# Trio-ER: The Trio System as a Workbench for Entity-Resolution

Parag Agrawal, Robert Ikeda, Hyunjung Park, and Jennifer Widom
{paraga,rmikeda,hyunjung,widom}@cs.stanford.edu
Stanford University

## I. Introduction

*Entity-resolution* (also known as *deduplication*, *record linkage*, and *reference reconciliation*, among others) was one of the original motivating applications [6] for the *Trio* system, which has been under development at Stanford over the past several years.

- Entity-resolution is the process of determining when multiple data records are likely to represent the same real-world entity, and possibly merging such records [5].
- Trio is a research prototype DBMS for managing data that includes *uncertainty*, and for tracking *lineage* automatically as queries are performed [2].

Uncertainty is an important component of entity-resolution: match functions may return confidence values, and merged records may retain alternative possible values for attributes. Lineage also is important, since it tracks the original records contributing to a merged result—useful for debugging, and for entity-resolution algorithms that may "unmerge" records. In Trio, lineage also enables computing confidence values for merged records, and it supports useful processing steps in iterative entity-resolution (Section IV-E).

*Trio-ER* is a new variant of the Trio system tailored as a workbench for entity-resolution. The goal of the first version of Trio-ER has been to enable rapid prototyping of an important basic class of entity-resolution algorithms using Trio.

As an example of rapid prototyping in Trio-ER, consider a table `Hotel(URL,name,zip)` containing hotel information. (One of the real data sets we have been using is hotel information from *Yahoo! Travel*.) Assume we have a string comparison function `StrComp`($s_1$,$s_2$) that returns a similarity value in $[0, 1]$. Suppose we believe two hotel records represent the same hotel if their `zip` is identical, and they have more than a 0.95 string-match on either `URL` or `name`. When records are merged, all variants of `URL` and `name` are retained. The following query, expressed in Trio's SQL extension called *TriQL* [1], performs one iteration of pairwise matching and merging:

```
Select Combine *
From Hotel h1 ERJ Hotel h2
On Exists [h1.zip = h2.zip
     And (StrComp(h1.URL,h2.URL) > 0.95
     Or StrComp(h1.name,h2.name) > 0.95)]
```

Placed in a loop, this query will match and merge until there are no more matches and entity-resolution is complete. New features beyond standard TriQL are the `Combine` modifier, which creates alternative values for shared attributes, and `ERJ`, a variant of SQL outerjoin. `StrComp()` is a foreign function, and the square brackets in the join condition denote a TriQL *horizontal subquery*.

In the remainder of this description, we:

- Provide additional background on Trio (Section II).
- Discuss the scope of the current Trio-ER system in terms of entity-resolution capabilities (Section III).
- Explain how existing Trio features are used for entity-resolution, and what customizations were added for the first version of Trio-ER (Section IV); several examples are included.

## II. Background

The Trio system is based on the *ULDB* data model [4]. ULDBs extend the relational model with:

- *Alternatives:* ULDB relations are comprised of *x-tuples*. Each x-tuple contains one or more *alternatives*. Each alternative is a regular tuple over the schema of the relation, denoting one possible value for the tuple.
- *Confidences:* Numerical confidence values may be attached to alternatives in an x-tuple. Under the default probabilistic interpretation, the confidence of an alternative represents the probability of the tuple taking the value of that alternative.
- *Lineage:* In Trio, lineage is created automatically on query results, connecting each alternative in the result to the alternatives from which it was derived. In the general case, lineage may take the form of any Boolean formula.

(ULDBs may also contain "?" annotations, indicating uncertainty about the presence of tuples in relations, but ?'s have not been needed yet in Trio-ER.)

Trio's query language, TriQL, extends SQL with features to query and manipulate alternatives, confidence values, and lineage. The Trio system is implemented as a layer on top of a conventional relational DBMS: ULDBs are encoded in relational tables, and TriQL queries and commands are translated to SQL queries and procedures executed against the encoding. For details, see [2].

## III. Scope

The first version of Trio-ER targets entity-resolution algorithms based on *local-pairwise* matching. In this setting, entity-resolution typically involves an iterative process of matching records based on exhaustive pairwise comparisons (with *blocking* techniques for efficiency; see Section IV-F), merging matched records, and then continuing the process looking for additional matches. A number of entity-resolution algorithms fall into this category [3], [5]. Subsequent versions of Trio-ER will expand the types of entity-resolution supported, including:

- *Ranking* algorithms that perform entity-resolution using global best-matches, but still based on pairwise comparisons. It should not be difficult to extend Trio-ER to support such algorithms.
- *Clustering* algorithms that group records into entities based on proximity measures. Clustering approaches map less cleanly to Trio's SQL-based query processing paradigm, so they are likely to require more substantial extensions to the system.

## IV. TRIO MODIFICATIONS

### A. From X-Tuples to Or-Tuples

Recall that Trio x-tuples are comprised of alternative tuple values. For example, for a table `Person(name, street, city, photo)` we might have an x-tuple with two alternatives:

| |
|---|
| (Jim, Maple, PA, p1.jpg) ‖ (Tim, Birch, MP, p2.jpg) |

In entity-resolution it is common to retain independent alternative values for individual attributes or groups of attributes. For example, a tuple might have `name` as either 'Jim' or 'Tim' and `photo` as either 'p1.jpg' or 'p2.jpg.' To avoid the multiplicative effect that occurs when representing independent alternative attribute values as x-tuples, in Trio-ER we support a generalization of x-tuples called *or-tuples*. In or-tuples, each attribute or designated *attribute group* contains its own independent alternative values.

For example, we can define our table as `Person(name, [street, city], photo)`, denoting independent alternative values for `name`, for `street-city` pairs, and for `photo`. Suppose we have a person named 'Jim' or 'Tim,' whose address is either 'Maple' street in Palo Alto ('PA') or 'Birch' street in Menlo Park ('MP'), and whose photo is in 'p1.jpg' or 'p2.jpg.' The or-tuple representation is:

| |
|---|
| Jim ‖ Tim, [Maple, PA] ‖ [Birch, MP], p1.jpg ‖ p2.jpg |

Note that representing this or-tuple as an x-tuple would require eight alternatives.

Bracketed attribute groups define the granularity for alternative values, at the schema level. The two extremes are all attributes independent (no brackets), or all attributes bracketed in one group (the original x-tuples model). Given a TriQL query with or-tuple input tables, we infer the independent attribute groups in the query result; doing so is easy for most queries, but in some cases we are conservative.

We have completed a new Trio that has efficient or-tuple support throughout the system. In the remainder of this proposal, "tuple" is synonymous with or-tuple unless specified otherwise.

### B. Language Constructs

The general form of TriQL query to perform one round of exhaustive pairwise matching and merging on a single table `R` is:

```
Select Combine *
From R as r1 ERJ R as r2 On <Match-Cond>
Where <Filter>
```

The following subsections explain the new (and some old) TriQL constructs used in this query, then how the query is used as part of an iterative entity-resolution process.

*1)* `ERJ` *Join Operator:* Pairwise entity-resolution on a single table obviously requires a "self-join" of some type, however a regular TriQL self-join is not appropriate for several reasons: we don't want to match tuples with themselves, we don't want to pair $t_1$ with $t_2$ as well as $t_2$ with $t_1$, and we want outerjoin-like semantics for tuples with no matches. To achieve this behavior, we added a new join operator, `ERJ` (for *Entity-Resolution Join*), to TriQL. ERJ was not too difficult to implement in Trio's translation-based scheme, and it supports both self-joins and regular binary joins. We define their semantics in turn.

**Self-ERJ.** For "`R as r1 ERJ R as r2 On <Cond>`", the result contains all tuple-pairs $(t_1, t_2)$ such that:

- $t_1$ and $t_2$ are distinct tuples in `R`, and $t_1$'s tuple identifier precedes $t_2$'s (so pairs are not included twice)
- $t_1$ and $t_2$ satisfy `<Cond>` when substituted for `r1` and `r2`

In addition:

- For every $t_1$ such that there is no $t_2$ satisfying `<Cond>`, $(t_1, t_1)$ is included

As an example, consider the following `Person` table, with no alternative values for now:

| |
|---|
| Jim, [Maple, PA], p1.jpg |
| Tim, [Birch, MP], p2.jpg |
| Emily, [Oak, PA], p3.jpg |

If we perform `ERJ` using a match condition such that the first two tuples match each other but the third has no match, then we get the following two tuple-pairs:

| |
|---|
| Jim, [Maple, PA], p1.jpg, Tim, [Birch, MP], p2.jpg |
| Emily, [Oak, PA], p3.jpg, Emily, [Oak, PA], p3.jpg |

We will see later how self-pairing of otherwise unmatched records (e.g., the second tuple in this example) works together with the `Combine` modifier for an appropriate end result.

**Binary-ERJ.** Consider "`R1 ERJ R2 On <Cond>`." The result contains all tuple-pairs $(t_1, t_2)$ such that:

- $t_1$ is in `R1`, $t_2$ is in `R2`, and $t_1$ and $t_2$ satisfy `<Cond>`
- For every $t_1$ in `R1` such that there is no $t_2$ in `R2` satisfying `<Cond>`, ($t_1$, NULL) is included

- For every $t_2$ in `R2` such that there is no $t_1$ in `R1` satisfying `<Cond>`, (NULL, $t_2$) is included

The self-join version of ERJ is used most commonly, however the binary version can be useful when beginning the entity-resolution process with data scattered across multiple tables, as illustrated by an example in Section IV-C.

One important subtlety is that, in TriQL, conditions are specified using standard SQL syntax over attributes and values. Thus, conditions are evaluated over conventional tuples (i.e., one value for each attribute), not over or-tuples (multiple possible values for attributes). Since TriQL `Where` conditions filter at the alternative level [1] there is no problem with this approach, however the outerjoin-like semantics of `ERJ` dictates that `On` conditions should be true for all combinations of attributes in each tuple-pair, or none of them, to avoid ambiguity. Fortunately this property is easy and natural to satisfy through TriQL's *horizontal subqueries*, explained in the next section.

*2) Match Condition:* The `ERJ On` predicate specifies the *match condition* applied to pairs of records. Recall that each tuple may contain multiple alternative values for some or all of its attributes. Match conditions typically perform comparisons of possible attribute values, often invoking foreign functions—Trio-ER includes a library of *attribute comparison functions* for base data types. The comparison results are then combined in some fashion to decide whether there is an overall match. TriQL's *horizontal subqueries* turn out to be very useful for specifying such match conditions, and they generally satisfy the desirable "all or nothing" property of `On` conditions motivated above.

We briefly explain horizontal subqueries, with several examples to follow; for full details see [1]. A horizontal subquery applies to a single tuple $t$, or to a tuple-pair as in the previous section. Logically, the horizontal subquery "verticalizes" $t$ into a relation containing one tuple for every possible combination of attribute values in $t$. The subquery then operates on this logical relation. Because horizontal subqueries typically operate on the same schemas as their enclosing queries, their `Select` and `From` clauses are often omitted; see [1] for details.

As an example, consider the following tuple-pair that might be generated during `ERJ` processing:

| |
|---|
| (Jim ‖ Tim ‖ Tom, [Maple, PA], p1.jpg), (Tim, [Oak, PA] ‖ [Spruce, PA], p1.jpg ‖ p2.jpg) |

When "verticalized," this tuple-pair generates a logical relation containing 12 tuples. (The astute reader will observe the correspondence to Trio's original x-tuple model.) The following `On` condition, comprised of two horizontal subqueries (denoted by square brackets), would be satisfied by the above tuple-pair:

```
Exists [r1.name = r2.name]
And Not Exists [r1.city <> r2.city]
```

but the following horizontal subquery would not be satisfied:

```
Exists [r1.street = r2.street
        And r1.city = r2.city]
```

*3) `Combine` Modifier:* The tuple-pairs produced by the `ERJ` operator are filtered by the optional `Where` condition, then processed by the `Select` clause before being added to the query result. `Select` is where record merging occurs, typically performed by TriQL's new `Combine` operator, although alternative merging techniques (expressed in TriQL or by invoking a foreign function) are supported.

The `Combine` modifier is simple and general: it unions possible values for all identically-named attribute groups appearing in the `Select` list. The common case, in which a self-ERJ is coupled with "`Select Combine *`", has a `Select` list with two copies of each attribute-group, which are then merged. For example, if we performed `Combine` on the example tuple-pair from the previous section, we would get:

| |
|---|
| Jim ‖ Tim ‖ Tom, [Maple, PA] ‖ [Oak, PA] ‖ [Spruce,PA], p1.jpg ‖ p2.jpg |

Note that applying `Combine` to a tuple-pair $(t, t)$ results in the single tuple $t$. This behavior works together with the semantics of self-ERJ (Section IV-B.1) to produce the desired result when tuples have no matches.

## *C. More Examples*

Now that we have introduced the TriQL constructs that are the building blocks for basic entity-resolution queries, we illustrate their use with additional examples. After these examples, we will discuss lineage and confidences, then we will see how this type of query is used as the core of an iterative entity-resolution process.

We continue with table `Person(name, [street, city], photo)`. Suppose two records match if: the average string-match similarity across all pairs of possible names is more than 0.9; at least one pair of addresses is identical; and at least one pair of photos is similar. For photo similarity, we invoke a foreign function `ImgCmp(`$n_1$,$n_2$`)` that takes two file names, compares the images in the files, and returns a value in $[0, 1]$. The following TriQL query performs one iteration of matching and merging.

```
Select Combine *
From Person p1 ERJ Person p2
On [Avg(StrComp(p1.name,p2.name))]>0.9
And Exists [p1.street = p2.street
            And p1.city = p2.city]
And Exists [ImgCmp(p1.photo,p2.photo)>0.9]
```

Note that since horizontal subqueries are applied to all combinations of possible attribute values, the two `Exists` subqueries in the above `On` condition could be combined into one. On the other hand, possible values for `street` and `city` are correlated (since they form an attribute group; recall Section IV-A), so their `Exists` subquery cannot be split. In general, TriQL's horizontal subqueries, which include most SQL constructs, offer a flexible and expressive means of specifying record matching conditions built from individual attribute comparisons.

Our second example demonstrates several additional features. Consider two tables, Pname(name,photo) and Pid(ID,photo). Suppose our goal is to combine the two tables, merging pairs of records whose photographs match; thus, here we use a binary ERJ operator. Suppose also we have three different image comparison functions, ImgComp1, ImgComp2, and ImgComp3, and we want to experiment with weighted combinations of them. Finally, suppose we only want results with ID in the range 100-200. In the following query we assume that the tables to be combined are *certain*, i.e., they do not include alternative possible values, and w1,w2,w3 are weights that might be adjusted experimentally.

```
Select Combine *
From Pname as p1 ERJ Pid as p2
On [Avg(w1*ImgComp1(p1.photo,p2.photo) +
   w2*ImgComp2(p1.photo,p2.photo) +
   w3*ImgComp3(p1.photo,p2.photo))] > 0.95
Where p2.ID >= 100 And p2.ID <= 200
```

Note that the Combine operator creates an output table with schema (name,ID,photo), with alternative photo values for those tuple-pairs satisfying the match condition. Tuples with no matches contain NULL for either name or ID, although the latter tuples are filtered out by the Where condition.

## D. Lineage and Confidence Values

One of Trio's salient features is tracking *lineage* in query results. In the context of entity-resolution, the result table generated by a matching-and-merging query such as those shown above automatically includes, for each result tuple $t$, logical pointers to the original tuples (or tuple) from which $t$ was produced. The Trio system includes features for querying and for visually exploring lineage [2], useful for drilling-down or debugging during and after the entity-resolution process. In the next subsection we will see how lineage also enables some important steps during iterative entity-resolution.

Trio supports optional *confidence values* attached to alternative values, typically denoting their relative probability. Three factors may influence the confidence values (still optional) attached to result data during entity-resolution:

(1) Confidence values on the input tuples, when present.

(2) How many instances of a value are combined during merging. For example, if we perform Combine on 'Jim || Tim' and 'Tim || Tom' (with no confidence values of their own), then in the result by default 'Tim' is assigned confidence 0.5 while the other two names are assigned confidence 0.25.

(3) Comparison function results. For example, if two records are merged because StrComp(name1,name2) returns an over-threshold value $p$, then we might wish to incorporate $p$ into confidence values assigned to the merged record.

(1) and (2) closely follow the confidence computations that were already present in Trio, with only minor modifications needed. (3) requires some language extensions, currently under design. Also under design are mechanisms for the user to override the default confidence computations in (1) and (2).

## E. Iteration

In the local-pairwise style of entity-resolution targeted by the first version of Trio-ER, it is typical to perform matching and merging iteratively [3]: more than two records may represent the same entity, and merges may create additional matches. To do so, we place one or more entity-resolution queries inside a loop. Suppose we begin with a table $T$. After one execution of a self-ERJ query, we obtain a new table $T_1$, containing lineage to $T$. If additional merges are possible, we execute the query again, now on $T_1$ producing $T_2$. $T_2$ contains lineage to $T_1$, although at this point we have a choice: we can retain $T_1$, or we can "unfold" $T_2$'s lineage so it points to $T$ instead of $T_1$, then discard intermediate table $T_1$. We continue in this fashion, creating $T_3$, $T_4$, ..., retaining or discarding intermediate tables, until at some point no merges are possible—a fixed-point has been reached and entity-resolution is complete.

Unfortunately this iteration is not always efficient or even correct; depending on the data, some intermediate or post-processing may need to be performed. For example, if matching and merging are transitive, we can improve efficiency by merging "connected components" after each iteration, reducing the total number of iterations required to reach a fixed-point [5]. For some data sets, correctness and convergence require discarding *dominated* records between iterations [3]. We've been delighted to discover that all of the natural intermediate and post-processing steps we've encountered so far can be implemented easily by operating on lineage, rather than through more complex algorithms on the data itself.

## F. Blocking and Canopies

In the worst case the type of query we've proposed may require $O(n^2)$ record comparisons in each iteration, a cost that may be unacceptable even for moderately-sized tables. *Blocking* and *canopies* are two pre-filtering techniques for eliminating comparisons that are unlikely to yield matches [5]. For example, before performing expensive string comparisons, we might first apply a *soundex* algorithm to "bucketize" records: records with similar-sounding names land in the same bucket. (Records with multiple possible names may be placed in multiple buckets.) Then, the match condition is applied only to records in the same bucket. In theory the query processor can deduce this execution strategy from the following On clause:

```
On Exists
    [Soundex(p1.name) = Soundex(p2.name)
     And StrComp(p1.name,p2.name) > 0.95
     And StrComp(p1.addr,p2.addr) > 0.95]
```

So that we needn't rely on a sophisticated query optimizer, and so that we may tune the blocking techniques (e.g., specifying the number of hash buckets), we have introduced a separate Blocking clause for indicating clearly which predicate(s) are to be applied first to partition the records. Note that if we do have a sophisticated query optimizer, it can also exploit

`Where` predicates to avoid unnecessary comparisons, e.g., in the second example of Section IV-C.

## *G. Cached Comparison-Function Results*

An additional feature in Trio-ER, improving both efficiency and usability, is caching the results of attribute comparison functions. Typically, the most expensive part of query execution in an entity-resolution setting is the evaluation of domain-specific, sometimes quite complex, comparisons over many attribute pairs. Trio-ER can be configured to save the results of such functions.

Consider for example the `On` condition in the previous subsection. Trio-ER will create special data tables caching the results of `StrComp` over all compared names and addresses. These tables are indexed for efficient lookups, and they are referenced in place of re-evaluating the functions when the query is called multiple times during iterative ER. The tables also can be used when an ER query is modified but still calls some of the same comparison functions, e.g., if in our example a comparison threshold is changed from 0.95 to 0.9.

Another useful feature of the comparison function cache is for lineage: The lineage of a merged tuple can now include not only the tuples from which it was derived, but also the function results that caused the `On` condition to be satisfied and the tuples to be matched.

## References

[1] TriQL: The Trio query language. Available at http://i.stanford.edu/widom/triql.html.

[2] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[3] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: a generic approach to entity resolution. *VLDB J.*, 18(1):255–276, 2009.

[4] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[5] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.

[6] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, pages 262–276, 2005.