The VLDB Journal manuscript No.
(will be inserted by the editor)

Joint Entity Resolution on Multiple Datasets

Steven Euijong Whang · Hector Garcia-Molina

Received: date / Accepted: date

Abstract Entity resolution (ER) is the problem of identifying which records in a database represent the same entity. Often, records of different types are involved (e.g., authors, publications, institutions, venues), and resolving records of one type can impact the resolution of other types of records. In this paper we propose a flexible, modular resolution framework where existing ER algorithms developed for a given record type can be plugged in and used in concert with other ER algorithms. Our approach also makes it possible to run ER on subsets of similar records at a time, important when the full data is too large to resolve together. We study the scheduling and coordination of the individual ER algorithms, in order to resolve the full data set, and show the scalability of our approach. We also introduce a "state-based" training technique where each ER algorithm is trained for the particular execution context (relative to other types of records) where it will be used.

Keywords Entity resolution · Joint entity resolution · Physical execution · Influence graph · Execution plan · Expander function · State-based training · Data cleaning

1 Introduction

Entity Resolution (ER) (also referred to as deduplication) is the process of identifying and merging records judged to represent the same real-world entity. For example, two companies that merge may want to combine their customer records: for a given customer that dealt with the two companies they create a composite record that combines the known information.

In practice, many data integration tasks need to jointly resolve multiple datasets of different entity types together. Since relationships between the datasets exist, the result of resolving one dataset may benefit the resolution of another dataset. For example, the fact that two (apparently different) authors a_1 and a_2 have published the same papers could be strong evidence that the two authors are in fact the same. Also, by identifying the two authors to be the same, we can further deduce that two papers p_1 and p_2 written by a_1 and a_2, respectively, are more likely to be the same paper as well. This reasoning can easily be extended to more than two datasets. For example, resolving p_1 and p_2 can now help us resolve the two corresponding venues v_1 and v_2. Compared to resolving each dataset separately, joint ER can achieve better accuracy by exploiting the relationships between datasets.

Among the existing works on joint ER [12,6,10,22, 27,2], few have focused on scalability, which is crucial in resolving large data (e.g., hundreds of millions of people records crawled from the Web). The solutions that do address scalability propose custom joint ER algorithms for efficiently resolving records. However, given that there exists ER algorithms that are optimized for specific types of records (e.g., there could be an ER algorithm that specializes in resolving authors only and another ER algorithm that is good at resolving venues), replacing all the ER algorithms with a single joint ER algorithm that customizes to all types of records may be challenging for the application developer. Instead, we propose a flexible framework for joint ER where

S. E. Whang
Google Research, Mountain View, CA 94043
E-mail: swhang@google.com

H. Garcia-Molina
Stanford University Computer Science Department, Stanford, CA 94305
E-mail: hector@cs.stanford.edu

Paper	Title	Venue
p_1	The Theory of Joins in Relational Databases	v_1
p_2	Efficient Optimization of a Class of Relational Expressions	v_1
p_3	The Theory of Joins in Relational Databases	v_2
p_4	Optimizing Joins in a Map-Reduce Environment	v_3

Venue	Name	Papers
v_1	ACM TODS	$\{p_1, p_2\}$
v_2	ACM Trans. Database Syst.	$\{p_3\}$
v_3	EDBT	$\{p_4\}$

Table 1 Papers and Venues

one can simply "plug in" her existing ER algorithms, and our framework can then schedule and resolve the datasets individually using the given ER algorithms.

While many previous joint ER works assume that all the datasets are resolved at the same time in memory using one processor, our framework allows efficient resource management by resolving a few datasets at a time in memory using multiple processors. Our framework extends an ER technique called blocking [20] where the entire data of one type is resolved in small subsets or blocks. In addition, our framework resolves multiples types of data and divides the resolution based on the data type. Our approach may especially be useful when there are many large datasets that cannot be resolved altogether in memory. Thus one of the key challenges is determining a good sequence for resolution. For instance, should we resolve all venues first, and then all papers and then all authors? Or should we consider a different order? Or should we resolve some venues first, then some related papers and authors, and then return to resolve more venues? And we may also have to resolve a type of record multiple times, since subsequent resolutions may impact the work we did initially.

As a motivating example, consider two datasets P and V (see Table 1) that contain paper records and venue records, respectively. (Note that Table 1 is a simple example used for illustration. In practice, the datasets can be much larger and more complex.) The P dataset has the attributes Title and Venue while V has the attributes Name and Papers. For example, P contains record p_1 that has the title "The Theory of Joins in Relational Databases" presented at the venue v_1. The Papers field of a V record contains a set of paper records because one venue typically has more than one paper presented.

Suppose that two papers are considered the same and are clustered if their titles and venues are similar while two venues are clustered if their names and papers are similar. Say that we resolve the paper records first. Since p_1 and p_3 have the exact same title, p_1 and p_3 are considered the same paper. We then resolve the venue records. Since the names of v_1 and v_2 are significantly different, they cannot match based on name similarity alone. Luckily, using the information that p_1 and p_3 are the same paper, we can infer that v_1 and v_2 are most likely the same venue. (In fact "ACM TODS" and "ACM Trans. Database Syst." stand for the same journal.) We can then re-resolve the papers in case there are newly matching records. This time, however, none of the papers match because of their different titles. Hence, we have arrived at a joint ER result where P was resolved into the partition $\{\{p_1, p_3\}, \{p_2\}, \{p_4\}\}$ while V was resolved into $\{\{v_1, v_2\}, \{v_3\}\}$. Notice that we have followed the sequence of resolving papers, venues, then papers again.

Given enough resources, we can improve the runtime performance by exploiting parallelism and minimizing unnecessary record comparisons. For example, if we have two processors, then we can resolve the papers and venues concurrently. As a result, p_1 and p_2 match with each other. After the papers and venues are resolved, we resolve the venues again, but only perform the incremental work. In our example, since p_1 and p_2 matched in the previous step, and p_1 was published in the venue v_1 while p_2 was published in the venue v_2, we only need to check if v_1 and v_2 are the same venue and can skip any comparison involving v_3. Notice that the papers do not have to be resolved at the same time because none of the venues merged in the previous step. However, after v_1 and v_2 are identified as the same venue, we resolve the papers once more. Again, we only perform the incremental work necessary where we resolve the three records p_1, p_2, and p_3 (because v_1 and v_2 matched in the previous step), but not p_4. In total, we have concurrently resolved the papers and venues, then incrementally resolved the venues, and then incrementally resolved the papers. If the incremental work is much smaller than resolving a dataset from the beginning, the total runtime may improve.

In the case where the same dataset is resolved multiple times, an interesting question to ask is whether we should use the exact same ER algorithm for resolving that dataset again. For example, the paper dataset above was resolved twice: once before the venues were resolved and once after. Given that the venues are resolved, we may want to "re-train" the ER algorithm for that context. For instance, we may want to weight venue similarity more (relative to paper title similarity) now that venues have been resolved. Thus, in our paper we propose a state-based training method that trains

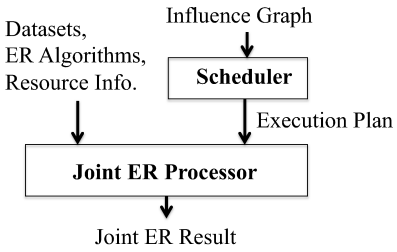

Fig. 1 System Architecture

an ER algorithm based on the current state of the other datasets being resolved.

In summary, we make the following contributions:

- We present a modular joint ER framework, where existing ER algorithms, tuned to a particular type of records, can be effectively used. We define the physical executions of multiple ER algorithms that produce joint ER results (Section 2).
- We introduce the concept of a scheduler, whose output (a logical execution plan) specifies the order for datasets to be resolved in order to produce correct joint ER results (Section 3).
- We show how the joint ER processor uses an execution plan to physically execute the ER algorithms and return a joint ER result while satisfying resource constraints (Section 4).
- We propose a state-based training method that uses the current state of other datasets for fine tuning joint ER algorithms (Section 5).
- We experiment on both synthetic and real data to demonstrate the behavior and scalability of joint ER. We then use real data to explore the accuracy of state-based training (Section 6).

2 Framework

In this section, we define the framework for joint entity resolution. Figure 1 shows the overall architecture of our system. Given an influence graph (defined in Section 3.1), a scheduler constructs a logical "execution plan," which specifies a high-level order for resolving datasets using the ER algorithms. Next, given the physical resource constraints (e.g., memory size and number of CPU cores), the joint ER processor uses the execution plan to "physically execute" the ER algorithms on the given datasets using the available resources to produce a joint ER result.

In the following sections, we define a general model for ER enabling a large class of ER algorithms that resolve a single entity type dataset to fit in our framework. We then formalize joint ER using the ER algorithms. Finally, we formalize physical executions of ER algorithms that can return joint ER results.

2.1 ER Model

Each *record* r is of one entity type. For example, a record can represent a paper or a venue. A *dataset* R contains records of the same type. Thus we may have a dataset for papers and a dataset for venues. We do not assume that datasets follow the relational or any other data model. There may also be more than one dataset containing records of the same entity type. For example, we could have two datasets containing paper records. As a result, we also allow one dataset to be split into multiple datasets. For instance, a common technique to resolve a large set of records R is to split it into smaller sets or blocks. In this case, each block is viewed as one dataset. The entire collection of datasets is denoted as the set $\mathcal{D}$.

We also assume that a record of one entity type may *refer to* another record of a different type. For example, if a paper p was published in a venue v, then p may refer to v by containing a pointer to v. (Further details on references can be found in Section 4.4.) We define a cluster of records c to be a set of records. A partition of the dataset R is defined as a set of clusters $P = \{c_1, \ldots, c_m\}$ where the following properties hold: $c_1 \cup \ldots \cup c_m = R$ and $\forall c_i, c_j \in P$ where $i \neq j$, $c_i \cap c_j = \emptyset$.

An ER algorithm E_R resolves a single dataset R. Given a dataset $R = \{r_1, \ldots, r_n\}$, E_R receives as input a partition P_R of R and returns another partition P'_R of R. A partition is a general way to represent the output of an ER algorithm because the clusters provide the lineage information on which records end up representing the same entity. In addition, one could optionally merge the records within the same cluster to produce composite records. We also require the input of ER to be a partition of R so that we may also run ER on the output of a previous ER result. In our motivating example in Section 1, the input for papers was a set of records $R = \{p_1, p_2, p_3\}$, which can be viewed as a partition of singletons $P_R = \{\{p_1\}, \{p_2\}, \{p_3\}\}$, and the output was the partition $P'_R = \{\{p_1, p_3\}, \{p_2\}\}$. We say that an ER algorithm *resolves* R if we run ER on a partition of R to produce another partition of R. Constructing ER algorithms is a traditional topic that has been extensively studied in the literature [9,13,34].

While we have defined an ER algorithm to resolve exactly one dataset, we can easily extend an ER algorithm to resolve multiple datasets as well. For example, say we have data sets R, S, T and U. Suppose that algorithm E_{RS} resolves both R and S at the same time,

while T and U have their own algorithms. We can easily treat data sets R and S as a single data set, where each record in this set has a special field indicating if it actually belongs to R or to S. A collective ER algorithm that jointly resolves multiple sets (e.g., [24]) can be handled as one of these multi-set ER algorithms.

During the resolution of R, E_R may use the information of records in other datasets that the records in R refer to. For example, if paper records refer to author records, we may want to use the information on which authors are the same when resolving the papers. To make the potential dependence on other data sets explicit, we write the invocation of E_R on P_R as $E_R(P_R, \mathcal{P}_{-R})$ where $\mathcal{P}_{-R} = \{P_Y | Y \in \mathcal{D} - \{R\}\}$ represents the partitions of the remaining datasets. For example, if $\mathcal{D} = \{R, S\}$, and the partitions of R and S are P_R and P_S, respectively, then $\mathcal{P}_{-R} = \{P_S\}$ and running E_R on P_R returns the result of $E_R(P_R, \mathcal{P}_{-R})$. Notice that the records in R may not refer to records in all the datasets in $\mathcal{P}_{-R}$.

We now define a valid ER algorithm that resolves a dataset.

Definition 1 Given any input partition P_R of a set of records R, a *valid* ER algorithm E_R returns an ER result $P'_R = E_R(P_R, \mathcal{P}_{-R})$ that satisfies the two conditions:

1. P'_R is a partition of R
2. $E_R(P'_R, \mathcal{P}_{-R}) = P'_R$

The first condition says that E_R returns a partition P'_R of R. The second condition requires that the output is a fixed point result where applying E_R on P'_R will not change the result further unless the partitions in $\mathcal{P}_{-R}$ change. For example, say that there are two datasets $R = \{r_1, r_2, r_3\}$ and $S = \{s_1, s_2\}$ where $P_R = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and $P_S = \{\{s_1\}, \{s_2\}\}$. Say that E_R is a valid algorithm where $E_R(P_R, \{P_S\}) = \{\{r_1, r_2\}, \{r_3\}\} = P'_R$. Then we know by the second condition of Definition 1 that $E_R(P'_R, \{P_S\})$ returns P'_R as well. Notice that a valid ER algorithm is not required to be deterministic, and the ER result does not have to be unique.

In the case where a single dataset R is too large to fit in memory, we can use blocking techniques where R is split into distinct blocks $R_1, R_2, \ldots, R_k$. Typically, each R_i is small enough to fit in memory. Only the records with the same block are compared assuming that records in different blocks are unlikely to match. For example, we might partition a set of people records according to the zip codes in address fields. We then only need to compare the records with the same zip code. We thus treat each R_i as a separate dataset.

2.2 Joint ER Model

Using valid ER algorithms, we can define a joint ER result on all the datasets $\mathcal{D}$ as follows.

Definition 2 A *joint ER* result on $\mathcal{D}$ is the set of partitions $\{P_R | R \in \mathcal{D}$, P_R is a partition of R, and $E_R(P_R, \mathcal{P}_{-R}) = P_R\}$.

A joint ER result is thus a fixed point: running any additional ER on any dataset does not change the results. Continuing our example from Definition 1, say we have the partitions $P'_R = \{\{r_1, r_2\}, \{r_3\}\}$ and $P'_S = \{\{s_1, s_2\}\}$. In addition, say that $E_R(P'_R, \{P'_S\}) = P'_R$, and $E_R(P'_S, \{P'_R\}) = P'_S$. Then according to Definition 2, $\{P'_R, P'_S\}$ is a joint ER result of R and S. Notice that a joint ER result is not necessarily unique because even a single dataset may have multiple ER results satisfying Definition 1.

2.3 Physical Execution

In our framework, we assume the datasets are resolved in parallel in synchronous steps. At each step, some datasets are resolved using valid ER algorithms while other datasets are left unchanged. As we will see in the example below, a resolution in a given step refers to the previous state of the datasets.

For example, Figure 2 pictorially shows a physical execution of three datasets R, S, and T where we first resolve R and S concurrently then S and T sequentially. (For now ignore the (i, j) annotations.) At Step 0, no resolution occurs, but each dataset $X \in \mathcal{D}$ is initialized as the partition $P^0_X = \{\{r\} | r \in X\}$. In our example, we initialize P^0_R, P^0_S, and P^0_T. After n steps of synchronous transitions, a partition of dataset X is denoted as P^n_X. We denote the entire set of partitions after Step n except for P^n_R as $\mathcal{P}^n_{-R} = \{P^n_Y | Y \in \mathcal{D} - \{R\}\}$ (e.g., $\mathcal{P}^1_{-R}$ in Figure 2 is $\{P^1_S, P^1_T\}$). There are two options for advancing each partition P^n_X into its next step partition P^{n+1}_X. First, we can simply set P^{n+1}_X to P^n_X without change, which we pictorially express as a double line from P^n_X to P^{n+1}_X. For example, we do not run ER on P^0_T during Step 1 and thus draw a double line from P^0_T to P^1_T below. Second, we can run ER on P^n_X using the information in the other partitions $\mathcal{P}^n_{-X}$. We pictorially express the flow of information as arrows from P^n_X and the partitions in $\mathcal{P}^n_{-X}$ to P^{n+1}_X. For instance, producing P^2_S using ER may require the information of all the previous-step partitions, so we draw three arrows from P^1_R, P^1_S, and P^1_T to P^2_S. Notice that we do not allow ER to use the information of partitions in more than 1 step behind because in general it is more helpful to use the most recent partition information possible.

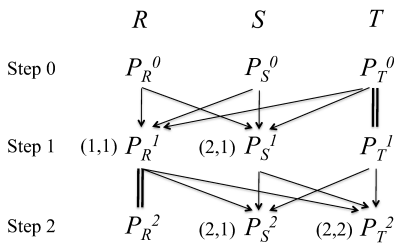

Fig. 2 The Physical Execution (((R), (S)),((),(S, T)))

The datasets resolved in the same step can be resolved in parallel by several machines. After each step, a synchronization occurs where datasets are re-distributed to different machines for the next step of processing. (In Section 4, we elaborate on how datasets are distributed to machines.) We call this step-wise sequence of resolutions a *physical execution*. For each dataset R being resolved, we also specify the machine number m resolving R and the execution order o of R by m during the current step as (m, o). For example, in Step 2, the dataset S is the 1^{st} dataset to be resolved by machine 2.

The physical execution information can be compactly expressed as a nested list of three levels where the outermost level specifies the steps, the middle level the machine order, and the inner-most level the order of datasets resolved within a single machine. Our physical execution can thus be represented as $(((R),(S)),((),(S,T)))$. Given a set of initial partitions $\{P_X^0|X \in \mathcal{D}\}$, a physical execution $\mathcal{T}$ produces the partitions $\{P_X^{|\mathcal{T}|}|X \in \mathcal{D}\}$ where $|\mathcal{T}|$ is the number of synchronous steps within $\mathcal{T}$. Throughout the paper, we will omit the step numbers of partitions if the context is clear. Since a physical execution is a sequence of datasets to resolve, we can concatenate two physical executions $\mathcal{T}_1$ and $\mathcal{T}_2$ into one execution $\mathcal{T}_1+\mathcal{T}_2$. For example, concatenating two physical executions $(((R)),((S),(T)))$ and $(((U)))$ becomes $(((R)),((S),(T)),((U)))$. We can access each synchronous execution within a physical execution using a list index notation. For instance, the first synchronous execution of $\mathcal{T} = (((R)),((S),(T)))$ is $\mathcal{T}[1] = ((R))$.

Validity A correct physical execution should produce a joint ER result satisfying Definition 2. We capture this desirable property in the following definition.

Definition 3 A *valid* physical execution $\mathcal{T}$ of the initial partitions of $\mathcal{D}$ returns a set of partitions $\{P_R|R \in \mathcal{D}\}$ that is the same as the partitions produced by running the physical execution $\mathcal{T} + (((S)))$ for any $S \in \mathcal{D}$.

Intuitively, resolving datasets after running a valid physical execution plan does not change the final result. In our motivating example of Section 1, the physical execution $(((P)),((V)),((P)))$ is valid because running ER on the final partitions of P or V no longer generates new clusterings of records, so running the physical execution $(((P)),((V)),((P)))$ and then either $(((P)))$ or $(((V)))$ produces the same partitions as well. However, the physical execution $(((V)),((P)))$ is not valid because we fail to cluster the venues v_1 and v_2 together, and resolving V again after the physical execution results in v_1 and v_2 clustering (i.e., executing $(((V)),((P))) + (((V))) = (((V)),((P)),((V)))$ will produce a different result than executing $(((V)),((P))))$.

We now prove that running valid ER algorithms using a valid physical execution returns a correct joint ER result.

Proposition 1 *Running valid ER algorithms on a valid physical execution of $\mathcal{D}$ returns a joint ER result of $\mathcal{D}$ satisfying Definition 2.*

Proof We show that the result of running a valid physical execution $\mathcal{T}$ results in a set of partitions $\{P_R|R \in \mathcal{D}, P_R$ is a partition of $R, E_R(P_R, \mathcal{P}_{-R}) = P_R\}$. Since we run $\mathcal{T}$ on the initial partitions of the datasets in $\mathcal{D}$ using valid ER algorithms, we return a set of partitions of $\mathcal{D}$. We also know that running E_R on each final partition P_R of dataset R results in P_R because we know that running $\mathcal{T}$ produces the same result as running $\mathcal{T}$ + $(((R)))$ by the condition in Definition 3. □

Notice that while all valid physical executions return correct joint ER results, the converse is not true. That is, not all joint ER results that satisfy Definition 2 can be produced by valid physical executions. The reason is that Definition 2 does not require a step-wise execution of ER algorithms on the datasets for producing the joint ER result. For example, suppose that we only have one dataset $R = \{r, r'\}$ and an ER algorithm E_R where $E_R(\{\{r\},\{r'\}\}) = \{\{r\},\{r'\}\}$ and $E_R(\{\{r,r'\}\})$ $= \{\{r,r'\}\}$. Then a physical execution can only produce the joint ER result $\{\{\{r\},\{r'\}\}\}$ by running ER on the initial partition of R. However, there is another correct joint ER result $\{\{\{r,r'\}\}\}$, which cannot be produced by running E_R. Since our joint ER results are based on running ER algorithms on datasets, however, our desirable outcome is a valid physical execution that leads to a joint ER result satisfying Definition 2.

Feasibility In practice, there is a limit to the available resources for joint ER. For example, there may be a fixed amount of memory we can use at any point. Or there may be a fixed number of CPU cores we can use at the same time. In our paper, we only restrict the number of processors and define a physical plan to be

feasible if it can be executed using the given number of processors.

Definition 4 A *feasible* physical execution $\mathcal{T}$ satisfies the following condition.

- $\forall i \in \{1, \ldots, |\mathcal{T}|\}, |\mathcal{T}[i]| \leq$ number of processors

We denote the estimated running time of E_R on R as $t(E_R, R)$. For example, if E_R has a runtime quadratic to the size of its input and $|R| = 1000$, then we can estimate the runtime $t(E_R, R)$ as 10^6. The total runtime of a physical execution $\mathcal{T}$ is then $\Sigma_{i=1...|\mathcal{T}|} \max\{\Sigma_{R \in C} t(E_R, R) | C \in \mathcal{T}[i]\}$. For example, suppose we have the physical execution $\mathcal{T} = (((R, S), (T)), ((U)))$ and running E_R on R and E_S on S both take 3 hours, running E_T on T takes 5 hours, and running E_U on U takes 1 hour. Then the estimated total runtime is $\max\{3 + 3, 5\} + \max\{1\} = 7$ hours.

Our goal is to produce a valid and feasible physical execution that minimizes the total runtime. This problem can be proved to be NP-hard [15], so evaluating every possible physical execution may not be acceptable for resolving a large number of datasets. Hence in the following sections, we provide a step-wise approach where we first produce an "execution plan" that represents a class of valid physical executions (Section 3) and then produce feasible and efficient physical executions based on the execution plan (Section 4).

3 Scheduler

The scheduler receives an "influence graph" that captures the relationships among datasets and produces a logical execution plan using the influence graph. We identify which logical execution plans are correct in a sense that they can be used to produce valid physical executions. We propose an algorithm that generates efficient execution plans that satisfy desirable properties and are thus likely to result in fast physical executions. We also discuss how to construct influence graphs when blocking techniques are used.

3.1 Influence Graph

An influence graph G of the datasets $\mathcal{D}$ is generated by the application developer and captures the semantic relationships between the datasets. The vertices in G are exactly the datasets in $\mathcal{D}$, and there is an edge from dataset R to S if R "influences" S. We define the influence relationship between two datasets as follows.

Definition 5 A dataset R *influences* another dataset S (denoted as $R \rightarrow S$) if there exist partitions P_R of R and P_S of S such that the physical execution $(((R)), ((S)))$ applied to P_R and P_S may give a different result than when $(((S)))$ is applied.

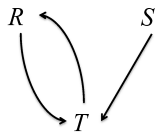

Fig. 3 An Influence Graph

In our motivating example of Section 1, the dataset P of papers influences the dataset V of venues because of the following observations. First, clustering the two papers $\{p_1\}$ and $\{p_3\}$ in P_P^1 resulted in the two venues $\{v_1\}$ and $\{v_2\}$ in P_V^2 clustering as well. Thus the entire physical execution $(((P)), ((V)))$ on the initial partitions P_P^0 and P_V^0 produces the partitions $P_P^2 = \{\{p_1, p_3\}, \{p_2\}\}$ and $P_V^2 = \{\{v_1, v_2\}\}$. On the other hand, if $\{p_1\}$ and $\{p_3\}$ had not been clustered, then $\{v_1\}$ and $\{v_2\}$ would not have clustered because the two venues have a low string similarity for names. So applying the physical execution $(((V)))$ on P_P^0 and P_V^0 produces the partitions $P_P^1 = \{\{p_1, p_3\}, \{p_2\}\}$ and $P_V^1 = \{\{v_1\}, \{v_2\}\}$. Hence by Definition 5, P influences V. An influence graph could possibly be generated automatically based on the ER algorithms. For example, the scheduler may consider R to influence S if the code in E_R for comparing two records r and r' in R also compares the S records that r and r' refer to.

The influence relationships among multiple datasets can be expressed as a graph. For example, suppose that there are three datasets R, S, and T where R influences T, T influences R, and S influences T. In this case, we can create an influence graph G (shown in Figure 3) that contains three vertices R, S, and T and three directed edges: $R \rightarrow T$, $T \rightarrow R$, and $S \rightarrow T$.

The influence graph provides guidance on the order for resolving datasets. According to the influence graph in Figure 3, it seems clear that we should resolve S before T. However, it is unclear how to order R and T. One possible physical execution we could use is $(((S)), ((R)), ((T)))$. However, since T influences R, we may want to resolve R one more time after resolving T just in case there are newly merging (splitting) clusters in the partition P_R of R. As a result, we might end up running the physical execution $(((S)), ((R)), ((T)), ((R)))$ instead. Furthermore, after resolving the last R, we may want to resolve T again just in case the partition P_T of T may change and so on. In general, we can only figure out the correct physical execution by actually running the ER algorithms until the ER results converge according to Definition 2. In the next section, we define the execution plan as a way to capture multiple pos-

sible physical executions into a more compact logical expression.

There are several ways to construct an influence graph. An automatic method is to view the ER algorithm where we draw an edge from R to S if the ER algorithm for S uses any information in R. Another method is to draw the edges based on the known semantics of the datasets. For example, while resolving papers may influence the resolution of venues, it is unlikely that the resolved papers would influence say phone numbers. One could also use probabilistic graphical models [24,27] to see if one dataset influences another dataset by checking if there are interrelationships across the matching records of the two datasets. For example, if a conditional random field (CRF) implies that two authors are the same if their papers are the same, then there is an influence from papers to authors. If the user initially does not know what datasets influence which, she can simply provide a fully connected graph. This worst-case graph causes the scheduling algorithm to be the most conservative. As the user discovers influences that do not exist (e.g., set R cannot influence set S), she can remove arcs from the graph and obtain better performance.

When constructing an influence graph, It is desirable to avoid unnecessary edges in the influence graph. Intuitively, the fewer the edges, the scheduler can exploit the graph to generate efficient plans for resolving the datasets as we shall see in Section 3.2. On the other hand, if most datasets influence each other, then there is not much optimization the scheduler can perform.

In the case where blocking techniques are used on a dataset R, we can construct an influence graph with fewer edges. Recall that blocking divides R into possibly overlapping smaller datasets $R_1, \ldots, R_k$ where each R_i fits in memory. In general, there can be edges between any R_i and R_j. If we assume that the resolutions of one block do not affect the resolutions in another block, however, we may remove the edges between the R_i's.

Furthermore, if blocking is used on multiple datasets, there are several possible ways for reducing the edges of the influence graph depending on the application. We illustrate the choices by considering a scenario of two datasets where R is a set of people and S is a set of organizations. Suppose that R influences S because some people may be involved in organizations. We assume that both R and S are too large to fit in memory and are thus blocked into smaller datasets. Hence, we would like to know which individual blocks of R influence which blocks in S. In the most general case, a person can be involved in any organization. Thus, each block in R influences all the blocks in S. However, suppose we know that the blocking for R and S is done on the country of residence and that the organizations in S are only domestic. By exploiting these application semantics, we can reduce the number of edges by only drawing an edge from each block of R to its corresponding block in S in the same country. In Section 3.3, we show how reducing the edges in the influence graph can improve the efficiency of joint ER.

3.2 Execution Plan

Directly generating a physical execution from an influence graph can be a complex process. We thus take a systematic approach by adding a level of abstraction and generating an intermediate execution plan first. The analogy is a DBMS producing a logical query plan for a given query before generating the actual physical query plan. As a result, the joint ER processor can be flexible in generating the actual physical execution based on the high-level execution plan. In this section, we show how the scheduler generates an execution plan based on an influence graph.

3.2.1 Syntax and Properties

An execution plan L is a sequence of concurrent and fixed-point sets (see Table 2). A concurrent set contains one or more datasets that are resolved concurrently once. A fixed-point set contains one or more datasets to resolve together until "convergence." That is, the datasets in a fixed-point set F are resolved possibly more than once in a sequence of concurrent sets to return the set of partitions defined below. (Note that $\mathcal{P}_{-R}$ below is still defined as $\{P_X | X \in \mathcal{D} - \{R\}\}$ and not as $\{P_X | X \in F - \{R\}\}$. That is, when resolving $R \in F$, the ER algorithm E_R may still use the information in a dataset outside F.)

Definition 6 A *converged result* of a fixed-point set F is the set of partitions $\{P_R | R \in F, P_R$ is a partition of R, and $E_R(P_R, \mathcal{P}_{-R}) = P_R\}$.

For example, datasets with the influence graph of Figure 3 can be resolved by the execution plan $L = (\{S\}, \{R, T\}+)$ (see Section 3.2.2 for details on the execution plan generation) where S is resolved once, and R and T are repeatedly resolved until convergence. Hence, one possible physical execution for L is $(((S), ()), ((R), (T)), ((R), (T)), \ldots)$ where the number of $((R), (T))$'s depends on the contents of the records. (In Section 4.2, we show other techniques for generating physical executions of fixed-point sets that can improve the physical execution runtime.)

We can access a concurrent or fixed-point set of an execution plan using a list index notation. For instance,

Name	Syntax	Description
Concurrent Set	$\{\dots\}$	Resolve datasets concurrently once
Fixed-point Set	$\{\dots\}+$	Resolve datasets until convergence

Table 2 Execution Plan Syntax

for the execution plan L above, $L[1] = ((S), ())$ and $L[2] = ((R), (T))$. Since an execution plan is a sequence of fixed-point and concurrent sets, one can concatenate two execution plans into a longer sequence.

We would like to define "good" execution plans that can lead to valid physical executions satisfying Definition 3. For example, suppose that the influence graph G contains two datasets R and S, and that R influences S. Then the execution plans $(\{R, S\}+)$ and $(\{R\}, \{S\})$ seem to be good because for each R resolved, either S is resolved with R in the same fixed-point set (in the first plan) or after R (in the second plan). However, the execution plan $(\{S\},\{R\})$ may not lead to a correct joint ER result because resolving S again after the resolution of R may generate a different joint ER result.

We capture the desirable notion above into the property of conformance below.

Definition 7 An execution plan L *conforms to* an influence graph G of $\mathcal{D}$ if

1. $\forall R \in \mathcal{D}$, $\exists i$ s.t. $R \in L[i]$ and
2. $\forall R, S \in \mathcal{D}$ s.t. R has an edge to S in G, for each $L[i]$ that contains R, either
   - $L[i]$ is a fixed-point set and $S \in L[i]$ or
   - $\exists j$ where $j > i$ and $S \in L[j]$.

For example, suppose there are two datasets R and S where R influence S according to the influence graph G. Then the execution plans $(\{R, S\}+)$ and $(\{R\}, \{S\})$ conform to G because in both cases, R and S are resolved at least once and S is either resolved with R in the same fixed-point set or resolved after R. However, the execution plan $(\{S\})$ does not conform to G because R is not resolved at least once (violating the first condition of Definition 7). Also, the execution plan $(\{R, S\})$ does not conform to G because S is neither resolved with R in the same fixed-point set nor resolved after R (violating the second condition). In Proposition 6 (see Section 4), we show that an execution plan that conforms to an influence graph G results in a valid physical execution given that the physical execution terminates.

3.2.2 Construction

A naïve execution plan that conforms to an influence graph G of the datasets $\mathcal{D}$ is $(\{\mathcal{D}\}+)$, which contains a single fixed-point set containing all the datasets in $\mathcal{D}$. The following Lemma shows that $(\{\mathcal{D}\}+)$ conforms to G.

Lemma 1 *Given an influence graph G of the datasets $\mathcal{D}$, the execution plan $(\{\mathcal{D}\}+)$ conforms to G.*

Proof The first condition of Definition 7 is satisfied because all the datasets in $\mathcal{D}$ are contained in the one fixed-point set of $(\{\mathcal{D}\}+)$. The second condition holds as well because for any datasets R and S where R influences S, S is in the same fixed-point set as R. □

However, $(\{\mathcal{D}\}+)$ is not an efficient execution plan in terms of the runtime of the physical execution since we would need to repeatedly resolve all the datasets until convergence. We thus explore various optimizations for improving an execution plan in general. We assume the scheduler only has access to the influence graph and not the runtime and memory physical statistics of the datasets. Hence, we use heuristics that are likely to improve the execution plan. The analogy is logical query optimization in database systems where pushing down selects in a query plan most likely (but not necessarily) improves the query execution time. An important requirement for the optimization techniques is that the final joint ER result must still satisfy Definition 2 (although the result does not have to be unique). We present three optimization techniques for execution plans and then present an algorithm that can produce efficient execution plans according to the optimization criteria.

Remove Redundant Datasets We can improve the efficiency of an execution plan by removing datasets that are redundant. For example, in the execution plan $(\{R, S\}+, \{R, S\})$, running ER on the concurrent set $\{R, S\}$ is unnecessary because the fixed-point set $\{R, S\}+$ already returns a converged result of R and S. We say that an execution plan L is *minimal* if for each dataset $X \in \mathcal{D}$, X is inside at most one concurrent or fixed-point set in L. For example, $(\{R, S\}+, \{R, S\})$ is not a minimal execution plan because R and S occur in two sets while $(\{R, S\}+)$ is a minimal plan. As another example, while $(\{R, S\}, \{R, S\})$ is not minimal, $(\{R, S\})$ is minimal. Notice that in the second example, the two plans are not necessarily equivalent.

Avoid Large Fixed-Point Sets A minimal execution plan is not necessarily the most efficient one. Consider the naïve execution plan $(\{\mathcal{D}\}+)$, which contains a single fixed-point set containing all the datasets. While $(\{\mathcal{D}\}+)$ is minimal, it is not necessarily an efficient plan as we discussed above and could possibility split into smaller concurrent and fixed-point sets. For example, suppose that $\mathcal{D} = \{R, S, T\}$, and the execution plan $L = (\{S\}, \{R, T\}+)$, which conforms to the influence graph of Figure 3. When resolving the datasets in the

order of L, we only need to resolve S once and do not have to worry about the ER result of S after resolving R and T together. However, if we were to run the execution plan $(\{\mathcal{D}\}+)$, we might have to resolve S multiples times because we do not have any information of which datasets can safely be resolved exactly once.

Hence, our second optimization for execution plans is to avoid large fixed-point sets. As a result, we can save the time needed to check for convergence as we illustrated above. Given an execution plan L that conforms to the influence graph G, we say that L is *compact* if no fixed-point set in L can be divided into smaller concurrent or fixed-point sets while still guaranteeing that L conforms to G. For example, suppose we have the two datasets R and S where R influences S. Then the execution plan $L = (\{R\}, \{S\})$ is compact because L conforms to G and does not contain any fixed-point sets. However, the plan $(\{R, S\})$ is not compact because it does not conform to G. Also, the plan $(\{R, S\}+)$ is not compact as well because the fixed-point set $\{R, S\}+$ can be divided into the sequence of two concurrent sets $\{R\}$ and $\{S\}$.

Notice that a minimal execution plan is not necessarily compact and vice versa. For instance, if there are two datasets R and S that do not influence each other, the plan $L_1 = (\{R, S\}+)$ is minimal while $L_2 = (\{R\}, \{S\}, \{R\}, \{S\})$ is not minimal. However, L_2 is compact while L_1 is not.

Maximize Parallelism Assuming we have enough cores and memory, we can improve an execution plan by placing more datasets in the same concurrent set to resolve concurrently. For example, if R and S do not influence each other and the execution plan is $(\{R\}, \{S\}, \{T, U\}+)$, then given that we have two cores and enough memory, we could use the better plan $(\{R, S\}, \{T, U\}+)$ where we resolve R and S concurrently. In general, we say an execution plan is *maximally parallel* if for each concurrent or fixed-point set S resolved, we resolve as many datasets that are not influenced by other unresolved datasets as possible. For example, suppose we have the influence graph $R \to S$, $T \to U$. Then an execution plan $(\{R\}, \{S, T\}, \{U\})$ is not maximally parallel because T could have been resolved with R during the first concurrent set. On the other hand, the execution plan $(\{R, T\}, \{S, U\})$ is maximally parallel because while the first concurrent set $\{R, T\}$ is being resolved, neither S nor U can be resolved as well because R influences S and T influences U.

Construction Algorithm Algorithm 1 uses the three heuristics above to produce execution plans that conform to the given influence graph G. To illustrate Algorithm 1,

Algorithm 1: Constructing an execution plan

```
   input : An influence graph G
   output: An execution plan L
 1 Group each strongly connected component of G
   into a node in the graph G';
 2 For each pair of nodes n, n' ∈ G', draw an edge
   from n to n' if ∃R ∈ n and ∃S ∈ n' where R has an
   edge to S in G;
 3 repeat
 4     Z ← nodes in G' that are not pointed from
       other nodes;
 5     if a node n ∈ Z has a size |n| > 1 then
 6         L ← L + (n+);
 7         remove n from G';
 8     else
 9         L ← L + (⋃_{n∈Z} n);
10         remove nodes in Z from G';
11 until G' is empty;
12 return L;
```

suppose that the influence graph G contains four datasets R, S, T, and U and has three edges $R \to T$, $T \to R$, and $S \to T$. In Step 1, we identify the strongly connected components of G, which are $\{R, T\}$, $\{S\}$, and $\{U\}$. In Step 2, we create the graph G' where each strongly connected component in G forms a single node in G'. A node in G' is thus a set of datasets that are strongly connected in G. A node n in G' points to another node n' if a dataset in n points to any dataset in n' according to G. Hence, G' in our example has three nodes $n_1 = \{R, T\}$, $n_2 = \{S\}$, and $n_3 = \{U\}$ where n_2 points to n_1 (since S influences T). This construction guarantees that G' is always a directed acyclic graph. Starting from Step 3, we identify the nodes in G' that are not influenced by any other node. In our example, we identify the set $Z = \{\{S\}, \{U\}\}$. Since both nodes in Z have a size of 1, we add to L the combined set $\{S, U\}$ (Step 9). Next, we update Z to $\{\{R, T\}\}$. Since the node $\{R, T\}$ has two datasets, it is added to L as a fixed-point set $\{R, T\}+$ (Step 6). As a result, our final execution plan is $L = (\{S, U\}, \{R, T\}+)$. Finally we return L as the output in Step 12.

The following proposition shows the correctness of Algorithm 1.

Proposition 2 *Given an influence graph G of the datasets $\mathcal{D}$, Algorithm 1 returns an execution plan that conforms to G.*

Proof The first condition of Definition 7 is satisfied because every dataset in $\mathcal{D}$ is allocated to exactly one fixed-point or concurrent set in L by construction. We now prove the second condition of Definition 7. Suppose that R influences S in G. First, suppose there is a directed path from S to R in G. Then R and S form

a strongly connected component and are thus grouped into the same node n in Step 1. In Step 6, n is added to L as a fixed-point set because n contains at least two datasets. Thus the first OR-condition of the second condition is satisfied. Second, if there is no directed path from S to R in G, then by Steps 4 and 8 of the algorithm, the node n containing S is added to L after the node n' containing R, satisfying the second OR-condition of the second condition. As a result, the second condition of Definition 7 is always satisfied. □

The following proposition shows the efficiency of the execution plans produced by Algorithm 1.

Proposition 3 *An execution plan produced by Algorithm 1 is minimal, compact, and maximally parallel.*

Proof An execution plan L generated by Algorithm 1 is minimal because any dataset $X \in \mathcal{D}$ is not resolved in more than one concurrent or fixed-point set.

Next, we prove that L is also compact where none of the fixed-point sets in L can be split into smaller fixed-point sets while still guaranteeing that L conforms to G. Suppose that Algorithm 1 returns an execution plan L, and we split a fixed-point set $\mathcal{S}$ in L into two sets A and B. Since the datasets in $\mathcal{S}$ form a strongly connected component C in G, the sets A and B form two subgraphs C_1 and C_2 of C that are mutually connected in G. As a result, there exists a pair of datasets $R \in C_1$ and $S \in C_2$ where R influences S and a pair of datasets $R' \in C_1$ and $S' \in C_2$ where S' influences R'. Since L conforms to G and R influences S, the fixed-point set of C_2 must follow the fixed-point set of C_1. However, since S' influences R', the fixed-point set of C_1 must follow the fixed-point set of C_2, which is a contradiction. Thus, C_1 and C_2 must be combined into a single fixed-point set for L to conform to G.

Finally, the execution plan L is also maximally parallel because we add as many datasets that are not influenced by other datasets as possible in Step 8. □

We now show that Algorithm 1 runs in linear time.

Proposition 4 *Given an influence graph G with V vertices and E edges, Algorithm 1 runs in $O(|V| + |E|)$ time.*

Proof Identifying the strongly connected components in G (Step 1) can be done in $O(|V| + |E|)$ time using existing method such as Tarjan's algorithm [29]. Next, generating the execution plan in Steps 3–9 can be done in $O(|V| + |E|)$ time by keeping track of nodes without incoming edges and removing edges from the nodes added to L. Hence, the total complexity of Algorithm 1 is $O(|V| + |E|)$. □

Again, the optimizations used in the scheduler are heuristics that are likely (but not guaranteed) to improve the actual runtime of the physical execution. For example, maximizing parallelism is only effective when we have enough cores and memory to resolve datasets concurrently. For now, we consider the execution plans produced by Algorithm 1 to be reasonably efficient for our experiments. In Section 4, we discuss how to produce efficient physical executions based on the optimized execution plans generated from Algorithm 1.

3.3 Exploiting the Influence Graph

The more we know about what data sets do *not* influence others, the better the scheduler can exploit the given influence graph and improve the efficiency of the execution plan. We illustrate this point by considering a blocking scenario where two datasets R and S are blocked into R_1, R_2 and S_1, S_2, respectively. We also assume that the blocks of R may influence the blocks of S and vice versa, but the blocks within the same dataset do not influence each other.

In the case where each block in R influences all the blocks in S and vice versa, we need to draw edges from R_i to S_j and S_j to R_i for $i, j \in \{1, 2\}$, resulting in a total of 8 edges. However, the more strongly connected the influence graph, the more difficult it becomes for the scheduler to generate an efficient execution plan. In our example, the scheduler generates the execution plan $L_1 = (\{R_1, R_2, S_1, S_2\}+)$ using Algorithm 1.

Now suppose we know through application semantics that each block in R only influences its corresponding block in S with the same index and vice versa. As a result, we only need to draw four influence edges: R_1 to S_1, S_1 to R_1, R_2 to S_2, and S_2 to R_2. Hence, the scheduler generates the execution plan $L_2 = (\{R_1, S_1\}+, \{R_2, S_2\}+)$ using Algorithm 1. The plan L_2 is more efficient than L_1 in a sense that we can exploit the information that R_1 and S_1 can be resolved separately from R_2 and S_2.

In an extreme case, we might know that R_2 does not influence any block in S. That is, only R_1 and S_1 influence each other. As a result, the execution plan is now $L_3 = (\{R_1, S_1\}+, \{R_2, S_2\})$ where we exploit that fact that R_2 and S_2 do not influence each other. Again, the plan L_3 is more efficient than L_2 where we know R_2 and S_2 only need to be resolved once.

4 Joint ER Processor

In this section, we discuss how the joint ER processor uses an execution plan to generate a valid phys-

ical execution. We first discuss how a concurrent set can be resolved within a constant factor of the optimal schedule. Next, we discuss how to resolve fixed-point sets. We then prove that sequentially resolving the concurrent and fixed-point sets according to the execution plan produces a valid physical execution as long as the execution terminates. Finally, we introduce expander functions, which can be used to significantly enhance the efficiency of joint ER.

4.1 Concurrent Set

When resolving a concurrent set, we are given a set of datasets to resolve once and a number of processors that can run in parallel. The overall runtime is thus the maximum runtime among all processors. We would like to assign the datasets to the processors such that the overall runtime is minimized.

We use the List scheduling algorithm [16] for scheduling datasets to processors. Whenever there is an available processor, we assign a dataset to that processor that start running ER. This algorithm is guaranteed to have a runtime within $(2-\frac{1}{p})$ times the optimal schedule time where p is the number of processors. A key advantage of the List scheduling algorithm is that it is an "on-line" algorithm, i.e., executed while ER is running. Predicting the runtime of ER is challenging because it may depend on how other datasets have been resolved. While there are other scheduling works that improve on the constant bound, they do not make significant improvements and are off-line, i.e., executed in advanced of any actual invocations of ER algorithms.

4.2 Fixed-Point Set

We now resolve a fixed-point set, where we are given a number of datasets that must be resolved at least once until convergence on parallel processors. The goals is to again minimize the overall runtime of the processors by scheduling the resolutions of datasets to the processors.

We propose an on-line iterative algorithm for resolving a fixed-point set. Again, the advantage of an on-line algorithm is that we do not have to worry about estimating the runtime of ER. The idea is to repeatedly resolve the fixed-point set, but only the datasets that need to be resolved according to the influence graph. Initially, all datasets need to be resolved. However, after the first step, we only resolve the datasets that are influenced by datasets that have changed in the previous step. For example, suppose we are resolving the fixed-point set $\{R, S, T\}+$ where R and T influence each other while S and T influence each other. We first resolve all three datasets using the greedy algorithm in Section 4.1. After the first step, say that only R and S had new partitions. Then in the next step we only need to resolve T. If T no longer has new partitions after its resolution, then we terminate.

Finding the optimal schedule for fixed-point sets is even more challenging than that of resolving a concurrent set. In our example above, since T is resolved twice, we might not have needed to resolve T during the first step. However, performing this optimization requires a deterministic way of figuring out if T will be resolved in future steps and whether it is indeed sufficient to resolve T just once, which are both hard to predict. We plan to study such optimizations in future work.

In general, resolving a fixed-point set is not guaranteed to converge. In order to see when we are guaranteed termination, we first categorize influence as either positive or negative. A positive influence from R to S occurs when for some P_R and P_S there exists two records $s, s' \in S$ that are not clustered when we run the physical execution $(\{S\})$, but are clustered when we run the physical execution $(\{R\}, \{S\})$. Conversely, a negative influence occurs when s and s' are clustered when we run $(\{S\})$, but not clustered when we run $(\{R\}, \{S\})$. Note that an influence can be both positive and negative.

We now show that, if all influences are positive, then the iterative algorithm for fixed-point sets always returns a converged set of partitions for a fixed-point set.

Proposition 5 *If all the influences among the datasets in a fixed-point set F are not negative, then the iterative algorithm for fixed-point sets terminates and produces a set of partitions satisfying Definition 6.*

Proof Suppose we have the fixed-point set F and repeatedly resolve the datasets that can potentially change. If all the influences are non-negative, clusters can only merge and never split. Since there is only a finite number of possible merges, the partitions of the datasets eventually converge into a set of partitions $\{P_R | R \in F\}$. In addition, since no clusters merge anymore, $E_R(P_R, \mathcal{P}_{-R}) = P_R$ for each dataset $R \in F$. Hence, there exists a converged result satisfying Definition 6 for the datasets in F. □

In the case where both negative and positive influences occur, the fixed-point set may not have a joint ER result. For example, say that two authors a_1 and a_2 merging influences a cluster containing two papers p_1 and p_2 to split, but then if p_1 and p_2 split, a_1 and a_2 split as well. Also, if a_1 and a_2 split, say that p_1 and p_2 merge, and p_1 and p_2 merging causes a_1 and a_2 to

merge as well. As a result, the splits and merges continue indefinitely, and there is no fixed-point result that satisfies Definition 2.

If negative influences prevent joint ER from terminating, we can restrict the number of ER invocations in an execution plan L. Even if we do not have a fixed-point result that satisfies Definition 2, we may still improve accuracy compared to the simple case where the ER algorithms are run in isolation. For instance, if we are resolving the fixed-point set $\{R, S\}+$ where R and S influence each other, we can limit the repetition to, say, 2. Although the physical execution $(\{R, S\}, \{R, S\})$ is not guaranteed to return a joint ER result satisfying Definition 2, it may be close enough with a reasonable amount of runtime spent. In Section 6.3, we show that in practice, the ER results converge quickly (within 3 resolutions per dataset) while producing the correct joint ER result.

4.3 Joint ER Algorithm

We discuss how the joint ER processor uses an execution plan to generate a physical execution. Our joint ER algorithm first initializes each dataset by creating a set of singleton clusters of the records. The algorithm then sequentially resolves each concurrent or fixed-point set using the algorithms in Sections 4.1 and 4.2, respectively.

We show that the resulting physical execution is valid as long as it terminates.

Proposition 6 *Given an execution plan L that conforms to an influence graph G of the datasets $\mathcal{D}$, if the joint ER algorithm using L terminates, then the resulting physical execution is valid.*

Proof Suppose that we have the datasets $\mathcal{D}$ and an influence graph G. Given an execution plan L that conforms to G, suppose that the joint ER algorithm terminates, producing a physical execution $\mathcal{T}$. First, the result of the basic algorithm is a set of partitions for all the datasets in $\mathcal{D}$ because we start from the initial partitions of $\mathcal{D}$ and only use valid ER algorithms on the partitions. Second, we show that running $\mathcal{T}$ produces the same partitions as running $\mathcal{T}+ [\{R\}]$ for any $R \in \mathcal{D}$. For each dataset R, we show that, after running $\mathcal{T}$, its partition P_R cannot change further by running $[\{R\}]$. Suppose P_R has changed while running $[\{R\}]$. Then there must be some dataset X that was resolved after the last resolution of R in $\mathcal{T}$ and influenced the resolution of R during the execution of $[\{R\}]$. If X was in the same concurrent set as R, we have a contradiction because X should have been in the same fixed-point set as R by Definition 7. If X was in the same fixed-point set as R, then since the joint ER algorithm terminates, we know that the resolved fixed-point set satisfies Definition 6. Hence, resolving R again should not change P_R further, contradicting our assumption. Finally, X could be in a concurrent or fixed-point set other than R's set. Since L conforms to G, however, there must have been another resolution of R after the resolution of X in $\mathcal{T}$, which is a contradiction. Hence, $\mathcal{T}$ is a valid physical execution. □

4.4 Expander Function

We optimize a physical execution where we focus on minimizing redundant computation as much as possible when a dataset is resolved multiple times. A key property we satisfy is that the resulting joint ER result is the same regardless of the optimization.

A record $r \in R$ *refers to* a record $s \in S$ if r contains a pointer to s. For example, if the paper record r contains an attribute with a label "venue" and value "ACM TODS," and s represents the venue ACM TODS, then r refers to s. Or the venue attribute could simply contain s's ID. In general, r could refer to more than one record in S. Hence, we denote the set of S records r refers to as $\mathcal{R}_S(r)$ where $\mathcal{R}_S(r) \subseteq S$. In our motivating example of Section 1, $\mathcal{R}_P(v_1) = \{p_1, p_2\}$ while $\mathcal{R}_V(p_1) = \{v_1\}$. Conversely, the set of records in S that refer to r is denoted as $\mathcal{R}_S^{-1}(r)$. In our motivating example, $\mathcal{R}_P^{-1}(v_1)$ is again $\{p_1, p_2\}$. However, if p_1 did not refer to v_1, then $\mathcal{R}_P^{-1}(v_1)$ would be $\{p_2\}$ while $\mathcal{R}_P(v_1)$ is still $\{p_1, p_2\}$.

Compared to the case where all datasets are resolved in isolation, joint ER has the overhead of possibly resolving a dataset multiple times. To reduce this overhead, we would like to narrow down the set of records influenced by a previous resolution and only run ER on those records. For example, suppose that we have the execution plan $(\{R\}, \{S\}, \{R\})$ and have already resolved the first R and S. When resolving the second R, we would like to avoid running ER on the entire P_R. Instead, the idea is to run ER on only a small subset of clusters $O \subseteq P_R$ and produce the same result as resolving the entire P_R again. Determining O can be done by exploiting the given ER algorithm as we describe below. We note that the idea of avoiding resolution from scratch does not always work for all ER algorithms. That is, certain ER algorithms may perform global operations and thus require that all the records are resolved from the beginning. An in-depth study of the properties that enable incremental ER can be found in reference [31].

To construct the candidate records to resolve, we first construct the set of records M that contains R

records that are referenced by records in other datasets that have changed since the last time R was resolved. That is, if the records in $c \subseteq S$ are newly clustered, and the records in c refer to the records in $c' \subseteq R$, then we add the records in c' to M. We repeat this operation for all datasets that influence R. However, resolving just R' may not produce a correct result. For example, suppose we want to compare the records r and r' when resolving R the second time. However, in order to see if r and r' are indeed the same entity, we might have to resolve the two records along with another record r'' because r can only match with r' if r' matches and clusters with r''. Hence, we must run ER on a sufficiently large superset of R' that would guarantee a correct ER result.

An *expander function* X_R for dataset R produces this superset by receiving M and returning a set of clusters $O \subseteq P_R$ such that running ER on O produces a result just as if we have run ER on the entire partition P_R. More formally, we define a valid expander function as follows.

Definition 8 Given an input partition P_R of R for ER and a set of records M to resolve, a *valid* expander function X_R for the ER algorithm E_R satisfies the following condition:

- $E_R(P_R) = E_R(X_R(M)) \cup (P_R - X_R(M))$

If $|X_R(M)|$ is much smaller than $|P_R|$, then running ER on $X_R(M)$ can be much faster than running ER on the entire P_R. We note that not all ER algorithms have a valid expander function that returns a set $X_R(M)$ smaller than P_R.

To illustrate an expander function, we use the sorted neighborhood technique [17] (SN) as our ER algorithm. The SN algorithm first sorts the records in P_R (i.e., we extract all the records from the clusters in P_R) using a certain key assuming that closer records in the sorted list are more likely to match. For example, suppose that we have the input partition $P_R = \{\{r_1\}, \{r_2\}, \{r_3\}\}$ and sort the clusters by their names (which are not visible in this example) in alphabetical order to obtain the list $Z = (r_1, r_2, r_3)$. The SN algorithm then slides a fixed-sized window on the sorted list of records and compares all the pairs of records that are inside the same window at any point. More formally, we only compare the records r and r' where $|Rank(r, Z) - Rank(r', Z)| < W$ where $Rank(r)$ denotes the index of r within Z while W denotes the window size. If the window size is 2 in our example, then we compare r_1 with r_2 and then r_2 with r_3, but not r_1 with r_3 because they are never in the same window. We thus produce pairs of records that match with each other. After collecting all the pairs of records that match, we perform a transitive closure on all the matching pairs of records to produce a partition P'_R of records. For example, if r_1 matches with r_2 and r_2 matches with r_3, then we merge r_1, r_2, r_3 together into the output $P'_R = \{\{r_1, r_2, r_3\}\}$.

Given a previous SN result P_R and a set of records M to re-resolve, we define the expander function X_R to return the set of records $O = \bigcup_{c \in Y} c$ where $Y = \{c \mid c \in P_R \wedge \exists r \in c, r' \in M \text{ s.t. } |Rank(r, Z) - Rank(r', Z)| < M\}$. (We later prove that O correctly contains all the records that may potentially match with records in M.) For example, if we have the sorted list $Z = (r_1, r_2, r_3, r_4)$ and $M = \{r_1, r_2\}$, then given the partition $P_R = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, we need to resolve M's records with r_3 because the cluster $\{r_3\}$ overlaps with one of the windows (r_2, r_3) of r_2. However, we do not have to resolve the records of M with r_4 because no window of r_1 or r_2 overlaps with the cluster $\{r_4\}$.

Proposition 7 *The function X_R is a valid expander function for the SN algorithm satisfying Definition 8.*

Proof Suppose that we are given a previous SN result P_R and a set of records M to re-resolve. Then for each $r \in M$, the SN algorithm must resolve r with any record r' that is within a cluster in P_R that overlaps with any window of any record in M. We revisit our example above where we have the sorted list $Z = (r_1, r_2, r_3, r_4)$ and $M = \{r_1, r_2\}$. Then given the partition $P_R = \{\{r_1, r_2\}, \{r_3\}, \{r_4\}\}$, we need to resolve the records of M with r_3 because r_2 and r_3 are in the same window and might match because $r_2 \in M$. In general, any record satisfying the condition above must be in the output of X_R because there is a chance of a transitive match of records $r, s_1, \ldots, s_k, r'$ where r matches with s_1, s_2 matches with s_3, ..., and s_k matches with r'.

We now show that if r' does not satisfy the above condition, then r and r' are guaranteed not to match when SN is run on the entire P_R. Since r and r' are not within the same window, then r and r' are never directly compared. Also, since the cluster of r' does not overlap with any window of any record in M, there is no chance of a transitive match of records $r, s_1, s_2, \ldots, s_k, r'$ either. In our example above, we do not have to match the records of M with r_4 because we know that r_3 and r_4 did not previously match with each other according to P_R and will not match now either because both r_3 and r_4 are not in M.

Our definition $X_R(M) = \bigcup_{c \in Y} c$ (where $Y = \{c \mid c \in P_R \wedge \exists r \in c, r' \in M \text{ s.t. } |Rank(r, Z) - Rank(r', Z)| < M\}$) exactly captures the condition above. Hence, the result of running E_R only on $X_R(M)$, i.e., $E_R(X_R(M)) \cup (P_R - X_R(M))$, is equivalent to the result of running E_R on P_R. □

We can now improve the resolution of a fixed-point set by using expander functions. Instead of running $E_R(P_R, \mathcal{P}_{-R})$ for each dataset R, we run $E_R(X_R(Cand(P_R)), \mathcal{P}_{-R})$ where $Cand(P_R)$ returns the records in R referenced by newly-clustered records in other datasets.

5 ER Algorithm Training

In this section, we discuss how to train joint ER algorithms. (While we present the ER training in a high-level fashion, a detailed training example is presented in Section 6.3.) We can train on small samples of the datasets in $\mathcal{D}$ where we know the correct joint ER results. The samples (collectively called $\mathcal{D}'$) "represent" $\mathcal{D}$ in a sense that the parameters trained for the samples will also work well for resolving the entire datasets in $\mathcal{D}$.

If all the datasets in $\mathcal{D}$ are resolved in isolation, we would only need to train the ER algorithm for each dataset in $\mathcal{D}'$ once. Since we are in a joint ER environment, however, training based on which other datasets have been resolved can be useful. For example, suppose that we are resolving papers mostly based on their title comparison results. However, if an author dataset that influences the paper dataset is resolved, then when resolving papers it might help to put more emphasis on the author comparison result. Moreover, if the papers have previously been resolved once already, then we would actually like to give less emphasis to the title comparison results. Hence, we propose a *state-based training* technique that can train an ER algorithm based on the state of other datasets being resolved.

Suppose that we are training an ER algorithm E_X that resolves the dataset $X \in \mathcal{D}'$. Given P_X, the gold standard G_X of X, and the other datasets in $\mathcal{D}'$, the supervised training algorithm returns the parameter values (e.g., comparison thresholds) for E_X. For example, say that we are using the execution plan $L = (\{R\}, \{S, T\}+, \{U\})$ where R, S, T, and U are the datasets in $\mathcal{D}'$. In order to resolve R, we train the parameters of E_R using the gold standard G_R and initial partitions P_R^0, P_S^0, P_T^0, and P_U^0.

When we resolve S in the above example, however, which partitions should be used to train E_S? Using P_R^0, P_S^0, P_T^0, and P_U^0 does not seem right because S is resolved after R is resolved. Instead, we would like to train E_S using $P_R^1 (= E_R(P_R^0, \mathcal{P}_{-R}^0))$, $P_S^1 (= P_S^0)$, $P_T^1 (= P_T^0)$, and $P_U^1 (= P_U^0)$, which are the resulting partitions after resolving R. To capture this notion, we define the *state* of the current resolution of a dataset X to be the physical execution until X's iteration starts. We would like to train the parameters of E_X using the partitions generated by each possible state. For instance, if the entire physical execution of L is $(((R)), ((S), (T)), ((S), (T)), ((U)))$, then we can train S twice, once on state $(((R)))$ and once on state $(((R)), ((S), (T)))$.

Once we have trained the ER algorithms, we can use them to resolve the entire datasets $\mathcal{D}$. We make the assumption that the resolution of $\mathcal{D}'$ (using the same scheduling algorithm as $\mathcal{D}$) generates the same physical execution as that of resolving $\mathcal{D}$. The intuition is that even if we take a sample of $\mathcal{D}$, the same influences will hold among the datasets in $\mathcal{D}'$. As a result, the same influence graph will be generated, leading to similar physical executions. (Formally verifying that our assumption holds is beyond the scope of this paper, and we leave it as a future work.) Hence, we can directly used the trained ER algorithms when resolving $\mathcal{D}$. In Section 6.3, we demonstrate that state-based training can significantly improve the accuracy of training without states.

6 Experimental Results

We evaluate our joint ER techniques on synthetic datasets and show the runtime behavior of our techniques. We then evaluate the scalability and behavior of joint ER on a large real dataset (called the Spock dataset [28]). We then evaluate our training techniques using another real dataset (called the Cora dataset) to demonstrate state-based training. Our algorithms were implemented in Java, and our experiments were run on a 2.4GHz Intel(R) Core 2 processor with 4 GB of RAM.

6.1 Synthetic Data Experiments

We evaluate the runtime behavior of joint ER using synthetic data. The main advantage of synthetic data is that they are much easier to generate for different scenarios and provide more insights into the operation of our joint ER algorithms.

Table 3 shows the parameters used for generating the synthetic data and the default values for the parameters. We first create d datasets where each dataset contains records that represent a total of s entities. For each entity, there are u records that represent that entity. As a result, each dataset contains $s \times u$ records. While each dataset thus has $200 \times 5 = 1,000$ records as a default, one could easily scale this data to much larger sizes. Each record r in a dataset contains one integer value $r.v$. While a record may contain many attributes in practice, we simplify our model and assume that $r.v$ represents all the attributes of r that are not references to records in other datasets. In addition, we later use the values to "emulate" the comparison rule, i.e., if two

Param.	Description	Val.
Data Generation		
d	Number of datasets	15
s	Number of entities per dataset	200
u	Number of duplicate records per entity	5
i	Value difference between consecutive entities	10
v	Maximum deviation of value per entity	5
Comparison Rule		
a	Value similarity weight	0.5
t	Record comparison threshold	0.5
Resource		
p	Number of processors	2

Table 3 Parameters for generating synthetic data

records have values that are close, then they will more likely be considered the same entity by the comparison rule. We assume that the entities of a dataset have the values 0, i, $2 \times i$, ..., $(s-1) \times i$. A record that represents an entity with a value of e contains a value randomly selected from $[e, e+v]$. If a dataset R influences S, we create an attribute in each R record that refers to an existing record in S. When assigning references, we require that for any two records of the same entity, they can only refer to two records of the same entity of another dataset. The set of datasets influencing the dataset S is denoted as $\mathcal{I}(S)$. As defined in Section 4.4, the set of records in S referring to the record r is denoted as $\mathcal{R}_S^{-1}(r)$. When running joint ER, the parameter p indicates the number of CPU processors that can resolve datasets concurrently.

The comparison rule B compares two records and returns true if they are similar and false otherwise. When comparing the records r and r', B considers two similarities: the value similarity, which compares the values $r.v$ and $r'.v$, and the reference similarity, which compares the references of r and r' to records in other datasets. We use the parameter a to balance the value and reference similarities as follows:

$$B(r,r') = a \times \frac{1}{|r.v - r'.v| + 1} + (1-a) \times \frac{\sum_{X \in \mathcal{I}(R)} I(r,r',X)}{\sum_{X \in \mathcal{I}(R)} I(r,r',X) + N(r,r')} \geq t$$

where $I(r,r',X) = |\mathcal{R}_X^{-1}(r) \cap \mathcal{R}_X^{-1}(r')|$ and $N(r,r') = |\{Y | Y \in \mathcal{I}(R) \wedge I(r,r',Y) = 0\}|$. That is, $I(r,r',X)$ is the number of records in X that refer to both r and r', and $N(r,r')$ is the number of datasets that influence R and that do not have any records that refer to both r and r'. The first term weighted by a is the normalized value similarity, which ranges from $\frac{1}{d+1}$ to 1, and increases as the values of r and r' are more similar. The second term weighted by $(1-a)$ is the normalized reference similarity, which ranges from 0 to 1 and increases as more references in r and r' overlap. B returns true if the sum of these normalized values is larger or equal to the comparison threshold t.

For our ER algorithm we use the R-Swoosh algorithm [4], which uses a Boolean pairwise comparison rule to compare records and a pairwise merge function to combine two records that match into a composite record. When two records r and r' are merged, the composite record r'' contains either $r.v$ or $r'.v$ as its value.

We measure the runtime in terms of the "critical" number of record comparisons. That is, for each synchronous step of joint ER, we add the maximum number of record comparisons among all parallel running processors. For example, if processor 1 ran 10 comparisons in the first and second iterations while processor 2 ran 9 and 11 comparisons, respectively, then the critical number of record comparisons is $max\{9,10\} + max\{10,11\} = 10 + 11 = 21$ comparisons. Although the record comparison time itself may change, we believe that counting the comparisons is a reasonable representation of the amount of work done.

6.1.1 Influence Graph Pattern

In this section, we study how our joint ER algorithm improves over a naïve implementation of joint ER that does not exploit the influence graph. In the naïve solution, all the datasets are repeatedly resolved until all of them converge. Notice that *all* the datasets need to be resolved for each step because, even if a dataset does not change in the current step, we do not know if a change in some other dataset might influence this dataset later on. Given a set $\mathcal{D}$ of datasets, the naïve solution is thus the most efficient way to resolve the execution plan $(\{\mathcal{D}\}+)$ without exploiting the influence graph. We use the default settings in Table 3 to construct our datasets and do not use expander functions.

When constructing the influence graph used by our joint ER algorithm, we consider two patterns: linear and random. In the linear model, the influence graph is a collection of "chains" where each chain contains datasets that form a linked list of influence in one direction. Given a maximum chain length of l, we use the first 10 datasets to create $n = \lfloor \frac{10}{l} \rfloor$ chains of length l and one more chain of length $10 - n \times l$ if $10 - n \times l > 0$. The remaining 5 datasets neither influence nor are influenced by other datasets. In the random model, we use a given probability c for creating a more connected structure. For any pair R and S in the first 10 datasets, R influences S with a probability of c. The remaining 5 datasets do not influence or are influenced by other datasets.

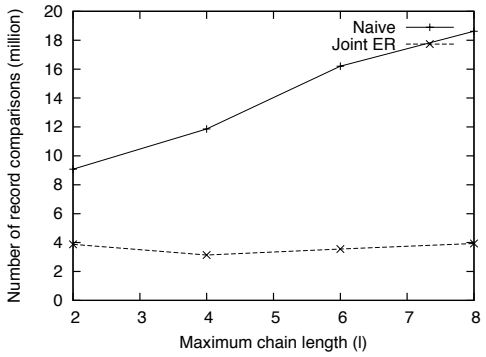

Fig. 4 Linear structure results

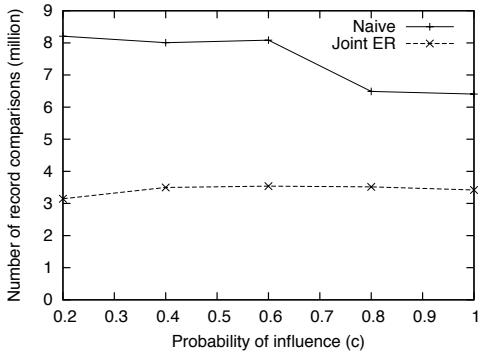

Fig. 5 Random structure results

Figure 4 shows how adjusting l in the linear model influences the critical number of record comparisons. Since at least five datasets are redundantly resolved for each step, the total work of the naïve solution increases linearly for larger l values. In comparison, the joint ER algorithm does an almost constant amount of work for any l by only resolving the necessary datasets at each step. As a result, the joint ER algorithm outperforms the naïve solution by 2.3–4.7x.

Figure 5 shows how adjusting c in a random model influences the critical number of record comparisons. If c is small, only a few datasets influence each other, so our joint ER algorithm can avoid redundantly resolving datasets by exploiting the influence graph. If c is larger, then the first 10 datasets are more likely to be connected with each other, and our joint ER algorithm performs similarly to the naïve solution for those 10 datasets. Notice that, since the remaining five datasets do not influence and are not influenced by other datasets, the joint ER algorithm outperforms the naïve solution by a certain degree even if $c = 1$. As a result, the joint ER algorithm outperforms the naïve solution by 1.8–2.6x.

In summary, the joint ER algorithm outperforms the naïve solution by exploiting the influence graph. If the influence graph has a linear structure, then the joint ER algorithm performs better for longer chains. If the influence graph has a random structure, then the joint ER algorithm performs better for sparser graphs.

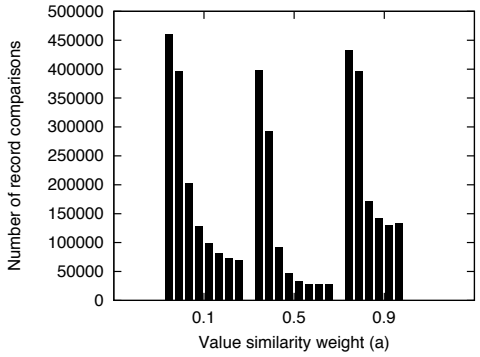

Fig. 6 Value similarity weights versus iterations

6.1.2 Number of Iterations

We study how certain parameters influence the number of iterations of the joint ER process. In order to be able to interpret our results more clearly, we focus on a single cycle with two datasets (called R and S) that influence each other. We then run ER alternatively on the two datasets until both of them converge.

Figure 6 shows how the value similarity weight a influences the number of iterations. We count each resolution of R or S as one iteration. For example, the resolution of R, S, and R is viewed as three iterations. Each group of bars show the results for one a value. The nth bar in a group shows the number of record comparisons for the nth dataset resolved. That is, the first bar represents the result of resolving R, the second bar the result of S, the third bar the result of R, and so on. Hence, the number of bars per group is the number of iterations it took for resolving R and S together. For example, if $a = 0.1$, there are eight iterations. As a increases, the value similarity of B becomes the dominant factor when comparing records. As a result, there are fewer iterations (if $a = 0.9$, there are only six iterations) because the overlap in references have less impact on the comparison of records.

6.1.3 Expander Function

We now study the performance of expander functions in various scenarios. We use an expander function E that receives a record $r \in R$ and returns all the records in R that have a value within the range of $[\lfloor \frac{r.v}{i} \rfloor \times i, \lfloor \frac{r.v}{i} \rfloor \times i + v]$, which covers all the records that represent the same entity as r by our construction. We prove that E is valid as long as $i > v$ and $t \geq a \times \frac{1}{i-v+1}$. Suppose that $i > v$. Then E returns exactly the records that

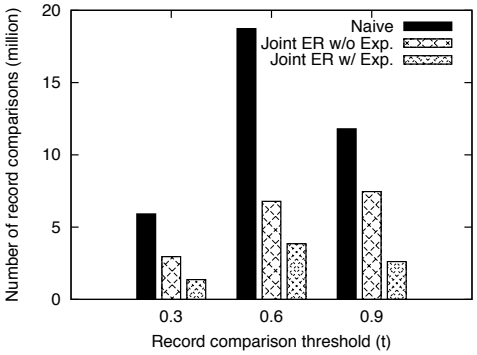

Fig. 7 Threshold versus expander function performance

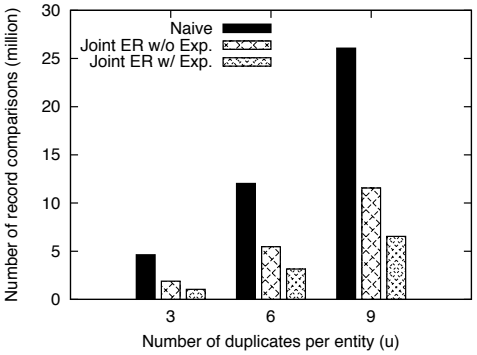

Fig. 8 Number of duplicates versus expander function performance

refer to the same entity as r. Also, for any two records r and r' in R that do not represent the same entity, $|r.v - r'.v| \geq i - v$. By construction of references, r and r' have no overlapping references to any other dataset, so the reference similarity is always 0. Hence, the weighted similarity of $B(r, r')$ is $a \times \frac{1}{|r.v - r.v'|+1} + (1-a) \times 0 \leq a \times \frac{1}{i-v+1}$. Hence, as long as $t > a \times \frac{1}{i-v+1}$, $B(r, r')$ is always false, and E is valid. Since E only returns the records that represent the same entity as r, E should be viewed as an optimal expander function that returns the best result possible. We compare the total number of critical comparisons among three methods: the naïve solution, the joint ER algorithm that does not use E, and the joint ER algorithm that uses E. We use a random model with an influence probability of $c = 0.3$ for generating the influence graph.

In Figure 7, we evaluate the performance of E by varying the comparison threshold t. For E to be valid in our default setting, we need t to be larger than $a \times \frac{1}{i-v+1} = \frac{1}{60}$. If $t = 0.3$, most records are likely to match with each other, so there are fewer record comparisons performed because of the frequent merging of records. As t increases to 0.6, fewer records start to match, so more record comparisons are performed with fewer merges. In addition, more iterations are needed to completely resolve all the datasets, which further increases the number of record comparisons. If t increases to 0.9, then very few records match, so there are more record comparisons. However, there are fewer iterations as well, so the number of record comparisons actually decreases for the naïve solution and the joint ER solution with expander functions. Throughout the three configurations of t, the joint ER algorithm with E outperforms the naïve solution and the joint ER algorithm without E by 4.3–4.9x and 1.8–2.9x, respectively. Since we are experimenting on a best-case expander function, the performance of using any other expander function should be somewhere between our results of running joint ER with and without E.

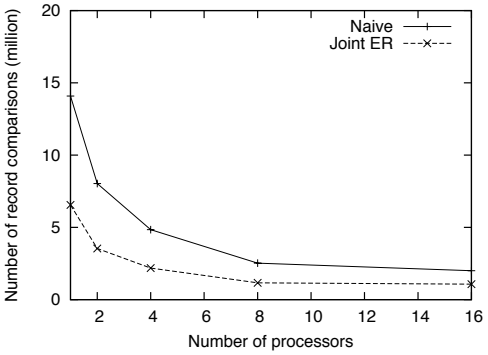

Fig. 9 Number of processors versus record comparisons

Next in Figure 8, we vary the number of duplicates per entity u from 3 to 9. As u increases, the number of record comparisons increases because there are more records to compare. However, the number of iterations does not change for different u values. In the three configurations of u, the joint ER algorithm with E outperforms the naïve solution and the joint ER algorithm without E by 3.8–4.4x and 1.7–1.8x, respectively.

In summary, the joint ER algorithm using expander functions can outperform the other two techniques for various joint ER scenarios.

6.1.4 Number of Processors

In Figure 9, we compare our joint ER algorithm with the naïve solution varying the number of processors p. Both plots are proportional to the plot $y = \frac{1}{p}$. If $p = 1$, joint ER outperforms the naïve solution by 2.1x in record comparisons. However, if $p = 16$, the naïve solution starts to perform relatively better (only 1.8x worse in record comparisons) because all datasets are being resolved at the same time, making convergence faster and thus reducing the number of iterations. Hence, our joint ER algorithms are more effective when there are fewer processors, i.e., when the resources are more scarce.

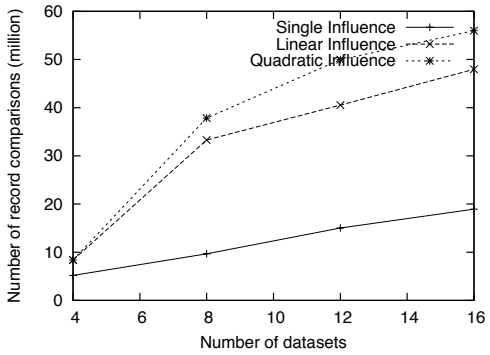

Fig. 10 Blocking scenario versus record comparisons

6.1.5 Exploiting Semantic Knowledge

We now resolve datasets using blocking techniques and show how reducing the edges in the influence graph by exploiting application semantics can improve the runtime performance. We experiment on 4–16 datasets that form a random influence graph where for each pair of datasets R and S, R influences S with probability 0.5. For half of the datasets, we increase each size from 1,000 to 10,000 records. We assume that only 1,000 records can be resolved in memory, so we divided each large dataset into 10 blocks of size 1,000. We use three scenarios for influence relations among blocks. In the first scenario, we assume a "quadratic" connection scenario where for every R that influences S, all the blocks of R influence S as well. In the case where S itself is divided into blocks, all the blocks of R influence each of the blocks of S. In the second scenario, we consider a "linear" connection scenario where each block in R only influences its corresponding block in S with the same index. In our third scenario, we consider an extreme "single" connection scenario where only one random block of R influences the corresponding block in S with the same index. Notice that the single connection scenario produces a best-possible influence graph that can be generated when R influences S. For the other parameters, we use the default settings in Table 3 to construct our datasets and do not use expander functions.

Figure 10 shows that, the single-connection scenario outperforms the quadratic-connection scenario by 1.6–3.9x and the linear-connection scenario by 1.6–3.4x depending on the number of datasets resolved. The improvement is due to the better exploitation of the influence graph by the scheduler.

6.2 Real Data Experiments

We now test the scalability of our joint ER algorithm on a real dataset provided by a commercial people search engine called Spock [28], which collects hundreds of millions of person information records from various websites such as Facebook, MySpace, and Wikipedia. The records are then resolved to generate one profile per person. The Spock schema contains various datasets that contain personal information, education, employment, addresses, and tags of people. We obtained a subset of the entire Spock dataset that contained information of about 70 million people whose names started with one of the characters 'c', 's', or 'k'. We chose this particular subset because relatively many people have names that start with the three characters, increasing the chance for finding duplicates [1].

In our experiments, we used a simplified version of the Spock schema and resolve the following types of records: persons, addresses, schools, and jobs. For our scalability experiments, we generated subsets of the data of different sizes, as follows. First we select a random subset (called P) of the desired size from the 70 million person records. Then we select the addresses (called A), schools (called S), and jobs (called J) that refer to at least one person among the randomly selected person records. The largest dataset we generated in this fashion contains 1M people records, 0.8M addresses, 3.5K schools, and 6.6K jobs. We also generated datasets with 0.25M, 0.5M and 0.75M people records.

Our datasets may not be representative of the Spock data in terms of the number of duplicates per entity because we take a random sample of the entire 70 million person records. For example, if there are 100 duplicates per entity, and we take a 10% sample of the entire data, then there would be on average only 10 duplicates per entity in the sample set. An alternative sampling method that preserves the number of duplicates per entity is to take a random subset of the actual entities and collect all the records that refer to those entities. Since we did not have a gold standard for the Spock data, however, determining the actual entities was challenging.

Each record contains either values or references to other types of records. All the letters in the records were converted to lowercase. An address record in A contains a street address, city, and state, but no attributes that refer to other types of records. A person record in P contains a first name, last name, gender, age, city, and state, and also contains attributes that refer to records in A, S, and J. A school record in S record contains a

[1] Spock was unable to give us all the data for legal reasons.

name and an attribute that refers to records in P. A job record in J also contains a name and an attribute that refers to records in P. As a result, we used the influence graph where A influences P, P and S influence each other, and P and J influence each other.

Since both A and P were too large to fit in memory, we used blocking techniques to divide the two datasets. The blocking on A was based on the first character of the city appended with the state for each address record. For example, if an address record has the city "stanford" and state "ca", then the blocking key is "sca". For the records that did not contain a state, there were only a few thousand of them so they were all put in a single block containing records without states. As a result, we may miss matches between records that have a state and those that do not have a state. For example, an address in "atlanta, ga" will never be compared with a record in "atlanta" without a state although the two addresses may be the same. On the other hand, the blocking prevents unnecessary matches that may occur due to the sparse information. That is, if the second "atlanta" was in the state "tx", then the two addresses should not be compared even if they look similar. The blocking on P was based on the first two characters of the last name appended with the state for each person record. For example, if a person record has the last name "smith" and the state "tx", then the blocking key is "smtx". (We used two characters of the last name instead of one because all last names started with a 'c', 's', or 'k'.) The records without a state were distributed by the first two characters of their last names into blocks containing records without states. The S and J datasets were small enough to fit in one block each. While the blocks containing the records were stored on disk, the influence graph and the information of which records were clustered together was kept in memory.

We generate an execution plan using Algorithm 1. In our experiments, A was divided into 10 blocks $A_1, \ldots, A_{10}$, and P was divided into another 10 blocks $P_1, \ldots, P_{10}$. Since A influences P, all the blocks of A influence all the blocks of P. In addition, all the blocks of P influence S and J, and vice versa. Hence, the resulting execution plan is $(\{A_1, \ldots, A_{10}\}\ \{P_1, \ldots, P_{10}, S, J\}+)$.

We use fixed-sized extents called segments to store the blocks on disk [33]. A segment acts as a unit of transfer for reading blocks into memory and thus cannot exceed the memory size. We allocate a fixed number of segments consecutively on disk and then randomly assign the blocks to the segments. The advantage of using segments is that the different block sizes are evened out when they are randomly assigned to the segments. As a result, we can approximately do the same amount of work for each segment processed.

We use two ER algorithms to demonstrate that our framework can use different ER algorithms for each type of data.

- The Sorted Neighbor (SN) algorithm (see Section 4.4) was used for the A and P datasets. When resolving A, we sorted the records by their cities and then used a sliding window of size 100 for comparing the records. When comparing two address records, we performed a string similarity comparison of the street address using the Jaro distance function [34]. We considered the records to match if either the street addresses were near identical or if the street addresses were similar and the states were the same. When resolving P, we sorted the records by their last names and then used a sliding window of size 100 for comparing the records. When comparing two person records, we compared the appended first and last names using the Jaro distance function. If the names were similar, we also checked if the two records had the same age and gender or the same city and state or the same school or the same job to determine a match.
- The R-Swoosh algorithm (see Section 6.1) was used for resolving the S and J datasets. For both S and J, two records were considered to match if they either had near-identical names or had similar names and referred to at least one common P record.

Our setting thus illustrates the flexibility of our framework where one can plug in any ER algorithm for each dataset resolved.

Figure 11 shows the runtime of joint ER as the number of person records increases. The sizes of the other types of data are not shown in the x-axis, but increase in proportion to the number of person records. The different plots show the results for using 1–4 concurrent threads. For any number of threads, the joint ER runtime increases linearly to the number of records resolved, mainly because the SN algorithm has linear scalability. As the number of threads increases, the runtime improves in a sub-linear fashion. For example, the runtime for resolving the Spock data with 1M person records improves by 1.4x when increasing the number of threads from 1 to 2, but only improves by 1.1x when increasing the number of threads from 2 to 4. The main reason is that the block sizes were not evenly distributed, so the workload of record comparisons was not evenly distributed to the threads as well. If all the blocks had the same size, then we would see runtime improvements more proportional to the number of threads. In addition, there is a fixed cost of initially distributing the records to the blocks on disk, which further reduces the benefit of concurrent pro-

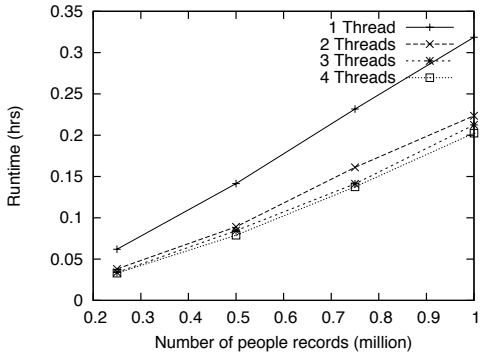

Fig. 11 Scalability results on the Spock dataset

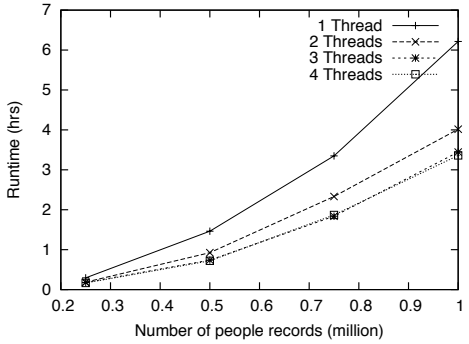

Fig. 12 Scalability results when running R-Swoosh on P

cessing. Nevertheless, the results show that our joint ER algorithm can scale to millions of records.

Figure 12 shows the scalability results as above, except that the largest dataset P is now resolved with the R-Swoosh algorithm instead of the SN algorithm. Since R-Swoosh has a quadratic complexity, the joint ER runtime increases quadratically as the number of person records increases. Compared to Figure 11, the joint ER runtimes are about an order of magnitude larger as well. The results show that the joint ER scalability heavily depends on performances of the specific ER algorithms plugged into the framework.

We now study the physical execution of running joint ER on the Spock data. Instead of showing all the details of the physical execution, we summarize by showing the number of segments that were read into memory for each dataset resolved. (Recall that a segment is a unit of transfer for reading blocks into memory.) For example, the summarized physical execution ((*R*:1), (*S*:2, *T*:3)) says that one segment containing the records of R was read into memory for resolution, and then two segments of S and three segments of T were read into memory for resolution. The first row of Table 4 shows the physical execution using the setting of Figure 11 for resolving the Spock data with 1M people records using one thread where the SN algorithm was used to resolve P. The second row shows the execution when we use the setting of Figure 12 for resolving the same data using one thread where the R-Swoosh algorithm was used to resolve P. Compared to the physical execution in the first row, one fewer segment of P was read into memory for the second row because R-Swoosh is more "optimistic" in matching records compared to the SN algorithm and thus finds all the matching records early on. Both executions do not have many iterations because there were relatively few matching records. On average, 5% of the records in A clustered with other records for each block resolved (including repetitions where some blocks were resolved multiple times). For the other datasets, the percentages were 0.5–1.1%, 0.2–0.3%, and 0.4% for P, S, and J, respectively.

Setting	Physical Execution Summary
Figure 11	((*A*:40), (*P*:40,*S*:1,*J*:1), (*P*:26,*S*:1,*J*:1), (*P*:1))
Figure 12	((*A*:40), (*P*:40,*S*:1,*J*:1), (*P*:26,*S*:1,*J*:1))

Table 4 Physical Execution Summary on Spock Data

6.3 Training Accuracy

We now address the question on whether state-based training can improve ER accuracy. If training the comparison rules once already produces perfect ER results, then there is no need to train the comparison rule multiple times. However, if producing correct ER results depends on the resolution of other datasets, then our techniques may help.

6.3.1 Setting

Data We evaluate state-based training using real data. We experiment on the Cora dataset, which is a publicly available list of 1,879 papers. From the paper records, we extracted 1,879 venues into a separate list. The gold standard for resolving the papers is given within the Cora dataset. For the venues, we manually created a gold standard by grouping the same venues together.

Comparison Rule We train ER algorithms that use *comparison rules* to decide if records represent the same real world entity. A comparison rule B is defined as a Boolean function that takes two records and returns true or false. We assume that B is commutative, i.e., $\forall r_i, r_j, B(r_i, r_j) = B(r_j, r_i)$. For example, two person records may match according to B if their names and addresses are similar. How exactly an ER algorithm uses B to derive the output partition P'_R depends on the specific ER algorithm used. For example, say that records r and s match, s and t match, but r and t do

Rule	Definition
B_P	$w_0 + w_1 \times S_T + w_2 \times S_A + w_3 \times O_V \geq 0$
B_V	$w_0 + w_1 \times S_N + w_2 \times O_P \geq 0$

Table 5 Cora data comparison rules

not match according to a comparison rule B. If the ER algorithm performs a connected component operation on the matching records, then the output is $\{\{r, s, t\}\}$. On the other hand, if the ER algorithm only clusters the records that all match with each other, then the answer may now be $\{\{r, s\}, \{t\}\}$ or $\{\{r\}, \{s, t\}\}$.

We assume that each record $r \in R$ consists of a set of attributes and that a comparison rule has the following form: $B(r, s) = w_0 + \Sigma_{i=1,\ldots,k} w_i \times Sim_i(r, s) \geq 0$. Each similarity function Sim_i computes the similarity of r and s using the attributes in r and s.

Table 5 shows the comparison rules for the papers and venues. We denote the paper dataset as P and the venue dataset as V. The paper comparison rule B_P compares the similarity of titles (S_T), the similarity of author lists (S_A), and the venue overlap (O_V) between papers. The venue comparison rule B_V compares the similarity of names (S_N) and the paper overlap (O_P) between venues. When computing the similarity for titles and names, we removed special characters and used the Jaro measure [34] to compute a string similarity ranging from 0 to 1. When comparing the author lists, we extracted the last names of the authors, sorted and concatenated them in alphabetical order, and used the Jaro distance to compute the string similarity. We emphasize that our comparison rules are not designed to be the most accurate rules. There are other attributes (date, publisher, volume, page numbers, etc.) that we could have exploited and better models [23] for constructing the comparison rules. Hence, we use our simple rules to clearly demonstrate how the joint training and resolution of datasets can significantly enhance accuracy compared to resolving each dataset individually.

Training We train the weights of B_P and B_V using a standard SVM method with a linear kernel function [19]. For each record pair (r, s), the attribute similarities form a feature vector $f(r, s) = (Sim_1(r, s), \ldots, Sim_k(r, s))$ that represents (r, s). For example, $B_P(r, s)$ has the feature vector $(S_T(r, s), S_A(r, s), O_V(r, s))$. We train the weights assuming that all the records in the same cluster of the gold standard match with each other. For example, if the gold standard of R is $G_R = \{\{r, s, t\}, \{u, v\}\}$, then we train the weights using the matching pairs $\{r - s, s - t, r - t, u - v\}$.

We create the training set $\mathcal{D}'$ by taking 10% random subsets of the papers and venues. We use the two following training schemes:

Iter.	Rule	State	w_0	w_1	w_2	w_3
Paper weights trained first						
1	B_P	()	-7.89	8.02	0.92	0
2	B_V	$(((P)))$	-8.99	9.99	2.22	n/a
3	B_P	$(((P)), ((V)))$	-6.71	6.39	0.79	0.76
4	B_V	$(((P)), ((V)), ((P)))$	-8.99	9.99	2.22	n/a
Venue weights trained first						
1	B_V	()	-8.42	9.42	0	n/a
2	B_P	$(((V)))$	-8.1	7.73	1.36	0.83
3	B_V	$(((V)), ((P)))$	-8.99	9.99	2.22	n/a
4	B_P	$(((V)), ((P)), ((V)))$	-6.39	5.25	1.44	0.89
5	B_V	$(((V)), ((P)), ((V)), ((P)))$	-8.99	9.99	2.22	n/a

Table 6 Trained weights

1. *state-based training*: The weights of the ER algorithms are trained for each possible state that occurs while resolving $\mathcal{D}'$.
2. *training without states*: The weights of the ER algorithms are trained only once using the initial partitions of the datasets in $\mathcal{D}'$.

When performing state-based training for the papers and venues, we consider the scenario where one processor alternatively resolves the papers and venues until both of them converge. Hence, we also train the papers and venues in an alternating fashion. For example, we can train the paper weights, and then the venue weights, then the paper weights again, and so on until all the weights converge. The top part of Table 6 shows the trained weights of B_P and B_V where the paper weights were trained first. For each state, we have listed the datasets resolved. Notice that after the first iteration, w_3 of B_P is trained to 0 because none of the venues have been merged yet, so comparing the venues of papers does not improve accuracy. The bottom part of Table 6 shows the weights when the venues are trained first. This time, in the first iteration, the weight w_2 of B_V is trained to 0 because none of the papers have been resolved and merged yet. Notice that within three resolutions of the papers and venues, we have produced a fixed point joint ER result, which suggests that the number of additional resolutions per dataset is small in practice.

Table 7 shows the trained weights without considering the states of resolved datasets. When training the paper weights, since no venues have merged, the training results are identical to the paper weights after the first iteration in the top part of Table 6. Similarly, for the venues, the training results are the same as the venue weights after the first iteration in the bottom part of Table 6.

Evaluation In order to evaluate our training techniques, we first train on the sample datasets $\mathcal{D}'$ and then run joint ER on the entire datasets $\mathcal{D}$. For each iteration

Rule	w_0	w_1	w_2	w_3
B_P	-7.89	8.02	0.92	0
B_V	-8.42	9.42	0	n/a

Table 7 Trained weights without states

Iteration	Type	Pr	Re	F1
Papers resolved first				
1	P	0.83	0.92	0.87
2	V	0.69	0.88	0.78
3	P	0.83	0.96	0.89
4	V	0.69	0.88	0.78
5	P	0.83	0.96	0.89
Venues resolved first				
1	V	0.94	0.54	0.69
2	P	0.83	0.96	0.89
3	V	0.84	0.89	0.86
4	P	0.83	0.96	0.89
5	V	0.84	0.89	0.86

Table 8 Accuracy results of state-based training

of joint ER, we compare the accuracy results of state-based training and training without states.

To measure accuracy, we compare the ER results with the gold standards of the datasets in $\mathcal{D}$. We consider all the input records that clustered together to be identical to each other. For instance, if records r and s clustered into $\{r, s\}$ and then clustered with t, all three records r, s, t are considered to be the same. Suppose that the set G contains the set of record pairs that cluster in the gold standard while set S contains the matching pairs for our algorithm. Then the precision Pr is $\frac{|G \cap S|}{|S|}$ while the recall Re is $\frac{|G \cap S|}{|G|}$. Using Pr and Re, we compute the F_1-measure, which is defined as $\frac{2 \times Pr \times Re}{Pr + Re}$, and use it as our accuracy metric.

6.3.2 Results

Using the trained weights, we now run the Swoosh algorithm on each dataset following the same sequence as that of the training. Hence if B_P was trained first, we run ER on the papers first, then run ER on the venues, and then on the papers, and so on until the ER results converge. If B_V was trained first, we run ER on the venues first. The top part of Table 8 shows the progression of accuracy as each dataset is resolved when the papers are resolved first. The bottom part of Table 8 shows the corresponding results when the venues are resolved first. As a result, the accuracy of the resolved papers is 0.89 regardless which dataset was resolved first while the accuracy of the resolved venues is 0.78 if the papers were resolved first and 0.86 if the venues were resolved first.

We now compare our state-based training results with the training results without states. Table 9 shows

Type	Pr	Re	F1
P	0.83	0.92	0.87
V	0.94	0.54	0.69

Table 9 Accuracy results of training without states

the accuracy results when the states are not considered. Notice that the paper and venue results are identical to the first iteration results of the top and bottom parts of Table 8, respectively. As a result, using the state information has improved the accuracy of papers by 2% and that of the venues by 9–17%. The larger improvement for venues matches our intuition because many identical venues that have significantly different names (e.g., the names "NIPS" and "Advances in Neural Information Processing Systems" represent the same conference) started to match once they shared many papers.

In summary, we have shown that state-based training can improve the accuracy of ER compared to training without states. In general, state-based training does not always outperform training without states. For example, if training without states already produces very good comparison rules, then there is no benefit in retraining the rules. Hence, state-based training is mainly useful when references to other record types play a significant role in matching records as in our demonstration.

7 Related Work

Originally introduced by Newcombe et al. [21] as "record linkage", entity resolution has been studied under various names, such as merge/purge [17], deduplication [26], reference reconciliation [12], object identification [30], and others (see [9,13,34] for recent surveys). Most of the ER works [18] have focused on resolving one dataset. In contrast, our approach attempts to resolve multiple datasets of records at the same time, which can significantly improve the accuracy of ER. Although recent work [33] has proposed general scalable ER algorithms for a single entity type of data, it does not discuss how to coordinate the resolution of multiple types of records that refer to each other.

Many works have considered joint ER focusing on accuracy only. Dong et al. [12] presents an ER method where record pairs of different datasets are resolved simultaneously. Bhattacharya et al. [6] proposes joint ER techniques for a specific domain (citations) using hard-coded ER algorithms. Several probabilistic models for joint ER has been proposed as well. Culotta et al. [10, 11] uses conditional random fields to improve the propagation of information when resolving records of multiple datasets. Domingos et al. [22] supports a restricted

version of joint ER where comparison results of basic attributes are shared when resolving records. Markov Logic Networks [27] can also be used to specify a rich set of constraints for joint ER. Recently, Sadinle et al. [25] extended the traditional Fellegi and Sunter classification model [14] to more than two datasets. While the above approaches significantly enhance the accuracy of joint ER, they do not scale to large datasets and do not provide a general framework for custom ER algorithms either.

Arasu et al. [2] has proposed joint ER techniques based on a declarative language for constraints. Unlike previous approaches, their work focuses on scalability as well as accuracy of joint ER. A declarative set of constraints is reflected in a graph of records with different types of edges that indicate the likelihood of two records matching. Then a correlation clustering algorithm is used to connect records while minimizing the number of violated constraints in the graph. In comparison, our work provides generality where one can simply plug in her application-specific ER algorithm and get joint ER results. In addition, our joint ER algorithm can resolve few datasets at a time based on an execution plan, providing flexibility in resource (e.g., CPU, memory) management.

A recent line of work proposes collective ER techniques that are scalable as well. (The notion of collectively resolving records is similar to joint ER, but has traditionally focused more on accuracy.) Bhattacharya et al. [6] compares records not only by their attributes, but also by their relational similarities. Hence, matching records can influence the resolution of other records as well. A priority queue is used to iteratively resolve one record pair at a time, and blocking techniques are used to reduce the total number of record comparisons. While all the records are resolved in concert, they all are of the same type. In comparison, our joint ER framework extends the ER model by resolving multiple types of datasets (e.g., authors, papers, institutions, and venues) together using different ER algorithms. In addition, we significantly improve the priority queue by proposing a scheduling scheme that takes into account the influence graph of datasets, multiple processors, and limited memory.

Another relevant work on collective ER was proposed by Rastogi et al. [24]. Here, the entire set of records is divided into covers (similar to overlapping blocks), where one cover is resolved at a time, and the resolution of one cover can influence the resolution of other covers. Message passing is used to convey the matching information of records across covers. The covers are iteratively resolved using a priority queue. A complete and formal analysis is performed on the soundness of the message passing scheme. While the authors of [24] mention that their priority queue scheduling can be run in parallel, little detail of the execution is provided. In comparison, our joint ER framework explicitly addresses parallel processing by defining logical and physical executions and incorporating performance factors related to parallelism into the scheduling. By dividing records of different types into separate datasets, we can exploit the influence relationships between the datasets for efficient scheduling.

Job scheduling [8] and software pipelining [1] are related topics to our problem of assigning datasets to processors. The goal is to assign fixed-length jobs (in software pipelining, an instruction is a job) that may depend on each other to processors in order to minimize the parallel runtime. While joint ER can also be viewed as a job scheduling problem, a major distinction is that it is very difficult to predict the runtime of ER on each dataset unlike job scheduling and software pipelining where each job has a fixed runtime. In addition, even if a dataset R influences S, we are not necessarily restricted to resolving S after R unlike in job scheduling and software pipelining where jobs must be processed by strictly following a dependence relation.

Several works have proposed training techniques for ER algorithms. Bilenko et al. [7] provides training for the specific domain of string similarity measures. Sarawagi et al. [26] proposes interactive learning techniques that minimize the work needed to achieve high accuracy of ER. Bhattacharya et al. [5] proposes unsupervised learning techniques for ER. While most of these works perform training once, we propose a state-based training technique where an ER algorithm of a dataset may be trained multiple times based on different resolution states of other datasets. State-based training can be broadly viewed as a general data mining operation like the SEMMA method [3].

This paper significantly extends a previous conference publication [32]. Most notably, we add optimization techniques for efficient execution plans (Section 3), discuss the convergence of resolving fixed-point sets (Section 4), propose training techniques for joint ER algorithms (Section 5), and perform extensive experiments including state-based training (Section 6).

8 Conclusion

When performing entity resolution on multiple types of datasets, resolving records of one type can impact the resolution of other types of records. In this paper, we have explored the following questions: given a limited amount of resources, what is the most efficient schedule

for resolving the datasets? How do we train the ER algorithms? We have answered the first question by proposing a flexible and modular resolution framework where existing ER algorithms that are developed for given record types can be plugged in and used with other ER algorithms. In our experiments, we have demonstrated that our joint ER method can resolve large datasets efficiently by scheduling and coordinating the individual ER algorithms. We have answered the second question by proposing a state-based training technique where ER algorithms can be trained multiple times based on the state of other datasets. We have demonstrated that our state-based training techniques can indeed improve the accuracy compared to training the ER algorithms only once.

9 Acknowledgements

We thank Makoto Tachibana and David Menestrina for their early support on the project.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley (2006)
2. Arasu, A., Ré, C., Suciu, D.: Large-scale deduplication with constraints using dedupalog. In: ICDE, pp. 952–963 (2009)
3. Azevedo, A., Santos, M.F.: Kdd, semma and crisp-dm: a parallel overview. In: IADIS European Conf. Data Mining, pp. 182–185 (2008)
4. Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Whang, S.E., Widom, J.: Swoosh: a generic approach to entity resolution. VLDB J. **18**(1), 255–276 (2009)
5. Bhattacharya, I., Getoor, L.: A latent dirichlet model for unsupervised entity resolution. In: SDM (2006)
6. Bhattacharya, I., Getoor, L.: Collective entity resolution in relational data. TKDD **1**(1) (2007)
7. Bilenko, M., Mooney, R.J.: Adaptive duplicate detection using learnable string similarity measures. In: KDD, pp. 39–48 (2003)
8. Brucker, P.: Scheduling algorithms (4. ed.). Springer (2004)
9. Christen, P.: Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Data-Centric Systems and Applications. Springer (2012)
10. Culotta, A., Mccallum, A.: A conditional model of deduplication for multi-type relational data. Tech. rep., Univ. of Massachusetts (2005)
11. Culotta, A., McCallum, A.: Joint deduplication of multiple record types in relational data. In: CIKM, pp. 257–258 (2005)
12. Dong, X., Halevy, A.Y., Madhavan, J.: Reference reconciliation in complex information spaces. In: SIGMOD, pp. 85–96 (2005)
13. Elmagarmid, A.K., Ipeirotis, P.G., Verykios, V.S.: Duplicate record detection: A survey. IEEE Trans. Knowl. Data Eng. **19**(1), 1–16 (2007)
14. Fellegi, I.P., Sunter, A.B.: A theory for record linkage. Journal of the American Statistical Association **64**(328), 1183–1210 (1969)
15. Garey, M.R., Johnson, D.S., Sethi, R.: The complexity of flowshop and jobshop scheduling. Mathematics of Operations Research **1**, 117–129 (1976). DOI 10.1287/moor.1.2.117
16. Graham, R.L., Grahamt, R.L.: Bounds on multiprocessing timing anomalies. SIAM Journal on Applied Mathematics **17**, 416–429 (1969)
17. Hernández, M.A., Stolfo, S.J.: The merge/purge problem for large databases. In: SIGMOD, pp. 127–138 (1995)
18. Köpcke, H., Thor, A., Rahm, E.: Evaluation of entity resolution approaches on real-world match problems. PVLDB **3**(1), 484–493 (2010)
19. Lib{SVM}. Http://www.csie.ntu.edu.tw/ cjlin/libsvm/
20. Newcombe, H.B., Kennedy, J.M.: Record linkage: making maximum use of the discriminating power of identifying information. Commun. ACM **5**(11), 563–566 (1962)
21. Newcombe, H.B., Kennedy, J.M., Axford, S.J., James, A.P.: Automatic linkage of vital records. Science **130**(3381), 954–959 (1959)
22. Parag, Domingos, P.: Multi-relational record linkage. In: KDD-2004 Workshop on Multi-Relational Data Mining, pp. 31–48 (2004)
23. Poon, H., Domingos, P.: Joint inference in information extraction. In: AAAI, pp. 913–918 (2007)
24. Rastogi, V., Dalvi, N.N., Garofalakis, M.N.: Large-scale collective entity matching. PVLDB **4**(4), 208–218 (2011)
25. Sadinle, M., Hall, R., Fienberg, S.E.: Approaches to multiple record linkage. In: ISI (2011)
26. Sarawagi, S., Bhamidipaty, A.: Interactive deduplication using active learning. In: KDD, pp. 269–278 (2002)
27. Singla, P., Domingos, P.: Entity resolution with markov logic. In: ICDM, pp. 572–582 (2006)
28. Spock. Http://spock.com
29. Tarjan, R.E.: Edge-disjoint spanning trees and depth-first search. Acta Inf. **6**, 171–185 (1976)
30. Tejada, S., Knoblock, C.A., Minton, S.: Learning object identification rules for information integration. Information Systems Journal **26**(8), 635–656 (2001)
31. Whang, S.E., Garcia-Molina, H.: Entity resolution with evolving rules. PVLDB **3**(1), 1326–1337 (2010)
32. Whang, S.E., Garcia-Molina, H.: Joint entity resolution. In: ICDE, pp. 294–305 (2012)
33. Whang, S.E., Menestrina, D., Koutrika, G., Theobald, M., Garcia-Molina, H.: Entity resolution with iterative blocking. In: SIGMOD, pp. 219–232 (2009)
34. Winkler, W.: Overview of record linkage and current research directions. Tech. rep., Statistical Research Division, U.S. Bureau of the Census, Washington, DC (2006)