Evaluating Top-N Join Queries with Real-time Entity Resolution

Liang Zhu1*, Ye Cheng1, Yu Wang1*, Qin Ma1* and Weiyi Meng2

1Hebei University, Baoding, Hebei 071002, China
2State University of New York at Binghamton, Binghamton, NY 13902, USA
*Email: zhu@hbu.edu.cn; wy@hbu.edu.cn; maqin@hbu.edu.cn

Abstract. Using nonmonotone ranking functions in top-N queries is a challenge. Traditional techniques for top-N queries are based on clean data without entity resolution (ER). For dirty datasets with duplicate tuples referring to the same real-world entity, these techniques may yield top-N tuples duplicates for a query. Consequently, the effective size of the result set of the query is less than \( N \), and some useful tuples may fail to be retrieved from the datasets, which leads to poor effectiveness. Using an ER-Index based on a divide-and-conquer mechanism and nonmonotone ranking functions, in this paper, we propose a method for processing top-N join queries with real-time ER. This method integrates ER with the processing of a top-N join query over dirty datasets on the fly. Extensive experiments are conducted to measure the effectiveness and efficiency of the method over dirty datasets.

1. Introduction
A top-N (also top-k, kNN, ranking, or ranked) query against a relation/dataset is to find a sorted set of \( N \) (e.g., \( N = 10, 50, \) or 100) tuples that best, but not necessarily completely, satisfy the query condition. Top-N queries with approximate-match methods and ranking functions can handle some kinds of low-quality data (say, inaccurate, incomplete or erroneous data). Thus, top-N queries are more extensive, flexible, and powerful than traditional queries with exact-match paradigm, and are useful in many fields such as search engine, decision support system, information retrieval, and data mining.

The join operation is necessary for a query with Primary-Foreign key relationships or vertical fragmentations [13]. The processing and optimization of join queries have received significant attention in the field of information system.

Entity Resolution (ER) is the process of identifying tuples/records in one or multiple data sources that refer to the same real-world entity and then merging the matching duplicate tuples into a cluster or a tuple. An entity can be a person, a business, or any other object that exists in the real world. ER is also known as record linkage, data deduplication, merge-purge, and tuple matching [5, 11]. Accurate and fast ER is often needed in many fields such as statistics, artificial intelligence, information retrieval, information quality, databases, machine learning, and bank credit. ER is a challenging problem, especially for big data [8, 11], because people represent and misrepresent information about real-world entities in various ways and databases often do not contain unique entity identifiers across different sources [10, 15].

In general, ER techniques are classified into two categories: generic ER and real-time ER [1]. In 1946, Dunn used the term “record linkage” in the process of assembling a Book of life for each person in the world [9]. Surveys on this topic can be found in [6, 8]. The traditional techniques for generic ER and the strategies for data cleaning in data warehouse are offline and enormously expensive [17], and in general cannot be directly applied to query processing, which needs the real-time ER [2, 8]. Real-
time ER (also known as on-the-fly ER, query-time ER, or query-driven ER) is the process of matching a query record in sub-second time with records in a database that represent the same real-world entity [15], or in (near) real time, ideally within a few seconds at most [7]. In 2007, the first approach for query-time ER was proposed in [3].

Top-N queries have been studied extensively since late 1990’s [12, 14]. Most proposed techniques for top-N queries take into account a fixed query point \( Q \), e.g., \( Q = (1, 1, ..., 1) \) over a dataset \( R \subset [0, 1]^n \), and monotone ranking functions including linear functions; while few proposals deal with general functions as using nonmonotone ranking functions in top-N queries is a challenge [12]. Moreover, integrating real-time ER with top-N queries is one of the open directions for future investigation [2].

The contributions of this paper are summarized below: (1) For an arbitrary query point \( Q \) (in [15], or in (near) real time, ideally within a few seconds at most [7]).

For a top-N query \( Q_i \) over \( R \), (1 ≤ i ≤ s), where \( Q_i = (q_{i1}, ..., q_{im}) \) \( \in \mathbb{R}^m \), \( Q_2 = (q_{21}, ..., q_{2n}) \) \( \in \mathbb{R}^n \), and \( Q_s = (q_{s1}, ..., q_{sn}) \) \( \in \mathbb{R}^n \). We will define effective and efficient algorithm \( \mathcal{A} \) with \( \{Q_1, Q_2, ..., Q_s\} \) over \( \{R_1, R_2, ..., R_s\} \) to obtain the set \( T_{\mathcal{A}} \) of deduplicated top-N results of \( Q \), which is equivalent to an answer \( T \) of \( Q \) over \( R \), i.e., \( T_{\mathcal{A}} = T \).

2. Problem Definition

Consider a relation \( R(\text{tid}, U) \), and a set of \( s \) relations \( \{R_1(\text{tid}, U_1), R_2(\text{tid}, U_2), ..., R_r(\text{tid}, U_r)\} \), where \( U = (A_1, ..., A_d) \), \( U_1 = (A_{11}, ..., A_{1n}) \), \( U_2 = (A_{21}, ..., A_{2n}) \), ..., \( U_r = (A_{rn}, ..., A_{rn}) \), and \( U = U_1 \cup U_2 \cup ... \cup U_r \). That is, \( R \) is the natural join of \( \{R_1, R_2, ..., R_r\} \) with the primary key \( \text{tid} \) (tuple identifier) denoted by \( R = R_1 \bowtie R_2 \bowtie ... \bowtie R_r \) or \( \{R_1, R_2, ..., R_r\} \) is the schema decomposition of \( R \) and each \( R_i \) (1 ≤ i ≤ s) contains the primary key \( \text{tid} \) of \( R \) in a (distributed) database system [13]. Without loss of generality, let \( U_1 \cup U_2 \cup ... \cup U_r = \emptyset \), then \( d = m + n + ... + r \).

The relation/dataset \( R = \{t_1, t_2, ..., t_N\} \) is considered dirty if there exist at least two tuples \( t_i, t_j \in R \) that refer to the same real-world entity \( e \), that is, \( t[A_1, ..., A_d] = t[A_1, ..., A_d] \) but \( t[\text{tid}] \neq t[\text{tid}] \), hence \( t_i \) and \( t_j \) are duplicates (1 ≤ i, j ≤ |R|, i ≠ j), and they will be grouped into a cluster \( C \) to act as “one tuple” in this paper. An arbitrary tuple \( t \) in a nonempty cluster \( C \) will be chosen as the representative tuple of \( C \) (usually, it is the first one inserted into \( C \)), which refers to the entity \( e \). Note: the terms/phrases “dirty data, low-quality data, or poor-quality data” may have different meanings. More details about poor-quality data can be found in [18].

Let \( (\mathbb{R}^d, \rho(\cdot, \cdot)) \) be the real vector space with the Euclidean distance \( \rho(\cdot, \cdot) = L_2(\cdot, \cdot) \). \( C \subset \mathbb{R}^d \), its schema is \( R(\text{tid}, A_1, ..., A_d) \) with \( d \) attributes \( (A_1, ..., A_d) \) corresponding to \( \mathbb{R}^d = \mathbb{R}_1 \times ... \times \mathbb{R}_d \) where the \( i \)-th axis \( \mathbb{R}_i = \mathbb{R} \) for every \( i \), and \( R \) is stored generally as a base table in a relational database system. Our method can apply with impunity to any \( L_p \) distance function (1 ≤ p ≤ ∞), all of which are nonmonotone.

For a query point \( Q = (q_{11}, ..., q_{im}) \in \mathbb{R}^m \), a set \( T \subset R \), and a tuples \( t \in R \), we define \( \rho(t, Q) = \rho(t[A_1, ..., A_d], Q) \), the distance between \( T \) and \( Q \) is \( \rho(T, Q) = \min\{\rho(t, Q) : t \in T\} \). For a cluster \( C \) above, thus, \( \rho(C, Q) = \min\{\rho(t, Q) : t \in C\} = \rho(t, Q) \) for any \( t \in C \), and \( t \) is usually the representative tuple of \( C \).

For a dirty dataset \( R \), let \( \mathcal{C} = \{C_1, C_2, ..., C_N\} \subset \mathbb{R} \) be a sorted set of \( N \) clusters, that is, \( C_i \subset R \) and \( C_i \cap C_j = \emptyset \), \( \forall t_{i1} \in C_i, t_{j2} \in C_j(t_{i1}[A_1, ..., A_d] = t_{j2}[A_1, ..., A_d]) \), and \( \forall t_i \in C_i \forall t_j \in C_j(t_i[A_1, ..., A_d] \neq t_j[A_1, ..., A_d]) \) (1 ≤ i, j ≤ N, i ≠ j). The set \( \mathcal{C} \) is one of the answers to \( (Q, N) \) according to \( \rho(\cdot, \cdot) \) if and only if \( \rho(C_1, Q) \leq \rho(C_2, Q) \leq ... \leq \rho(C_N, Q) \) and \( \rho(C_N, Q) \leq \rho(t, Q) \) for arbitrary \( t \in R - (C_1 \cup C_2 \cup ... \cup C_N) \) (ties are broken arbitrarily).

For a top-N query \( Q = (q_{11}, ..., q_{im}) \in \mathbb{R}^m \), we decompose \( Q = (Q_1, Q_2, ..., Q_s) \), and \( Q_i \) is a top-K query over \( R_i \) (1 ≤ i ≤ s), where \( Q_i = (q_{i1}, ..., q_{im}) \in \mathbb{R}^m \), \( Q_2 = (q_{21}, ..., q_{2n}) \in \mathbb{R}^n \), and \( Q_s = (q_{s1}, ..., q_{sn}) \in \mathbb{R}^n \). We will define effective and efficient algorithm \( \mathcal{A} \) with \( \{Q_1, Q_2, ..., Q_s\} \) over \( \{R_1, R_2, ..., R_s\} \) to obtain the set \( T_{\mathcal{A}} \) of deduplicated top-N results of \( Q \), which is equivalent to an answer \( T \) of \( Q \) over \( R \), i.e., \( T_{\mathcal{A}} = T \), the two answers to \( (Q, N) \) obtained over \( R \) and \( \{R_1, R_2, ..., R_s\} \) respectively contain the same tuples except for the different tuple identifiers.
3. Algorithm

This work is a continuation of the work in [19], which only studied real-time ER, without top-N query processing. Using the structures and algorithms in [19], we create the ER-Index($\mathcal{R}_i$) for each $\mathcal{R}_i$.

Let $\mathcal{P}_0 = \mathcal{P} = \prod_{k=1}^{h}(a_j, b_j)$ be the region containing $\mathcal{R}_i$(tid, $\mathcal{U}_i$) (1 ≤ i ≤ s). We use the algorithm PRC in [19] to split $\mathcal{P}_0$ into q cells ($\mathcal{P}_k$, k = 1, ..., q), and provide a complete and disjoint partitioning of $\mathcal{P}_0$, that is, $\mathcal{P}_0 = \mathcal{P}_1 \cup \cdots \mathcal{P}_q$, and $\mathcal{P}_k \cap \mathcal{P}_l = \emptyset$ for 1 ≤ k ≠ l ≤ q. The algorithm PRC will be called recursively in the process of creating the ER-index. Each $\mathcal{P}_k$ is a cell/region with the form $\prod_{k=1}^{h}(c_k, d_k)$, where h is the dimensionality of $\mathcal{R}_i$, the child node is the sub-region of a cell. The details are described in [19]. In our experiments, each non-leaf cell/region is split into q = 12 child cells. Let $\mathcal{A} \in \mathcal{U}_i$ be an attribute with the maximum number of distinct values $\{t[A] \mid t \in \mathcal{R}_i\}$. The attribute $\mathcal{A}$ is utilized to generate a sorted list with non-decreasing $\mathcal{A}$-values in the ER-Index, and $\mathcal{A}$ is called the sorted-attribute. Thus, each leaf cell points to a sorted list with its sorted-attribute. Without loss of generality, let the sorted-attribute $\mathcal{A}$ be the first attribute $\mathcal{A}_1 \in \mathcal{U}_i$. We denote the dimensionality of $\mathcal{R}_i$ by $d[i]$ in our algorithm. Thus, for a query $Q$ with the decomposition $Q = (Q_1, Q_2, ..., Q_s)$, we have $Q_i = (q_{i1}, ..., q_{i\mathcal{A}})$ (1 ≤ i ≤ s).

The main idea of $\text{TupleJoin}$ is as follows: we retrieve the top-$K$ tuple of $Q_i$ over $\mathcal{R}_i$, one by one, and join these top-$K$ tuples for all $Q_i$ (i = 1, 2, ..., s), then obtain the top-$N$ tuples of $Q$ from the joined tuples.

Algorithm $\text{TupleJoin}$

Input: ($Q$, $N$), ($\mathcal{R}_1, \mathcal{R}_2, ..., \mathcal{R}_s$); // $Q$ with dimensionality $d[1] + \cdots + d[s]$
Output: $\mathcal{C}$; //a sorted set of N tuples in $\mathcal{R} \bowtie \mathcal{R} \bowtie \cdots \bowtie \mathcal{R}$, with distances $\{p(t, Q)\}$

1. Let $Y = Q$; // $Y$ is the threshold point, and its initialization is $Q$
2. Let $r = 0$; // $r$ is the search radius for sorted $\mathcal{A}$-value
3. for i = 1 to s //creation and initialization of structures
4. Create two lists $\text{LeafList}$ and $\text{CandidateList}$;
5. Put all pointers of all leaf nodes of ER-Index($\mathcal{R}_i$) into $\text{LeafList}$;
6. Let $\text{CandidateList} = \emptyset$;
7. end for
8. while (TRUE)
9. for i = 1 to s //access in parallel to each $\mathcal{R}_i$ with $Q_i$ for i = 1, ..., s
10. $t_i[\mathcal{A}_i] = \text{SearchCandidate}(Q_i, \mathcal{R}_i)$; //get the top-$K$ tuple of $Q_i$
11. $Y_{q_{i1}} = t_i[\mathcal{A}_i]$; //join operations in $B$
12. $\tau = p(Y, Q)$; //update $\tau$, one of two thresholds
13. if ($\exists$ a complete joined tuple $u$ in $B$)
14. Remove $u$ from $B$;
15. if ($u$ is not a duplicate in $\mathcal{C}$) //we deal with ER in $\mathcal{C}$
16. $\text{dis} = p(u, Q)$;
17. if ($\text{dis} \leq \gamma$ OR $|\mathcal{C}| < N$)
18. Insert $u$ into $\mathcal{C}$ in ascending order by $\text{dis}$;
19. $\gamma = \max\{p(u, Q) \mid u \in \mathcal{C}\}$; //update $\gamma$ such that $\gamma = p(t, Q)$
20. end if
21. if ($\gamma \leq \eta$ AND $\gamma \leq \tau$ AND $|\mathcal{C}| \geq N$ OR (all of $\{\mathcal{R}_i\}$ are exhausted)
22. Halt; // the top-$N$ tuples of $Q$ are obtained.
23. end if
24. end if
25. else if ($\exists$ a tuple $x$ in $B$ can join with $t_i[\mathcal{A}_i]$ but $x \bowtie t_i[\mathcal{A}_i]$ is not complete)
26. $x_{q_{i0}} = t_i[\mathcal{A}_i];$ with $j = 1, 2, ..., d[i]$;
27. Insert $x$ into $B$ in descending order by $p(x, Q)$;
28. $\eta = \min\{p(x, Q) \mid x \in B\}$; //update $\eta$, the other of two thresholds
29. else if ($\exists$ a tuple $x$ in $B$ can join with $t_i[\mathcal{A}_i]$)
30. $x = Q$;
31. $x_{q_{i0}} = t_i[\mathcal{A}_i];$ with $j = 1, 2, ..., d[i]$;
32. Insert $x$ into $B$ in descending order by $p(x, Q)$;
33. $\eta = \min\{p(x, Q) \mid x \in B\}$; //update $\eta$, the other of two thresholds
34. end if
35. end if
36. end for
37. end while
38. return $\mathcal{C}$;
In *TupleJoin*, we access in parallel each of the *s* datasets \( \{R_1, R_2, \ldots, R_s\} \), obtain a top-*K* tuple for each \( Q \), over \( R \), each time the function *SearchCandidate* is called, perform join operations of these top-*K* tuples for all \( Q, (i = 1, 2, \ldots, s) \) in the buffer \( B \), cluster and deduplicate complete joined tuple in \( C \) until the top-*N* tuples of \( Q \) are obtained, which satisfy the Halt condition “\((γ ≤ η AND γ ≤ τ AND |C| ≥ N) OR \((\{R_i\} are all exhausted)\)” in Line 22. We define *SearchCandidate* below.

**Function SearchCandidate**(\( Q, R \))

1. if (CandidateList = = null)  //locate \( Q \) for the first access to the dataset \( R \)
2. Find the qNode in LeafList such that \( Q = q\)Node by using ER-Index(\( R \));
3. if (qNode = = null)  //no leaf node contains \( Q \)
\hspace{1em} Let \( q = (c_1, \ldots, c_N) \) be the center of each Node in LeafList;
4. Find a Node in LeafList with minimum \( ρ(q, Q) \) and minimum \( |t_i - q_i| \), and then
\hspace{1em} qNode = Node;
5. end if
6. Remove qNode and List from LeafList;
7. Insert qNode and List into CandidateList;
8. end if
9. \( (\text{MinRadius}, \text{minTpl}) = \text{getMin}(\text{CandidateList}, \{Q\}) \);  //get two fields of a structure
10. \( (\text{MinRadius}, \text{minTpl}) = \text{rRange}(\text{MinRadius}); \)         //get two fields of a structure
11. \( \text{return} \min\text{Tpl}; \)  //the top-*K* tuple of \( Q \)

The function *SearchCandidate* will utilize some structures to obtain the top-*K* tuples one by one over a single dataset \( R(tid, A_1, \ldots, A_h) \) for a query \( Q = (q_1, \ldots, q_h) \in \mathbb{R}^h \), where \( h = m, n, \ldots, r, \) or \( d \). For “CandidateList = = null” in Line 1, we need to locate \( Q \) by using ER-Index(\( R \)), and then retrieve a top-*K* tuple for \( Q = (q_1, \ldots, q_h) \) each time the two functions *getMin* and *rRange* are called.

**Function getMin**(CandidateList, \( Q \))

1. for each List in CandidateList  //\( a_i = \text{List}[A_i], a_R = \text{List}.\text{Tail}[A_i] \)
2. if List is new
3. \hspace{1em} Locate \( Q, q_i \) in each List in CandidateList by Binary Search;
4. \hspace{1em} Get one or two new pointers of the List; //there is only one pointer if \( Q, q_i \notin (a_i, a_R) \)
5. \hspace{1em} end if
6. for each new or moved pointer
7. \hspace{1em} Get one neighbor of \( Q \) by sorted \( A_i \)-value in the List;
8. \hspace{1em} Insert the nearest neighbor \( v \) into a buffer \( G \);
9. end for
10. end for
11. Let \( r = \min\{|v[A_i] - q_i|: v \in G\} \);  //\( r \) is the search radius about \( A_i \)-value
12. Remove the neighbor \( v_i \) with the minimum \( r \) from \( G \);
13. Move the pointer of \( v_i \) up or down if the List is not exhausted;
14. Get \( T = \{t = (t_1, \ldots, t_h) \in R: t_i = q_i - r \) or \( t_i = q_i + r \}; \)
15. \( \min\text{Tpl} = \text{t}_0 = \text{min}\{p(t, Q): t \in T\}; \)  //get top-*K* tuple and the distance \( p(t_0, Q) \)
16. Remember \( i \) with \( p(t_0, Q) \); \( \text{t}_0 = \text{List}[i]; \)
17. Remove \( t_0 \) from \( T \);
18. \( \text{return} (r, \min\text{Tpl}); \)  //\( (r, \min\text{Tpl}) \) are two fields of a structure

A top-*K* tuple \( t_0 \) obtained by the function *getMin* is the local one, only a candidate top-*K* tuple, may or may not be the genuine one at this time. In order to obtain the genuine top-*K* tuple, we will use the function *rRange* as described below.

**Function rRange**(MinRadius)

1. Let \( r = \text{MinRadius}; \)
2. Get the range \( [q_i - r, q_i + r] \) for the sorted attribute \( A_i \);
3. for each cell \( \mathcal{P} \) with the form \( \prod_{i=1}^h (c_i, d_i) \) in the structure LeafList
\hspace{1em} if \( \mathcal{P} \) intersects the region \([q_i - r, q_i + r] \times \prod_{i=1}^h (-\infty, \infty) \)
\hspace{2em} Remove the Node from LeafList and put it into CandidateList;
4. end if
5. end for
6. \( (\text{MinRadius}, \text{minTpl}) = \text{getMin}(\text{CandidateList}, \{Q\}); \)
7. \( \text{return} (\text{MinRadius}, \text{minTpl}); \)  //\( \min\text{Tpl} \) is the top-*K* tuple of \( Q \) in \( R \)

The function *rRange* will find all candidate cells \( \{\mathcal{P}\} \) that may contain the genuine top-*K* tuple \( t \) for \( Q = (q_1, \ldots, q_h) \) with the distance \( p(t, Q) < p(t_0, Q) \), and call *getMin* again to get \( t \).
4. Experimental Results

Our experiments are carried out by using Microsoft's SQL Server 2014, Microsoft's VC++ 2015 and Windows 10 on a PC with Intel(R) Core(TM) i7-6700 CPU@3.40GHz 3.41GHz, and 16 GB memory.

Two original datasets Cover54D [4] and Sift128D [16] are used to generate eight relation schemas C4D, C6D, S6D, S9D, S10D, S15D, S20D and S30D with 4-30 dimensions, in which suffix “d” indicates that the dataset has d dimensions, i.e., d attributes (A₁, …, Aₙ) in addition to the tuple identifier tid. We generate C4D = πₜid, A₁, A₂, A₃, A₄, Covers54D, C6D = πₜid, A₁, …, A₆, Covers54D, and SdD = πₜid, A₁, …, Aₙ, Sift128D (d = 6, 9, 10, 15, 20, and 30). Each of the eight datasets contains 100,000 tuples corresponding the first 100,000 tuples in Cover54D or Sift128D.

For each of the eight datasets R(tid, U) with U = (A₁, …, Aₙ), it is decomposed vertically into two or three relations with tid, i.e., {Rₜ(tid, U₁), Rₜ(tid, U₂)} or {Rₜ(tid, V₁), Rₜ(tid, V₂), Rₜ(tid, V₃)}, where U = U₁∪U₂ = V₁∪V₂∪V₃, U₁∩U₂ = V₁∩V₂ ∩V₃ = ∅, and |U₁| = |U₂| = |U₃| = d/3. Obviously, each Rᵢ (1 ≤ i ≤ 2 or 3) has 100,000 tuples. For example, S6D(tid, A₁, …, A₆) is decomposed vertically into {Rₜ(tid, A₁, A₂, A₃), Rₜ(tid, A₄, A₅, A₆)}; while C6D(tid, A₁, …, A₆) is decomposed into {Rₜ(tid, A₁, A₂, A₃), Rₜ(tid, A₄, A₅, A₆), Rₜ(tid, A₆, A₇)}.

For each dataset, the workload includes 100 queries that are the tuples randomly selected from the respective d-dimensional dataset (4 ≤ d ≤ 30), while N = 100 for top-N queries against all datasets.

For a given query Q = (q₁, …, qₖ)∈Rᵈ, a Naïve algorithm over a single dataset R will be utilized as a baseline to show the effectiveness of TupleJoin. The Naïve algorithm is described as follows: it retrieves all tuple t∈R, computes the distance ρ(t, Q) between each tuple t and the query Q, ranks distances, deduplicates/clusters the tuples, and then returns the top-N tuples with distances. We will not report the above measures of Naïve over a single dataset to compare that of TupleJoin over multiple datasets.

4.1. The Effectiveness of Algorithm

For each of the eight datasets, (1) we use Naïve method and our Function SearchCandidate(Q, R), respectively, to get the sorted sets T₁ and T₂ of the top-N (N = 100) tuples for each query in a workload over R (= R₁×R₂ or = R₁×R₂×R₃), (2) TupleJoin is used to obtain a sorted set T of top-N tuples for the same workload over {R₁, R₂} or {R₁, R₂, R₃}, (3) By comparing the top-N results of each query, we show that T₁, T₂, and T are equivalent, i.e., T₁ ≡ T₂ ≡ T.

Table 1. The Number of Seen Tuples and the Elapsed Time over Two Datasets.

|       | C4D | S6D | S10D | S20D |
|-------|-----|-----|------|------|
| #T   | 6786| 9139| 7404 | 5699 |
| Time (ms) | 3741 | 7332 | 8624 | 8673 |

Table 2. The Number of Seen Tuples and the Elapsed Time over three Datasets.

|       | C6D | S9D | S15D | S30D |
|-------|-----|-----|------|------|
| #T   | 9187| 11112| 10326| 7895 |
| Time (ms) | 8889 | 8943 | 8299 | 8624 |

4.2. The Number of Seen Tuples and the Elapsed Time

For a query Q = (q₁, …, qₖ)∈Rᵈ, by using TupleJoin, Table 1 and Table 2 show the average numbers of seen tuples (denoted by #T) and the average elapsed time (millisecond, ms) to find its top-N tuples from the respective dataset. In the case of a dataset with two join relations {R₁, R₂} in Table 1, 5410 ≤ #T ≤ 9139. In the situation of three relations {R₁, R₂, R₃} in Table 2, 7895 ≤ #T ≤ 11112.

The average elapsed time is between 2546ms and 3741ms over two join relations, and from 7332ms to 8624ms over three join relations. the elapsed times over two join relations are much less than that over three ones, because the numbers of seen tuples, join operations and candidate tuples over two join relations are much smaller than that over three ones. For an algorithm, generally, the number of seen tuples has a direct impact on the elapsed time for evaluating top-N queries.
5. Conclusions
For processing top-N join query model integrated with real-time Entity Resolution (ER), we proposed a method \textit{TupleJoin} by using an ER-Index based on a divide-and-conquer mechanism. Using various datasets, we carried out extensive experiments to measure the effectiveness and efficiency of this method. The sorted sets of top-N tuples of queries by \textit{TupleJoin} over join relations \{\textit{R} \_1, \textit{R} \_2\} or \{\textit{R} \_1, \textit{R} \_2, \textit{R} \_3\} are equivalent to the results by Naïve method over \textit{R} = \textit{R} \_1 \bowtie \textit{R} \_2 or \textit{R} = \textit{R} \_1 \bowtie \textit{R} \_2 \bowtie \textit{R} \_3, which showed the effectiveness of our proposed method. Using \textit{TupleJoin} for various dimensional datasets, the average elapse times of processing a top-100 join query are less than 4 seconds over two join relations, and are smaller than 9 seconds over three join relations.

6. Acknowledgments
This work is supported in part by NSF of Hebei Province (F2017201208) and an NSF grant IIS-1546441.

7. References
[1] Altwaijry H, Kalashnikov D V and Mehrotra S 2013 Query driven approach to entity resolution \textit{Proc. VLDB Endowment} \textit{6} 1846–57
[2] Altwaijry H, Mehrotra S and Kalashnikov D V 2015 QuERy: a framework for integrating entity resolution with query processing \textit{Proc. VLDB Endowment} \textit{9} 120–31
[3] Bhattacharya I and Getoor L 2007 Query-time entity resolution \textit{J. Artif. Intell. Res} \textit{30} 621–57
[4] Blake C and Merz C 1998 UCI repository of machine learning databases
[5] Burdick D, Fagin R, Kolaitis P, Popa L and Tan W 2015 A declarative framework for linking entities \textit{In: 18th Int. Conf. Database Theory} pp 25–43
[6] Christen P 2012 A survey of indexing techniques for scalable record linkage and deduplication \textit{IEEE Trans. Knowl. Data Eng.} \textit{24} 1537–55
[7] Christen P 2012 \textit{Data matching: concepts and techniques for record linkage, entity resolution, and duplicate detection} (Berlin, Heidelberg: Springer)
[8] Christophides V, Ethymiou V, Palpanas T, Papadakis G and Stefanidis K 2019 End-to-end entity resolution for big data: a survey \textit{arXiv: 1905.06397v2}
[9] Dunn H L 1946 Record linkage \textit{Am J Public Health Nations Health} \textit{36} 180–83
[10] Firmani D, Saha B and Srivastava D 2016 Online entity resolution using an oracle \textit{Proc. VLDB Endowment} \textit{9} 384–95
[11] Getoor L and Machanavajjhala A 2013 Entity resolution for big data \textit{In: Int. Conf. Knowledge Discovery & Data Mining} http://www.umiacs.umd.edu/~getoor/Tutorials/ER_KDD2013.pdf
[12] Ilyas I, Beskales G, and Soliman M A 2008 A survey of top-k query processing techniques in relational database systems \textit{ACM Comput. Surv} \textit{40} 1–58
[13] Özsu MT and Valduriez P 2011 \textit{Principles of Distributed Database Systems 3rd edn} (New York NY: Springer)
[14] Qiao B, Hu B, Qiao X, Zhu J and Wu, G 2020 A top-k spatial join querying processing algorithm based on spark \textit{Information Systems} \textit{87} 101419
[15] Ramadan B and Christen P 2014 Forest-based dynamic sorted neighborhood indexing for real-time entity resolution \textit{In: 23rd Int. Conf. Information and Knowledge Management} pp 1787–90
[16] Singh V and Singh A K 2012 SIMP: accurate and efficient near neighbor search in high dimensional spaces \textit{In: Proc.15th Int. Conf. Extending Database Technol.} pp 492-503
[17] Sismanis Y, Wang L, Fuxman A P, Haas J and Reinwald, B 2009 Resolution-aware query answering for business intelligence \textit{In: Proc. 25th Int. Conf. Data Eng.} pp 976-87
[18] Wang R Y, Pierce E M, Madnick S and Fisher C W 2015 \textit{Information Quality} (New York: Routledge Taylor & Francis Group)
[19] Zhu L, Lu W, Ma Q and Meng, W 2018 Region-tree Based Sorted List Indexing for Real-time Entity Resolution in n-Dimensional Data Space \textit{IOP Conference Series: Materials Science and Engineering} \textit{466}