Query-Time Record Linkage and Fusion over Web Databases

El Kindi Rezig
Department of Computer Science
Purdue University
West Lafayette, IN 47907
erezig@cs.purdue.edu

Eduard C. Dragut
Computer and Information
Sciences Department, Temple University
Philadelphia, PA 19140
edragut@temple.edu

Mourad Ouzzani, Ahmed K. Elmagarmid
Qatar Computing Research Institute
Doha, Qatar
{mouzzani, aelmagarmid}@qf.org.qa

***Abstract*—Data-intensive Web applications usually require integrating data from Web sources at query time. The sources may refer to the same real-world entity in different ways and some may even provide outdated or erroneous data. An important task is to recognize and merge the records that refer to the same real world entity at query time. Most existing duplicate detection and fusion techniques work in the off-line setting and do not meet the online constraint. There are at least two aspects that differentiate online duplicate detection and fusion from its off-line counterpart. (i) The latter assumes that the entire data is available, while the former cannot make such an assumption. (ii) Several query submissions may be required to compute the "ideal" representation of an entity in the online setting. This paper presents a general framework for the online setting based on an iterative record-based caching technique. A set of frequently requested records is deduplicated off-line and cached for future reference. Newly arriving records in response to a query are deduplicated jointly with the records in the cache, presented to the user and appended to the cache. Experiments with real and synthetic data show the benefit of our solution over traditional record linkage techniques applied to an online setting.**

I. Introduction

A key task in integrating data from multiple Web sources is to recognize records referring to the same real world entity—*the record linkage problem* [1]. This task is known to be difficult since the attribute values of an entity may be represented in different ways or even conflict with each other, e.g., different addresses for the same business. Conflicting data may also occur because of multiple correct values for the same real world entity, incomplete data, out-of-date data or erroneous data. Thus, another important task is to identify the correct attribute values of an entity—*the data fusion problem* [2].

In a virtual integration system (VIS), such as a vertical search engine, all of the above tasks must be performed at query time. Given that the data integration step is just one part of the process of getting data to users (which includes network communication, ranking, etc.), it needs to be performed very *fast*. Fusing all data upfront is obviously not an option in a VIS. *The goal of this work is to provide efficient solutions to the record linkage and fusion (*RL&F*) problems at query-time.*

An example will help illustrate the challenges tackled in this paper. Consider a VIS that integrates data from the following Web databases: Metromix.com, DexKnows.com, Yelp.com and Menuism.com. The following query is submitted to the VIS: $Q1 =$ (`Name` = "Pizz%"; `Cuisine` = "Pizza"; `Price` = "Affordable"; `Neighborhood` = "Downtown, Chicago"). (% is used as a wildcard character). Among all the returned records, we look at those associated with the restaurant "Pizzeria Uno" (Table I). Metromix does not return any record, while the records returned by the other Web databases do not agree on the address and phone number. At this time, the best we can do is to flip a coin to decide on the address of the restaurant and to take a majority voting to decide on the correct phone number. Suppose now that at a later time a new query is posted: $Q2 =$ (`Name` = "Pizz%"; `Cuisine` = "Pizza"; `Price` = "%"; `Neighborhood` = "%"). Table I shows the set of records for $Q2$. The list of results of the two queries gives us the opportunity to see, although at different time intervals, multiple records about the entity "Pizzeria Uno". Thus, if we stored the answer to query $Q1$, then, using the answer to $Q2$ together with that to $Q1$, we could make a more informed decision about the correct address and phone number of "Pizzeria Uno" ("49 E. Ontario St." and "312-280-5115", respectively) for any ulterior queries where "Pizzeria Uno" is a relevant answer.

TABLE I: An example of inconsistent data on the Web.

Engine	Name	Address	Phone
Query Q1			
Metromix			
Menuism	Pizzeria Uno	49 E. Ontario St.	312-280-5115
DexKnows	Pizza Uno		312-280-5111
Yelp	Pizzeria Uno	29 E. Ohio St.	
Query Q2			
Metromix	Pizzeria Uno	49 E. Ontario St.	312-280-5115
Menuism	Pizzaria Uno	49 E. Ontario St.	312-280-5115
DexKnows	Pizza Uno	49 E. Ontario Street	312-280-5115
Yelp			

The example shows that regardless of the effectiveness of the RL&F algorithms, a qualitative answer to a query Q cannot be given considering the records in response to Q alone.

To support the above claim we conducted an empirical study where we constructed a toy VIS that connects 10 Web databases of local business listings. For efficiency purposes, the VIS collects the top-k (k = 10, 20 are commonly used) results from each Web database [3]. We submitted 1,000 randomly generated queries to each of these databases and then analyzed the overlap between the returned result lists (k = 10 was used). For example, the lists of results of Zagat and DexKnows did *not* share any record for 721 queries. As shown in Table II,

TABLE II: Number of queries for which the overlap w/ Zagat is empty.

Web Database	ChicagoReader	YellowPages	Metromix	MenuPages	Yelp	Yahoo	CitySearch	Menuism	DexKnows
# occurrences	511	587	649	667	673	676	702	714	721

this was not a rare occurrence. The table shows the number of queries for which Zagat and each of the Web databases have *zero* records in common. We observe that, on average, in the merged list of results of a query, about 70% of the records appear in one or two sources (out of 10) and only about 15% of them appear in more than 5 sources. These observations clearly suggest that effective record linkage cannot in general be undertaken *query by query in isolation*, because there is not sufficient "cleaning evidence" from the records returned by a single query.

We can alleviate the lack of "cleaning evidence" by collecting "evidence" from queries as we process them. We approach the online RL&F problems from an *iterative caching* perspective (see Fig. 1). Specifically, the set of records corresponding to frequently posted queries is deduplicated off-line and cached for future references. Newly arriving records in response to a user query are deduplicated jointly with the records in the cache, presented to users and appended to the cache. The framework of the problem addressed in this paper is as follows:

The Setting: Let $\mathcal{D}$ be a set of Web databases. Let $\mathcal{E}$ be a set of real-world entities in the same application domain (e.g., real estate, book). Each entity has a set of attributes (e.g., name, address, phone for a business entity) and an attribute may have zero or several values. Different sources may supply different values for an attribute of an entity and the same value may be represented differently. A subset of the entities in $\mathcal{E}$ are frequently requested and a fraction of the volume of queries occurs frequently. Thus, RL&F in this environment faces unique challenges (e.g., time) and opportunities (e.g., temporal locality in queries) compared to a traditional setting.

The Problem: Let Q be a query. From the lists of records returned by the $\mathcal{D}$ Web databases in response to Q, we need to identify the set of records R referring to the same real-world entity in $\mathcal{E}$ and fuse the records in R into a single and "clean" representation. That is, solve the RL&F problems for Q.

The Proposed Solution: A solution for the above problem must achieve a trade-off between *efficiency* and *effectiveness*. Nevertheless, it needs to do so without significantly deteriorating effectiveness. We seek to improve efficiency by (1) avoiding repetitions of data cleaning steps such as unnecessary fusion, and (2) identifying the duplicates of a record in a constant time that is independent of the number of database records by using an index. Concretely, we propose an online record linkage and fusion framework (*ORLF*, for short) based on iterative caching. *ORLF* stores fused records as plain records together with information about how they were created, i.e., provenance. A non-trivial problem in the proposed RL&F framework is that of quickly finding the candidate matching records in the cache of a record in the query answer. We cannot afford to go through all the records in the cache to match them against the query records. In Section IV-A, we give a novel indexing data structure for fast similarity record lookup based on the B^{ed}-tree data structure [4].

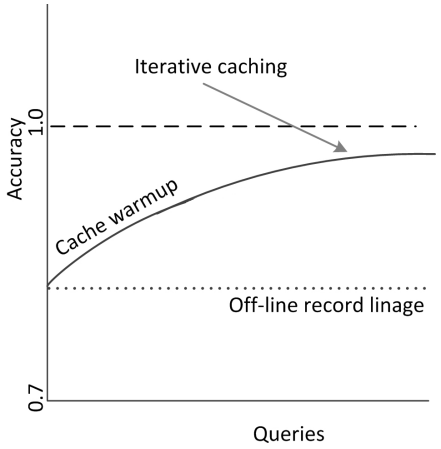


Fig. 1: *ORLF* versus typical RL algorithms.

Fig. 1 is a simple sketch to illustrate our goal in *ORLF*. The Y-axis is the RL accuracy, usually reported with the classical F1-measure. The X-axis is the number of queries whose answer sets were processed. With a typical RL algorithm (aka off-line in this paper) [5], [6], [7], [8], the accuracy is about the same (linear) across processed queries, oscillating about their average accuracies (the dotted line). The solid line is the behavior of RL in the proposed *ORLF* framework. There is an initial warmup phase, where the accuracy is about the same as that of a typical RL algorithm. As the system processes more queries, the accuracy steadily improves. The key benefit of *ORLF* however is that the RL&F steps take milliseconds rather than seconds as with a typical RL&F algorithms. Completely building *ORLF* requires the implementation of the following components: (1) cache attribute selection, (2) cache data structures, (3) efficient data lookup, (4) online RL&F algorithms, (5) caching policies, (6) cache warm up and (7) cache refreshing.

The contributions of this paper are:

- We propose, *ORLF*, an end-to-end framework to support efficient RL&F at query-time.
- We present an indexing scheme for records based on the B^{ed}-tree index [4]. B^{ed}-tree supports relatively fast record linkage and dynamic updates (for dynamic caching). To our knowledge, no other record linkage indexing technique satisfies both properties.
- We leverage query locality to perform query-time RL&F efficiently and show that smart caching avoids unnecessary fusion operations.
- We conduct extensive experiments on real and synthetic data showing the accuracy and scalability of *ORLF*.

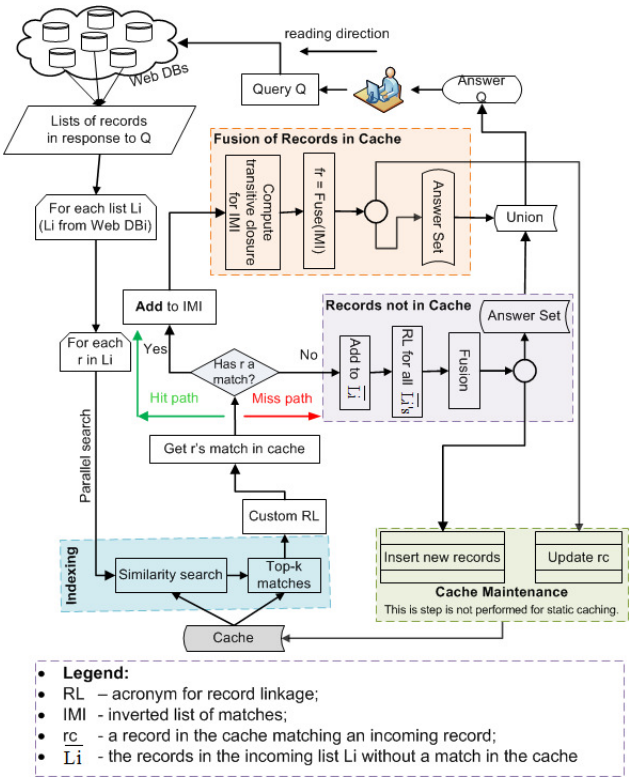


Fig. 2: Query answering using RL&F with iterative caching.

The paper is organized as follows. Section II describes *ORLF*. Section III discusses caching. Section IV describes the query-time answering. Section V describes *ORLF* in action with a comprehensive example. Section VI describes the experiments. Related word is in Section VII. Section VIII concludes the paper.

II. The *ORLF* System

We describe in this section the workflow of the proposed iterative caching approach for online RL&F. In a nutshell, newly arriving records in response to a user query are cleaned jointly with the records in the cache, presented to users and appended to the cache. The workflow is drawn in Fig. 2.

There are two processing paths: the *hit and the miss paths*. Both start by looking up the records returned by each source in response to a query in the cache. The lookup process is a two-step process. First, for each incoming record r we use an indexing data structure to retrieve the matching record of r in the cache. We employ a string-based similarity search (see Section IV. We collect the top-k most similar records to r in the cache. A *custom record matching function* is then used to find the matching record r in the top-k records. If a matching record, denoted r_c, exists, called a *hit*, then we take the *hit path*. If r has no match in the cache, called a *miss*, we take the *miss path*. In the former, we append r_c and r to a temporary indexing data structure *IMI* (inverted match index). Each index entry in IMI is of the form $\langle r_c, LM \rangle$, where r_c is a cache record and LM is the list of incoming records matching r_c. At the end of the lookup step, IMI will contain all the record matches between the cache records and the incoming records.

Note that we do not move to the next processing block on the hit path, i.e., Fusion of Records in Cache, until we process all incoming records in response to the user query. In the Fusion of Records in Cache processing step, IMI is traversed and each r_c is fused with the incoming records in its corresponding list of matches. On the miss path, we collect all incoming records without a match in the cache and perform record matching and then fusion among them. Finally, we union the lists of records of the two paths and pass the result to the user. In practice, the union is input to a ranking algorithm (not treated in this paper), which orders the records according to some user criteria (e.g., price or user rating).

The cache content is updated based on the adopted cache policy (Section III). We will show that there are substantial differences between traditional caching and caching for online RL&F. Iterative caching allows a "fast" response to the current query and an "improved" data quality for subsequent queries.

III. Caching Mechanisms

The decision of what to cache can be taken either off-line (static) or online (dynamic) in general. A static cache is based on historical information. A dynamic cache has a limited number of entries and stores items according to the sequence of requests. Upon a new request, the cache system decides whether to evict some entries in the case of a cache miss. Online decisions are based on the cache policy. Two common policies are: evicting the least recently used (LRU) or the least frequently used (LFU) items from the cache [9]. We analyze the suitability of these strategies to online RL&F. We also introduce locking of cached items and record provenance to improve efficiency when interacting with the cache.

A. Static Cache

With a static cache we need to identify the most frequently accessed entities and load their corresponding records in the cache. These entities are derived from the records corresponding to the most frequent queries gathered from the Web databases. Results of the frequent queries are processed off-line using off-line RL&F algorithms. We used FRIL [5] for RL and majority voting for fusion in our prototype. More sophisticated fusion techniques can be used [10].

We introduce a variation of static caching where records in the cache are allowed to be updated, called *static cache with in-place updates* (SCU). The reason is that even though the cache content is determined off-line, there can be cached records for which the number of pieces of evidence is not enough to decide whether they are correct (see example in Section I). Hence, it is of practical importance to allow online updates on these records. A cache record is updated by fusing it with incoming matching records from sources which have *not* yet contributed to the cache record. We keep track of the provenance of each cache record for that purpose.

B. Dynamic Cache

The semantics of a miss is different in our caching from the traditional one. In the latter, upon receiving a request for an item v, the cache is probed for v. If v is not found, v is brought into the cache from the disk. If the cache is full, then a cached item is evicted to make space for v. In our caching

setting there is no disk: all the known records are those in the cache and the ones in response to a query. Thus, on a miss, i.e., the match of an incoming record r is not in the cache, r itself is brought into the cache after RL&F. Each cache record has a *cache tag* that stores the information needed for accomplishing dynamic caching, e.g., the number of times (or the last time) the fused record was output to users.

C. Locking

Avoiding unnecessary cleaning operations is key to improving online efficiency. We use a *locking* mechanism to avoid unnecessary invocations of the fusion step. Specifically, a cache record is locked if our confidence in its quality is "high". Determining when "high" is reached for a unlocked record is orthogonal to our system. It can, for example, be probabilistic [10]. In our prototype, we implemented a voting strategy for this purpose: a record is locked if the value of each of its attributes is obtained by fusing the records from p of the sources. We empirically set $p = \frac{|\mathcal{D}|}{2} - 1$. Note also that locking provides a means for implementing online cache refreshing: certain locked records are unlocked at certain time intervals and are refreshed via fusion. A thorough treatment of refreshing is left for future work.

D. Record Provenance

If a cached record was previously constructed using a record from a source S and, for a new query, S returns a record r that matches the fused record, then we may decide to discard r as it is very likely that r has been previously seen from source S or is a duplicate record. If the record was updated in the source, then the next refresh of the cache will reflect this change. We encode provenance using a bit string of length $|\mathcal{D}|$, such that the i^{th} bit is turned on if a record from the i^{th} source was involved in the construction of the fused record.

IV. Query-Time Answering

In *ORLF*, the cache contains *fused records*. Specifically, if fr_i is a record in the cache at time t_i then at time t_{i+1} the new version of fr_i is either fr_i itself or a new record fr_{i+1} that is constructed out of fr_i and a set of incoming records not in the cache, which were linked to fr_i.

Algorithm 1 describes the query answering algorithm. A key novelty in our approach is that the merging is not only performed between the incoming lists of records, but the relevant records in the cache are also involved. As explained earlier, there are two processing paths: the *hit and the miss paths*. Each list of responses is processed by Procedure RecordLookupBySource (Algorithm 2) which updates the IMI (inverted match index) data structure, which will contain the set of cache records matching records in R_i and outputs Y, the set of records in R_i without a match in the cache. Y is appended to the set of unmatched records $\overline{MR}$ (Line 7, Algorithm 1). The matching records from all the incoming lists R_i and cache are fused (Line 9). The unmatched records are linked among themselves and then fused (procedure HierarchicalRLF, Line 10). The resulting fused records that do not already exist in the cache are appended to the cache (Line 11). This step is only executed for dynamic caching. Those that already exist in the cache are updated (Line 10). This step is executed when either SCU or dynamic cache are used. The union of the fused records, from both the matched and unmatched records, are then returned to the user (Line 13).

Algorithm 1: QueryProcessing in the *ORLF* system

Input : Query Q and Cache
Output: The set of fused records FR in response to Q and an updated cache.

1 Let $\{R_1, ..., R_m\}$ be the lists of records from m Web databases in response to Q;
2 Initialize IMI ; `// the inverted match index`
3 $MR \leftarrow \emptyset$; `// matched records in the cache`
4 $\overline{MR} \leftarrow \emptyset$; `// unmatched incoming records`
5 **for** $i = 1$ **to** m **do**
6 $\quad Y \leftarrow$RecordLookupBySource(R_i, S_i, IMI);
7 $\quad \overline{MR} \leftarrow \overline{MR} \cup Y$;
8 $MR \leftarrow$ **Transitive closure of IMI**;
9 $FR_1 \leftarrow$**fuse**(IMI);
10 $FR_2 \leftarrow$**HierarchicalRLF**($\overline{MR}$);
11 **addToCache**(FR_2) ; `// only in dynamic caching`
12 **updateCache**(FR_1) ; `// in dynamic caching and SCU`
13 **return** $FR_1 \cup FR_2$;

HierarchicalRLF performs pairwise RL. For example, if $\overline{R_1}, \overline{R_2}, \overline{R_3}$ and $\overline{R_4}$ are the four lists of records, then it performs RL on $\overline{R_1}$ and $\overline{R_2}$, and on $\overline{R_3}$ and $\overline{R_4}$. RL is performed again on their outputs to obtain the final list of linked records. This scheme is captured as a full binary tree, where a leaf node is the list of unmatched records from a source and an internal node represents the RL outcome on its children. While HierarchicalRLF is not as effective as a RL procedure that exhaustively compares the records of the m sources across each other, it is more amenable to a parallel implementation. Nevertheless, it is significantly more efficient and very effective within our overall framework (Section VI).

A. Record Look Up and Record Linkage

Let r be an incoming record and fr the record in the cache *most similar* to r. The problem is how to find fr in an online caching setting — fr may be substantially different from r, because fr has been subject to several fusion iterations. Hence, fr can only be found using similarity search. And operationally, how to check if fr indeed exists in the cache? An exhaustive search of the cache to look for fr is clearly prohibitive. We propose a two-step process: (1) fast approximate nearest-neighbor search and (2) exhaustive record matching, where we compare r with the (smaller) set of nearest neighbors (records). In the light of (1), our proposed cache is an instance of *similarity caching* [11].

1) Nearest Neighbor Search: Given an incoming record r, we need to obtain the cached record fr such that $sim(r, fr)$ is maximized, where the similarity function is defined on the space of the "keys" of the records. A subset of the attributes of entities in $\mathcal{E}$ is a "key" if it can serve as *entity identifier*, i.e., the attributes uniquely identify an entity. For example, when matching business listings the subsets {`Name`, `Address`} and {`Name`, `Phone`} are such attributes. That is, if two business records have very similar names and addresses

or very similar names and the same phone number, then the records are very likely to refer to the same business entity. Many indexing strategies can be used with static caching: e.g., locality sensitive hashing (LSH) [12]. For dynamic caching however, we need an indexing structure that supports efficient live updates. Since in many practical cases the entity identifier attributes are strings, we choose the B^{ed}-tree index [4], a string similarity indexing. B^{ed}-tree is a B^{+}-tree based index structure, which has a number of properties that suit our environment very well. B^{ed}-tree: (i) efficiently answers selection queries, (ii) can handle arbitrary edit distance thresholds, (iii) supports normalized edit distance for all query types, in particular, top-k queries, (iv) has good performance for long strings and large datasets, and (v) supports incremental updates efficiently.

2) Modifying the B^{ed}-tree index for use in similarity record search: B^{ed}-tree index was developed for string similarity search and hence cannot directly be used in our setting. For us, it all boils down to finding a suitable record representation such that records can be indexed with this data structure and carrying the good performance of the index on strings to records. We index a set of records using their "key" (matching) attributes $Key_r = \{A_1, A_2, ..., A_n\}$.

B^{ed}-tree index overview. B^{ed}-tree [4] is an index for string similarity search based on (normalized) edit distance built on top of a B^{+}-tree. To index the strings with a B^{+}-tree index, it is necessary to construct a mapping from the string domain to an ordered domain (e.g., the integer space). Since we are interested in top-k searches, the mapping must give an ordering that satisfies two properties when used with the edit distance: comparability and lower bounding. The former property requires linear time to verify if a string is ahead of another in the given string order, whereas the latter requires that it is efficient to find the minimal edit distance between a string q and any string in an interval $[s, s']$. The latter property is needed to efficiently prune out the intervals with no strings within the given edit distance from the query string during search. Three orderings are given in [4]: (1) dictionary, (2) gram counting, and (3) gram location. Gram counting is superior to the other two for top-k queries for relatively large strings [4]. We use it in our implementation. It is defined as follows: A string s is decomposed into a set of q-grams. A q-gram is a contiguous sequence of q characters from s. A hash function maps each q-gram to a set of L buckets. We count the number of q-grams in each bucket. At this point, s is mapped into an L-dimensional vector v of non-negative integers. The final representation of s is obtained by applying a z-order on the bit representation of the components of v. z-order interleaves the bits from all vector components in a round robin fashion. Consider the string s = "Red Lion", $n = 2$ and $L = 4$; Assuming $v = \langle 3, 2, 1, 3 \rangle$, the corresponding z-order is 11011011.

Adapting B^{ed}-tree to Record Linkage. We now describe our solution using B^{ed}-tree with the gram counting order for top-k record similarity search. A naive approach to represent *Key$_r$* is to simply concatenate the key attribute values of a record to form a string. The index key is then generated by taking the z-order representation for the obtained string. One issue with this scheme is that we may end up comparing q-grams of different attributes; e.g., if we compare two business records whose names are not of the same length, then, we will compare the q-grams of the longer name to the q-grams of the address attribute of the other record.

To avoid this problem we need to treat the attribute values as first-class citizens. We analyzed two z-order representations of record keys: *z-order concatenation* and *z-order interleaving*. In both representation schemes, we first obtain the z-order representation of each attribute. Then, in the z-order concatenation scheme the index key is obtained by concatenating the resulting bit-strings of each attribute, whereas in the latter z-order interleaving scheme, we interleave the bits from all attributes in a round robin fashion.

These representation schemes are bounded by the size of L. First, L cannot be arbitrarily large. For example, efficiency-wise $L = 4$ gives the best results in [4]. This is too small to adequately represent an index key with multiple attributes. As shown in Fig. 3c (Section VI), increasing L increases the time it takes to retrieve top-k. Second, for each of the n attributes we need to allocate a contiguous number of buckets. However, each attribute may require a different number of buckets. For instance, the attribute name may require more buckets than the attribute phone. Third, we experimentally noticed that the z-order concatenation scheme is influenced by the order of the attributes, while the z-order interleaving scheme is not. So, in general we need to find a solution to the following linear equation. If we denote by L_i the number of buckets required by attribute A_i, then we have

$$L_1 + \cdots + L_n = L, \text{ where } L_i \in [b_i, B_i], b_i, B_i \in \mathbb{N}^*. \quad (1)$$

b_i and B_i denote the minimum and respectively maximum (ideal) number of buckets required by the i^{th} attribute. This equation may not have solutions, i.e., $b_1 + \cdots + b_n > L$. Thus, there may be application domains where B^{ed}-tree index is not suitable. Alternatively, it may have multiple solutions (this is a linear Diophantine equation). Ideally, all solutions need to be tested out and one that fits the best the application at hand is chosen. Enumerating all possible solutions in search for the ideal one is an overkill for some application domains because for each solution to Equation 1 we need to construct the corresponding B^{ed}-tree and carry out the empirical evaluation. We give here a heuristic procedure to locate a suitable solution.

We first order in descending order the attributes based on their selectivity property (selectivity of an attribute A is the ratio between the number of distinct values in A and the total number of records). This can be given by a domain expert or estimated by sampling. Intuitively, the more selective an attribute is the more useful is to distinguish between records about different entities and thus more buckets should be allocated to it. Let $A_{i_1}, \ldots A_{i_n}$ be the desired ordering. We make $L_{i_j} = b_{i_j}$, $1 \leq j \leq n$. Then we apply a greedy strategy as follows. Let $L' = L - \sum_{j=1}^{n} b_j$. As long as $L' > 0$ we proceed as follows. For each $1 \leq j \leq n$, if $B_{i_j} - b_{i_j} \leq L'$ then $L_{i_j} = B_{i_j}$, else $L_{i_j} = b_{i_j} + L'$ and stop. For instance, suppose the attributes are name, address and phone. Suppose that this is the desired ordering and that for each of them $b = 4$ and $B = 6$. Let $L = 15$. Applying the greedy strategy, we allocate 6 buckets to name, 5 to address, and 4 to phone.

Algorithm 2: RecordLookupBySource

Input : (R, S, IMI): the list of incoming records R from a source S and IMI - the inverted match index
Output: $\overline{M}$ - the records in R without a match in the cache. updated IMI.

```
1  foreach r ∈ R do
2      hasMatch ← false;
3      TK ←getTopK(r);
4      foreach fr ∈ TK do
5          if RecordMatch(r, fr) then
6              if S ∉ fr.Provenance then
7                  update(IMI, fr, r);
8              hasMatch ← true;
9              break;
10     if hasMatch = false then
11         add(r, M̄);
```

B^{ed}-tree Setup. Due to space constraints, we cannot give the entire set of experiments for setting up the index. We summarize the key points here. We empirically determine that beyond $L = 12$ the performance of B^{ed}-tree deteriorates considerably. We set $b_i = 4$ and $B_i = 6$. For the matching of business listings with the attributes name, address and phone, the best configuration is to allocate each attribute 4 buckets. Also, z-order concatenation gives better retrieval times on average than z-order interleaving scheme. The order of the attributes in the z-order concatenation is name, address, phone. We further evaluate the z-order concatenation scheme in Section VI-A under different parameters of the B^{ed}-tree.

3) Record Matching: The custom record matching function must predict with high confidence if the records r and fr refer to the same entity. Otherwise, duplicates may be inserted in both the cache and query answer. The actual function is application-dependent and, thus, orthogonal to the current work. For our proof of concept, we developed such a function as follows. We obtained a training sample by posting a number of random queries to the component search engines, then we manually labeled the pairs of matching records and learnt a binary classifier: match or not match. We constructed a decision tree from the labeled data. Once we had the decision tree we transformed it into a procedure with IF-THEN rules [13] that was plugged into our system.

4) Look up Algorithm: Algorithm 2 describes the procedure to look up an incoming record in the cache. We first post a top-k query to the B^{ed}-tree to get the k most similar records to r in the cache. We then perform pairwise comparisons between r and the top-k records to determine fr. In our implementation, we empirically set $k = 5$ (Section VI). Each new record is compared against its top-k matches from the index (Lines 4-10). Our reasoning assume: (1) cache records are distinct and (2) a cache record can match at most one record in a given source. (1) can be seen as a cache invariant and its consequence is that a new record can match at most one record in the cache. (2) may seem restrictive, but note that our goal is not to clean the individual sources, but rather to return relevant and clean records. Thus, if r matches a cache record fr, then r is retained to be later fused with fr only if a record from S has not previously contributed to the construction of fr (Lines 6-7).

TABLE III: An example cache.

ID	Provenance	Name	Address	Phone
cr_1	(1, 2, 3, 5)	Pizzeria Uno	49 E. Ontario St.	312-280-5115
cr_2	(1, 3)	Pizzeria	40 E. Ontario St.	312-280-3344
cr_3	(1, 3)	Pizza Uno	39 N. Ontario St.	312-280-2355
cr_4	(7)	Pizzeria Uno	E. Ontario St.	312-280-5115

TABLE IV: Q_1's answer set

R_i	ID	Name	Address	Phone
R_1	r_1	Pizzeria Uno	49 E. Ontario St.	312-280-5115
R_2	r_2	Pizza Uno	E. Ontario St.	312-280-5115
R_3	r_3	Pizza King	11 W. Ontario St.	312-443-7844
R_4	r_4	Giordano's pizza	33 N. Ontario St.	312-544-9033

If r has no match in the cache, it is appended to the list of unmatched records (Lines 10-11).

B. Fusion Procedure

For each set of records representing the same entity, we need to compute the fusion. For each cache attribute, the value representations that have a similarity of at least some threshold (current implementation 0.9) are considered to be "identical". Among a set of representations for a value, we choose the one provided by the largest number of sources. When a cache record fr is fused with a list of new records, the value representation of an attribute from fr receives $\lceil \frac{m}{2} \rceil$ votes, where m is the number of sources from which fr was previously derived. Other fusion schemes (e.g., [14], [10]) can easily be plugged into our framework.

C. Algorithm Complexity

The non-parallel worst-case time complexity of Algorithm 1 is $O(\frac{(m-2)(m-1)}{2}\delta(cR)^2)$, where m is the number of databases, each returning R records; c is the cache miss rate; $\delta \simeq 1-or$, where or is the overlap rate of the m sources. This occurs when most of the records have no match in the cache, i.e., c is close to 1. In this case *ORLF* is simply as good as traditional RL. The best case occurs when c is close to 0. On average however, the algorithm is near linear in mR.

V. A Walkthrough Example

We give a step-by-step illustration of Algorithms 1 and 2 through an example in this section. Consider the following query: Q_1 = {`Address` = "%Ontario%", `Cuisine` = "Pizza"}. Table III shows the cache content. We assume four sources. Q_1 is posted to all the sources. The list of records returned from a source i is stored in a list R_i. The lists are shown in Table IV. For the sake of simplicity, we assume that each source returns one record in response to Q_1. For each record r in list R_i, the system finds its top-k similar records according to the string edit distance in the cache. We assume $k = 2$ in this example. Note that there is no guarantee that the returned records are indeed real matches. Hence, we next find which ones among the k records are valid matches of r. Table V shows the incoming records after the "filtering" process is applied to the top-k set. The "edit" column in the table shows the sum of the string edit distance between r_i and cr_i on attributes {name, address, phone}.

TABLE V: Q_1's' Top-2 records and their matching status

	r_i	cr_i	Match?	edit(r_i, cr_i)
MR	r_1	cr_1	Yes	0
	r_1	cr_2	No	5
	r_2	cr_4	Yes	0
	r_2	cr_1	Yes	3
$\overline{MR}$	r_3	cr_3	No	13
	r_3	cr_2	No	13
	r_4	cr_3	No	20
	r_4	cr_2	No	20

TABLE VI: Q_1's' Inverted Index (IMI)

cr_i	Matching incoming records	Matching incoming records (after transitive closure)
cr_1	$\{r_1, r_2\}$	$\{r_1, r_2\}$
cr_4	$\{r_2\}$	$\{r_1, r_2\}$

TABLE VII: Updated cache after processing Q_1

ID	Provenance	Name	Address	Phone
cr_1	(1, 2, 3, 5, **7**)	Pizzeria Uno	49 E. Ontario St.	312-280-5115
cr_2	(1, 3)	Pizzeria	40 E. Ontario St.	312-280-3344
cr_3	(1, 3)	Pizza Uno	39 N. Ontario St.	312-280-2355
cr_4	(3)	Pizza King	11 W. Ontario St.	312-443-7844
cr_5	(4)	Giordano's pizza	33 N. Ontario St.	312-544-9033

r_1 and r_2 match cr_1 and cr_4, respectively, in the cache. r_3 and r_4 have no match in the cache. The records r_1 and r_2 along with their matches cr_1 and cr_2 are processed in the "Hit path" (as shown in Fig. 2), whereas r_3 and cr_3 are processed in the "Miss path".

Table VI illustrates the usage of IMI: cr_1 matches $\{r_1, r_2\}$ and cr_4 matches $\{r_2\}$. We are assuming that the matching relation is transitive, so, by transitivity, cr_4 also matches $\{r_1, r_2\}$ after computing the transitive closure on IMI. Then, the system fuses the records cr_1, cr_4, r_1 and r_2 into cr_1. The records r_3 and r_4 do not have matches in the cache and are appended to the cache. Table VII shows the new version of the cache after Q_1 is processed. The records that were either added or updated are highlighted. Finally, the system returns to the user the records: cr_1, cr_4 and cr_5.

The example shows that cache records may also be fused, such as cr_1 and cr_4. The iterative (and incremental) process gives us the opportunity to clean the cache itself of duplicates, should there be any, as more and more new queries are seen. Duplicates may sneak into the cache because there is no perfect record linkage procedure and true positives may be missed in early iterations, but discovered in later iterations.

VI. Experiments

The goal of our experiments is to show that *ORLF* is feasible in practice; its effectiveness and efficiency significantly exceeds those of off-line solutions, such as *Febrl* [6], when applied to the online setting. We also evaluate the main components of *ORLF* to demonstrate its robustness. All of the experiments are conducted on a machine that runs Linux, has eight Intel Xeon E5450 3.0 GHz cores and 32 GB of physical memory. We implemented the framework in C++ and used MySQL to manage the data in the sources and the cache.

A. B^{ed}-tree Index Experiments

We assess the effectiveness of the modified B^{ed}-tree index in the record linkage task. The effectiveness of B^{ed}-tree is not analyzed in [4].

We define the *sensitivity* of the modified B^{ed}-tree index as its ability to return a record r in response to the top-k query r given that r is present in the index. If the index has a low sensitivity, then *ORLF* does not benefit from caching because it cannot locate the matching records of the incoming records even when they are in the cache. We assess the sensitivity of the B^{ed}-tree index with the two strategies of representing the matching key, naive and z-order concatenation.

The sensitivity experiment requires a set T of N distinct records w.r.t the record matching function discussed in Section IV. We randomly generated over 1M records for the attributes (name, address, phone) with an approximate error rate of 20%, i.e., around 20% of the records have duplicates in the set. This does not bias the experiments as we are interested in measuring the sensitivity of the B^{ed}-tree when *most* of the records are distinct– Recall that the cache is assumed to be duplicate-free in general. We have also conducted an experiment where we indexed a set T_D of 1K records that are guaranteed to be 100% distinct from each other.

The experiment runs as follows: (1) records in the set T are indexed using the modified B^{ed}-tree index ; (2) a sample set of records, ST, of size 100 is taken from T; (3) a top-k query to the B^{ed}-tree with every record r in ST is posted; and (4) if $r \in T_r$, where T_r is the set of returned records for r, then it is a match; otherwise, it is a miss.

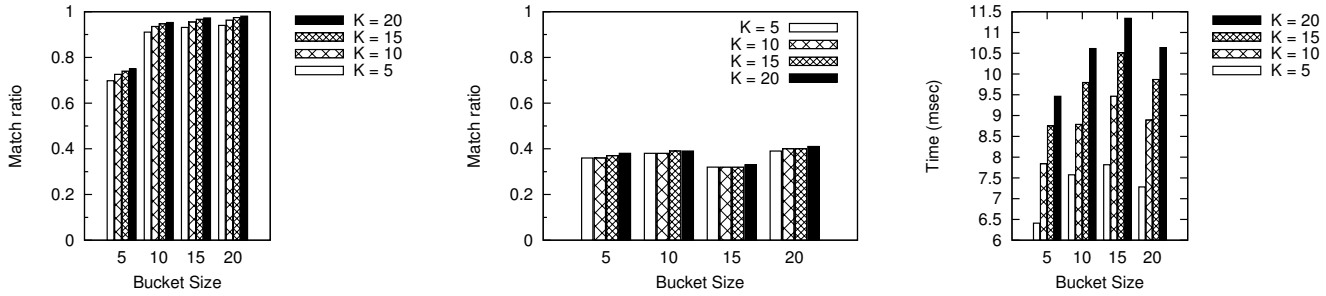
When the indexing key of a record is obtained by mere concatenation of its attribute values, B^{ed}-tree exhibits a poor quality as illustrated in Fig. 3b. The main reason is that attribute values have different lengths, thus different parts of these attribute values are compared when computing the lower-bound of the edit distance.

Fig. 3a shows the matching ratios when the indexing key is the z-order concatenation of the attribute values of a record. The matching ratios are reported for different values of k and L (the bucket size). The matching ratios are reasonable and comparable to those for the original B^{ed}-tree index. This z-order concatenation strategy is used in our implementation and all the subsequent experiments. In another set of experiments, we indexed T_D and then queried the index with all of the records in T_D. We obtained the average matching ratios 0.97, 0.98, 0.99, 0.99 for k = 5, 10, 15 and 20, respectively. This shows that the z-order concatenation scheme is very effective and suitable for indexing records with the B^{ed}-tree index.

While increasing L and k has a clear positive impact on the match ratio as shown in Fig. 3a, Fig. 3c shows a negative impact on efficiency. As a reasonable trade-off between quality and performance, we set the overall bucket size L = 12 with 4 buckets per attribute, and k = 5.

1) B^{ed}-tree Effectiveness for Plain Strings: We also need to have a sense of the effectiveness of the B^{ed}-tree index in general. We use the original implementation of the B^{ed}-tree[1]. We employ a set of 12K distinct strings corresponding

[1]We thank the authors of [4] for sharing their source code.



(a) Matching ratios according to different values of k and L (z-order concatenation).

(b) Matching ratios according to different values of k and L (Strings concatenation).

(c) Average Top-k query time using z-order concatenation to index records.

Fig. 3: B^{ed}*-tree* Sensitivity Experiments

to business names and post top-k queries. On average, we obtained a 0.99 matching ratio for k = 5, 10, 15 and 20. Larger k values are not used in practical systems in the online setting. The B^{ed}-tree index is indeed effective when applied to plain strings. This will serve as the baseline behavior for the modified version of B^{ed}-tree.

B. ORLF *Experiments*

We implemented an in-house metasearch engine to create a controlled experimental environment, we used two datasets: (1) Real Web data, to show how the system behaves in the wild; (2) Synthetic data, to show that the system is not sensitive to the data domain being used, and to better evaluate the RL quality since we know the set of duplicates for all the records. To obtain the Web dataset, we crawled the data of 9 Web databases that provide information about restaurant listings in the metropolitan Chicago, US. We also compared *ORLF*'s fusion with *Solaris* [14] on a book dataset.

Table VIII gives a general picture of the "dirtiness" of the data crawled from the 9 Web databases. The part titled "Number of records with distinct" includes four of the cache attributes. "City" and "State" are omitted as most records are from Chicago, IL. We show the number of distinct values in each of the cache attributes, e.g., there are 8,200 distinct names in Yelp. A "0" in the column of an attribute means that there are no values for the attribute in the corresponding crawled data, e.g., no record has a zip code in the crawled data from MenuPages. The part of the table titled "Number of records without" looks at the missing values in all the attributes. If we analyze the two tables jointly, we note that all the records have a name value, but not all of them have a phone number, e.g., in Yelp there are 8,744 distinct phone numbers and 367 records have no phone number.

A key question is whether the content of real Web databases overlaps significantly. We use *overlap rate* to measure the degree of overlap among a group of databases [15]: $\frac{\sum_{i=1}^{m} |D_i| - |D_u|}{(n-1)|D_u|}$, where m is the number of databases, $|D_i|$ is the number of records in database D_i and $|D_u|$ is the total number of distinct records in the union of the n databases. The overlap rate for our metasearch engine is 0.38. This is a global score; pairwise, the databases have higher rates. Consequently, the likelihood of at least two records referring to the same entity to occur in a query is very high. This emphasizes the practical importance of addressing the problem of online RL&F.

Matching Function Thresholds: We use the dataset 12Q[2] for learning and the Restaurant dataset from the RIDDLE[3] repository for testing. We obtain 12Q by posting queries to the 9 component search engines. The thresholds to match restaurant entities have been learned and applied by the *ORLF* system custom function to match restaurant pairs from the real-world data crawled from the Web.

Cache Warmup: A well-known result from information retrieval is that query frequencies follow a power-law distribution for text search engines [16], [17]. Thus, a few queries have very high frequencies and the rest appears very infrequently. To our knowledge, there is no similar study for the queries posted over search engines for (semi)structured data. It is however known that the sales of books, music recordings and almost every other branded commodity follow a power law distribution [18], [19]. Hence, we can "deduce" that the queries used to find them also follow a power law distribution. To warm up the cache, we generate a stream of structured queries following a power law distribution. We select the top 20% most frequent queries, post them, and download their results. We then apply an off-line RL&F algorithm, namely FRIL [5], on their results.

Generating Simulated Queries: To empirically analyze the behavior of *ORLF* we need to generate queries that simulate the influx of queries faced by a search engine. To our knowledge, there is no published method for simulating a *stream of structured queries* to a search engine. We give a method here. First, suppose that we know the set of query fields, say F. For example, $F = \{\text{Cuisine, Price, Location}\}$ are common fields in restaurant search engines. We assume that each field f has a predefined list of values D_f. This is quite common on the Web. Otherwise, we can draw values from some sample datasets. We then generate the entire query space by taking the cross-product of the domains of the fields, $\prod_{f \in F} D_f$. We also insert a null value in the domain of f in order to account for the queries when f is not mentioned. For instance, the query (`Cuisine` = "Mexican"; `Neighborhood` = "Loop,

[2]www.cis.temple.edu/~edragut/research.htm

[3]www.cs.utexas.edu/users/ml/riddle/data.html

TABLE VIII: Dirtiness Statistics about Crawled Data

		Number of records with distinct				Number of records without								
Source	# Recs.	**Name**	**Address**	**Zip**	**Phone**	**Name**	**Address**	**City**	**State**	**Zip**	**Phone**	**Rating**	**Reviews**	**Price**
ChicagoReader	3,096	2,759	2,877	0	2,930	0	0	0	0	3,096	35	1,436	1,436	209
CitySearch	12,695	9,162	9,867	124	0	0	0	0	0	3,035	12,695	8,883	6,866	12,695
DexKnows	5,843	4,577	5,509	61	5,706	0	0	0	0	3	3	5,636	5,636	5,843
MenuIsm	8,508	6,492	7,022	0	0	0	0	0	0	8,508	8,508	6,795	6,795	889
MenuPages	3,629	3,032	3,382	0	0	0	0	0	0	3,629	3,629	1,465	1,464	3,629
Metromix	5,044	4,599	4,719	0	0	0	7	7	0	5,044	5,044	1,627	1,627	5,044
Yahoo	10,820	8,049	9,724	0	10,490	0	47	0	0	10,820	0	5,854	5,854	10,820
YellowPages	7,798	6,547	7,159	101	7,485	0	21	1295	0	17	0	4,741	4,741	7,798
Yelp	10,115	8,200	8,914	107	8,744	0	0	87	0	37	367	2,080	2080	255

Chicago") does not mention the price. We draw a stream of queries from the set of all queries according to a power law distribution, i.e., $p(x) = C\,x^{\beta}$. $C = \frac{\beta+1}{Q^{\beta}}$, Q is the total number of distinct queries. We set β to values observed in literature (e.g., [17]) for keyword queries, e.g., $\beta \in \{0.83, 1.06\}$. We denote the generated stream of queries by Q.

Obtaining the Gold Standard: Since no automatic RL tool can guarantee perfect results, we manually construct a subset of matching records. We randomly select a subset R_M of 100 records from the crawled data. Then, we apply the record linkage tool *Febrl* to R_M and the entire crawled data and obtain a set of candidate matches for the records in R_M. We manually investigate the generated pairs to keep only the correct matching pairs P_{GS}. P_{GS} contains 420 pairs. We use P_{GS} to measure the effectiveness of *ORLF*.

Dynamic Cache with Infinite Size: In these experiments, *ORLF* is set up with a dynamic cache of infinite size, and k=5, L=12 for the B^{ed}-tree index. In the first part, we evaluate the quality of *ORLF* in the task of RL against P_{GS}. We simply count correct matching pairs that *ORLF* returns as it goes through the stream of queries. The experiment is conducted in the following manner:

1) Let Q be a stream of queries and $Q_M \subset Q$ a sub-stream of queries with the property that the list of results of each query contains exactly one record in R_M and each record in R_M appears in the list of results of some query.
2) We execute 1,000 queries from $Q - Q_M$. Then, we execute all queries in Q_M randomly.
3) For each query $q \in Q_M$, *ORLF* yields a set of duplicate record pairs P_q corresponding to q.
4) We collect all matching pairs P_{ORLF} generated by *ORLF* (Eq. 2) over the entire stream Q_M. Then, we extract the set of correct matching pairs from P_{ORLF} (Eq. 3) .

$$P_{ORLF} = \cup_{q \in Q_M} P_q \tag{2}$$

$$Correctness_{ORLF} = |P_{ORLF} \cap P_{GS}| \tag{3}$$

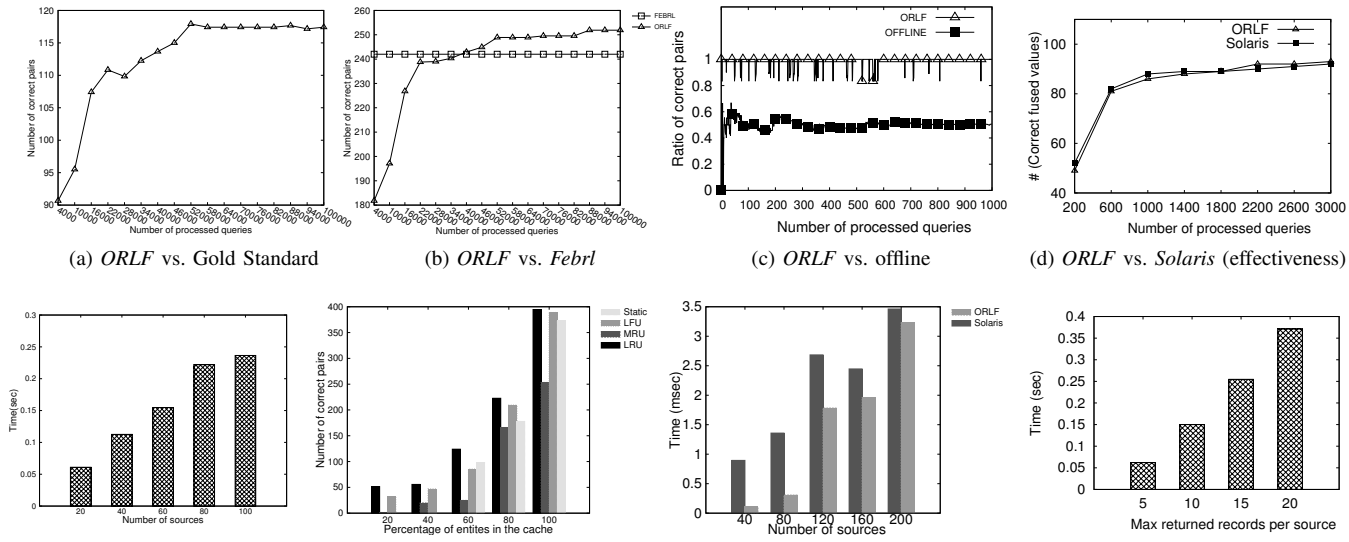
5) We repeat 2-4 until *ORLF* processes 100,000 queries. The goal is to show that *ORLF* incrementally benefits from past queries and yields significantly improved RL results.

In general, both P_q and P_{ORLF} are different at subsequent iterations because more and more records are appended to the cache. $Correctness_{\text{ORLF}}$ is independent of the number of processed records when performing RL. Hence, it is a good indicator of *ORLF* effectiveness since it provides a uniform way to measure effectiveness across iterations. The goal is to show that *ORLF* converges to P_{GS} as the system processes more queries. Note that in this and in the following experiment, to increase the randomness of the testing, we tested *ORLF* with three different query streams, corresponding to different values for the parameter $\beta \in \{0.64, 0.7, 1.3\}$ of the power law distribution, and reported the average number of correct pairs across the three runs.

Fig. 4a shows that the overall quality of *ORLF* improves sharply as the system sees more queries, then it remains relatively stable. More importantly, it does not deteriorate; *ORLF* benefits from previously processed queries and improves the quality of the current and future queries.

The second part of the experiment compares our system to simply applying an off-the-shelf off-line RL tool. We choose *Febrl* [6] because its source code is readily available online. *Febrl* takes two sets of records S_1 and S_2 as input and outputs the set P_{Febrl} of duplicate pairs from these two sets. We employ the same setting described above. *Febrl* is used as follows. For each query $q \in Q_M$ let $D_i(q), 1 \leq i \leq 9$ be the set of results returned by the i$^{\text{th}}$ Web database in response to q. *Febrl* is then applied to every pair $D_i(q)$ and $D_j(q)$ of result sets, $1 \leq i < j \leq 9$. *Febrl* is applied to 36 pairs of result sets per query. (This experiment is the most time consuming and it took several days to complete). We then union the pairs of matching records obtained from the 36 runs of *Febrl*, we call this set $P_{Febrl}(q)$. The set of pairs of record matchings produced by *Febrl* for all the queries in Q_M is given by $P_{Febrl} = \bigcup_{q \in Q_M} P_{Febrl}(q)$. P_{Febrl} is computed every 1,000 queries as is P_{ORLF}. The set of correct pairs of matching records for *Febrl* is given by $Correctness_{Febrl} = |P_{Febrl} \cap P_{GS}|$.

Fig. 4b plots $Correctness_{Febrl}$ and $Correctness_{ORLF}$ side by side. The graph clearly shows that for the initial set of queries, *Febrl* outperforms *ORLF*. However, as more queries are processed, *ORLF* starts to gradually catch up with *Febrl* and eventually outperforms it. Fig. 4b approximately mirrors the trend presented in Fig. 1. Observe that *Febrl*'s effectiveness remains about the same across the query stream.



(a) *ORLF* vs. Gold Standard (b) *ORLF* vs. *Febrl* (c) *ORLF* vs. offline (d) *ORLF* vs. *Solaris* (effectiveness)

(e) Scalability *w.r.t.* to the number of sources (f) Different cache configuration policies (g) *ORLF* vs. *Solaris* (average response time) (h) Average response time

Fig. 4: *ORLF* Experiments

Dynamic Cache with Eviction and Static Cache: We evaluated two caching strategies: dynamic caching with three eviction policies (LRU, MRU and LFU) and static caching. We use *Febrl* to clean all the crawled data from the 9 databases off-line; the resulting clusters (or entities) are loaded increasingly by fraction into the cache. We report the number of returned correct pairs (Eq. 3) computed from randomly selected queries from Q. We report the numbers for each of the considered cache sizes.

Overall, LRU has the best results and MRU has the poorest as shown in Fig. 4f– MRU evicts the most recently used entity from the cache, which is likely to be needed for future queries. The static cache performs worse than the dynamic cache, but it becomes almost as good as the dynamic cache as the cache size approaches 100% of the total entities. LRU and LFU dynamic caches show better peformance than the static cache since a dynamic cache continues to update the content of the cache over time, while a static cache does not. Increasing the cache capacity allows *ORLF* to evict less records (and thus, to keep more "useful data"); this decreases the likelihood to miss an incoming record.

We report experiments that assess *ORLF*'s performance. First, we evaluate the average query response time by varying the number of returned records per source, we used a total of 10 sources in this experiment. Fig. 4h shows the average query response time when using different values for the maximum number of returned records per source, we can see that *ORLF* is very efficient, even when processing 20 records from each source, the average query response time does not exceed 0.4 second. We also evaluate the scalability of *ORLF* by computing the average query response time for a set of 200 randomly selected queries, while increasing the number of sources to which we submit the queries. The response time is the time between querying the cache for top-k matches and producing the query's output. We do not take the database querying time into account as it depends on factors (e.g., access method, internet bandwidth) that are outside the scope of this work. We start with 9 distinct sources. We gradually increase the number of sources up to 100 by duplicating the original sources. Fig. 4e shows the scalability results. We observe that the increase in the query response time is linear in the number of sources. We have observed that the query response time is primarily dominated by the top-k query time, this is expected because the top-k queries in the B^{ed}-tree index require computing edit distance for each string in the tree leaves to find the possible matches to the querying record. In addition, the similarity search algorithms based on B-tree indices explore many branches (in the worst case all of them) in the tree that may contain candidate matches [20].

C. Synthetic Dataset

In this experimental study, we compare *ORLF* against an ideal system that has a perfect RL algorithm. That is, the ideal system does not miss a pair of matches in the list of returned records of a query. We want to showcase that because the system is stateless (does not keep information from previous queries), it cannot deliver a "clean" representation of an entity in most cases, while *ORLF* does. The experiment is set up as follows. For an entity e, let L_e be the set of records from all sources that need to be known for a fusion algorithm to compute the correct representation of e. L_e is computed by RL (over time). Missing any portion of L_e would render an imperfect version of e regardless of the fusion algorithm being used. The "cleanliness" of e is measured as the ratio $|L|/|L_e|$, where L is the set of records discovered by one of the two systems. For *ORLF*, L is the union of the pairs discovered across processed queries, while for the perfect off-line RL system, it is the set of matching record pairs for a given query.

We use synthetically generated data to track the duplicates of all the records in the dataset, and hence, to accurately measure the effectiveness of the two systems. We generated a

synthetic dataset of 1M records that contains made-up personal information such as first name, last name, and social security number. We used the tool *dsgen* which is part of *Febrl* to generate this dataset. We generated 200K records along with 800K duplicates. The maximum number of duplicates per record was set to 10. The tool introduces different types of noise to the original records to generate duplicates. In order to simulate the setup of a VIS, we randomly split the set of synthetic data into 10 sources, each containing 100K records.

The matching function checks if two records have a similar SSN, surname and phone. The string edit distance threshold for the three attributes is empirically set to 0.8.

Fig. 4c shows the results obtained for a query stream of 1K queries posted to 10 sources; we can see that *ORLF* greatly improves the quality of the returned results to the posted queries over the ideal tool. Such improvement comes from the proposed caching system which makes *ORLF* benefit from previously processed queries.

D. Comparison with Solaris

We compare *ORLF* to *Solaris* [14], which performs online fusion of records returned by a query with copying and accuracy constraints on the sources. *Solaris* has two methods: *ACCU* and *PRAGMATIC*. As reported in [14], *PRAGMATIC* has a slightly better precision than *ACCU*, but it is slower. We only implemented *ACCU* for the comparison.

Dataset: We use the dataset *Books* [14][4] since *Solaris* requires the accuracy information of the Web sources. *Books* has book records (ISBN, title and authors) from 894 sources. It comes with a gold standard set of 100 books for which we know the correct authors.

Query stream: We generate a set Q_{books} of 3000 queries that ask for the books in the gold data. For a book entity B_i that has a set of records $R_i = \{r_1^i, r_2^i, ..., r_n^i\}$ across the sources, there exists a set of queries that have different coverage ratios on R_i. $ORLF$ is expected to do well even when the query coverage ratio is low (thanks to the cache), whereas *Solaris*, due to its stateless nature, performs well when the query has a high coverage ratio. We consider queries of different coverage ratios to be fair to both systems.

We post the queries in Q_{books} to *ORLF* and *Solaris* to assess the accuracy of their fusion results. We measure this accuracy by assessing their ability to return the correct value for the authors of the books in the gold data. We consider only the sources that contain books in the gold data. There are 238 such sources. As we see in Fig. 4d, the two systems perform quite similarly. However, unlike *Solaris*, $ORLF$ performs RL (besides data fusion). $ORLF$ is thus exposed to RL mistakes (which may lead to low-quality fusion). Data fusion in the current implementation of $ORLF$ is naive and only considers majority voting among conflicting values. We observe that in the beginning, *Solaris* slightly outperforms $ORLF$. As $ORLF$ processes more queries, it starts outperforming *Solaris* as it benefits from the cached fused results.

Using the same dataset, we compare the average response time for both systems, excluding the data sources querying time. We use a query stream of 100 randomly chosen queries from Q_{books}. Fig. 4g shows that *ORLF* slightly outperforms *Solaris* since it does not perform any preprocessing for the conflicting values to be fused, it just takes the majority value. As the number of data sources increases, *Solaris*'s average time becomes closer to *ORLF*'s; this is because *Solaris* does not necessarily query all the data sources, and hence it is not as sensitive to the number of sources as *ORLF*.

[4]We thank the authors for sharing with us the dataset, the accuracy of the sources, and the gold standard.

VII. Related Work

Our work is related to four areas: metasearch engines for structured data, data fusion, record linkage and caching. It also builds upon existing off-line RL&F techniques, most of them comprehensively summarized in [1] and [21]. We are not aware of any caching method for metasearch engines over structured data. Caching nonetheless has received substantial consideration in text Web search engines [22].

To our knowledge, an RL&F system for online settings as presented in this paper, i.e., for metasearching, has not been proposed before. We are aware of three other recent works that propose online solutions [23], [14], [10]. In [23], the term "online" denotes an entirely different setting than ours, namely, a set of distributed databases whose records need to be matched over a network. The goal is to minimize the communication overhead, i.e., to minimize the number of records transferred over the network.

The approach proposed in [14] (referred to as *Solaris*) performs online fusion with copying and accuracy constraints on the data sources. The key idea is to stop probing additional sources, in response to a query, once the system is confident enough that data from the remaining sources are unlikely to change the answer computed from the probed sources. The main differences between *Solaris* and our system are: (1) *Solaris* does not perform RL (in the reported experiments, ISBN is assumed clean and used as key for RL purposes) and (2) *Solaris* is stateless, thus its fusion accuracy for an attribute value is as good as the number of records having that value or related values in the query result set. Our experiments on the same dataset show similar accuracy for both systems.

A Bayesian approach for fusing records is also described in [10]. It infers the quality of a data source for different attribute types without any supervision and incorporates this quality in the fusion process. While both works present interesting frameworks for online fusion, there are at least two issues with these solutions. First, as illustrated in Section I, the lists of returned records for a query do not a have high degree of overlap. Second, as shown in these works, a large number of sources may need to be probed to reach the desired quality level at query-time. For example, in [14] the authors show experimentally that 73 out of 100 (book) records reach a stable version after 14 sources are probed and all 100 are stable after over 90 sources are probed. Although a metasearch engine may connect to hundreds of component search engines, in practice and for efficiency reasons, it submits a query to a very small number of them: a few tens of them [24]. The databases are selected based on the query at hand and the profile of each database [25], [3]. The above two observations suggest that the proposed fusion approach may not be suitable for metasearch engines. Data from past queries, i.e., caching, is needed for accurate online data cleaning.

While iterative caching was not used to efficiently perform data quality, efficient (off-line) RL nonetheless is a major topic. Methods to speed up the performance of RL include iterative blocking [7], size filtering [26], order filtering [27], suffix filtering [28], iterative hashing [8] or "hints" as in the pay-as-you-go technique proposed in [29]. Three type of hints are proposed: sorted lists of record pairs, partition hierarchy, and sorted list of records.

Incremental record linkage is also a related topic [30], [31], [32]. An *incremental clustering* technique is proposed in [30]. For a new record, it estimates its likely cluster through a voting scheme and then it recursively updates the clusters. The algorithm is not suitable for online settings because the recursion may pass through the entire database. The solution in [31] is a heuristic incremental clustering algorithm that ignores the propagation step. The neighboring objects are never analyzed, the clusters never merge or split: a new record is either added to a cluster or forms a new cluster. [32] proposes two graph incremental clustering algorithms for RL. The proposed algorithms are intractable in general and the proposed solutions are not suitable for the online setting.

[33] reports a probabilistic approach to online RL. The aim is to return alternative linkage assignments with assigned probabilities in a response to a query, whereas we return duplicate-free query results along with fused attribute values.

VIII. Conclusions

We proposed a novel approach for record linkage and fusion in an online setting. Our approach is based on *iterative caching*: a set of frequently requested records (obtained from the different Web databases through sampling) is cleaned off-line and cached for future references. Newly arriving records in response to a query are cleaned jointly with the records in the cache, presented to users and appropriately appended to the cache. Our solution allows a "fast" response to the current query and an "improved" data quality for subsequent queries.

There are at least two items for future work: (1) Devise a measure of degradation of the cache, which would trigger a cache refresh. (2) Incorporate better fusion algorithms in *ORLF*. The solution presented in [10] seems to be better amenable to our framework due to its incremental nature.

Acknowledgement

This work was supported by National Science Foundation Grants IIS 0916614, IIS 1117766, and the Qatar Computing Research Institute.

References

[1] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," *TKDE*, vol. 19, no. 1, 2007.

[2] J. Bleiholder and F. Naumann, "Data fusion," *ACM Comput. Surv.*, vol. 41, pp. 1:1–1:41, January 2009.

[3] W. Meng and C. Yu, *Advanced Metasearch Engine Technology*. Morgan & Claypool Publishers, 2010.

[4] Z. Zhang, M. Hadjieleftheriou, B. C. Ooi, and D. Srivastava, "Bed-tree: an all-purpose index structure for string similarity search based on edit distance," in *SIGMOD Conference*, 2010, pp. 915–926.

[5] P. Jurczyk, J. J. Lu, L. Xiong, J. D. Cragan, and A. Correa, "Fril: A tool for comparative record linkage," in *AMIA*, 2008.

[6] P. Christen, "Febrl an open source data cleaning, deduplication and record linkage system with a graphical user interface (demonstration session," in *SIGKDD*, 2008, pp. 1065–1068.

[7] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina, "Entity resolution with iterative blocking," in *SIGMOD*, 2009.

[8] H.-s. Kim and D. Lee, "Harra: fast iterative hashed record linkage for large-scale data collections," in *EDBT*, 2010, pp. 525–536.

[9] N. Megiddo and D. S. Modha, "Outperforming LRU with an Adaptive Replacement Cache Algorithm," *Computer*, vol. 37, pp. 58–65, 2004.

[10] B. Zhao, B. I. P. Rubinstein, J. Gemmell, and J. Han, "A bayesian approach to discovering truth from conflicting sources for data integration," *PVLDB*, vol. 5, no. 6, 2012.

[11] F. Chierichetti, R. Kumar, and S. Vassilvitskii, "Similarity caching," in *PODS*, 2009, pp. 127–136.

[12] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998, pp. 604–613.

[13] J. Han, *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 2005.

[14] X. Liu, X. L. Dong, B. C. Ooi, and D. Srivastava, "Online data fusion," in *PVLDB*, 2011.

[15] S. Wu and S. McClean, "Result merging methods in distributed information retrieval with overlapping databases," *Information Retrieval*, 2007.

[16] Y. Xie and D. O'Hallaron, "Locality in search engine queries and its implications for caching," in *In IEEE Infocom*, 2002, pp. 1238–1247.

[17] R. Baeza-Yates, A. Gionis, F. Junqueira, V. Murdock, V. Plachouras, and F. Silvestri, "The impact of caching on search engines," in *SIGIR*, 2007, pp. 183–190.

[18] R. A. K. Cox, J. M. Felton, and K. H. Chung, "The concentration of commercial success in popular music: An analysis of the distribution of gold records," *Journal of Cultural Economics*, vol. 19, 1995.

[19] R. Sah and R. Kohli, "Market shares: Some power law results and observations," Harris School of Public Policy Studies, University of Chicago, Working Papers 0401, Jan. 2004.

[20] M. A. Bender, M. Farach-Colton, and B. C. Kuszmaul, "Cache-oblivious string b-trees," in *PODS*. ACM, 2006, pp. 233–242.

[21] X. L. Dong and F. Naumann, "Data fusion: resolving data conflicts for integration," *PVLDB.*, vol. 2, pp. 1654–1655, 2009.

[22] E. P. Markatos, "On caching search engine query results," in *Computer Communications*, 2000, p. 2001.

[23] D. Dey, V. Mookerjee, and D. Liu, "Efficient techniques for online record linkage," *TKDE*, vol. 23, 2011.

[24] T. T. Avrahami, L. Yau, L. Si, and J. Callan, "The fedlemur project: Federated search in the real world," *JASIST*, 2006.

[25] J. Bleiholder, S. Khuller, F. Naumann, L. Raschid, and Y. Wu, "Query planning in the presence of overlapping sources," in *EDBT*, 2006.

[26] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *VLDB*, 2006, pp. 918–929.

[27] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling up all pairs similarity search," in *WWW*, 2007, pp. 131–140.

[28] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *WWW*, 2008, pp. 131–140.

[29] S. E. Whang, M. David, and H. Garcia-Molina, "Pay-as-you-go entity resolution," *IEEE Trans. on Knowl. and Data Eng.*

[30] G. Costa, G. Manco, and R. Ortale, "An incremental clustering scheme for data de-duplication," *Data Min. Knowl. Discov.*, vol. 20, 2010.

[31] M. J. Welch, A. Sane, and C. Drome, "Fast and accurate incremental entity resolution relative to an entity knowledge base," in *CIKM '12*.

[32] A. Gruenheid, X. L. Dong, and D. Srivastava, "Incremental record linkage," in *VLDB*, 2014.

[33] E. Ioannou, W. Nejdl, C. Niederée, and Y. Velegrakis, "On-the-fly entity-aware query processing in the presence of linkage," *PVLDB*, 2010.