Incremental Record Linkage

Anja Gruenheid
ETH Zurich
anja.gruenheid@inf.ethz.ch

Xin Luna Dong
Google Inc.
lunadong@google.com

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

ABSTRACT

Record linkage clusters records such that each cluster corresponds to a single distinct real-world entity. It is a crucial step in data cleaning and data integration. In the big data era, the *velocity* of data updates is often high, quickly making previous linkage results obsolete. This paper presents an end-to-end framework that can incrementally and efficiently update linkage results when data updates arrive. Our algorithms not only allow merging records in the updates with existing clusters, but also allow leveraging new evidence from the updates to fix previous linkage errors. Experimental results on three real and synthetic data sets show that our algorithms can significantly reduce linkage time without sacrificing linkage quality.

1. INTRODUCTION

Record linkage (surveyed in [8]) clusters database records such that each cluster corresponds to a single distinct real-world entity (*e.g.*, a business, a person). It is a crucial step in data cleaning and data integration. The big data era raises two challenges for record linkage. First, the volume of data is often huge and applying record linkage usually takes a long time. Second, the velocity of data updates is often high, quickly making previous linkage results obsolete. These challenges call for an incremental strategy, such that we can quickly update linkage results when data updates arrive. There are two goals for incremental linkage. First, we wish that the incremental approach obtains the same or very similar results as applying batch linkage. Second, we wish to conduct incremental linkage significantly faster than batch linkage.

A natural thought for incremental linkage is that for each inserted record, we compare it with existing clusters, then either put it into an existing cluster (*i.e.*, referring to an already known entity), or create a new cluster for it (*i.e.*, referring to a new entity). However, every linkage algorithm may make mistakes and the extra information from the data updates can often help us identify and fix such mistakes, as we illustrate next with an example.

EXAMPLE 1.1. *Figure 1(a) shows a set of 10 business records that represent 5 businesses. For the purpose of illustration, we compute pairwise similarity in a simple way: we compare (1)* name*, (2)* street address *excluding house number, (3) house number in* street address*, (4)* city*, and (5)* phone*; the similarity is* 1 *if all five values are the same,* .9 *if four are the same,* .8 *if three are the same, and* 0 *otherwise. Figure 1(b) shows the* similarity graph *between the records, where each node represents a record and each edge represents the pairwise similarity. It also shows the results of* correlation clustering *(we describe it in Section 2) as the linkage result. Note that it wrongly clusters r_4 with $r_1 - r_3$ because of the wrong phone number from r_4 (in italics); it fails to merge r_5 and r_6 because of the missing information in r_6; and it wrongly merges r_9 with $r_7 - r_8$ instead of with r_{10}, because r_9 appears similar to $r_7 - r_8$ while r_{10} does not (different name, different house number, and missing phone).*

Now consider four updates $\Delta D_1 - \Delta D_4$ (Figure 2(a)); they together insert records $r_{11} - r_{17}$. Figure 3 shows the updated similarity graph and the results of the aforementioned naive approach. It creates a new cluster for r_{11} as it is different from any existing record, and adds the rest of the inserted records to existing clusters.

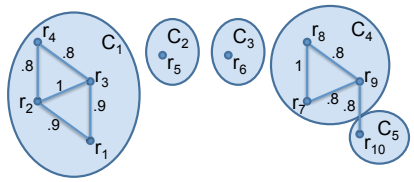
However, a more careful analysis of the inserted nodes allows fixing some previous mistakes and obtaining a better clustering (shown in Figure 2(b)). First, because $r_{12} - r_{13}$ are similar both to r_5 and to r_6, they provide extra evidence to merge r_5 and r_6. Second, when we consider $r_1 - r_4, r_{14} - r_{15}$ jointly, we find that $r_1 - r_3, r_{14} - r_{15}$ are very similar, but r_4 is different from most of them, suggesting moving r_4 out. Third, with $r_{16} - r_{17}$, r_9 appears to be more similar to r_{10} and r_{16} than to $r_7 - r_8$, suggesting moving r_9 from C_4 to C_5. □

Incremental record linkage has been studied before in [12, 13], where the main focus is the case when the matching rules evolve over time. In [13] the authors briefly discussed the case of evolving data and identified a *general incremental* condition under which incremental linkage can be easily carried out using the batch linkage method. This condition requires that for any arbitrary subset of records and its batch clustering results, if we treat each of the rest of the records as a singleton cluster and apply the same algorithm on all of the resulting clusters, we obtain exactly the same results as we apply the algorithm directly on all singleton clusters. As an example, *agglomerative clustering*, which iteratively merges similar clusters, is general incremental. However, not many clustering algorithms satisfy this condition. For example, the aforementioned naive approach that iteratively adds each record into an existing clustering is order-dependent, so does not satisfy this condition. Moreover, many clustering algorithms, such as correlation clustering (we shall explain it soon), operate on records rather than subsets of records, so the batch algorithm cannot directly apply on previous clustering results. In this paper we ask two questions. First, in case the batch linkage algorithm is not general incremental, can we do better than just conducting linkage from scratch? Second, how

Proceedings of the VLDB Endowment, Vol. 7, No. 9

	BizID	ID	name	street address	city	phone
$\mathbf{D_0}$	B_1	r_1	Starbucks	123 MISSION ST STE ST1	SAN FRANCISCO	4155431510
	B_1	r_2	Starbucks	123 MISSION ST	SAN FRANCISCO	4155431510
	B_1	r_3	Starbucks	123 Mission St	San Francisco	4155431510
	B_2	r_4	Starbucks Coffee	340 MISSION ST	SAN FRANCISCO	*4155431510*
	B_3	r_5	Starbucks Coffee	333 MARKET ST	SAN FRANCISCO	4155434786
	B_3	r_6	Starbucks	MARKET ST	San Francisco	
	B_4	r_7	Starbucks Coffee	52 California St	San Francisco	4153988630
	B_4	r_8	Starbucks Coffee	52 CALIFORNIA ST	SAN FRANCISCO	4153988630
	B_5	r_9	Starbucks Coffee	295 California St	San Francisco	4159862349
	B_5	r_{10}	Starbucks	295 California St	San Francisco	

(a)

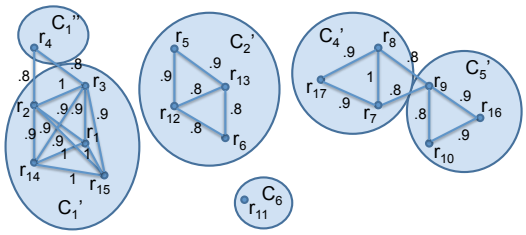


(b)

Figure 1: Original business listings and record linkage results.

	BizID	ID	name	street address	city	phone
$\Delta\mathbf{D}_1$	B_6	r_{11}	Starbucks Coffee	201 Spear Street	San Francisco	4159745077
$\Delta\mathbf{D}_2$	B_3	r_{12}	Starbucks Coffee	MARKET ST	San Francisco	4155434786
	B_3	r_{13}	Starbucks	333 MARKET ST	San Francisco	4155434786
$\Delta\mathbf{D}_3$	B_1	r_{14}	Starbucks	123 MISSION ST STE ST1	SAN FRANCISCO	4155431510
	B_1	r_{15}	Starbucks	123 Mission St Ste St1	San Francisco	4155431510
$\Delta\mathbf{D}_4$	B_5	r_{16}	Starbucks	295 CALIFORNIA ST	SAN FRANCISCO	4159862349
	B_4	r_{17}	Starbucks	52 California Street	SF	4153988630

(a)



(b)

Figure 2: Updates for business listings and record linkage results with *all* updates.

can we make a trade-off between quality of the linkage results and efficiency of the algorithm?

This paper presents a set of algorithms that can incrementally conduct record linkage when new records are inserted and when existing records are deleted or changed (*i.e.*, values are modified). In particular, we make the following three contributions.

- We describe an end-to-end solution for incremental record linkage. Our solution incrementally maintains a similarity graph for the records, and conducts incremental graph clustering, resulting in clusters of records that refer to the same real-world entity.
- For incremental graph clustering, we first propose two optimal algorithms that apply clustering on subsets of the records rather than all records. We then design a greedy approach that conducts linkage incrementally in polynomial time by merging and splitting clusters connected to the updated records, and moving records between those clusters.
- We instantiate our algorithms on two clustering methods that do not require knowing the number of clusters *a priori* and are used often in record linkage: correlation clustering and DB-index clustering. Our experiments on real-world data sets show that our algorithms run significantly faster than batch linkage while obtaining similar results.

While we evaluate our approaches with tabular datasets, they apply to any entity resolution setting where entities can be modeled as nodes and similarities between entities as edges in a graph.

The rest of the paper is organized as follows. Section 2 formally defines the problem and describes an end-to-end solution for incremental record linkage. Sections 3-4 describe our incremental linkage algorithms. Section 5 presents our experimental results, Section 6 discusses related work, and Section 7 concludes.

2. PROBLEM STATEMENT

This section formally defines the problem of incremental record linkage (Section 2.1). We then describe the framework for incremental linkage (Section 2.2). and review techniques for graph clustering, which is a key component in record linkage (Section 2.3).

2.1 Problem definition

Given a set of records, record linkage is essentially a clustering problem, where each cluster contains records that correspond to a single distinct real-world entity. We denote by $\mathbf{D}$ a set of records and by $\mathcal{L}_\mathbf{D}$ a clustering of records in $\mathbf{D}$ as record-linkage results. Ideally, the clustering should have both high *precision* (*i.e.*, records in the same cluster refer to the same real-world entity) and high *recall* (*i.e.*, records referring to the same real-world entity belong to the same cluster). We denote by F the batch linkage method that obtains $\mathcal{L}_\mathbf{D}$ on $\mathbf{D}$; that is, $F(\mathbf{D}) = \mathcal{L}_\mathbf{D}$.

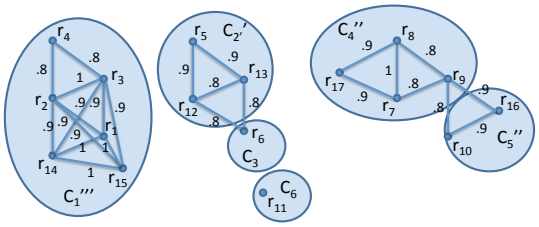


Figure 3: Results of a baseline incremental linkage algorithm.

We consider three types of update operations: `Insert` adds a new record; `Delete` removes an existing record; and `Change` modifies one or a few values of an existing record. Note that `Change` can be achieved by first removing the old record and then inserting the new record; however, as we show later, considering `Change` directly can be more efficient. We call those update operations (`Insert`, `Delete`, and `Change`) made at the same time an *increment*, denoted by $\Delta\mathbf{D}$. We denote the result of applying $\Delta\mathbf{D}$ to $\mathbf{D}$ by $\mathbf{D} + \Delta\mathbf{D}$. Note that because $\Delta\mathbf{D}$ can contain deletes and changes, the number of the resulting records may be lower than the sum of the number of original records and the number of records in the increment; that is, $|\mathbf{D} + \Delta\mathbf{D}| \leq |\mathbf{D}| + |\Delta\mathbf{D}|$. In this paper, we assume every increment $\Delta\mathbf{D}$ is *valid*: the record in a `Delete` or `Change` operation already exists in $\mathbf{D}$, and the record in an `Insert` does not exist in $\mathbf{D}$. We now define incremental linkage.

DEFINITION 2.1 (INCREMENTAL LINKAGE). *Let $\mathbf{D}$ be a set of records and $\Delta\mathbf{D}$ be an increment to $\mathbf{D}$. Let $\mathcal{L}_\mathbf{D}$ be the clustering of records in $\mathbf{D}$.* Incremental linkage *clusters records in $\mathbf{D} + \Delta\mathbf{D}$ based on $\mathcal{L}_\mathbf{D}$. We denote the incremental linkage method by f, and denote the results by $f(\mathbf{D}, \Delta\mathbf{D}, \mathcal{L}_\mathbf{D})$.* □

The goal for incremental linkage is two-fold. First, incremental linkage should be *much faster* than conducting batch linkage, especially when the number of operations in the increment is small; that

is, applying $f(\mathbf{D}, \Delta\mathbf{D}, \mathcal{L}_\mathbf{D})$ should be much faster than applying $F(\mathbf{D} + \Delta\mathbf{D})$ when $|\Delta\mathbf{D}| \ll |\mathbf{D}|$. Second, incremental linkage should obtain results of *similar quality* to batch linkage; that is, $f(\mathbf{D}, \Delta\mathbf{D}, \mathcal{L}_\mathbf{D}) \approx F(\mathbf{D} + \Delta\mathbf{D})$, where $\approx$ denotes clustering with similar precision and recall.

EXAMPLE 2.2. *Consider the motivating example. The original data set is $\mathbf{D_0} = \{r_1 - r_{10}\}$, and the linkage result $\mathcal{L}_{\mathbf{D_0}}$ is shown in Figure 1(b). As we have explained, the clustering is incorrect.*

Figure 2(a) shows 4 increments $\mathbf{\Delta D_1} - \mathbf{\Delta D_4}$, each containing one to two `Insert` *operations. We apply incremental linkage four times, one for each increment. The final result contains 6 clusters, as shown in Figure 2(b). Indeed, this is the correct result and as we show later, it is the result we would obtain when we conduct batch linkage on records $r_1 - r_{17}$.* □

Graph representation: Once we know the similarity between each pair of records, we can construct a *similarity graph* $G(V, E)$ for records in $\mathbf{D}$, where each node $v_r \in V$ represents a record $r \in \mathbf{D}$ and each edge $(v_r, v_{r'}) \in E$ with weight $\text{sim}(r, r')$ $(0 \leq \text{sim}(r, r') \leq 1)$ represents the similarity between records $r, r' \in \mathbf{D}$. We can simplify the graph by omitting an edge if the similarity is below a threshold. As an example, the similarity graph for $\mathbf{D} = \{r_1 - r_{10}\}$ is shown in Figure 1(b). The result of record linkage can be considered as clustering of the nodes in G; we denote the result as $\mathcal{L}_G$. We now consider how an update would change the graph.

- `Insert`: Inserting a record is equivalent to adding a node and edges to the node.
- `Delete`: Deleting a record is equivalent to removing a node and edges to the node.[1]
- `Change`: Changing a record is equivalent to removing existing edges and adding new edges to the corresponding node.

We denote the changes of an increment to a graph by ΔG, the result graph by $G + \Delta G$, and the result of incremental linkage also as $f(G, \Delta G, \mathcal{L}_G)$.

2.2 An end-to-end framework

Record linkage typically proceeds in three steps. First, it puts records into (multiple, possibly overlapping) blocks, such that records that share some commonality and may refer to the same real-world entity co-occur in at least one block. Recent blocking techniques handle the volume aspect of our problem. Second, for records in the same block, it computes pairwise similarity to construct the similarity graph G. Third, it conducts graph clustering, such that records that refer to the same real-world entity belong to the same cluster, and records that refer to different entities belong to different clusters. Thus, the approaches proposed in this work handle the velocity aspect of the record linkage problem. We now describe how we may conduct each step in an incremental fashion.

Blocking: To avoid exhaustive pairwise comparison, the blocking step builds an index for the records; each index entry forms a block [2]. The index key can be a word, a token, or a k-gram. In incremental blocking, we go over the records in the increment. For each inserted record, we index the new record. For each deleted record, we remove it from the index. For each changed record, we remove it from the previous entries and re-index it. We mark the inserted or deleted records in each index entry. Note that a changed record may be re-indexed to the same entry and we mark it as "re-inserted". In our motivating example, there is an entry for "Starbucks" and it contains all records in D_0. When we index the inserted records in $\Delta D_1 - \Delta D_4$, we also add them to that entry.

Similarity computation: This step essentially computes ΔG. We go over each index entry. In case the record is newly inserted, we include in ΔG the new node. For each inserted record in an entry, we compare it with the rest of the records in the entry. We include in ΔG the new edges.

For each deleted record in an entry, if the record is deleted in the increment, we include in ΔG the deleted node and its associated edges; otherwise, the record must be a changed record and we include in ΔG the deleted edges from the corresponding node.

For each re-inserted record, we recompute its similarity with the rest of the records in the entry. We include in ΔG the inserted edges, the deleted edges, and the edges with changed weights.

As in batch linkage, we avoid comparing the same pair of records multiple times if they co-occur in multiple index entries. At the end of this step, ΔG contains (1) inserted nodes with their associated edges; (2) deleted nodes with their associated edges; and (3) changed edges. In our motivating example and the "Starbucks" entry, we need to compute the similarity between each pair of inserted records, and between each inserted record and each previous record; ΔG contains the new nodes and their associated edges.

Graph clustering: We re-cluster the nodes according to the changes in the graph. This step is the most challenging step in incremental linkage and is the focus of the rest of our paper. Rather than applying clustering on the whole $G + \Delta G$, we wish to consider only a subgraph of $G + \Delta G$. To obtain similar results as batch linkage, our incremental clustering algorithm should be designed according to the batch clustering algorithm, which we review next.

2.3 Background for graph clustering

We review two classical clustering methods used in record linkage: *correlation clustering* [1] and *DB-index clustering* [6]. Both methods evaluate a clustering by an *objective function* and choose the clustering that optimizes the value of the objective function; in this way, we reward high *cohesion*, measuring the similarity or closeness of nodes in the same cluster, and penalize high *correlation*, measuring the similarity or closeness of nodes across clusters.

We focus on these two methods for three reasons. First, unlike the clustering methods that require *a priori* knowledge of the number of clusters, such as K-means clustering, these two methods can be applied when such knowledge does not exist, so are suitable for record linkage. Second, each of these two methods represents one of the two categories of graph-clustering methods [10]: correlation clustering represents the category that uses adjacency-based measures and DB-index clustering represents the category that uses distance-based measures. Third, as we have discussed previously, agglomerative clustering methods such as Swoosh [3] satisfy the general incremental condition so incremental linkage is straightforward. We now review each method in more detail.

Correlation clustering: The goal of correlation clustering is to find a partition of nodes in G that agrees as much as possible with the edge labels. To achieve this goal, we can either *maximize agreements* or *minimize disagreements* between the clustering and the labels. The two strategies are equivalent but differ from the approximation point of view. We focus on the latter strategy in the rest of the paper. For each pair of nodes in the same cluster, there is a *cohesion penalty* being the complement of the similarity; for each pair of nodes in different clusters, there is a *correlation penalty* being the similarity. We wish to minimize the sum of the penalties:

[1] Note that one may decide to use other semantics; for example, if a business record is deleted because of business closing, the node and edges may be kept to facilitate linkage in the future.

$$CC(\mathcal{L}_G) = \sum_{C\in\mathcal{L}_G, r,r'\in C} (1-\text{sim}(r,r')) + \sum_{C,C'\in\mathcal{L}_G, C\neq C', r\in C, r'\in C'} \text{sim}(r,r'). \quad (1)$$

A special case for correlation clustering is when we take binary similarities: the similarity between two records is either 0 (dissimilar) or 1 (similar). It has been proved that correlation clustering is NP-complete even for this special case and an algorithm called CAUTIOUS with complexity $O(|V|^2)$ can obtain a $9(\frac{1}{\delta^2}+1)$-approximation, where δ is a threshold applied in the algorithm [1]. It is also shown in [1] that for graphs with weighted edges, rounding the weights to 0 or 1 and applying CAUTIOUS can obtain a $(\frac{18}{\delta^2}+10)$-approximation.

EXAMPLE 2.3. *Consider clustering $\mathcal{L}_{\mathbf{D_0}}$ in Figure 1(b). The clustering has a cohesion penalty $.2+.2+.1+.1+1=1.6$ for C_1 and $.2+.2=.4$ for C_4. It also has a correlation penalty of .8 between C_4 and C_5. Thus, $CC(\mathcal{L}_{\mathbf{D_0}})=1.6+.4+.8=2.8$; it is the lowest penalty among all possible clusterings for $\mathbf{D_0}$.* □

DB-Index clustering: Davies-Bouldin index was originally defined for a Euclidean space [6]; applying it to record linkage requires some adjustment for the definition of *distance*. We adopt the definition in [9], described as follows.

For each cluster C, the *intra-cluster distance* is defined as the complement of average similarity between records in the cluster; that is, $D(C)=1-\text{Avg}_{r,r'\in C}\text{sim}(r,r')$. For each pair of distinct clusters C and C', the *inter-cluster distance* is defined as the complement of average similarity between records across the clusters; that is, $D(C,C')=1-\text{Avg}_{r\in C,r'\in C'}\text{sim}(r,r')$. The *separation measure* between C and C' is then defined as $M(C,C')=\frac{D(C)+D(C')+\alpha}{D(C,C')+\beta}$, where α and β are small positive numbers such that the denominator or numerator would affect the result even when the other is 0.[2] For each cluster C, we define its *separation measure* as $M(C)=\max_{C'\neq C}M(C,C')$. DB-index is defined as the average separation measure for all clusters and we wish to minimize it:

$$DB(\mathcal{L}_G)=\text{Avg}_{C\in\mathcal{L}_G}M(C). \quad (2)$$

Guo et al. [9] showed that DB-index clustering is also intractable and presented a hill-climbing algorithm with complexity $O(l|V|^4)$, where l is the number of iterations in hill climbing.

EXAMPLE 2.4. *Consider the clustering in Figure 1(b). The intra-cluster distance for C_1 is $1-Avg\{.8,.8,.9,.9,1,0\}=.27$; that for C_4 is .13; and that for the other clusters is 0. The inter-cluster distance between C_4 and C_5 is $1-Avg\{.8,0,0\}=.73$ and that between any other pair of clusters is 1. Taking C_4 as an example. If $\alpha=.01$ and $\beta=.001$, the separation measure for C_4 and C_5 is $\frac{.13+0+.01}{.73+.001}=.19$; that for C_4 and C_1 is $\frac{.13+.27+.01}{1+.001}=.41$; and that for C_4 and C_2 (or C_3) is $\frac{.13+0+.01}{1+.001}=.14$. Thus, we have $M(C_4)=\max\{.41,.14,.14,.19\}=.41$. The DB-index has value $Avg\{.41,.28,.28,.41,.28\}=.332$ and this clustering has the lowest DB-index among all possible clusterings.* □

Both correlation clustering and DB-index clustering aim at minimizing a penalty function. Ideally, we wish to design an incremental graph clustering algorithm that can still find an optimal clustering on $G+\Delta G$. We say such an algorithm is *optimal*.

[2]Consider a graph containing only two nodes with a similarity of 1. If we split the two nodes in clustering, the separation measure for the two singleton clusters is $\frac{\alpha}{\beta}$. We wish to set a high penalty for such a clustering, so α should be much larger than β.

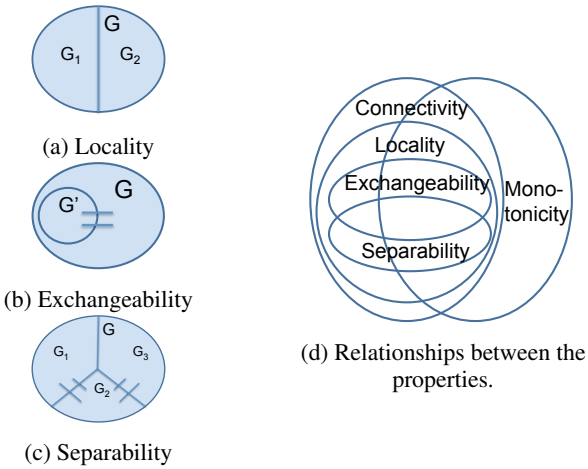


Figure 4: Linkage properties.

DEFINITION 2.5 (OPTIMAL INCREMENTAL LINKAGE). *Let $\mathcal{L}_G^{opt}$ be an optimal clustering on G. An incremental linkage method f is* optimal *if for every $G, \Delta G$, and $\mathcal{L}_G^{opt}$, result $f(G,\Delta G,\mathcal{L}_G^{opt})$ is an optimal clustering on $G+\Delta G$.* □

We focus on incremental graph clustering in the next two sections. Section 3 describes two algorithms that can be optimal for certain graph clustering approaches but would take exponential time. Section 4 describes a greedy algorithm that may not find the optimal clustering but takes only polynomial time.

3. OPTIMAL INCREMENTAL SOLUTION

In this section, we present two incremental linkage algorithms that are optimal for correlation clustering. However, they are not optimal for DB-index clustering, which lacks some desirable properties of clustering.

3.1 Desirable properties of linkage

Before we present our algorithms, we first describe several desirable properties for objective functions used in graph clustering. As we show later, these properties are critical for designing optimal incremental linkage methods. In the following definitions, we denote by O an objective function in graph clustering and assume without losing generality that we aim at minimizing the value of O. We denote by $\mathcal{L}_G^{O.opt}$ an optimal clustering of G according to O.

The first two definitions are basic properties asserting that each cluster should contain a connected subgraph. Between them, *locality* implies *connectivity*.

DEFINITION 3.1 (CONNECTIVITY). *Let $\mathcal{L}_G$ be a clustering of G and $\mathcal{L}'_G$ be a clustering obtained by putting two disconnected clusters in $\mathcal{L}_G$ into the same cluster. We say O satisfies* connectivity *if for every such $\mathcal{L}_G$ and $\mathcal{L}'_G$, $O(\mathcal{L}_G)<O(\mathcal{L}'_G)$.* □

DEFINITION 3.2 (LOCALITY). *Let G_1 and G_2 be a split of G such that there is no edge between G_1 and G_2 (Figure 4(a)). We say O satisfies* locality *if for every such G, G_1, and G_2, $\mathcal{L}_{G_1}^{O.opt}\cup\mathcal{L}_{G_2}^{O.opt}$ forms an optimal clustering for G under O.* □

EXAMPLE 3.3. *Consider the graph in Figure 5 and two clusterings. The first, $\mathcal{L}_G$, contains $n+1$ clusters: $C_0-C_n, n\geq 2$. Among them, C_0 contains two disconnected subgraphs C'_0 and C''_0, where C'_0 has m nodes. The second, $\mathcal{L}'_G$, contains $n+2$ clusters: C'_0, C''_0, C_1-C_n. If the objective function satisfies connectivity, $\mathcal{L}'_G$ should be a better clustering than $\mathcal{L}_G$.*

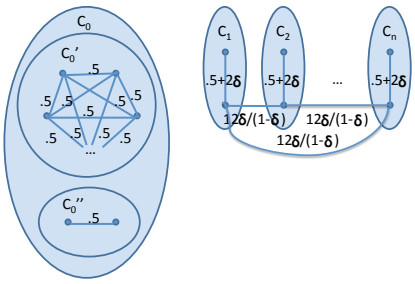


Figure 5: An example illustrating that DB-index violates the desired graph clustering properties.

Now consider the DB-index. According to Eq. (2), the intra-cluster distance of C_0 is $1 - \frac{.5*m(m-1)/2+.5}{(m+2)(m+1)/2} = .5 + \frac{2m}{(m+2)(m+1)}$. When m is large, the distance can be arbitrarily close to .5, and we denote it by $.5 + \delta$, where δ is a positive number close to 0. For $C_k, k \in [1, n]$, the intra-cluster distance is $1 - (.5 + 2\delta) = .5 - 2\delta$. For C_0 and C_k, the inter-cluster distance is 1. For C_k and $C_{k'}, k, k' \in [1, n]$, the inter-cluster distance is $1 - \frac{12\delta}{1-\delta}/4 = \frac{1-4\delta}{1-\delta}$. For simplicity, we assume $\alpha = \beta = 0$; we have the same conclusion when $\alpha > 0$ or $\beta > 0$. The separation measure for C_0 is $M(C_0) = \frac{.5+\delta+.5-2\delta}{1} = 1 - \delta$. The separation measure for C_k and C_0 and that for C_k and $C_{k'}$ are the same, $1 - \delta$, so $M(C_k) = 1 - \delta$. Thus, the DB-index for $\mathcal{L}_G$ is $1 - \delta$.

When we split C_0 into C_0' and C_0'' to obtain $\mathcal{L}_G'$, the intra-cluster distance for C_0' (or C_0'') is .5 and the separation measure is 1. The separation measure for C_k remains the same. Thus, the DB-index is $\frac{1+1+(1-\delta)n}{n+2} = 1 - \frac{n\delta}{n+2} > 1 - \delta$, indicating that $\mathcal{L}_G$ is better than $\mathcal{L}_G'$. Indeed, $\mathcal{L}_G$ is the optimal clustering for the graph in Figure 5, so DB-index violates connectivity and locality. □

The next two properties are stronger than locality, requiring that even in a connected graph, there should be subgraphs that are "independent" of the rest of the graph.

DEFINITION 3.4 (EXCHANGEABILITY). *Let $\bar{C} \subseteq \mathcal{L}_G^{opt}$ be a subset of clusters and $G' \subseteq G$ be the subgraph containing only nodes in $\bar{C}$ and edges between them (Figure 4(b)). We say O satisfies* exchangeability *if for every such G and $\bar{C}$, $\bar{C}$ is an optimal clustering for G' and replacing $\bar{C}$ with any other optimal clustering of G' obtains an optimal clustering for G under O.* □

DEFINITION 3.5 (SEPARABILITY). *Let G_1, G_2, G_3 be a partition of G such that (1) G_1 and G_3 are disconnected; (2) there exists an optimal clustering for $G_1 \cup G_2$ with no cluster across G_1 and G_2; and (3) there exists an optimal clustering for $G_2 \cup G_3$ with no cluster across G_2 and G_3 (Figure 4(c)). We say O satisfies* separability *if for every such G, G_1, G_2 and G_3, there exists an optimal clustering for G with no cluster across two or three of the subgraphs $G_1 - G_3$ under O.* □

EXAMPLE 3.6. *Continue with Figure 5 and DB-index. Recall that the optimal clustering contains clusters $C_0 - C_n$. Consider the subgraph G' with nodes in C_0. If exchangeability holds, C_0 should be an optimal clustering for G'; however, the optimal clustering for G' is actually $\{C_0', C_0''\}$. Thus, DB-index violates exchangeability.*

Now consider splitting G into $G_1 = \{C_0'\}, G_3 = \{C_0''\}, G_2 = \{C_1, \ldots, C_n\}$. Obviously, (1) G_1 and G_3 are disconnected; (2) an optimal clustering for $G_1 \cup G_2$ is $\{C_0', C_1, \ldots, C_n\}$; and (3) an optimal clustering for $G_2 \cup G_3$ is $\{C_0'', C_1, \ldots, C_n\}$. However, the optimal clustering for G contains cluster C_0, across G_1 and G_3. Thus, DB-index violates separability. □

Finally, we define a property that is independent of the aforementioned properties. It states that increasing similarity between nodes in the same cluster or decreasing similarity between nodes across clusters would not change clustering results.

DEFINITION 3.7 (MONOTONICITY). *Let $v_1, v_2 \in V$ be two nodes in the same cluster in $\mathcal{L}_G^{O.opt}$. Let G' be a graph obtained by increasing the weight of edge (v_1, v_2) in G. We say O satisfies* positive monotonicity *if for every such G and G', $\mathcal{L}_G^{O.opt}$ is also an optimal clustering of G'.*

Let $v_1, v_2 \in V$ be two nodes in different clusters in $\mathcal{L}_G^{O.opt}$. Let G' be a graph obtained by decreasing the weight of edge (v_1, v_2) in G. We say O satisfies negative monotonicity *if for every such G and G', $\mathcal{L}_G^{O.opt}$ is also an optimal clustering of G'.*

We say O satisfies monotonicity *if it satisfies both positive monotonicity and negative monotonicity.* □

Figure 4(d) shows the relationship between these five properties. We formally state them in the next theorem.

THEOREM 3.8. *For the relationships between the properties,*

- *locality implies connectivity;*
- *exchangeability implies locality;*
- *separability implies locality;*
- *there exists an objective function that satisfies exchangeability but not separability and vice versa;*
- *there exists an objective function that satisfies connectivity but not monotonicity and vice versa.* □

As we have shown in the examples, DB-index does not satisfy any of the five properties; nevertheless, we consider it in this paper because it represents distance-based clustering. In contrast, correlation clustering satisfies all properties.

THEOREM 3.9. *The objective function of Correlation clustering (Eq.(1)) satisfies connectivity, locality, monotonicity, exchangeability, and separability. DB-index does not satisfy any of them.* □

Comparison with [13]: Incremental linkage was briefly discussed for data updates in [13]. It defines a property for linkage methods that operate on clusters; we restate it as follows.

DEFINITION 3.10 (GENERAL INCREMENTAL). *We define F as a batch linkage algorithm whose input is the clustering of records. Let $\mathbf{S}(G)$ be the set of singleton clusters for each node in graph G. We say F is* general incremental *if for every subgraph $G' \subseteq G$, we have $F(\mathbf{S}(G \setminus G') \cup F(\mathbf{S}(G'))) = F(\mathbf{S}(G))$.* □

When a batch linkage algorithm F is general incremental, we can apply it directly on the clustering results for G and the singleton clusters for ΔG. In other words, we can define $f(G, \Delta G, \mathcal{L}_G) = F(\mathcal{L}_G \cup \mathbf{S}(\Delta G))$. However, both CAUTIOUS for correlation clustering and the batch algorithm for DB-index clustering [9] operate on individual nodes rather than on clusters, so they are not general incremental; [13] does not present any solution for them.

Our approach differs in two aspects. First, we define properties for the objective function O used in a clustering method rather than for the clustering algorithm F itself (*e.g.*, there can be many different algorithms aiming at minimizing the penalty for correlation clustering). Second, we show that even if F is not general incremental and we cannot directly apply the batch algorithm, we can still design incremental linkage algorithms that are typically much more efficient than applying batch linkage. We show that under our defined properties our proposed algorithms can be *optimal*; but even if these properties do not hold, we show empirically that our algorithms still generate high-quality linkage results.

We next describe two incremental clustering algorithms that are optimal under these aforementioned properties.

3.2 Connected component algorithm

Intuitively, when the clustering algorithm satisfies locality, it is safe to consider only the subgraph that is directly or indirectly connected to the changed nodes. We call this subgraph the *connected component* of the increment.

DEFINITION 3.11 (CONNECTED COMPONENT). *Let G be a similarity graph and ΔG be an increment on G. We define the* transitive closure of a node *as the connected subgraph in $G + \Delta G$ including the node. We define the* transitive closure of an edge *as the connected subgraph in $G + \Delta G$ including the edge and its two nodes. The* connected component of ΔG, *denoted by $T(\Delta G)$, contains the union of the transitive closures for each inserted, deleted, or changed node or edge.* □

In addition, when monotonicity holds, we can simplify the connected component by ignoring changes of increasing weights for intra-cluster edges and of decreasing weights for inter-cluster edges. Similarly, for a deleted node, we can ignore its associated edges to other clusters, as their weights essentially drop to 0. We call the resulting subgraph the *monotone connected component*.

DEFINITION 3.12 (MONOTONE CONNECTED COMPONENT). *Let G be a similarity graph and $\mathcal{L}_G^{opt}$ be the given optimal clustering on G. Let ΔG be an increment on G. The* monotone connected component of ΔG, *denoted by $\hat{T}(\Delta G)$, is defined as follows.*

- *For each inserted node $v \in \Delta G$, $\hat{T}(\Delta G)$ contains its transitive closure.*
- *For each deleted node $v \in \Delta G$, $\hat{T}(\Delta G)$ contains its cluster in $\mathcal{L}_G^{opt}$, but does not contain v and edges to v.*
- *For each edge $e \in \Delta G$ with increased weight, if e is across clusters in $\mathcal{L}_G^{opt}$, $\hat{T}(\Delta G)$ contains its transitive closure.*
- *For each edge $e \in \Delta G$ with decreased weight, if e is within a cluster in $\mathcal{L}_G^{opt}$, $\hat{T}(\Delta G)$ contains its transitive closure.* □

Given $G, \Delta G, \mathcal{L}_G^{opt}$, the connected component algorithm, CONNECTED, proceeds in three steps.

1. Find the connected component $T(\Delta G)$.
2. Find the optimal clustering on $T(\Delta G)$.
3. Construct the new clustering from $\mathcal{L}_G^{opt}$ by replacing the old clusters for $T(\Delta G)$ with the new optimal clusters.

Note that instead of using connected component, we can also use monotone connected component and we call this alternative MONOCONNECTED.

EXAMPLE 3.13. *Consider increment ΔD_4 in Figure 2(a). It inserts nodes r_{16}, r_{17}, and the associated edges. The transitive closure of r_{16} contains nodes $r_7 - r_{10}, r_{17}$ and similarly for r_{17} (see Figure 2(b)). So the connected component for ΔD_4 contains the subgraph with nodes $r_7 - r_{10}, r_{16} - r_{17}$. The optimal clustering under correlation clustering for this subgraph contains two clusters: C_4' and C_5'. Thus, we replace the old C_4 and C_5 with C_4' and C_5' to obtain a new clustering, which leads to the optimal clustering for the whole graph under correlation clustering.*

Now consider an increment ΔD_5 that removes node r_4. The transitive closure of r_4 contains nodes in C_1' (C_1'' contains only r_4 so is not included). Accordingly, CONNECTED *applies correlation clustering on C_1' and obtains the same clusters as before. However, the monotone connected component for r_4 is empty, so* MONOCONNECTED *can simply remove C_1'' to obtain the new clustering.* □

We next show the optimality and complexity of MONOCONNECTED.

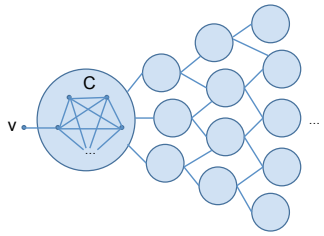


Figure 6: An instance where ITERATIVE can be much faster than MONOCONNECTED.

LEMMA 3.14. *Algorithm* CONNECTED *is optimal if and only if locality holds.* □

THEOREM 3.15 (OPTIMALITY OF MONOCONNECTED). *Algorithm* MONOCONNECTED *is optimal if and only if locality and monotonicity hold.* □

COROLLARY 3.16. MONOCONNECTED *is optimal for correlation clustering but not optimal for DB-index clustering.* □

PROPOSITION 3.17 (COMPLEXITY OF MONOCONNECTED). *Let $c(|G|)$ be the complexity of finding the optimal clustering on G. The complexity of* MONOCONNECTED *is $O(c(|G + \Delta G|))$.* □

When $G + \Delta G$ is a connected graph, $\hat{T}(\Delta G)$ can be the same as $G + \Delta G$ in the worst case; then, MONOCONNECTED needs to apply clustering on the whole graph, so it has the same complexity as batch linkage. However, when G is not well connected, $\hat{T}(\Delta G)$ could be much smaller than the full graph and MONOCONNECTED can be much faster. Also note that if finding an optimal clustering is intractable, c is exponential in the size of the graph and MONOCONNECTED takes exponential time.

3.3 Iterative algorithm

Although MONOCONNECTED requires examining only a subgraph, the subgraph can be large when the similarity graph is well connected. One opportunity for optimization is to consider the nodes that are only *closely* connected. The *iterative algorithm* first considers a subgraph with only clusters that are directly connected to the increment, which we call *directly connected component*, and expands the subgraph iteratively if the optimal clustering changes.

DEFINITION 3.18 (DIRECTLY CONNECTED COMPONENT). *Let G be a similarity graph and $\mathcal{L}_G^{opt}$ be the given optimal clustering on G. Let ΔG be an increment on G. The* directly connected component of ΔG, *denoted by $\bar{T}(\Delta G)$, is defined as follows.*

- *For each inserted node $v \in \Delta G$, $\bar{T}(\Delta G)$ contains v and its connected clusters in $\mathcal{L}_G^{opt}$.*
- *For each deleted node $v \in \Delta G$, $\bar{T}(\Delta G)$ contains its cluster in $\mathcal{L}_G^{opt}$, but does not contain v and edges to v.*
- *For each edge $e \in \Delta G$ with increased weight, if e is across clusters $C_1, C_2 \in \mathcal{L}_G^{opt}$, $\bar{T}(\Delta G)$ contains C_1 and C_2.*
- *For each edge $e \in \Delta G$ with decreased weight, if e is within a cluster $C \in \mathcal{L}_G^{opt}$, $\bar{T}(\Delta G)$ contains C.* □

EXAMPLE 3.19. *Consider ΔD_4 in Figure 2(a). The inserted node r_{16} is connected to C_4 and C_5, whereas r_{17} is connected to C_4. Thus, the directly connected component contains $r_7 - r_{10}, r_{16} - r_{17}$, the same as the monotone connected component. Now consider Figure 6 with the inserted node v. The directly connected component contains v and its neighbor cluster C, much smaller than the monotone connected component, the whole graph.* □

The iterative algorithm, ITERATIVE, starts with the directly connected component and expands it only when necessary. In particular, it proceeds in four steps.

1. Obtain the directly connected component of the increment, $\bar{T}(\Delta G)$, and put each of its connected subgraphs into queue $\mathbf{Q}$. The *previous clustering* stored for each subgraph follows $\mathcal{L}_G^{opt}$ and puts each inserted node into a singleton cluster.
2. For each subgraph $G' \in \mathbf{Q}$, dequeue it and find the optimal clustering. For each cluster that does not exist in the previous clustering, find its directly connected cluster, denoted by G''.
3. If G'' has never been added to $\mathbf{Q}$, go over $\mathbf{Q}$ for subgraphs that are connected or overlapping with G''. Remove them from $\mathbf{Q}$ and merge them with G''. Repeat this until there is no such subgraph in $\mathbf{Q}$. Add G'' to $\mathbf{Q}$.
4. Repeat Steps 2-3 until $\mathbf{Q}$ is empty.

For the example in Figure 6, ITERATIVE would start with the subgraph containing the inserted node v and its neighbor cluster C. Since v is connected to only one node in C, ITERATIVE would decide to keep the current clustering and so terminate without considering any other cluster; thus, it can be much faster than MONOCONNECTED. However, in some extreme cases ITERATIVE can iteratively expand to the whole monotone connected component; in such cases ITERATIVE can be slower than MONOCONNECTED and the number of iterations is bounded by the longest path between clusters in $\mathcal{L}_G^{opt}$ (the length is at most $|\mathcal{L}_G^{opt}| - 1$).

PROPOSITION 3.20 (COMPLEXITY OF ITERATIVE). *The complexity of* ITERATIVE *is* $O(|\mathcal{L}_G^{opt}| \cdot c(|G + \Delta G|)$. □

Finally, we show that ITERATIVE is guaranteed to be optimal if and only if the clustering method satisfies all the properties.

THEOREM 3.21 (OPTIMALITY OF ITERATIVE). *Algorithm* ITERATIVE *is optimal if and only if exchangeability, separability, and monotonicity hold.* □

COROLLARY 3.22. ITERATIVE *is optimal for correlation clustering but not optimal for DB-index clustering.* □

4. AN EFFICIENT SOLUTION

As we have shown, the connected component algorithm may require considering an unnecessarily big subgraph when the similarity graph is well-connected, while the iterative algorithm may require repeated efforts in examining quite a few subgraphs before convergence. In addition, as we have discussed, finding an optimal solution for correlation clustering or DB-index clustering is intractable. In this section, we describe a greedy solution, GREEDY, with two goals. First, the algorithm should take only polynomial time. Second, although the algorithm iteratively expands the subgraphs for examination as ITERATIVE does, clustering in each later round should be built upon the clustering of the previous round. Specifically, GREEDY differs from ITERATIVE in two ways. In ITERATIVE, the working queue $\mathbf{Q}$ stores *subgraphs* that consist of multiple directly connected clusters, and each iteration applies batch clustering on a subgraph. In GREEDY, the working queue, denoted by $\mathbf{Q^c}$, stores *clusters*, and for each cluster we examine whether we wish to adjust nodes between it and its neighbor clusters. In other words, ITERATIVE iterates at the coarse granularity of subgraphs with multiple clusters, whereas GREEDY iterates at the finer granularity of individual clusters.

In the rest of the section, we first describe the framework of the algorithm (Section 4.1), and then briefly discuss how to instantiate it for particular clustering methods (Section 4.2).

4.1 Greedy algorithm

In the greedy algorithm, each time we examine a cluster C from the working queue $\mathbf{Q^c}$ and consider three possible operations that we may apply to the cluster: *merging* C with some other cluster(s), *splitting* C to one or more clusters, and *moving* some of the nodes of C to another cluster or vice versa. We next describe the three operations in detail and then give the full algorithm.

Merge: Given a cluster $C \in \mathbf{Q^c}$, we consider whether merging it with other clusters would generate a better clustering (lower value for the objective function). To finish the exploration in polynomial time, we consider merging only pairs of clusters. The algorithm, MERGE, proceeds as follows.

1. For each neighbor cluster C' of C, evaluate whether merging C with C' generates a better clustering.
2. Upon finding a better clustering, (1) merge C with C', (2) add $C \cup C'$ to $\mathbf{Q^c}$, and (3) remove C' from $\mathbf{Q^c}$ if $C' \in \mathbf{Q^c}$.

EXAMPLE 4.1. *First, consider increment ΔD_1 in Figure 2(a) and correlation clustering. Cluster $C_6 = \{r_{11}\}$ is not connected to any node so we do not merge it with another cluster.*

Next, consider ΔD_2, which puts clusters $C_7 = \{r_{12}\}$ and $C_8 = \{r_{13}\}$ to the working queue $\mathbf{Q^c}$. We first merge C_7 with $C_2 = \{r_5\}$ (reducing the penalty from 4.2 to 3.4), then gradually merge also with C_8 and $C_3 = \{r_6\}$ (final penalty .8), obtaining C_2' in Figure 2(b). □

Split: Given a cluster $C \in \mathbf{Q^c}$, we consider whether splitting it into several clusters would generate a better clustering. To restrict the algorithm to polynomial time, we consider splitting into two clusters and we examine one node each time. The algorithm, SPLIT, proceeds as follows.

1. For each node $v \in C$, evaluate whether splitting v out generates a better clustering.
2. Upon finding such a node v, create a new cluster $C' = \{v\}$ and conduct steps 3-4.
3. For each remaining node $v' \in C$, evaluate whether moving v' to C' obtains a better clustering. If so, move v' to C' and repeat Step 3.
4. Add C and C' to $\mathbf{Q^c}$ if they are connected to other clusters.

EXAMPLE 4.2. *Consider increment ΔD_3 in Figure 2(a), which adds clusters $C_{11} = \{r_{14}\}$ and $C_{12} = \{r_{15}\}$ to $\mathbf{Q^c}$. Since they are closely connected with C_1, merging them into C_1 reduces the penalty under correlation clustering from 8.2 to 4. When we examine the new cluster $\{r_1 - r_4, r_{14}, r_{15}\}$, we find that splitting out r_4 reduces the penalty to 2.2. There is no more node to be moved out and we terminate with two clusters C_1' and C_1''.* □

Move: Given a cluster $C \in \mathbf{Q^c}$, we consider whether moving some of its nodes to other clusters or moving some nodes of other clusters into C would generate a better clustering. Again, we consider node moving between two clusters such that the algorithm finishes in polynomial time. The algorithm, MOVE, proceeds as follows.

1. For each neighbor cluster C' of C, do Steps 2-3.
2. For each node $v \in C$ that is connected to C' and for each $v \in C'$ connected to C, evaluate whether moving v to the other cluster generates a better clustering. Upon finding such a node v, move it to the other cluster.
3. Repeat Step 2 until there is no more node to move. Then, (1) add the two new clusters to $\mathbf{Q^c}$, and (2) dequeue C' if $C' \in \mathbf{Q^c}$.

Algorithm 1: Greedy($G(V, E), \Delta G, \mathcal{L}_G$)

Input : $G(V, E)$: Original similarity graph;
ΔG: Increment;
$\mathcal{L}_G$: clustering of the original graph
Output: New clustering in $\mathcal{L}_G$

1 $\mathbf{Q^c} \leftarrow \emptyset$;
2 $G' \leftarrow \bar{T}(\Delta G)$;
3 Put each cluster in G' to $\mathbf{Q^c}$;
4 **while** $\mathbf{Q^c} \neq \emptyset$ **do**
5 dequeue $C \in \mathbf{Q^c}$;
6 $changed \leftarrow false$;
7 *// operations return* true *if they change the clustering*
8 $changed \leftarrow$ MERGE($C, G + \Delta G, \mathcal{L}_G, \mathbf{Q^c}$);
9 **if** $\neg changed$ **then**
10 $changed \leftarrow$ SPLIT($C, G + \Delta G, \mathcal{L}_G, \mathbf{Q^c}$);
11 **if** $\neg changed$ **then**
12 $changed \leftarrow$ MOVE($C, G + \Delta G, \mathcal{L}_G, \mathbf{Q^c}$);
13 **return** $\mathcal{L}_G$;

Table 1: Working queue for Example 4.4.

Rnd	Removed	Added	$\mathbf{Q^c}$
1	-	$C = \{r_{16}\}, C' = \{r_{17}\}$	$\{C, C'\}$
2	C	$C'' = \{r_{10}, r_{16}\}$	$\{C', C''\}$
3	C'	$C''' = \{r_7 - r_9, r_{17}\}$	$\{C'', C'''\}$
4	C''	$C'_4 = \{r_7 - r_8, r_{17}\}$	$\{C'_4, C'_5\}$
		$C'_5 = \{r_9 - r_{10}, r_{18}\}$	
5	C'_4	-	$\{C'_5\}$
6	C'_5	-	$\emptyset$

EXAMPLE 4.3. *Consider C''_4 and C''_5 in Figure 3, where no merging or splitting can improve the clustering. However, moving r_9 from C''_4 to C''_5 reduces the penalty under correlation clustering from 2.4 to 2.2.* □

Full algorithm: We show the full algorithm GREEDY in Algorithm 1. Initially, it starts with the directly connected component (Ln.2) It then puts each cluster in $\bar{T}(\Delta G)$ (each inserted node is considered as a singleton cluster) into the working queue $\mathbf{Q^c}$ (Ln.3). For each cluster $C \in \mathbf{Q^c}$ in the queue, it checks the three operations for C in the order of merging (Ln.8), splitting (Ln. 10), and moving (Ln.12). This is because (1) moving is more expensive than merging or splitting, and (2) in our experiments we observed much more merging than splitting, and in turn than moving (Section 5). Once there are changes to any cluster, the algorithm puts the changed clusters back to the queue and considers the next cluster in $\mathbf{Q^c}$ (Lns.9, 11). This process continues until $\mathbf{Q^c}$ is empty (Ln.4).

EXAMPLE 4.4. *Consider increment ΔD_4 in Figure 2(a). Table 1 shows the trace of* GREEDY *under correlation clustering.*

Initially, we put $C = \{r_{16}\}$ and $C' = \{r_{17}\}$ into $\mathbf{Q^c}$. We first examine C and decide to merge it with $C_5 = \{r_{10}\}$; this puts $C'' = \{r_{10}, r_{16}\}$ to $\mathbf{Q^c}$. We then examine C' and decide to merge it with $C_4 = \{r_7 - r_9\}$; this puts $C''' = \{r_7 - r_9, r_{17}\}$ to $\mathbf{Q^c}$. After that we examine C'' and decide to move r_9 from C''' to C'', generating clusters C'_4 and C'_5 (Figure 2(b)); we remove C''' from $\mathbf{Q^c}$ and add C'_4 and C'_5. Examining C'_4 and C'_5 does not make any change, so we terminate. □

PROPOSITION 4.5 (COMPLEXITY OF GREEDY). *Let $g(|G|)$ be the time of evaluating the objective function on graph G. The complexity of* GREEDY *is $O(|G + \Delta G|^6 g(|G + \Delta G|))$.* □

Since typically we can evaluate an objective function in polynomial time, GREEDY takes only polynomial time. Although the complexity bound of GREEDY seems high, it is quite fast in practice for three reasons. First, each cluster is typically small, much smaller than the whole graph $G + \Delta G$. Second, for each examined cluster, the number of neighboring clusters is typically small, much smaller than $|G + \Delta G|$. Third, although in the worst case there can be $O(|G + \Delta G|^3)$ clusters that have ever appeared in $\mathbf{Q^c}$, in practice there are much fewer. Finally, we note that the approximation bound of the greedy algorithm remains an open problem, but we show by empirical study that it works well in practice.

4.2 Instantiation for correlation clustering

We can easily instantiate the greedy algorithm for correlation clustering or DB-index clustering. In particular, we show that under correlation clustering we can further simplify the algorithm for each operation (the framework remains the same).

Merge: According to the objective function Eq.(1) for correlation clustering, we merge two clusters C and C' when

$$\sum_{v \in C, v' \in C'} w(v, v') > \frac{|C| \cdot |C'|}{2}.$$

Split: We can simplify SPLIT as follows. First, in Step 1, instead of considering every node in C, we only consider the node that has the lowest *connectivity* within C, where connectivity is computed as the average similarity with other nodes in C. If the lowest connectivity is above .5, we can stop. Second, in Step 3, instead of considering every remaining node in C, we consider the node v with the lowest difference of $p_C(v) - p_{C'}(v)$, where $p_C(v)$ is the sum of the edge weights between v and each node in C, and $p_{C'}(v)$ is the sum of the edge weights between v and each node in C'. If $p_C(v) - p_{C'}(v) > \frac{|C|-|C'|-1}{2}$, we can stop. We note that this condition is the same as the previous condition if we consider $C' = \emptyset$. Instead of computing $p_C(v) - p_{C'}(v)$ from scratch each time, we can maintain it incrementally as we split out nodes.

Move: We can simplify MOVE in a similar way to SPLIT. In Step 2, instead of considering every node in $C \cup C'$, we choose the node $v \in C$ with the lowest $p_C(v) - p_{C'}(v)$ and do the moving if $p_C(v) - p_{C'}(v) \leq \frac{|C|-|C'|-1}{2}$; otherwise, we choose the node $v \in C'$ with the lowest $p_{C'}(v) - p_C(v)$ and do the moving if $p_{C'}(v) - p_C(v) < \frac{|C'|-|C|-1}{2}$.

These simplifications can reduce the complexity of the algorithm, called GREEDYCORR, shown as follows.

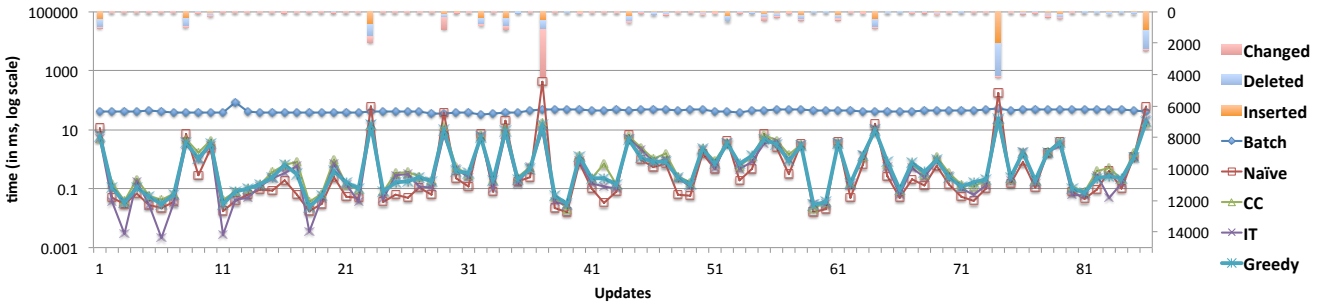
PROPOSITION 4.6 (COMPLEXITY OF GREEDYCORR). *The complexity of* GREEDYCORR *is $O(|G + \Delta G|^6)$.* □

5. EXPERIMENTAL EVALUATION

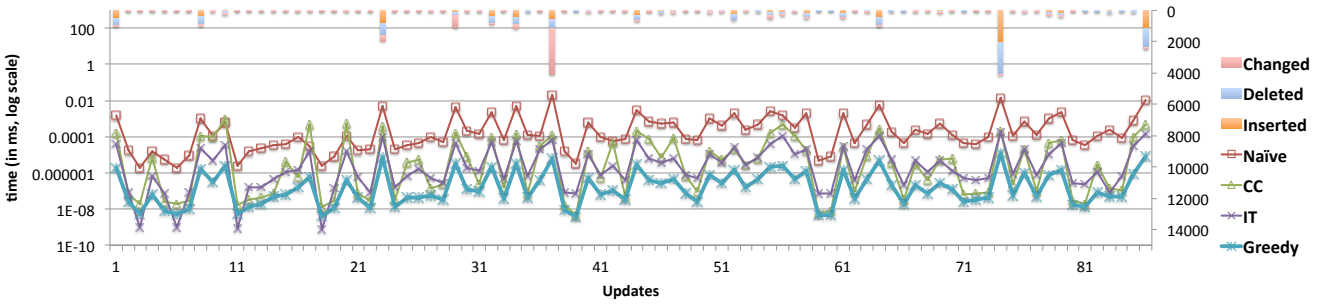
We present experimental results on two real-world datasets and a synthetic dataset. The results show that our incremental algorithms significantly improve over batch linkage on efficiency without sacrificing linkage quality, and significantly improve over naive incremental linkage algorithms on linkage quality.

5.1 Experiment setup

Datasets: We experimented with three datasets. The first dataset, *Biz*, contains 87 snapshots of business records in the San Francisco area; we took the first snapshot as the original dataset, and computed an increment for each later snapshot. Every snapshot of this dataset contains approx. 5K records with slight variations depending on the applied update. The increments contain on average 120 `Inserts`, 118 `Deletes`, and 59 `Changes`, and the



(a) Correlation clustering on *Biz*.



(b) DB-index clustering on *Biz*.

Figure 7: Comparison of various algorithms under CONT on *Biz*.

Table 2: Statistics of real-world datasets according to CAUTIOUS .

Statistics		*Biz*	*Cora*
Node	Number	4892	1916
	Avg #neighbors	3.05	41.8
	Max #neighbors	26	106
Cluster	Number	2054	575
	Avg #nodes	2.38	3.3
	Avg #neighbors	.7	27.8
	Max #neighbors	10	92
Subgraph	Number	1624	88
	Avg #nodes	3.01	21.7
	Max #nodes	29	18

maximum number of operations in an increment is 4120. The top part of Figure 7(a) (with the Y-axis on the right side of the figure) shows a break down of the updates for each increment. We indexed the records on 3-grams for blocking. We then applied the *Monge-Elkan* [4] string similarity for pairwise similarity computation and ignored edges with a similarity below .7. The similarity graph is fairly sparse as shown for the first snapshot of *Biz* in Table 2.

The second real-world dataset, *Cora*[3] that we examine has been widely used for record linkage and contains 1916 publication records. On this dataset, we have a single snapshot, we indexed the records on words for blocking and, following [7], we applied the weighted Jaccard measure with a threshold of .9 for similarity computation. As *Cora* is not a naturally incremental dataset, we generated a range of possible increments as follows: in the first increment we randomly remove 1 record; in the i-th increment we add back the records removed in the $(i-1)$-th increment and randomly remove 2^{i-1} records; in the last (*i.e.*, 11-th) increment, we only add back the previously removed (1024) records.

Our synthetic dataset uses the *Febrl* data generator.[4] We vary the generation parameters which we will further explain in Section 5.4.

Implementations: To determine the effectiveness of our incremental approaches, we implemented the following algorithms:

- BATCH applies CAUTIOUS [1] for correlation clustering and the hill climbing algorithm in [9] for DB-index clustering.
- NAIVE, the baseline incremental algorithm, compares each inserted record with existing clusters, then either adds it into an existing cluster or creates a new cluster for it (Section 1).
- CC applies CONNECTED (Section 3.2).
- IT applies ITERATIVE (Section 3.3).
- GREEDY applies GREEDY (Section 4).

Our implementation has two variations: in RESET the starting point for each increment is reset to the batch linkage results from the previous increment; in CONT the starting point is the incremental linkage results from the previous increment. In practice, we are likely to use CONT for updates and periodically apply batch linkage. In CAUTIOUS, we used parameter $\delta = .1$ [1]; in DB-INDEX, we set $\alpha = .2$ and $\beta = .1$. We implemented the algorithms in Java, and experimented on a Linux machine with eight Intel Xeon L5520 cores (2.26GHz, cache 24MB).

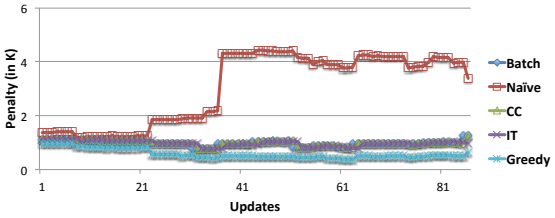
Measures: We measure efficiency and quality of our algorithms. For efficiency, we repeated the experiments 100 times and reported the average execution time. We focused on clustering and only reported clustering time; note however that if we count also blocking and pairwise similarity computation, incremental linkage would have even higher benefit over batch linkage. For quality, we report (1) the penalty (i.e., cut inter-cluster and missing intra-cluster edges) and (2) the *F-measure* if we have the gold standard. Here, *precision* measures among the pairs of records that are clustered together, how many are correct; *recall* measures among the pairs of records that refer to the same real-world entity, how many are clustered together; and the *F-measure* is computed as $\frac{2 \cdot precision \cdot recall}{precision + recall}$.

Objective: The goal of these experiments is three-fold. First, we want to establish incremental record linkage as desirable in a dynamic environment because of performance improvement and quality consistency. Second, we will show that iterative incremental approaches can reduce the linkage space drastically and decrease execution time even further. Last, we will identify the tradeoffs between the three incremental algorithms in this paper.

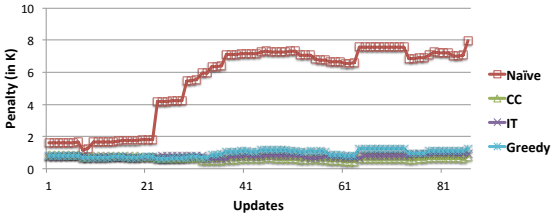
[3] *http://secondstring.sourceforge.net/.*
[4] *http://sourceforge.net/projects/febrl/.*

Table 3: Comparison of various algorithms on *Biz*. Highest performance is highlighted in bold. Penalty values are averaged. Improvement is calculated for NAIVE w.r.t. BATCH and for the other methods w.r.t. NAIVE.

Method			Time (s)	Impro.	Penalty
Corr Clust.		BATCH	3.7	-	988
	CONT	NAIVE	.86	76.7%	3037
		CC	.18	78.7%	988
		IT	0.16	81.4%	981
		GREEDY	**0.14**	**84.1%**	**592**
	RESET	NAIVE	0.79	79.7%	1072
		CC	0.20	74.2%	987
		IT	**0.17**	**77.7%**	987
		GREEDY	0.20	74.3%	**922**
DB-Index	CONT	NAIVE	997	99.9%	5426
		CC	57.1	94.3%	**651**
		IT	14.4	98.6%	783
		GREEDY	**.79**	**99.9%**	941



(a) Correlation clustering on *Biz*.



(b) DB-index clustering on *Biz*.

Figure 8: Penalty for CONT on *Biz*.

5.2 Experiments on *Biz*

Overview: Table 3 gives a summary of the quality and performance of our five implemented methods; Figure 7 (with the Y-axis on the left side of the figure) shows the execution time for these algorithms under CONT; Figure 8 shows the corresponding penalty. We observe that all incremental linkage algorithms significantly improved over BATCH on efficiency: for correlation clustering the slowest one reduced the execution time by 76.7% while the fastest one by 95.7%; for DB-index the slowest one reduced execution time by three orders of magnitude while the fastest one by nearly five orders of magnitude as BATCH takes 3.8 hours on average per snapshot. Among the incremental algorithms, the iterative algorithms (IT and GREEDY) outperform both NAIVE and CC in execution time and achieve comparable or better result quality.

Observations for NAIVE: The naive algorithm has competitive performance and output quality for small updates. As the performance of this algorithm is quadratic in the size of updates, it decreases with an increasing update size as shown in Figure 7 where large updates trigger execution times that are higher than those for BATCH. Aggregated over all increments, NAIVE still improves execution time for correlation clustering by 76.7% and 99.9% for DB-Index. Nevertheless, NAIVE is the slowest of our implemented incremental approaches. NAIVE is also the worst incremental approach in terms of quality which shows that a merge-only strategy does not take the changing graph structure into consideration appropriately. It has an average penalty of 3037 for correlation clustering and 5426 for DB-Index which is three times, resp. six times, worse than any other incremental approach.

Table 4: Comparison of CC, IT, and GREEDY for varying update sizes and *Biz* dataset (CONT).

Method			Performance Impro.		
			Update Size		
	Comparison		small	med.	large
Corr. Clust.	IT	CC	**94.1%**	**86.4%**	**53.9%**
	GREEDY	CC	**92.2%**	**100%**	**92.3%**
		IT	37.4%	**72.7%**	**69.2%**
DB-Index	IT	CC	**96.1%**	**95.5%**	**100%**
	GREEDY	CC	**100%**	**100%**	**100%**
		IT	31.4%	**72.7%**	**92.3%**

Table 5: Details on *Biz* (Correlation Clutering, CONT).

	CC	IT	GREEDY
time (ms)	189	167	171
#Iterations	1	2.0	2.5
#Nodes	480	293	474
#Edges	2003	1482	1968
#Total-nodes	-	329	1237
#Total-edges	-	1817	11204
#Examined-Merge	-	-	115
#Real-Merge	-	-	88
#Examined-Split	-	-	44
#Real-Split	-	-	1
#Examined-Move	-	-	7
#Real-Move	-	-	0

Observations for CC: In contrast to NAIVE, CC achieves basically the same result quality as BATCH. Minor variations (for example an average penalty for RESET of 987 instead of 988) may occur as CAUTIOUS is an approximation and is thus not guaranteed to make the optimal decision. CC improves the execution time of NAIVE by at least 74.2%, and we observe that it is especially effective for large increments in correlation clustering. Table 4 shows an overview of the performance of CC in comparison to our iterative approaches. A small update contains at most 50 updated records, a large update contains at least 500 updated records, all other updates are medium-sized. Our incremental dataset then consists of 51 small, 22 medium, and 13 large updates. The table shows how often either IT or GREEDY is better compared to CC. We observe that CC has worse performance than either GREEDY or IT in at least 92.2% (96.1% for DB-index) of the small and 86.4% (95.5% for DB-index) of the medium-sized updates. In contrast, CC outperforms IT in 46.1% of the large updates. More specifically, CC excels in those increments where the graph significantly changes, i.e. where a lot of nodes and edges are touched redundantly by the iterative approaches. An overview of which approach touches how many nodes and edges is shown in Table 5. The number of total nodes and edges describes the average number of nodes and edges that have been iterated over every increment where one node can be counted multiple times if, e.g., it is checked in both, a merge and split operation. As shown here, IT has lower values for both touched nodes and edges than CC which also explains the lower execution time. GREEDY on the other hand has higher values here, because it evaluates more options with its three operators. These operations are nevertheless efficient especially due to our selection criteria for *split* and *move*. As a result, the execution time of GREEDY is lower than CC for all DB-index experiments and at least better in 92.2% of the updates for correlation clustering.

Observations for IT and GREEDY: Both, GREEDY and IT, have similar execution times for correlation clustering but for DB-index clustering GREEDY is 94.5% faster than IT. This difference can be explained by the applied objective function: GREEDY does not use the hillclimbing algorithm but rather uses its three operators to determine a candidate clustering which reduces its execution time sig-

nificantly. In general, we observe that IT outperforms GREEDY for small updates on correlation clustering where it is faster in 58.6% of the updates. GREEDY inherently has an overhead for clusters that do not change because it checks for all three operations. In fact, we observe that on average GREEDY attempts 115 merges, 44 splits, and 7 moves per iteration. Checking for all these operations is obviously more costly than IT if the modified cluster is not well-connected or will not change its clustering.

5.3 Experiments on *Cora*

Overview: Table 6 and Figure 9 show the execution time and quality of our proposed methods using *Jaccard* similarity and correlation clustering on *Cora*. Compared to *Biz*, the similarity graph for *Cora* is much denser, as shown in Table 2.

Observations for NAIVE: On small increments, NAIVE was much faster than the other methods while having similar linkage quality. However, large updates trigger more mistakes made by NAIVE. The reason is the merge-only policy of NAIVE which does not allow to reconsider more beneficial clustering alternatives that may be possible if records are split or moved first.

Observations for CAUTIOUS algorithms: Because of the high connectivity of this dataset, CC finished in nearly the same time as BATCH starting from the fifth increment; indeed, in this round CC touched 64% of the nodes in the graph. Similarly, IT was even slower than CC after the fifth increment; this is because IT ended up examining nearly the same subgraph as CC but did additional work in early iterations on smaller subgraphs.

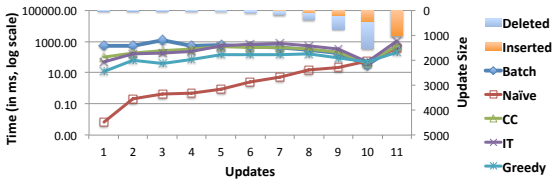
Observations for GREEDY: GREEDY is much faster than IT on this dataset and reduced the execution time by 66.8% over CC and by 75.8% over IT. Recall that we are using CAUTIOUS as implementation for both CC and IT. This algorithm decides whether to combine records into a cluster based on set logic which obviously gets more expensive the more connected the graph is because more candidates need evaluation. In contrast, *merge* operations as suggested for GREEDY are structured much more simply: the penalty of two clusters separated or combined is computed following the objective function and we then choose the better solution. An efficient *merge* operator thus decreases execution time effectively, making up for the more time-costly *split* and *move* operations.

5.4 Experiments on *Febrl*

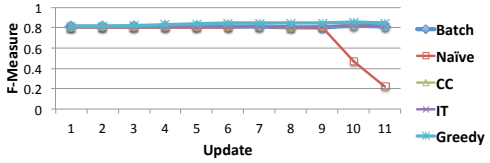
Data generation: We generate synthetic data using the *Febrl* dataset generator. It allows us to specify how many original and duplicate records we generate as well as the distribution of noise within the generated dataset. For our experiments we vary the following parameters: the number of duplicates per original record d; the number of modifications within one attribute and how many attributes are modified m (we use the same value for both as we want to show the behavior of our approaches if the noise level increases, separating them is less relevant); the number of inserted and deleted nodes within one increment n_{ins} and n_{del}; and finally θ, the similarity threshold applied in generating the similarity graph. For comparability, we generate 10K original records and 10k duplicate records as the original dataset given d and m. Note that θ modifies the internal similarity graph but not the underlying dataset. We used the blocking generated by *Febrl*, which guarantees that duplicates are in the same block. To determine the pairwise similarity between records, we use Jaro Winkler similarity, as suggested in [7] for *Febrl* data. We apply correlation clustering as the objective function in all reported experiments. Like *Cora*, *Febrl* is not a naturally incremental dataset. As a result, we need to simulate the increments. Every experiment that we run has 10 increments, by default we randomly selected 10K records as the initial dataset; we

Table 6: Algorithm comparison for *Cora*. F-Measure is averaged.

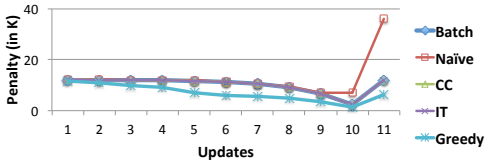
Method		Time (s)	F-Measure	Penalty
	BATCH	5.24	.811	10074
CONT	NAIVE	**.47**	.722	12832
	CC	3.3	.81	10163
	IT	4.5	.81	10170
	GREEDY	1.09	**.837**	**6847**
RESET	NAIVE	**.318**	.754	11853
	CC	3.04	.81	10113
	IT	4.31	.811	10119
	GREEDY	1.31	**.838**	**6945**



(a) Time



(b) F-Measure



(c) Penalty

Figure 9: Experimental results on *Cora*.

then randomly selected n_{ins} =1K (i.e., 5%) records from the remaining records for insertion and n_{del} =.5K (i.e., 2.5%) existing records for deletion per increment. We use $d = 9$, $m = 3$, and $\theta = .8$ as default values for the remaining parameters.

Results: We make four observations when varying these parameters: First, varying θ confirms the observations made for *Biz* and *Cora*. If the graph is denser, i.e. the average number of neighbors increases from 0 ($\theta = 1$) to 25.01 for ($\theta = .7$), GREEDY is more resilient in terms of quality and requires less execution time than the CAUTIOUS approaches (IT and CC) for denser environments (Figures 10 and 11). We also observe (not shown in the graph) that GREEDY improves the result quality with each increment because it touches more nodes. Second, increasing d from 1 to 9 monotonically increases the average execution time by a factor of 1.1 (NAIVE), 1.7 (CC), 4.6 (IT), and 1.4 (GREEDY). The quality for all approaches increases from $d = 1$ at .61 (NAIVE), .59 (IT and CC), and .68 (GREEDY) to $d = 9$ with .82 (NAIVE), .87 (IT and CC), and .9 (GREEDY). The increase can be attributed to the fact that more duplicates mean a better chance of having high cohesion within an entity. Third, an increase of m results in a similar performance per method as shown in Figure 10 for $\theta = .8$. At the same time, the change causes a decrease in F-measure for all approaches. More specifically, the quality of NAIVE, CC, and IT decreases by 5% increasing m from 1 to 5 while the result quality of GREEDY is decreased by 1%. Last, we note that delete-only workloads are more efficient to process for all approaches except for CC. It takes the whole connected component as input which makes it indifferent to workload characteristics while IT, GREEDY, and NAIVE a) evaluate where the inserted or deleted record fits in and then b) how the clustering changes because of that. For deleted records, the

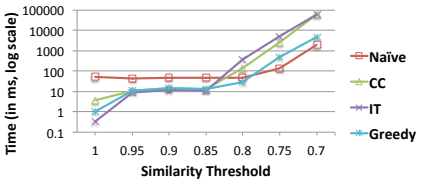


Figure 10: Time, varying θ (*Febrl*)

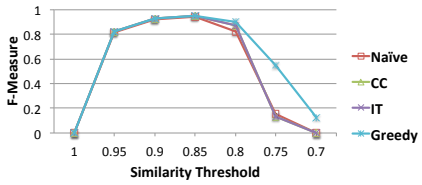


Figure 11: F-Measure, varying θ (*Febrl*)

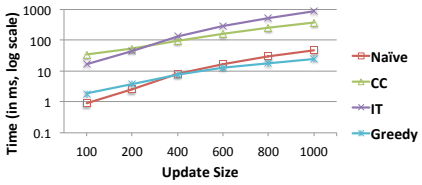


Figure 12: Time, varying n_{ins} (*Febrl*)

first part is obviously less costly as the position of a record in the clustering is already known. Specifically, we observe an increase by factor 60 for NAIVE, 8 for IT, and 2 for GREEDY when comparing a delete-only to an insert-only workload (n_{ins} =1K and n_{del} =1K). The varying factors are directly correlated to the way that deletes are handled for each approach: while GREEDY creates the clustering with costly *split* and *move* operations, IT initiates a cluster check on a smaller, less connected cluster, and NAIVE simply removes the record from the dataset. If we focus on an insert-only workload, we can show how the algorithms scale differently in noisy environments (Figure 12): While the performance of NAIVE is 52% better than GREEDY for $n_{ins} = 100$, GREEDY is 47% better than NAIVE for $n_{ins} = 1000$. At the same time, GREEDY maintains a higher F-measure (.9 compared to .82 of NAIVE). We can observe in this figure that a high connectivity of the graph is highly correlated with the performance of CC and IT: As both approaches (iteratively) explore at least 64.4% of the records in the graph for $n_{ins} = 1000$, their performance is clearly worse than GREEDY which only evaluates 13% of the graph on average. These observations are confirmed by varying the threshold θ thus increasing the level of noise as shown in Figure 10.

6. RELATED WORK

Record linkage has been extensively studied in the literature (surveyed in [8]); however, most of the research focuses on batch linkage rather than performing linkage in an incremental fashion to improve the efficiency. To the best of our knowledge, incremental linkage has been studied only in [12, 13]; however, they focused on evolving matching rules and discussed evolving data only very briefly. We present an end-to-end solution for incremental record linkage and we compared it to their work in detail in Section 3.1.

Another body of work close to ours is incremental graph clustering. Mathieu et al. [11] studied incremental correlation clustering for the case where (1) one vertex is added each time, and (2) already identified clusters need to be preserved. Their algorithm is analogous to the naive algorithm in our experiments. Charikar et al. [5] studied incremental clustering when the number of clusters is known *a priori*. Both papers focused on theoretical analysis and neither reported experimental results. Our algorithm is different in three aspects. First, we consider updates including not only inserting a record, but also deleting or changing an existing record. Second, we allow merging previously formed clusters, splitting them, or moving nodes between them, such that we can leverage new evidence from updates for fixing previous clustering errors. Third, we do not require any prior knowledge of the number of clusters, which is not realistic for the record linkage context. We show by extensive experiments that our algorithms achieve high efficiency without sacrificing the quality of linkage results.

7. CONCLUSION

This paper describes a set of algorithms that conduct record linkage in an incremental fashion when updates of the data arrive. Our algorithms not only allow merging records in the updates with existing clusters, each representing records that refer to the same entity, but also allow leveraging new evidence to modify existing clusters. We have shown on three real-world and synthetic data sets that they can handle a variety of increments: CC is a good alternative to BATCH even for large increments as it solely modifies those components that are connected to the increment. Our iterative approaches on the other hand are best applied for changes that affect a small portion of the connected component, i.e. when the update has local rather than global impact. Here, we have shown that both IT and GREEDY have well-defined use cases: as IT applies the original objective function, it will obtain similar quality to the respective batch algorithm while its performance is dependent on the implementation of the same. GREEDY has a predictable overhead which becomes less relevant, the more time-consuming the implementation of the original objective function as we have shown for DB-INDEX. Additionally, we show in our experiments that GREEDY is robust in noisy environments. Future work includes studying the problem on Web-scale data sets and for incremental linkage involving entity mentions in unstructured texts.

8. REFERENCES

[1] N. Bansal, A. Blum, and S. Chawla. Correlation clustering. In *Machine Learning*, pages 238–247, 2002.

[2] R. Baxter, P. Christen, and T. Churches. A comparison of fast blocking methods for record linkage. In *ACM SIGKDD Workshop on Data Cleaning, Record Linkage, and Object Consolidation*, 2003.

[3] 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.

[4] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *SIGKDD*, pages 39–48, 2003.

[5] M. Charikar, C. Chekuri, T. Feder, and R. Motwani. Incremental clustering and dynamic information retrieval. *SIAM J. Comput.*, 33(6):1417–1440, 2004.

[6] D. L. Davies and D. W. Bouldin. A Cluster Separation Measure. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):224–227, Apr. 1979.

[7] U. Draisbach and F. Naumann. On choosing thresholds for duplicate detection. In *Proceedings of the 18th International Conference on Information Quality (ICIQ)*, 2013.

[8] L. Getoor and A. Machanavajjhala. Entity resolution: Theory, practice, & open challenges. *PVLDB*, 5(12):2018–2019, 2012.

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

[10] O. Hassanzadeh, F. Chiang, R. J. Miller, and H. C. Lee. Framework for evaluating clustering algorithms in duplicate detection. *PVLDB*, 2(1):1282–1293, 2009.

[11] C. Mathieu, O. Sankur, and W. Schudy. Online correlation clustering. In *STACS*, pages 573–584, 2010.

[12] S. Whang and H. Garcia-Molina. Entity resolution with evolving rules. *PVLDB*, 3(1):1326–1337, 2010.

[13] S. E. Whang and H. Garcia-Molina. Incremental entity resolution on rules and data. *VLDB J.*, 23(1):77–102, 2014.