Linking Business Listings to Identify Chains

Pei Li
University of Milan-Bicocca
pei.li@disco.unimib.it

Xin Luna Dong
AT&T Labs-Research
lunadong@research.att.com

Songtao Guo
AT&T Interactive Research
sguo@attinteractive.com

Andrea Maurino
University of Milan-Bicocca
maurino@disco.unimib.it

Divesh Srivastava
AT&T Labs-Research
divesh@research.att.com

ABSTRACT

Many local search engines, such as *Google Maps, Yahoo! Local, Yellowpages*, can benefit from the knowledge of *business chains* to improve the search experiences. However, business chains are rarely specified explicitly in business-listing data, but need to be inferred from the listings. While records that belong to the same chain typically share some similar values such as names and URL domains, they can also vary a lot, providing different addresses, different local phone numbers, etc. Traditional record-linage techniques, which often require high similarity between records for linking them, can fall short in this context.

This paper studies identifying business listings that belong to the same chain. Our solution contains two stages: the first stage identifies *cores* containing business listings that are very likely to belong to the same chain; the second stage collects strong evidence from the cores and leverages it for merging more listings in the same chain, while being tolerant to differences in other values. Our algorithm is designed to ensure efficiency and scalability. An experiment shows that it finished in 2.4 hours on a real-world data set containing 6.8 million listings, and obtained both a precision and a recall of above .95.

1. INTRODUCTION

With the advent of the Web and mobile devices, we are observing a boom in *local search*, searching local businesses under geographical constraints. Local search engines include *Google Maps, Yahoo! Local, YellowPages, yelp, ezlocal,* etc. These search engines can often benefit from knowledge of business chains, connected business entities that share a brand name and provide similar products and services (*e.g.*, *Walmart, McDonald's*). Such knowledge would allow users to search by business chain, allow search engines to render the returned results by chains, allow the associated review system to connect reviews on branches of the same chain, and allow data collectors to clean and enrich information on chains. However, business listings are rarely associated with specific chains explicitly, so we need to infer the chains from the listings.

Identifying business chains is different from any existing record-linkage problem, where the goal is to link records that refer to the same real-world entity. Chain identification, on the other hand, is essentially to link records that refer to entities in the same group.

Table 1: Business listings. There is a wrong value in italic font.

RID	name	phone	URL (domain)	location	category
r_1	Home Depot, The	808		NJ	furniture
r_2	Home Depot, The	808		NY	furniture
r_3	Home Depot, The	808	homedepot	MD	furniture
r_4	Home Depot, The	808	homedepot	AK	furniture
r_5	Home Depot, The	808	homedepot	MI	furniture
r_6	Home Depot, The	101	homedepot	IN	furniture
r_7	Home Depot, The	102	homedepot	NY	furniture
r_8	Home Depot, USA	103	homedepot	WV	furniture
r_9	Home Depot USA	808		SD	furniture
r_{10}	Home Depot - Tools	808		FL	furniture
r_{11}	Taco Casa		tacocasa	AL	restaurant
r_{12}	Taco Casa	900	tacocasa	AL	restaurant
r_{13}	Taco Casa	900	tacocasa, *tacocasatexas*	AL	restaurant
r_{14}	Taco Casa	900		AL	restaurant
r_{15}	Taco Casa	900		AL	restaurant
r_{16}	Taco Casa	701	tacocasatexas	TX	restaurant
r_{17}	Taco Casa	702	tacocasatexas	TX	restaurant
r_{18}	Taco Casa	703	tacocasatexas	TX	restaurant
r_{19}	Taco Casa	704		NY	food store
r_{20}	Taco Casa		tacodelmar	AK	restaurant

While such records typically share some similar values, they can also vary a lot since they represent different entities. For example, different branches in the same business chain can share the same name, some phone number and URL domain name, but meanwhile can also provide different local phone numbers, different addresses, etc. The real data also contain various representations for the same value and sometimes erroneous values, which can make the problem even more challenging. Traditional record-linkage techniques (surveyed in [6, 16]) often require high similarity between records for linking them, so can fall short in identifying business chains. We next illustrate this with an example.

EXAMPLE 1.1. *Table 1 shows a set of business listings extracted from a real-world data set (described in Section 5). They describe twenty business entities at different locations by their name, phone number, URL domain name, location (we show only state), and category (some values simplified). Among them, $r_1 - r_{18}$ belong to three business chains: $Ch_1 = \{r_1 - r_{10}\}$, $Ch_2 = \{r_{11} - r_{15}\}$, and $Ch_3 = \{r_{16} - r_{18}\}$; r_{19} and r_{20} do not belong to any chain. Note the slightly different names for businesses in chain Ch_1; also note that r_{13} contains two URLs, one (*tacocasatexas*) being wrong.*

If we require only high similarity on name *for chain identification, we may wrongly decide that $r_{11} - r_{20}$ all belong to the same chain as they share a popular restaurant name* Taco Casa*. If we iteratively merge records with high similarity on* name *and shared* phone *or* URL*, we can wrongly merge Ch_2 and Ch_3 because of the wrong URL from r_{13}. If we require high similarity between listings on* name, phone, URL, category*, we may split $r_6 - r_8$ out of chain Ch_1 because of their different phone numbers.* □

Another challenge in chain identification is scalability. There are often millions of business listings, and a chain can contain tens of thousands of listings. Even a simple pairwise record comparison within a chain can already be very expensive.

The key idea in our solution is to find strong evidence that can glue chain members together, while be tolerant to differences in values specific for individual chain member. For example, we wish to reward sharing of primary phone numbers or URL domain names, but would not penalize differences from locations, local phone numbers, and even categories. For this purpose, our algorithm proceeds in two stages. First, we identify *cores* containing business listings that are very likely to belong to the same chain. Second, we collect strong evidence from the resulting cores, such as primary phone numbers and URL domain names, based on which we cluster the cores and remaining records into chains. The use of cores and strong evidence distinguishes our clustering algorithm from traditional clustering techniques for record linkage. In this process, it is crucial that core generation makes few false positives even in presence of erroneous values, such that we can avoid ripple effect on clustering later. Our algorithm is designed to ensure efficiency and scalability.

To the best of our knowledge, this paper is the first one studying the chain-identification problem. We make three contributions.

1. We study core generation in presence of erroneous data. Our core is robust in the sense that even if we remove a few possibly erroneous listings, we still have strong evidence that the rest of the listings must belong to the same chain. We propose efficient algorithm for core generation.
2. We then reduce the chain-identification problem into clustering cores and remaining listings. Our clustering algorithm leverages strong evidence collected from cores and meanwhile is tolerant to value variety of listings in the same chain.
3. We conducted experiments on real-world data containing 6.8 million of business listings; our algorithm finished in 2.4 hours and obtained a precision and recall of over .95, improving over traditional record linkage techniques, which obtained a precision of .8 and a recall of .5.

Note that while we focus on business-chain identification, the key idea of our techniques can also be applied in other applications where we need to identify records in the same group, such as finding people from the same affiliation (according to address and phone number) when institute information is absent or incomplete, finding players that belong to the same team, finding papers in the same conference, and so on.

In the rest of the paper, Section 2 defines the problem and provides an overview of our solution. Sections 3-4 describe the two stages in our solution. Section 5 describes experimental results. Section 6 discusses related work and Section 7 concludes.

2. OVERVIEW

This section formally defines the business-chain identification problem and provides an overview of our solution.

2.1 Problem definition

Let $\mathbf{R}$ be a set of records that describe real-world businesses by a set of attributes $\mathbf{A}$, including but not limited to business names and locations. We call each $r \in \mathbf{R}$ a *business listing* and denote by $r.A$ its value on attribute $A \in \mathbf{A}$. A *business entity* is a real-world business at a particular location. Because of duplicates and heterogeneity of the data, one business entity may be represented by several listings in $\mathbf{R}$, with different representations of the same attribute value and sometimes even erroneous values. A *business chain* is a set of business entities with the same or highly similar names at different locations, either under shared corporate ownership (i.e., sharing a brand and central management) or under franchising agreements (i.e., operating with the right or license granted by a company for marketing its products in a specific territory). Examples of the former type of chains include *Walmart* and *Home Depot*, and examples of the latter include *Subway* and *McDonald's*.[1] However, real data seldom have chains explicitly specified.

This paper aims to solve the problem of *business-chain identification*; that is, finding business listings that represent business entities belonging to the same chain. We assume we have already applied entity-resolution techniques (*e.g.*, [12]) to merge different listings that represent the same business entity.

DEFINITION 2.1 (BUSINESS-CHAIN IDENTIFICATION). *Given a set $\mathbf{R}$ of business listings,* business-chain identification *identifies a set of clusters $\mathbf{CH}$ of listings in $\mathbf{R}$, such that (1) listings that represent business entities in the same chain belong to one and only one cluster in $\mathbf{CH}$, and (2) listings that represent business entities not in any chain do not belong to any cluster in $\mathbf{CH}$.* □

EXAMPLE 2.2. *Consider records in Example 1.1, where each record describes a business entity (at a distinct location) by attributes* name, phone, URL, location, *and* category.

The ideal solution to the chain-identification problem contains 3 clusters: $Ch_1 = \{r_1 - r_{10}\}$, $Ch_2 = \{r_{11} - r_{15}\}$, and $Ch_3 = \{r_{16} - r_{18}\}$. Among them, Ch_2 and Ch_3 represent two different chains with the same name. Listings r_{19} and r_{20} do not belong to any chain, so not to any cluster in the solution either. □

2.2 Overview of our solution

Business-chain identification is different from traditional entity-resolution problem because it essentially looks for records that represent entities in the same group (*i.e.*, business chain), rather than records that represent exactly the same entity (*e.g.*, business entity). Different members in the same group often share a certain amount of commonality (*e.g.*, common name, primary phone and URL domain), but meanwhile can also have a lot of differences (*e.g.*, different addresses, local phone numbers); thus, we need to allow much higher variety in some attribute values to avoid false negatives. On the other hand, as we have shown in Section 1, simply lowering our requirement on similarity in clustering can lead to a lot of false positives, which we also wish to avoid.

The key idea of our solution is to distinguish between *strong* evidence and *weak* evidence. For example, different branches in the same business chain often share the same URL domain name and those in North America often share the same 1-800 phone number. Thus, a URL domain or phone number shared among many business listings with highly similar names can serve as strong evidence for chain identification. In contrast, a phone number alone shared by only a couple of business entities is much weaker evidence, since one might be an erroneous or out-of-date value.

To facilitate finding strong evidence, our solution contains two stages. The first stage groups records that are highly likely to belong to the same chain; for example, a set of listings with the same name and phone number are very likely to be in the same chain. We call the results *cores* of the chains; from them we can collect strong evidence such as name of the chain, primary phone number, and primary URL domain. The key of this stage is to be robust against erroneous values and make as few false positives as possible, so we can avoid identifying strong evidence wrongly and causing ripple effect later; however, being too strict and miss the cores of many chains can prevent from collecting important strong evidence.

[1] http://en.wikipedia.org/wiki/Chain_store.

The second stage leverages the discovered strong evidence and clusters cores and remaining records into chains. It decides whether several cores belong to the same chain, and whether a record that does not belong to any core actually belongs to some chain. It also employs weak evidence, but treats it differently from strong evidence. The key of this stage is to leverage the strong evidence and meanwhile be tolerant to diversity of values in the same chain, so we can remove false negatives made in the first stage.

We next illustrate our approach on the motivating example.

EXAMPLE 2.3. *Continue with the motivating example. In the first stage we generate three cores:* $Cr_1 = \{r_1 - r_7\}, Cr_2 = \{r_{14}, r_{15}\}, Cr_3 = \{r_{16} - r_{18}\}$. *Records* $r_1 - r_7$ *are in the same core because they have the same name, five of them* ($r_1 - r_5$) *share the same phone number* 808 *and five of them* ($r_3 - r_7$) *share the same URL* homedepot. *Similar for the other two cores. Note that* r_{13} *does not belong to any core, because one of its URLs is the same as that of* $r_{11} - r_{12}$, *and one is the same as that of* $r_{16} - r_{18}$, *but except name, there is no other common information between these two groups of records. To avoid mistakes, we defer the decision on* r_{13}. *Indeed, recall that* tacocasatexas *is a wrong value for* r_{13}. *For a similar reason, we defer the decision on* r_{12}.

In the second stage, we generate the business chains. We merge $r_8 - r_{10}$ *with core* Cr_1, *because they have similar names and share either the primary phone number or the primary URL. We also merge* $r_{11} - r_{13}$ *with core* Cr_2, *because (1)* $r_{12} - r_{13}$ *share the primary phone* 900 *with* Cr_2, *and (2)* r_{11} *shares the primary URL* tacocasa *with* $r_{12} - r_{15}$. *We do not merge* Cr_2 *and* Cr_3 *though, because they share neither the primary phone nor the primary URL. We do not merge* r_{19} *or* r_{20} *to any chain, because there is again not much strong evidence. We thus obtain the ideal result.* □

To facilitate this two-stage solution, we classify the attributes of business listings into four categories. Attributes in a data set can be categorized by domain experts.

- *Common-value attribute:* We call an attribute A a common-value attribute if all business entities in the same chain have the same or highly similar A-values. In our example, name falls in this category.
- *Dominant-value attribute:* We call an attribute A a dominant-value attribute if business entities in the same chain often share one or a few primary A-values (but there can also exist other less-common values), and these values are seldom used by business entities outside the chain. In our example, phone and URL fall in this category.
- *Distinct-value attribute:* We call an attribute A a distinct-value attribute if each business entity in the same chain has a distinct A-value and these values are seldom used by business entities outside the chain. In our example, location falls in this category.
- *Multi-value attribute:* We call the rest of the attributes mutli-value attributes as there is often a many-to-many relationship between chains and values of such attributes. In our example, category falls in this category.

In the rest of the paper, we describe core identification in Section 3 and chain identification in Section 4.

3. CORE IDENTIFICATION

The first stage of our solution creates cores consisting of records that are very likely to belong to the same business chain. The key in core identification is to be robust to possible erroneous values. This section starts with presenting the criteria we wish the cores to meet (Section 3.1), then describes how we efficiently construct similarity graphs to facilitate core finding (Section 3.2), and finally gives the algorithm for core identification (Section 3.3).

3.1 Criteria for a core

At the first stage we wish to make only decisions that are highly likely to be correct; thus, we require that each core contains only highly similar records, and different cores are fairly different and easily distinguishable from each other. In addition, we wish that our results are robust even in presence of a few erroneous values in the data. In the motivating example, $r_1 - r_7$ form a good core, because *808* and *homedepot* are very popular values among these records. In contrast, $r_{13} - r_{18}$ does not form a good core, because records $r_{14} - r_{15}$ and $r_{16} - r_{18}$ do not share any phone number or URL domain; the only "connector" between them is r_{13}, so they can be wrongly merged if r_{13} contains erroneous values. Also, considering $r_{13} - r_{15}$ and $r_{16} - r_{18}$ as two different cores is risky, because (1) it is not very clear whether r_{13} is in the same chain as $r_{14} - r_{15}$ or as $r_{16} - r_{18}$, and (2) these two cores share one URL domain name so are not fully distinguishable.

We capture this intuition with *connectivity of a similarity graph*. We define the *similarity graph* of a set $\mathbf{R}$ of business listings as an undirected graph, where each node represents a listing in $\mathbf{R}$, and an edge indicates high similarity between the connected two listings (we describe in which cases we consider two listings as highly similar later). Figure 1 shows the similarity graph for the motivating example.

Each core would correspond to a connected sub-graph of the similarity graph. We wish such a sub-graph to be *robust* such that even if we remove a few nodes the sub-graph is still connected; in other words, even if there are some erroneous records, without them we still have enough evidence showing that the rest of the records must belong to the same chain. The formal definition goes as follows.

DEFINITION 3.1 (k-ROBUSTNESS). *A graph* G *is* k-robust *if after removing arbitrary* k *nodes and edges to these nodes,* G *is still connected. A clique or a single node is* k*-robust for any* k. □

In Figure 1, the subgraph with nodes $r_1 - r_7$ is 2-robust, but that with $r_{13} - r_{18}$ is not 1-robust as removing r_{13} can disconnect it.

According to the definition, we can partition the similarity graph into a set of k-robust subgraphs. As we do not wish to split any core unnecessarily, we require the *maximal* k*-robust partitioning*; that is, merging any two subgraphs should not obtain a graph that is also k-robust.

DEFINITION 3.2 (MAXIMAL k-ROBUST PARTITIONING). *Let* G *be a similarity graph. A partitioning of* G *is a* maximal k-robust partitioning *if it satisfies the following properties.*

1. *Each node belongs to one and only one subgraph.*
2. *Each subgraph is* k*-robust.*
3. *The result of merging any two subgraphs is not* k*-robust.* □

Note that a data set can have more than one maximal k-robust partitioning. Consider $r_{11} - r_{18}$ in Figure 1. There are three maximal 1-robust partitionings: $\{\{r_{11}\}, \{r_{12}, r_{14} - r_{15}\}, \{r_{13}, r_{16} - r_{18}\}\}$; $\{\{r_{11} - r_{12}\}, \{r_{14} - r_{15}\}, \{r_{13}, r_{16} - r_{18}\}\}$; and $\{\{r_{11} - r_{15}\}, \{r_{16} - r_{18}\}\}$. If we treat each partitioning as a possible world, records that belong to the same partition in all possible worlds have high probability to belong to the same chain and so form a core. Accordingly, we define a core as follows.

DEFINITION 3.3 (CORE). *Let* $\mathbf{R}$ *be a set of business listings and* G *be the similarity graph of* $\mathbf{R}$. *The records that belong to the same subgraph in every maximal* k*-robust partitioning of* G *form a* core *of* $\mathbf{R}$. *A core contains at least 2 records.* □

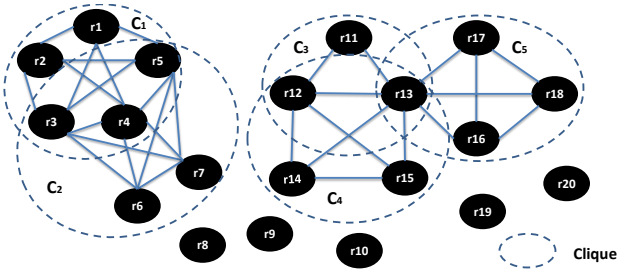

Figure 1: Similarity graph for records in Table 1.

Table 2: (a) Inverted index for the similarity graph in Figure 1. (b) Simplified inverted index for (a).

Record	V-Cliques
r_1	C_1
r_2	C_1
r_3	C_1, C_2
r_4	C_1, C_2
r_5	C_1, C_2
r_6	C_2
r_7	C_2
r_{11}	C_3
r_{12}	C_3, C_4
r_{13}	C_3, C_4, C_5
r_{14}	C_4
r_{15}	C_4
r_{16}	C_5
r_{17}	C_5
r_{18}	C_5

(a)

Record	V-Cliques	Represent
$r_{1/2}$	C_1	$r_1 - r_2$
r_3	C_1, C_2	r_3
r_4	C_1, C_2	r_4
r_5	C_1, C_2	r_5
$r_{6/7}$	C_2	$r_6 - r_7$
r_{11}	C_3	r_{11}
r_{12}	C_3, C_4	r_{12}
r_{13}	C_3, C_4, C_5	r_{13}
$r_{14/15}$	C_4	$r_{14} - r_{15}$
$r_{16/17/18}$	C_5	$r_{16} - r_{18}$

(b)

EXAMPLE 3.4. *Consider Figure 1 and assume $k = 1$. There are two connected sub-graphs. For records $r_1 - r_7$, the subgraph is 1-robust, so they form a core. For records $r_{11} - r_{18}$, there are three maximal 1-robust partitionings for their subgraph, as we have shown. Two subsets of records belong to the same subgraph in each partitioning: $\{r_{14} - r_{15}\}$ and $\{r_{16} - r_{18}\}$; they form 2 cores.* □

3.2 Constructing similarity graphs

Generating the cores requires analysis on the similarity graph. Real-world data can often contain millions of records; it is unscalable to compare every pair of records and create edges accordingly. We next describe how we construct and represent the similarity graph in a scalable way.

We add an edge between two records if they have the same value for each common-value attribute and share at least one value on a dominant-value attribute.[2] All records that share values on the common-value attributes and share the same value on a dominant-value attribute form a clique, which we call a *v-clique*. We can thus represent the graph with a set of v-cliques, denoted by $\mathbf{C}$; for example, the graph in Figure 1 can be represented by 5 v-cliques. In addition, we maintain an *inverted index*, where each entry corresponds to a record r and contains the v-cliques that r belongs to (see Table 2(a) as an example). We denote the index by $\bar{L}$ and the v-cliques that record r belongs to by $\bar{L}(r)$. Whereas the size of the similarity graph is quadratic in the number of the nodes, the size of the inverted index is only linear in that number. The inverted index also makes it easy to find *adjacent v-cliques*, v-cliques that share nodes, as they appear in the same entry.

Graph construction is then reduced to v-clique finding, which can be done by scanning values of dominant-value attributes. In this process, we wish to prune a v-clique if it is a sub-clique of another one. Pruning by checking every pair of v-cliques can be very expensive since the number of v-cliques is also huge. Instead, we do it together with v-clique finding. Specifically, our algorithm GRAPHCONSTRUCTION takes $\mathbf{R}$ as input and outputs $\mathbf{C}$ and $\bar{L}$. We start with $\mathbf{C} = \bar{L} = \emptyset$. For each value v of a dominant-value attribute, we denote the set of records with v by $\bar{R}_v$ and do the following.

1. Initialize the v-cliques for v as $\mathbf{C_v} = \emptyset$. Add a single-record cluster for each record $r \in \bar{R}_v$ to a working set $\bar{T}$. Mark each cluster as "unchanged".
2. For each $r \in \bar{R}_v$, scan $\bar{L}$ and consider each v-clique $C \in \bar{L}(r)$ that has not been considered yet. For all records in $C \cap \bar{R}_v$, merge their clusters. Mark the merged cluster as "changed" if the result is not a proper sub-clique of C. If $C \subseteq \bar{R}_v$, remove C from $\mathbf{C}$. This step removes the v-cliques that must be sub-cliques of those we will form next.
3. For each cluster $C \in \bar{T}$, if there exists $C' \in \mathbf{C_v}$ such that C and C' share the same value for each common-value attribute, remove C and C' from $\bar{T}$ and $\mathbf{C_v}$ respectively, add $C \cup C'$ to $\bar{T}$ and mark it as "changed"; otherwise, move C to $\mathbf{C_v}$. This step merges clusters that share values on common-value attributes. At the end, $\mathbf{C_v}$ contains the v-cliques with value v.
4. Add each v-clique with mark "changed" in $\mathbf{C_v}$ to $\mathbf{C}$ and update $\bar{L}$ accordingly. The marking prunes size-1 v-cliques and the sub-cliques of those already in $\mathbf{C}$.

PROPOSITION 3.5. *Let $\mathbf{R}$ be a set of business listings. Denote by $n(r)$ the number of values on dominant-value attributes from $r \in \mathbf{R}$. Let $n = \sum_{r \in \mathbf{R}} n(r)$ and $m = \max_{r \in \mathbf{R}} n(r)$. Let s be the maximum v-clique size. Algorithm* GRAPHCONSTRUCTION *(1) runs in time $O(ns(m + s))$, (2) requires space $O(n)$, and (3) its result is independent of the order in which we consider the records.* □

PROOF. We first prove that GRAPHCONSTRUCTION runs in time $O(ns(m + s))$. Step 2 of the algorithm takes in time $O(nsm)$, where it takes in time $O(ns)$ to scan all records for a dominant-value attribute, and a record can be scanned maximally m times. Step 3 takes in time $O(ns^2)$. Thus, the algorithm runs in time $O(ns(m + s))$.

We next prove that GRAPHCONSTRUCTION requires space $O(n)$. For each value v of a dominate-value attribute, the algorithm keeps three data sets: $\bar{L}$ that takes in space $O(n)$, $\mathbf{C_v}$ and $\bar{T}$ that require space in total no greater than $O(|\mathbf{R}|)$. Since $O(n) \geq O(|\mathbf{R}|)$, the algorithm requires space $O(n)$.

We now prove that the result of GRAPHCONSTRUCTION is order independent. Given $\bar{L}$ and $\bar{R}_v$, Step 2 scan $\bar{L}$ and apply transitive rule to merge clusters of records in $C \cap \bar{R}_v$, for each v-clique $C \in \bar{L}$. The process is independent from the order in which we consider the records in $\bar{R}_v$. The order independence of the result in Step 3 is proven in [2]. Therefore, the final result is independent from the order in which we consider the records. □

EXAMPLE 3.6. *Consider graph construction for records in Table 1. Figure1 shows the similarity graph and Table 2(a) shows the inverted list. We focus on records $r_1 - r_8$ for illustration.*

First, $r_1 - r_5$ share the same name and phone number 808*, so we add v-clique $C_1 = \{r_1 - r_5\}$ to $\mathbf{C}$. Now consider URL* homedepot *where $\bar{R}_v = \{r_3 - r_8\}$. Step 1 generates 6 clusters, each marked "unchanged", and $\bar{T} = \{\{r_3\}, \ldots, \{r_8\}\}$. Step 2 looks up $\bar{L}$ for each record in $\bar{R}_v$. Among them, $r_3 - r_5$ belong to v-clique C_1, so it merges their clusters and marks the result $\{r_3 - r_5\}$ "unchanged" ($\{r_3 - r_5\} \subset C_1$); then, $\bar{T} = \{\{r_3 - r_5\}, \{r_6\}, \{r_7\}, \{r_8\}\}$. Step 3 compares these clusters and merges the first three as they share the same name, marking the result as "changed". At the end,*

[2] In practice, for common-value attributes we require only highly similar values. We can also adapt our method for other edge-adding strategies; we compare different strategies in experiments (Section 5).

$\mathbf{C}_v = \{\{r_3 - r_7\}, \{r_8\}\}$. *Finally, Step 4 adds* $\{r_3 - r_7\}$ *to* $\mathbf{C}$ *and discards* $\{r_8\}$ *since it is marked "unchanged".* □

Given the sheer number of records in $\mathbf{R}$, the inverted index can still be huge. The following proposition shows that records in the same v-clique but not any other v-clique must belong to the same core. As we show later, such records cannot affect robustness judgment, so we do not need to distinguish them.

PROPOSITION 3.7. *Let G be a similarity graph. Let r and r' be two nodes in G that belong to the same v-clique C and not any other v-clique. Then, r and r' must belong to the same partition in any maximal k-robust partitioning.* □

PROOF. We prove that there does not exist such a maximal k-robust partitioning where r and r' are in different partitions. Suppose r and r' belong to subgraphs P and P' respectively in a maximal k-robust partitioning, where r, r' are connected to n, n' nodes in P, P' respectively. P and P' are connected by at least m nodes , $m = \min\{n+1, n'+1\}$. We next prove that $P \cup P'$ is k-robust.

If P, P' are both v-cliques, $P \cup P' \subseteq C$ is also a v-clique and k-robust.

We next consider the case where one of the partitions is not a v-clique. Suppose P is not a v-clique, each node in P is connected to more than k nodes in P, where $k < n$. If P' is a v-clique, each node in $P' \subseteq C$ is connected to at least $n+1$ nodes in P, thus $P \cup P'$ is k-robust. If P' is not a v-clique, we have $k < \min\{n, n'\} < m$, thus $P \cup P'$ is also k-robust.

Since the fact that $P \cup P'$ is k-robust violates Condition 3 in Definition 3.2, it proves that r, r' must belong to the same partition in any maximal k-robust partitioning. □

Thus, we simplify the inverted index such that for each v-clique we keep only a *representative* for nodes belonging only to this v-clique. Table 2(b) shows the simplified index for Table 2(a).

Case study: On a data set with 6.8M records (described in Section 5), our graph-construction algorithm finished in 2.2 hours. The original similarity graph contains 6.8M nodes and 569M edges. The inverted index contains 0.8M entries, each associated with at most 6 v-cliques; in total there are 94K v-cliques. The simplified inverted index further reduces the size of the index to 0.15M entries, where an entry can represent up to 11K records. Therefore, the simplified inverted index reduces the size of the similarity graph by 3 orders of magnitude.

3.3 Identifying cores

We solve the core-identification problem by reducing it to a Max-flow/Min-cut Problem. However, computing the max flow for a given graph G and a source-destination pair takes time $O(|G|^{2.5})$, where $|G|$ denotes the number of nodes in G; even the simplified inverted index can still contain hundreds of thousands of entries, so it can be very expensive. We thus first merge certain v-cliques according to a sufficient (but not necessary) condition for k-robustness and consider them as a whole in core identification; we then split the graph into subgraphs according to a necessary (but not sufficient) condition for k-robustness. We apply reduction only on the resulting subgraphs, which are substantially smaller as we show at the end of this section. Section 3.3.1 describes screening before reduction, Section 3.3.2 describes the reduction, and Section 3.3.3 gives the full algorithm, which iteratively applies screening and the reduction. Note that the notations in this section are with respect to v-cliques, thus can be slightly different from those in Graph Theory.

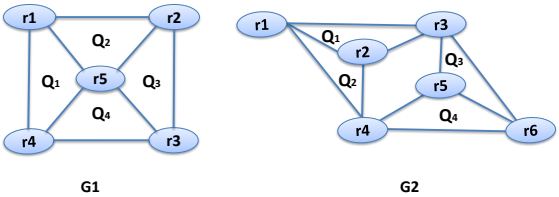

Figure 2: Two example graphs.

3.3.1 Screening

A graph can be considered as a union of v-cliques, so essentially we need to decide if a union of v-cliques is k-robust. First, we can prove the following sufficient condition for k-robustness.

THEOREM 3.8 ($(K+1)$-CONNECTED CONDITION). *Let G be a graph consisting of a union Q of v-cliques. If for every pair of v-cliques $C, C' \in Q$, there is a path of v-cliques between C and C' and every pair of adjacent v-cliques on the path share at least $k+1$ nodes, graph G is k-robust.* □

PROOF. Given Menger's Theorem [3], graph G is k-robust if for any pair of nodes r, r' in G, there exists at least $k+1$ independent paths that do not share any nodes other than r, r' in G. We now prove that for any pair of nodes r, r' in graph G that satisfies $(k+1)$-connected condition, there exists at least $k+1$ independent paths between r, r'. We consider two cases, 1) r, r' are adjacent such that there exists a v-clique in G that contains r, r'; 2) r, r' are not adjacent such that there exists no v-clique in G that contains r, r'.

We first consider Case 1 where there exists a v-clique C containing r, r'. Since each v-clique in G has more than $k+1$ nodes, there exist at least k 2-length paths and one 1-length path between $r, r' \in C$. It proves that there exists at least $k+1$ independent paths between r and r'.

We next consider Case 2 where there exists no v-clique containing r, r' in G. Suppose $r \in C, r' \in C'$, where C, C' are different v-cliques in G. Since there exists a path of v-cliques between C and C' where every pair of adjacent v-cliques in the path share at least $k+1$ nodes, there exists at least $k+1$ independent paths between r and r'.

Given the above two cases, we have that there exist at least $k+1$ independent paths between every pair of nodes in G, therefore G is k-robust. □

We call a single v-clique or a union of v-cliques that satisfy the $(k+1)$-connected condition a $(k+1)$-*connected v-union*. A $(k+1)$-connected v-union must be k-robust but not vice versa. In Figure 1, subgraph $\{r_1 - r_7\}$ is a 3-connected v-union, because the only two v-cliques, C_1 and C_2, share 3 nodes. Indeed, it is 2-robust. On the other hand, graph G_1 in Figure 2 is 2-robust but not 3-connected (there are 4 v-cliques, where each pair of adjacent v-cliques share only 1 or 2 nodes). Accordingly, we can consider a v-union as a whole in core identification.

Next, we present a necessary condition for k-robustness.

THEOREM 3.9 ($(K+1)$-OVERLAP CONDITION). *Graph G is k-robust only if for every $(k+1)$-connected v-union $Q \in G$, Q shares at least $k+1$ common nodes with the subgraph consisting of the rest of the v-unions.* □

PROOF. We prove that if graph G contains a $(k+1)$-connected v-union Q that shares at most k common nodes with the rest of the graph, G is not k-robust. Since Q shares at most k common nodes with the subgraph consisting of the rest of the v-unions, removing the common nodes will disconnect Q from G, it proves that G is not k-robust. Thus, $(k+1)$-overlap condition holds. □

We call a graph G that satisfies the $(k+1)$-overlap condition a $(k+1)$-*overlap graph*. A k-robust graph must be a $(k+1)$-overlap graph but not vice versa. In Figure 1, subgraph $\{r_{11} - r_{18}\}$ is not a 2-overlap graph, because there are two 2-connected v-unions, $\{r_{11} - r_{15}\}$ and $\{r_{13}, r_{16} - r_{18}\}$, but they share only one node; indeed, the subgraph is not 1-robust. On the other hand, graph G_2 in Figure 2 satisfies the 3-overlap condition, as it contains four 3-connected v-unions (actually four v-cliques), $Q_1 - Q_4$, and each v-union shares 3 nodes in total with the others; however, it is not 2-robust (removing r_3 and r_4 disconnects it). Accordingly, for $(k+1)$-overlap graphs we still need to check k-robustness by reduction to a Max-flow Problem.

Now the problem is to find $(k+1)$-overlap subgraphs. Let G be a graph where a $(k+1)$-connected v-union overlaps with the rest of the v-unions on no more than k nodes. We split G by removing these overlapping nodes. For subgraph $\{r_{11} - r_{18}\}$ in Figure 1, we remove r_{13} and result with two subgraphs $\{r_{11} - r_{12}, r_{14} - r_{15}\}$ and $\{r_6 - r_8\}$ (recall from Example 3.4 that r_{13} cannot belong to any core). Note that the result subgraphs may not be $(k+1)$-overlap graphs (*e.g.*, $\{r_1 - r_2, r_4 - r_5\}$ contains two v-unions that share only one node), so we need to further screen them.

We now describe our screening algorithm, SCREEN, (details in Algorithm 1), which takes a graph G, represented by $\mathbf{C}$ and $\bar{L}$, as input, finds $(k+1)$-connected v-unions in G and meanwhile decides if G is a $(k+1)$-overlap graph. If not, it splits G into subgraphs for further examination.

1. If G contains a single node, output it as a core if the node represents multiple records that belong only to one v-clique.
2. For each v-clique $C \in \mathbf{C}$, initialize a v-union. We denote the set of v-unions by $\bar{Q}$, the v-union that C belongs to by $Q(C)$, and the overlap of v-cliques C and C' by $\bar{B}(C, C')$.
3. For each v-clique $C \in \mathbf{C}$, we merge v-unions as follows.

   (a) For each record $r \in C$ that has not been considered, for every pair of v-cliques C_1 and C_2 in r's index entry, if they belong to different v-unions, add r to overlap $\bar{B}(C_1, C_2)$.

   (b) For each v-union Q where there exist $C_1 \in Q$ and $C_2 \in Q(C)$ such that $|\bar{B}(C_1, C_2)| \geq k+1$, merge Q and $Q(C)$.

   At the end, $\bar{Q}$ contains all $(k+1)$-connected v-unions.
4. For each v-union $Q \in \bar{Q}$, find its border nodes as $\bar{B}(Q) = \cup_{C \in Q, C' \notin Q} \bar{B}(C, C')$. If $|\bar{B}(Q)| \leq k$, split the subgraph it belongs to, denoted by $G(Q)$, into two subgraphs $Q \setminus \bar{B}(Q)$ and $G(Q) \setminus Q$.
5. Return the remaining subgraphs.

PROPOSITION 3.10. *Denote by $|\bar{L}|$ the number of entries in input $\bar{L}$. Let m be the maximum number of values from dominant-value attributes of a record, and a be the maximum number of adjacent v-unions that a v-union has. Algorithm* SCREEN *takes time $O((m^2 + a) \cdot |\bar{L}|)$ and the result is independent of the order in which we examine the v-cliques.* □

PROOF. We first prove the time complexity of SCREEN. It takes in time $O(m^2|\bar{L}|)$ to scan all entries in $\bar{L}$ and find common nodes between each pair of adjacent v-cliques (Step 3(a)). It takes in time $O(a|\mathbf{C}|)$ to merge v-unions, where $|\mathbf{C}|$ is the number of v-cliques in G (Step 3(b)). Since $|\mathbf{C}| < |\bar{L}|$, the algorithm runs in time $O(m^2 + a) \cdot |\bar{L}|$.

We next prove that the result of $Screen$ is independent of the order in which we examine the v-cliques, that is, 1) finding all maximal $(k+1)$-connected v-unions in G is order independent; 2) removing all nodes in $\bar{B}(Q)$ from G where $|barB(Q)| \leq k$ is order independent.

Algorithm 1 SCREEN$(G, \bar{C}, \bar{L}, k)$

Input: G: Simplified similarity graph.
$\bar{C}$: Set of cores.
$\bar{L}$: Inverted list of the similarity graph.
k: Robustness requirement.
Output: $\bar{G}$ Set of subgraphs in G.

```
1: if G contains a single node r then
2:    if r represent multiple records then
3:       add r to C̄.
4:    end if
5:    return Ḡ = φ.
6: else
7:    initialize v-union Q(C) for each v-clique C and add Q(C)
      to Q̄.
8:    // find v-union
9:    for each v-clique C ∈ G do
10:      for each record r ∈ C that is not proceeded do
11:         for each v-clique pair C1, C2 ∈ L̄(r) do
12:            if C1, C2 are in different v-unions then
13:               add r to overlap B̄(Q(C1), Q(C2)).
14:            end if
15:         end for
16:      end for
17:      for each v-union Q where B̄(Q, Q(C)) ≥ k do
18:         merge Q and Q(C) as Q_m.
19:         for each v-union Q' ≠ Q, Q' ≠ Q(C) do
20:            set B̄(Q', Q_m) = B̄(Q', Q) ∪ B̄(Q', Q(C))
21:         end for
22:      end for
23:   end for
24:   // screening
25:   for each v-union Q ∈ Q̄ do
26:      compute B̄(Q) = ∪_{Q'∈Q̄} B̄(Q, Q').
27:      if |B̄(Q)| < k then
28:         add subgraphs Q \ B̄(Q) and G(Q) \ Q into Ḡ
29:      end if
30:   end for
31: end if
32: return Ḡ;
```

Consider order independency of finding all v-unions in G. To find all v-unions in G is conceptually equivalent to find all connected components in an abstract graph G_A, where each node in G_A is a v-clique in G and two nodes in G_A are connected if the two corresponding v-cliques share more than k nodes. SCREEN checks whether each node in G is a common node between two v-cliques (Step 3(a)), and if two cliques share more than k nodes, merges their v-unions (Step 3(b)), which is equivalent to connect two nodes in G_A. Once all nodes in G is scanned, all edges in G_A are added, and the order in which we examine nodes in G is independent from the structure of G_A and the connected components in G_A. Therefore, finding all v-unions in G is order independent.

Consider order independency of removing nodes in G. Suppose $Q_1, Q_2, ..., Q_m, m > 0$ are all v-unions in G with $|\bar{B}(Q_i)| \leq k, i \in [1, m]$. Since G is finite, Q_i is finite and unique; thus, removing all nodes in $\bar{B}(Q))$ from G where $|\bar{B}(Q)| \leq k$ is order independent. □

Note that m and a are typically very small, so SCREEN is basically linear in the size of the inverted index. Finally, we have results similar to Proposition 3.7 for v-unions, so we can further simplify the graph by keeping for each v-union a single representative for

all nodes that only belong to it. Each result k-overlap subgraph is typically very small.

EXAMPLE 3.11. *Consider Table 2(b) as input and $k = 1$. Step 2 creates five v-unions $Q_1 - Q_5$ for the five v-cliques in the input.*

Step 3 starts with v-clique C_1. It has 4 nodes (in the simplified inverted index), among which 3 are shared with C_2. Thus, $\bar{B}(C_1, C_2) = \{r_3 - r_5\}$ and $|\bar{B}(C_1, C_2)| \geq 2$, so we merge Q_1 and Q_2 into $Q_{1/2}$. Examining C_2 reveals no other shared node.

Step 3 then considers v-clique C_3. It has three nodes, among which $r_{12} - r_{13}$ are shared with C_4 and r_{13} is shared with C_5. Thus, $\bar{B}(C_3, C_4) = \{r_{12} - r_{13}\}$ and $\bar{B}(C_3, C_5) = \{r_{13}\}$. We merge Q_3 and Q_4 into $Q_{3/4}$. Examining C_4 and C_5 reveals no other shared node. We thus result with three 2-connected v-unions: $\bar{Q} = \{Q_{1/2}, Q_{3/4}, Q_5\}$.

Step 4 then considers each v-union. For $Q_{1/2}$, $\bar{B}(Q_{1/2}) = \emptyset$ and we thus split subgraph $Q_{1/2}$ out and merge all of its nodes to one $r_{1/\ldots/7}$. For $Q_{3/4}$, $\bar{B}(Q_{3/4}) = \{r_{13}\}$ so $|\bar{B}(Q_{3/4})| < 2$. We split $Q_{3/4}$ out and obtain $\{r_{11} - r_{12}, r_{14/15}\}$ (r_{13} is excluded). Similar for Q_5 and we obtain $\{r_{16/17/18}\}$. Therefore, we return three subgraphs. □

3.3.2 Reduction

Intuitively, a graph $G(V, E)$ is k-robust if and only if between any two nodes $a, b \in V$, there are more than k paths that do not share any node except a and b. We denote the number of non-overlapping paths between nodes a and b by $\kappa(a, b)$. We can reduce the problem of computing $\kappa(a, b)$ into a Max-flow Problem.

For each input $G(V, E)$ and nodes a, b, we construct the (directed) *flow network* $G'(V', E')$ as follows.

1. Node a is the source and b is the sink (there is no particular order between a and b).
2. For each $v \in V, v \neq a, v \neq b$, add two nodes v', v'' to V', and two directed edges $(v', v''), (v'', v')$ to E'. If v' represents n nodes, the edge (v', v'') has weight n, and the edge (v'', v') has weight ∞.
3. For each edge $(a, v) \in E$, add edge (a, v') to E'; for each edge $(u, b) \in E$, add edge (u'', b) to E'; for each other edge $(u, v) \in E$, add two edges (u'', v') and (v'', u') to E'. Each edge has capacity ∞.

LEMMA 3.12. *The max flow from source a to sink b in $G'(V', E')$ is equivalent to $\kappa(a, b)$ in $G(V, E)$.* □

PROOF. According to Menger's Theorem [3], the minimum number of nodes whose removal disconnects a and b, that is $\kappa(a, b)$, is equal to the maximum number of independent paths between a and b. The authors in [7] proves that the maximum number of independent paths between a and b in an undirected graph $G(V, E)$ is equivalent to the maximal value of flow from a to b or the minimal capacity of an $a - b$ cut, the set of nodes such that any path from a to b contains a member of the cut, in $G'(V', E')$. □

EXAMPLE 3.13. *Consider nodes r_1 and r_6 of graph G_2 in Figure 2. Figure 3 shows the corresponding flow network, where the dash line (across edges (r_3', r_3''), (r_4', r_4'')) in the figure cuts the flow from r_1 to r_6 with a minimum cost of 2. The max flow/min cut has value 2. Indeed, $\kappa(r_1, r_6) = 2$.* □

Recall that in a $(k + 1)$-connected v-union, between each pair of nodes there are at least $k + 1$ paths. Thus, if $\kappa(a, b) = k + 1$, a and b belong to different v-unions, while a and a' belong to the same v-union, we must have $\kappa(a', b) \geq k + 1$. We thus have the following sufficient and necessary condition for k-robustness.

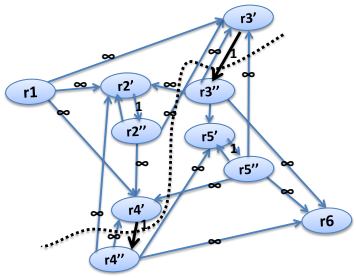

Figure 3: Flow network for G_2 in Figure 2.

THEOREM 3.14 (MAX-FLOW CONDITION). *Let $G(V, E)$ be an input similarity graph. Graph G is k-robust if and only if for every pair of adjacent $(k+1)$-connected v-unions Q and Q', there exist two nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$ such that the max flow from a to b in the corresponding flow network is at least $k + 1$.* □

PROOF. According to Menger's Theorem [3], $\kappa(a, b)$ in G is equivalent to the max-flow from a to b in the corresponding flow network. We need to prove that graph G is k-robust if and only if for each pair of adjacent $(k + 1)$-connected v-unions Q and Q', there exists two nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$ such that $\kappa(a, b) \geq k + 1$.

We first prove that if G is k-robust, for each pair of adjacent $(k + 1)$-connected v-unions Q and Q', there exists two nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$ such that $\kappa(a, b) \geq k + 1$. Since G is k-robust, for each pair of nodes a and b in G, we have $\kappa(a, b) \geq k+1$.

We next prove that if G is not k-robust, there exists a pair of adjacent $(k + 1)$-connected v-unions Q and Q' such that for each pair of nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$, we have $\kappa(a, b) < k+1$. Since G is not k-robust, there exists a separator $\bar{S}$, a set of nodes in G with size no greater than k whose removal disconnects G into two sub-graphs $\bar{X}$ and $\bar{Y}$. Suppose Q and Q' are two v-unions in G such that $Q \subseteq \bar{X}, Q' \subseteq \bar{Y}$ and $Q \cap Q' \neq \emptyset$. For each pair of nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$, we have $a \in \bar{X}$ and $b \in \bar{Y}$, and removing the set of nodes in $\bar{S}$ disconnects a and b; thus $\kappa(a, b) < k + 1$.

The above two cases proves that graph G is k-robust if and only if for every pair of adjacent $(k + 1)$-connected v-unions Q and Q', there exist two nodes $a \in Q \setminus Q'$ and $b \in Q' \setminus Q$ such that $\kappa(a, b) \geq k+1$, i.e. the max flow from a to b in the corresponding flow network is at least $k + 1$. □

If a graph G is not k-robust, we shall split it into subgraphs for further core finding. In the corresponding flow network, each edge in the minimum cut must be between a pair of nodes derived from the same node in G (other edges have capacity ∞). These nodes cannot belong to any core and we use them as separator nodes, denoted by $\bar{S}$. Suppose the separator separates G into $\bar{X}$ and $\bar{Y}$ (there can be more subgraphs). We then split G into $\bar{X} \cup \bar{S}$ and $\bar{Y} \cup \bar{S}$.

Note that we need to include $\bar{S}$ in both sub-graphs to maintain the integrity of each v-union. To understand why, consider G_2 in Figure 2 where $\bar{S} = \{r_3, r_4\}$. According to the definition, there is no 2-robust core. If we split G_2 into $\{r_1 - r_2\}$ and $\{r_5 - r_6\}$ (without including $\bar{S}$), both subgraphs are 2-robust and we would return them as cores. The problem happens because v-cliques $Q_1 - Q_4$ "disappear" after we remove the separators r_3 and r_4. Thus, we should split G_2 into $\{r_1 - r_4\}$ and $\{r_3 - r_6\}$ instead and that would further trigger splitting on both subgraphs. Eventually we wish to exclude the separator nodes from any core, so we mark them as "separators" and exclude them from the returned cores.

Algorithm SPLIT (details in Algorithm 2) takes a $(k+1)$-overlap subgraph G as input and decides if G is k-robust. If not, it splits G into subgraphs on which we will then re-apply screening.

Algorithm 2 SPLIT($G, \bar{C}, k$)

Input: G: Simplified similarity graph.
$\bar{C}$: Set of cores.
k: Robustness requirement.
Output: $\bar{G}$ Set of subgraphs in G.

```
1: for each adjacent (k+1)-connected v-unions Q, Q' do
2:    find a pair of nodes a ∈ Q \ Q', b ∈ Q' \ Q.
3:    construct flow-network G' and compute κ(a,b) by Ford &
      Fulkerson Algorithm.
4:    if κ(a,b) ≤ k then
5:       get separator S̄ from G' and remove S̄ from G to obtain
         disconnected subgraphs; mark S̄ as "separator" and add it
         to each subgraph in G.
6:       return the set Ḡ of subgraphs.
7:    end if
8: end for
9: if Ḡ = φ then
10:   add G to C̄.
11: end if
12: return Ḡ;
```

1. For each pair of adjacent $(k+1)$-connected v-unions $Q, Q' \in G$, find $a \in Q \setminus Q', b \in Q' \setminus Q$. Construct flow network $G'(V', E')$ and apply Ford & Fulkerson Algorithm [10] to compute the max flow.
2. Once we find nodes a, b where $\kappa(a, b) \leq k$, use the min cut of the corresponding flow network as separator $\bar{S}$. Remove $\bar{S}$ and obtain several subgraphs. Add $\bar{S}$ back to each subgraph and mark $\bar{S}$ as "separator". Return the subgraphs.
3. Otherwise, G passes k-robustness test and output it as a core.

EXAMPLE 3.15. *Continue with Example 3.13 and $k = 2$. There are four 3-connected v-unions. When we check Q_1 and Q_3, we find $\bar{S} = \{r_3, r_4\}$. We then split G_2 into two subgraphs $\{r_1 - r_4\}$ and $\{r_3 - r_6\}$, while marking r_3 and r_4 as "separators".*

Now consider graph G_1 in Figure 2 and $k = 2$. There are four 3-connected v-unions (actually four v-cliques) and six pairs of adjacent v-unions. For Q_1 and Q_2, we check nodes r_2 and r_4 and find $\kappa(r_2, r_4) = 3$. Similarly we check for every other pair of adjacent v-unions and decide that the graph is 2-robust. □

PROPOSITION 3.16. *Let p be the total number of pairs of adjacent v-unions, and g be the number of nodes in the input graph. Algorithm* SPLIT *runs in time $O(pg^{2.5})$.* □

PROOF. Authors in [7] proves that it takes in time $O(g^{2.5})$ to compute $\kappa(a, b)$ for a pair of nodes a and b in G. In the worst case SPLIT needs to compute $\kappa(a, b)$ for p pairs of adjacent v-unions. Thus, SPLIT runs in time $O(pg^{2.5})$. □

Recall that if we solve the Max-Flow Problem directly for each pair of sources in the original graph, the complexity is $O(|\bar{L}|^{4.5})$, which would be dramatically higher.

3.3.3 Full algorithm

We are now ready to present the full algorithm, CORE (Algorithm 3). Initially, it initializes the working queue $\mathbf{Q}$ with only input G (Line 1). Each time it pops a subgraph G' from $\mathbf{Q}$ and invokes SCREEN (Lines 3-4). If the output of SCREEN is still G' (so G' is a $(k+1)$-overlap subgraph) (Line 5), it removes any node with mark "separator" in G' and puts the new subgraph into the working queue (Line 7), or invokes SPLIT on G' if there is no separator (Line 9). Subgraphs output by SCREEN and SPLIT are added

Algorithm 3 CORE($G, \bar{L}, k$)

Input: G: Simplified similarity graph.
$\bar{L}$: Inverted list of the similarity graph.
k: Robustness requirement.
Output: $\bar{C}$ Set of cores in G.

```
1: Let Q = {G}, C̄ = ∅;
2: while Q ≠ φ do
3:    Pop G' from Q;
4:    Let P̄ = SCREEN(G', L̄, k, C̄);
5:    if P̄ = {G'} then
6:       if G' contains "separator" nodes then
7:          Remove separators from G' and add the result to Q if
            it is not empty;
8:       else
9:          Let S̄ = SPLIT(G', k, C̄);
10:         add graphs in S̄ to Q;
11:      end if
12:   else
13:      add graphs in P̄ to Q;
14:   end if
15: end while
16: return C̄;
```

to the queue for further examination (Lines 10, 13) and identified cores are added to $\bar{C}$, the core set. It terminates when $\mathbf{Q} = \emptyset$.

The correctness of algorithm CORE is guaranteed by the following Lemmas.

LEMMA 3.17. *For each pair of adjacent nodes r, r' in graph G, there exists a maximal k-robust partitioning such that r, r' are in the same subgraph.* □

PROOF. For each pair of adjacent nodes r, r' in G, we prove the existence of such a maximal k-robust partitioning by constructing it.

By definition, adjacent node r, r' form a v-clique C. Therefore, there exists a maximal v-clique C' in G that contains r, r', i.e., $C \subseteq C'$. V-clique C' can be obtained by keep adding nodes in G to C so that each newly-added node is adjacent to each node in current clique until no nodes in G can be added to C'. By definition, any v-clique is k-robust, therefore there exists a maximal k-robust subgraph G' in G such that $C' \subseteq G'$. Graph G' can be obtained by keep adding nodes in G to C' so that each newly-added node is adjacent to at least $k + 1$ nodes in current graph G' until no nodes in G can be added to G'. We remove G' from G and take G' as a subgraph in the desired partitioning.

We repeat the above process to a randomly-selected pair of adjacent nodes in the remaining graph $G \setminus G'$ until it is empty. The desired partitioning satisfies Condition 1 and 2 of Definition 3.2 because the above process makes sure each subgraph is exclusive and k-robust; it satisfies Condition 3 of Definition 3.2 because the above process makes sure each subgraph is maximal, which means merging arbitrary number of subgraphs in the partitioning would violate Condition 2.

In summary, the desired partitioning is a maximal k-robust partitioning. It proves that for each pair of adjacent nodes r and r' in graph G, there exists a maximal k-robust partitioning such that r and r' are in the same subgraph. □

LEMMA 3.18. *The set of nodes in a separator $\bar{S}$ of graph G does not belong to any core in G, where $|\bar{S}| \leq k$.* □

PROOF LEMMA 3.18. Suppose the set $\bar{S}$ of nodes separate G into m disconnected sets $\bar{X}_i, i \in [1, m], m > 0$. To prove that each

node $r \in \bar{S}$ does not belong to any core in G, we prove that for a node $r' \in G, r' \neq r$, there exists a maximal k-robust partitioning such that r and r' are separated. Node r' falls into the following cases: 1) $r' \in \bar{X}_i, i \in [1, m]$; 2) $r' \in \bar{S}$.

Consider Case 1) where $r' \in \bar{X}_i, i \in [1, m]$. We construct a maximal k-robust partitioning of G where r and r' are in different subgraphs. We start with a maximal k-robust subgraph G' in G that contains r and r'' where r'' is adjacent to r and in $\bar{X}_j, j \neq i, j \in [1, m]$, and find other maximal k-robust subgraphs as in Lemma 3.17. Since $\bar{S}$ separates $\bar{X}_i$ and $\bar{X}_j$, maximal k-robust subgraph G' that contains r and r'' does not contain any node in $\bar{X}_i$. It proves that there exists a maximal k-robust partitioning of G where r and r' are not in the same subgraph.

Consider Case 2) where $r' \in \bar{S}$. We construct a maximal k-robust partitioning of G such that r and r' are in different subgraphs. We create two maximal k-robust subgraphs G' and G'', where G' contains r and an adjacent node $r_i \in \bar{X}_i, i \in [1, m]$, G'' contains r' and an adjacent node $r_j \in \bar{X}_j, j \neq i, j \in [1, m]$. We create other subgraphs as in Lemma 3.17. Since each path between $r_i \in \bar{X}_i$ and $r_j \in \bar{X}_j$ contains at least one node in $\bar{S}$ and $|\bar{S}| \leq k$, graph $G' \cup G''$ is not k-robust. Therefore, the created partitioning is a maximal k-robust partitioning. It proves that there exists a maximal k-robust partitioning of G where r and r' are not in the same subgraph.

Given the above two cases, we have that any node in separator $\bar{S}$ of G does not belong to any core in G, where $|\bar{S}| \leq k$. □

PROPOSITION 3.19. *Let G be the input graph and q be the number of $(k+1)$-connected v-unions in G. Define a, p, g, m, and $|\bar{L}|$ as in Proposition 3.10 and 3.16. Algorithm* CORE *finds correct cores of G in time $O(q((m^2+a)|\bar{L}|+pg^{2.5}))$ and is order independent.* □

PROOF. We first prove that CORE correctly finds cores in G, that is 1) nodes not returned by CORE do not belong to any core; 2) each subgraph returned by CORE forms a core.

We prove that nodes not returned by CORE do not belong to any core in G. Nodes not returned by CORE belong to separators of subgraphs in G. Suppose $\bar{S}$ is a separator of graph $G_n \in \mathbf{Q}$ found in either SCREEN or SPLIT phase, where $G_n \subseteq G, n \geq 0, G_0 = G$, and $\bar{S}$ separates G_n into m sub-graphs $\bar{X}_n^i, i \in [1, m], m > 1$. Graph $G_n^i \in \mathbf{Q}$ is a subgraph of G_n such that any node $r \in \bar{X}_n^j, j \in [1, m], j \neq i$ does not belong to G_n^i. Nodes removed in G_n^i by CORE belong to separator $\bar{S}$ in G_n. Given Lema 3.18, such nodes do not belong to any core in G_n and thus does not belong to any core in G.

We next prove that each subgraph returned by CORE forms a core in G. We prove two cases: 1) subgraph G' in G forms a core if there exists a separator $\bar{S}$ that disconnects G' from G, where $|\bar{S}| \leq k$ and $G' \cup \bar{S}$ and G' are both k-robust; 2) if a subgraph is a core in G_n^i, it is a core in graph G_n.

We consider Case 1) that subgraph G' in G forms a core if there exists a separator $\bar{S}$ that disconnects G' from G, where $|\bar{S}| \leq k$ and $G' \cup \bar{S}$ and G' are both k-robust. For a pair of nodes r_1, r_2 in G', we prove that there exists no maximal k-robust partitioning where r_1 and r_2 are in different subgraphs. Suppose such a partitioning exists, and G_1, G_2 are subgraphs containing r_1, r_2 respectively. Since $G_1, G_2 \subseteq G' \cup \bar{S}$, we have that $G_1 \cup G_2$ is k-robust, it violates the fact that the result of merging any two subgraphs in a maximal k-robust partitioning is not k-robust. Therefore, there exists no maximal k-robust partitioning where r_1 and r_2 are in different subgraphs. It proves that G' is a core in G.

We next consider Case 2) that if a subgraph G' is a core in G_n^i, it is a core in graph G_n. We prove that a pair of nodes $r_1, r_2 \in G'$ belong to the same subgraph of all maximal k-robust partitioning in G_n. Suppose there exists such a partitioning of G_n where $r_1 \in G_1, r_2 \in G_2$. Since $G_n^i \subseteq \bar{X}_n^i \cup \bar{S}$, we have $G_1, G_2 \subseteq G_n^i$, otherwise G_1, G_2 are not k-robust. Since r_1, r_2 belong to the same core in G_n^i, we have $G_1 = G_2$. It proves that if G' is a core in G_n^i, it is a core in G_n.

The above two cases prove that each subgraph returned by CORE forms a core in G. In summary, nodes not returned by CORE do not belong to any core, and each subgraph returned by CORE forms a core in G. Thus, CORE correctly finds all cores in G. It further proves that the result of CORE is independent from the order in which we find and remove separators of graphs in $\mathbf{Q}$.

We now analyze the time complexity of CORE. For each $(k+1)$-connected v-unions in G, it takes in time $O(m^2+a)|\bar{L}|$ to proceed SCREEN phase and in time $O(pg^{2.5})$ to proceed SPLIT phase. In total there are q v-unions in G, thus the algorithm takes in time $O(q((m^2+a)|\bar{L}|+pg^{2.5}))$. □

Table 3: Step-by-step core identification in Example 3.20.

Input	Method	Output
G_2	SCREEN	G_2
G_2	SPLIT	$G_2^1 = \{r_1 - r_4\}, G_2^2 = \{r_3 - r_6\}$
G_2^1	SCREEN	$G_2^3 = \{r_3\}, G_2^4 = \{r_4\}$
G_2^2	SCREEN	$G_2^3 = \{r_3\}, G_2^4 = \{r_4\}$
G_2^3	SCREEN	-
G_2^4	SCREEN	-
G	SCREEN	$G^1 = \{r_{1/\ldots/7}\}, G^2 = \{r_{11}, r_{12}, r_{14/15}\}$, $G^3 = \{r_{16/17/18}\}$
G^1	SCREEN	Core $\{r_1 - r_7\}$
G^2	SCREEN	$G^4 = \{r_{11}\}, G^5 = \{r_{14/15}\}$
G^3	SCREEN	Core $\{r_{16} - r_{18}\}$
G^4	SCREEN	-
G^5	SCREEN	Core $\{r_{14} - r_{15}\}$

EXAMPLE 3.20. *First, consider graph G_2 in Figure 2 and $k = 2$. Table 3 shows the step-by-step core identification process. It passes screening and is the input for* SPLIT. SPLIT *then splits it into G_2^1 and G_2^2, where r_3 and r_4 are marked as "separators".* SCREEN *further splits each of them into $\{r_3\}$ and $\{r_4\}$, both discarded as each represents a single node (and is a separator). So* CORE *does not output any core.*

Next, consider the motivating example, with the input shown in Table 2(b) and $k = 1$. Originally, $\mathbf{Q} = \{G\}$. After invoking SCREEN *on G, we obtain three subgraphs G^1, G^2, and G^3.* SCREEN *outputs G^1 and G^3 as cores since each contains a single node that represents multiple records. It further splits G^2 into two single-node graphs G^4 and G^5, and outputs the latter as a core. Note that if we remove the 1-robustness requirement, we would merge $r_{11} - r_{18}$ to the same core and get false positives.* □

Case study: On the data set with 6.8M records, our core-identification algorithm finished in 1.4 minutes. SCREEN is invoked 102K times in total; except the original graph, the size of the input is at most 10.5K and on average 2.2. SPLIT is invoked only 384 times; the size of the input is at most 175 and on average 10. Recall that the simplified inverted index contains .15M entries, so SCREEN reduces the size of the input to SPLIT by three orders of magnitude.

4. CHAIN IDENTIFICATION

The second stage clusters the cores and the remaining records, which we call *satellites*, into chains. To avoid merging records based only on weak evidence, we require that *a cluster cannot contain more than one satellite but no core*. The key in clustering is two fold: first, we wish to leverage the strong evidence that we collect from the cores generated in the first stage; second, we need to

be tolerant to variety of values within the same chain. This section first describes the objective function for clustering (Section 4.1) and then proposes a greedy algorithm for clustering (Section 4.2).

4.1 Objective function

Ideally, we wish that each cluster is *cohesive* (each element, being a core or a satellite, is close to other elements in the same cluster) and different clusters are *distinct* (each element is fairly different from those in other clusters). Since we expect that businesses in the same chain may have fairly different attribute values, we adopt *Silhouette Validation Index (SV-index)* [12] as the objective function as it is more tolerant to diversity within a cluster. Given a clustering $\mathcal{C}$ of elements $\mathbf{E}$, the SV-index of $\mathcal{C}$ is defined as follows.

$$S(\mathcal{C}) = Avg_{e\in\mathbf{E}} S(e); \tag{1}$$

$$S(e) = \frac{a(e) - b(e) + \alpha}{\max\{a(e), b(e)\} + \beta}. \tag{2}$$

Here, $a(e) \in [0,1]$ denotes the similarity between element e and its own cluster, $b(e) \in [0,1]$ denotes the maximum similarity between e and another cluster, $\beta > \alpha > 0$ are small numbers to keep $S(e)$ finite and non-zero (we discuss in Section 5 how we set the parameters in this section). A nice property of $S(e)$ is that it falls in $[-1,1]$, where a value close to 1 indicates that e is in an appropriate cluster, a value close to -1 indicates that e is mis-classified, and a value close to 0 while $a(e)$ is not too small indicates that e is equally similar to two clusters that should possibly be merged. Accordingly, we wish to obtain a clustering with the maximum SV-index. We next describe how we maintain set signatures and compare an element with a cluster.

Set signature: When we maintain the signature for a core or a cluster, we keep all values of an attribute and assign a high *weight* to a popular value. Specifically, let $\bar{R}$ be a set of records. Consider value v and let $\bar{R}(v) \subseteq \bar{R}$ denote the records in $\bar{R}$ that contain v. The weight of v is computed by $w(v) = \frac{|\bar{R}(v)|}{|\bar{R}|}$.

EXAMPLE 4.1. *Consider* phone *for core* $Cr_1 = \{r_1 - r_7\}$ *in Table 1. There are* 7 *business listings in* Cr_1*,* 5 *providing* 808 *(*$r_1 - r_5$*), one providing* 101 *(*r_6*), and one providing* 102 *(*r_7*). Thus, the weight of* 808 *is* $\frac{5}{7} = .71$ *and the weight for* 101 *and* 102 *is* $\frac{1}{7} = .14$*, showing that* 808 *is the primary phone for* Cr_1*.* □

Note that when we compare an element e with its own cluster Ch, we generate the signature of Ch using its elements excluding e.

Similarity computation: We consider that an element e is similar to a chain Ch if they have highly similar values on common-value attributes (*e.g.*, name), share at least one *primary* value (we explain "primary" later) on dominant-value attributes (*e.g.*, phone, domain-name); in addition, our confidence is higher if they also share values on multi-value attributes (*e.g.*, category). Note that although we assume different branches in a business chain should have different values on distinct-value attributes, for some coarse-granularity values, such as state or region of the location, we still often observe sharing of values (*e.g.*, a business chain in one state, or in a few neighboring states). We can treat such attributes (*e.g.*, state) as a multi-value attribute. Formally, we compute the similarity $sim(e, Ch)$ as follows.

$$sim(e, Ch) = \min\{1, sim_s(e, Ch) + \tau w_m sim_{multi}(e, Ch)\}; \tag{3}$$

$$sim_s(e, Ch) = \frac{w_c sim_{com}(e, Ch) + w_o sim_{dom}(e, Ch)}{w_c + w_o}; \tag{4}$$

$$\tau = \begin{cases} 0 \text{ if} sim_s(e, Ch) < \theta_{th}, \\ 1 \text{ otherwise.} \end{cases} \tag{5}$$

Here, sim_{com}, sim_{dom}, and sim_{multi} denote the similarity for common-, dominant-, and multi- attributes respectively. We take the weighted sum of sim_{com} and sim_{dom} as strong indicator of e belonging to Ch (measured by $sim_s(e, Ch)$), and only reward weak indicator sim_{multi} if $sim_s(e, Ch)$ is above a pre-defined threshold θ_{th}. Weights $0 < w_c, w_o, w_m < 1$ indicate how much we reward value similarity or penalize value difference; we learn the weights from sampled data.

Common-Value attribute: Similarity sim_{com} is computed as the average of similarities on each common-value attribute A. For each A, e and Ch may each contain a set of values. We penalize values with low similarity and apply cosine similarity (in practice, we take into consideration various representations of the same value):

$$sim_{com}(e.A, Ch.A) = \frac{\sum_{v\in e.A\cap ch.A} w(v)^2}{\sqrt{\sum_{v\in e.A} w(v)^2}\sqrt{\sum_{v'\in ch.A} w(v')^2}}. \tag{6}$$

Dominant-value attribute: Similarity sim_{dom} rewards similarity on primary values (values with high weights) but meanwhile is tolerant to other different values. If the primary value of an element is the same as that of a cluster on a dominant-value attribute, we consider them having probability p to be in the same chain. Then, if they share n such values, the probability becomes $1 - (1-p)^n$. Since we use weight to measure whether the value is primary and allow slight difference on values, with a value v from e and v' from Ch, we consider the probability that e and Ch belong to the same chain as $p \cdot w_e(v) \cdot w_{Ch}(v') \cdot s(v, v')$, where $w_e(v)$ measures the weight of v in e, $w_{Ch}(v')$ measures the weight of v' in Ch, and $s(v, v')$ measures the similarity between v and v'. Therefore, we compute $sim_{dom}(r.A, Ch.A)$ as follows.

$$sim_{dom}(e.A, Ch.A) = 1 - \prod_{v\in e.A, v'\in ch.A} (1 - p\cdot w_e(v)\cdot w_{Ch}(v')\cdot s(v, v')). \tag{7}$$

Multi-Value attribute: We allow diversity in such attributes and use a variant of Jaccard distance for similarity computation. Formally, $sim_{multi}(e, Ch)$ is computed as the sum of the similarities for each multi-value attribute A. For each A, without losing generality, assume $|e.A| \le |Ch.A|$ and we treat two values as the same if their similarity is deemed high (above a threshold); we have

$$sim_{multi}(e.A, Ch.A) = \frac{\sum_{v\in e.A} \max_{v'\in Ch.A, s(v,v')>\theta} s(v, v') \max\{w_e(v), w_{Ch}(v')\}}{\sum_{v\in e.A} \max_{v'\in Ch.A, s(v,v')>\theta} \max\{w_e(v), w_{Ch}(v')\}} \tag{8}$$

EXAMPLE 4.2. *Consider element* $e = r_{13}$ *and cluster* $Ch_2 = \{r_{14} - r_{15}\}$ *in Example 1.1. Assume* $w_c = w_o = .5, w_m = .05, \theta_{th} = .8, p = .8$*. Since* e *and* Ch_2 *share exactly the same value on* name, category *and* state*, we have* $sim_{\mathsf{name}}(e, Ch_2) = sim_{\mathsf{state}}(e, Ch_2) = sim_{\mathsf{cat}}(e, Ch_2) = 1$*. For dominant-value attributes, both* e *and* Ch_2 *provide* 900 *with weight* 1*, and they do not share URL, so* $sim_{dom}(e, Ch_2) = 1 - (1 - .8\cdot 1\cdot 1\cdot 1) = .8$*. Thus, we have* $sim_s(e, Ch_2) = \frac{.5\cdot 1 + .5\cdot .8}{.5 + .5} = .9 > \theta_{th}$*,* $sim_w(e, Ch_2) = .05 \cdot (1 + 1) = .1$*, so* $sim(e, Ch_2) = \min\{1, .87 + .1\} = .97$*.* □

Attribute weights: We apply attribute weights in order to reward attribute value consistency and penalize value difference. Accordingly, for attribute A, we define two types of weights: *agreement weight* w^{agr} and *disagreement weight* w^{dis}. Disagreement weight is defined as the probability of two records belonging to different chains given that they disagree on A-values. Agreement weight is defined as the probability of two records belonging to the same chain given that they agree on A-values. We distinguish between

ambiguous and unambiguous A-values, i.e., values shared by multiple chains (*e.g.*, name) are considered ambiguous and not strong indicator of two records belonging to the same chain, thus have a lower weight; on the other hand, values owned by a single chain are considered unambiguous and are associated with a high weight. We learn both agreement and disagreement weights from labeled data.

As aforementioned, we consider both agreement and disagreement weights for common-value and dominant-value attributes. For multi-value attributes, we only apply agreement weights to reward weak evidence, and down-weight the weights to make sure the overall similarity is within $[0, 1]$. How we apply agreement and disagreement weights to compute overall similarity can be found in previous work [17].

4.2 Clustering algorithm

In most cases, clustering is intractable [11, 19]. We next propose a greedy algorithm that approximates the optimal clustering. Our algorithm starts with an initial clustering and then iteratively examines if we can improve the current clustering (increase SV-index) by adjusting subsets of the clusters. According to the definition of SV-index, in both initialization and adjusting, we always assign an element to the cluster with which it has the highest similarity.

Initialization: Initially, we (1) assign each core to its own cluster and (2) assign a satellite r to the cluster with the highest similarity if the similarity is above threshold θ_{ini} and create a new cluster for r otherwise. We update the signature of each core along the way. Note that initialization is sensitive in the order we consider the records. Although designing an algorithm independent of the ordering is possible, such an algorithm is more expensive and our experiments show that the iterative adjusting can smooth out the difference.

EXAMPLE 4.3. *Continue with the motivating example in Table 1 and assume $\theta_{th} = .8$. First, consider records $r_1 - r_{10}$, where $Cr_1 = \{r_1 - r_7\}$ is a core. We first create a cluster Ch_1 for Cr_1. We then merge records $r_8 - r_{10}$ to Ch_1 one by one, as they share similar names, and either primary phone number or primary URL.*

Now consider records $r_{11} - r_{20}$; recall that there are 2 cores and 5 satellites after core identification. Figure 4 shows the initialization result $\mathcal{C}_a$. Initially we create two clusters Ch_2, Ch_3 for cores Cr_2, Cr_3. Records $r_{11}, r_{19} - r_{20}$ do not share any primary value on dominant-value attributes with Ch_2 or Ch_3, so have a low similarity with them; we create a new cluster for each of them. Records r_{12} and r_{13} share the primary phone with Cr_2 so have a high similarity; we link them to Ch_2. □

Cluster adjusting: Although we always assign an element e to the cluster with the highest similarity so $S(e) > 0$, the result clustering may still be improved by merging some clusters or moving a *subset* of elements from one cluster to another. Recall that when $S(e)$ is close to 0 and $a(e)$ is not too small, it indicates that a pair of clusters might be similar and is a candidate for merging. Thus, in cluster adjusting, we find such candidate pairs, iteratively adjust them by merging them or moving a subset of elements between them, and choose the new clustering if it increases the SV-index.

We first describe how we find candidate pairs. Consider element e and assume it is closest to clusters Ch and Ch'. If $S(e) \leq \theta_s$, where θ_s is a threshold for considering merging, we call it a *border* element of Ch and Ch' and consider (Ch, Ch') as a candidate pair. We rank the candidates according to (1) how many border elements they have and (2) for each border element e, how close $S(e)$ is to 0. Accordingly, we define the *benefit* of merging Ch and Ch' as $b(Ch, Ch') = \sum_{e \text{ is a border of } Ch \text{ and } Ch'}(1 - S(e))$, and rank the candidate pairs in decreasing order of the benefit.

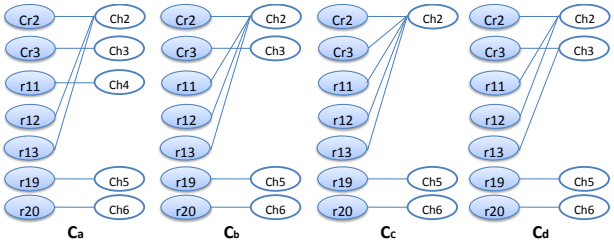

Figure 4: Clustering of $r_{11} - r_{20}$ in Table 1.

Table 4: Element-cluster similarity and SV-index for clusterings in Figure 4. Similarity between an element and its own cluster is in bold and the second-to-highest similarity is in italic. Score $S(e)$ for a border element is in italic.

	Ch_2	Ch_3	Ch_4	Ch_5	Ch_6	$S(e)$
Cr_2	**.9**	.5	.5	.5	.5	.44
Cr_3	*.6*	**1**	.5	.5	.5	.4
r_{11}	*.7*	.5	**1**	.5	.5	*.3*
r_{12}	**.99**	.5	*.95*	.5	.5	*.05*
r_{13}	**1**	.9	*.95*	.5	.5	*.05*
r_{19}	*.5*	*.5*	*.5*	**1**	*.5*	.5
r_{20}	*.5*	*.5*	*.5*	*.5*	**1**	.5

(a) Cluster $\mathcal{C}_a$.

	Ch_2	Ch_3	Ch_5	Ch_6	$S(r)$
r_{11}	**.79**	*.5*	*.5*	*.5*	.37
r_{12}	**.96**	*.5*	*.5*	*.5*	.48
r_{13}	**.97**	*.9*	*.5*	*.5*	*.07*
Cr_2	**.87**	*.5*	*.5*	*.5*	.43
Cr_3	*.58*	**1**	*.5*	*.5*	.42
r_{19}	*.5*	*.5*	**1**	*.5*	.5
r_{20}	*.5*	*.5*	*.5*	**1**	.5

(b) Cluster $\mathcal{C}_b$.

We next describe how we re-cluster elements in a candidate pair (Ch, Ch'). We adjust by merging the two clusters, or moving the border elements between the clusters, or moving out the border elements and merging them. Figure 5 shows the four re-clustering plans for a candidate pair. Among them, we consider those that are valid (*i.e.*, a cluster cannot contain more than one satellite but no core) and choose the one with the highest SV-index. When we compute SV-index, we consider only elements in Ch, Ch' and those that are second-to-closest to Ch or Ch' (their $a(e)$ or $b(e)$ can be changed) such that we can reduce the computation cost. After the adjusting, we need to re-compute $S(e)$ for these elements and update the candidate-pair list accordingly.

EXAMPLE 4.4. *Consider adjusting cluster $\mathcal{C}_a$ in Figure 4. Table 4(a) shows similarity of each element-cluster pair and SV-index of each element. Thus, the SV-index is .32.*

Suppose $\theta_s = .3$. Then, $r_{11} - r_{13}$ are border elements of Ch_2 and Ch_4, where $b(Ch_2, Ch_4) = .7 + .95 + .95 = 2.6$ (there is a single candidate so we do not need to compare the benefit). For the candidate, we have two re-clustering plans, $\{\{r_{11} - r_{13}, Cr_2\}\}$, $\{\{r_{11} - r_{13}\}, \{Cr_2\}\}$, while the latter is invalid. For the former ($\mathcal{C}_b$ in Figure 4), we need to update $S(e)$ for every element and the new SV-index is .4 (Table 4(b)), higher than the original one. □

The full clustering algorithm CLUSTER (details in Algorithm 4) goes as follows.

1. Initialize a clustering $\mathcal{C}$ and a list Que of candidate pairs ranked in decreasing order of merging benefit. (Lines 1-2).
2. For each candidate pair (Ch, Ch') in Que do the following.
   (a) Examine each valid adjusting plan and compute SV-index for it, and choose the one with the highest SV-index. (Line 4).
   (b) Change the clustering if the new plan has a higher SV-index than the original clustering. Recompute $S(e)$ for each relevant element e and move e to a new cluster if appropriate. Update Que accordingly. (Lines 6-16).

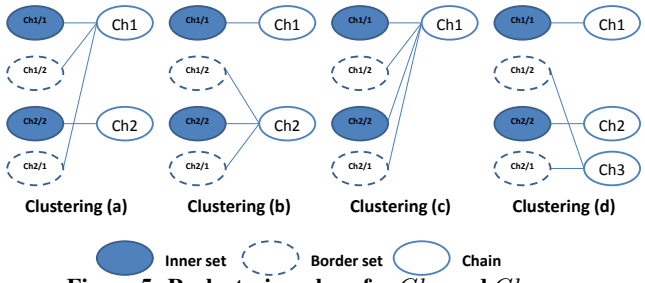

Figure 5: Reclustering plans for Ch_1 and Ch_2.

Algorithm 4 CLUSTER$(\mathbf{E}, \theta_s)$

Input: $\mathbf{E}$: A set of cores and satellites for clustering.
θ_s: Pre-defined threshold for considering merging.
Output: $\mathcal{C}$: A clustering of elements in $\mathbf{E}$.

```
1: Initialize C according to E;
2: Compute S(C) and generate a list Que of candidate pairs;
3: for each candidate pair (Ch, Ch') ∈ Que do
4:    compute SV-index for its valid re-clustering plans and
      choose the clustering C_max with the highest SV-index;
5:    if S(C) < S(C_max) then
6:       let C = C_max, change = true;
7:       while change do
8:          change = false;
9:          for each relevant element e do
10:            recompute S(e);
11:            When appropriate, move e to a new cluster and set
               change = true;
12:            if S(e) < θ_s in the previous or current C then
13:               update the merging benefit of the related candi-
                  date pair and add it to Que or remove it from Que
                  when appropriate;
14:            end if
15:         end for
16:      end while
17:   end if
18: end for
19: return C;
```

3. Repeat Step 2 until $Que = \emptyset$.

PROPOSITION 4.5. *Let l be the number of distinct candidate pairs ever in Que and $|\mathbf{E}|$ be the number of input elements. Algorithm* CLUSTER *takes time $O(l \cdot |\mathbf{E}|^2)$.* □

PROOF. It takes time $O(|\mathbf{E}|^2)$ to initialize clustering $\mathcal{C}$ and list Que. It takes $|\mathbf{E}|^2$ to check each distinct candidate pair in Que, where it takes $O(|\mathbf{E}|)$ to examine all valid clustering plans and select the one with highest SV-index (Step 2(a)), and it takes $O(|\mathbf{E}|^2)$ to recompute SV-index for all relevant elements and update Que (Step 2(b)). In total there are l distinct candidate pairs ever in Que, thus CLUSTER takes time $O(l \cdot |\mathbf{E}|^2)$. □

Note that we first block records according to name similarity and take each block as an input, so typically $|\mathbf{E}|$ is quite small. Also, in practice we need to consider only a few candidate pairs for adjusting in each input, so l is also small.

EXAMPLE 4.6. *Continue with Example 4.4 and consider adjusting $\mathcal{C}_b$. Now there is one candidate pair (Ch_2, Ch_3), with border r_{13}. We consider clusterings $\mathcal{C}_c$ and $\mathcal{C}_d$. Since $S(\mathcal{C}_c) = .37 < .40$ and $S(\mathcal{C}_d) = .32 < .40$, we keep $\mathcal{C}_b$ and return it as the result. We do not merge records $Ch_2 = \{r_{11} - r_{15}\}$ with $Ch_3 = \{r_{16} - r_{18}\}$, because they share neither phone nor the primary URL.* CLUSTER *returns the correct chains.* □

Table 5: Statistics of the business-listing subsets.

	#Records	#Chains	Chain sizes	#Single-business records
Random	2062	30	[2, 308]	503
AI	2446	1	2446	0
UB	322	7	[2, 275]	5
FBIns	1149	14	[33, 269]	0

5. EXPERIMENTAL EVALUATION

This section describes experimental results on a real-world business-listing data set. Experimental results show high accuracy and scalability of our techniques.

5.1 Experiment settings

Data and golden standard: We experimented on a set of business listings in the US obtained from *YellowPages.com*. There are 6.8 million business listings, each with attributes name, phone, URL, location and category. We experimented on the whole data set to study scalability of our techniques.

To evaluate accuracy of our techniques, we considered four subsets. First, we considered a *Random data set* with 2062 records, where 1559 belong to 30 randomly selected business chains, and 503 do not belong to any chain; among the 503 records, 86 are highly similar in name to records in the business chains and the rest are randomly selected. We also considered three hard cases. (1) *AI data set* contains 2446 records for the same business chain *Allstate Insurance*. These records have the same name, but 1499 provide URL *"allstate.com"*, 854 provide another URL *"allstateagencies.com"*, while 130 provide both, and 227 records do not provide any value for phone or URL. (2) *UB data set* contains 322 records with exactly the same name *Union Bank* and highly similar category values; 317 of them belong to 9 different chains while 5 do not belong to any chain. (3) *FBIns data set* contains 1149 records with similar names and highly similar category values; they belong to 14 different chains. Among the records, 708 provide the same wrong name *Texas Farm Bureau Insurance* and meanwhile provide a wrong URL *farmbureauinsurance-mi.com*. For each data set, we manually verified all the chains by checking store locations provided by the business-chain websites and used it as the golden standard. Table 5 shows statistics of the four subsets.

Measure: We considered each business chain as a cluster and compared pairwise linking decisions with the golden standard. We measured the quality of the results by *precision* (P), *recall* (R), and *F-measure* (F). If we denote the set of true-positive pairs by TP, the set of false-positive pairs by FP, and the set of false-negative pairs by FN, then, $P = \frac{|TP|}{|TP|+|FP|}$, $R = \frac{|TP|}{|TP|+|FN|}$, $F = \frac{2PR}{P+R}$.

Implementation: We implemented the technique we proposed in this paper, and call it CORECLUSTER. In core generation, we considered two records to be similar if (1) their name similarity is above .95; and (2) they share at least one phone or URL domain name. We required 1-robustness for cores. In clustering, (1) for blocking, we put records whose name similarity is above .8 in the same block; (2) for similarity computation, we computed string similarity by Jaro-Winkler distance [4], we set $\alpha = .01, \beta = .02, \theta_{th} = .6, p = .8$, we learned other weights from 1000 records randomly selected from *Random* data and we used the learnt weights on all data sets; (3) for clustering, we set $\theta_{ins} = .8$ for initialization and $\theta_s = .2$ for adjusting. We discuss later how these choices may affect our results.

For comparison, we also implemented the following baselines:

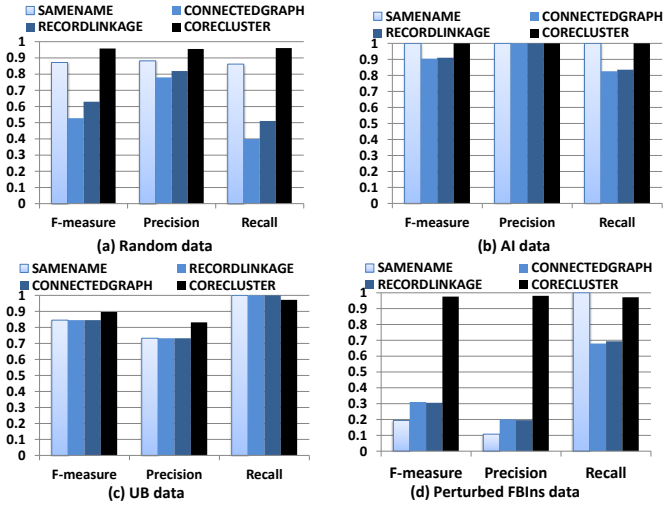

Figure 6: Overall results on each data set.

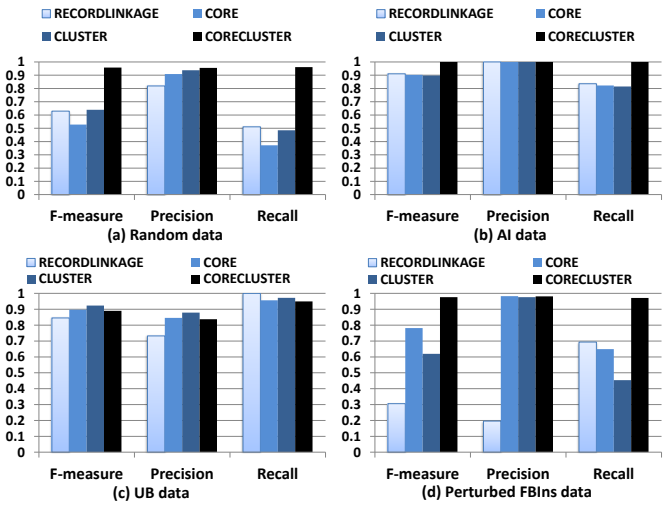

Figure 7: Contribution of different components.

- SAMENAME links records with highly similar names (similarity above .95);
- CONNECTEDGRAPH generates the similarity graph as CORECLUSTER but considers each connected subgraph as a chain;
- RECORDLINKAGE computes record similarity by Eq.(3) with the same weights as in CORECLUSTER and then applies one baseline clustering technique PARTITION [13], which links all records that are transitively similar. (We experimented on two other clustering methods and obtained similar results.)

We implemented the algorithms in Java. We used a Linux machine with Intel Xeon X5550 processor (2.66GHz, cache 8MB, 6.4GT/s QPI). We used MySQL to store the business listings.

5.2 Evaluating effectiveness

We first evaluate effectiveness of our algorithms. Figure 6 compares CORECLUSTER with the three baseline methods on the data sets. We have the following observations on data sets *Random, AI,* and *UB*. (1) CORECLUSTER obtains the highest F-measure (above .9) on each data set. It has the highest precision on each subset as it applies core identification and leverages the strong evidence collected from resulting cores. It also has a very high recall (above .95) on each subset because the clustering phase is tolerant to diversity of values within chains. (2) SAMENAME can have false positives when listings of highly similar names belong to different chains and can also have false negatives when some listings in a chain have fairly different names from other listings. It only performs well in *AI*, where it happens that all listings have the same name and belong to the same chain. (3) CONNECTEDGRAPH requires in addition sharing at least one phone or URL domain. As a result, it has a lower recall than SAMENAME; it has less false positives than SAMENAME, but because it has less true positives, its precision can appear to be lower too. (4) RECORDLINKAGE requires high weighted similarity between the records, which is a weaker requirement for merging records than CONNECTEDGRAPH. On various data sets it has very similar numbers of false positives to but much more true positives than CONNECTEDGRAPH; as a result, it has a higher recall and also a higher precision.

On the *FBIns* data set, because a large number of listings (708) have both a wrong name and a wrong URL, each method wrongly puts all records in the same chain. We manually perturbed the data as follows: (1) among the 708 listings with wrong URLs, 408 provide a single (wrong) URL and we fixed it; (2) for all records we set name to *"Farm Bureau Insurance"*, so removed hints from business names. Even after perturbing, this data set remains the hardest data set and Figure 6(d) shows the results. We observe that all baseline methods have very low F-measure (below .3) while CORECLUSTER still obtains a F-measure as high as .98. We used the perturbed *FBIns* data set hereafter instead of the original one for other experiments.

Contribution of different components: We compared CORECLUSTER with (1) CORE, which applies Algorithm COREIDENTIFICATION but does not apply clustering, and (2) CLUSTER, which considers each individual record as a core and applies Algorithm CLUSTER. Figure 7 shows the results on each data set. First, we observe that CORE improves over the baseline method RECORDLINKAGE on precision but has a lower recall, because it sets a high requirement for merging records into chains. Note however that its goal is indeed to obtain a high precision such that the strong evidence collected from the cores are trustworthy for the clustering phase. Second, CLUSTER often performs better than RECORDLINKAGE. On some data sets (*Random, UB*) it can obtain an even higher precision than CORE, because CORE can make mistakes when too many records have erroneous values, but CLUSTER may avoid some of these mistakes by considering also similarity on state and category. However, applying clustering on the results of CLUSTER would not change the results, but applying clustering on the results of CORE can obtain a much higher F-measure, especially a higher recall (98% higher than CLUSTER on *Random*). This is because the result of CLUSTER lacks the strong evidence collected from high-quality cores so the final results would be less tolerant to diversity of values, showing the importance of core identification. Finally, we observe that CORECLUSTER obtains the best results in most of the data sets. The only exception is *UB*, where its F-measure is 2% lower than that of CLUSTER; this data set contains less diverse values for the same chain so CLUSTER has a high recall. We note also that although CORECLUSTER has more false positives than CORE, it can have a higher precision as it has much more true positives.

We next evaluate various choices in the two stages. Unless specified otherwise, we observed similar patterns on each data set and report the results on *Random*; in some cases we report the results on perturbed *FBIns* since results on other data sets are not very distinguishable.

5.2.1 *Core identification*

Robustness requirement: We first show how the robustness requirement can affect the results. Figure 8 shows the results when we vary k. We have three observations. (1) When $k = 0$, we

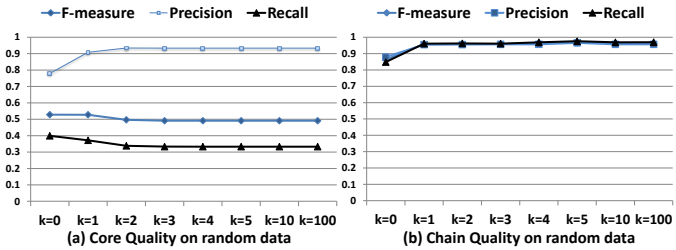

Figure 8: Effect of robustness requirement on *Random* data.

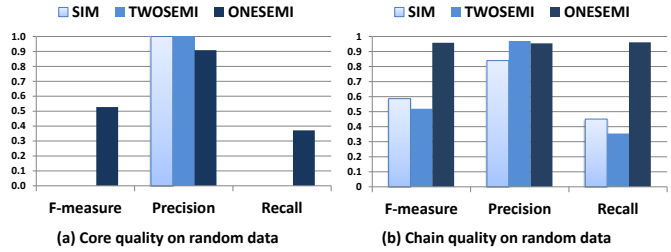

Figure 9: Effect of graph generation on *Random* data.

essentially take every connected subgraph as a core, so the generated cores can have a much lower precision; those false positives cause both a low precision and a low recall for the resulting chains because we may collect some wrong strong evidence while miss some other such evidence. (2) When we vary k from 1 to 4, the number of false positives decreases while that of false negatives increases for the cores, and the F-measure of the chains increases but only very slightly. (3) When we continue increasing k, the results of cores and clusters remain stable. This is because setting k=4 already splits the graph into subgraphs, each containing a single v-clique, so further increasing k would not change the cores. This shows that considering k-robustness is important, but k does not need to be too high.

Graph generation: We compared three edge-adding strategies for similarity graphs: SIM takes weighted similarity on name, phone, URL, category and requires a similarity of over .8; TWOSEMI requires sharing name and at least two values on dominant-value attributes; ONESEMI requires sharing name and one value on dominant-value attributes. Recall that by default we applied ONESEMI. Figure 9 compares these three strategies. We observe that (1) SIM requires similar values on each attribute except location and so has a high precision, with a big sacrifice on recall for the cores; as a result, the F-measure of the chains is very low (.59); (2) TWOSEMI has the highest requirements and so even lower recall than SIM for the cores, and in turn it has the lowest F-measure for the chains (.52). This shows that only requiring high precision for cores with big sacrifice on recall can also lead to low F-measure for the chains.

We also varied the similarity requirement for names and observed very similar results (varying by .04%) when we varied the threshold from .8 to .95.

Core identification: We compared two core-generation strategies: COREIDENTIFICATION iteratively invokes SCREEN and SPLIT, ONLYSCREEN only iteratively invokes SCREEN. Recall that by default we apply COREIDENTIFICATION. We observe the same results on all data sets, showing that the inputs to SPLIT all pass the k-robustness test. This shows that although SCREEN in itself cannot guarantee soundness of the resulting cores (k-robustness), it already does well in obtaining k-robust cores. The cases in which SPLIT can make a difference appear to be rare in practice.

5.2.2 *Clustering*

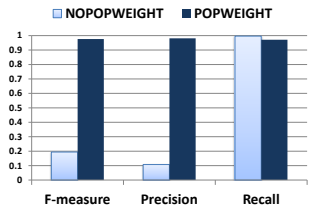

Figure 10: Value weights on perturbed *FBIns* data.

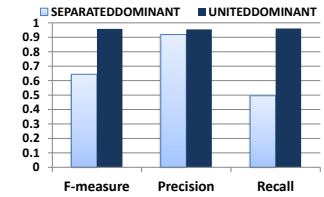

Figure 11: Dominant-value attributes on *Random*.

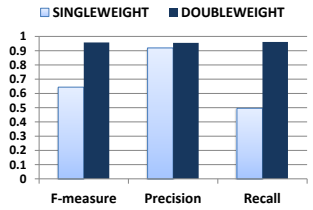

Figure 12: Distinct values on *Random* data.

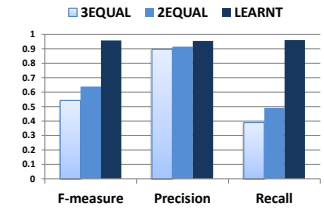

Figure 13: Attribute weights on *Random* data.

Value weight: We first show importance of setting popularity weights in set signature. Figure 10 compares the results with and without setting popularity weights on perturbed *FBIns* data. We observe that setting the popularity weight helps distinguish primary values from unpopular values, thus can significantly improve the precision and so improve F-measure. We also observe an improvement of 1% on *Random* and 5% on *UB* for F-measure.

Attribute weight: We next considered our weight learning strategy. We first compared SEPARATEDDOMINANT, which learns separated weights for different dominant-value attributes, and UNITEDDOMINANT, which considers all such attributes as a whole and learns one single weight for them. By default we applied UNITEDDOMINANT. Figure 11 shows that the latter improves over the former by 95.4% on recall and obtains slightly higher precision, because it penalizes only if neither phone nor URL is shared and so is more tolerant to different values for dominant-value attributes.

Next, we compared SINGLEWEIGHT, which learns a single weight for each attribute, and DOUBLEWEIGHT, which learns different weights for distinct values and non-distinct values for each attribute. By default we applied DOUBLEWEIGHT. Figure 12 shows that DOUBLEWEIGHT significantly improves the recall (by 94%) since it rewards sharing of distinct values, and so can link some satellite records with null values on dominant-value attributes to the chains they should belong to.

We also compared three weight-setting strategies: (1) 3EQUAL considers common-value attributes, dominant-value attributes, and multi-value attributes, and sets the same weight for each of them; (2) 2EQUAL sets equal weight of .5 for common-value attributes and dominant-value attributes, and weight of .1 for each multi-value attribute; (3) LEARNED applies weights learned from labeled data. Recall that by default we applied LEARNED. Figure 13 compares their results. We observe that (1) 2EQUAL obtains higher F-measure than 3EQUAL, since it distinguishes between strong and weak indicators for record similarity; (2) LEARNED significantly outperforms the other two strategies, showing effectiveness of weight learning.

Attribute contributions: We then consider the contribution of each attribute for chain classification. Figure 14 shows the results when we consider only a subset of the attributes on the perturbed *FBIns* data. We have four observations. (1) Considering only name but not any other attribute obtains a high recall but a very low precision, since all listings on this data set have the same name.

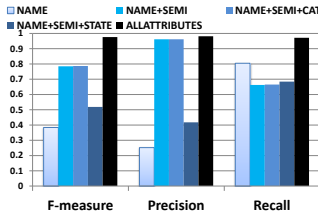

Figure 14: Attribute contribution on perturbed *FBIns*.

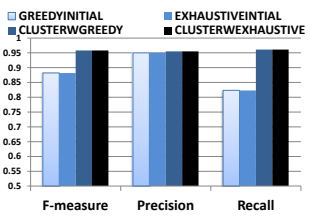

Figure 15: Clustering strategies on *Random* data.

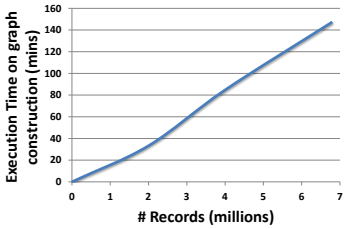

Figure 16: Scalability of our algorithm.

(2) Considering dominant-value attributes in addition to name can improve the precision significantly and improve the F-measure by 104%. (3) Considering category in addition does not further improve the results while considering state in addition even drops the precision significantly, since three chains in this data set contain the same wrong value on state. (4) Considering both category and state improves the recall by 46% and obtains the highest F-measure.

Clustering strategy: We compared four clustering algorithms: GREEDYINITIAL performs only initialization as we described in Section 4; EXHAUSTIVEINITIAL also performs only initialization, but by iteratively conducting matching and merging until no record can be merged to any core; CLUSTERWGREEDY applies cluster adjusting on the results of GREEDYINITIAL, and CLUSTERWEXHAUSTIVE applies cluster adjusting on the results of EXHAUSTIVEINITIAL. Recall that by default we apply CLUSTERWGREEDY. Figure 15 compares their results. We observe that (1) applying cluster adjusting can improve the F-measure a lot (by 8.6%), and (2) exhaustive initialization does not significantly improve over greedy initialization, if at all. This shows effectiveness of the current algorithm CLUSTER.

Robustness w.r.t. parameters: We also ran experiments to test robustness against parameter setting. We observed very similar results when we ranged p from .8 to 1, θ_{th} from .5 to .7, θ_{ini} from .6 to .9, and θ_s from .1 to .4.

5.3 Evaluating efficiency

Our algorithm finished in 2.4 hours on the whole data set, which contains 6.8 million listings. It spent 2.2 hours for graph construction, 1.4 minutes for core generation, and 15 minutes for clustering. This is reasonable given that it is an offline process.

We next focus on graph construction since it costs the longest time. We randomly divided the whole data set into 3 subsets of the same size. We started with one subset and gradually added more. Figure 16 shows that the execution time grows linearly in the size of the data.

5.4 Summary and recommendations

We summarize our observations as follows.

1. Identifying cores and leveraging evidence learned from the cores is crucial in business-chain identification.
2. There are often erroneous values in real data and it is important to be robust against them; applying ONESEMI and requiring $k \in [1, 5]$ already performs well on most data sets that have reasonable number of errors.
3. Learning different weights for different attributes, distinguishing the weights for distinct and non-distinct values, and setting weights of values according to their popularity are critical for obtaining good clustering results.
4. Our algorithm is robust on reasonable parameters setting.
5. Our algorithm is efficient and scalable.

6. RELATED WORK

Record linkage has been extensively studied in the past (surveyed in [6, 16]). Traditional linkage techniques aim at linking records that refer to the same real-world entity, so implicitly assume value consistency between records that should be linked. Business identification is different in that it aims at linking business records that refer to entities in the same chain. The variety of individual business entities requires better use of strong evidence and tolerance on different values even within the same chain. These two features differ our work from any previous linkage technique.

For record clustering in linkage, existing work may apply transitive rule [14], or do match-and-merge [20], or reduce it to an optimization problem [9]. Our work is different in that our core-identification algorithm aims at being robust to a few erroneous records; and our chain-identification algorithm emphasizes leveraging the strong evidence collected from the cores. There has been a lot of work in the clustering community that also first finds cores and uses them as seeds for clustering [1, 21]; however, they identify cores in different ways. Techniques in [21] identify cores either randomly or according to the weighted degrees of nodes in the graph. Techniques in [1] identify cores as *bi-connected components*, where removing any node would not disconnect the graph. Although this is essentially the same as the 1-robustness requirement, they generate overlapping clusters whereas we do not allow overlap between chains or cores.

For record-similarity computation, existing work can be rule based [14], classification based [8], or distance based [5]. There has also been work on weight (or parameter) learning from labeled data [8, 22]. Our work is different in that (1) we weight the values according to their popularity within a core or cluster such that similarity on primary values is rewarded more, (2) we distinguish weights for distinct values and non-distinct values such that similarity on distinct values is rewarded more, and (3) we are tolerant to different values for dominant-value, distinct-value, and multi-value attributes. Note that some previous works are also tolerant to different values but are under different contexts: [12] is tolerant to possibly false values, and [17] is tolerant to out-of-date values; we are tolerant to diversity within the same group.

Finally, we distinguish our work from *group linkage* [15, 18], which has different goals from our work. Authors in [18] decided similarity between groups of records. Authors in [15] studied distinguishing groups of entities with the same identity, such as papers by authors of the same name, by analysis of the social network. Our goal is to find records that belong to the same group.

7. CONCLUSIONS

In this paper we studied how to link business listings for identifying business chains. Our solution consists of two stages: the first stage generates cores with listings that are highly likely to belong to the same chain; the second stage leverages the strong evidence collected from cores and clusters cores and satellite records into

chains. Experimental results show both high efficiency and high accuracy of our methods over a real-world data set. For future work, we plan to apply the same idea in other contexts, such as traditional record linkage applications and temporal record linkage applications, and study its effectiveness.

8. REFERENCES

[1] N. Bansal, F. Chiang, N. Koudas, and F. W. Tompa. Seeking stable clusters in the blogosphere. In *VLDB*, pages 806–817, 2007.

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

[3] H. Bruhn, R. Diestel, and M. Stein. Menger's theorem for infinite graphs with ends. *Journal of Graph Theory*, 50(3):199–211, 2005.

[4] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proc. of IIWEB*, pages 73–78, 2003.

[5] D. Dey. Entity matching in heterogeneous databases: A logistic regression approach. *Decis. Support Syst.*, 44:740–747, 2008.

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

[7] S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

[8] I. P. Fellegi and A. B. Sunter. A theory for record linkage. *Journal of the Americal Statistical Association*, 64(328):1183–1210, 1969.

[9] G. Flake, R. Tarjan, and K. Tsioutsiouliklis. Graph clustering and minimum cut trees. *Internet Mathematics*, 1:385–408, 2004.

[10] L. R. Ford and D. R. Fulkerson. *Flows in networks*. Princeton University Press, 1962.

[11] T. Gonzalez. On the computational complexity of clustering and related problems. *Lecture Notes in Control and Information Sciences*, page 174182, 1982.

[12] S. Guo, X. Dong, D. Srivastava, and R. Zajac. Record linkage with uniqueness constraints and erroneous values. *PVLDB*, 3(1):417–428, 2010.

[13] O. Hassanzadeh, F. Chiang, H. C. Lee, and R. J. Miller. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, pages 1282–1293, 2009.

[14] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2:9–37, 1998.

[15] S. Huang. Mixed group discovery: Incorporating group linkage with alternatively consistent social network analysis. *International Conference on Semantic Computing*, 0:369–376, 2010.

[16] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, 2006.

[17] P. Li, X. L. Dong, A. Maurino, and D. Srivastava. Linking temporal records. *PVLDB*, 4(11):956–967, 2011.

[18] B. W. On, N. Koudas, D. Lee, and D. Srivastava. Group linkage. In *ICDE*, pages 496–505, 2007.

[19] J. Sima and S. E. Schaeffer. On the np-completeness of some graph cluster measures. *Lecture Notes in Computer Science*, pages 530–537, 2006.

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

[21] D. T. Wijaya and S. Bressan. Ricochet: A family of unconstrained algorithms for graph clustering. In *DASFAA*, pages 153–167, 2009.

[22] W. E. Winkler. Methods for record linkage and bayesian networks. Technical report, U.S. Bureau of the Census, 2002.