Joint Entity Resolution

Steven Euijong Whang, Hector Garcia-Molina

Computer Science Department, Stanford University
353 Serra Mall, Stanford, CA 94305, USA
{swhang, hector}@cs.stanford.edu

***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. We then evaluate our joint ER techniques on synthetic and real data and show the scalability of our approach.**

I. Introduction

Entity Resolution (ER) (sometimes 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 [1], [2], [3], [4], [5], [6], 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 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.

Paper	Title	Venue
p_1	The Theory of Joins in Relational Databases	v_1
p_2	Efficient Optimization of a Class of Relational ...	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 I
Papers and Venues

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 [7] 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 I) that contain paper records and venue records, respectively. (Note that Table I 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.

If 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 resolving the venues 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. Although not presented here due to space constraints, we propose in our extended report [8] a state-based training method that trains 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 II).
- 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 III).
- 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 IV).
- We experiment on both synthetic and real data to demonstrate the behavior and scalability of joint ER (Section V).

II. 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 III-A), 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.

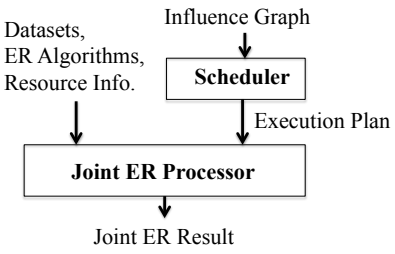

Fig. 1. System Architecture

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 ER algorithms. Finally, we formalize physical executions of ER algorithms that can return joint ER results.

A. 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. 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 IV-D.) 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 I, 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.

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 2.1: 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 2.1 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 (possibly overlapping) 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.

B. Joint ER Model

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

Definition 2.2: 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 2.1, 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 2.2, $\{P'_R, P'_S\}$ 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 2.1.

C. 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.

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 IV, 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.

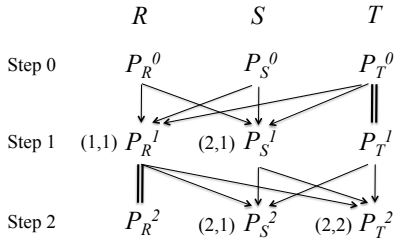

Fig. 2. The Physical Execution (((R), (S)),((),(S, T)))

The physical execution information can be compactly expressed as a nested list of three levels where the outer-most 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 2.2. We capture this desirable property in the following definition.

Definition 2.3: 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 I, 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. The proof is in our extended report [8].

Proposition 2.4: Running valid ER algorithms on a valid physical execution of $\mathcal{D}$ returns a joint ER result of $\mathcal{D}$ satisfying Definition 2.2.

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 2.2 can be produced by valid physical executions. The reason is that Definition 2.2 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 2.2.

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 2.5: 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 [9], 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 III) and then produce feasible and efficient physical executions based on the execution plan (Section IV).

III. 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.

A. 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 3.1: 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.

In our motivating example of Section I, 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 3.1, P influences V. An influence graph could possibly be generated automatically based on the ER algorithms. For example, the scheduler may view 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 $S \rightarrow T$.

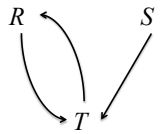

Fig. 3. An Influence Graph

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 2.2. In the next section, we define the execution plan as a way to capture multiple possible 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. 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 III-B. 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.

B. Execution Plan

While an influence graph can be used to directly generate a physical execution, it is more convenient to add a level of abstraction and generate 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.

1) Syntax and Properties: An execution plan L is a sequence of concurrent and fixed-point sets (see Table II). 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 3.2: 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 III-B2 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 IV-B, we show other techniques for generating physical executions of fixed-point sets that can improve the physical execution runtime.)

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

TABLE II
EXECUTION PLAN SYNTAX

We can access a concurrent or fixed-point set of an execution plan using a list index notation. For instance, 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 2.3. 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 3.3: 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 3.3). 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 4.2 (see Section IV), 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.

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. The proof is in our extended report [8].

Lemma 3.4: Given an influence graph G of the datasets $\mathcal{D}$, the execution plan $(\{\mathcal{D}\}+)$ conforms to G.

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 2.2 (although it 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.

We now propose a construction algorithm (called CP) that produces an execution plan that conforms to a given influence graph G. The full algorithm is in our extended report [8]. To illustrate our algorithm, suppose that the influence graph G contains four datasets R, S, T, and U and has three edges $R \rightarrow T$, $T \rightarrow R$, and $S \rightarrow T$. We first identify the strongly connected components of G, which are $\{R, T\}$, $\{S\}$, and $\{U\}$. Next, 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. We then 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 the execution plan L the combined set $\{S, U\}$. Next, we update Z to $\{\{R, T\}\}$. Since the node $\{R, T\}$ has two datasets, we add it to L as a fixed-point set $\{R, T\}+$. As a result, our final execution plan is $L = (\{S, U\}, \{R, T\}+)$.

We can show that the execution plan L returned by the CP algorithm conforms to G and satisfies the following desirable properties (see our extended report [8] for more details):

- No Redundant Datasets: Every dataset is in at most one concurrent or fixed-point set in L.
- No Large Fixed-Point Sets: No fixed-point set in L can be divided into smaller concurrent and fixed-point sets while L still conforms to G.
- Maximized Parallelism: For each concurrent or fixed-point set resolved, we resolve as many datasets that are not influenced by other unresolved datasets as possible.

The proof for the efficiency of the CP algorithm is in our extended report [8].

Proposition 3.5: Given an influence graph G with V vertices and E edges, the CP algorithm runs in $O(|V| + |E|)$ time.

IV. Joint ER Processor

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

A. 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 so the overall runtime is minimized.

We use the List scheduling algorithm [10] 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.

B. 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 IV-A. Say that only R and S had new partitions. Then we only need to resolve T in the next step. If T no longer has new partitions after its resolution, then we terminate.

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. The proof is in our extended report [8].

Proposition 4.1: 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 3.2.

C. 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 IV-A and IV-B, respectively.

We show that the resulting physical execution is valid as long as it terminates. The proof is in our extended report [8].

Proposition 4.2: 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.

D. 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 I, $\mathcal{R}_P(v_1) = \{p_1, p_2\}$ 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 [11].

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 4.3: 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. A detailed illustration of an expander function is in our extended report [8]. We note that not all ER algorithms have a valid expander function that returns a set $X_R(M)$ smaller than P_R.

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 III
PARAMETERS FOR GENERATING SYNTHETIC DATA

V. 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 [12]). 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.

A. 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 III 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 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 IV-D, 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 $N(R)$ 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 [13], 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.

We now evaluate our joint ER algorithm against the synthetic datasets. Although not presented here due to space restrictions, we also show in our extended report [8] how reducing the edges in the influence graph by exploiting application semantics can improve the runtime performance.

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 III 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.

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.

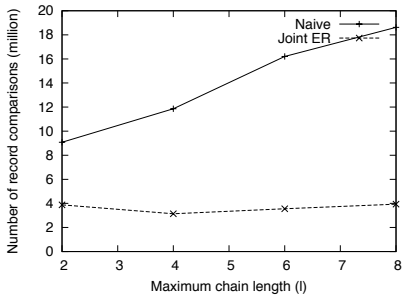

Fig. 4. Linear structure results

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.

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

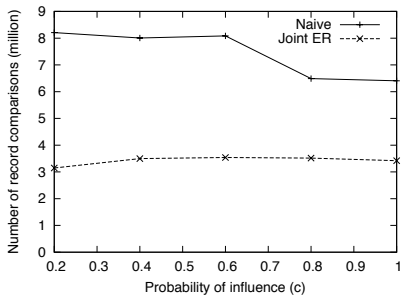

Fig. 5. Random structure results

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.

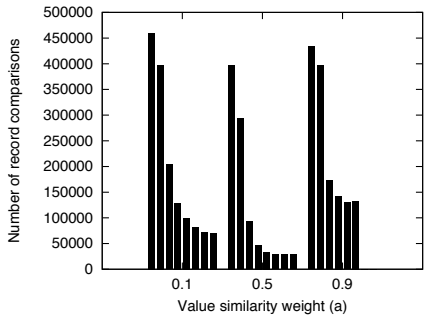

Fig. 6. Value similarity weights versus iterations

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. In our extended report [8], we prove that E is valid as long as $i > v$ and $t \geq a \times \frac{1}{i-v+1}$. In fact, E only returns the records that represent the same entity as r, so 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.

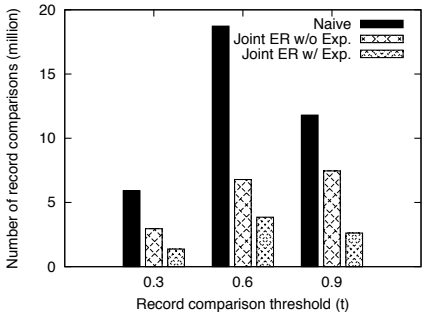

Fig. 7. Threshold versus expander function performance

In addition, when varying the number of duplicates per entity u from 3 to 9, the joint ER algorithm with E outperforms the naïve solution and the joint ER algorithm without E by 4.3–5x and 1.6–1.9x, respectively (see our extended report [8] for details). Hence, the joint ER algorithm using expander functions can outperform the other two techniques for various joint ER scenarios.

4) Number of Processors: In Figure 8, 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.

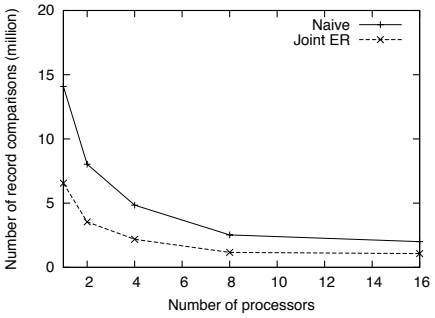

Fig. 8. Number of processors versus record comparisons

B. 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 [12], 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. Since both A and P were too large to fit in memory, we used blocking techniques to divide the two datasets (see our extended report [8] for more details).

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 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. Hence, the execution plan generated by the CP algorithm was $(\{A\}\{P, S, J\}+)$.

We use two ER algorithms to demonstrate that our framework can use different ER algorithms for each type of data. Our setting illustrates the flexibility of our framework where one can plug in any ER algorithm for each dataset resolved.

The Sorted Neighbor (SN) algorithm (see Section IV-D) 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 [14]. 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 V-A) 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.

Figure 9 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 processing. Nevertheless, the results show that our joint ER algorithm can scale to millions of records.

In our extended report [8], we also show the scalability results when P is 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 9, 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

[1] Spock was unable to give us the full dataset for legal reasons.

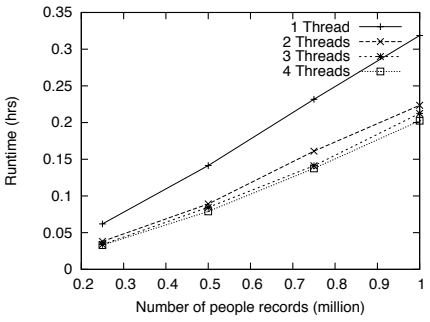

Fig. 9. Scalability results on the Spock dataset

into the framework.

VI. Related Work

Entity resolution has been studied under various names including record linkage, merge/purge, deduplication, reference reconciliation, object identification, and others (see [15], [14] for recent surveys). Most of the ER works [16] 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 several recent works [17], [18] have proposed general scalable ER algorithms for a single entity type of data, they do not discuss how to coordinate the resolution of multiple types of records that refer to each other.

Many works have considered joint ER mainly focusing on accuracy. Dong et al. [1] presents an ER method where record pairs of different datasets are resolved simultaneously. Bhattacharya et al. [2] proposes joint ER techniques for a specific domain (citations) using hard-coded ER algorithms. Several probabilistic models [3], [4], [5] for joint ER using conditional random fields and markov logic networks have been proposed as well. While the above approaches significantly enhance the accuracy of joint ER, they do not scale to large datasets. More recently, Arasu et al. [6] have 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. In comparison, our work provides modularity where one can simply plug in her application-specific ER algorithm and get joint ER results. In addition, our framework supports the scheduling and coordination of the ER algorithms.

Job scheduling [19] and software pipelining [20] 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.

VII. 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 problem of finding the most efficient schedule for jointly resolving the datasets given a limited amount of resources. We have proposed 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. We have shown in our synthetic data experiments that joint ER can be efficient by exploiting the influence graph and utilizing expander functions. We have also shown with real data that joint ER can scale to millions of records provided that the ER algorithms are efficient.

VIII. Acknowledgements

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

References

[1] X. Dong, A. Y. Halevy, and J. Madhavan, "Reference reconciliation in complex information spaces," in *SIGMOD Conference*, 2005, pp. 85–96.
[2] I. Bhattacharya and L. Getoor, "Collective entity resolution in relational data," *TKDD*, vol. 1, no. 1, 2007.
[3] A. Culotta and A. Mccallum, "A conditional model of deduplication for multi-type relational data," Univ. of Massachusetts, Tech. Rep., 2005.
[4] Parag and P. Domingos, "Multi-relational record linkage," in *In Proceedings of the KDD-2004 Workshop on Multi-Relational Data Mining*, 2004, pp. 31–48.
[5] P. Singla and P. Domingos, "Entity resolution with markov logic," in *ICDM*, 2006, pp. 572–582.
[6] A. Arasu, C. Ré, and D. Suciu, "Large-scale deduplication with constraints using dedupalog," in *ICDE*, 2009, pp. 952–963.
[7] H. B. Newcombe and J. M. Kennedy, "Record linkage: making maximum use of the discriminating power of identifying information," *Commun. ACM*, vol. 5, no. 11, pp. 563–566, 1962.
[8] S. E. Whang and H. Garcia-Molina, "Joint entity resolution," Stanford University, Tech. Rep., available at http://ilpubs.stanford.edu:8090/1002/.
[9] M. R. Garey, D. S. Johnson, and R. Sethi, "The complexity of flowshop and jobshop scheduling," *Mathematics of Operations Research*, vol. 1, pp. 117–129, 1976.
[10] R. L. Graham and R. L. Grahamt, "Bounds on multiprocessing timing anomalies," *SIAM Journal on Applied Mathematics*, vol. 17, pp. 416–429, 1969.
[11] S. Whang and H. Garcia-Molina, "Entity resolution with evolving rules," *PVLDB*, vol. 3, no. 1, pp. 1326–1337, 2010.
[12] "Spock," http://spock.com.
[13] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom, "Swoosh: a generic approach to entity resolution," *VLDB J.*, vol. 18, no. 1, pp. 255–276, 2009.
[14] W. Winkler, "Overview of record linkage and current research directions," Statistical Research Division, U.S. Bureau of the Census, Washington, DC, Tech. Rep., 2006.
[15] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey." *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 1, pp. 1–16, 2007.
[16] H. Köpcke, A. Thor, and E. Rahm, "Evaluation of entity resolution approaches on real-world match problems," *PVLDB*, vol. 3, no. 1, pp. 484–493, 2010.
[17] S. E. Whang, D. Menestrina, G. Koutrika, M. Theobald, and H. Garcia-Molina, "Entity resolution with iterative blocking," in *SIGMOD Conference*, 2009, pp. 219–232.
[18] V. Rastogi, N. N. Dalvi, and M. N. Garofalakis, "Large-scale collective entity matching," *PVLDB*, vol. 4, no. 4, pp. 208–218, 2011.
[19] P. Brucker, *Scheduling algorithms (4. ed.)*. Springer, 2004.
[20] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.