu(\bar{a}_l).pid \in \{\&2\})) \wedge \\
  &(\mu(\bar{a}_i).pid \neq \mu(\bar{a}_k).pid) \wedge \\
  &(\mu(\bar{a}_i).pid \neq \mu(\bar{a}_l).pid) \wedge \\
  &(\mu(\bar{a}_j).pid \neq \mu(\bar{a}_k).pid) \wedge \\
  &(\mu(\bar{a}_j).pid \neq \mu(\bar{a}_l).pid).
  \end{aligned}$$

  The last four conjuncts require coordination.

[3] Note that a simpler statement *"Node* &5 *has either* &8 *or* &9 *as its parent"* can be captured by a basic assertion of the form $\langle\{\&5\}, D, \{\&8, \&9\}\rangle$, **plus** the structural axiom which enforces a single parent to each node.

[4] Note that the simpler statement "*Node* &2 *has either* &5 *or* &6 *as its child*" can be captured by a basic assertion of the form $\langle\{\&5, \&6\}, D, \{\&2\}\rangle$.

[5] Note again that a statement *"Node* &2 *has either* &5 *or* &7 *as its child*" can be captured by a basic assertion of the form $\langle\{\&5, \&7\}, D, \{\&2\}\rangle$.

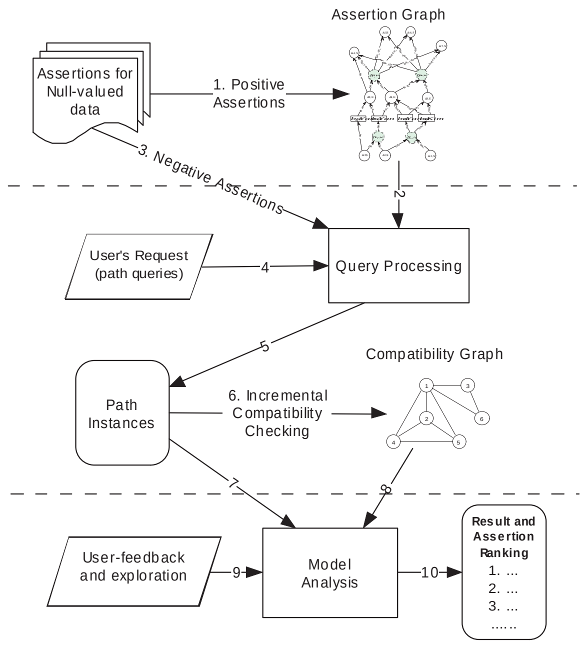

Assertion Graph
Assertions for Null-valued data
1. Positive Assertions
3. Negative Assertions
2
User's Request (path queries)
4
Query Processing
5
Compatibility Graph
Path Instances
6. Incremental Compatibility Checking
7
8
User-feedback and exploration
9
Model Analysis
10
Result and Assertion Ranking
1. ...
2. ...
3. ...
......

Figure 2: Overview of the query evaluation and exploration processes

- *Node* &5*'s tag is either "Antelope" or "Deer". Node* &6*'s tag is either "Antelope" or "Deer". Furthermore,* &5 *and* &6 *have the same tag.*

  This statement requires a mapping, μ, where

$$(\mu(\bar{a}_i) \in N \to (\mu(\bar{a}_i).id \in \{\&5\})\wedge$$
$$(\mu(\bar{a}_i).tag \in \{\text{``}Antelope'', \text{``}Deer''\})\wedge$$
$$(\mu(\bar{a}_j) \in N \to (\mu(\bar{a}_j).id \in \{\&6\})\wedge$$
$$(\mu(\bar{a}_j).tag \in \{\text{``}Antelope'', \text{``}Deer''\})\wedge$$
$$(\mu(\bar{a}_i).tag = \mu(\bar{a}_j).tag).$$

  The last conjunct requires coordination.

Comparing the complex statement examples above with their simpler counterparts, which can be captured using basic choice assertions, illustrates that the problem arises from the need to enforce coordinated selections (among possible alternatives) for multiple nodes. The implementation of coordinated choice assertions is beyond the scope of this paper.

5 Query- and Feedback-Driven Exploration Process

Figure 2 shows the outline of the query-driven exploration process underlying the QUEST: first, based on the available positive assertions, QUEST creates an assertion graph (representing the null-valued document; see Figure 1). When the user provides a path query, matching path instances corresponding to the query and satisfying the pruning constraints imposed by negative assertions are identified. In a sense

- positive assertions *produce* path instances. In case there are conflicts among the given positive assertions, this results in alternative solution *models*, consisting of intra-compatible, but pairwise-incompatible sets of paths.
- negative assertions *delete* path instances from the result. Thus, *negative assertions* can reduce the size or collapse solution models.

Naturally, the number of these solution models can be large. Therefore, a particular challenge is to postpone the computation and visualization of these alternative solution models until absolutely necessary. Thus, QUEST helps the user explore the alternative solution spaces in an informed manner without having access to explicit materializations of the solution models: QUEST first identifies an initial subset of matches to the query and constructs (in an incremental way) an intermediary path compatibility graph during query evaluation. Once the query is evaluated and the path compatibility graph is constructed, the user can interact with QUEST to turn on and off various assertions and observe how the solution set (and the solution models) are affected. Pairwise compatibility graphs of logic rules are also used in non-monotonic reasoning systems [25]. Unlike these, however, in QUEST, the compatibility graphs are not only for the base rules (or assertions), but for the result paths obtained within the context of a query. This enables the user to explore the available data within the context of a query and *drill-down* to assertions or *zoom-out* to solution models. Once the user feedback is reflected on the assertions, the user is provided with a new subset of ranked results and the feedback-based exploration process is repeated. Below, we provide sketches of these steps.

5.1 Path Query and Results

Let us focus on path queries of type $P^{\{/,//,*\}}$ [2]. In QUEST, a path query is represented as

$$q = \theta_1 t_1 \theta_2 t_2 \dots \theta_q t_q, \quad \theta_i \in \{/, //\}, \quad t_i \in D \cup \{*\},$$

where t_i are query tags (including "*" wildcards) and θ_i are parent/child or ancestor/descendant axes. An example of such a query is $'/Mammalia/Sheep/Skull'$. Results for a given path query are included in a set $R = \{r_1, r_2, \dots, r_m\}$, where for each $r_i \in R$, we have

$$r_i = v_{i,1}[e_{i,1}]v_{i,2}[e_{i,2}] \dots [e_{i,q-1}]v_{i,q}.$$

Here, $v_{i,j}$ is a label for one vertex in the assertion graph and $e_{i,j}$ is a set of labels for the assertions supporting the edge connecting the node $v_{i,j-1}$ and $v_{i,j}$. For example, the following is a result for the above query:

$$\langle \&8, E\rangle[\{\langle ba_8, -\rangle\}]\langle ba_8, Mammalia\rangle[\{\langle ba_8, -\rangle\}]\langle \&8, S\rangle[\{\langle ba_5, -\rangle\}]$$
$$\langle \&5, E\rangle[\{\langle ba_5, -\rangle\}]\langle ba_5, Sheep\rangle[\{\langle ba_5, -\rangle\}]\langle \&5, S\rangle[\{\langle ba_1, -\rangle\}]$$
$$\langle fp_{ba_1}, v_E\rangle[\{\langle ba_1, -\rangle\}]\langle \&1, E\rangle[\{\langle ba_1, -\rangle\}]\langle ba_1, Skull\rangle[\{\langle ba_1, -\rangle\}]$$
$$\langle ft_{ba_1}, v_E\rangle[\{\langle ba_1, -\rangle\}]\langle \&1, S\rangle.$$

Note that a valid path cannot contain any loops and for each data node on the path S and E vertices as well as the assertion labels need to match.

5.2 Path Compatibility Graph

Because of conflicting assertions, all results satisfying a path query might not be compatible. For example, two paths can assume that a given object node has different parents or two paths considered together may imply a loop. Furthermore, the mutual exclusion nodes introduced in Section 3 can render paths that share a given mutual exclusion node in different ways incompatible with each other. `QUEST` captures the compatibility between paths and sets of paths using a reflexive and symmetric "$\sim$" relation:

- Given two path instances p_i and p_j, $p_i \sim p_j$ iff the path instances together do not violate any structural constraints introduced in Section 2.
- Given a path instance p' and a set of path instances $P = \{p_1, p_2, ..., p_N\}$, $p' \sim P$, if and only if $\forall p_i \in P$, $p' \sim p_i$.
- Given two sets of path instances $P = \{p_1, p_2, ..., p_N\}$ and $Q = \{q_1, q_2, ..., q_M\}$, $P \sim Q$ if and only if $\forall p_i \in P, p_i \sim Q$.

Given a set of paths, P, a compatibility graph, C, captures all pairwise compatibility relationships.

5.3 Result Exploration

Let us assume that a path query q results in a set $R = \{p_1, p_2, \ldots, p_N\}$ of paths. As stated above, not all of these paths are compatible with each other. Therefore, `QUEST` provides various result exploration options to the user to enable her to get a high level understanding of the available data relative to her query:

- *Checking whether a given set, P, of paths is a model*; i.e., checking whether the given set of paths are compatible with each other. The result set, R, being a model would imply that the data does not contain any conflict relative to this query.
- Given a path p and a set of paths P, *checking whether $p \sim P$ or $p \not\sim P$.*
- *Given a path instance p and a set P, computing the number of path instances in P that are compatible with p.* This number informs the user regarding the degree of compatibility of the path p with others in P.
- *Given a path instance $p \in P$, computing the number of different models in which p occurs.* This informs the user regarding how supported each path is with the available knowledge.
- *Given a path instance $p \in P$, computing the number of models that would collapse when p is removed.* This informs the user regarding the entropy introduced by p in the integrated system.

Note that, additionally, the models themselves can be weighed based on their sizes or their compatibilities with other models. This information, then can be propagated to the weights of the paths included in these models. With these, it is possible to rank result paths and provide users with alternative exploration opportunities to observe the results, based on different definitions of likelihood (Figure 3(a)). The user can pick and choose between available result paths in an informed manner and observe the impact to the assertion and path compatibility graphs immediately. In particular, when a path is marked invalid by the user,

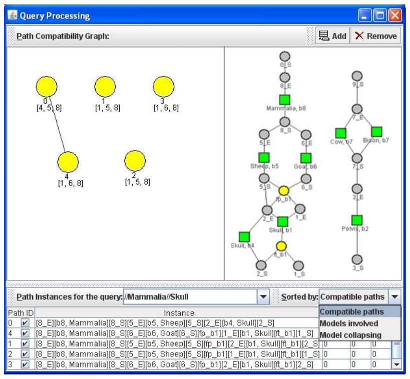
- if the path can be eliminated without affecting any other paths (by eliminating some choice in an assertion or by removing an assertion altogether), then this alternative is executed (Figure 3(b));
- if there is no way to remove it without affecting the assertions that support other paths, then the paths that might be impacted and the corresponding assertions are highlighted (Figure 3(c)).

Note that users are not always interested in ranking the result paths, but in ranking those assertions that generate and constrain the various solutions and solution models. Therefore, to support ranking of the assertions, we further propagate the various scores to the assertions on the paths. This enables the user to pick and choose between available assertions in an informed manner and observe the impacts of her actions on the solution immediately.

5.4 Computation

A model, composed of compatible result paths, corresponds to a maximal clique in the compatibility graph. Maximal cliques in a graph can be exponential in the number of vertices [26]. There are polynomial time delay algorithms for enumeration of cliques (i.e., if the graph of size n contains C cliques, the time to output all cliques is bounded by $O(n^k C)$ for some constant k) [18], but in general graphs, C can be exponential in n; for example as many as $3^{n/3}$ in Moon-Moser's graphs [26]. We, on the other hand, see that it is possible to avoid enumeration of cliques or finding of the maximal cliques in the entire compatibility graph, when supporting many of the relevant exploration tasks. For instance, the task of counting the number of maximal cliques a path occurs in can be performed by counting those maximal cliques containing only its neighbors. For sparse compatibility graphs, this can lead to significant gains in computation time. When the compatibility graph is dense, on the other hand, the number and sizes of cliques need to be estimated using alternative analysis techniques.

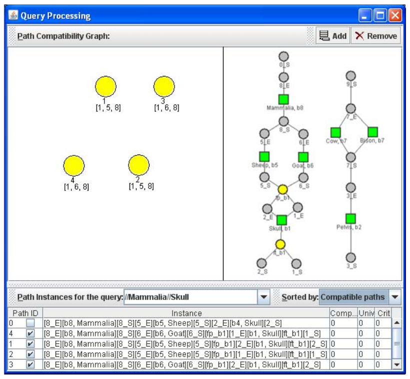
Thus, we are currently developing efficient algorithms to process queries on (null-valued) semi-structured data in the presence of a multitude of alternative models, without having to materialize all alternatives. In particular, we are exploring polynomial-time path and assertion ranking techniques based on structural analysis of the path and assertion compatibility graphs.



Path Instances for the query: //Mammalia//Skull   Sorted by: Compatible paths

Path ID		Instance	Comp...	Univ	Crit
0	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][2_E][b4, Skull][2_S]			
4	✓	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]			
1	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0
2	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]	0	0	0
3	✓	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0

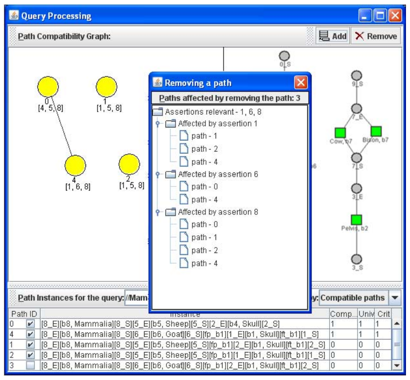
(a)



Path Instances for the query: //Mammalia//Skull   Sorted by: Compatible paths

Path ID		Instance	Comp...	Univ	Crit
0	☐	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][2_E][b4, Skull][2_S]	0	0	0
4	✓	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]	0	0	0
1	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0
2	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]	0	0	0
3	✓	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0

(b)



Paths affected by removing the path: 3

- Assertions relevant - 1, 6, 8
  - Affected by assertion 1
    - path - 1
    - path - 2
    - path - 4
  - Affected by assertion 6
    - path - 0
    - path - 4
  - Affected by assertion 8
    - path - 0
    - path - 1
    - path - 2
    - path - 4

Path Instances for the query: //Mam   Compatible paths

Path ID		Instance	Comp...	Univ	Crit
0	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][2_E][b4, Skull][2_S]	1	1	1
4	✓	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]	1	1	1
1	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0
2	✓	[8_E][b8, Mammalia][8_S][5_E][b5, Sheep][5_S][fp_b1][1_E][b1, Skull][ft_b1][1_S]	0	0	0
3	☐	[8_E][b8, Mammalia][8_S][6_E][b6, Goat][6_S][fp_b1][2_E][b1, Skull][ft_b1][2_S]	0	0	0

(c)

Figure 3: (a) Result visualization and exploration screen; (b) elimination of the path #0 changes the assertion graph accordingly; (c) elimination of path #3, on the other hand, would affect other paths in the result

5.5 Tree Queries

A tree query can be processed navigationally or split into multiple path queries and their structural joins [4, 30]. In QUEST, tree queries are handled as an extension of path query processing. After paths that satisfy the path sub-queries are identified, they need to be put together to form answers to the tree query. When paths might be incompatible, each set of paths that is put together to form an answer must be constrained to be self compatible. Therefore tree query processing involves merging of the ranked paths from multiple subqueries subject to compatibility constraints.

6 Conclusion

In this paper, we presented an *assertion*-based data model to describe hierarchical data and meta-data. We then extended this model with *basic choice assertions* which enables us to describe various types of value- and structure-based nulls in a uniform manner. We also highlighted the need for *coordinated* assertions to describe certain types of nulls. We introduced a graphical representation for hierarchical data with nulls and discussed how to enable query execution and query-driven data exploration processes using this graphical representation. We introduced the concept of path-compatibility and we highlighted how results of a query can be leveraged to have develop a high-level understanding of conflicts in the data. We also provided an overview of the QUEST system which leverages the concepts introduced in this paper to support exploratory research on incomplete and conflicting data.

References

[1] Document object model (dom) level 1 specification. http://www.w3.org/TR/REC-DOM-Level-1/.

[2] Xquery. http://www.w3.org/TR/xquery/.

[3] R. Agrawal, R. J. Cochrane, and B. G. Lindsay. On maintaining priorities in a production rule system. VLDB 1991.

[4] S. Al-Khalifa, *et al.* Structural joins: A primitive for efficient xml query pattern matching. *ICDE*, 2002.

[5] A. Bonifati, E.Q. Chang, and L.V. Lakshmanan. Heptox: Marrying xml and heterogeneity in your p2p databases. In *VLDB*, 2005. Demo.

[6] R. Boppana and M. M. Halldórsson. Approximating maximum independent sets by excluding subgraphs. SWAT, 1990

[7] P. Buneman, W. Fan, and S. Weinstein. Query optimization for semistructured data using path constraints in a deterministic data model. *DBPL*, pages 208–223, 1999.

[8] K. Candan, J. Grant, and V. Subrahmanian. A unified treatment of null values using constraints. *Information Systems Journal*, 98(1-4):99–156, May 1997.

[9] J. Doyle. A truth maintenance system. *J.of Artificial Intel.*, 12: 231–272, 1979.

[10] S. Flesca, *et al.* Repairs and consistent answers for xml data with functional dependencies. *Xsym* pages 238–253, 2003.

[11] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *IEEE Data Eng. Bulletin*,22(3):27-34, 1999.

[12] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. *International Conference and Symposium on Logic Programming.* 1988.

[13] L. Giordano, A. Martelli and M.L. Sapino". Extending negation as failure by abduction: a 3-valued stable model semantics. *J. of Logic Programming*, 1996.

[14] A. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management. In *ICDE*, 2003.

[15] T. Imielinski and W. Lipski. On representing incomplete information in a relational data base. VLDB, 1981.

[16] T. Imielinski and W. Lipski. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

[17] Y. E. Ioannidis and T. K. Sellis. Conflict resolution of rules assigning values to virtual attributes. SIGMOD, 1989.

[18] D. S. Johnson and C. H. Papadimitriou. On generating all maximal independent sets. *Info. Proc. Letters*, 27, 1988.

[19] K. W. Kintigh. *et al.* Enabling the study of long-term human and social dynamics: A cyberinfrastructure for archaeology. http://cadi.asu.edu/HSDPosterSlidesLR.ppt.

[20] K. W. Kintigh *et al.* Workshop on cybertools for archaeological data integration. http://cadi.asu.edu/, December 2004.

[21] K. W. Kintigh. The promise and challenge of archaeological data integration. *American Antiquity*, 2006. in press.

[22] K.-C. Liu and R. Sunderraman. A generalized relational model for indefinite and maybe information. TKDE, 3(1), 1991.

[23] M. Liu and T. W. Ling. A data model for semistructured data with partial and inconsistent information. *LNCS*, 1777, 2000.

[24] J. McHugh, *et al.* Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3):54–66, 1997.

[25] R. E. Mercer and V. Risch. Properties of maximal cliques of a pair-wise compatibility graph for three nonmonotonic reasoning system. In *Answer Set Programming*, 2003.

[26] J.W. Moon and L. Moser On cliques in graphs. Israel Journal of Mathematics, 3, 23–28, 1965.

[27] W. Ng. Repairing inconsistent merged xml data. In *DEXA*, pages 244–255, 2003.

[28] K. Spielmann, J. Driver, D. Grayson, E. Reitz, S. Kanza, and C. Szuter. Faunal working group. http://cadi.asu.edu.

[29] I. Tatarinov, *et al.* Storing and querying ordered XML using a relational database system. *SIGMOD*, pages 204–215, 2002.

[30] Y. Wu, J. M. Patel, and H. V. Jagadish. Structural join order selection for xml query optimization. In *ICDE*, 2003.

[31] C. Zaniolo. Design and implementation of a logic-based language for data-intensive applications. *ICLP* 1988.

[32] C. Zaniolo. *A United Semantics for Active and Deductive Databases*, chapter Rules in Database Systems. 1994.

Column Heterogeneity as a Measure of Data Quality

Bing Tian Dai
National Univ. of Singapore
daibingt@comp.nus.edu.sg

Nick Koudas
University of Toronto
koudas@cs.toronto.edu

Beng Chin Ooi
National Univ. of Singapore
ooibc@comp.nus.edu.sg

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Suresh Venkatasubramanian
AT&T Labs–Research
suresh@research.att.com

ABSTRACT

Data quality is a serious concern in every data management application, and a variety of quality measures have been proposed, including accuracy, freshness and completeness, to capture the common sources of data quality degradation. We identify and focus attention on a novel measure, *column heterogeneity*, that seeks to quantify the data quality problems that can arise when merging data from different sources. We identify desiderata that a column heterogeneity measure should intuitively satisfy, and discuss a promising direction of research to quantify database column heterogeneity based on using a novel combination of *cluster entropy* and *soft clustering*. Finally, we present a few preliminary experimental results, using diverse data sets of semantically different types, to demonstrate that this approach appears to provide a robust mechanism for identifying and quantifying database column heterogeneity.

1. MOTIVATION

Data quality is a serious concern in every data management application, severely degrading common business practices, and industry consultants often quantify the adverse impact of poor data quality in the billions of dollars annually. Data quality issues have been studied quite extensively in the literature (e.g., [3, 5, 1]). In particular, a variety of quality measures have been proposed, including accuracy, freshness and completeness, to capture the common sources of data quality degradation [6, 9]. Data profiling tools like Bellman [4] compute concise summaries of the values in database columns, to identify various errors introduced by poor database design; these include approximate keys (the presence of null values and defaults in a column may result in the approximation) and approximate functional dependencies in a table (possibly due to inconsistent values). This vision paper identifies and focuses attention on a novel measure, *column heterogeneity*, that seeks to quantify the data quality problems that can arise when merging data from different sources.

Textbook database design teaches that it is desirable for a database column to be homogeneous, i.e., all values in a column should be of the same "semantic type". For example, if a database contains email addresses, social security numbers, phone numbers, machine names and IP addresses, these semantically different types of values should be represented in separate columns. For example, the column in Figure 1(a) contains only email addresses and is quite homogeneous, even though there appears to be a wide diversity in the actual set of values present. Such homogeneity of database column values has obvious advantages, including simplicity of application-level code that accesses and modifies the database.

In practice, operational databases evolve over time to contain a great deal of "heterogeneity" in database column values. Often, this is a consequence of large scale data integration efforts that seek to preserve the "structure" of the original databases in the integrated database, to avoid having to make extensive changes to the application level code. For example, one application might use email addresses as a unique customer identifier, while another might use phone numbers for the same purpose; when their databases are integrated into a common database, it is feasible that the CUSTOMER_ID column contains both email addresses and phone numbers, both represented as strings, as illustrated in Figure 1(b). A third independently developed application that used, say, social security numbers as a customer identifier might then add such values to the CUSTOMER_ID column, when its database is integrated into the common database. As another example, two different inventory applications might maintain machine domain names (e.g., abc.def.com) and IP addresses (e.g., 105.205.105.205) in the same MACHINE_ID column for the equivalent task of identifying machines connected to the network. While these examples may appear "natural" since all of these semantically different types of values have the same function, namely, to serve as a customer identifier or a machine identifier, potential data quality problems can arise in databases accessed and modified by legacy applications that are unaware of the heterogeneity of values in the column.

For example, an application that assumes that the CUSTOMER_ID column contains only phone numbers might choose to "normalize" column values by removing all special characters (e.g., '-', '.') from the value, and writing it back into the database. While such a transformation is appropriate for phone numbers, it would clearly mangle the email addresses represented in the column and can severely degrade common business practices. For instance, in our previous example, the unanticipated transformation of email addresses in the CUSTOMER_ID column (e.g., "john.smith@noname.org" to "johnsmith@nonameorg") may mean that a large number of customers are no longer reachable.

Locating poor quality data in large operational databases is a non-trivial task, especially since the problems may not be due to the data alone, but also due to the interactions between the data and the multitude of applications that access this data (as the previous

CUSTOMER_ID
lkjkjjk@321.zzz.info
h8742@yyy.com
kkjj+@haha.org
qwerty@keyboard.us
555-1212@fax.in
alpha@beta.ga
john.smith@noname.org
jane.doe@1973law.us
gwb.dc@universe.gov
jamesbond.007@action.com

(a)

CUSTOMER_ID
lkjkjjk@321.zzz.info
h8742@yyy.com
kkjj+@haha.org
qwerty@keyboard.us
555-1212@fax.in
(908)-555.1234
973-360-0000
360-0007
8005551212
(877)-807-4596

(b)

CUSTOMER_ID
lkjkjjk@321.zzz.info
h8742@yui.com
kkjj+@haha.org
qwerty@keyboard.us
555-1212@fax.in
alpha@beta.ga
john.smith@noname.org
jane.doe@1973law.us
gwb.dc@universe.gov
(877)-807-4596

(c)

CUSTOMER_ID
123-45-6789
135-79-2468
159-24-6837
789-12-3456
987-65-4321
(908)-555.1234
973-360-0000
360-0007
8005551212
(877)-807-4596

(d)

Figure 1: Example homogeneous and heterogeneous columns.

example illustrates). Identifying heterogeneous database columns becomes important in such a scenario, permitting data quality analysts to then focus on understanding the interactions of applications with data in such columns, rather than having to simultaneously deal with the tens of thousands of columns in today's complex operational databases. If an analyst determines that a problem exists, remedial actions can include:

- modification of the applications to explicitly check for the semantic type of data (phone numbers, email addresses, etc.) assumed to exist in the table, or
- a horizontal splitting of the table to force homogeneity, along with a simpler modification of the applications accessing this table to access and update the newly created tables instead.

We next identify desiderata that a column heterogeneity measure should intuitively satisfy, and discuss a promising direction of research to quantify database column heterogeneity.

2. HETEROGENEITY: DESIDERATA

Consider the example shown in Figure 1. This illustrates many of the issues that need to be considered when coming up with a suitable measure for column heterogeneity.

Number of Semantic Types: Many semantically different types of values (email addresses, phone numbers, social security numbers, circuit identifiers, IP addresses, machine domain names, customer names, etc.) may be represented as strings in a column, with no *a priori* characterization of the set of possible semantic types present.

Intuitively, the more semantically different types of values there are in a database column, the greater should be its heterogeneity; thus, heterogeneity is better modeled as a numerical value rather than a boolean (yes/no). For example, a column with both email addresses and phone numbers (e.g., Figure 1(b)) can be said to be more heterogeneous than a column with only email addresses (e.g., Figure 1(a)) or only phone numbers.

Distribution of Semantic Types: The semantically different types of values in a database column may occur with different frequencies.

Intuitively, the relative distribution of the semantically different types of values in a column should impact its heterogeneity. For example, a column with many email addresses and phone numbers (e.g., Figure 1(b)) can be said to be more heterogeneous than a column that has mainly email addresses with just a few outlier phone numbers (e.g., Figure 1(c)), or vice versa.

Distinguishability of Semantic Types: Semantically different types of values may overlap (e.g., social security numbers and phone numbers) or be easily distinguished (e.g., email addresses and phone numbers).

Intuitively, with no a priori characterization of the set of possible semantic types present in a column, we cannot always be sure that a column is heterogeneous, and our heterogeneity measure should conservatively reflect this possibility.

The more easily distinguished are the semantically different types of values in a column, the greater should be its heterogeneity. For example, a column with roughly equal numbers of email addresses and phone numbers (e.g., Figure 1(b)) can be said to be more heterogeneous than a column with roughly equal numbers of phone numbers and social security numbers (e.g., Figure 1(d)), due to the greater similarity between the values (and hence the possibility of being of the same unknown semantic type) in the latter case.

3. QUANTIFYING HETEROGENEITY

We now discuss approaches to quantify database column heterogeneity that meet the desiderata outlined above.

Number of Semantic Types: A first approach to obtaining a heterogeneity measure is to use a *hard clustering*. By partitioning values in a database column into clusters, we can get a sense of the number of semantically different types of values in the data. However, merely counting the number of clusters does not suffice to quantify heterogeneity. Two additional issues, as outlined above, make the problem challenging: the relative sizes and the distinguishability of the clusters. A few phone numbers in a large collection of email addresses (e.g., Figure 1(c)) may look like a distinct cluster, but should not impact the heterogeneity of the column as much as having a significant number of phone numbers with the same collection of email addresses (e.g., Figure 1(b)). Again, a social security number (see the first few values in Figure 1(d)) may look similar to a phone number, and we would like the heterogeneity measure to reflect this overlap of sets of values, as well as be able to capture the idea that certain data might yield clusters that are close to each other, and other data might yield clusters that are far apart.

Distribution of Semantic Types: To take into account the relative sizes of the (possibly multiple) clusters, *cluster entropy* is a better measure for quantifying heterogeneity of data in a database column than merely counting the number of clusters. Cluster entropy is computed by assigning a "probability" to each cluster equal to the fraction of the data values it contains, and computing the entropy of the resulting distribution [2]. Consider a hard clustering $T = \{t_1, t_2, \ldots t_k\}$ of a set of n values X, where cluster t_i has n_i values, and denote $p_i = n_i/n$. Then the *cluster entropy* of the hard clustering T is the entropy of the cluster size distribution, defined as $\sum p_i \ln(1/p_i)$. By using cluster entropy, the mixture of email addresses and phone numbers in column Figure 1(b) would have a higher value of heterogeneity than the data in Figure 1(c), which consists of a few phone numbers in a collection of mainly email addresses.

Distinguishability of Semantic Types: The cluster entropy of a hard clustering does not effectively take into account distinguishability of semantic types in a column. For example, given a column with an equal number of phone numbers and social security numbers (e.g., Figure 1(d)), hard clustering could either determine the column to have one cluster (in which case its cluster entropy would be 0, which is the same as that of a column with just phone numbers) or have two equal sized clusters (in which case its cluster entropy would be $\ln(2)$, which is the same as that of a column with equal numbers of phone numbers and email addresses). Intuitively, however, the heterogeneity of such a column should be somewhere in between these two extremes to capture the uncertainty in assigning values to clusters due to the syntactic similarity of values. *Soft clustering* has the potential to address this problem; each data value in soft clustering has the flexibility of assigning a probability distribution for its cluster membership, instead of belonging to a single cluster (equivalently, assigning its entire probability distribution to a single cluster), as in hard clustering. Heterogeneity can now be computed as the cluster entropy of the soft clustering.

To summarize, the desiderata that a column heterogeneity measure should depend on the number, the distribution and the distinguishability of the semantic types of string values in a column have the potential of being satisfied by using a novel combination of *cluster entropy* and *soft clustering*. We next discuss some promising results that we have obtained by following this research direction.

4. PRELIMINARY RESULTS

As a concrete realization of our vision, we present a few experimental results using diverse data sets of semantically different types, mixed together in various ways, to provide different levels of heterogeneity.

Data Sets: We consider mixtures of four different data sets. `email` is a set of 509 email addresses collected from attendees at the 2001 SIGMOD/PODS conference, `ID` is a set of 609 employee identifiers, `phone` is a diverse collection of 3064 telephone numbers, and `circuit` is a set of 1778 network circuit identifiers. Strings in `ID` and `phone` are numeric (`phone` data contains the period as well). Strings in `email` and `circuit` are alphanumeric, and may contain special characters like '@' and '-'.

Soft Clustering: We will use the *Information Bottleneck Method*, developed by Tishby et al. [8], and implemented by Slonim in his thesis as the algorithm `iIB` [7], to compute a soft clustering of the data sets. Intuitively, `iIB` takes as input a joint distribution (X, Y), where $x \in X$ represents a string value in the data set, $y \in Y$ is chosen to represent tokens (q-grams) extracted from the string values, and the joint distribution reflects an entropy weighting of the tokens. The output of `iIB` is a cluster membership distribution $p(T|x)$ for each x, representing the conditional probability

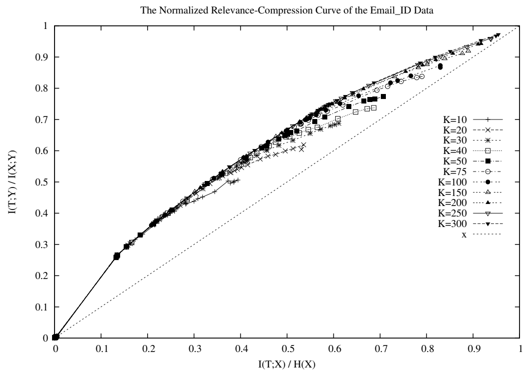


Figure 2: Rate-Distortion curve for example data.

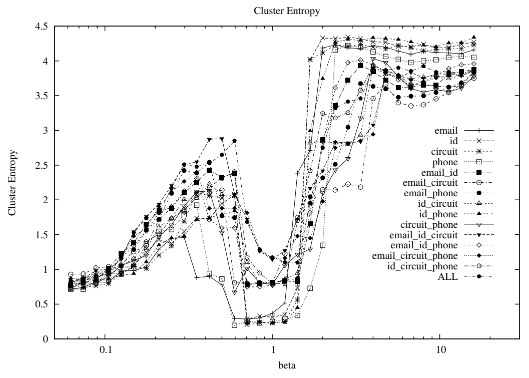


Figure 3: Cluster entropy as a function of β. The x-axis plots a normalized version of β on a logscale.

that string value x is placed in cluster $t \in T$.

Canonical Rule: `iIB` uses a parameter β that trades off cluster quality against cluster compression; increasing β increases cluster quality while decreasing cluster compression. Interestingly, *for all the data sets*, there is a unique value of β given by $\beta^* = H(X)/I(X;Y)$ (where $H(X)$ is the entropy of the data values X and $I(X;Y)$ is the mutual information between the data values X and the tokens Y), which marks the "point of diminishing returns"; after this β value, the loss we suffer from reducing the (normalized) cluster compression is not paid for by a commensurate increase in (normalized) cluster quality. This behavior can be observed in the rate distortion curve for our example data, shown in Figure 2; this curve is always concave, and the point on the curve with a slope of 1 identifies β^*. This is also the point that is the closest to the $(0, 1)$ point, which is the point representing perfect quality with no space penalty.

Cluster Entropy: Using the soft clustering output of `iIB` for different values of β in the vicinity of β^*, and computing heterogeneity by combining estimates of the cluster entropies of the various hard clusterings derived from the soft clustering via the soft clustering distribution, we empirically observed that the cluster entropy is minimized at β^*. This behavior can be observed in Figure 3. Further, the relative ordering of cluster entropy values obtained at $\beta = \beta^*$ is consistent with the expected relative heterogeneities of these data sets, as shown in Figure 4. Specifically, all the individual data sets have very small cluster entropies, and are distinguishable from the mixtures. Further, mixtures of two data

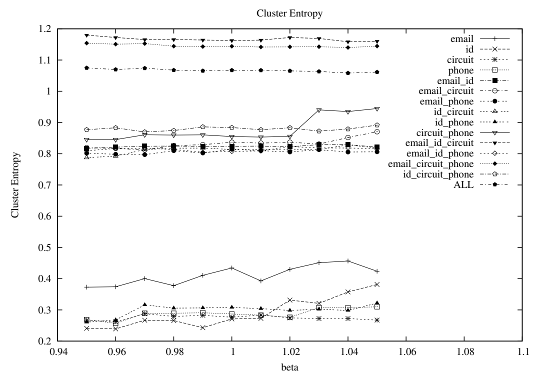


Figure 4: Cluster entropy as a measure of heterogeneity. The x-axis plots a normalized version of β on a logscale.

sets in general have lower cluster entropy than mixtures of three and four data sets. We observe that as the number of elements in the mixture increases, the heterogeneity gap decreases, and that the separations are not strict for the more heterogeneous sets; this is natural, as individual data sets may have characteristics that are somewhat similar (for example, `ID` and `phone`).

Validating the Soft Clustering: Cluster entropy appears to capture our intuitive notion of heterogeneity. However, it is derived from a soft clustering returned by the `iIB` algorithm. Does that soft clustering actually reflect natural groupings in the data? It turns out that this is indeed the case. In Figure 5, we display bitmaps that visualize the clusterings obtained for different mixtures. In this representation, columns are clusters, rows are data values, and darker probabilities are larger. For clarity, we have reordered the rows so that all data elements coming from the same source are together, and we reordered the columns based on their distributional similarities. To interpret the figure, recall that each row of a bitmap represents the cluster membership distribution of a data point. A collection of data points having the same cluster membership distributions represent the same cluster. Thus, notice how the clusters separate out quite cleanly, clearly displaying the different data mixtures. Also observe how, without having to specify k, the number of clusters, `iIB` is able to separate out the groups. Further, if we look at Figure 5(d), we notice how the clusters corresponding to `ID` and `phone` overlap and have similar cluster membership distributions, reinforcing our observation that they form two very close (not well-separated) clusters.

To summarize the experimental results, our novel combination of *cluster entropy* and *soft clustering* appears to provide a robust mechanism for identifying and quantifying database column heterogeneity.

5. CONCLUSION

In this vision paper, we identified a new data quality measure, column heterogeneity, and outlined a general approach to quantify this measure in database columns. The rapid identification of heterogeneous columns in a database with tens of thousands of columns provides a unique opportunity to understand and characterize the quality of data in today's complex operational databases, using the tools of information theory.

6. REFERENCES

[1] C. Batini, T. Catarci, and M. Scannapieco. A survey of data quality issues in cooperative information systems. In *ER*, 2004. Pre-conference tutorial.

[2] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.

[3] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.

[4] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure or how to build a data quality browser. In *SIGMOD*, 2002.

[5] T. Johnson and T. Dasu. Data quality and data cleaning: An overview. In *SIGMOD*, 2003. Tutorial.

[6] G. Mihaila, L. Raschid, and M.-E. Vidal. Querying "quality of data" metadata. In *Proc. of IEEE META-DATA Conference*, 1999.

[7] N. Slonim. *The Information Bottleneck: Theory and Applications*. PhD thesis, The Hebrew University, 2003.

[8] N. Tishby, F. Pereira, and W. Bialek. The information bottleneck method. In *Proceedings of the 37-th Annual Allerton Conference on Communication, Control and Computing*, pages 368–377, 1999.

[9] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

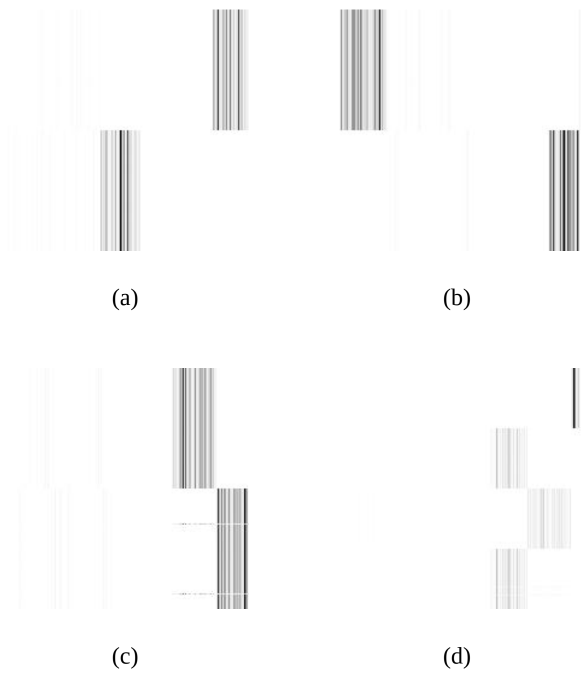


Figure 5: Soft Clustering of `email`/`ID`, `email`/`circuit`, `circuit`/`phone`, `email`/`ID`/`circuit`/`phone` mixtures.

Generic Entity Resolution with Data Confidences

David Menestrina
dmenest@cs.stanford.edu

Omar Benjelloun
benjello@cs.stanford.edu

Hector Garcia-Molina
hector@cs.stanford.edu

Stanford University

ABSTRACT

We consider the *Entity Resolution* (*ER*) problem (also known as deduplication, or merge-purge), in which records determined to represent the same real-world entity are successively located and merged. Our approach to the ER problem is *generic*, in the sense that the functions for comparing and merging records are viewed as black-boxes. In this context, managing numerical confidences along with the data makes the ER problem more challenging to define (e.g., how should confidences of merged records be combined?), and more expensive to compute. In this paper, we propose a sound and flexible model for the ER problem with confidences, and propose efficient algorithms to solve it. We validate our algorithms through experiments that show significant performance improvements over naive schemes.

1. INTRODUCTION

When data from different sources is cleansed and integrated, often multiple input records refer to the same real-world entity, e.g., to the same customer, the same product or the same organization. *Entity resolution* (ER) identifies the records that refer (or are likely to refer) to the same entity, and merges these records. A merged record becomes a "composite" of the source records. In general, a merged record needs to be compared and possibly merged with other records, since the composition of information may now make it possible to identify new relationships. For instance, say record r_1 gives the name and driver's license of a person, while record r_2 gives an address and the same driver's license number. Say we merge r_1 and r_2 based on the matching driver's license. Now we have both a name and an address for this person, and this combined information may make it possible to connect this merged record with say r_3, containing a similar name and address. Note that neither r_1 nor r_2 may match with r_3, because they do not contain the combined information that the merged record has. Entity resolution is also known as deduplication and record linkage.

Often, numerical confidences (or data quality) play a role in entity resolution. For instance, the input records may come from unreliable sources, and have confidences or quality associated with them. The comparisons between records may also yield confidences that represent how likely it is that the records refer to the same real-world entity. Similarly, the merge process may introduce additional uncertainties, as it may not be certain how to combine the information from different records. In each application domain, the interpretation of the quality or confidence numbers may be different. For instance, a confidence number may represent a "belief" that a record faithfully reflects data from a real-world entity, or it may represent how "accurate" a record is.

Even though ER is a central problem in information integration, and even though confidences are often an integral part of resolution, relatively little is known about how to *efficiently* deal with confidences. Specifically, confidences may make the ER process computationally more expensive, as compared to a scenario where confidences are not taken into account. For instance, without confidences, the order in which records are merged may be unimportant, and this property can be used to find efficient ER strategies. However, confidences may make order critical. For instance, say we merge r_1 to r_2 and then to r_3, giving us a record r_{123}. Because r_1 and r_2 are "very similar", we may have a high confidence in the intermediate result, which then gives us high confidence in r_{123}. However, say we merge r_1 to r_3 and then to r_2, giving us record r_{132}. In this case, r_1 and r_3 may not be "that similar", leading t a lower confidence r_{132}. Records r_{123} and r_{132} may even have the same attributes, but may have different confidences because they were derived differently. Thus, ER must consider many more potential derivations of composite records.

Our goal in this paper is to explore ways to reduce the high computational costs of ER with confidences. We wish to achieve this goal without making too many assumptions about the confidence model and how confidences are computed when record are merged. Thus, we will use a generic *black-box model* for the functions that compare records, that merge records, and that compute new confidences. We will then postulate properties that these functions may have: if the properties hold, then efficient ER with confidences will be possible. If they do not hold, then one must run a more-general version of ER (as we will detail here). Since we use generic match and merge functions, the algorithms we present can be used in many domains. All that is required is to check what properties the match and merge functions have, and then to select the appropriate algorithm.

The contributions of this paper are the following:

- We define a generic framework for managing confidences during entity resolution (Sections 2 and 3).
- We present Koosh, an algorithm for resolution when confidences are involved (Section 4).
- We present three improvements over Koosh that can significantly reduce the amount of work during resolution: domination, packages and thresholds. We identify properties that must hold in order for these improvements to be achievable (Sections 5, 6, and 7).
- We evaluate the algorithms and quantify the potential performance gains (Section 8).

2. MODEL

Each *record* r consists of a *confidence* $r.\mathcal{C}$ and a set of *attributes* $r.\mathcal{A}$. For illustration purposes, we can think of each attribute as a label-value pair, although this view is not essential for our work. For example, the following record may represent a person:

0.7 [name: "Fred", age: $\{45, 50\}$, zip: 94305]

In our example, we write $r.\mathcal{C}$ (0.7 in this example) in front of the attributes. (A record's confidences could simply be considered as one of its attributes, but here we treat confidences separately to make it easier to refer to them.) Note that the value for an attribute may be a set. In our example, the age attribute has two values, 45 and 50. Multiple values may be present in input records, or arise during integration: a record may report an age of 45 while another one reports 50. Some merge functions may combine the ages into a single number (say, the average), while others may decide to keep both possibilities, as shown in this example.

Note that we are using a single number to represent the confidence of a record. We believe that single numbers (in the range 0 to 1) are the most common way to represent confidences in the ER process, but more general confidence models are possible. For example, a confidence could be a vector, stating the confidences in individual attributes. Similarly, the confidence could include lineage information explaining how the confidence was derived. However, these richer models make it harder for application programmers to develop merge functions (see below), so in practice, the applications we have seen all use a single number.

Generic ER relies on two black-box functions, the *match* and the *merge* function, which we will assume here work on two records at a time:

- A match function $M(r, s)$ returns true if records r and s represent the same entity. When $M(r, s) = true$ we say that r and s match, denoted $r \approx s$.
- A merge function creates a composite record from two matching records. We represent the merge of record r and s by $\langle r, s \rangle$.

Note that the match and merge functions can use global information to make their decisions. For instance, in an initialization phase we can compute say the distribution of terms used in product descriptions, so that when we compare records we can take into account these term frequencies. Similarly, we can run a clustering algorithm to identify sets of input records that are "similar." Then the match function can consult these results to decide if records match. As new records are generated, the global statistics need to be updated (by the merge function): these updates can be done incrementally or in batch mode, if accuracy is not essential.

The pairwise approach to match and merge is often used in practice because it is easier to write the functions. (For example, ER products from IBM, Fair Isaac, Oracle, and others use pairwise functions.) For instance, it is extremely rare to see functions that merge more than two records at a time. To illustrate, say we want to merge 4 records containing different spellings of the name "Schwartz." In principle, one could consider all 4 names and come up with some good "centroid" name, but in practice it is more common to use simpler strategies. For example, we can just accumulate all spellings as we merge records, or we can map each spelling to the closest name in a dictionary of canonical names. Either approach can easily be implemented in a pairwise fashion.

Of course, in some applications pairwise match functions may not be the best approach. For example, one may want to use a set-based match function that considers a set of records and identifies the pair that should be matched next, i.e., $M(S)$ returns records $r, s \in S$ that are the best candidates for merging. Although we do not cover it here, we believe that the concepts we present here (e.g., thresholds, domination) can also be applied when set-based match functions are used, and that our algorithms can be modified to use set-based functions.

Pairwise match and merge are generally not arbitrary functions, but have some properties, which we can leverage to enable efficient entity resolution. We assume that the match and merge functions satisfy the following properties:

- *Commutativity:* $\forall r, s,\ r \approx s \Leftrightarrow s \approx r$ and if $r \approx s$ then $\langle r, s \rangle = \langle s, r \rangle$.
- *Idempotence:* $\forall r, r \approx r$ and $\langle r, r \rangle = r$.

We expect these properties to hold in almost all applications (unless the functions are not property implemented). In one ER application we studied, for example, the implemented match function was not idempotent: a record would not match itself if the fields used for comparison were missing. However, it was trivial to add a comparison for record equality to the match function to achieve idempotence. (The advantage of using an idempotent function will become apparent when we see the efficient options for ER.)

Some readers may wonder if merging two identical records should really give the same record. For example, say the records represent two observations of some phenomena. Then perhaps the merge record should have a higher confidence because there are two observations? The confidence would only be higher if the two records represent independent observations, not if they are identical. We assume that independent observations would differ in some way, e.g., in an attribute recording the time of observation. Thus, two *identical* records should really merge into the same record.

3. GENERIC ENTITY RESOLUTION

Given the match and merge functions, we can now ask what is the correct result of an entity resolution algorithm. It is clear that if two records match, they should be merged together. If the merged record matches another record, then those two should be merged together as well. But what should happen to the original matching records? Consider:

$r_1 = 0.8[\mathit{name} : \text{Alice}, \mathit{areacode} : 202]$

$r_2 = 0.7[\mathit{name} : \text{Alice}, \mathit{phone} : 555\text{-}1212]$.

The merge of the two records might be:

$r_{12} = 0.56[\mathit{name} : \text{Alice},\ \mathit{areacode} : 202,\ \mathit{phone} : 555\text{-}1212]$

In this case, the merged record has all of the information in r_1 and r_2, but with a lower confidence. So dropping the original two records would lose information. Therefore, to be conservative, the result of an entity resolution algorithm must contain the original records as well as records derived through merges. Based on this intuition, we define the correct result of entity resolution as follows.

DEFINITION 3.1. *Given a set of records R, the* result of Entity Resolution $ER(R)$ *is the smallest set S such that:*

1. $R \subseteq S$,
2. *For any records* $r_1, r_2 \in S$, *if* $r_1 \approx r_2$, *then* $\langle r_1, r_2 \rangle \in S$.

We say that S_1 is smaller than S_2 if $S_1 \subseteq S_2$. The terminology "smallest" implies that there exists a unique result, which is proven in the extended version of this paper [15].

Intuitively, $ER(R)$ is the set of all records that can be derived from the records in R, or from records derived from them. A natural "brute-force" algorithm (BFA) for computing $ER(R)$ involves comparing all pairs, merging those that match, and repeating until no new records are found. This algorithm is presented formally in the extended version of this paper [15].

4. KOOSH

A brute-force algorithm like BFA is inefficient, essentially because the results of match comparisons are forgotten after every iteration. As an example, suppose $R = r_1, r_2$, $r_1 \approx r_2$, and $\langle r_1, r_2 \rangle$ doesn't match anything. In the first round, BFA will compare r_1 with r_2, and merge them together, adding $\langle r_1, r_2 \rangle$ to the set. In the second round, r_1 will be compared with r_2 a second time, and then merged together again. This comparison is redundant. In data sets with more records, the number of redundant comparisons is even greater.

We give in Figure 1 the Koosh algorithm, which improves upon BFA by removing these redundant comparisons. The algorithm works by maintaining two sets. R is the set of records that have not been compared yet, and R' is a set of records that have all been compared with each other. The algorithm works by iteratively taking a record r out of R, comparing it to every record in R', and then adding it to R'. For each record r' that matched r, the record $\langle r, r' \rangle$ will be added to R.

Using our simple example, we illustrate the fact that redundant comparisons are eliminated. Initially, $R = \{r_1, r_2\}$ and $R' = \emptyset$. In the first iteration, r_1 is removed from R and compared against everything in R'. There is nothing in R', so there are no matches, and r_1 is added to R'. In the second iteration, r_2 is removed and compared with everything in R', which consists of r_1. Since $r_1 \approx r_2$, the two records are merged and $\langle r_1, r_2 \rangle$ is added to R. Record r_2 is added to R'. In the third iteration, $\langle r_1, r_2 \rangle$ is removed from R and compared against r_1 and r_2 in R'. Neither matches, so $\langle r_1, r_2 \rangle$ is added to R'. This leaves R empty, and the algorithm terminates. In the above example, r_1 and r_2 were compared against each other only once, so the redundant comparison that occurred in BFA has been eliminated.

The Koosh algorithm correctly computes $ER(R)$. Moreover, it is efficient. No other algorithm that computes $ER(R)$ can perform fewer comparisons. These facts are proven in the extended version of this paper.

THEOREM 4.6. Koosh is optimal, in the sense that no algorithm that computes $ER(R)$ makes fewer comparisons.

```
 1: input: a set $R$ of records
 2: output: a set $R'$ of records, $R' = ER(R)$
 3: $R' \leftarrow \emptyset$
 4: while $R \neq \emptyset$ do
 5:    $r \leftarrow$ a record from $R$
 6:    remove $r$ from $R$
 7:    for all $r' \in R'$ do
 8:       if $r \approx r'$ then
 9:          $merged \leftarrow \langle r, r' \rangle$
10:          if $merged \notin R \cup R' \cup \{r\}$ then
11:             add $merged$ to $R$
12:          end if
13:       end if
14:    end for
15:    add $r$ to $R'$
16: end while
17: return $R'$
```

Algorithm 1: The Koosh algorithm for ER(R)

5. DOMINATION

Even though Koosh is quite efficient, it is still very expensive, especially since the answer it must compute can be very large. In this section and the next two, we explore ways to tame this complexity, by exploiting additional properties of the match and merge functions (Section 6), or by only computing a still-interesting subset of the answer (using thresholds, in Section 7, or the notion of domination, which we introduce next).

To motivate the concept of domination, consider the following records r_1 and r_2, that match, and merge into r_3:
$r_1 = 0.8[name : \text{Alice},\ areacode : 202]$
$r_2 = 0.7[name : \text{Alice},\ phone : \text{555-1212}]$.
$r_3 = 0.7[name : \text{Alice},\ areacode : 202,\ phone : \text{555-1212}]$.
The resulting r_3 contains all of the attributes of r_2, and its confidence is the same. In this case it is natural to consider a "dominated" record like r_2 to be redundant and unnecessary. Thus, a user may only want the ER answer to contain only non-dominated records. These notions are formalized by the following definitions.

DEFINITION 5.1. *We say that a record r* dominates *a record s, denoted* $s \leq r$, *if the following two conditions hold:*

1. $s.\mathcal{A} \subseteq r.\mathcal{A}$
2. $s.\mathcal{C} \leq r.\mathcal{C}$

DEFINITION 5.2. *Given a set of base records R, the non-dominated entity-resolved set,* $NER(R)$ *contains all records in* $ER(R)$ *that are* non-dominated. *That is,* $r \in NER(R)$ *if and only if* $r \in ER(R)$ *and there does not exist any* $s \in ER(R)$, $s \neq r$, *such that* $r \leq s$.

Note that just like $ER(R)$, $NER(R)$ may be infinite. In the case that $ER(R)$ is finite, one way to compute $NER(R)$ is to first compute $ER(R)$ and then remove dominated records. This strategy does not save much effort since we still have to compute $ER(R)$. A significant performance improvement is to discard a dominated record as soon as it is found in the resolution process, on the premise that a dominated record will never participate in the generation of a non-dominated record. This premise is stated formally as follows:

- *Domination Property:* If $s \leq r$ and $s \approx x$ then $r \approx x$ and $\langle s, x\rangle \leq \langle r, x\rangle$.

This domination property may or may not hold in a given application. For instance, let us return to our r_1, r_2, r_3 example at the beginning of this section. Consider a fourth record $r_4 = 0.9[\textit{name} : \text{Alice}, \textit{areacode} : 717, \textit{phone} : 555\text{-}1212, \textit{age} : 20]$. A particular match function may decide that r_4 does *not* match r_3 because the area codes are different, but r_4 and r_2 may match since this conflict does not exist with r_2. In this scenario, we cannot discard r_2 when we generate a record that dominates it (r_3), since r_2 can still play a role in some matches.

However, in applications where having more information in a record can never reduce its match chances, the domination property can hold and we can take advantage of it. If the domination property holds then we can throw away dominated records as we find them while computing $NER(R)$. We prove this fact in the extended version of this paper.

5.1 Algorithm Koosh-ND

Koosh can be modified to eliminate dominated records early as follows. First, Koosh-ND begins by removing all dominated records from the input set. Second, within the body of the algorithm, whenever a new merged record m is created (line 10), the algorithm checks whether m is dominated by any record in R or R'. If so, then m is immediately discarded, before it is used for any unnecessary comparisons. Note that we do *not* check if m dominates any other records, as this check would be expensive in the inner loop of the algorithm. Finally, since we do not incrementally check if m dominates other records, we add a step at the end to remove all dominated records from the output set.

Koosh-ND relies on two complex operations: removing all dominated records from a set and checking if a record is dominated by a member of a set. These seem like expensive operations that might outweigh the gains obtained by eliminating the comparisons of dominated records. However, using an inverted list index that maps label-value pairs to the records that contain them, we can make these operations quite efficient.

The correctness of Koosh-ND is proven in the extended version of this paper.

6. THE PACKAGES ALGORITHM

In Section 3, we illustrated why ER with confidences is expensive, on the records r_1 and r_2 that merged into r_3:

$r_1 = 0.8[\textit{name} : \text{Alice}, \textit{areacode} : 202]$,

$r_2 = 0.7[\textit{name} : \text{Alice}, \textit{phone} : 555\text{-}1212]$,

$r_3 = 0.56[\textit{name} : \text{Alice}, \textit{areacode} : 202, \textit{phone} : 555\text{-}1212]$.

Recall that r_2 cannot be discarded essentially because it has a higher confidence than the resulting record r_3. However, notice that other than the confidence, r_3 contains more label-value pairs, and hence, if it were not for its higher confidence, r_2 would not be necessary. This observation leads us to consider a scenario where the records minus confidences can be resolved efficiently, and then to add the confidence computations in a second phase.

In particular, let us assume that our merge function is "information preserving" in the following sense: When a record r merges with other records, the information carried by r's attributes is not lost. We formalize this notion of "information" by defining a relation "$\sqsubseteq$": $r \sqsubseteq s$ means that the attributes of s carry more information than those of r. We assume that this relation is transitive. Note that $r \sqsubseteq s$ and $s \sqsubseteq r$ does *not* imply that $r = s$; it only implies that $r.\mathcal{A}$ carries as much information as $s.\mathcal{A}$.

The property that merges are information preserving is formalized as follows:

- *Property P1:* If $r \approx s$ then $r \sqsubseteq \langle r, s\rangle$ and $s \sqsubseteq \langle r, s\rangle$.
- *Property P2:* If $s \sqsubseteq r$, $s \approx x$ and $r \approx x$, then $\langle s, x\rangle \sqsubseteq \langle r, x\rangle$

For example, a merge function that unions the attributes of records would have properties P1 and P2. Such functions are common in "intelligence gathering" applications, where one wishes to collect all information known about entities, even if contradictory. For instance, say two records report different passport numbers or different ages for a person. If the records merge (e.g., due to evidence in other attributes) such applications typically gather all the facts, since the person may be using fake passports reporting different ages.

Furthermore, we assume that adding information to a record does not change the outcome of match. In addition, we also assume that the match function does not consider confidences, only the attributes of records. These characteristics are formalized by:

- *Property P3:* If $s \sqsubseteq r$ and $s \approx x$, then $r \approx x$.

Having a match function that ignores confidences is not very constraining: If two records are unlikely to match due to low confidences, the merge function can still assign a low confidence to the resulting record to indicate it is unlikely. The second aspect of Property P3 rules out "negative evidence": adding information to a record cannot rule out a future match. However, negative information can still be handled by decreasing the confidence of the resulting record.

The algorithm of Figure 2 exploits these properties to perform ER more efficiently. It proceeds in two phases: a first phase bypasses confidences and groups records into disjoint packages. Because of the properties, this first phase can be done efficiently, and records that fall into different packages are known not to match. The second phase runs ER with confidences on each package separately. We next explain and justify each of these two phases.

6.1 Phase 1

In Phase 1, we may use any generic ER algorithm, such as those in [2] to resolve the base records, but with some additional bookkeeping. For example, when two base records r_1 and r_2 merge into r_3, we combine all three records together into a *package* p_3. The package p_3 contains two things: (i) a root $r(p_3)$ which in this case is r_3, and (ii) the base records $b(p_3) = \{r_1, r_2\}$.

Actually, base records can also be viewed as packages. For example, record r_2 can be treated as package p_2 with $r(p_2) = r_2$, $b(p_2) = \{r_2\}$. Thus, the algorithm starts with a set of packages, and we generalize our match and merge functions to operate on packages.

For instance, suppose we want to compare p_3 with a package p_4 containing only base record r_4. That is, $r(p_4) = r_4$ and $b(p_4) = \{r_4\}$. To compare the packages, we only compare their roots: That is, $M(p_3, p_4)$ is equivalent to $M(r(p_3), r(p_4))$, or in this example equivalent to $M(r_3, r_4)$. (We use the same symbol M for record and package matching.) Say these records do match, so we generate a new

```
1: input: a set $R$ of records
2: output: a set $R'$ of records, $R' = ER(R)$
3: Define for Packages:
4: match: $p \approx p'$ iff $r(p) \approx r(p')$
5: merge: $\langle p, p' \rangle = p''$ :
         with root: $r(p'') = \langle r(p), r(p') \rangle$
         and base: $b(p'') = b(p) \cup b(p')$
6: Phase 1:
7: $P \leftarrow \emptyset$
8: for all records $rec$ in $R$ do
9:    create package $p$:
         with root: $r(p) = rec$
         and base: $b(p) = \{rec\}$
10:   add $p$ to $P$
11: end for
12: compute $P' = ER(P)$ (e.g., using Koosh) with the
    following modification: Whenever packages $p, p'$ are
    merged into $p''$, delete $p$ and $p'$ immediately, then proceed.
    Phase 2:
13: $R' \leftarrow \emptyset$
14: for all packages $p \in P'$ do
15:   compute $Q = ER(b(p))$ (e.g. using Koosh)
16:   add all records in $Q$ to $R'$
17: end for
18: return $R'$
```

Algorithm 2: The Packages algorithm

package p_5 with $r(p_5) = \langle r_3, r_4 \rangle$ and $b(p_5) = b(p_3) \cup b(p_4) = \{r_1, r_2, r_4\}$.

The package p_5 represents not only the records in $b(p_5)$, but also any records that can be derived from them. That is, p_5 represents all records in $ER(b(p_5))$. For example, p_5 implicitly represents the record $\langle r_1, r_4 \rangle$, which may have a higher confidence that the root of p_5. Let us refer to the complete set of records represented by p_5 as $c(p_5)$, i.e., $c(p_5) = ER(b(p_5))$. Note that the package does not contain $c(p_5)$ explicitly, the set is just implied by the package.

The key property of a package p is that the attributes of its root $r(p)$ carry more information (or the same) than the attributes of any record in $c(p)$, that is for any $s \in c(p)$, $s \sqsubseteq r(p)$. This property implies that any record u that does *not* match $r(p)$, cannot match any record in $c(p)$.

Theorem 6.3. For any package p, if a record u does not match the root $r(p)$, then u does not match any record in $c(p)$.

This fact in turn saves us a lot of work! In our example, once we wrap up base records r_1, r_2 and r_4 into p_5, we do not have to involve them in any more comparisons. We only use $r(p_5)$ for comparing against other packages. If p_5 matches some other package p_8 (i.e., the roots match), we merge the packages. Otherwise, p_5 and p_8 remain separate since they have nothing in common. That is, nothing in $c(p_5)$ matches anything in $c(p_8)$.

6.2 Phase 2

At the end of Phase 1, we have resolved all the base records into a set of independent packages. In Phase 2 we resolve the records in each package, now taking into account confidences. That is, for each package p we compute $ER(b(p))$, using an algorithm like Koosh. Since none of the records from other packages can match a record in $c(p)$, the $ER(b(p))$ computation is completely independent from the other computations. Thus, we save a very large number of comparisons in this phase where we must consider the different order in which records can merge to compute their confidences. The more packages that result from Phase 1, the finer we have partitioned the problem, and the more efficient Phase 2 will be.

6.3 Packages-ND

As with Koosh, there is a variant of Packages that handles domination. To remove dominated records from the final result, we simply use Koosh-ND in Phase 2 of the Packages algorithm. Note that it is not necessary to explicitly remove dominated packages in Phase 1. To see this, say at some point in Phase 1 we have two packages, p_1 and p_2 such that $r(p_1) \leq r(p_2)$, and hence $r(p_1) \sqsubseteq r(p_2)$. Then p_1 will match p_2 (by Property P3 and idempotence), and both packages will be merged into a single one, containing the base records of both.

7. THRESHOLDS

Another opportunity to reduce the resolution workload lies within the confidences themselves. Some applications may not need to know every record that could possibly be derived from the input set. Instead, they may only care about the derived records that are above a certain confidence threshold.

Definition 7.1. *Given a threshold value T and a set of base records R, we define the above-threshold entity-resolved set, $TER(R)$ that contains all records in $ER(R)$ with confidences above T. That is, $r \in TER(R)$ if and only if $r \in ER(R)$ and $r.\mathcal{C} \geq T$.*

As we did with domination, we would like to remove below-threshold records, not after completing the resolution process (as suggested by the definition), but as soon as they appear. However, we will only be able to remove below-threshold records if they cannot be used to derive above-threshold records. Whether we can do that depends on the semantics of confidences.

As we mentioned earlier, models for the interpretation of confidences vary. Under some interpretations, two records with overlapping information might be considered as independent evidence of a fact, and the merged record will have a higher confidence than either of the two base records.

Other interpretations might see two records, each with their own uncertainty, and a match and merge process which is also uncertain, and conclude that the result of a merge must have lower confidence than either of the base records. For example, one interpretation of $r.\mathcal{C}$ could be that it is the probability that r correctly describes a real-world entity. Using the "possible worlds" metaphor [13], if there are N equally-likely possible worlds, then an entity containing at least the attributes of r will exist in $r.\mathcal{C} \times N$ worlds. With this interpretation, if r_1 correctly describes an entity with probability 0.7, and r_2 describes an entity with probability 0.5, then $\langle r_1, r_2 \rangle$ cannot be true in more worlds than r_2, so its confidence would have to be less than or equal to 0.5.

To be more formal, some interpretations, such as the example above, will have the following property.

- *Threshold Property:* If $r \approx s$ then $\langle r, s \rangle.\mathcal{C} \leq r.\mathcal{C}$ and $\langle r, s \rangle.\mathcal{C} \leq s.\mathcal{C}$.

Given the threshold property, we can compute $TER(R)$ more efficiently. In the extended version of this paper, we prove that if the threshold property holds, then all results can be obtained from above-threshold records.

7.1 Algorithms Koosh-T and Koosh-TND

As with removing dominated records, Koosh can be easily modified to drop below-threshold records. First, we add an initial scan to remove all base records that are already below threshold. Then, we simply add the following conjunct to the condition of Line 10 of the algorithm:

$$merged.\mathcal{C} \geq T$$

Thus, merged records are dropped if they are below the confidence threshold.

THEOREM 7.2. When $TER(R)$ is finite, Koosh-T terminates and computes $TER(R)$.

By performing the same modification as above on Koosh-ND, we obtain the algorithm Koosh-TND, which computes the set $NER(R) \cap TER(R)$ of records in $ER(R)$ that are neither dominated nor below threshold.

7.2 Packages-T and Packages-TND

If the threshold property holds, Koosh-T or Koosh-TND can be used for Phase 2 of the Packages algorithm, to obtain algorithm Packages-T or Packages-TND. In that case, below-threshold and/or dominated records are dropped as each package is expanded.

8. EXPERIMENTS

To summarize, we have discussed three main algorithms: BFA, Koosh, and Packages. For each of those basic three, there are three variants, adding in thresholds (T), non-domination (ND), or both (TND). In this section, we will compare the three algorithms against each other using both thresholds and non-domination. We will also investigate how performance is affected by varying threshold values, and, independently, by removing dominated records.

To test our algorithms, we ran them on synthetic data. Synthetic data gives us the flexibility to carefully control the distribution of confidences, the probability that two records match, as well as other important parameters. Our goal in generating the data was to emulate a realistic scenario where n records describe various aspects of m real-world entities ($n > m$). If two of our records refer to the same entity, we expect them to match with much higher probability than if they referred to different entities.

To emulate this scenario, we assume that the real-world entities can be represented as points on a number line. Records about a particular entity with value x contain an attribute A with a value "close" to x. (The value is normally distributed with mean x, see below.) Thus, the match function can simply compare the A attribute of records: if the values are close, the records match. Records are also assigned a confidence, as discussed below.

For our experiments we use an "intelligence gathering" merge function as discussed in Section 6, which unions attributes. Thus, as a record merges with others, it accumulates A values and increases its chances of matching other records related to the particular real-world entity.

To be more specific, our synthetic data was generated using the following parameters (and their default values):

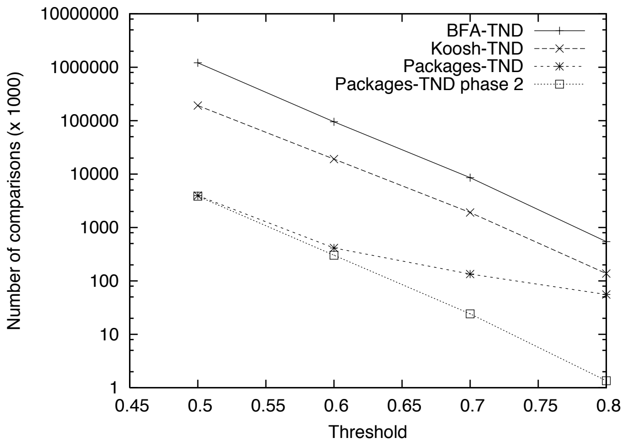


Figure 1: Thresholds vs. Matches

- n, the number of records to generate (default: 1000)
- m, the number of entities to simulate (default: 100)
- $margin$, the separation between entities (default: 75)
- σ, the standard deviation of the normal curve around each entity. (default: 10)
- μ_c, the mean of the confidence values (default: 0.8)

To generate one record r, we proceed as follows: First, pick a uniformly distributed random integer i in the range $[0, m-1]$. This integer represents the value for the real-word entity that r will represent. For the A value of r, generate a random floating point value v from a normal distribution with standard deviation σ and a mean of $margin \cdot i$. To generate r's confidence, compute a uniformly distributed value c in the range $[\mu_c - 0.1, \mu_c + 0.1]$ (with $\mu_c \in [0.1, 0.9]$ so that c stays in $[0, 1]$). Now create a record r with $r.\mathcal{C} = c$ and $r.\mathcal{A} = \{A : v\}$. Repeat all of these steps n times to create n synthetic records.

Our merge function takes in the two records r_1 and r_2, and creates a new record r_m, where $r_m.\mathcal{C} = r_1.\mathcal{C} \times r_2.\mathcal{C}$ and $r_m.\mathcal{A} = r_1.\mathcal{A} \cup r_2.\mathcal{A}$. The match function detects a match if for the A attribute, there exists a value v_1 in $r_1.\mathcal{A}$ and a value v_2 in $r_2.\mathcal{A}$ where $|v_1 - v_2| < k$, for a parameter k chosen in advance ($k = 25$ except where otherwise noted).

Naturally, our first experiment compares the performance of our three algorithms, BFA-TND, Koosh-TND and Packages-TND, against each other. We varied the threshold values to get a sense of how much faster the algorithms are when a higher threshold causes more records to be discarded. Each algorithm was run at the given threshold value three times, and the resulting number of comparisons was averaged over the three runs to get our final results.

Figure 1 shows the results of this first experiment. The first three lines on the graph represent the performance of our three algorithms. On the horizontal axis, we vary the threshold value. The vertical axis (logarithmic) indicates the number of calls to the match function, which we use as a measure of the work performed by the algorithms. The first thing we notice is that work performed by the algorithms grows exponentially as the threshold is decreased. Thus, clearly thresholds are a very powerful tool: one can get high-

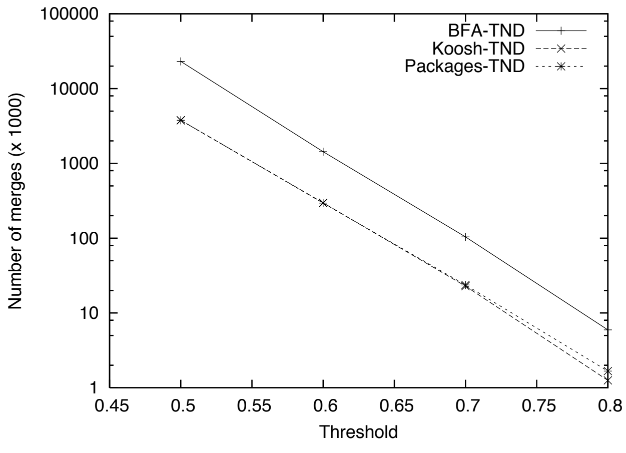


Figure 2: Thresholds vs. Merges

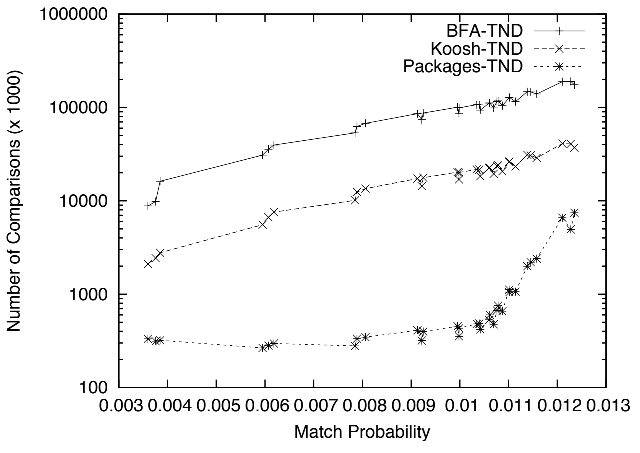


Figure 3: Selectivity vs. Comparisons

confidence results at a relatively modest cost, while computing the lower confidence records gets progressively more expensive! Also interestingly, the BFA-TND and Koosh-TND lines are parallel to each other. This means that they are consistently a constant factor apart. Roughly, BFA does 10 times the number of comparisons that Koosh does.

The Packages-TND algorithm is far more efficient than the other two algorithms. Of course, Packages can only be used if Properties P1, P2 and P3 hold, but when they do hold, the savings can be dramatic. We believe that these savings can be a strong incentive for the application expert to design match and merge function that satisfy the properties.

We also compared our algorithms based on the number of merges performed. In Figure 2, the vertical axis indicates the number of merges that are performed by the algorithms. We can see that Koosh-TND and the Packages-TND are still a great improvement over BFA. BFA performs extra merges because in each iteration of its main loop, it recompares all records and merges any matches found. The extra merges result in duplicate records which are eliminated when they are added to the result set. Packages performs slightly more merges than Koosh, since the second phase of the algorithm does not use any of the merges that occurred in the first phase. If we subtract the Phase 1 merges from Packages (not shown in the figure), Koosh and Packages perform roughly the same number of merges.

In our next experiment, we compare the performance of our algorithms as we vary the probability that base records match. We can control the match probability by changing parameters k or σ, but we use the resulting match probability as the horizontal axis to provide more intuition. In particular, to generate Figure 3, we vary parameter k from 5 to 55 in increments of 5 (keeping the threshold value constant at 0.6). During each run, we measure the match probability as the fraction of base record matches that are positive. (The results are similar when we compute the match probability over all matches.) For each run, we then plot the match probability versus the number of calls to the match function, for our three algorithms.

As expected, the work increases with greater match probability, since more records are produced. Furthermore, we note that the BFA and Koosh lines are roughly parallel, but the Packages line stays level until a quick rise in the amount of work performed once the match probability reaches about 0.011. The Packages optimization takes advantage of the fact that records can be separated into packages that do not merge with one another.

In practice, we would expect to operate in the range of Figure 3 where the match probability is low and Packages outperforms Koosh. In our scenario with high match probabilities, records that refer to different entities are being merged, which means the match function is not doing its job. One could also get high match probabilities if there were very few entities, so that packages do not partition the problem finely. But again, in practice one would expect records to cover a large number of entities.

9. RELATED WORK

Originally introduced by Newcombe et al. [17] under the name of record linkage, and formalized by Fellegi and Sunter [9], the ER problem was studied under a variety of names, such as Merge/Purge [12], deduplication [18], reference reconciliation [8], object identification [21], and others. Most of the work in this area (see [23, 11] for recent surveys) focuses on the "matching" problem, i.e., on deciding which records do represent the same entities and which ones do not. This is generally done in two phases: Computing measures of how similar atomic values are (e.g., using edit-distances [20], TF-IDF [6], or adaptive techniques such as q-grams [4]), then feeding these measures into a model (with parameters), which makes matching decisions for records. Proposed models include unsupervised clustering techniques [12, 5], Bayesian networks [22], decision trees, SVM's, conditional random fields [19]. The parameters of these models are learned either from a labeled training set (possibly with the help of a user, through active learning [18]), or using unsupervised techniques such as the EM algorithm [24].

All the techniques above manipulate and produce numerical values, when comparing atomic values (e.g. TF-IDF scores), as parameters of their internal model (e.g., thresholds, regression parameters, attribute weights), or as their output. But these numbers are often specific to the techniques at hand, and do not have a clear interpretation in terms of "confidence" in the records or the values. On the other hand, representations of uncertain data exist, which soundly model confidence in terms of probabilities (e.g., [1, 10]), or beliefs [14]. However these approaches focus on computing the results and confidences of exact queries, extended with simple “fuzzy” operators for value comparisons (e.g., see [7]), and are not capable of any advanced form of entity resolution. We propose a flexible solution for ER that accommodates any model for confidences, and proposes efficient algorithms based on their properties.

Our generic approach departs from existing techniques in that it interleaves merges with matches. The first phase of the Packages algorithm is similar to the set-union algorithm described in [16], but our use of a merge function allows the selection of a true representative record. The presence of “custom” merges is an important part of ER, and it makes confidences non-trivial to compute. The need for iterating matches and merges was identified by [3] and is also used in [8], but their record merges are simple aggregations (similar to our “information gathering” merge), and they do not consider the propagation of confidences through merges.

10. CONCLUSION

In this paper we look at ER with confidences as a “generic database” problem, where we are given black-boxes that compare and merge records, and we focus on efficient algorithms that reduce the number of calls to these boxes. The key to reducing work is to exploit generic properties (like the threshold property) than an application may have. If such properties hold we can use the optimizations we have studied (e.g., Koosh-T when the threshold property holds). Of the three optimizations, thresholds is the most flexible one, as it gives us a “knob” (the threshold) that one can control: For a high threshold, we only get high-confidence records, but we get them very efficiently. As we decrease the threshold, we start adding lower-confidence results to our answer, but the computational cost increases. The other two optimizations, domination and packages, can also reduce the cost of ER very substantially but do not provide such a control knob.

11. REFERENCES

[1] D. Barbará, H. Garcia-Molina, and D. Porter. The management of probabilistic data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[2] O. Benjelloun, H. Garcia-Molina, J. Jonas, Q. Su, and J. Widom. Swoosh: A generic approach to entity resolution. Technical report, Stanford University, 2005.

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

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

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

[6] William Cohen. Data integration using similarity joins and a word-based information representation language. *ACM Transactions on Information Systems*, 18:288–321, 2000.

[7] Nilesh N. Dalvi and Dan Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, pages 864–875, 2004.

[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):1183–1210, 1969.

[10] Norbert Fuhr and Thomas Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

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

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

[13] Saul Kripke. Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94, 1963.

[14] Suk Kyoon Lee. An extended relational database model for uncertain and imprecise information. In Li-Yan Yuan, editor, *VLDB*, pages 211–220. Morgan Kaufmann, 1992.

[15] David Menestrina, Omar Benjelloun, and Hector Garcia-Molina. Generic entity resolution with data confidences (extended version). Technical report, Stanford University, 2006.

[16] A. E. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DMKD*, pages 0–, 1997.

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

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

[19] Parag Singla and Pedro Domingos. Object identification with attribute-mediated dependences. In *Proc. of PKDD*, pages 297 – 308, 2005.

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

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

[22] Vassilios S. Verykios, George V. Moustakides, and Mohamed G. Elfeky. A bayesian decision model for cost optimal record matching. *The VLDB Journal*, 12(1):28–40, 2003.

[23] 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.

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

Circumventing Data Quality Problems Using Multiple Join Paths

Yannis Kotidis
Athens University of Economics and Business
kotidis@aueb.gr

Amélie Marian
Rutgers University
amelie@cs.rutgers.edu

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

ABSTRACT

We propose the Multiple Join Path (MJP) framework for obtaining high quality information by linking fields across multiple databases, when the underlying databases have poor quality data, which are characterized by violations of integrity constraints like keys and functional dependencies within and across databases. MJP associates quality scores with candidate answers by first scoring individual data paths between a pair of field values taking into account data quality with respect to specified integrity constraints, and then agglomerating scores across multiple data paths that serve as corroborating evidences for a candidate answer. We address the problem of finding the top-few (highest quality) answers in the MJP framework using novel techniques, and demonstrate the utility of our techniques using real data and our Virtual Integration Prototype testbed.

1. INTRODUCTION

In any large organization, there are many database-centric applications, with overlapping features and functionality, ranging from sales and ordering tools to inventory and provisioning applications. These applications have authority over different pieces of data, and the difficulty of integrating legacy applications into a unified application for a given task typically results in the data being spread across multiple, autonomously managed databases. For instance, a multitude of ordering and provisioning tools can lead to customer accounts and billing data being present in different databases depending on, among other things, location, type of customer, etc. This fragmentation of data makes investigations across these databases problematic. A standard technique used for the task of querying across databases is the join path, linking two data fields, possibly in different databases, through intermediate data. Given a value for one of the data fields, a join path enables the identification of values reachable in the other field using the join path.

Compounding the difficulty of querying across databases is the prevalence of data quality problems, within and across databases (see, e.g., [6]). A typical phenomenon is the existence of duplicate, default and null values in columns of database tables that are supposed to be treated as primary/foreign keys, due to the inability to enforce integrity constraints across independent databases. For instance, a provisioning database may have a place-holder (i.e., a field) for storing customer contact information. However, often this field is empty (null values) or populated with dummy (default) values, since this information is of no immediate use for the application that deals with inventory and provisioning and which oversees this data. Data inconsistencies (e.g., multiple records with the same key value) are widespread, and can often be traced back to human errors, e.g., during manual data entry. Default values and data inconsistencies are examples of poor data quality prevalent in large databases.

1.1 VIP: Motivating Example

VIP is an integration platform, developed at AT&T, covering more than 30 legacy systems. It was developed in an effort to provide a platform for doing quick investigations and resolving disputes (due to data inconsistencies) between different applications.

A basic query that often arises in VIP is of the form "given the value of a field X, find the value of a field Y". For instance, when processing telecom data, an example query is: given the telephone number (TN) of a customer that shows up in a sales application (SALES), find the circuit id of the attached line. Since circuit ids are not part of SALES application, the users need to access the inventory application INVENTORY that can look up circuit ids using a provisioning order number (PON). Users have access to a front-end web interface that provides authentication and allows querying the underlying inventory dataset by pasting a single PON value into a form. The same front-end can also retrieve circuit information when queried using a TN, but the internal mapping is incomplete and contains inconsistencies. Thus, we need to devise additional strategies for locating the target circuit id by considering other applications that we may have access to.

By examining patterns of user interactions with the SALES, ORDERING, PROV and INVENTORY applications, we have been able to compute the schema graph, depicted in Figure 1, to help answer the query; the meaning of the numbers along the edges will be made clear when discussing our experimental results. PROV is an application that maintains provisioning records, while ORDERING is an ordering tool used primarily for small-business customers. Table 1 describes the fields depicted in the schema graph of Figure 1. The combined size of the databases behind these four applications is in the order of 100 million records.

The schema graph provides multiple paths to link a TN value in SALES to a CircuitID in INVENTORY. We list here a few of them:

- Using a TN value in SALES, we obtain PON values, based on the "intra-application" edge (SALES.TN, SALES.PON) depicted in Figure 1. We then access the INVENTORY application using these PON values and the "inter-application" edge (SALES.PON, INVENTORY.PON). There, we look up CircuitIDs using the

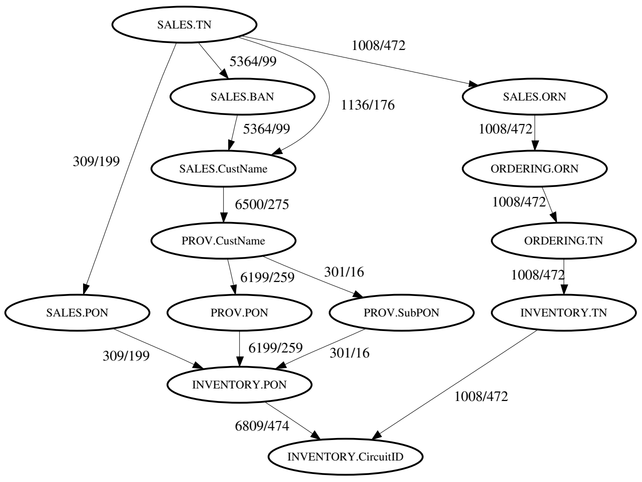


Figure 1: **Schema graph for subset of VIP**

Field Name	Description
TN	Telephone Number under investigation
BAN	Billing Account Number (primary key in biller)
CustName	Customer name (in biller and provisioning)
PON	Provisioning Order Number (key in provisioning applications)
SubPON	subsequent/related Provisioning Order Number (links multiple provisioning records for a customer)
ORN	Order Number (key in ordering applications)
CircuitID	Circuit the line is attached to

Table 1: **Description of Fields in Figure 1**

(INVENTORY.PON, INVENTORY.CircuitID) intra-application edge in INVENTORY. This corresponds to the left-most path in the schema graph.

- Given a TN in SALES, we can look up the customer name. This may be done directly, or via the billing account number for the customer. Notice that due to internal inconsistencies the two methods might give us different results. We can then input the customer name in the PROV application to retrieve all known PONs for the customer (from the PON and SubPON fields) which can be then used to probe INVENTORY, as in the first case. This corresponds to the set of middle paths in the schema graph of Figure 1.
- Small-business customers typically have multiple working telephone lines sharing the same circuits. For such customers, we can obtain the order number (ORN) in SALES, probe ORDERING and get all other lines ordered by the customer. Using this set of telephone numbers, we can probe INVENTORY multiple times. Even though, as explained, the internal TN-to-CircuitID mapping in INVENTORY is often incomplete, we can use the expanded set of all TNs in the customer order to try and find matching circuit ids in INVENTORY. This corresponds to the right-most path in the schema graph of Figure 1.

Given the different schema graph paths that link the TN input field (in SALES) to the CircuitID output field (in INVENTORY), which *join path* should be used to identify query answers?

1.2 Multiple Join Path Framework

When querying across multiple databases, in the presence of data quality problems, choosing any one join path results in missing answers, but choosing multiple join paths may lead to conflicting answers, especially when only a single answer is expected. Efficiency of query answering is also a concern. For instance, the return of a default value by an application may result in a significant number of probes to applications that follow it in a join path. Furthermore, different join paths that share edges need to be processed in a coordinated manner so that we avoid probing with the same input values multiple times.

The *Multiple Join Path* (MJP) framework proposed in this paper resolves these problems as follow:

- It takes *all* join paths in the schema graph into account.
- Each data path (schema path instance) is *scored*, taking the quality of integrity constraints (keys, functional dependencies), possibly across multiple databases/applications, and the quality of the data with respect to the integrity constraints into account.
- Multiple data paths between the same TN, CircuitID value pairs are treated as corroborating evidences, and data path scores are agglomerated to yield scores for CircuitID values.
- All join paths are considered when deciding the next application to probe. Intersecting data paths help re-use results of other join paths and reduce the number of probes to the applications.
- The top-few (typically 1) matches are returned as the desired answers. The schema graph and the computed data paths are used to prune unnecessary accesses to the applications.

When we are interested only in the top-few matches, it is extremely expensive to repeatedly probe the legacy applications, one schema graph edge at a time, to find all matching answers. This leads to the main technical problem addressed in this paper, the *Multiple Join Path Problem*:

> Given a schema graph identifying multiple join paths between field X and field Y, and a value $X = x$, find the top-few values of Y that are reachable from $X = x$ using the schema join paths.

The contributions of our paper are as follows:

- We introduce the MJP framework, and an agglomerative scoring methodology, to quantify answer quality in the presence of data quality problems arising due to integrity constraint violations in primary and foreign key columns, across multiple databases (Section 2).
- We develop novel techniques to limit the probing of legacy applications to efficiently compute the top-few answers to the MJP Problem. The agglomerative scoring methodology essentially renders previous mechanisms for computing top-k answers inapplicable for our problem (Section 3).
- Finally, we evaluate our techniques using real data and our VIP testbed. In particular, we demonstrate both the utility of the agglomerative scoring methodology in the presence of data quality problems, and the efficiency of our algorithmic techniques for computing top-few answers. In our real telecom example, we observe a reduction in the number of probes to the legacy applications by a factor of up to 18 in some cases (Section 4).

2. THE MJP PROBLEM

In this section, we introduce the Multiple Join Path framework, and our agglomerative scoring methodology, to quantify answer quality in the presence of data quality problems in multiple databases.

2.1 Queries and Answers

A basic query of interest is of the form "given the value of a field X, find values of a field Y", where X and Y refer to specific fields of individual applications. For instance, when processing telecom data, example queries include:

- Q1: given the telephone number of a customer (in SALES), find the circuit id (in INVENTORY) that the line is attached to.
- Q2: given a circuit id (in INVENTORY), find the customer names (in SALES) whose telephone numbers attach to this circuit id.

In the case of query Q1, one would expect there to be exactly one resulting answer. Since multiple telephone numbers may be attached to a circuit, query Q2 may have more than one answer. In both cases, X and Y are fields in different databases, so we need to establish *join paths* that link these two fields. There may be multiple possible join paths between any two given fields, and the schema graph, discussed next, identifies these possibilities.

2.2 Schema and Data Graphs

A schema graph is a 3-tuple (G, X, Y), where:

- $G = (V, E)$ is a *directed acyclic graph*, whose nodes $V = \{X, Y, \ldots\}$ are labeled by field names of accessible applications, and $E \subset V \times V$ are directed edges.
- $X \in V$ is the unique source (no incoming edges), and $Y \in V$ is the unique sink (no outgoing edges) of G.

A directed edge $(v1, v2) \in E$ is referred to as an *intra-application* edge, if $v1$ and $v2$ are fields in the same application; otherwise, it is an *inter-application edge*. A directed path P from X to Y in G is referred to as a *join path*. For instance, the schema graph of Figure 1 has six possible join paths from SALES.TN to INVENTORY.CircuitID, which can be used to answer query Q1. Thus, join paths in a schema graph identify different ways in which a basic query can be answered.

To ensure that join paths yield meaningful associations, not spurious correlations, we focus attention on the case where (i) all nodes in the schema graph (except, possibly, for source and sink nodes) are (possibly approximate) primary keys or foreign keys in their respective applications, (ii) inter-application edges correspond to (approximate primary key, approximate foreign key) associations, and (iii) intra-application edges are incident on an approximate primary key.

Given a specific value x of the source node X (e.g., telephone number, 555-5555, in query Q1), all join paths in the schema graph need to be explored to find all matching y values for the sink node Y (i.e., particular circuit ids). Intuitively, the data graph, defined below, captures these data associations. Given a schema graph (G, X, Y), a *data graph* is a triple (G_D, X_D, Y_D), where:

- $G_D = (V_D, E_D)$ is a *directed acyclic graph*, whose nodes V_D have labels of the form $T.A.v$, such that $T.A \in V$ and v is a value of field $T.A$, and $E_D \subset V_D \times V_D$ are directed edges such that $(T1.A1.v1, T2.A2.v2) \in E_D \Rightarrow (T1.A1, T2.A2) \in E$.
- $X_D \in V_D$ is the unique source of G_D, corresponding to value x of source node X of G, and $Y_D \subset V_D$ is a subset of the sink nodes of G_D, corresponding to values y_i of sink node Y of G.

For instance, given the schema graph of Figure 1, an example data graph is shown in Figure 2. There are two paths in this data graph from source node SALES.TN.555-5555 to the answer depicted by sink node INVENTORY.CircuitID.c1. Both these data paths correspond to the leftmost join path in the schema graph of Figure 1. Two additional nodes are present in this data graph, SALES.PON.pon3 and SALES.ORN.o1 (corresponding to schema graph nodes SALES.PON and SALES.ORN), which do not join with values in INVENTORY.PON and ORDERING.ORN, respectively. Note that the data graph can have multiple or no nodes corresponding to any specific node in the schema graph.

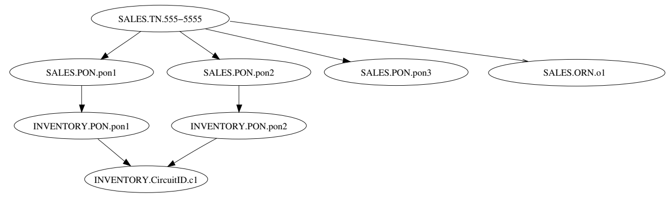


Figure 2: Data graph for query SALES.TN=555-5555

2.3 Scoring Answers

In a perfect world, the applications would have no internal data quality problems, and our basic query (given $X = x$, find Y) could be answered correctly by following all the join paths across the multiple applications starting from $X = x$, and taking the union of all the Y values that are reached along these individual join paths. But data quality problems are prevalent in large data-centric applications. For example, a primary key field (like the billing account number (BAN) field) may only be an *approximate* key [6]. Similarly, a functional dependency expected of an intra-application edge in the schema graph may be violated. As an example, we might find that the same telephone number is associated with two customer names due to manual data-entry errors in the SALES application.

So we are faced with the considerable challenge of answering our basic queries *without a priori knowledge of which values in the underlying databases are clean, and which ones are not*. To meet this challenge, we employ a probabilistic technique that scores data edges using values in the range $[0 \ldots 1]$. Thus, the score of a data edge $(T1.A1.v1, T2.A2.v2)$ represents our belief that the association between values $v1$ and $v2$ of fields $T1.A1$ and $T2.A2$ is correct. We will describe later how these scores are obtained. What is important is that this probabilistic interpretation of the scores allows us to combine scores across a data path.

Recall that a data path is just a sequential composition of data edges. Using a probabilistic interpretation of the data edge scores, assuming independence of the data edges in a data path, the *score of a data path* is defined to be the product of the scores of the constituent data edges. More formally, if $sc_1, sc_2, \ldots, sc_n$ are the scores of the constituent data edges of a data path P, then the score of P is given by:

$$sequential\_com(sc_i, 1 \leq i \leq n) \quad = \quad \Pi_{i=1}^{n}(sc_i) \qquad (1)$$

As will be explained, this probabilistic interpretation assigns scores on data paths using data quality metrics on the edges. Thus, a high quality data path will get high scores independent of the length of the path, unlike, e.g., techniques like [1]. In fact, it is easy to see that the latter technique is just a special case of our framework when all data edges are scored with the same value in $(0, 1)$.

An answer may be corroborated by multiple data paths, and our scoring methodology agglomerates the scores of these data paths, using *parallel composition*, to compute the score of a Y value. For example in Figure 2 there are two data paths from SALES.TN.555-5555 to answer INVENTORY.CircuitID.c1. Different data paths are considered independent evidences and their scores are combined in a probabilistic manner. Formally, if $sc_1, sc_2, \ldots, sc_n$ are the scores of individual data paths $P_i, 1 \leq i \leq n$, between two nodes in the data graph, then, to ensure that all scores are in $[0, 1]$, the score of the parallel composition of the P_i's is given by:

$$parallel\_com(sc_i, 1 \leq i \leq n) \quad = \quad s1 + s2 - (s1 * s2) \quad (2)$$

where $s1 = sc_1$ and $s2 = parallel\_com(sc_i, 2 \leq i \leq n)$.

Finally, the score of a Y value y_i is the score of the parallel composition of all the data paths from the source $X.x$ to the sink $Y.y_i$. This *agglomerative* scoring takes into consideration *all* the data paths that corroborate an answer.

Other combining functions may also be used without affecting the generality of the proposed methodology. The process that we describe in Section 3 requires the following two monotonicity properties, which allow for a broad selection of scoring functions:

- *Property 1:* the score of a data path is a non-increasing function of the scores of the constituent data edges.

- *Property 2:* the score of an answer is a non-decreasing function of the scores of the constituent data paths.

2.4 Data Edge Scores

Without a priori knowledge of the internals of the applications, or expertise on the quality of specific data items, our approach is to rely on expected functional dependencies between the exported data fields. For instance, in the telecom example, we expect a telephone number to uniquely identify a customer. Thus, when probing the SALES application, if we get two customer names for an assigned TN, this is a violation of an expected functional dependency and we should assign a lower score to the instantiated data edges.

Recall that intra-application schema edges $(T.A, T.B)$ capture associations where at least one of $T.A$ and $T.B$ is an approximate key in the corresponding application T. Assume, without loss of generality, that $T.A$ is the approximate key. Then the edge captures a *forward functional dependency* (FFD) from $T.A$ to $T.B$. Assume also that while answering a posed query, due to internal data quality problems, the following data edges are instantiated: $(T.A.v1, T.B.v11)$, $(T.A.v1, T.B.v12)$ and $(T.A.v2, T.B.v21)$. It is then obvious that the two different values $T.B.v11$, $T.B.v12$ associated with $T.A.v1$ are witnesses that $T.A.v1$ is in violation of the FFD and therefore data edge $(T.A.v2, T.B.v21)$ should have a higher score than edges $(T.A.v1, T.B.v11)$ and $(T.A.v1, T.B.v12)$.

Let $\{(T.A.v1, T.B.v1i), i = 1, \ldots\}$ be the set of data edges instantiated for value $T.A.v1$ following this schema edge, and let $|.|$ denote the size of a set. To achieve the desired behavior, the score of each data edge $(T.A.v1, T.B.v1i)$ is set to:

$$sc(T.A.v1, T.B.v1i) = \frac{1}{|\{(T.A.v1, T.B.v1i), i = 1, \ldots\}|} \quad (3)$$

The case when the schema edge captures a *backward functional dependency* (BFD) is handled symmetrically:

$$sc(T.A.v1i, T.B.v1) = \frac{1}{|\{(T.A.v1i, T.B.v1), i = 1, \ldots\}|} \quad (4)$$

Finally, when both $T.A$ and $T.B$ are approximate keys, the edge captures a *symmetric functional dependency* (SFD) and the score is computed as:

$$sc(T.A.vi, T.B.vj) = \frac{1}{|\{(T.A.vi, T.B.*)\} \cup \{(T.A.*, T.B.vj)\}|} \quad (5)$$

where '*' means any value and is used to capture all data edges emanating from $T.A.vi$ (resp. leading to $T.B.vj$).

For an inter-application schema edge $(T1.A, T2.A)$, the score of a data edge corresponding to this schema edge is always 1, since the association between the fields is assured by the schema graph. An interesting extension is to consider *approximate matching* between values of field A in applications $T1$ and $T2$. In that case the score of the inter-application data edge is adjusted by using some notion of error metric (e.g., normalized edit distance or tf.idf for strings) between the values.

2.5 Multiple Join Path Problem

Our goal is to locate high quality information across multiple databases, in the presence of data quality problems. Since the different Y values that are reached from a given X value may have very different scores, we are interested only in the top-few matches. When we are interested only in the top-few matches, it is extremely expensive to repeatedly probe the legacy applications, one schema graph edge at a time, to find all matching answers, only to eventually discard the low scoring answers.

This leads to the main technical problem addressed in this paper, referred to as the *Multiple Join Path* (MJP) Problem:

> Given a schema graph identifying multiple join paths between field X and field Y, and a value $X = x$, find the top-few values of Y (with the highest scores) reachable from x using the multiple join paths.

Conventional top-k evaluation requires exact scores to be returned along with the matching answers, resulting in a ranking of the k results. In our agglomerative scoring methodology, since any unexplored data path could eventually corroborate a known Y value, resulting in a score increase (however slight), one would not be able to perform any early pruning for the MJP Problem, if one insisted on returning exact scores.

A more promising approach is where one can return top-k answers, where each answer is associated with a score range, and the result is a *set* of answers, not a ranking. In Section 3, we shall discuss novel solutions to the MJP Problem, and subsequently experimentally validate the utility and efficiency of our approach using real data and the VIP testbed.

3. THE MJP PROBLEM: SOLUTION

3.1 Incremental Data Graph Computation

Given a specific value x of the source node X in the schema graph, the data graph is initially instantiated with a unique (source) node $X_D = X.x$. For each newly inserted data node T_D in the data graph (excluding those in set Y_D of sink nodes), we create the set *open_edges(T_D)* to be the set of all schema edges $e \in E$ that emanate from the corresponding node T in the schema graph. As an example, for the schema graph shown in Figure 1 and for TN = 555-5555 being the TN in query $Q1$, the data graph is instantiated with a single node SALES.TN.555-5555. The set *open_edges(SALES.TN.555-5555)* will then include the following schema edges: (SALES.TN, SALES.PON), (SALES.TN, SALES.BAN),
(SALES.TN, SALES.CustName) and (SALES.TN, SALES.ORN).

An *open node* in the data graph is any node T_D, not in Y_D, for which the set *open_edges(T_D)* is not empty. Our algorithms will proceed by carefully choosing an open node T_D and selecting one of the edges e in set *open_edges(T_D)* to explore. Following an intra-application edge $(T.A, T.B)$ for open node $T.A.u$ results in probing application T and retrieving a set of values for field $T.B$. For each unique value v_i of attribute $T.B$ in the result of this probe, we add a new node $T.B.v_i$ to the data graph and generate set *open_edges($T.B.v_i$)*. We further instantiate the data edge $(T.A.u, T.B.v_i)$ and compute its score. In Figure 3, we depict the data graph after exploring schema edge (SALES.TN, SALES.PON) for open node SALES.TN.555-5555. The application in this case returned three distinct values for SALES.PON: pon1, pon2 and pon3.

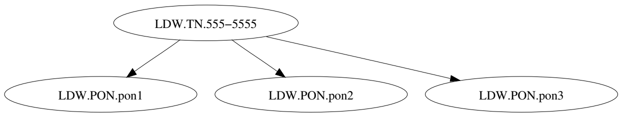


Figure 3: Data graph, after processing of edge (SALES.TN, SALES.PON)

Following an inter-application edge $(T1.A, T2.A)$ does not incur additional probes to the applications. Values of field $T1.A$ that do not appear in application $T2$ will not generate any new data nodes when a follow-up intra-application edge is processed. In either case, after edge e is explored it is removed from *open_edges(T$_D$)*.

We adopt a simple cost model that enumerates the number of probes to the applications while expanding the data graph to answer the user query. This cost model is reasonable in the absence of internal knowledge of the behavior of the applications.

3.2 Scheduling of Open Nodes

While building the data graph, we often have many open nodes to explore, each with at least one unexplored edge in *open_edges()*. We thus need a strategy that will lead to early pruning when computing top-k answers.

Since the data graph (G_D, X_D, Y_D) has a strong correspondence with the schema graph, we can pick the next open node/schema edge to explore using standard graph searching techniques like depth-first-search (DFS) or breadth-first-search (BFS) guided by the schema graph. Such techniques however are oblivious to the statistics we can collect both at the schema graph as well as at the (incomplete) data graph while processing the query. As is demonstrated by our experiments in Section 4, this results in substantially more probes to the applications. In what follows, we describe a greedy scheduling technique that is based on the notion of the *maximum benefit* of unexplored paths that go through open nodes.

Benefit computation involves two components. The first uses the statistics accumulated in the data graph to compute the score of all paths leading to an open node. The second component calculates the best way that the data graph can be augmented when following unexplored edges from an open node on the way to an answer. The fusion of these two components provides our benefit metric.

At each step, our algorithm maintains this benefit metric per open node/schema-edge in the data graph and schedules the next move using this metric. At an abstract level, our methodology for processing a user query can be summarized as follows:

- Start from the sole instance of source node X_D and expand one data node at a time. For any open node T_D, maintain the multiset of scores along data paths from X_D to T_D.
- By associating open node T_D with its schema node T, we can quantify the *residual benefit* of an unexplored schema edge e in *open_edges(T$_D$)* as the maximum possible contribution of the subgraph from X_D to any possible data node in set Y_D, passing through T_D using instances of e in the data graph.

As an example, we consider the data graph of Figure 3. For open node SALES.TN.555-5555 there are three unexplored edges in the set *open_edges(SALES.TN.555-5555)*: (SALES.TN, SALES.BAN), (SALES.TN, SALES.CustName), and (SALES.TN, SALES.ORN). Figure 4 shows the maximal subgraph that can be generated by exploring these edges in a way that maximizes the score of an answer. In this figure there are five paths from SALES.TN.555-5555 to schema node INVENTORY.CircuitID. For each path P_i, a schema edge is only instantiated once (since all edges are treated as FFD/SFD). However, a schema edge e' may generate one distinct

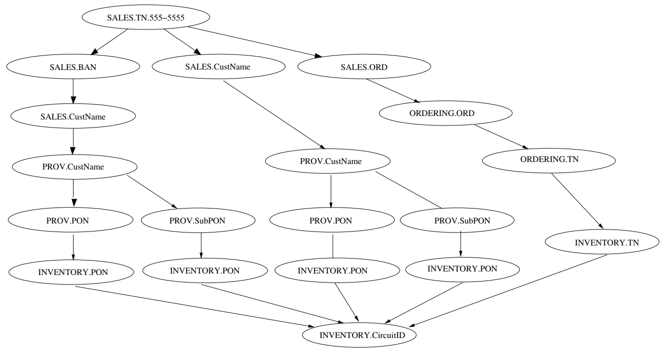


Figure 4: Maximum paths for unexplored edges of node SALES.TN.555-5555, after processing of edge (SALES.TN, SALES.PON)

data edge for each path P_i that contains e'. As an example, schema edge (PROV.CustName, PROV.PON) instantiates two distinct data edges in Figure 4.

Given one or more open nodes T_D in the data graph, we pick the next edge to explore as the one that maximizes our benefit metric. This is our *maximum benefit* policy, MAXB. In our experiments we see that MAXB outperform DFS and BFS, by a factor of up to 18:1.

3.3 Pruning Criteria

Unlike conventional top-k evaluation, where exact scores of answers are returned, for our MJP framework a more promising approach is to return the top-few answers, where each answer is associated with a score range. We distinguish between two versions of the problem:

- The exact top-k set $Y_D = (y_1, \ldots, y_k)$ is returned. For each answer y_i, we provide a score range $[s_{min}(y_i) \ldots s_{max}(y_i)]$.
- The top cluster of answers that is guaranteed to contain the top-k values is returned. Each answer is associated with a score range. We call this the *top-few* evaluation. Top-few is valuable when doing quick ad hoc investigations, since it allows for more pruning because of the weaker stopping condition.

Let $y \in Y_D$ be an answer present in the (incomplete) data path. Let $scores(y) = (sc_1, sc_2, \ldots, sc_n)$ be the scores of all data paths from X_D to y. Then, the minimum score of answer $Y = y$ is the parallel composition of the scores of all known paths to y:

$$s_{min}(y) = parallel\_com(sc_i, 1 \leq i \leq n) \qquad (6)$$

The maximum score of answer y is computed by additionally considering the maximum benefit of each open node and unexplored edge in the data graph, as discussed previously.

Through similar arguments we can compute the range of scores $[s_{min}(y_{unseen}) \ldots s_{max}(y_{unseen})]$ of an answer y_{unseen} that we have not encountered in our evaluation as $[0 \ldots max\_contribution]$. The lower bound is trivial (when no new answer exists). The upper bound follows easily if we consider that all paths from the open nodes in the data graph terminate to a new answer y_{unseen}.

In a naive evaluation of the MJP Problem we stop when all open nodes in the data graph have been explored. However, one may stop earlier without exploring all open nodes, depending on the version of the problem. Assume set $\mathcal{Y}$ contains all answers that we have seen so far and also y_{unseen} (a placeholder for some answer we have not yet encountered). Thus, $\mathcal{Y}$=$Y_D \bigcup \{y_{unseen}\}$. We order the answers in $\mathcal{Y}$ using their minimum scores as $y_1, y_2, \ldots$, where $s_{min}(y_i) \geq s_{min}(y_j)$ when $i < j$. This order also implies

$s_{max}(y_i) \geq s_{max}(y_j)$. If set $\mathcal{Y}$ contains more than k answers, we may stop further processing under the following condition:

- In top-k evaluation, we stop when $s_{max}(y_{k+1}) \leq s_{min}(y_k)$. That is, the upper bound on the score of the k+1'th candidate y value is no larger than the score of the current k'th candidate.
- In the top-few evaluation, we may stop if $s_{max}(y_{unseen}) \leq s_{min}(y_k)$. If this condition holds then any new answer cannot possible be scored higher that our current k'th candidate y_k. Thus, the top cluster is identified and we return those y_i's with $s_{max}(y_i) \geq s_{min}(y_k)$.

4. EXPERIMENTS

In this section, we experimentally evaluate our solution using our VIP testbed. Due to lack of space we provide detailed results for one query in our real dataset (query Q1, Section 2.1). Results for other queries between pairs of nodes in the schema graph of Figure 1 were similar. Our main experimental results can be summarized as follows:

- Real datasets have a multitude of data quality problems and no join path is immune to these problems. Using a fixed path or the maximum path for answering a query can lead to missing answers (low recall). That is why, in our MJP framework, all paths are considered.
- The data graphs can be fairly large (for instance when default values are encountered). Our scheduling techniques based on the maximum benefit metric achieve substantial pruning by eliminating a large number of candidate paths from evaluation.

The rest of this section is organized as follows. In Section 4.1 we illustrate that real applications are faced with significant data quality problems. When joining data across diverse applications, we typically find many answers, even when a single answer is expected (for instance a single CircuitID for a TN in Q1). Thus, ranking is required to help users identify the correct answer. In Section 4.2 we demonstrate that top-1 answers typically have several instantiated data paths leading to them and an agglomeration of their scores is needed. An important observation is that even join paths with small schema weights in their edges are useful in determining top-1 answers. In Section 4.3 we demonstrate that using our benefit metric results in substantially fewer probes to the applications, often by a factor of 1:18. Using the top-few execution model, this reduction is further increased by a factor of 2.

4.1 Nature and Quality of Data

We used traces of real user queries and obtained a random sample of 150 TNs that users ran investigations upon. We then used the schema graph to obtain circuit ids for these TNs (i.e., using k=∞). We noticed that there is a large number of TNs (56) that return no matching circuit ids. This is because (i) the INVENTORY dataset is incomplete and (ii) the provisioning key is often missing in SALES, forcing join paths either through customer names (CustName) or order numbers (ORN). The distribution is heavy-tailed, as there are many TNs for which we obtain 50 or more circuits through the schema graph. The maximum number of circuits returned for a single TN was 257. It is clear that most queries return a lot of answers. In fact only 2 TNs returned just one circuit! Thus, we need to be able to prune the long lists of matching circuit ids in order to provide meaningful answers to the user.

Using answer scores, we classify user queries into the following classes (in parentheses we show the number of TNs in each class).

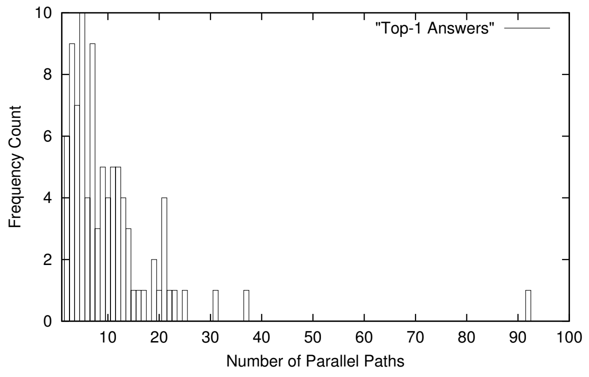


Figure 5: **Number of parallel paths in top-1 answers**

- **hH: heavyHitters(10):** These are the top-10 queries (TNs) ranked by the number of matching circuits in our data. TNs in that group returned between 128 and 257 circuits.
- **oL: oneLarge(47):** This is the subset of TNs that returned one circuit id with score at least 1% and zero or more circuits with scores less that this threshold.[1]
- **mL: manyLarge(4):** This set of TNs have at least 5 matching circuits with score at 1% or higher.
- **mS: manySmall(8):** This set of TNs returned at least 5 answers, while no answer had score greater or equal to 1%.
- **aA: anyAnswer(94):** All TNs with any matching circuits.
- **nA: noAnswer(56):** These are TNs for which no answer (circuits) can be obtained from the data.

4.2 Benefit of Agglomerative Scoring

We now address the utility of our agglomerative scoring methodology. In Figure 5 we plot the number of parallel data paths that contribute to the top-1 answer for each TN with a non-empty answer (set `anyAnswer`). For the 94 top-1 answers, there is an average of 10 parallel paths per answer (for a total of 946), out of which roughly 2.5 parallel paths per answer (for a total of 229) are significant (score of the path is greater than 10% of final score). In contrast, when looking at all answers for each TN there are on the average just 1.7 parallel paths contributing to each answer.

A natural question one may ask is whether all the schema join paths are really relevant, or if one of them dominates in its contribution to the final scores. In Figure 1, we annotate the schema graph edges with two numbers. The first is the number of data paths leading to an answer that instantiated this edge. The second number is for top-1 answers only. Some interesting observations on the nature of the data can be drawn by interpreting these numbers. First, paths that go through the SALES.PON node are more likely to end up in a top-1 answer: 199 out of 309 overall. Similarly, probing the ORDERING application leads to a top-1 answer in almost half the cases. In contrast, many paths that use instances of nodes SALES.BAN, SALES.CustName do not end up in top-1 answers. However, it is still beneficial to include these nodes in the schema graph. We notice that 275/946 top-1 paths (paths that result in a top-1 answer) go through instances of these nodes. If we remove these nodes from the schema graph along with all paths that

[1]The low value of the threshold has been chosen to capture as many potentially relevant answers as possible, given the scoring methodology.

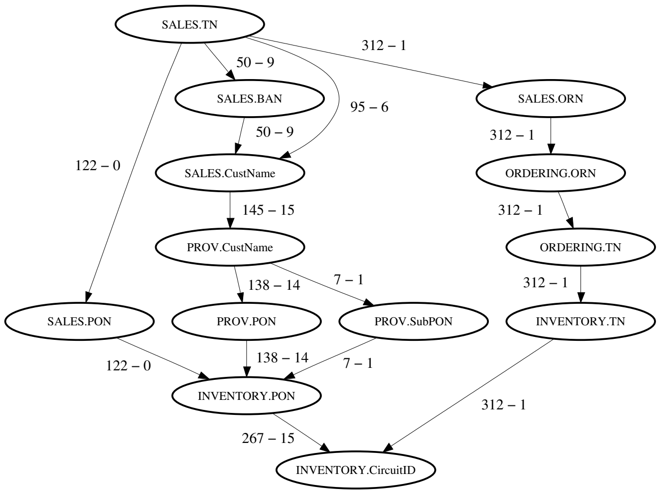


Figure 6: **Aggregate Path Statistics for top-1 answers (sets oneLarge - manySmall)**

		Top-1			Top-few		
	$k=\infty$	DFS	BFS	MAXB	DFS	BFS	MAXB
aA	246.9	245.4	125.1	47.1	244.9	97.3	24.6
hH	724.1	722.9	453.0	109.5	722.3	359.6	56.1
oL	261.0	259.1	104.6	14.6	251.4	95.5	14.1
mL	365.8	365.4	249.8	40.0	326.8	136.8	19.3
mS	258.5	258.5	253.8	184.8	258.5	231.7	119.6
nA	24.6						

Table 2: **Cost of top-1/top-few evaluation**

use them, then for the 94 queries with non empty top-1 answers, 8 return no result while for 77 the top-1 answer differs.

We next discuss the issue if there is any correlation between the join paths used and the class of queries. In Figure 6 we aggregate at the schema level the number of paths in top-1 answers for sets `oneLarge` (first number next to an edge) and `manySmall` (second number). The number of paths in set `oneLarge` is larger because more TNs belong to that set (47 compared to 8 in set `manySmall`). In the case of set `oneLarge`, paths through ORDERING schema nodes appear in more than half of the cases. However, again paths through the PROV dataset using customer names are significant in top-1 answers. In set `manySmall` there is only one top-1 answer with a path through ORDERING. Most of the paths (9 out of 15) go through the (SALES.TN,SALES.BAN) edge and subsequently to PROV, joining on the CustName attribute.

4.3 Efficiency of Top-k Evaluation

We use the number of probes to the applications to determine the efficiency of top-k evaluation. In Table 2, we present the average number of probes per TN during the top-k evaluation, for k=1. We also present the average number of probes when all circuits were requested (k=∞). For scheduling the next open node to explore, we tested the MAXB policy (discussed in Section 3) as well as the standard DFS and BFS orders obtained from the schema graph.

Top-1 evaluation, using MAXB, reduces the number of probes by a factor of more than 5, on the average, for TNs with matching circuit ids. A large number of queries (56/150) returned no answer, and for these queries top-1 evaluation cannot prune any paths. For the remaining TNs the greatest benefits arise for subset `oneLarge`. These are TNs for which one circuit stands out from

		Top-1		
	$k=\infty$	DFS	BFS	MAXB
aA	207/1189	206/1188	130/1109	59/679
hH	903/1189	902/1188	618/1109	162/679
oL	305/1189	304/1188	141/890	23/231
mL	415/793	414/793	327/625	52/83
mS	310/1128	310/1128	306/1106	237/629
nA	47/232			

Table 3: **Avg and Max Data Graph Size (edge set E_D)**

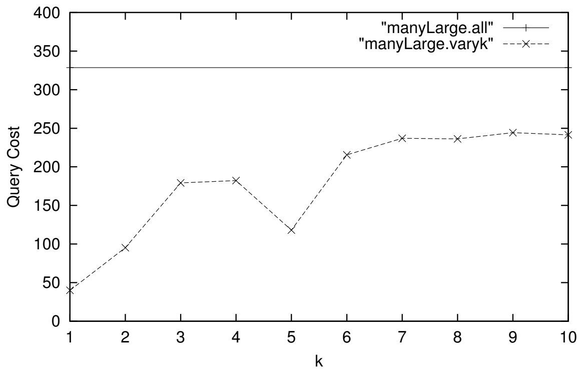


Figure 7: **Query cost, varying k (manyLarge set)**

the answer set and, thus, we can expect a lot of pruning during top-1 evaluation. Here, the MAXB policy reduces the cost of processing by a factor of 18:1 compared to the case where k=∞. For the set `manyLarge`, savings are smaller but still substantial (9:1). Even in the case of set `manySmall`, MAXB results in significant pruning. These are TNs for which a large number of circuits with very low scores were discovered. Looking at the instance graphs of these TNs we observed that most were due to a default value of field BAN in SALES, resulting in many matching customer names when following the intra-application (SALES.BAN, SALES.CustName) edge. Because of our scoring mechanism, all these paths were assigned very low scores and MAXB was able to prune a substantial number of them. In contrast, DFS for top-1 is almost as bad as getting all answers. This is because DFS follows deep paths through the schema graph to the end without concern of the current scores leading to an open node or potential benefits of open paths. BFS is slightly better since all paths are explored in unison.

In Table 3, we show the average (first number) and maximum (second number) size of the data graph for the same experiment. We notice that for k=∞ the data graph size is on the average 207 with a maximum instance of 1189 (edges). Thus, evaluation of Multiple Join Path queries in our framework has very modest requirements in terms of memory usage. We further notice that top-1 evaluation with the MAXB policy reduces these numbers by a factor of up to 13:1. This reduction of the data graph size will become significant in a multi-user environment when the server processes several queries at a time.

In Figure 7, we plot the query cost, varying k between 1 and 10. For comparison, we also show the cost when k=∞ (flat line). It is interesting that there is a drop in query cost for k=5. This suggests that the size of the top-cluster is, on the average, around five (circuits per TN) with the last 3 having similar scores. Thus, for k=3 or 4, additional queries are required to distinguish among them, while when k=5, we can stop earlier and report all of them.

In Table 2 we show the cost of the top-few execution, for k=1. The top-few execution, allows us to stop a query at an earlier stage, when a superset containing the top-k cluster has been identified. As in the top-1 case, the MAXB policy by far outperforms the other alternatives. Comparing the numbers with the top-1 case, we see that we get a reduction in evaluation cost by a factor of two on the average. In most cases the number of answers returned to the user is very small, typically one. There are only 5 instances where we see more than 10 and all of them are for TNs with many small, indistinguishable, answers.

5. RELATED WORK

Scheuermann et al. [14] consider querying multiple database paths by allowing for some uncertainty in the attribute correspondences between databases in a multidatabase system. They return multiple query results ranked by some degree of confidence in the answer. However, to the best of our knowledge, our work is the first to take into account similar results from multiple paths as corroborating evidence and using this information to rank query results.

There has been much work in addressing the problem of identifying keyword query results in an RDBMS and ranking them based on some quality metric [8, 1, 2, 10, 9]. In such scenarios, the user queries multiple relations for a set of keywords and gets back tuples that contain all keywords, ranked by a measure of the proximity of the keywords. DBXplorer [1] and DISCOVER [10] use index structures coupled with the DBMS schema graph to identify answer tuples and rank answers based on the *number of joins* between the keywords. Our framework can also benefit from auxiliary structures like indexes and materialized views to speed up processing. BANKS [2] creates a data graph (a similar graph is used by [8]), containing all database tuples, allowing for a finer ranking mechanism that takes prestige (i.e., in-link structure) as well as proximity into account. Hristidis et al. [9] use an IR-style technique to assign relevance scores to keyword matches and take advantage of these relevance rankings to process answers in a top-k framework that allows for efficient computations through pruning. As with proximity search techniques, we consider all possible join paths and, as in [9], we want to allow for pruning of irrelevant data paths in order to speed up query execution time. A key difference with previous proximity search techniques is that none of these techniques deal with data quality issues, or agglomerate scores of multiple data paths that contribute to the same answer.

Top-k query evaluation algorithms that aim at identifying the k highest ranking answers to a query have been proposed for a variety of scenarios: multimedia [7, 11], web [12], expensive predicates [4], and RDBMS [3, 11]. Adaptive top-k strategies [4, 12] dynamically choose which operation to perform next based on current tuple scores and estimated statistics. In this paper, we use such adaptive techniques to select which join paths to investigate next. Most existing top-k techniques focus on cases where answer tuples can be mapped into a single relation, with all attributes values accessible through a unique ID, and rank the result tuples according to a predefined aggregation function (e.g., *min* or *weighted-sum*). While some of the proposed techniques [13, 11] apply to scenarios involving joins, and therefore deal with a potential explosion in the number of tuples, we are not aware of any top-k technique that does not consider each possible answer tuple as a single entity.

Traditional top-k techniques require exact top-k answer scores to be returned. In contrast, NRA [7], which only considers sorted accesses to multimedia sources, allows for the top-k answers to be returned as soon as they are identified, along with their possible range of scores. We use this relaxed stopping condition for our top-k evaluation, and present another efficient stopping condition: *top-few*, which returns a set of answers that are guaranteed to contain the best k answers.

Recently, Chaudhuri et al. [5] investigated the problem of ranking answers of database queries that are not very selective (*Many-Answers* problem) and propose a ranking function based on Probabilistic Information Retrieval ranking models. Our scoring functions also have a probabilistic interpretation and, similar to [5], ranking is proposed in order to prune a potentially large answer set. However, while in [5] the problem arises from loosely constrained queries, the complexity of our problems stems from (i) the existence of multiple join (schema) paths that can potentially link two attributes in the same or different databases, and (ii) low data quality that further increases the number of instantiated data paths for a given query. Approximating top-k answers, by offering guaranteed answer quality wrt the correct top-k scores [7, 4], or probabilistic guarantees [15], is an issue we do not address here.

6. CONCLUSIONS

This paper addressed the Multiple Join Path problem, of finding high quality query results that can be reached from a query node, by following one or more join paths in the schema graph, across multiple databases, in the presence of data quality problems. The framework proposed in this paper scores each data path that instantiates the schema join paths, taking data quality with respect to specified integrity constraints into account. Multiple data paths between the same nodes are treated as corroborating evidences, and data path scores are agglomerated to yield scores for matching answers. We develop novel techniques to efficiently compute the top-few answers within the Multiple Join Path framework, taking the agglomerative scoring mechanism into consideration. We evaluate our techniques using real data and our Virtual Integration Prototype testbed, and demonstrate both the utility of the agglomerative scoring methodology, and the efficiency of our algorithmic techniques for computing top-few answers.

7. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A system for keyword-based search over relational databases. *ICDE*, 2002.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. *ICDE*, 2002.

[3] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM TODS*, 27(2), 2002.

[4] K. C.-C. Chang and S.-W. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. *SIGMOD*, 2002.

[5] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. *VLDB*, 2004.

[6] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. John Wiley, 2003.

[7] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *PODS*, 2001.

[8] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina. Proximity search in databases. *VLDB*, 1998.

[9] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient IR-style keyword search over relational databases. *VLDB*, 2003.

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. *VLDB*, 2002.

[11] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB*, 2003.

[12] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM TODS*, 29(2), 2004.

[13] A. Natsev, Y. Chang, J. R. Smith, C. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. *VLDB*, 2001.

[14] P. Scheuermann, W.-S. Li, and C. Clifton. Multidatabase query processing with uncertainty in global keys and attribute values. *JASIS*, 49(3), 1998.

[15] M. Theobald, G. Weikum, R. Schenkel. Top-k query evaluation with probabilistic guarantees. *VLDB*, 2004.

In-network Outlier Cleaning for Data Collection in Sensor Networks

Yongzhen Zhuang and Lei Chen
Hong Kong University of Science and Technology
{cszyz, leichen}@cse.ust.hk

Abstract

Outliers are very common in the environmental data monitored by a sensor network consisting of many inexpensive, low fidelity, and frequently failed sensors. The limited battery power and costly data transmission have introduced a new challenge for outlier cleaning in sensor networks: it must be done in-network to avoid spending energy on transmitting outliers. In this paper, we propose an in-network outlier cleaning approach, including wavelet based outlier correction and neighboring DTW(Dynamic Time Warping) distance-based outlier removal. The cleaning process is accomplished during multi-hop data forwarding process, and makes use of the neighboring relation in the hop-count based routing algorithm. Our approach guarantees that most of the outliers can be either corrected, or removed from further transmission within 2 hops. We have simulated a spatial-temporal correlated environmental area, and evaluated the outlier cleaning approach in it. The results show that our approach can effectively clean the sensing data and reduce outlier traffic.

1 Introduction

A sensor network is equipped with thousands of inexpensive, low fidelity motes, which can easily generate sensing errors. The abnormal unreal sensor readings generated in a temporally or permanently failed sensor is called *outliers*. In many cases, outliers introduce errors in sensing queries and sensing data analysis. For example, a Sum query is less accurate if a large value outlier is counted. In addition, transmitting outliers to the sink is useless, adds additional traffic burden to the network, and consumes precious sensor energy without any benefit. Outlier cleaning tries to capture the outliers, correct or remove them from the data stream. Outlier cleaning in sensor networks is challenging because data are distributed among a large amount of sensors. It is for sure that outlier detection can be conducted centrally after all the data are collected to the sink. However, it is not energy efficient to transmit outliers, especially when the network size is large. For example, if an outlier is routed through a 15-hop path to the sink, the energy used to transmit this 15-hop datum is wasted. Therefore, in-network outlier cleaning tries to detect outliers during the data collection process as early as possible along the routing path of the data. It either corrects the outlier or removes it from further forwarding. Eventually, an outlier-free data stream is provided to the sensor network applications.

In this paper, we propose an in-network outlier cleaning approach for data collection over sensor networks. We can correct *short simple outliers* in 0 hop and remove *long segmental outliers* within 2 hops. We adopt wavelet approximation to correct short, occasionally appeared outliers. Since these short outliers are of high frequency, they can be corrected if we use the first few wavelet coefficients to represent the sensing series. An extraordinary advantage of using wavelet representation is that it can greatly reduce the dimension of the sensing data, as a consequence, reduces the energy cost of transmitting these data. If an outlier is a long segmental outlier, we can detect it by comparing its similarity with the neighboring nodes, given the nature that environmental data are spatially correlated [1]. Similarity is measured by Dynamic Time Warping (DTW) distance, which can capture the shape similarity in the elastic shifting sensing series [2]. The sensing series are routed as before to the sink, using a hop-count based routing algorithm [3]. The detection is conducted within 2 forwarding hops. A sensing series is not forwarded, if it is dissimilar with its network neighbors. Outlier cleaning requires in-network data processing on the individual sensor mote. In sensor networks, it is admitted that data processing is more economical than data transmission [4]. The outlier cleaning process adds O(KN) running time on each sensor. In the erroneous sensor network, this energy cost is trivial compared to that of the reduced traffic.

The remainder of this paper is organized as follows: Section 2 provides the background of outlier cleaning in sensor network data collection. In Section 3, we describe our in-network outlier cleaning approach in detail. Section 4 presents the evaluation results. Section 5 discusses the related work, and we conclude this paper in Section 6.

2 Background and Overview

2.1 Sensor Network

In recent years, wireless sensor networks have been growing as a platform for environmental monitoring in agriculture fields, battle fields, wild forests, coal mine tunnels, and so on [5, 6, 7]. Sensors are massively deployed to cover a wide geographical area. They have the capability of sensing the area, performing some computation, transmitting and forwarding the data to a centralized sink node. However, these small sensors, also called motes, have their limitations. Two of the most important ones are the limited battery power and the high transmission cost. These limitations make the design of sensor network data processing challenging. It is commonly recognized that in-network processing (aggregation) is beneficial [8, 9, 10, 11]. Part of the data processing is performed earlier, when the data are still in the network. Notice that the centralized approach processes the data only after all of them are collected to the sink. In in-network processing, each sensor takes up some computation according to the applications (e.g. query processing, data collection, event detection, and so on). The sensors try to compute and send the "aggregated" results to reduce network traffic. Since data transmission is the most costly operation in sensors [4], compared with it, the energy cost of in-network computation is trivial and negligible.

2.2 Data Collection

Sensor network applications can be classified into several categories. One kind of popular applications is query processing, which sends out a SQL-like query to the distributed sensors, and expects them to answer it by sending back the results to a sink node [8]. Another is event detection, in which a sensing report is triggered not by a query, but by the occurrence of an event [12]. Such an event can be a fire in a forest, a gas leakage in a coal mine, or a flood in an agriculture field. The third kind of applications is data collection, which is considered in this paper. These applications collect the entire sensing data over a long time, and store them centrally in a centralized database. Sophisticated data processing and analysis, which is not suitable to be run in sensors, can be carried out in the central server. Data collection is required in many scientific applications, where a scientist usually wants to record the historical monitoring data of the whole geographical area for his/her research. For example, a research for the cause of a freshet would need soil PH, river level, and humidity data over several years. In this paper, the design of our outlier cleaning approach is described based on the data collection applications. However, the idea of using wavelet-based outlier correction and neighboring DTW distance-based outlier removal can be modified to apply to query processing and event detection applications without losing generality.

2.3 Outlier Definition

In this paper, outliers of sensing data are referred to as abnormal sensed values that are from out of order sensors. The nature of environmental monitoring shows that sensing series are always temporally and geographically similar. Thus, outliers are those weird sensor readings that are dissimilar with the others. More specifically, we define two kinds of outliers based on this observation:

Short simple outlier A short simple outlier is a high frequency noise or error. It is usually represented as an abnormal sudden burst and depression, which is dissimilar to the other part of the same sensing series.

Long segmental outliers A long segmental outlier is the erroneous sensed readings that last for a certain time period. It is unreal and cannot reflect the environmental change of its monitoring area during that time period.

2.4 Outlier Cleaning

Outlier cleaning in this paper means both outlier correction and removal. In *outlier correction*, each sensor tries to correct a sensing series that contains outliers. The outlier value is substituted by a close approximation of the real value. It then sends the corrected sensing data to the sink. On the other hand, *outlier removal* discards the sensing data that are detected to have outliers, and are largely damaged or considered to have little usage. Intuitively, the two outlier cleaning approaches should be connected in series. One approach should correct the sensing data first, and the other one should then be used to detect long segmental outliers. It is not valid to simply delete the outliers, because many of them, containing only a little, short, occasionally appeared outliers are still usable after correction. Every piece of sensing data is valuable in a data analysis. It is only when the outliers in the sensing series are too erroneous to correct, then discarding this sensing series becomes the only choice to save transmission power. Mapping into the outlier definition above, the short simple outlier is much easier to be corrected, and the long segmental outlier need to be removed when it is not correctable.

2.5 Temporal and Spatial Similarity

The environmental data collected from widely distributed sensors are by their nature similar temporally and spatially [13]. This temporal and spatial similarity has special meaning in outlier cleaning. Given a sensing series, a short simple outlier is easy to be identified by human observation because it is shown as a sudden change and extremely different from the rest of the data. Theoretically, this sudden burst or depression is of high frequency in the frequency domain. They can be removed by de-noising techniques that transform the data into another domain where the high frequency noise and the low frequency true data can be separated. A long segmental outlier that lasts for a certain time period is not easy to be detected by only examining one sensing series, because it is hard to tell whether it is an outlier or the true data are changing in that pattern. However, making use of the spatial similarity of the sensing data, the outlier sensor should stand out when compared with the other sensors that monitor the same area. Here we make an assumption that sensors are largely and redundantly deployed, and each sensing area is monitored by several sensors. Therefore, an environmental change in an area will have similar, not necessary the same, effect on all the geographically close sensors.

3 Outlier Cleaning

We propose two outlier cleaning approaches:

1. using wavelet-based approach to correct outliers;
2. using neighboring DTW distance-based similarity comparison to detect and remove outliers.

These two outlier cleaning approaches are intended for in-network data processing within sensor networks. However, they are also applicable to centralized outlier cleaning.

3.1 Outlier Correction

Wavelet analysis has long been acknowledged as an efficient de-noising approach. A time series is transformed into the time-frequency domain. The wavelet coefficients represent a gradually refined resolution of the original time series. Most of the energy and information of the data are concentrated in a small number of coefficients, usually the first few coefficients. The sensing noises and errors are of high frequency and reside in high-order coefficients. Therefore, the true data and outliers, which are a kind of noise, can be separated in the wavelet space.

In outlier cleaning, simple short outliers can be corrected by wavelet de-noising. Figure 1 shows an example of using 5, 10, 20, or 30 wavelet coefficients to represent a sensing series with 128 points, which has an outlier at the 48^{th} point. We can observe that the fewer the coefficients used, the smoother and coarser the wavelet restored sensing series. Choosing an appropriate number of coefficients, 10 or 20, we can remove the outlier while keeping a close approximation of the original sensing series.

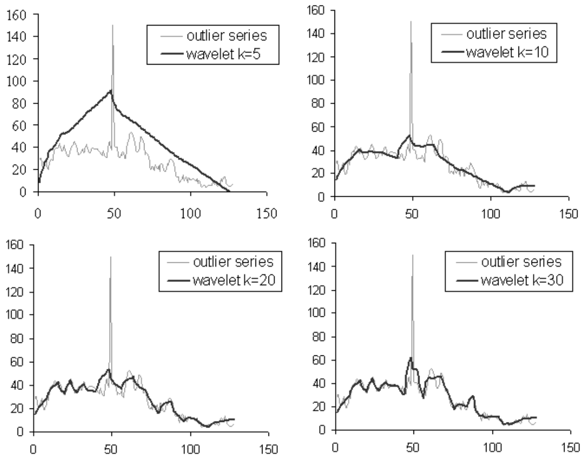


Figure 1: Outlier correction (Using k= 5, 10, 20, or 30 wavelet coefficients to represent the sensing series. The gray curve is the sensing series with an outlier occurring at the 48th point. The bold curve is the restored sensing series.)

In addition, wavelet transform also acts as a dimension reduction method. Transmitting the selected wavelet coefficients instead of the original sensing series can reduce data traffic by more than a magnitude. Moreover, wavelet transform, like DWT, only takes O(n) running time, which is a reasonable computational complexity for small limited devices like sensors.

By a small modification, the outlier correction approach can be applied to the case that the exact values of the non-outlier points are required. This means the raw sensing data should be sent to the sink instead of the smaller amount of wavelet coefficients. Correction is done by comparing the original sensing series (with possible outlier contained) with the wavelet restored sensing series. An outlier threshold is predefined by the user. If at a point p, the difference between the original and restored values is larger than the outlier threshold, p is counted as an outlier. We then use the restored value at p to correct the outlier value, and send out this corrected series. However, transmitting the raw data is not energy efficient in sensor networks, which will only be used when a specific application requires so. In the rest of this paper, we will stay with the preferred approach of transmitting wavelet coefficients.

3.2 Outlier Removal

Long segmental outlier detection is based on the neighboring similarity measurement. We notice that environmental change is not isolated, which means any change (increasing or decreasing) will affect a close area instead of only a single point. Since sensors are always densely and redundantly deployed, nearby sensors will have similar patterns. Here we assume that each environmental area is monitored by several sensors. The idea of outlier detection is to compare a sensing series with that of its neighbors. If a sensing series has a similar counterpart among one of its

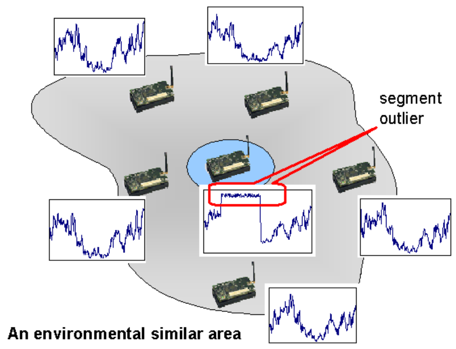


Figure 2: Spatially similar sensing series: sensors in the gray region are monitoring the same environmental area, and therefore have similar sensing series. The circled sensor is an outlier sensors, which can be known from its distinct sensing series.

neighbors, it is not an outlier because the probability of two failed sensors to generate similar erroneous series is very small. If a sensing series does not have any similar counterpart among its neighbors, it is higly possible to be an outlier sensor. Figure 2 plots an example of a spatially similar region, in which all the sensors should have similar sensing series. The circled sensor, which is quite different from the other sensors around it, is detected as an outlier sensor.

We use Dynamic Time Warping distance (DTW) to measure the similarity of two sensing series. The reason for not using the simpler Euclidean distance is that: (a) first, sensors in a network are loosely synchronized, so the sensing series are not aligned exactly; (b) second, there are different delays for different sensors to detect an environmental change, e.g. a fire occurs at one sensor takes a little while to spread to its neighbors. Due to the above two reasons, Euclidean distance is not suitable for measuring the similarity of sensing series.

DTW is a method that can compare two time series having elastic shifting on the time axis. They are considered to be similar, although out of phase. In Figure 3, the two time series are of similar shape, but not aligned in the time axis. Euclidean distance compares the i^{th} point of one series with the i^{th} point of the other, and reports a dissimilar result. However, DTW distance compares the dynamic warped points as shown in the figure, and therefore can capture the similar shape of the two series. The DTW algorithm are based on dynamic programming. The classic DTW algorithm takes $O(n^2)$ time to warp two time series each with n points. This quadratic algorithm is too much for limited sensing devices. In practice, accurate approximation like FastDTW is installed in the sensors, which can run in linear time and space [14].

3.3 Centralized Cleaning Process

Centralized outlier cleaning is carried out after all the data are collected to the sink. Outlier cleaning is done step by step:

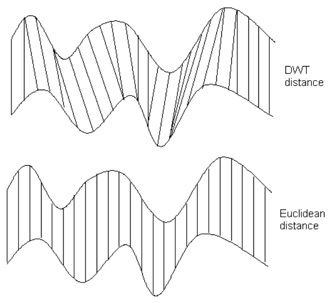


Figure 3: Euclidean distance and DTW distance

1. transform the sensing series to the wavelet domain;
2. reconstruct the sensing series using the first few coefficients;
3. compare the original series with the restored series to detect outlier points;
4. use the value in the restored series to correct the outlier points;
5. compare the sensing series with that of its geographically close neighbors;
6. detect an outlier sensing series if it is dissimilar with its neighboring series.

3.4 In-network Cleaning Process

The above outlier cleaning process can be moved down to the network level. It depends on the underlying data routing, so that the distributed sensors can clean the outliers during the data collection process.

3.4.1 Data Routing

In almost all techniques for in-network aggregation, a routing tree or graph based on hop count is established. Data are propagated from sensors to a sink through a minimum hop-count path [3]. This minimum hop-count based routing is constructed as follows: The sink broadcasts an initial message to the sensor network, containing a hop count parameter. All the sensors receiving this message select the sender (now is the sink) as their parent. They then increase the hop count parameter by one, and rebroadcast the message. Finally, the message is propagated to the entire network, and each sensor is assigned a hop count number. During the data collection, a current hop count number is sent with the data message. By this means, the message is routed through the reversed path, which is a hop count decreasing path, to the sink as illustrated in Figure 4(a). For the sake of aggregation, sensors are loosely synchronized and the data are collected hop by hop up to the sink. All the sensors with hop count number N are scheduled to transmit at a time period. Sensors with hop count number N-1 are scheduled in the next time period after the hop count N sensors have transmitted their data.

In this routing algorithm, each sensor can obtain some local topology information. Assume that sensor A has *hopcount* $= N$. First, A knows its direct

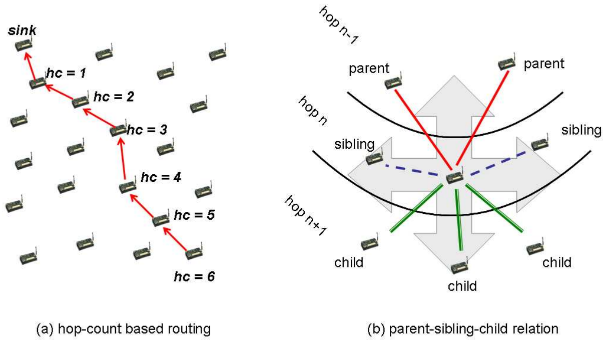


Figure 4: Topology of the in-network outlier cleaning

parents, the nodes who propagated the initial message with *hopcount* $= N - 1$ to it. Second, A knows its siblings, who are the nodes that propagated the initial message with *hopcount* $= N$. Third, A knows its direct children, the nodes from whom a data message was received with *hopcount* $= N + 1$. The above information is obtained defaultly in the routing protocol. Besides these, the sensor will actively keep a sibling list for each of its children. The child node attaches its sibling list with the first data message sent to the parents, therefore, the sibling list for each child is known by the parent.

3.4.2 Neighboring Relation

Unlike the centralized approach, in in-network aggregation, each individual sensor does not know its geographically nearest neighbors. The neighboring nodes considered in in-network outlier cleaning are the children, one-hop away siblings, and parents.

The parent-sibling-child relation is set up in the data routing. As illustrated in Figure 4(b), parents, children, and siblings can cover the four major directions of a sensor - up, down, left, and right. We can detect a outlier sensor by comparing it with its network neighbors in the four directions.

3.4.3 Cleaning Process

In the outlier cleaning process, wavelet-based outlier correction is done by each sensor, and the neighboring DTW similarity is compared along the routing path of the data that goes to the sink.
At the sensor level, each sensor

1. transforms the sensing series to the wavelet domain, and
2. selects the first few coefficients to transmit.

At the network level, sensor A receives its children's sensing series, and decides which to forward and which to delete. The outlier detection process is as follows"

1. Sensor A reconstructs the children sensing series from their wavelet coefficients.
2. A calculates the DTW similarity of its children's sensing series and itself's.
   (a) If they have similar sensing series, both A and those similar children are flagged as **Non-outlier**.
   (b) If all the children series are dissimilar, A is flagged as **Unknown**, since other neighbors need to be compared before making an outlier decision for A's sensing series.
3. When A's sensing series is transmitted to its parent B. For a child's sensing series flagged as **Unknown**, B compares it first with the series of the siblings of this child, then with the sensing series of B itself. Remember that a sibling list for each child is maintained when the routing paths are set up.
   (a) If there is a similar sensing series to that of this **Unknown** child, it is not an outlier and should be forwarded.
   (b) If all the sensing series are dissimilar, this child is finally detected as an outlier after comparing with all its network neighbors. The sensing series of this child is removed from the forwarding list.

For each sensor, it's sensing series is compared with those of its children when it receives the children's sensing series. The comparison of siblings and parents is done by the parent sensors. In the case when a sensor node has multiple parents, each parent would conduct its comparison independently. An outlier sensing series can be detected and removed by all the parents.

4 Evaluation

We have simulated a sensor network of about 900 sensors deployed in a grid topology. The transmission range of each sensor covers the upper, lower, left, right, upper and lower left, and upper and lower right sensors. The environmental sensing series of each sensor is generated from a temporal-spatial model. We have conducted simulations to evaluation our outlier cleaning approach under several evaluation metrics.

4.1 Evaluation Dataset

The evaluation datasets are generated from a model that simulates an area with temporal-spatial correlated environment. A number of points called event trigger have been placed in the simulation area. The sensed value of the event trigger follows a random walk. The location of the event trigger is also a random walk on the 2D simulation area. The value of a sensor at time t is the weighted combination of the values of the event triggers, where the weight is the inverse of the normalized distance between the target point and the event trigger point. Figure 5 plots a snapshot of the changing environment on a square monitoring area at a certain time. The values on the vertical axis are the current sensing readings. Finally,

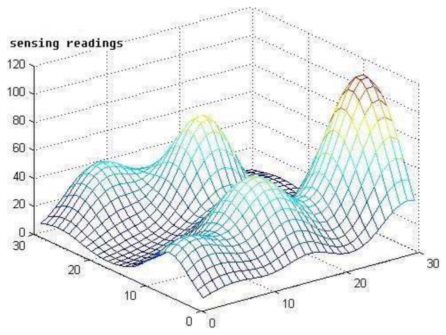


Figure 5: A snapshot of a temporal-spatial correlated area

outliers are added to the dataset as random errors. An outlier may affect only one data point in the sensing series, or affect a segment lasting for a certain time period. Different amounts of outliers can be introduced, distributed randomly in the area and in time. These outliers can be of different lengths. We have adjusted the model parameters and generated several environmental datasets in the simulation.

4.2 Evaluation Metrics

correction ratio It measures how much the outliers are corrected in order to be closer to the real value. Given the error of the outlier value, and the error of the corrected value, the correction ratio is calculated as follows:

$$cratio = \frac{\texttt{outlier error} - \texttt{corrected error}}{\texttt{outlier error}}$$

precision and recall This metric is used to measure the performance of outlier detection using DTW based outlier removal. Precision is the ratio of the correctly detected outliers and the total number of the detected outliers. Recall is the ratio of the correctly detected outliers and the total number of outliers.

transmission bytes We use the total number of transmission bytes in the network to measure the reduced traffic amount. This can represent the amount of energy saved in data transmission.

4.3 Results

In this section, we will evaluate our in-network outlier cleaning approach in different scenarios. If not explicitly specified, the default parameters listed in Figure 6 are used in the simulation.

4.3.1 Outlier Correction Ratio

We first evaluate the wavelet-based outlier correction approach when choosing different number of wavelet coefficients to represent the sensing series. We have added different amounts of outliers into the simulation scenarios - 500, 1000, 1500, and 2000 outliers. Each outlier was a single burst. The percentage of outliers

network size	30×30
the number outlier sensors	300
sensing series length	128
outlier length	$10 \sim 100$
the number of wavelet coefficients	10
DTW threshold	20

Figure 6: Default parameters in the simulation. We test the change of the parameters in our evaluation.

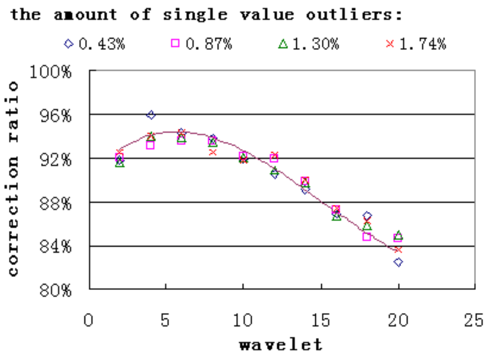


Figure 7: Correction ratio of wavelet-based correction

in the dataset was $0.42\% \sim 1.74\%$. Figure 7 plots the simulation results. The more the wavelet coefficients used, the finer the granularity. The high order wavelet coefficients can capture the outlier burst, so the correction ratio keeps decreasing. On the other hand, if too few wavelet coefficients are used, the restored sensing series is too coarse to correct the outlier. The best correction ratio exists at 5 coefficients. Choosing 5 to 12 coefficients can give a correction ratio of over 90%. To have a good approximation of the original sensing series, we have chosen 10 coefficients in the rest of the simulations.

4.3.2 DTW Threshold

In neighboring DTW distance-based outlier removal, we have used a DTW threshold to decide whether two sensing series are similar or not. If their DTW distance is smaller than the DTW threshold, they are regarded as similar sensing series, and vice versa. We have sim-

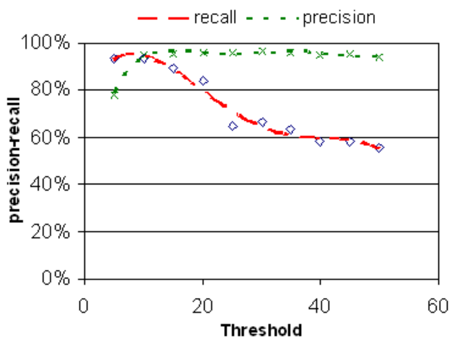


Figure 8: Recall and precision in changing DTW threshold

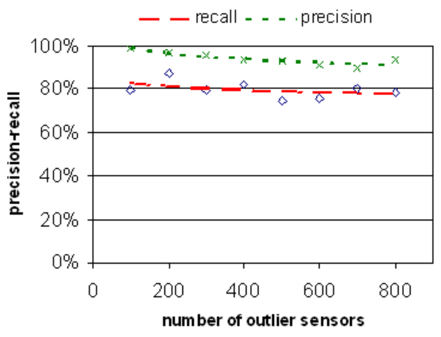


Figure 9: Recall and precision in changing outlier amount

ulated the different setting of DTW threshold and the results are shown in Figure 8. The precision of detecting outliers can be very high when using a threshold larger than 10. However, recall keeps decreasing because when the threshold is high some outliers are mistakenly counted as valid data.

4.3.3 Outlier Amount

We have also simulate different amount of outliers. As in Figure 6, the length of these outliers is randomly chosen from 10 to 100. The number of outliers increases from 100 to 800. We have limited each sensor to have at most one outlier. Hence, 800 outliers means 8/9 sensors suffer from failure. Figure 9 shows that both recall and precision are high. They are almost not affected by the amount of outliers.

4.3.4 Outlier Segment Length

In all the other scenarios, the length of an outlier segment was randomly chosen in the region $[10, 100]$. In this part, we have tested how the outlier length would affect outlier cleaning. We have explicitly set the outlier length to be 10 to 100 in different runs of simulations, and compared their results. The simulation results are plotted in Figure 10. Precision remains high under different outlier lengths, which means our algorithm rarely reports non-outlier sensors as outliers. However, recall is low when the outlier length is short, which means many of the true outliers are not detected. One possible reason is that the shorter outliers have already been corrected by wavelet approximation. We have justified this by examining the missing outliers (undetected outliers) to see how many of them are corrected by wavelet-based outlier correction. Figure 11 shows the total amount of detected and corrected outliers, where an outlier is counted as corrected if its error after correction is smaller than 1.0.

4.3.5 Traffic Reduction

Finally, we have evaluated the amount of traffic reduction in the outlier cleaning process. Since only 10 wavelet coefficients have been used for each 128 point

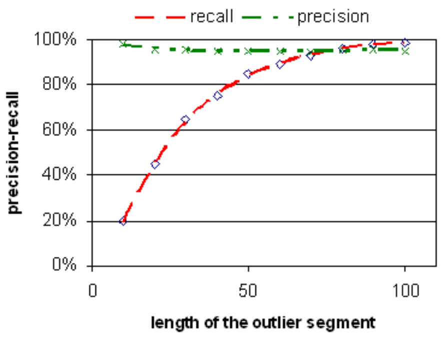


Figure 10: Recall and precision in changing outlier segment length

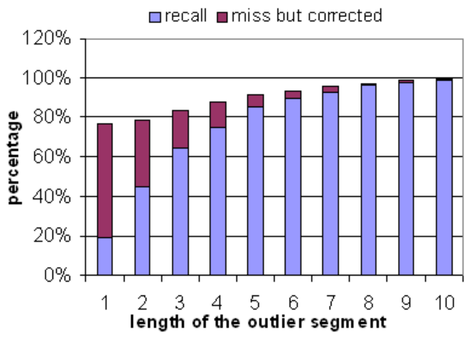


Figure 11: The total amount of corrected and removed outliers

long time series, the traffic reduction in wavelet correction is about 92.19%. If an outlier is detected, the traffic of transmitting and forwarding this outlier is saved. Since a sensor is normally routed through a multihop path to the sink, one outlier detection will save several hops of transmission. Figure 12 shows that with the increasing number of outliers, the amount of reduced traffic in DTW-based outlier removal is also increasing.

5 Related Work

Outlier detection is a fundamental issue in data management. Hawkins defines outlier as an observation that deviates a lot from other observations, and is

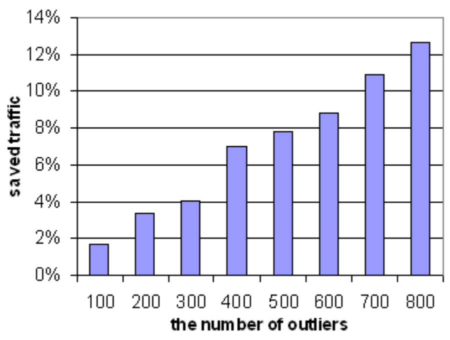


Figure 12: The percentage of traffic reduction

very possible to be generated from a different mechanism [15]. Thus, outlier detection is also called deviation detection. Most of the outlier detection techniques are based on data mining. Hodge and Austin classified a variety of outlier detection methods into three categories in their survey paper [16]: *Unsupervised clustering* based on distance and density, which determines the outliers with no prior knowledge of the data [17]; *supervised classification* that required prelabelled data and a machine learning process [18]; and *semi-supervised methods* that can tune the detection model incrementally as new data arrive [19]. Most of the proposed outlier detection approaches are centralized and off-line. They cannot be applied to sensor network applications directly.

Only a few papers have tried to address in-network outlier detection in the context of sensor networks. Palpanas *et al.* have proposed an in-network approach for distributed online deviation detection for streaming data [20]. However, this approach highly depends on the existence of high capacity sensors to manage groups of other sensors and perform outlier detection. Another related work proposed by Branch *et al.* uses a non-parametric, unsupervised method to detect outliers. They also use the distance-based metrics in the detection [21]. Hida *et al.* proposed a method to perform outlier detection in query processing (such as Max and Avg), so that query aggregation can be more reliable [22]. These approaches do not combine the temporal spatial similarity in outlier detection, because they detect outliers as a single value. However, in this paper we try to detect outliers in a number of time series. As a first step, we use wavelet based outlier correction and DTW distance-based outlier removal, which can be thought of as a distance based approach. This requires that the data in the whole area exhibit the same distribution, and the user should have some knowledge of the data to set an appropriate threshold. Our future work tries to address the outlier problem when the data are of different distribution. In this case, a single threshold may not be appropriate, and a sophisticated statistical model is required [23].

6 Conclusions

In this paper, we have presented an in-network outlier cleaning approach for sensor network data collection applications, using wavelet based outlier correction and DTW distance-based outlier removal. We have considered the spatial-temporal correction of environmental data; we not only detected but also tried to correct the outliers; we were able to remove the outliers within 2 network forwarding hops and reduce a large amount of the traffic. We have evaluated our approach under comprehensive simulations.

References

[1] S. Shekhar, C. T. Lu, and P. Zhang, "Detecting graph-based spatial outliers: Algorithms and applications," in *SIGKDD*, 2001.

[2] Berndt and Clifford, "Using dynamic time warping to find patterns in time series," in *KDD Workshop*, 1994.

[3] J. Al-Karaki and A. Kamal, "Routing techniques in wireless sensor networks: a survey," *IEEE Wireless Comm.*, no. 4, pp. 6–28, 2004.

[4] G. Pottie and W. Kaiser, "Wireless integrated network sensors," *Communications of the ACM*, vol. 43, no. 5, p. 51C58, 2000.

[5] A. Mainwaring, J. Polastre, R. Szewczyk, and D. Culler, "Wireless sensor networks for habitat monitoring," Intel Research, Tech. Rep. IRB-TR-02-006, Jun. 2002.

[6] R. Cardell-Oliver, K. Smettem, M. Kranz, and K. Mayer, "A reactive soil moisture sensor network: Design and field evaluation," *International Journal of Distributed Sensor Networks*, vol. 1, 2005.

[7] L. M. Ni, Y. Liu, Y. C. Lau, and A. Patil, "Landmarc: Indoor location sensing using active rfid," *ACM Wireless Networks*, vol. 10, no. 6, pp. 701–710, November 2004.

[8] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *OSDI*, 2002.

[9] M. A. Sharaf, J. Beaver, A. Labrinidis, and P. K. Chrysanthis, "Balancing energy efficiency and quality of aggregate data in sensor networks," in *VLDB*, 2004.

[10] A. Manjhi, S. Nath, and P. B. Gibbons, "Tributaries and deltas: Efficient and robust aggregation in sensor network streams," in *SIGMOD*, Baltimore, Maryland, USA, 2005.

[11] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *ICDE*, Boston, Massachusetts, USA, 2004.

[12] W. Xue, Q. Luo, L. Chen, and Y. Liu, "Contour map matching for event detection in sensor networks," in *SIGMOD*, 2006.

[13] D. Nychka, W. Piegorsch, and L. Cox, *Case Studies in Environmental Statistics.* Springer, New York, 1998.

[14] S. Salvador and P. Chan, "Fastdtw: Toward accurate dynamic time warping in linear time and space," in *KDD Workshop*, 2004.

[15] D. Hawkins, *Identification of Outliers.* Chapman and Hall, 1980.

[16] V. Hodge and J. Austin, "A survey of outlier detection methodologies," *Artificial Intelligence Review*, vol. 22, no. 2, pp. 85–126, October 2004.

[17] Rousseeuw and A. Leroy, *Robust Regression and Outlier Detection*, 3rd ed. J. Wiley, New York, 1996.

[18] G. Williams, R. Baxter, H. He, S. Hawkins, and Lifang.Gu, "A comparative study of rnn for outlier detection in data mining," in *ICDM*, 2002.

[19] D. Zhang, D. Gatica-Perez, S. Bengio, and I. McCowan, "Semi-supervised adapted hmms for unusual event detection," in *CVPR*, 2005.

[20] T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos, "Distributed deviation detection in sensor networks," *ACM SIGMOD*, vol. 32, no. 4, pp. 77–82, October 2003.

[21] J. W. Branch, B. K. Szymanski, C. Giannella, R. Wolff, and H. Kargupta, "In-network outlier detection in wireless sensor networks," in *ICDCS*, 2006.

[22] Y. Hida, P. Huang, and R. Nishtala, "Aggregation query under uncertainty in sensor networks," Department of Electrical Engineering and Computer Science. University of California, Berkeley, Tech. Rep., 2004.

[23] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, and D. Gunopulos, "Online outlier detection in sensor data using non-parametric models," in *SIGMOD*, 2006.

The Bellman Data Quality Browser

Divesh Srivastava

AT&T Labs – Research
divesh@research.att.com

Keynote Talk Abstract

Data quality is a serious concern in complex industrial-scale databases, which often have thousands of tables and tens of thousands of columns. Commonly encountered problems include missing data (null values), duplicates and default values in columns supposed to treated as keys, data inconsistencies (violation of functional dependencies), and poor quality join paths (lack of referential integrity). Compounding the data quality problems are incomplete and out-of-date metadata about the database and the processes used to populate the database. These problems make the task of analyzing data particularly challenging. To effectively address such problems, we have built the Bellman data quality browser at AT&T. Bellman profiles the database and computes concise statistical summaries of the contents of the database, to identify approximate keys, frequent values of a field (often default values), joinable fields with estimates of join sizes paths, and to understand database dynamics (changes in a database over time). In this talk, I'll describe the technology underlying Bellman and how it is used to help make sense of complex databases.

Data Cleaning for Decision Support

Michael Benedikt Philip Bohannon Glenn Bruns

Bell Laboratories, Lucent Technologies

Abstract

Data cleaning may involve the acquisition, at some effort or expense, of high-quality data. Such data can serve not only to correct individual errors, but also to improve the reliability model for data sources. However, there has been little research into this latter role for acquired data. In this short paper we define a new data cleaning model that allows a user to estimate the value of further data acquisition in the face of specific business decisions. As data is acquired, the reliability model of sources is updated using Bayesian techniques, thus aiding the user in both developing reasonable probability models for uncertain data and in improving the quality of that data. Although we do not deal here with the problem of finding optimal methods for utilizing external data sources, we do show how our formalization reduces cleaning to a well-studied optimization problem.

1 Introduction

When decisions must be made without good information, it is critical to know how much the information at hand can be trusted. To this end, techniques to manage and improve data quality have been developed by the database research community. Data correctness can be captured with integrity constraints [1], while data quality and confidence can be captured in probabilistic databases (e.g. [11, 12]). Work on constraint repair (e.g. [3, 4]) and on purging duplicates (e.g. [5, 9]) supports the automated improvement of data quality, for example by removing unlikely tuples or identifying dissimilar representations of the same entity. However, the problem of connecting data cleaning to the decision making process it supports has not been studied in any detail. Similarly, little attention has been focused on *how* to develop probabilistic models for data or *when* to expend effort or money to acquire good data. We now introduce a running example to illustrate these issues and motivate a model of data cleaning driven by *business decisions.*

Example 1. Consider a hypothetical telecommunications service provider, ACME Telecom. ACME is interested in building a wireless network in Alberta, which requires the construction of cellular towers. To see if it can build a network with sufficient capacity, ACME requires data on the sites available for cellular towers, and the towers they can support.

ACME has obtained catalogs of possible sites from two consulting companies, and represents this information in the two tables of Figure 1. The tables conform to the following schema:

site(Locid: int, Towertype: ttype)

Locid is the deed number for a site and ttype gives the type of towers it can support: "High" for a high-capacity tower, "Low" for a low-capacity tower, and "CU" if the site is Currently Unavailable, but may be able support a cell tower in the future.

The data suggests that one or both lists are of low quality. To what extent can existing data cleaning techniques be applied to ACME's problem? As mentioned above, one class of techniques deals with finding *approximate matches* and reconciling entities across data sources [5, 9]. However, such "record linkage" problems are not an issue here, as we have assumed that the Locid information is correct. (If this assumption does not hold, approximate matches might be used to bring the data to the state shown in the example.)

A second class of techniques that might be applied are *constraint repair* techniques [3, 4]. To formalize this example as a constraint repair problem, one might compute the union of **Smith** and **Jones**, and then assert a key constraint from Locid to Towertype on the resulting table. The data could then be repaired by either deleting tuples or modifying values. Consider applying either approach to Site 121. Certainly, the key constraint can be repaired by deleting either the **Smith** or **Jones** tuple for Site 121, or by changing a "Low" value to "High", or vice versa. However, since there is no evidence to favor any of these repairs, neither delete-tuple constraint repair nor attribute-modification repair is likely to help ACME in this case.

Good repair decisions could be made if the source instances were enriched with reliability information

Smith

Locid	Towertype
121	"High"
128	"Low"
146	"CU"
177	"CU"
199	"Low"
203	"Low"

Jones

Locid	Towertype
121	"Low"
128	"Low"
146	"High"
177	"CU"
199	"Low"

Figure 1: Example site data.

giving the probability that a tuple or attribute value is correct. Such information could be modeled, for example, in a probabilistic data model [11, 12]. A critical question, however, is how this reliability information would be obtained. What probabilities should be assigned to each possible value for ttype for site 121? Furthermore, how can we assign confidence to our reliability estimate to reflect the amount of supporting data we have seen? Clearly, we must distinguish the case in which much historical data supports the unreliability of **Jones** from the case where there is little data available. To help ACME, a data cleaning system must have both a strategy to develop a reliability model, and a way to estimate and adjust the confidence in the model.

Data reliability estimates can be improved by obtaining data via external sources. For example, by obtaining the correct answer for site 121, ACME can help improve its reliability model. If **Smith** turns out to be correct, ACME might assume it is more likely that **Smith** is correct on site 146, and if this too turns out to be the case (or perhaps for several more tuples), ACME may choose to trust **Smith**'s value for site 203.

An important trade-off is thus between the expected improvement in data quality obtained by consulting external sources and the expense of data acquisition from these sources. Existing systems offer neither a way to quantify the expected improvement nor a way to estimate the value of that improvement. If a credible source offered ACME a perfect list, how much should ACME be willing to pay? We argue that this question can only be answered concretely with a model of the *business decisions* faced. To illustrate this point, we return to our example:

Example 1, continued. ACME has a critical decision to make: Should it build a new network in Alberta? There is a fixed overhead of 10,000,000 Canadian dollars for building the network, while once the network is built there will be a net profit of $100,000 for each high-capacity tower and $50,000 for each low-capacity tower. Clearly, errors in the data can have a dramatic impact on the correct choice for this decision. There is no doubt that ACME can determine the quality of this data and correct it: it can send a representative out to survey each site, and contact each site owner. But this is at an average cost of $1000 per record.

How should ACME proceed? Should it ignore the risk caused by the low quality data and make this decision before taking any cleaning action? Should it query a small number of tuples and then decide? Which tuples should be sampled, and from what source? Or should it sample more than once, dependent on the results of prior samples? The question of *what data to clean* and when is a crucial one for any organization managing decision-critical data. In general, the data may impact many decisions, and there may be many external sources available with which to validate data, each with a different reliability. We see that even in this simple example the correct cleaning policy is far from obvious.

In this work we propose a new data cleaning framework geared towards deciding which external data to query, based on its impact on business decisions and its value in updating the data reliability model. Our framework includes a formal model of **external sources**, in which a query on an external source entails a cost. We model **organizational decisions** as actions whose rewards are based on queries. This allows us to quantify the benefit of data cleaning actions. Our model of **source reliability** includes a parameterized data model in which the parameters can be estimated based on current knowledge, including data from external sources. A solution to the cleaning problem will then consist of a *sampling policy* telling which external data to query based on the current reliability estimates and the organization value of data.

In this short paper we concentrate on giving the formal model, and do not discuss solution techniques. However, we do indicate how our notion of a cleaning policy reduces the cleaning problem to a well-studied problem area, that of *Markov Decision Processes* [10].

2 Decision-driven Cleaning Framework

In this section we describe a framework for Decision-driven Cleaning, in the form of a *Cleaning Problem*, consisting of the following components:

- a relational signature for the observed data and true data
- an error model giving a distribution on errors, possibly with hidden parameters
- a set of data sources that can be queried to get higher-quality information on the target data, along with cost and quality information for these sources
- a set of business decisions with associated reward functions

A solution to a cleaning problem is an optimal *cleaning policy* that decides, for every possible history of source data sampling, whether to take a sampling action, or to make a choice for one or more business decisions.

We now discuss each of these elements in detail.

Relational Data Model Our data model is a simple model for integrated relational data. It consists of a fixed schema $\mathcal{R}$ and a set Src of sources. The schema $\mathcal{R}$ consists of a finite set of relations, a mapping att associating with each relation R a set $\mathsf{att}(R)$ of attributes, a mapping Dom associating with each a in $\mathsf{att}(R)$ a domain $\mathsf{Dom}(a)$, which can be either infinite ('string', 'int', 'real', etc.) or finite, and a subset $\mathsf{key}(R) \subset \mathsf{att}(R)$ representing a distinguished key of R.

An *instance* of a relation R is a set of tuples, where a tuple includes a value in $\mathsf{Dom}(a)$ for each a in $\mathsf{att}(R)$. A *global instance* includes a relation instance for each relation R in $\mathcal{R}$ and each source src in Src. We can equivalently present a global instance as one set of tuples, where each tuple t has a source annotation, $\mathsf{src}(t) \in \mathsf{Src}$. A global instance is required to have each of its relation instances satisfy the key constraints.

There is a distinguished source $\mathsf{src}_{\mathsf{true}}$ representing the true, or actual, data values. We let GLOB be the set of global instances for a given schema. We use GI to range over global instances, src to range over sources, and R to range over relations. We write $\mathsf{GI.src}$ for the instances with source src in global instance GI, and $\mathsf{GI.src}(R)$ for the instance of R in GI with source src. Given a source instance I for a relation R we let $\mathsf{key}(I)$ denote the set of key values witnessed.

Data Reliability Framework We first develop a general model of data quality in which source data quality models can be computed based on standard Bayesian reasoning. We then discuss a specific model which could be used in a concrete setting.

Our error models are probability distributions over global instances [7, 12]. An alternative would be to attach probabilities to attribute values [11]. For example, consider the tuple for site 121 in the **Smith** table of Figure 1. One could explicitly state that there is an 80% chance that the value of Towertype is actually 'Low', a 15% chance that it is 'CU', and a 5% chance that it is 'High'. However, such a distribution on attribute values can be derived from a distribution on global instances.

Clearly, a decision maker will not generally know the probability distribution P on global instances. As discussed earlier, we seek to support a) uncertainty about the probability distribution P and b) the ability to characterize and update that uncertainty. To do this we adopt a standard Bayesian approach. We assume that the errors in the observed instance are generated by some probabilistic process, where the parameters controlling this process are unknown. For example, one parameter might represent the accuracy of data entry clerks at **Jones**. Given n such *hidden parameters* in an application, we let PAR be the product space consisting of all n-tuples of hidden parameter values, and consider PAR as a probability space. Thus, the values of the hidden parameters control the probability of global instances, and the parameter values themselves lie in a probability space.

An *error model* over a schema $\mathcal{R}$ consists of a probability distribution $G(\theta)$ on GLOB indexed by an element θ in PAR, along with a smooth probability on PAR, called the *prior*. For example, in tossing a coin with an unknown bias, the probability of a head might be θ, and the prior on θ may be a uniform distribution (representing the case where we have no knowledge of the bias). For some set S of global instances, and some θ in PAR, we write $P(S \mid \theta)$ for $G(\theta)(S)$, the probability of seeing the set S of global instances given hidden parameters θ. Also, we write $P(\mathsf{GI} \mid \theta)$ for $P(\{\mathsf{GI}\} \mid \theta)$.

Although we cannot observe the hidden parameters directly, we can gain information about them by looking at the results of sampling. The updated density function represents our improved knowledge of the hidden parameters given the observation of a set S of global instances:

$$\mathsf{POST}_S(\theta) = P(\mathsf{par} = \theta \mid \mathsf{GI} \in S)$$

This is the *posterior distribution* on the hidden parameter space. For example, in coin tossing, the updated density function tells us how to adjust our prior as a result of observing some coin tosses.

Concrete Data Reliability Model We now specialize our general model to one having independent tuple-level error probabilities. For simplicity, we consider only schemas in which the domain of every non-key attribute is finite. Other schemas can be handled by replacing the uniform distribution over the finite domain used below by a probability distribution appropriate for the particular domain.

Our error model has the following hidden parameters. The parameter $\theta^R_{\mathsf{src},a}$ gives the probability that the value of attribute a of relation R in source src has been modified. The parameter $\theta^R_{\mathsf{src,Ins}}$ gives the probability that a tuple has been inserted into relation R of source src. The parameter $\theta^R_{\mathsf{src,Del}}$ gives the probability that a tuple has been deleted from relation R of source src.

A tuple in the parameter space PAR then contains a value for each instance of these parameters for every source src in Src, every relation R in $\mathcal{R}$, and every attribute a in $\mathsf{att}(R)$. The prior distribution on PAR is a β distribution, which is known to be convenient for calculation. In our examples, we generally assume a uniform distribution over the parameter space as the prior. However, we could easily accommodate other priors that represent error information gathered using historical data or other means.

Now we explain how to compute the probability of seeing an instance I' of a relation R for source src, assuming that we are given src, the true instance I for R, and the hidden parameters θ. For every key value kv appearing in I, the probability that kv is absent from I' is $\theta^R_{\mathsf{src,Del}}$. For every key value kv remaining in

I', the probability that the non-key attributes of kv are set in I' to particular values is as follows: for a non-key attribute a, the probability that the value for a in I' agrees with the value in I is $1-\theta^R_{\mathsf{src},a}$. The probability that the value for a in I' will be a particular value in $\mathsf{Dom}(a)$ different from the one in I is $\theta^R_{\mathsf{src},a}/(|\mathsf{Dom}(a)|-1)$. The probability that k new key values are inserted into I' is $\binom{|I|}{k}(\theta^R_{\mathsf{src},\mathsf{Ins}})^k(1-\theta^R_{\mathsf{src},\mathsf{Ins}})^{|I|}$ (each tuple in I leads to a new spurious tuple in I' with probability $\theta^R_{\mathsf{src},\mathsf{Ins}}$). If k new tuples are to be inserted, the key values for each of these are chosen randomly from the initial values in every interval in $\mathsf{Dom}(\mathsf{key}(R))-I$; for each of these tuples, the non-key value for attribute a is chosen to be a particular value from $\mathsf{Dom}(a)$ with probability $1/|\mathsf{Dom}(a)|$.

This calculation gives the probability $P(\mathsf{GI}.\mathsf{src}_{\mathsf{true}}(R) = I \wedge \mathsf{GI}.\mathsf{src}(R) = I' \mid \theta)$ for a source src and an element θ of PAR. The probability of a full global instance GI_0 given θ is then obtained by: $\Pi_{R\in\mathcal{R},\mathsf{src}\in\mathsf{Src}-\{\mathsf{src}_{\mathsf{true}}\}}P(\mathsf{GI}.\mathsf{src}_{\mathsf{true}}(R) = \mathsf{GI}_0.\mathsf{src}_{\mathsf{true}}(R) \wedge \mathsf{GI}.\mathsf{src}(R) = \mathsf{GI}.\mathsf{src}(R) \mid \theta)$

The model above allows for no correlation between errors in attributes. We can generalize to broader error models by fixing a Bayesian Network [13] with unknown weights, specifying the conditional probability of one attribute being correct given that another attribute is or is not correct. The use of such networks for modeling errors is the subject of future work.

Cleaning Model During the data cleaning process, a decision maker will sample data sources, paying to do so. A global instance in our data model represents the data to be potentially sampled, not the data already sampled in this process. The data already sampled is modeled here as a *sampling history*: a function SH that maps a source src and a relation R to a set $\mathsf{SH}(\mathsf{src}, R)$ of tuples. A tuple in the set either has a value for each attribute of R, or has a value for the key attributes and the distinguished value *null* for all other attributes. A sampling history SH is *consistent* with a global instance GI if, for every relation R and source src, 1) every tuple in $\mathsf{SH}(\mathsf{src}, R)$ not containing *null* also appears in $\mathsf{GI}.\mathsf{src}(R)$, and 2) if a tuple in $\mathsf{SH}(\mathsf{src}, R)$ does contain *null*, then no tuple having the same key appears in $\mathsf{GI}.\mathsf{src}(R)$.

We assume that our decision maker begins with some sampling history, which we refer to as the *initial sample*, denoted Init. This would typically contain all the data from some readily-available sources, and no data from expensive sources.

Each source src has an associated *cost function* $C_{\mathsf{src}} : N \to \mathbb{Z}$, where $C_{\mathsf{src}}(n)$ gives the cost of sampling src for a group of n tuples. The sampling cost might be linear in n, or might perhaps reflect that sampling can be done more cost-efficiently in bulk. Normally we expect that a user will not be able to sample the actual data $\mathsf{src}_{\mathsf{true}}$, but will be able to sample sources where the parameters $\theta^R_{\mathsf{src},\mathsf{Del}}$, $\theta^R_{\mathsf{src},\mathsf{Ins}}$, and $\theta^R_{\mathsf{src},a}$, are known and small.

A *sampling action* for a source src is a relation, R, and a set $\{\mathsf{kv}_1, \ldots, \mathsf{kv}_n\}$ of key values of appropriate type for R. The result of a sampling action is a sequence $\langle t_1, \ldots, t_n\rangle$ where t_i is either a tuple for R with key value kv_i, or *null*, indicating that no tuple with this value exists in the instance of R.

Example 2. Consider again the ACME example. For the data from Smith Inc. there is an (unknown) probability $\theta^{\mathsf{SM}}_{\mathsf{Towertype}}$ that the Towertype attribute is correct. In addition, there is a probability $\theta^{\mathsf{SM}}_{\mathsf{Del}}$ that a given true lot was deleted, and a parameter $\theta^{\mathsf{SM}}_{\mathsf{Ins}}$ controlling how many spurious tuples are inserted. Similarly, for the data from Jones, we have $\theta^{\mathsf{JS}}_{\mathsf{Towertype}}$, $\theta^{\mathsf{JS}}_{\mathsf{Del}}$, and $\theta^{\mathsf{JS}}_{\mathsf{Ins}}$ parameterizing the probability of an error.

We also assume that we have a correct source, with a linear cost per tuple of sampling.

Suppose we know from historical data that for each consulting company between 0 and 1% of the available lots for the coming year are missed, and between 0 and .5% of the Locid values correspond to non-existent lots. Hence we can take a prior distribution on $\theta^{\mathsf{SM}}_{\mathsf{Del}}$ and $\theta^{\mathsf{JS}}_{\mathsf{Del}}$ a uniform distribution on the interval $[0, .01]$, while taking a prior on $\theta^{\mathsf{SM}}_{\mathsf{Ins}}$ and $\theta^{\mathsf{JS}}_{\mathsf{Ins}}$ as a uniform distribution on $[0, .005]$. For the rate of modification of the Towertype attribute, we have no historical data, so we take these to have a prior that is uniformly distributed on $[0, 1]$.

Suppose that we have sampled our oracle on 100 key values, and determined that all 100 Locid values for the Smith data are valid lots, and that for 99 of these the Smith report has the correct value for Towertype.

Then we can estimate a new posterior distribution on $\theta^{\mathsf{SM}}_{\mathsf{Towertype}}$ given this sample data and the observed data, with the density of $\theta^{\mathsf{SM}}_{\mathsf{Towertype}}$ now $(1-\theta^{\mathsf{SM}}_{\mathsf{Towertype}})^{99}\theta^{\mathsf{SM}}_{\mathsf{Towertype}}/\int_0^1(1-x)^{99}x\,dx$.

Business Decisions Associated with a cleaning problem is a finite set $\{\tau_1 \ldots \tau_r\}$ of *business decisions*, where each decision τ has an associated finite set $\mathsf{Choices}(\tau)$ of choices. Each decision τ and choice ρ in $\mathsf{Choices}(\tau)$ has associated with it a relational query $Q_{\tau,\rho}$ over the signature $\mathcal{R}$. Evaluating $Q_{\tau,\rho}$ on a (true) instance I gives the outcome if the decision maker chooses ρ for decision τ.

In the ACME example, our business decision τ is Build, with choices ρ_1 =Yes and ρ_2 =No. The reward query associated with Build = Yes is given by Q_{τ,ρ_1}:

```
select (100,000 * High.tot + 50,000 * Low.tot -
10,000,000) as Profit
from
(select cnt(*) as tot from Site where
Towertype='High') High,
(select 50,000 cnt(*) as tot from Site where
Towertype='Low') Low
```

The reward query associated with Build = No is $Q_{\tau,\rho_2} = 0$.

Cleaning Policies We can now take a *cleaning problem* to be a tuple $(\mathcal{R}, \mathsf{Src}, \mathsf{EMOD}, C, \mathsf{BD})$, where $\mathcal{R}$ is a schema, Src a set of sources, EMOD an error model giving a distribution over global instances of $\mathcal{R}$ for the sources in Src, C a cost function, and BD a set of business decisions. A *policy* for a cleaning problem is a function deciding, for each sampling history, either a sampling action, or a choice for one or more of the business decisions. A policy is a recipe telling what should be sampled at any state: given an initial sampling history Init, and a policy λ, one can apply λ repeatedly to get a sequence of histories and choices for business decisions: $\mathsf{SH}_1 = \mathsf{Init}$ unioned with the response of $\lambda(\mathsf{Init})$ on GI, $\mathsf{SH}_2 = \mathsf{SH}_1$ unioned with the response of $\lambda(\mathsf{SH}_1)$ on GI, etc. In the process, we obtain larger and larger sampling histories and some sequence of choices for business decisions. A policy is *valid* if on every global instance GI it produces a sequence such that every business decision $\tau_i : i \leq r$ is decided exactly once. For a valid policy, we can evaluate its effectiveness on a global instance GI via the *cumulative reward*: if the policy produces sampling actions $S_1 \dots S_k$ when applied on GI, then the reward is $\Sigma_{i \leq r} Q_{\tau_i, \rho}(\mathsf{GI}.\mathsf{src}_{\mathsf{true}}) - \Sigma_{j \leq k} C_j(|S_j|)$, where ρ is the choice selected for τ_i when running policy λ on GI and $C_j(|S_j|)$ is the cost of the j^{th} sample. That is, the reward is the gain from the business decisions minus the total cost of sampling.

The goal of cleaning (in our sense) is then to find the *optimal policy* for a cleaning problem, given an initial history Init. This optimal policy maximizes the expected value of the reward, conditioned on the event that the global instance is consistent with Init. The optimal policy tells the cleaner the "best" data to sample in a precise sense.

3 Ongoing Work

In our framework, a solution to the cleaning problem is an optimal strategy for a certain game. Optimization strategies for more general planning problems, such as those for Markov Decision Processes (MDPs) [8], are applicable here. We now very briefly review MDPs and their application to cleaning.

An MDP describes a game between a player and the environment in which the player chooses an action and the environment chooses a resulting state according to a probability distribution associated with the action. Each action has an associated reward function, and the goal of the player is to choose a strategy that maximizes her expected cumulative reward. It is easy to translate the cleaning problem here into an MDP: the states of the MDP are the sampling histories, while actions are sampling actions and choices for the business decisions. The rewards for sampling actions are the negative of the cost of sampling (based on the associated cost function of the sources), while the rewards for decision actions are the expected values of the corresponding queries, where the expectation is conditioned on the information known from the sampling history. The naive translation will yield a very large MDP (exponential in the size of the data). Given the fact that the best algorithms for solving general MDPs (based on dynamic programming) are quadratic, one cannot hope to use the straightforward approach in practice. In ongoing work we are investigating the use of abstraction techniques, along the lines of [6, 2], which may yield a more manageable problem.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] J. Baum and A. E. Nicholson. Dynamic non-uniform abstractions for approximate planning in large structured stocastic domains. In *Proceedings of the 5th Pacific Rim International Conference on Artificial Intelligence*, pages 587–598, 1998.

[3] A. Cali, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *PODS*, 2003.

[4] J. Chomicki and J. Marcinkowski. On the computational complexity of minimal-change integrity maintenance in relational databases. In L. Bertossi, A. Hunter, and T. Schaub, editors, *Integrity Tolerance.* Springer, 2004.

[5] W. Cohen, P. Ravikumar, and S. Feinberg. A comparison of string-distance metrics for name-matching tasks. In *IIWeb*, 2003.

[6] T. Dean and R. Givan. Model minimization in markov decision processes. In *Proceedings of the 14th National Conference on Artificial Intelligence*, pages 106–111, 1997.

[7] N. Fuhr and T. Rol̈leke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Trans. Inf. Syst.*, 15(1):32–66, 1997.

[8] G. E. Monahan. A survey of partialy observable markov decision processes: Theory, models, and algorithms. *Management Science*, 28:1–16, 1982.

[9] A. Monge and C. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *DKMD*, 1997.

[10] M. Puterman. *Markov Decision Processes – Discrete Stochastic Dynamic Programming.* John Wiley & Sons, 1994.

[11] A. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *Proc. of ICDE 2006.* IEEE Computer Society, 2006.

[12] D. Suciu and N. Dalvi. Foundations of probabilistic answers to queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 963–963, New York, NY, USA, 2005. ACM Press.

[13] R. L. Winkler. *An Introduction to Bayesian Inference and Decision.* Holt, Rinehart, and Winston, 1972.

Efficiently Filtering RFID Data Streams

Yijian Bai [†*] Fusheng Wang [‡] Peiya Liu [‡]

[†]UCLA
bai@cs.ucla.edu

[‡] Siemens Corporate Research
{fusheng.wang,peiya.liu}@siemens.com

Abstract

RFID holds the promise of real-time identifying, locating, tracking and monitoring physical objects without line of sight, and can be used for a wide range of pervasive computing applications. To achieve these goals, RFID data has to be collected, filtered, and transformed into semantic application data. RFID data, however, contains false readings and duplicates. Such data cannot be used directly by applications unless they are filtered and cleaned. While RFID data often arrives quickly and is in high volume, its detection usually demands efficient processing, especially for those real-time monitoring applications. Meanwhile, the order preservation of RFID tag observations are critical for many applications. In this paper, we propose several effective methods to filter RFID data, including both noise removal and duplicate elimination. Our performance study demonstrates the efficiency of our methods.

1 Introduction

RFID (radio frequency identification) technology uses radio-frequency waves to transfer data between readers and movable tagged objects. Thus it is possible to create a physically linked world in which every object can be numbered, identified, cataloged, and tracked. RFID is automatic and fast, and does not require line of sight or contact between readers and tagged objects. With such significant technology advantages, RFID has been gradually adopted and deployed in a wide area of applications, such as access control, library checkin and checkout, document tracking, smart box [1], highway tolls, supply chain and logistics, security, and healthcare [2].

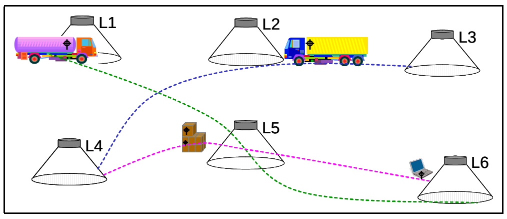


Figure 1: Pervasive Computing with RFID

One major problem to be solved in pervasive computing is to identify and track physical objects, and RFID technology is a perfect fit to solve this. By tagging objects with EPC [1] tags that virtually represent these objects, the identifications and behaviors of objects can be precisely observed and tracked. RFID readers can be deployed at different locations and networked together, which provides an RFID-based pervasive computing environment. This is illustrated in Figure 1, where L1 – L6 denote different locations mounted with readers. Tagged objects moving in this environment will then be automatically sensed and observed with their identifications, locations and movement paths.

Readers' observations, however, are raw data and can contain a lot of duplicate and false readings. Thus the first step to integrate RFID data into pervasive computing applications is to filter RFID observations.

RFID data are generated quickly and automatically, and can be used for real-time monitoring, or accumulated for object tracking. To filter the high volume real-time RFID data streams, efficient methods are essential, especially for real-time applications.

The filtered RFID data often need to preserve the original order, i.e., the first observed tagged object will be output first after filtering. Such order can be critical for many RFID applications. For example, a nurse uses a wearable reader to access RFID-tagged medical items according to medical procedures. The order the nurse accesses these medical items is critical: wrong orders may cause a medical error or even lead to fatal

*Work done while visiting Siemens Corporate Research.

CleanDB, Seoul, Korea, 2006

[1]EPC – Electronic Product Code – is an identification scheme for universally identifying physical objects, defined by EPC-Global [3].

result. Thus, the correct ordering of RFID observations together with a workflow monitoring system will minimize such errors.

In this paper, we propose effective and efficient algorithms for RFID data filtering, including noise removal and duplicate elimination.

The paper is organized as follows. We first introduce the background of RFID data filtering in Section 2. Then in Section 3 we propose algorithms to efficiently filter noise from RFID data, including the problem of order preservation in the output. Next we discuss algorithms for duplicate merging in Section 4. Performance study of these methods is discussed in Section 5, followed by Related Work and Conclusion.

2 Background

Due to the low-power and low-cost constraints of RFID tags, reliability of RFID readings is of concern in many circumstances [4, 5]. There are three typical undesired scenarios: *false negative readings*, *false positive readings* and *duplicate readings*, discussed as follows.

- *False negative readings.* In this case, RFID tags, while present to a reader, might not be read by the reader at all. This can be caused by i) When multiple tags are to be simultaneously detected, RF collisions occur and signals interfere with each other, preventing the reader from identifying any tags; ii) A tag is not detected due to water or metal shielding or RF interference.

- *False positive readings(or noise).* In this case, besides RFID tags to be read, additional unexpected readings are generated. This can be attributed to the following reasons. i) RFID tags outside the normal reading scope of a reader are captured by the reader. For example, while reading items from one case, a reader may read items from an adjacent case; ii) Unknown reasons from the reader or environment, for example, one of our readers periodically sends wrong IDs.

- *Duplicate Readings.* This can be caused by the following reasons: i) Tags in the scope of a reader for a long time (in multiple reading frames) are read by the reader multiple times; ii) Multiple readers are installed to cover larger area or distance, and tags in the overlapped areas are read by multiple readers; and iii) To enhance reading accuracy, multiple tags with same EPCs are attached to the same object, thus generate duplicate readings.

In practice, readings are often performed in multiple cycles to achieve higher recognition rate [5]. In this way, false negative readings can be significantly reduced. Meanwhile, noisy readings (or false positive readings) generally have a low occurrence rate compared to normal true readings. Thus only those readings that have significant repeats within certain interval are considered to be true readings. This, however, will further produce much more duplicate readings.

Based on above observations, we develop effective and efficient RFID data filtering techniques to generate clean RFID data, which can be further interpreted and integrated into RFID-based applications. In this paper, we study two types of filtering: noise is removed from RFID data (*denoising or smoothing*), and duplicates are merged into one distinct reading (*duplicate elimination, or merging*). We develop algorithms that, compared to baseline implementations, work more efficiently while requiring less buffer space for history storage for both *denoising* and *duplicate elimination.* Furthermore, we discuss the issue of output time ordering for *denoising* and show our method can address this issue efficiently.

3 Denoising in RFID Data Streams

Based on the discussion above, since multiple reading cycles are performed on tagged objects and noise readings normally have a low occurrence rate, we propose *sliding window* based approaches to solve the problem. A sliding window is a window with certain size that moves with time. Suppose the window with size `window_size` has a time coordinate of `[`t_1`, `t_1` + window_size]`, after τ, the coordinate will become `[`t_1` + `τ`, `t_1` + window_size + `τ`]`.

RFID reading tuples will enter the window and get expired as time moves. Therefore, the noise readings are readings with count of distinct tag EPC values below a noise `threshold`. Denoising essentially performs the following operations: within any time window with size of `window_size` surrounding an RFID reading, if the count of the readings with same tag EPC values appears equal to or above `threshold`, then the observed EPC value is not noise and needs to be forwarded for further processing; otherwise the reading is discarded. Two parameters used here are `window_size` of a sliding time window, and a `threshold` for noise detection.

An RFID observation (reading) is in the form of: (`reader_id, tag_id, timestamp`), which refers to the EPC [3] of the RFID reader, the EPC of the tagged object, and the timestamp of this observation respectively. In the algorithms presented below, the key of a reading can be usually considered to be the pair of (`reader_id, tag_id`) in the reading. For the case where multiple readers are used to observe same tags, the key will be `tag_id`.

Baseline Denoising: A Base Approach We first show a baseline implementation of denoising as shown in Algorithm 1, which we refer to as `baseline_denoising`.

In this algorithm, intuitively, for each incoming

Algorithm 1 Baseline_denoise (params: `window_size`, `threshold`)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   append INCOMING to the end of WINDOWBUFFER
5:   EXPIRETIME ← INCOMING.timestamp - window_size
6:   while the head of WINDOWBUFFER is older than EXPIRETIME do
7:     remove the head of WINDOWBUFFER
8:   end while
9:   COUNT ← count of readings in WINDOWBUFFER whose key equals to INCOMING.key
10:  if COUNT ≥ threshold then
11:    for each of the reading R in WINDOWBUFFER with key equals to INCOMING.key do
12:      if R has not been output before then
13:        output R
14:        set STATE-OF-OUTPUT as true
15:      end if
16:    end for
17:  end if
18: end loop
```

reading of value `R`, we perform a full scan of the preceding sliding time window of size `window_size`. If `R` appears more than `threshold` times within the window, we know this is not a noise reading thus we output every `R` in the window. To ensure a particular reading is never output more than once, we keep a `state-of-output` with each reading in the window buffer and set it to *true* once it is output once.

Complexity. Assume on average there are `n` readings in the sliding window, with `k` distinct keys. Since the operations are repeated for each incoming tag reading, we analyze the time cost on a per-reading basis. The bulk of the time cost is from 4 operations: inserting the incoming reading into the window, removing expired readings from the window, computing the count of the readings with the same key, setting `state-of-output` and outputting readings of the same key if threshold condition is satisfied. Since all readings are maintained in the same FIFO (First-In, First-Out) queue, both insertion of new readings (appending to the end of queue) and removal of expired readings (removing from the head of queue) can be considered constant time ($O(1)$) operations. (Strictly speaking, expiration is amortized ($O(1)$) per incoming reading here, since on average there is only one expiration per new arrival, although individual incoming reading may trigger different number of expirations.) On the other hand, both counting and setting `state-of-output` is performed by linearly scanning the full window. Counting is always performed for each incoming reading, thus the cost is $\Theta(n)$. Setting `state-of-output` and outputting only occur when threshold condition is satisfied, thus the cost can be considered to be bounded by $O(n)$, which leaves the total cost per incoming reading to be $O(1)+O(1)+\Theta(n)+O(n) = \Theta(n)$.

Space Cost. The space cost for the *baseline_denoise* algorithm is basically the storage for the sliding window itself, thus $\Theta(n)$.

It is natural to see that, with some additional space cost, we can incrementally maintain an extra counter for each distinct tag EPC value using a hashtable (which takes $\Theta(k)$ space), thus reduce the counting cost for each incoming key value. That is, for each incoming reading we increment the counter for the corresponding key in the hashtable, and for each expired reading we decrement the counter for the corresponding key. This reduces counting to an $O(1)$ operation, although we still can not avoid the $O(n)$ operation of setting `states-of-output` and outputting readings.

3.1 Lazy Denoising with Output Order Preserving Using Hashtable

There is one problem in the *baseline_denoise* algorithm: the output readings may be out of order if we output immediately upon determining a reading is non-noise, i.e., a reading observed earlier may be output later. This affects all further RFID data processing where correct ordering of observations is critical, such as complex RFID event detections for real-time RFID applications and RFID data aggregation [6]. For example, we may need to detect a certain sequence of events, A followed by B, if the order is reversed an alert has to be raised. In this scenario, not preserving output ordering of tags will result in both false alerts and false acceptances.

The following example shows how this out-of-order problem might happen.

Example 1: out-of-order observations. Suppose two tags are being read at two readers attached to the same host computer. Each tag is repeated 10 times with an interval of 100msec, thus the window size here is 1000msec. A reading is considered to be non-noise if it appears 6 times out of any 1000msec time-window around it. Assume the two tag keys are 1 and 2, and the actual readings appear in sequences as shown in Table 1, where tag 2 arrives 100msec later after tag 1 arrives. The readings of 4, 5, 8, 9 are noise[2].

Although tag 1 and tag 2 both have 2 noise readings in this example, due to different positions of the noise, ID 2 is actually determined as a non-noise reading first (at time 700msec), while ID 1 is determined as a non-noise later (at time 800msec), although tag 1 arrives earlier than tag 2. Therefore, if we output readings

[2]This example also illustrates how to set the `window_size` parameter for the algorithm. In most cases, this parameter is dictated by the repeat count of a tag, as well as the interval between repeats. The other parameter, `threshold`, however, will need to be tuned based on error rates.

Time(msec)	Tag 1 Reading	Tag 2 Reading
100	1	
200	4	2
300	1	2
400	1	2
500	5	2
600	1	2
700	1	2*
800	1*	8
900	1	2
1000	1	9
1100		2

Table 1: Arrival Time of Readings for Tag 1 and Tag 2 (* indicates the earliest time point that the reading can be determined as non-noise)

immediately after we detect them as non-noise, as is done in the *baseline_denoise* algorithm, we will then output readings with their timestamps out of order. If we represent the output as (`id, time`), then at time 700msec and 800msec the output for this example is:

```
Time 700: (2,200) (2,300) (2,400) (2,500) (2,600) (2,700)
Time 800: (1,100) (1,300) (1,400) (1,600) (1,700) (1,800)
```

Clearly, the reading of tag 1 at time 100msec will be output later than the reading of tag 2 at time 700mec. This will present a problem for any algorithm that is dependent on correct time-ordering of readings.

To solve the out-of-order problem, one solution is, when a reading is determined as non-noise, mark the reading as non-noise but not output it yet. The output happens only if a reading marked as non-noise gets expired from the window. With the FIFO queue for the window, it is therefore very efficient to output readings in their correct order.

Algorithm 2 – *Lazy_denoising* – incorporates the above-mentioned improvements. A hashtable of counters are maintained for each distinct key value `R` that is still present in the sliding window, and the corresponding counter is incrementally updated for each incoming tuple and expiring tuple. At any point of time, if the count of `R` in the window is higher than `threshold`, we mark all readings of `R` as non-noise. To ensure the correct output order, we delay the output of all non-noise tuples till they expire from the sliding window. At this point we know for sure all non-noise tuples will be in order, since the noise readings that have already expired will never turn to non-noise to affect the order.

Complexity. With incremental counter maintenance using a hashtable, the cost of counting operation for each incoming reading is reduced from Θ(n) to O(1), at the expense of an extra Θ(k) space. With output-on-expire, it guarantees that the output is in correct time order at no extra time or space cost. The cost of hashtable maintenance (inserting and removing keys from the hashtable) is on-average upper-bounded by O(1) for each incoming reading, and due to repeating, not every incoming reading will introduce a new key.

Algorithm 2 Lazy_denoising (params: `window_size`, `threshold`)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: TABLE ← empty hashtable {hashtable to map each key to a counter}
3: loop {loop forever for next incoming reading}
4:   INCOMING ← the next reading
5:   mark INCOMING as noise
6:   append INCOMING to the end of WINDOWBUFFER
7:   if the counter at TABLE[INCOMING.key] does not exist then
8:     store a counter at TABLE[INCOMING.key] with value 1
9:   else
10:    increment the counter at TABLE[INCOMING.key]
11:  end if
12:  EXPIRETIME ← INCOMING.timestamp - window_size
13:  while the head of WINDOWBUFFER is older than EXPIRETIME do
14:    if the head reading is marked as non-noise then
15:      output the head of WINDOWBUFFER
16:    end if
17:    remove the head of WINDOWBUFFER
18:    decrement the counter in TABLE for the corresponding key
19:    remove the slot in TABLE if the counter for this key becomes 0
20:  end while
21:  COUNT ← counter value at TABLE[INCOMING.key]
22:  if COUNT $\geq$ threshold then
23:    for each of the reading R in WINDOWBUFFER with key equals to INCOMING.key, by reverse time order do
24:      if R is marked as noise then
25:        mark R as non-noise
26:      else
27:        break the for loop
28:      end if
29:    end for
30:  end if
31: end loop
```

Notice that, in general, if each key is repeated for a fair amount of time (say 10 times, which is common in practice), and the noise ratio is small (say 1%), then `k` can be considered to be an order of magnitude smaller than `n`. As the noise ratio gets higher, the difference between `k` and `n` become smaller. If we assume each tag is repeated for r times, and overall there is a p percent chance that a reading is noise, then we have the relationship that $k = n * (\frac{1}{r} + p)$.

Baseline (Ordered). In the experiments section, a *Baseline (Ordered)* algorithm is used for comparison with *Baseline_denoising* and *Lazy_denoising*. This algorithm is exactly the same as *Baseline_denoising* when searching for non-noise readings, as it scans the full window each time. However, it also tries to output tuples in order by only outputting a reading when it expires from the window. The details of this algorithm are omitted here since it is a straightforward extension of *Baseline_denoising* and has exactly the same complexity bounds.

3.2 Eager Denoising: Output Data Early with Order Preservation

Although output-on-expire is efficient and straightforward, it does have a negative consequence of introducing more delay for outputting readings. Instead of being output on the fly at the time of determination to be non-noise, a reading will not be output until it is expired from the sliding window. This could be a problem if the width of the window is quite long. This indeed can be improved for situations where a reading can be output earlier while correct time order can still be preserved.

In fact, the issue of order disturbance occurs only if a reading has been output before the change of labeling on some earlier reading from noise to non-noise within the window. Therefore, for a non-noise reading that we know no other earlier noise reading is present in the sliding window, we can then safely output it without the risk of order problems. This technique is incorporated in Algorithm 3 – *Eager_denoise*.

Algorithm *Eager_denoise* (Algorithm 3) improves over Algorithm *Lazy_denoise* (Algorithm 2) by outputting non-noise readings more eagerly: as soon as there is no more noise before the non-noise reading within the sliding window, the non-noise reading is output. To achieve this, the algorithm keeps track of the first noise reading (FIRSTNOISE) inside the window at all times. Then an invariant is kept at the end of processing each incoming reading, such that all the non-noise readings before FIRSTNOISE are output, and all the non-noise readings after FIRSTNOISE are not. (In the case of no presence of noise, everything is output at the end of the processing of the incoming reading). To maintain this invariant, each time FIRSTNOISE changes – either by expiring the reading out of the window, or due to setting of non-noise when its key appearance is more frequent than the threshold – we output all non-noise readings by time order until we find the next FIRSTNOISE in the window.

Therefore in this algorithm, in a nutshell, for each incoming reading and each expiring reading we incrementally update the corresponding counter for each distinct tag EPC value in the hashtable. Once the counter for value `R` is higher than `threshold`, we set all readings of R in the window to be non-noise. We immediately output the non-noise reading of value `R`

Algorithm 3 Eager_denoise (params: `window_size`, `threshold`)

```
1:  WINDOWBUFFER ← empty queue
2:  TABLE ← empty hashtable
3:  FIRSTNOISE ← null {keep earliest noise in window}
4:  loop {loop forever for next incoming reading}
5:    INCOMING ← the next reading
6:    mark INCOMING as noise
7:    if FIRSTNOISE = null then
8:      FIRSTNOISE ← INCOMING
9:    end if
10:   append INCOMING to end of WINDOWBUFFER
11:   if the counter at TABLE[INCOMING.key] does not exist then
12:     initiate TABLE[INCOMING.key] with counter 1
13:   else
14:     increment TABLE[INCOMING.key]
15:   end if
16:   EXPIRETIME ← INCOMING.timestamp - window_size
17:   SEARCHFIRST ← false
18:   while the head of WINDOWBUFFER is older than EXPIRETIME do
19:     if SEARCHFIRST = false ∧ the head reading is marked as noise then
20:       SEARCHFIRST ← true
21:       FIRSTNOISE ← null
22:     else if SEARCHFIRST = true ∧ the head reading is marked as non-noise then
23:       output the head of WINDOWBUFFER {this is a non-noise reading after the previous expired FIRSTNOISE}
24:     end if
25:     remove the head of WINDOWBUFFER
26:     decrement the counter in TABLE for the corresponding key
27:     remove the slot in TABLE for 0-counts
28:   end while
29:   COUNT ← counter value at TABLE[INCOMING.key]
30:   if COUNT ≥ threshold ∨ SEARCHFIRST = true then {If either the threshold condition is met, or we need a new FIRSTNOISE, scan the window}
31:     for each of the reading R still in WINDOWBUFFER according to time order do
32:       if COUNT ≥ threshold ∧ R.key = INCOMING.key ∧ R is marked as noise then
33:         if SEARCHFIRST = false ∧ R = FIRSTNOISE then
34:           SEARCHFIRST ← true
35:           FIRSTNOISE ← null
36:         end if
37:         mark R as non-noise
38:         if SEARCHFIRST = true ∨ R.timestamp < FIRSTNOISE.timestamp then
39:           output R {output the newly-determined non-noise reading, if either the next FIRSTNOISE is unknown, or it is earlier than the known FIRSTNOISE}
40:         end if
41:       else if R is non-noise ∧ SEARCHFIRST = true then
42:         output R {output the existing non-noise reading, only if the next FIRSTNOISE is not determined yet}
43:       else if SEARCHFIRST = true ∧ R is marked as noise then
44:         SEARCHFIRST ← false
45:         FIRSTNOISE ← R
46:         if COUNT < threshold then
47:           break the while loop
48:         end if
49:       end if
50:     end for
51:   end if
52: end loop
```

once we can determine that there are no more noise readings before this reading in the sliding window.

Complexity. Compared to *Lazy_denoise*, *Eager_denoise* performs one more operation: the maintenance of FIRSTNOISE. An extra linear search on the window is performed whenever FIRSTNOISE changes, and the search is obviously less frequent than one time per incoming reading. Therefore the bound of O(n) processing time per incoming reading still remains the same.

4 Duplicate Elimination (Merging)

When noise in the readings is eliminated, duplicate readings for the same tag have to be recognized and only the first (or the earliest) one among all duplicates should be retained. Our duplicate-elimination (or merging) algorithms take one parameter – `max_distance`. If a reading is within `max_distance` in time from the previous reading with the same key, then this reading is considered a duplicate. Otherwise, it is considered a new reading and is output.

Algorithm 4 – *baseline_merge* – performs duplicate elimination by simply keeping a sliding-window of size `max_distance`. For each incoming reading, if there exists another reading in the window with the same key, then it is considered a duplicate, otherwise it is output as a new reading.

Algorithm 4 Baseline_merge (param: *max_distance*)

```
1: WINDOWBUFFER ← empty queue {FIFO queue to hold sliding window of readings}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   EXPIRETIME ← INCOMING.timestamp - max_distance
5:   while the head of WINDOWBUFFER is older than EXPIRETIME do
6:     remove the head of WINDOWBUFFER
7:   end while
8:   go through WINDOWBUFFER to look for another reading with the same key as INCOMING
9:   if nothing is found then
10:    output INCOMING
11:  end if
12:  append INCOMING to the end of WINDOWBUFFER
13: end loop
```

Complexity In *baseline_merge*, a linear scan is performed on the full window for each incoming reading, therefore the time cost is Θ(n). The space cost is simply the window itself in a FIFO queue, at Θ(n).

Baseline_merge is intuitive and can be also easily realized in some systems that support the concept of sliding windows. For example, a SQL-based DSMS(Data Stream Management System) can code *baseline_merge* as the following continuous query, assuming a data stream schema of `Readings(key, time)`:

```
SELECT key, time
FROM Readings R1
WHERE NOT EXISTS
    ( SELECT *
      FROM Readings R2
      OVER(max_distance milliseconds PRECEDING R1)
      WHERE R2.key = key
      AND R2.time <> time)
```

Algorithm 5 Hash_merge (param: `max_distance`)

```
1: TABLE ← empty hashtable {hashtable to store the last appearance time for each key}
2: loop {loop forever for next incoming reading}
3:   INCOMING ← the next reading
4:   if INCOMING.timestamp - TABLE[INCOMING.key] > max_distance then
5:     output INCOMING
6:   end if
7:   update TABLE[INCOMING.key] to be INCOMING.timestamp
8: end loop
```

Baseline_merge carries a Θ(n) time cost per incoming reading, and a Θ(n) space cost, both of which can be further improved. In fact, it is straightforward to see that it is not necessary to keep a *max_distance* window worth of readings in order to determine whether an incoming reading is a duplicate. All that needs to be maintained is a timestamp to indicate the last time a reading with the same key as the incoming reading appears. If the distance between the incoming timestamp and the last timestamp is larger than *max_distance*, then we treat it as a new reading and output it.

Algorithm 5 uses a hashtable to keep the last appearance timestamp for each distinct key value. For each incoming reading, its timestamp is compared to the corresponding entry for this key in the hashtable, and the reading is determined to be a new tag reading if the key does not appear in the table, or the time distance is larger than `threshold`.

Complexity. Since the hashtable keeps one entry per distinct key value, the average space cost is now Θ(k), compared to Θ(n) of *base_merge*. Furthermore, the time cost per incoming reading is now reduced to O(1) for hashtable lookup, instead of a full scan of Θ(n). The cost of maintaining the hashtable is less than O(1) on-average for each incoming tuple, since not every incoming/expiring tuple will cause insertion/deletion of keys from the hashtable.

5 Performance Study

For experiments, a random RFID reading generator was created, which generates RFID tag reading according to a Poisson process. The Poisson process generates tag readings with random arrival time, while the arrival time conforms to a Poisson distribution with a chosen average tag arrival rate. Each generated tag reading repeats for 10 times, with some chosen noise level (a certain percentage of the reading are noise).

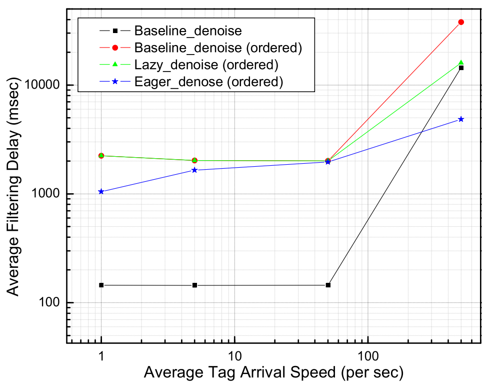


Figure 2: Noise Elimination: Delay under Different Arrival Rates

5.1 Performance of Denoising under Different Arrival Rates

In the first experiment, we study the performance of the various algorithms under different tag rates. The random generator is fixed with the following parameters: each tag reading repeats 10 times, with 200 milliseconds gap between the repeats, and a 5 percent of tag readings are noise. The average tag arrival rates tested include: 1 tag/sec, 5 tags/sec, 50 tags/sec and 500 tags/sec. (With repeats set to 10/tag, the total reading arrival rates are 10/sec, 50/sec, 500/sec and 5000/sec, respectively.) Average filtering delay over all output readings is used to measure the performance of the algorithms.

In Figure 2, four algorithms are used to filter the reading to perform denoising. *Baseline (Unordered)* corresponds to the *Baseline_denoise* algorithm presented above, which performs denoising without any optimization, and output the readings in incorrect timestamp order. *Baseline (Ordered)* is a modified version of the *Baseline_denoise* algorithm, which also performs denoising without any optimization, but outputs the readings in correct orders by outputting at the time of expiring from sliding window. *Lazy_denoise* and *Eager_denoise* are exactly as described above, and both output readings in correct time order.

All four algorithms function correctly to filter out the noise readings, and the three ordered-output algorithms also proved to maintain the correct ordering. Figure 2 shows the performance of the algorithms in terms of average delay of readings. *Baseline (Unordered)* works well with low tag rates, because it completely ignores the output time order issue and therefore has the advantage of output immediately on detection. Its performance degrades under high tag rate situations due to large overhead of linear scanning of the large sliding window under high rates. *Baseline (Ordered)* has the worst performance of all, since it has no optimization, while it still tries to maintain the timestamp ordering. *Lazy_denoise* performs better than *Baseline (Ordered)* under high loads because it utilizes hashtables to reduce the overhead. *Eager_denoise* has the best performance of all, since it not only utilizes the hashtable optimization, but also outputs readings as soon as they are safe to output. Overall, *Eager_denoise* has the best performance under all load conditions.

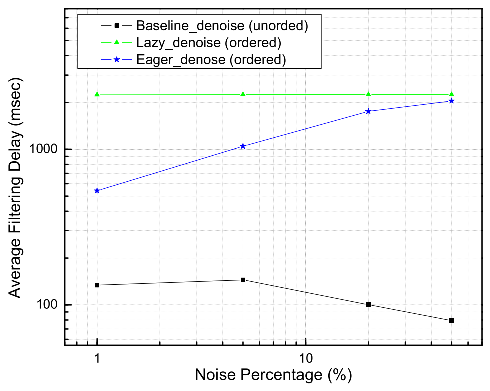


Figure 3: Noise Elimination: Delay under Different Noise Percentage

5.2 Performance of Denoising under Different Noise Ratio

The *Baseline (Unordered)*, *Lazy_denoise* and *Eager_denoise* algorithms are studied for the performance under different noise ratio. The random generator is fixed with the following parameters: each tag reading repeat 10 times, with 200 milliseconds gap between the repeats, and overall tag arrival rate is 1/second. Then different noise ratios are tested, including 1%, 5%, 20% and 50%.

Again, from Figure 3, *Baseline (Unordered)* works well in terms of performance since it ignores the ordering issue and outputs immediately upon detection, but its output readings are in incorrect time order. *Lazy_denoise* has to wait until the readings get expired from the sliding window, therefore it has the largest delay. The interesting observation is that, under low noise ratio, *Eager_denoise* works almost as well as *Baseline (Unordered)*, although it maintains the correct output time order. That is because when noise ratio is low, it is more likely for a non-noise reading to be output early under *Eager_denoise*, when there is no more noise preceding it in the sliding window. However, as noise ratio gets higher, *Eager_denoise* gets closer to *Lazy_denoise* since there are more and more noise readings present to prevent early outputting. Nonetheless, overall *Eager_denoise* always works better than *Lazy_denoise*.

5.3 Performance of Duplicate Elimination

We study the performance of the two duplicate elimination algorithms (*Baseline_merge* and *Hash_merge*) under different tag arrival rates. The random generator is fixed with the following parameters: each

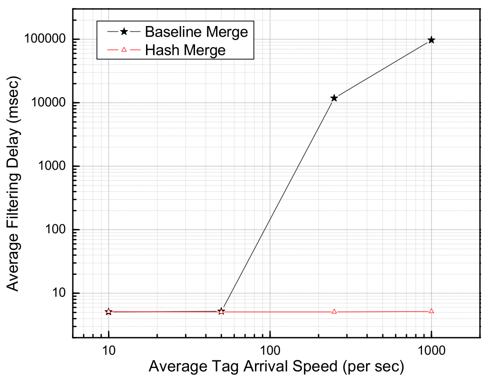


Figure 4: Duplicate Elimination: Delay under Different Arrival Rates

tag reading repeat 10 times, with 200 milliseconds gap between the repeats, and a 0 percent noise (since here we are testing duplication elimination only, noise are presumed already removed by previous filtering). Then the performance is tested under different average tag arrival rates, including 10 tags/sec, 50 tags/sec, 250 tags/sec and 1000 tags/sec. (With repeats set to 10/tag, the total reading arrival rates are 100/sec, 500/sec, 2500/sec and 10000/sec, respectively.)

Both algorithms are able to eliminate duplicate readings and only output the corresponding reading once. However, it is clear from Figure 4 that *Hash_merge* is far-superior than the baseline implementation. The delay is basically negligible even under an arrival rate of 10,000 readings/sec (1000 tags/sec) for *Hash_merge*, while *Baseline_merge* starts to cause large delays after tag rate reaches 500 readings/sec(50 tags/sec).

6 Related Work

RFID data filtering needs to remove noise and duplicate from continuous high volume RFID data streams generated from RFID readers. Such filtering is essential to provide accurate data used for RFID-enabled pervasive applications. While RFID data filtering is supported in RFID Middleware systems such as [7, 8, 9], large volume real-time RFID data streams demand more efficient approaches for filtering these data.

RFID data processing is a hybrid of event processing and stream processing. Past work on event detection and processing – such as [10, 11] – is not concerned with processing speed and memory management issues, where events are normally generated from databases and different from events from high-speed event streams. On the other hand, past work on data stream processing and continuous query optimization [12, 13, 14] assumes accurate stream sources and is not concerned with RFID application-specific issues, such as the existence of noisy and duplicate readings.

In [15], a probability-based approach is provided to detect duplicate in web click streams. This approach can not be applied to RFID data, since accuracy is among the top priority for RFID data processing.

7 Conclusion

In this paper, we identify the problem of RFID data filtering and develop efficient methods to eliminate noise and duplicate from RFID observations. Specially, for noise filtering (denoising or smoothing), we propose an approach for more efficiently maintaining the original time order of observations in the output; and for duplicate elimination, the approach that we formulate can minimize memory requirement for history buffering. We then perform experiments to validate our approaches through simulated RFID data generator and demonstrate that our approaches are effective and efficient. Our approach of data filtering is essential to provide clean and correct RFID data before they can be further processed, transformed, and integrated for RFID-enabled pervasive applications. The techniques also provide an important reference for building RFID Middleware [7, 8, 9] where filtering is a critical component.

References

[1] M. Lampe and C. Flrkemeier. The Smart Box Application Model. In *PerCom*, 2004.

[2] Siemens to Pilot RFID Bracelets for Health Care. http://www.infoworld.com/article/04/07/23/HNrfid implants_1.html, July 2004.

[3] EPC Tag Data Standards Version 1.1. Technical report, EPCGlobal Inc, April 2004.

[4] J. Brusey et. al. Reasoning About Uncertainty in Location Identification with RFID. In *RUR at IJCAI*, August 2003.

[5] H. Vogt. Efficient Object Identification with Passive RFID Tags. In *Pervasive*, 2002.

[6] F. Wang and P. Liu. Temporal Management of RFID Data. In *VLDB*, 2005.

[7] C. Bornhoevd et. al. Integrating Automatic Data Acquisition with Business Processes - Experiences with SAP's Auto-ID Infrastructure. In *VLDB*, 2004.

[8] Oracle Sensor Edge Server. http://www.oracle.com /technology/products/iaswe/edge_server.

[9] Sybase RFID Solutions. http://www.sybase.com/rfid, 2005.

[10] S. Chakravarthy et. al. Composite Events for Active Databases: Semantics, Contexts and Detection. In *VLDB*, 1994.

[11] N. H. Gehani et. al. Composite Event Specification in Active Databases: Model & Implementation. In *VLDB*, 1992.

[12] R. Motwani et. al. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[13] Sam Madden et. al. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.

[14] D. Abadi et. al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2), 2003.

[15] A.Metwally et. al. Duplicate Detection in Click Streams. In *WWW*, 2005.

Cleansing Databases of Misspelled Proper Nouns

Arturas Mazeika — arturas@inf.unibz.it
Michael H. Böhlen — boehlen@inf.unibz.it

Faculty of Computer Science, Free University of Bozen-Bolzano
Dominikanerplatz 3, I-39100 Bozen-Bolzano, Italy

Abstract

The paper presents a data cleansing technique for string databases. We propose and evaluate an algorithm that identifies a group of strings that consists of (multiple) occurrences of a correctly spelled string plus nearby misspelled strings. All strings in a group are replaced by the most frequent string of this group. Our method targets proper noun databases, including names and addresses, which are not handled by dictionaries.

At the technical level we give an efficient solution for computing the center of a group of strings and determine the border of the group. We use inverse strings together with sampling to efficiently identify and cleanse a database. The experimental evaluation shows that for proper nouns the center calculation and border detection algorithms are robust and even very small sample sizes yield good results.

1 Introduction

The high-dimensional nature of the string space puts forward a number of problems that do not exist in the numeric domain. However, besides the added complexity, strings also offer unique opportunities. In this paper we describe a solution that takes advantage of the high-dimensional space to clean databases of proper nouns, i.e., strings that do not occur in dictionaries.

Since strings are elements of a high-dimensional space the distance between any two strings is typically large. An exception are misspelled strings, which tend to be located near correctly spelled strings. The combination of these two properties means that small hyper-spheres can be used to cluster a string database. The hyper-spheres are far from each other, and each hyper-sphere encloses the correctly spelled string and the nearby misspelled strings.

CleanDB, Seoul, Korea, 2006

Figure 1 illustrates the setting for strings george, sydney, and jacob, together with misspelling of these strings. We describe a solution to group misspellings of a string by identifying the border and center of a hyper-sphere.

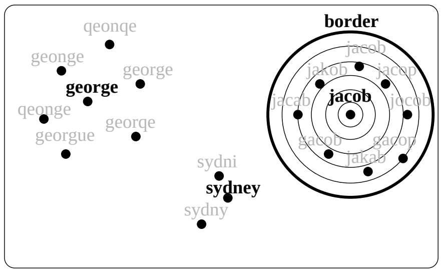


Figure 1: Database of Proper Nouns with Misspellings

The border detection algorithm is based on the *string proximity graph* (cf. Section 4.1), which captures the properties of proper noun databases with misspellings. The string proximity graph shows that in the immediate neighborhood of a string the number of strings is growing because of the misspellings. As we further increase the neighborhood the number of strings does not grow. There are no misspellings in this area and the other strings are further away because of the high-dimensional nature of the string space. The point at which the clusters stops to grow indicates the border of a group of misspelled strings.

The computation of the border and center is done in parallel. We start with a random string that has not yet been processed and identify all strings that are within distance one from this string. Next we adjust the center of the cluster and increase the radius. The adjustment of the center makes the method more robust, so that it also applies to groups of strings that are not far away from each other. As soon as an increase of the radius does not further increase the number of strings we have found a group and proceed with another string that has not yet been processed. The process stops when all the strings have been grouped.

The contributions of the paper are the following:

- We introduce a new cleansing technique for string data with typos. The solution is based on the (i) border detection and (ii) the center adjustment. The computation of the distance between strings is done with the help of q-grams of strings (substrings of length q). The center of the cluster is modeled as a bag of the most frequent substrings of length q of the strings in the group. Thus, the center reflects the substrings that are common for the strings and neglects substrings that are the result of infrequent misspelling.

- We use inverse strings (IS) to determine close-by strings and to compute the border of the cluster. Inverse strings associate with each q-gram the string IDs that contain the q-gram. Even with inverse strings the computational complexity of the border detection is combinatorial wrt the length of the center string and radius of the cluster. We use sampling to approximate the border detection. This yields a linear complexity wrt the size of the sample.

- We provide experimental result for the border detection and data cleansing algorithms. We show that the border detection is robust and that even small sample sizes ensure good approximations of the border of clusters and a low cleansing error.

The organization of the paper is the following. Section 2 presents related work. Q-grams and inverse strings are reviewed in section 3. Border detection and computation of the center string are introduced in section 4. Approximation of the border with the help of IS data structure and sampling are described in section 5. Section 6 presents the algorithm of the cleansing of the data. We give an experimental evaluation in section 7. Finally, section 8 concludes the paper and offers future work.

2 Related Work

Fuzzy retrieval is the closest related work to our approach. Fuzzy retrieval algorithms get as input a string and threshold, and output strings that are within the given threshold. Chaudhuri et al. [2] introduce an algorithm that retrieves tuples that exactly match the query string with a high probability. Jagadish et al [10] and Ciaccia et al. [4] propose a family of index structures that support exact, prefix, and approximate queries on multi-string attributes. Jin et al. [12] propose an index structure that supports mixed types (string and non-string) of attributes for approximate retrieval.

Automatic spell checking techniques [13, 9] compare a potentially misspelled word with the words in a dictionary or a model based on the dictionary. They output a correction (or a set of corrections) for a given error threshold or r number of requested answers. If r is given the dictionary (or the model) is queried a number of times for different incremental thresholds until the size r is reached. In this paper we show how to automatically compute the threshold (border of the cluster).

Efficient approximation of selectivity for a given string and edit distance (overlap threshold) is investigated in [11]. This provides important statistical information about the string data. In this paper we focus on precise computation of the center and the border of a cluster, though both our border detection and approximate selectivity solutions can be combined. Our border detection algorithm can query for approximate string selectivity, and use the result to detect border of the cluster. Then inverse strings can be used to cluster and cleanse the data.

There is a large body of work in the area of the similarity metrics for string attributes. Such measures include edit distance [8] q-grams, cosine similarity [6, 3, 7] and its variants [5, 14]. Ananthakrishna [1] proposes a textual similarity function for strings.

3 Background

3.1 Q-grams

Definition 3.1 [q-grams.] The q-grams of a string α are obtained by sliding a window of size q over the characters of α. Since at the beginning and at the end of the string we have fewer characters than q, we extend the string by prefixing it with $q-1$ occurrences of # and suffixing it with $q-1$ occurrences of \$. We assume that symbols # and \$ do not occur in the input strings.

Example 3.1 [q-grams.] Let α = george and q=2. The q-grams of string α are

$$B(\text{george}) = \{\#g^1, ge^1, eo^1, or^1, rg^1, ge^2, e\$^1\}.$$

In order to distinguish different occurrences of the same 2-gram we associate each q-gram with a sequence number (displayed as a superscript). For example, 2-gram ge^1 denotes the substring at the beginning of the string, and 2-gram ge^2 denotes the substring at the end of the string (positions 5–6 of the input string).

3.2 String Overlap

Overlaps of q-grams quantify the closeness of strings. The more two bags overlap, the closer the strings are to each other. We define the overlap of two strings as the number of q-grams they share.

Definition 3.2 [Overlap of strings α and β]. Let α and β be two strings. Then the overlap of the strings is

$$o(\alpha, \beta) = |B(\alpha) \cap B(\beta)|,$$

where $|X|$ denotes the cardinality of set X.

Our clustering strategy is based on the overlap between strings. We cluster strings together if they have a high overlap, and we assign strings to different clusters if the overlap between strings is low.

Example 3.2 [Overlap of strings.] Let α_1 = jacob, α_2 = jacop, β_1 = syndni, β_2 = syndny. Then

$$o(\text{jacob}, \text{jacop}) = |\{\#j^1, ja^1, ac^1, co^1\}| = 4$$

Since the overlap between the strings is high, we assign α_1 and α_2 to one cluster. Similarly, since $o(\text{sydny}, \text{sydni}) = 4$, β_1 and β_2 are clustered together. On the other hand, since $o(\text{sydny}, \text{jacob}) = 0$, strings $\alpha_1, \alpha_2, \beta_1, \beta_2$ are not put into one cluster.

3.3 Inverse Strings

Inverse strings associate with each q-gram κ all string IDs that contain κ as a q-gram.

Definition 3.3 [Inverse string.] Let $\alpha_1, \ldots, \alpha_n$ be a dataset and κ be a q-gram. The inverse string is the set of all strings (string IDs) that have κ as a q-gram:

$$IS(\kappa) = \{\alpha_i : \kappa \in B(\alpha_i)\}$$

Example 3.3 [Inverse string.] Let the input database consists of six strings: $\alpha_1 = jacob$, $\alpha_2 = jacop$, $\alpha_3 = jakob$, $\alpha_4 = sydny$, $\alpha_5 = sydni$, $\alpha_6 = sydney$. The bags of 2-grams for each string are:

$$\begin{aligned}
B(\alpha_1) = B(\text{jacob}) \quad &= \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\} \\
B(\alpha_2) = B(\text{jacop}) \quad &= \{\#j^1, ja^1, ac^1, co^1, op^1, p\$^1\} \\
B(\alpha_3) = B(\text{jakob}) \quad &= \{\#j^1, ja^1, ak^1, ko^1, ob^1, b\$^1\} \\
B(\alpha_4) = B(\text{sydny}) \quad &= \{\#s^1, sy^1, yd^1, dn^1, ny^1, y\$^1\} \\
B(\alpha_5) = B(\text{sydni}) \quad &= \{\#s^1, sy^1, yd^1, dn^1, ni^1, i\$^1\} \\
B(\alpha_5) = B(\text{sydney}) \quad &= \{\#s^1, sy^1, yd^1, dn^1, ne^1, ey^1, y\$^1\}
\end{aligned}$$

The inverse string structure for all 2-grams is:

$$\begin{array}{llll}
IS(\#j^1) = \{\alpha_1, \alpha_2, \alpha_3\} & IS(\#s^1) & = \{\alpha_4, \alpha_5, \alpha_6\} \\
IS(ja^1) = \{\alpha_1, \alpha_2, \alpha_3\} & IS(sy^1) & = \{\alpha_4, \alpha_5, \alpha_6\} \\
IS(ac^1) = \{\alpha_1, \alpha_2, \alpha_3\} & IS(yd^1) & = \{\alpha_4, \alpha_5, \alpha_6\} \\
IS(co^1) = \{\alpha_1, \alpha_2, \alpha_3\} & IS(dn^1) & = \{\alpha_4, \alpha_5, \alpha_6\} \\
IS(ob^1) = \{\alpha_1, \alpha_2, \alpha_3\} & IS(ny^1) & = \{\alpha_4\} \\
IS(b\$^1) = \{\alpha_1, \alpha_3\} & IS(y\$^1) & = \{\alpha_4, \alpha_6\} \\
IS(ak^1) = \{\alpha_3\} & IS(ni^1) & = \{\alpha_5\} \\
IS(p\$^1) = \{\alpha_2\} & IS(i\$^1) & = \{\alpha_5\} \\
IS(ko^1) = \{\alpha_3\} & IS(ne^1) & = \{\alpha_5\} \\
IS(op^1) = \{\alpha_2\} & IS(ey^1) & = \{\alpha_5\}
\end{array}$$

The inverse strings data structure pre-clusters strings. Intuitively, the example database consists of two clusters with data distributed around centers $\alpha_1 =$ jacob and $\alpha_5 =$ sydney. The inverse strings structure reflects the clusters: part of inverse strings consists of string IDs from the first cluster (cf. the first column), while the other parts consists of the IDs of the second cluster (cf. the second column).

4 Cluster Computation

This section presents our clustering technique. First, we formalize the computation of the border b for each cluster (cf. Section 4.1). Second, we formalize the computation of center ζ of the cluster (cf. Section 4.2).

4.1 Border Detection

Assume a center string ζ of a cluster. The border detection algorithm aims to find the smallest radius that separates strings of this cluster from strings of other clusters. Since we compare strings with the help of overlaps, this border is the smallest overlap o that separates the cluster from other cluster.

The border is computed by examining $|C_d(\zeta)| = |\{\alpha : o(\alpha, \zeta) \geq d\}|$, i.e., the number of strings that have an overlap of at least d with ζ. Consider the following example.

Example 4.1 [Border detection.] We continue examle 3.3. Let $\zeta =$ jacob, q=2. We compute the database strings that have all 2-grams in common (overlap is $o = 6$) with jacob: $C_6(\text{jacob}) = \{\alpha_1\}$, the database strings that have all but one 2-gram: $C_5(\text{jacob}) = \{\alpha_1\}$. Similarly:

$$\begin{array}{lcl}
C_4(\text{jacob}) & = & \{\alpha_1, \alpha_2, \alpha_3\} \\
C_3(\text{jacob}) & = & \{\alpha_1, \alpha_2, \alpha_3\} \\
C_2(\text{jacob}) & = & \{\alpha_1, \alpha_2, \alpha_3\} \\
C_1(\text{jacob}) & = & \{\alpha_1, \alpha_2, \alpha_3\} \\
C_0(\text{jacob}) & = & \{\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6\}.
\end{array}$$

Figure 2 shows the size of $|C_d(\text{jacob})|$ as overlap o decreases (cf. Axis X from right to left). For large overlaps ($o = 5{-}6$) the size of the cluster increases. Then the cluster size stops to increase for a range of the overlaps ($o = 1{-}4$). This is an indication that the border of the cluster has been reached. As the overlap is further decreased the cluster starts to include points from other clusters resulting in a very fast increase of its size ($o = 0 - 1$).

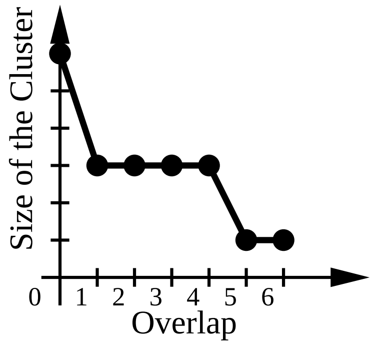


Figure 2: The String Proximity Graph

We compute the largest range of a constant size of the cluster (cf. $o = 1 - 4$ in Figure 2), and take the right end of the interval as the border.

The border detection algorithm takes a center string ζ and finds the border b of the cluster. We extend the notion of border detection for a bag of q-grams. Let $q = 2$. Then the following expressions are equivalent:

(i) b is the border for center string $\zeta =$ jacob

(ii) b is the border for the 2-grams $B(\text{jacob}) = \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\}$.

The extension of the border detection allows us to query for borders of centers that do not necessarily correspond to a database string (for example for a bag $\{\#j^1, ja^1, a\mathbf{X}^1, co^1, ob^1, b\$^1\}$). The motivation for this generalization comes from the computation of the center for a cluster and is discussed in detail in Section 4.2.

The following summarizes and defines the detection of the border.

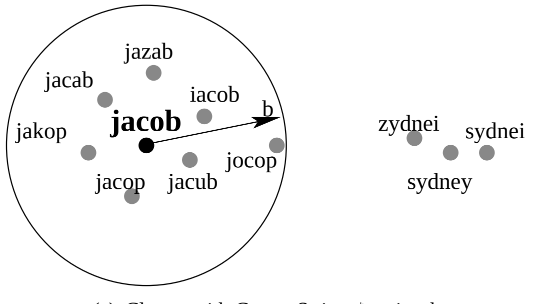
Definition 4.1 [Detection of the Border.] Let B be a (center) bag and $C_d(B) = \{\alpha : o(B(\alpha), B) \geq o\}$, $o = |B|, |B| - 1, |B| - 2, \ldots, 0$. Let $i_j, i_j + 1, \ldots, i_j + k_{i_j}$ the longest sequence of unvarying sizes of the cluster:

$$|C_{i_j}(B)| = |C_{i_j+1}(B)| = \cdots = |C_{i_j+k_{i_j}}(B)|.$$

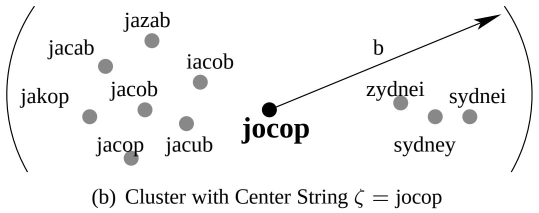
Then the border of the cluster with center B is $b = i_j$.

4.2 Computation of the Center

The border detection algorithm provides a simple and effective strategy to compute clusters in string data. One starts with a string in the database and selects the border that separates the cluster from the other clusters. If the initial string was chosen close to the center of the cluster, the border detection will yield good and robust results (cf. ζ = jacob, Figure 3(a)). If one chooses the initial string close to the border, two separate clusters might be assigned to one cluster (cf. ζ = jocop, Figure 3(a)).



(a) Cluster with Center String ζ = jacob



(b) Cluster with Center String ζ = jocop

Figure 3: Border Detection for Different Center Strings

The computation of the exact center for a given bag of strings B is expensive. One needs to compute distances between all strings in B and choose the one that minimizes the sum of distances from the center to other strings from B. We transform all strings into the space of bags of q-grams, and find the center bag there. The following example illustrates the computation.

Example 4.2 [Computation of the center for a given set of bags.] We continue Example 4.1. Let α_1, α_2, and α_3 be a set of strings. Then the set of bags for the strings is the following:

$$\begin{array}{lll} B(\alpha_1) & = B(\text{jacob}) = & \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\} \\ B(\alpha_2) & = B(\text{jakob}) = & \{\#j^1, ja^1, ak^1, ko^1, ob^1, b\$^1\} \\ B(\alpha_3) & = B(\text{jacap}) = & \{\#j^1, ja^1, ac^1, ca^1, ap^1, p\$^1\}, \end{array}$$

Our aim is to find a bag that represents bags $B(\alpha_1)$, $B(\alpha_2)$, and $B(\alpha_3)$. We compute such a bag in the following way. We compute the overall histogram for the set of bags, and neglect the infrequent 2-grams. The histogram of all 2-grams is presented in Figure 4 with the 2-grams in the second row, and the number of occurrences of the 2-gram in the first row.

3	3	3	3	2	1
$\#j^1$	ja^1	ac^1	ob^1	$b\$^1$	co^1

1	1	1	1	1
ak^1	ko^1	ca^1	ap^1	$p\$^1$

Figure 4: Histogram of 2-grams

The size of the center bag of 2-grams is determined by the average size S of the input bags $B(\alpha_1)$, $B(\alpha_2)$, and $B(\alpha_3)$. Therefore, the center bag is the following:

$$B^c = \{\#j^1, ja^1, ac^1, ob^1, b\$^1, co^1\}.$$

Note that the center bag might consist q-grams that correspond to typos in the input dataset. These occurrences do not decrease the quality of clustering. In fact, the opposite holds, since we are looking for a center bag that represents all the strings in the cluster as precisely as possible.

The following formalizes the computation of the center bag for a set of input bags.

Definition 4.2 [Center bag.] Let $B_1, B_2, \ldots, B_k$ be a set of input bags. Let

$$A = \frac{|B_1| + |B_2| + \cdots + |B_k|}{k}$$

be the average size of bags $B_1, B_2, \ldots, B_k$. Let

$$h(\kappa) = |\{B_i : \kappa \in B_i\}|$$

be the histogram value of q-gram κ. Let $\kappa_1, \kappa_2, \ldots, \kappa_m$ be an ordered sequence of q-grams of $B_1 \cup B_2 \cup \cdots \cup B_k$ such that $h(\kappa_i) \geq h(\kappa_{i+1})$. Then the center bag B is the set of q-grams:

$$B = \{\kappa_1, \kappa_2, \ldots, \kappa_A\}.$$

5 Sampling of Inverse Strings

In this section we show how to use inverse strings to identify strings that have an overlap with the center string above a given threshold. First, we develop a mathematical formula that shows how to identify strings of high overlap. The result has combinatorial complexity. Second, we approximate the computation of high overlap strings with a help of sampling.

The IS data structure allows to quickly identify database strings that have selected q-grams in common. For example, if one wants to find all string IDs that share all 2-grams with the string jacob, one needs to compute the following expression:

$$IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1)$$

Similarly, if one wants to identify strings that contain all but one 2-gram of jacob, one needs to compute the following:

$$\begin{array}{l} \; IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\ IS(\#j^1) \; \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\ IS(\#j^1) \cap IS(ja^1) \; \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\ IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \; \cap IS(ob^1) \cap IS(b\$^1) \bigcup \\ IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \; \cap IS(b\$^1) \bigcup \\ IS(\#j^1) \cap IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \end{array}$$

Definition 5.1 [Computation of strings of high overlap with the help of the IS data structure.] Let B be a center bag, such that $\kappa_1, \kappa_2, \ldots, \kappa_o \in B$, and o be the overlap threshold. Let

$$O(\kappa_1, \ldots, \kappa_o) = IS(\kappa_1) \cap IS(\kappa_2) \cap \cdots \cap IS(\kappa_o). \quad (1)$$

The IDs of strings that have at least o q-grams from B can be computed with the following equation:

$$\bigcup_{\kappa_1, \kappa_2, \ldots, \kappa_o \in B} O(\kappa_1, \ldots, \kappa_o) \quad (2)$$

where $\kappa_1, \kappa_2, \ldots, \kappa_o$ are different q-grams of B.

The computation of the strings of high overlap with the help of the IS data structure is expensive. Let $|B|$ be the size of the bag of q-grams, and o be the desired overlap threshold. Then the computational complexity of the computation is $o \cdot \binom{|B|}{o}$ number of set operations (cf. equation (2)). We approximate the computation of equation (2) with the help of sampling. We select a small sample of different o-tuples $(\kappa_1^i, \kappa_2^i, \ldots, \kappa_o^i)$, $i = 1, 2, \ldots, S$, where S is the size of the sample, and compute the union of the intersections:

$$\bigcup_{i=1}^{S} IS(\kappa_1^i) \cap IS(\kappa_2^i) \cap \cdots \cap IS(\kappa_o^i) \quad (3)$$

Example 5.1 [Computation of strings of high overlap with the help of the IS data structure and sampling.] We continue example 4.2. Let the center bag be $B = \{\#j^1, ja^1, ac^1, co^1, ob^1, b\$^1\}$ (the bag of string jacob). Let the overlap threshold be $o = 5$ (all 2-grams except one) and let the sample size be $S = 3$.

The computation of approximated strings is done in three steps. First, we generate $S = 3$ random 5-tuples from B:

$$\begin{aligned} \kappa^1 &= (\kappa_1^1, \ldots, \kappa_5^1) = (ja^1, ac^1, co^1, ob^1, b\$^1) \\ \kappa^2 &= (\kappa_1^2, \ldots, \kappa_5^2) = (\#j^1, ac^1, co^1, ob^1, b\$^1) \\ \kappa^3 &= (\kappa_1^3, \ldots, \kappa_5^3) = (\#j^1, ja^1, ac^1, co^1, ob^1) \end{aligned}$$

Second, we compute the intersections for the 5-tuples:

$$\begin{aligned} U_1(\kappa^1) &= IS(ja^1) \cap IS(ac^1) \cap IS(co^1) \cap IS(ob^1) \cap IS(b\$^1) \\ &= \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \\ &\quad \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1, \alpha_2, \alpha_3\} \cap \{\alpha_1\} \\ &= \{\alpha_1\}. \end{aligned}$$

Similarly, $U_1(\kappa^2) = \{\alpha_1\}$ and $U_1(\kappa^3) = \{\alpha_1\}$. Finally, we compute the union:

$$U(\kappa^1) \cup U(\kappa^2) \cup U(\kappa^3) = \{\alpha_1\}.$$

Therefore, the approximate database strings with overlap $o = 5$ and higher to the center string jacob are $\{\alpha_1\}$.

6 Algorithm

This section presents the algorithm of our data cleansing method. The algorithm cleanses data in 4 steps. First the algorithm initializes the variables (cf. block 1, Figure 5), then it clusters the string data (cf. block 2), merges overlapping clusters (cf. block 3), and finally it replaces the strings of a cluster with the most frequent string of the cluster (cf. block 4).

```
Input:
  $D = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$:database of strings
  $q$:size of q-grams
  $S$:sample size

Output:
  $\alpha_1, \alpha_2, \ldots, \alpha_n$:cleansed strings

Body:
  1. Initialize the clustered strings
     Clustered_Strings=$\emptyset$, Clusters = $\emptyset$
  2. Scan database strings. For each $\alpha \in D$ do
     2.1 If $\alpha \in$ Clustered_Strings then start a new iteration with the
         next DB string (go to step 2). Otherwise compute initial
         center bag:$B = B(\alpha)$, max border: $b_m = |B|$. Initialize
         the current cluster $O = \emptyset$
     2.2 For each overlap threshold $o = b_m - 1, \ldots, 1$ do
         2.2.1 Compute approximate strings with center bag $B$
               and overlap threshold $o$. For $i = 1, 2, \ldots, S$
               2.2.1.1 Generate $\kappa_1, \ldots, \kappa_o$ o-tuple of q-grams
               2.2.1.2 Compute the overlap strings
                       $O = O \cup O(\kappa_1, \ldots, \kappa_o)$ (cf. Eq (1))
         2.2.2 Update the center of the cluster.
               2.2.2.1 For each $\alpha \in O$, for each $\kappa \in B(\alpha)$ do
                       update histogram $h[\kappa] \leftarrow h[\kappa] + 1$
               2.2.2.2 Sort $h[\kappa]$ in descdending order
               2.2.2.3 Compute the average length of the strings
                       $A = \sum_{\alpha \in B} len(\alpha)/|B|$
               2.2.2.4 Assign the top $A$ q-grams of the histogram
                       to the center bag $B$
        2.2.3 Record the cluster for overlap $o$:
              2.2.3.1 Cluster[o] = $B$
    2.3 Find the longest sequence $i_b, i_b + 1, \ldots, i_b + \Delta$
        such that |Cluster[$i_b$]| = $\cdots$ = |Cluster[$i_b + \Delta$]|
    2.4 Update the clustered strings
        Clustered_Strings = Clustered_Strings $\cup$ Cluster[$i_b$]
    2.5 Insert a new cluster to the set of clusters
        Clusters = Clusters $\cup$ {Cluster[$i_b$]}
    2.6 Empty $h[\kappa]$, $O$, $B$
  3. Merge overlapping clusters. For each $C_i, C_j \in$ Clusters do
     if $C_i \cap C_j \neq \emptyset$ then $C_i \leftarrow C_i \cup C_j$
  4. Clean the clusters. For each cluster $C_i \in$ Clusters do
     4.1 Find the most frequent string $\varphi$ in $C_i$.
         Replace all strings $\alpha \in C_i$ with $\varphi$.
```

Figure 5: Data Cleansing Algorithm

Block 2 (cf. Figure 5) clusters the string data. It starts with a non clustered string α (block 2.1) and computes string IDs that have overlap with the center string (cf. Figure 2) for different overlap thresholds. For each overlap o the algorithm computes the strings of high overlap (block 2.2.1), and adjusts the center bag of the cluster (cf. block 2.2.2, Section 4.2). Then the method detects the border of the cluster (block 2.3), inserts the newly found cluster (block 2.4), and removes the IDs of clustered strings from the database (block 2.5). Four data containers are used to implement the clustering step: histogram of q-grams for the current cluster $h[\kappa]$ (cf. definition 4.2), center bag B, set of strings that have overlap o and higher wrt the center bag B (the container increases as o decreases), and set of strings for each overlap threshold o (the container is not affected by the increase of o). All containers are main memory data structures and are implemented as sorted associated containers for fast point-queries.

Block 3 merges overlapping clusters and block 4 cleanses clusters with the most frequent string of the cluster (the reasoning is that most of the strings are entered correctly, and the data consists only of a smaller number of strings with typos). Alternatively, one can identify the string ζ that shares the largest number of q-grams with the center bag, and use string ζ as the correct string for cleansing.

The intersection of inverse strings $IS(\kappa_1)\cap\cdots\cap IS(\kappa_d)$ (Block 2.2.1.2) is the most expensive part of the algorithm. We implemented and tested four different approaches of the computations of the intersection. Let $\kappa_1, \kappa_2, \ldots, \kappa_o$ be a sequence of the q-grams of a center string (in some random order). Then the implemented strategies are the following:

(i) Scan all inverse strings simultaneously, i.e., let $i = (i(\kappa_1), i(\kappa_2), \ldots, i(\kappa_o))$ be an index vector that scans $(IS(\kappa_1), IS(\kappa_2), \ldots, IS(\kappa_o))$. If all the components of index i point to the same string ID, then the cluster size is incremented, and all components of i are incremented. Otherwise, only index $i(\kappa_i)$ is incremented, if $IS(\kappa_i)$ contains the smallest string ID. Note that we require that inverse strings are ordered according to the string ID.

(ii) Organize the computation of the intersection as a sequence of intersections of two inverse strings, for e.g.:

$$((IS(\kappa_1)\cap IS(\kappa_2))\cap IS(\kappa_3))\cap IS(\kappa_4)$$

The strategy can be formalized in the following way. Let $IN_{i+1} = IN_i \cap IS(\kappa_{i+1})$, $i = 2, \ldots, o$, $IN_1 = IS(\kappa_1)$, then

$$IS(\kappa_1)\cap IS(\kappa_2)\cap\cdots\cap IS(\kappa_o) = IN_o(\kappa_o)$$

(iii) The same strategy as (ii) though the sequence is sorted started with the smallest inverse string, i.e., $|IS(\kappa_i)| \leq |IS(\kappa_{i+1})|$.

(iv) Similar strategy to (ii), though intersections are organized into a bushy tree:

$$\Big(\big(IS(\kappa_1)\cap IS(\kappa_2)\big)\cap\big(IS(\kappa_3)\cap IS(\kappa_4)\big)\Big)$$

The following recurrent equations formalizes the computation:

$$\begin{aligned} IN_i^0 &\leftarrow I(\kappa_i) \\ IN_{i+1}^j &\leftarrow IN_{2i-1}^{j-1} \cap IN_{2i}^{j-1} \\ IN_{\lfloor o/2^j \rfloor}^j &\leftarrow IN_{\lfloor o/2^j \rfloor}^j \cap IN_{\lfloor o/2^{j-1} \rfloor}^{j-1} \text{ iff } 2^j \not| o \end{aligned}$$

where $i = 1, 2, \ldots, \lfloor o/2^j \rfloor$, $j = 1, \ldots, \log_2 o$. Then the intersection can be rewritten as follows:

$$IS(\kappa_1)\cap\cdots\cap IS(\kappa_o) = IN_1^{\log_2 o}.$$

The results on different datasets has showed that strategy (ii) outperformed the other strategies by at least 30%. Therefore, we used strategy (ii) in our experiments. However, other alternatives might be more beneficial for distributed environment and in connection with caching techniques (cf. strategy (iv)).

7 Experiments

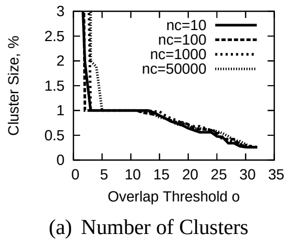
We organize the experiments in two sub-sections. First, we evaluate border detection criteria (cf. Section 7.1) and then we evaluate our cleansing method (cf. Section 7.2). We use synthetic datasets with different parameters in our experiments. Three classes of databases were generated in the experiments: (i) a class of databases with different number of clusters (nc), (ii) a class of databases with different cluster sizes (cs), and (iii) a class of databases with different radius of clusters ($radius$). All datasets were generated in the following way. First we generated nc number of center strings far away from each other. Then for each center string we generated cs number of strings in e edit distance[1] from the center string, where $0 \leq e \leq radius$.

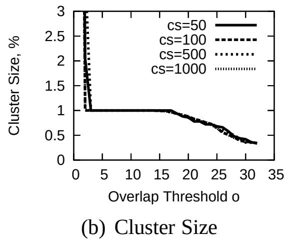
7.1 Border Detection

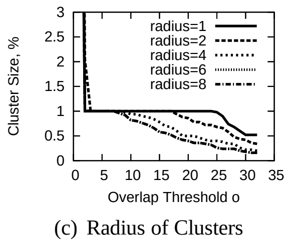
Figure 6 shows the experiments for our border detection algorithm for different number of clusters (cf. Figure 6(a)), cluster sizes (cf. Figure 6(b)), radius of the cluster (cf. Figure 6(c)), and sample size (cf. Figure 6(d)). All figures varies overlap from around $o = |B| = 35$ to $o = 1$ (cf. Axis X from right to left in Figure 6). Y axis reports the fraction of the size of the cluster that is covered by the overlap threshold o. There are three intervals of overlaps in the graphs: an interval $I_<$ of overlaps o that does not cover the entire cluster (cf. interval 35–17, Figure 6(b)), interval $I_=$ of overlaps that cover exactly the cluster (cf. rage 16–4, Figure 6(b)), and interval $I_>$ of overlaps that

[1]edit distance between string α and string β is the smallest number of character- insertions, deletions, and substitutions required in order to get string α from string β.

cover more strings than there are in the cluster (cf. range 3–0, Figure 6(b)). The border detection works if there is a (relatively long) interval of overlaps that covers the cluster exactly.


(a) Number of Clusters


(b) Cluster Size


(c) Radius of Clusters

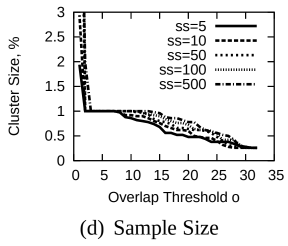

(d) Sample Size

Figure 6: Border Detection

Border detection algorithm successfully identifies borders of clusters provided a sufficient sample size.

The robustness of the algorithm is not affected by the cluster size (cf. Figure 6(b)). Indeed, the length of interval $I_=$ depends on the distance between the borders of the clusters and does not depend on the cluster size.

The robustness of the border detection is almost invariant to the number of clusters (cf. Figure 6(a)). As the number of clusters increases from 10 to $50,000$ the start of interval $I_=$ shifts from 16 to 13. However, the impact of the shift is negligible compared to the length of $I_=$, and therefore the border detection ensures robust results.

Radius of clusters (cf. Figure 6(c)) and sample size (cf. Figure 6(d)) impacts more significantly the robustness of border detection. The length of $I_=$ proportionally decreases as the radius decreases (by two for each decrease in radius). Decrease of the sample size lowers the shape of the curve, decreases the length of $I_=$, and in turn decreases the robustness of the border detection. However, we want to have the sample size as small as possible, since the smaller sample size means a lower computational time of data cleansing.

Figure 6(d) confirms that very small samples can be used to approximate the border detection robustly (cf. $ss = 10$ with the total number $\binom{35}{17} \approx 4.5 \times 10^9$ of intersection computations (cf. equation (1)) for the overlap threshold $o = 17!$)

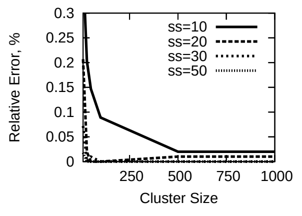
The default parameters in the series of experiments were: length of strings $l \approx 30$, number of cluster $nc = 100$, cluster size $cs = 50$, sample size $ss = 100$, cluster radius $radius = 3$.

7.2 Cleansing

We evaluate our cleansing algorithm for different cluster sizes (cf. sub-section 7.2.1) and different number of clusters (cf. sub-section 7.2.2). Two measurement are recorded for the experiments: relative error (recorded in relative number of misclustered strings compared to the total number of strings in the clusters) and clustering time (seconds).

7.2.1 Different Cluster Sizes

As the cluster size increases, the relative clustering error decreases (cf. Figure 7(a)). This is because the border detection algorithm is very effective, and the number of correctly clustered strings increases vs. the total number of strings in the cluster.


(a) Relative Error

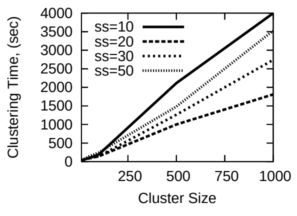

(b) Clustering Time

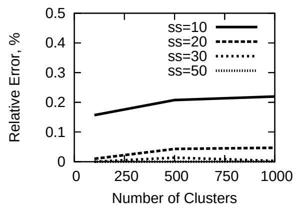
Figure 7: Different Cluster Sizes

The clustering time increases linearly as the number of strings per cluster increases (cf. Figure 7(b)). However a lower sample size does not necessarily mean a faster clustering time. This is because inadequately small sample size increases the number of total clusters, and in turn increases the number of iterations of the algorithm.

The default parameters in this series of experiments were: length of strings $l \approx 30$, number of cluster $nc = 100$, cluster radius $radius = 3$.

7.2.2 Different Number of Clusters

The relative error increases very slightly as the number of strings increases, (cf. Figure 8(a)). This is because the sharp borders between the inverse strings of different clusters gets blurred as the number of clusters increases. Note that our sampling technique is very effective: even a very small increase of the sample size (cf. $ss = 10$ and $ss = 20$) significantly reduces the relative error.


(a) Relative Error

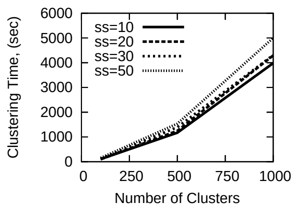
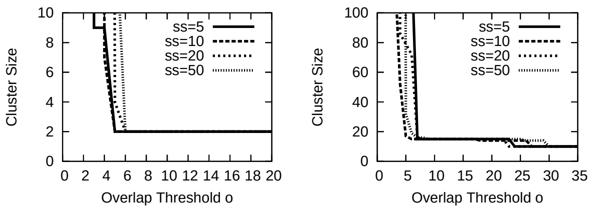

(b) Clustering Time

Figure 8: Different Cluster Sizes

The clustering time (cf. Figure 8(b)) increases linearly as the number of clusters increases. In contrast to the cluster size experiment (cf. Section 7.2.1) the clustering time for smaller samples does not exceed the clustering time for larger samples, since the number of clusters is very large compared to the size of the clusters.

7.3 Real World Data

This section evaluates the border detection algorithm for real world company names (database with around 15 character long strings) and company addresses (database with around 30 character long strings). Both databases consists of clusters that are far away from each other and a small number of strings within the clusters (cf. Figure 9). There is a large range of overlap levels for which the cluster size is constant (cf. o=[20–7] for the company names and o=[22–7] for the company addresses), and therefore our border detection algorithm detects the border correctly even for very small sample sizes. Our clustering algorithm detected small clusters (1–3 strings per cluster) for the company names and larger clusters (3–30 strings per cluster) for company addresses.



(a) Company Names (b) Company Addresses

Figure 9: String Proximity Graph for Real World Data

Intuitively, our data cleansing algorithm produces good cleansing results for string data with large distances between centers of clusters and small distances within the clusters. Examples of such datasets are databases of company names and company addresses. Our data cleansing algorithm is less applicable for natural language databases. In such databases two strings that are close to each other might have a very different meaning and therefore should be assigned to different clusters (for example “air” and “aim”, or “spouse” and “mouse”). In natural language databases spelling based and dictionary based techniques are more appropriate. For proper noun databases typically no dictionaries exists and the proposed solution is the preferred choice.

8 Conclusions and Future Work

In this paper we present our results of a new data cleansing algorithm. Data cleansing is done in two steps. First, the string data is clustered by identifying center and border of hyper-spherical clusters, and second, the cluster strings are cleansed with the most frequent string of the cluster. Clustering starts with a non-clustered string and computes the border b of the cluster. All strings within the overlap threshold b from the center of the cluster are assigned to one cluster. Experiments show that the border detection is robust provided a sufficient sample size.

There are a number of research directions for future work. One can further progress the IS data structure. Our investigation indicates that very few q-grams of the center strings are sufficient to identify strings of the cluster. An algorithm that robustly finds the identifying q-grams of the cluster is an interesting challenge.

The data cleansing method is robust if the distance between the clusters is large compared to the diameters of the clusters. In order to improve the precision for databases with small distances between the clusters one can introduce a number of string representatives for each cluster.

References

[1] R. Ananthakrishna, S. Chaudhuri, and V. Ganti. Eliminating fuzzy duplicates in data warehouses. In *VLDB*, pages 586–597, 2002.

[2] S. Chaudhuri, K. Ganjam, V. Ganti, and R .Motwani. Robust and efficient fuzzy match for online data cleaning. In *SIGMOD*, pages 313–324, 2003.

[3] S. Chaudhuri, V. Ganti, and L. Gravano. Selectivity estimation for string predicates: Overcoming the underestimation problem. In *ICDE:*, pages 227–239, 2004.

[4] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *VLDB*, pages 426–435, 1997.

[5] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. In *Data Cleaning Workshop in Conjunction with KDD*, 2003.

[6] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500, 2001.

[7] L. Gravano, P. G. Ipeirotis, N. Koudas, and D. Srivastava. Text joins in an rdbms for web data integration. In *WWW*, pages 90–101, 2003.

[8] D. Gusfield. *Algorithms on strings, trees and sequences: Computer science and computational biology*. Cambridge University Press, Cambridge, UK, 1997.

[9] V. J. Hodge and J. Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *TKDE*, 15(5):1073–1081, 2003.

[10] H. V. Jagadish, N. Koudas, and D. Srivastava. On effective multi-dimensional indexing for strings. In *SIGMOD*, pages 403–414, 2000.

[11] L. Jin and C Li. Selectivity estimation for fuzzy string predicates in large data sets. In *VLDB*, pages 397–408, 2005.

[12] L. Jin, C. Li, N. Koudas, and A. K. H. Tung. Indexing mixed types for approximate retrieval. In *VLDB*, pages 793–804, 2005.

[13] K. Kukich. Technique for automatically correcting words in text. *ACM Comput. Surv.*, 24(4):377–439, 1992.

[14] S. Sahinalp, M. Tasan, J. Macker, and Z. Ozsoyoglu. Distance based indexing for string proximity search. In *ICDE*, 2003.