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 finds records that refer to the same real-world entity and is often a crucial step in data cleaning and data integration. The rapid growth of data in the big data era raises the challenges that applying record linkage on the huge *volume* of data often takes a long time, and the high *velocity* of data quickly makes previous linkage results obsolete. To address these challenges, this paper presents a set of incremental linkage algorithms. These 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 the new evidence from the updates to fix previous linkage errors. Experimental results on two real-world data sets show that our algorithms can significantly reduce linkage time without sacrificing linkage quality.

1. INTRODUCTION

Record linkage finds records (*i.e.*, database tuples) that refer to the same real-world entity (*e.g.*, businesses, persons) and is often a crucial step in data cleaning and data integration (surveyed in [5, 8]). It 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. Second, for records in the same block, it computes pairwise similarity. Third, it clusters the records based on pairwise similarity, 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.

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 linkage scheme, 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 existing 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)* address *excluding house number, (3) house number in* address*, (4)* city*, and (5)* phone*; the similarity is* 1 *if all of the 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 italic); 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).*

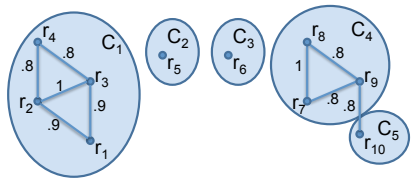
Now consider four updates $\Delta D_1 - \Delta D_4$ in Figure 2(a); they together insert records $r_{11} - r_{17}$; Figure 3 shows the updated similarity graph and the results of the aforementioned naive approach. This result (1) contains a separate cluster for r_{11} as it is different from any existing record, (2) merges $r_{12} - r_{13}$ to C_2 as they are more similar to r_5 than to r_6, (3) merges $r_{14} - r_{15}$ to C_1, (4) merges r_{16} to C_5 as it is not similar to $r_7 - r_8$ in C_4, and (5) merges r_{17} to C_4.

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 both similar to r_5 and r_6, they fill in the missing information for r_6 and provide extra evidence to merge r_5 and r_6. Second, because $r_{14} - r_{15}$ are both similar to $r_1 - r_3$ but quite different from r_4, they dilute the similarity of r_4 to the rest of the cluster and suggest 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 [10, 11], where the main focus is incremental linkage for evolving matching rules. In [11] the authors briefly discussed evolving data and identified a condition under which we can apply the same linkage algorithm on previous clustering results and the singleton clusters for newly inserted nodes, while obtaining the same results. This condition requires the linkage algorithm to be *general incremental*;

	BizID	ID	name	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)

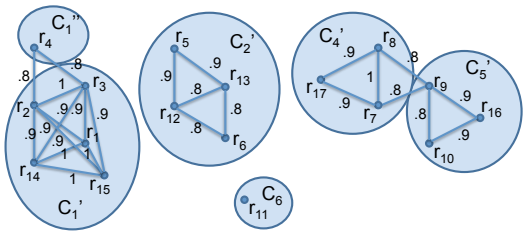


(b)

Figure 1: Original business listings and record linkage results.

	BizID	ID	name	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.

that is, for any arbitrary subset of records, first applying the linkage algorithm on the subset and then on the resulting clustering and the rest of the nodes obtain the same results. However, this condition is rather demanding: first, a lot of clustering algorithms, such as the aforementioned naive approach, do not satisfy this condition; second, many clustering algorithms, such as correlation clustering (we shall explain it soon), operate on records rather than subsets of records. 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 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, existing records are deleted or changed (*i.e.*, values are modified). We note that among the three steps of record linkage, performing blocking and pairwise similarity computation in an incremental fashion is fairly straight-forward when the previous results are available. Thus, we focus on the third step, clustering; that is, reclustering the existing records and the updated records based on the previous clustering results. In particular, we make the following three contributions.

- First, we propose two algorithms that apply clustering on a subset of the records rather than all records. We can prove their optimality if the clustering criteria satisfy a set of properties that are weaker than being general incremental.
- Second, we 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.
- Third, 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.

The rest of the paper is organized as follows. Section 2 formally defines the problem and reviews clustering algorithms used for batch 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.

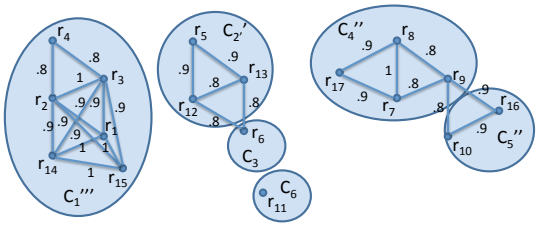


Figure 3: Results of a baseline incremental linkage algorithm.

2. PROBLEM STATEMENT

This section formally defines the problem of incremental record linkage and reviews techniques for batch record linkage.

2.1 Problem definition

Given a set of records, blocking functions, and a pairwise similarity function, record linkage is essentially a clustering problem, where each cluster should 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, $F(\mathbf{D}) = \mathcal{L}_\mathbf{D}$.

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 for an existing record. Note that `Change` can be achieved by first removing the old record and than inserting the new record; however, as we show later, considering `Change` directly can be more efficient. We call the 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 the 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 $\mathbf{D}$ (an identical record with the

same record ID would be removed from the increment). We can 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 clustering on records $r_1 - r_{17}$.* □

Graph representation: We 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). Record linkage can be considered as *graph clustering* and we denote the result also 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 node.

In the rest of the paper, we focus on adding a node, removing a node, and changing edges to a node as update operations. 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 Background

To obtain similar quality of linkage, we shall design the incremental linkage method f according to the batch linkage method F. We next review two classical methods for record linkage: *correlation clustering* [1] and *DB-index clustering* [4]. 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.

[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.

We focus on these two methods for three reasons. First, unlike some agglomerative clustering methods such as Swoosh [2], they allow splitting of previously formed clusters as we observe more records and collect more evidence; indeed, many agglomerative clustering algorithms are general incremental, so according to [11] we can simply apply the algorithm on the previous clustering results and the increments. Second, 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. Third, each of these two methods represents one of the two categories of graph-clustering methods [7]: correlation clustering represents the category that uses adjacency-based measures and DB-index clustering represents the category that uses distance-based measures. 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* between the clustering and the labels or *minimize disagreements*. 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:

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

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 is 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 [4]; applying it to record linkage requires some adjustment for the definition of *distance*. We adopt the definition in [6], 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[2] such that the denominator or numerator would affect the result even when the other is 0. For each cluster C, we define its *separation*

[2]In our experiments we set $\alpha = .05$ and $\beta = .001$ such that splitting a complete subgraph with edges of weight 1 would have a high penalty.

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. [6] 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.* □

3. OPTIMAL INCREMENTAL SOLUTION

Both correlation clustering and DB-index clustering aim at minimizing a penalty function. Ideally, we wish to design an incremental linkage algorithm that is guaranteed to find an optimal clustering on the update result. We say such an algorithm is *optimal*.

DEFINITION 3.1 (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 $G + \Delta G$.* □

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 basic desirable properties for clustering.

3.1 Desirable properties of linkage

Before we present our algorithm, we first describe several desirable properties for a graph clustering method. As we show later, these properties are critical for designing optimal incremental linkage methods.

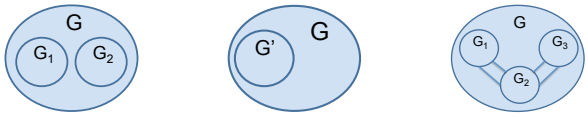
DEFINITION 3.2 (LINKAGE PROPERTIES). *Let F be a graph clustering method. Let G be a similarity graph and $\mathcal{L}_G^{opt}$ be an optimal clustering of G according to F.*

Connectivity: *Let $\mathcal{L}_G$ be a clustering of G and $\mathcal{L}'_G$ be a clustering obtained by merging two disconnected clusters in $\mathcal{L}_G$. We say F satisfies* connectivity *if for every such $\mathcal{L}_G$ and $\mathcal{L}'_G$, $\mathcal{L}_G$ is considered better than $\mathcal{L}'_G$ (i.e., having a lower penalty).*

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 F satisfies* locality *if for every such G, G_1, and G_2, $\mathcal{L}_{G_1}^{opt} \cup \mathcal{L}_{G_2}^{opt}$ forms an optimal clustering for G.*

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

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



(a) Locality (b) Exchangeability (c) Separability

Figure 4: Linkage properties (Definition 3.2).

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

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 F 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.*

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 F 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 $G_1 - G_3$.* □

Among these properties, the first three are basic and a good clustering model should be able to satisfy them. The last two are more demanding. We can prove that correlation clustering has all of these desired properties.

THEOREM 3.3. *Correlation clustering (Eq.(1)) satisfies connectivity, locality, monotonicity, exchangeability, and separability.* □

PROOF. *Connectivity:* The clustering $\mathcal{L}_G$ has lower cohesion penalty and the same correlation penalty than $\mathcal{L}'_G$. So $\mathcal{L}_G$ is considered better.

Locality: Suppose in contrast there are better clusterings and we denote the best by $\mathcal{L}_G$. According to connectivity, nodes in G_1 and G_2 cannot be in the same cluster in $\mathcal{L}_G$, so we can split $\mathcal{L}_G$ into clusters for G_1 and clusters for G_2. Since there is no correlation between the two sets of clusters, without losing generality, the clustering for G_1 must have a lower penalty than $\mathcal{L}_{G_1}^{opt}$, contradicting with $\mathcal{L}_{G_1}^{opt}$ being optimal.

Monotonicity: We first prove for *positive monotonicity*. Suppose the weight of (v_1, v_2) increases from w_1 to $w_2 > w_1$. Suppose $\mathcal{L}_G^{opt}$ has penalty p_{opt} on G, then it has penalty $p_{opt} + w_1 - w_2$ on G'. For every clustering $\mathcal{L}$ on G with penalty p, we have $p \geq p_{opt}$. If v_1 and v_2 are in the same cluster in $\mathcal{L}$, the penalty of $\mathcal{L}$ on G' becomes $p + w_1 - w_2 \geq p_{opt} + w_1 - w_2$. If v_1 and v_2 are in different clusters in $\mathcal{L}$, the penalty of $\mathcal{L}$ on G' becomes $p - w_1 + w_2 > p_{opt} + w_1 - w_2$. So $\mathcal{L}_G^{opt}$ is still optimal on G'. We can prove for *negative monotonicity* similarly.

Exchangeability: Suppose in contrast there are better clusterings for G' and we denote the best by $\mathcal{L}_{G'}^{opt}$. Replacing $\bar{C}$ in $\mathcal{L}_G^{opt}$ with clusters in $\mathcal{L}_{G'}^{opt}$ will obtain a clustering that has lower penalty for subgraph G', the same penalty for subgraph $G \setminus G'$, and the same correlation penalty between clusters in G and those in $G \setminus G'$. Thus, the new clustering has a lower penalty, contradicting with $\mathcal{L}_G^{opt}$ being optimal. Finally, replacing $\bar{C}$ with another optimal clustering on G' obtains a clustering with the same penalty for subgraph G' and the same correlation penalty between G' and $G \setminus G'$, so the new clustering is also optimal.

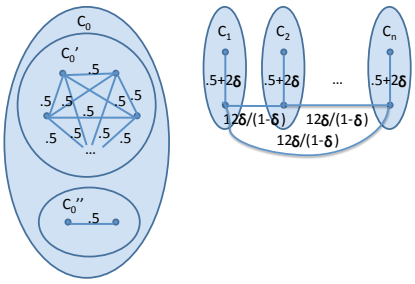


Figure 5: An example that violates properties in Definition 3.2 for DB-index.

Separability: According to connectivity, no cluster in $\mathcal{L}_G^{opt}$ is across G_1 and G_3. We now prove that there exists an optimal clustering that does not contain a cluster across $G_1 - G_3$. Then, according to exchangeability, we can find an optimal clustering of G such that no cluster is across G_1 and G_2, or across G_2 and G_3.

Suppose in an optimal clustering there is a cluster C across G_1, G_2, and G_3. We consider partitioning C into $C_1 \in G_1, C_2 \in G_2, C_3 \in G_3$ such that $C_1 \cup C_2 \cup C_3 = C$ and consider replacing C with $C_1, \ldots, C_3$ to obtain a new clustering $\mathcal{L}$. Since we have the same cohesion penalty within C_1, C_2, C_3 for C and for $C_1, \ldots, C_3$, the only difference is the cohesion penalty between $C_1 - C_3$ for $\mathcal{L}_G^{opt}$ and the correlation penalty between them for $\mathcal{L}$. Let p_{12} be the correlation penalty between C_1 and C_2 and p_{23} be the correlation penalty between C_2 and C_3 (note that the correlation penalty between C_1 and C_3 is 0 since they are disconnected). Then the correlation penalty for $\mathcal{L}$ is $p = p_{12} + p_{23}$ and the cohesion penalty for $\mathcal{L}_G^{opt}$ is $p' = |C_1||C_2| - p_{12} + |C_2||C_3| - p_{23} + |C_1||C_3|$. We next prove that $p \leq p'$, so $\mathcal{L}$ is optimal.

We first prove there must exist a clustering of $G_1 \cup G_2$ that contains clusters C_1 and C_2. Consider an optimal clustering $\mathcal{L}_{12}$ with no cluster across G_1 and G_2. First, if $\mathcal{L}_{12}$ contains a supercluster of C_1 (similar for C_2), denoted by C_1', we can show that adding $C_1' \setminus C_1$ into C would obtain a clustering with equal or less penalty for $\mathcal{G}$. Since $\mathcal{L}^{opt}$ is already optimal, there must be another optimal clustering of $G_1 \cup G_2$ that contains C_1 and C_2. Second, if $\mathcal{L}_{12}$ splits C_1 (similar for C_2), we can show that splitting C in a similar way would obtain a clustering with equal or less penalty for $\mathcal{G}$. Since $\mathcal{L}^{opt}$ is optimal, there must be another optimal clustering of $G_1 \cup G_2$ that contains C_1 and C_2. Finally, we can prove the same claim if $\mathcal{L}_{12}$ splits C_1 and each cluster also contains other nodes (similar for C_2). Since the optimal clustering contains C_1 and C_2, we must have $p_{12} \leq |C_1||C_2| - p_{12}$. Similarly, $p_{23} \leq |C_2||C_3| - p_{23}$. Therefore, we must have $p \leq p'$. □

We consider DB-index clustering because it represents distance-based clustering; however, because DB-index averages separation measures, none of these desirable properties holds, as the following theorem shows.

THEOREM 3.4. *DB-index clustering does not satisfy connectivity, locality, exchangeability, separability, or monotonicity.* □

PROOF. Figure 5 shows a counter example for these properties.

Connectivity: Clustering $\mathcal{L}_G'$ contains $n+1$ clusters: $C_0 - C_n, n > 2$. Among them, C_0 contains two subgraphs C_0' and C_0'', where C_0' has m nodes. 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 can prove 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 is $1 - \delta$.

Now consider splitting 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 $\frac{1+1+(1-\delta)n}{n+2} = 1 - \frac{n\delta}{n+2} > 1 - \delta$, so the new clustering is worse instead of better.

Locality: When $\alpha = 1$, clustering $\mathcal{L}_G'$ is the optimal clustering for G. Consider splitting G into $G_1 = \{C_0\}$ and $G_2 = \{C_1, \ldots, C_n\}$. The optimal clustering for G_1 contains two clusters C_0' and C_0''. The optimal clustering for G_2 contains clusters $C_1, \ldots, C_n$. However, as we know, $\{C_0', C_0'', C_1, \ldots, C_n\}$ is not the optimal clustering for G.

Monotonicity: Consider increasing the weight of the edge in cluster C_0'' from .5 to 1. Before the change, $\{C_0, \ldots, C_n\}$ forms the optimal clustering. After the change, however, $\{C_0', C_0'', C_1, \ldots, C_n\}$ forms the optimal clustering. We can prove violation of negative monotonicity by reducing the weight of edges between C_k and $C_k', k, k' \in [1, n]$ from $\frac{12\delta}{1-\delta}$ to 0, where $\{C_0', C_0'', C_1, \ldots, C_n\}$ becomes the optimal clustering after the change.

Exchangeability: The optimal clustering $\mathcal{L}_G$ is not optimal for the subgraph $\{C_0\}$.

Separability: Let $G_1 = \{C_0'\}, G_2 = \{C_0''\}, G_3 = \{C_1, \ldots, C_n\}$. They satisfy the condition, but the optimal clustering for G contains a cluster across G_1 and G_2. □

Comparison with [11]: Incremental linkage was briefly discussed for data updates in [11]. It is proved that if F (1) operates on clusters, and (2) is *general incremental*, then we can use F directly as f (*i.e.*, $F(G, \Delta G, \mathcal{L}_G) = F(G + \Delta G)$.) Here, F is general incremental if for every subgraph $G' \subset G$, we have $F(G', G \setminus G', \mathcal{L}_{G'}) = F(G)$. However, both CAUTIOUS for correlation clustering and the batch algorithm for DB-index clustering [6] operate on nodes rather than on clusters.

The properties in Definition 3.2 are for the objective function used in clustering rather than for the clustering algorithm F (*e.g.*, there can be many algorithms aiming at minimizing the penalty for correlation clustering). Our goal is to design the incremental algorithm f, which is based on F but can be different from F, such that $f(G, \Delta G, \mathcal{L}_G) \approx F(G + \Delta G)$ (recall that $\approx$ denotes that the results have similar quality).

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

3.2 Connected component algorithm

Intuitively, when the clustering algorithm satisfies connectivity and 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 3.5 (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, and the* transitive closure of an edge *as the connected subgraph in $G + \Delta G$ including the edge. The* connected component of ΔG, *denoted by $T(\Delta G)$, contains the transitive closure of each (added, removed, or changed) node or edge in ΔG.* □

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 result subgraph the *monotone connected component*.

DEFINITION 3.6 (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 clusters involving nodes in $T(\Delta G)$ with the optimal clusters for $T(\Delta G)$.

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

EXAMPLE 3.7. *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 the same 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 in contrast consider an increment ΔD_5 that removes node r_4. The transitive closure of r_4 within its cluster C_1'' contains only r_4. Thus, the connected component is empty and we simply remove C_1'' to obtain the new clustering, which is optimal for the graph with nodes $r_1 - r_3, r_5 - r_{17}$ under correlation clustering. □

We can prove that MONOCONNECTED is optimal if and only if connectivity, locality, and monotonicity hold for the clustering method.

LEMMA 3.8. *Algorithm* CONNECTED *is optimal if and only if the batch linkage method satisfies connectivity and locality.* □

PROOF. *If:* Under connectivity, the clusters in $\mathcal{L}_G^{opt}$ for the connected component (excluding the inserted nodes) must be disjoint from the clusters for the rest of the graph; we denote them by $\mathcal{L}_G'$ and $\mathcal{L}_G''$ respectively. CONNECTED finds an optimal clustering on the connected component; under locality, $\mathcal{L}_G''$ must be optimal on the rest of the graph. Thus, the new optimal clusters together with $\mathcal{L}_G''$ must form an optimal clustering for $G + \Delta G$ under locality.

Only if: First, the algorithm is valid only if connectivity holds; otherwise, $\mathcal{L}_G'$ and $\mathcal{L}_G''$ may be overlapping. Second, if locality does not hold, there must be an instance G, G_1, G_2 such that the optimal clustering for G_1 and that for G_2 do not form the optimal clustering for G. Let G_1 be the original graph and G_2 be the increment; CONNECTED does not find the optimal solution on $G_1 + G_2$. □

THEOREM 3.9 (OPTIMALITY OF MONOCONNECTED). *Algorithm* MONOCONNECTED *is optimal if and only if the batch linkage method satisfies connectivity, locality, and monotonicity.* □

PROOF. *If:* Consider the subgraph that is included in the connected component but not in the monotone connected component. We now prove clustering for this subgraph will not change under monotonicity. There are two cases:

- A node is connected to a changed edge but the edge has an increased weight and is within a cluster in $\mathcal{L}_G^{opt}$. Since such a weight change would not change the optimal clustering under positive monotonicity, clustering for these nodes will not change.
- A node is connected to a deleted node in another cluster in $\mathcal{L}_G^{opt}$ or to a changed edge but the edge has a decreased weight and is across clusters in $\mathcal{L}_G^{opt}$. Since such changes would not change the optimal clustering under negative monotonicity, clustering for these nodes will not change.

Only if: If positive monotonicity does not hold, there must be an instance G and an edge $(v_1, v_2) \in E$ whose weight increases such that the optimal clustering for G is not optimal for the new graph. Let G be the original graph and the weight change be the increment; the monotone connected component is empty, so MONOCONNECTED does not find the optimal solution on the new graph. We can prove for negative monotonicity similarly. □

COROLLARY 3.10. MONOCONNECTED *is optimal for correlation clustering but not optimal for DB-index clustering.* □

Finally, we show the complexity of the algorithm.

PROPOSITION 3.11 (COMPLEXITY OF MONOCONNECTED). *Let $s \leq |G|$ be the size of the monotone connected component for the increment and $f(s)$ be the complexity of finding the optimal clustering on a graph of s nodes. The complexity of* MONOCONNECTED *is $O(s + f(s))$.* □

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 3.12 (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 3.13. *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*

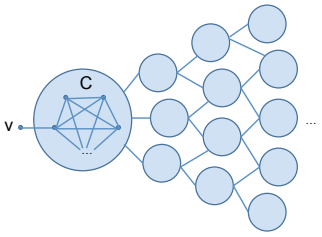


Figure 6: An instance where ITERATIVE can be much faster than MONOCONNECTED.

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. Find 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* 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 clusters and form a new subgraph 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'' for a new subgraph to be added 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, so can be slower than MONOCONNECTED. If we consider the longest chain between clusters in the graph, the number of iterations is bounded by the number of clusters on the chain. We next show the complexity of the algorithm formally.

PROPOSITION 3.14 (COMPLEXITY OF ITERATIVE). *Let $s' \leq |G|$ be the maximum size of the subgraphs in $\mathbf{Q}$, n be the number of connected subgraphs in G, and l be the number of clusters on the longest chain between clusters in G. The complexity of* ITERATIVE *is $O(nl(s' + f(s')))$.* □

Finally, we show that ITERATIVE is guaranteed to be optimal if and only if the clustering method satisfies connectivity, locality, exchangeability, separability, and monotonicity.

THEOREM 3.15 (OPTIMALITY OF ITERATIVE). *Algorithm* ITERATIVE *is optimal if and only if the batch linkage method satisfies connectivity, locality, exchangeability, separability, and monotonicity.* □

PROOF. We only need to prove that if and only if exchangeability and separability hold, ITERATIVE obtains the same clustering on the monotone connected component as MONOCONNECTED, which is optimal under connectivity, locality, and monotonicity.

If: Let G_1 be the subgraph that contains all clusters in the result clustering but not in $\mathcal{L}_G^{opt}$, G_2 be the subgraph with neighbor clusters of G_1, and G_3 be $\hat{T}(\Delta G) \setminus (G_1 \cup G_2)$. The algorithm guarantees that we have an optimal clustering for $G_1 \cup G_2$ with no cluster across G_1 and G_2. Exchangeability guarantees that we have an optimal clustering for $G_2 \cup G_3$ with no cluster across G_2 and G_3.

Suppose in contrast, there are better clusterings for $\hat{T}(\Delta G)$, and we denote the best by $\mathcal{L}$. Because of separability, we can find such $\mathcal{L}$ with no cluster across the three subgraphs. Let $\mathcal{L}_{12} \subseteq \mathcal{L}$ be the subset of clusters for nodes in $G_1 \cup G_2$ and $\mathcal{L}_3 = \mathcal{L} \setminus \mathcal{L}_{12}$ be those for G_3. Consider replacing $\mathcal{L}_{12}$ with the clusters ITERATIVE obtains on $G_1 \cup G_2$ and replacing $\mathcal{L}_3$ with those for G_3 in $\mathcal{L}_G^{opt}$. Because of exchangeability, the new clustering is also optimal on G. The result is exactly the result of ITERATIVE on $\hat{T}(\Delta G)$, contradicting with the assumption that $\mathcal{L}$ is better.

Only if: If exchangeability or separability does not hold, we can construct a counter example where ITERATIVE obtains a suboptimal clustering.

(1) If exchangeability does not hold, we must have G and $\bar{C}$ such that replacing $\bar{C}$ with an optimal clustering on G' obtains a non-optimal clustering on G. There must be one optimal clustering $\mathcal{L}$ for G' that contains multiple clusters. Let $\bar{C}' \subseteq \bar{C}$ be the subset of clusters that are connected to $G \setminus G'$.

Suppose $\bar{C}' \neq \bar{C}$. Let $\bar{C} \setminus \bar{C}'$ be the increment and the rest of the graph be the original similarity graph. ITERATIVE can first generate $\mathcal{L}$ for G'; because there is no change for clusters in $\bar{C}'$, ITERATIVE would terminate. However, the resulting clustering is not optimal on G.

Suppose instead, $\bar{C}' = \bar{C}$, so $\bar{C} \setminus \bar{C}'$ is empty. We add a new node r to G and for each $C \in \bar{C}$, add an edge from r to a node in C with a weight very close to 0. Because of connectivity, the optimal clustering for G and for G' remain the same except that there is a new cluster for r. Let r be the increment and the rest of the graph be the original similarity graph. ITERATIVE can first generate $\mathcal{L}$ for G'; because there is no change for clusters in $\bar{C}$, ITERATIVE would terminate. However, the resulting clustering is not optimal on G.

(2) If separability does not hold, we must have G_1, G_2, G_3 such that all optimal clusterings for $G_1 \cup G_2 \cup G_3$ contain at least a cluster across two or three clusters even when the required conditions are satisfied. Let G_1 be the increment and $G_2 \cup G_3$ be the original graph. ITERATIVE can terminate with a clustering where there is no cluster across the subgraphs, thus not optimal. □

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

4. GREEDY SOLUTION

As we have shown, neither the connected component algorithm nor the iterative algorithm is ideal: the former may require considering an unnecessarily big subgraph when the similarity graph is well-connected; the latter 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* 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 that consist of multiple directly connected clusters, whereas GREEDY iterates at the finer granularity of clusters.

In the rest of the section, we first describe the framework of the algorithm (Section 4.1), and then discuss how to instantiate it for particular clustering methods (Sections 4.2-4.3).

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''.* □

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  Q^c ← ∅;
2  G' ← T̄(ΔG);
3  Put each cluster in G' to Q^c;
4  while Q^c ≠ ∅ do
5      dequeue C ∈ Q^c;
6      changed ← false;
       // operations return true if they change the clustering
7      changed ← MERGE(C, G + ΔG, L_G, Q^c);
8      if ¬changed then
9          changed ← SPLIT(C, G + ΔG, L_G, Q^c);
10     if ¬changed then
11         changed ← MOVE(C, G + ΔG, L_G, Q^c);
12 return L_G;
```

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 it 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 $C' \in \mathbf{Q^c}$.

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.7), splitting (Ln. 9), and moving (Ln.11). 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.8, 10). 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 $\mathbf{Q^c}$.

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	-	$\emptyset$

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. □

We next discuss the complexity of the greedy algorithm. Note that in practice, the number of clusters that ever occur in $\mathbf{Q^c}$ is typically much smaller than the upper bound.

THEOREM 4.5 (COMPLEXITY OF GREEDY). *Let G and ΔG be the input. Let $s'' \leq |G|$ be the maximum size of the encountered clusters, c be the maximum number of neighbor clusters of each cluster, and m be the largest number of clusters in $G + \Delta G$ at the same time. Let $n \leq (m+s'')|G|^2$ be the total number of clusters in $\mathbf{Q^c}$. Denote by $g(|G|)$ the time of evaluating the objective function for graph G. The complexity of* GREEDY *is $O(ncs''^2 g(s''))$.* □

PROOF. We prove $n \leq (m+s'')|G|^2$. Consider a node $v \in G + \Delta G$ and the sequence of clusters containing v in $\mathbf{Q^c}$. According to the algorithm, we would observe that in the sequence, the initial cluster expands (*i.e.*, superset) for a few times, then reduces (*i.e.*, subset) for a few times, and then expands and reduces again till convergence. We call each sequence of expansions and reductions one *round*. In each round the number of expansions is at most m and the number of reductions is at most s''. In each round we move out at least one node that would never be added back to the same cluster as v; otherwise, there is a cluster in a later round that is a superset of the current cluster. So there are at most $|G| - 1$ rounds. Thus, there are no more than $(m + s'')|G|$ clusters in the sequence for v, and so no more than $(m + s'')|G|^2$ clusters in $\mathbf{Q^c}$. □

4.2 Instantiation for correlation clustering

We now instantiate the greedy algorithm for correlation clustering (GREEDYCORR). We show that 1) many simplifications made by the greedy algorithm, such as merging only two clusters each time or splitting out one node each time, would not affect the quality of the clustering results, and 2) 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 $\sum_{v \in C, v' \in C'} w(v, v') > \frac{|C| \cdot |C'|}{2}$. We next show that if merging a few clusters leads to a better clustering, MERGE can do so by iteratively merging two clusters in each step.

THEOREM 4.6 (PROPERTY OF MERGE). *Let $G, \Delta G, \mathcal{L}_G^{opt}$ be the input and $C_1, \ldots, C_n$ be the set of clusters in $\bar{T}(\Delta G)$ and $\mathcal{L}_G^{opt}$. If merging clusters $C_1, \ldots, C_n$ leads to a better clustering,* MERGE *would do the merging iteratively.* □

PROOF. According to the connectivity property of correlation clustering, $C_1, \ldots, C_n$ must be connected. We next prove that from $C_1 - C_n$ we must be able to find two connected clusters to merge; the result would be put to $\mathbf{Q^c}$ and the process would continue until all clusters are merged together. Because merging $C_1, \ldots, C_n$ leads to a better clustering, we must have $\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p_{ij} > \frac{\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} |C_i||C_j|}{2}$, where p_{ij} is the correlation penalty between C_i and C_j. There must exist $k, k' \in [1, n]$ such that $p_{k,k'} > \frac{|C_k||C_{k'}|}{2}$. One of C_k and $C_{k'}$ cannot be contained in $\mathcal{L}_G^{opt}$ so must be contained in $\mathbf{Q^c}$; otherwise, it contradicts with $\mathcal{L}_G^{opt}$ being optimal. Thus, we will merge C_k and C'_k. □

Algorithm 2: SPLITCORR($C, G, \mathcal{L}_\mathcal{G}, \mathbf{Q^c}$)

Input : C: cluster for consideration;
G: similarity graph after updates;
$\mathcal{L}_\mathcal{G}$: current clustering of G;
$\mathbf{Q^c}$: working queue
Output: $\mathcal{L}_\mathcal{G}$ and $\mathbf{Q^c}$ updated according to splitting of C

1 $C' \leftarrow \emptyset$;
2 **foreach** $v \in C$ **do**
3 $d(v) \leftarrow$ FETCHCONN(v, C); // FETCHCONN *obtains the sum of edge weights between v and every other node in C from an auxiliary data structure.*
4 **while** $|C| \neq \emptyset$ **do**
5 Select v_0 as the node with the lowest $d(v_0)$;
6 **if** $d(v_0) > \frac{|C|-|C'|-1}{2}$ **then**
7 **break**;
8 Move v_0 to C';
9 **foreach** $v \in C$ **do**
10 $d(v) \leftarrow d(v) - 2 * w(v, v_0)$;
11 **if** $C' \neq \emptyset$ **then**
12 Add C' into $\mathcal{L}_\mathcal{G}$;
13 Add C and C' into $\mathbf{Q^c}$ if connected with other clusters;

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. We show the revised algorithm, namely SPLITCORR, in Algorithm 2. Lines 2-3 show how we initialize the difference from an auxiliary data structure, which computes the sum of edge weights between a node v and a cluster C. Lines 9-10 show how we maintain the difference incrementally. Now we show that SPLITCORR can find the best way to split C into two clusters.

THEOREM 4.7. *If the best split of $C \in \mathbf{Q^c}$ is C_1 and C_2,* SPLITCORR *can generate these two clusters.* □

PROOF. We first prove that if such a split exists, there must be a node whose connectivity is at most .5; thus, SPLITCORR will move one node out. Let p_{12} be the correlation penalty between C_1 and C_2. Then, we must have $p_{12} < \frac{|C_1||C_2|}{2}$. Let $p(v)$ be the sum of the edge weights between $v \in C_1$ and each node in C_2. Then, $\sum_{v \in C_1} p(v) < \frac{|C_1||C_2|}{2}$. For this inequation to hold,

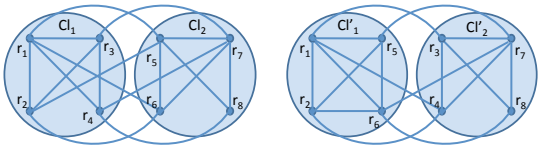


Figure 7: An example showing that the greedy algorithm cannot "switch" nodes between clusters in moving.

the node in C_1 with the lowest connectivity, denoted by v_1, must satisfy $p(v_1) < \frac{|C_2|}{2} \leq \frac{|C_1|+|C_2|-1}{2} = \frac{|C|-1}{2}$, so the connectivity of v_1 is at most .5.

Without losing generality, assume the node with the lowest connectivity, denoted by v_1, is contained in C_1. We now prove that given any $C'_1 \subset C_1$ and $C'_2 = C \setminus C'_1$, there must exist a node $v \in C''_1 = C_1 \setminus C'_1$ such that $p_{C'_2}(v) - p_{C'_1}(v) \leq \frac{|C'_2|-|C'_1|-1}{2}$; thus, SPLITCORR will move this node from C'_2 to C'_1. This process would continue until C_1 and C_2 are obtained. Let p'_{11} be the correlation penalty between C''_1 and C'_1, and p'_{12} be the correlation penalty between C''_1 and C_2. Since C_1 and C_2 are the best splitting, we must have a lower penalty for it; that is, $|C''_1||C'_1| - p'_{11} + p'_{12} < |C''_1||C_2| - p'_{12} + p'_{11}$, so $p'_{12} - p'_{11} < \frac{|C''_1|(|C_2|-|C'_1|)}{2}$. Let $p_{C'_2}(v)$ be the sum of the edge weights between $v \in C''_1$ and each node in C_2 and $p_{C'_1}(v)$ be the sum of the edge weights between v and each node in C'_1. Then, $\sum_{v \in C''_1}(p_{C'_2}(v) - p_{C'_1}(v)) < \frac{|C''_1|(|C_2|-|C'_1|)}{2}$. For this inequation to hold, the node in C''_1 with the lowest difference between $p_{C'_2}(v)$ and $p_{C'_1}(v)$ must satisfy $p_{C'_2}(v) - p_{C'_1}(v) < \frac{|C_2|-|C'_1|}{2} \leq \frac{|C'_2|-|C'_1|-1}{2}$. This proves the claim. □

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 $p_{C'}(v) - p_C(v) < \frac{|C'|-|C|-1}{2}$. We call the result algorithm MOVECORR. In a similar way to SPLITCORR, we can prove that MOVECORR can find the best way to move a subset of nodes between C and C'.

THEOREM 4.8. *Let $C, C' \in \mathbf{Q^c}$ be two clusters for consideration. If the best split of $C \cup C'$ is $\hat{C}$ and $\hat{C}'$ where $\hat{C} \subset C$ or $\hat{C} \supset C$,* MOVE *can generate these two clusters.* □

Note however that MOVE does not necessarily generate the best split of $C \cup C'$, as the next example shows.

EXAMPLE 4.9. *Consider Cl_1 and Cl_2 in Figure 7, where each edge has a weight of 1.* MOVE *will not move any node between them and the final penalty is $1+1+7=9$. The best split of $Cl_1 \cup Cl_2$, however, requires switching nodes $r_3 - r_4$ with nodes $r_5 - r_6$, obtaining Cl'_1 and Cl'_2 with a penalty of $0+1+6=7$.* □

The previous discussions show that for each operation, the greedy algorithm can find the "local" optimal solution. However, combining these local optimal solutions may not lead to a global optimal solution. Indeed, Mathieu et al. [9] shows that if the increment contains only inserted nodes and we apply only MERGE, the approximation bound is $O(2|G|+1)$.

These aforementioned simplifications can reduce the complexity of the algorithm, shown as follows. In addition, GREEDYCORR can benefit from pre-computing intra- and inter-penalties and incrementally maintaining them in the iterations.

PROPOSITION 4.10 (COMPLEXITY OF GREEDYCORR). *Define the same parameters as in Theorem 4.5. The complexity of* GREEDYCORR *is $O(ncs''^2)$.* □

Table 2: Statistics of data sets according to Corr. Clus.

Statistics		*Biz* (5k)	*Biz* (1k)	*Cora* (Jac.)	*Cora* (M-E)
Node	Number	5000	1000	997	997
	Avg #neighbors	2.7	2.1	34	50
	Max #neighbors	38	13	139	172
Cluster	Number	2769	477	481	726
	Avg #nodes	1.8	2.1	2.1	1.2
	Avg #neighbors	1.1	.4	16	40
	Max #neighbors	37	13	70	172
Subgraph	Number	1832	424	87	55
	Avg #nodes	2.7	2.4	12	18
	Max #nodes	84	14	795	900

4.3 Instantiation for DB-index clustering

Instantiation for DB-index (GREEDYDB) is exactly the same as the framework itself. However, to simplify the computation for DB-index, we recompute separation measures for only the clusters for examination and those that have the highest separation measure with these clusters. In addition, we can maintain the same auxiliary data structure as in GREEDYCORR to avoid repeated computation. We cannot prove the properties in Theorems 4.6-4.8; however, we show empirically that the greedy algorithm works well on real world data for DB-index clustering as well (Section 5).

PROPOSITION 4.11 (COMPLEXITY OF GREEDYDB). *Define the same parameters as in Theorem 4.5. The complexity of* GREEDYDB *is $O(nc^2 s''^3)$.* □

5. EXPERIMENTAL EVALUATION

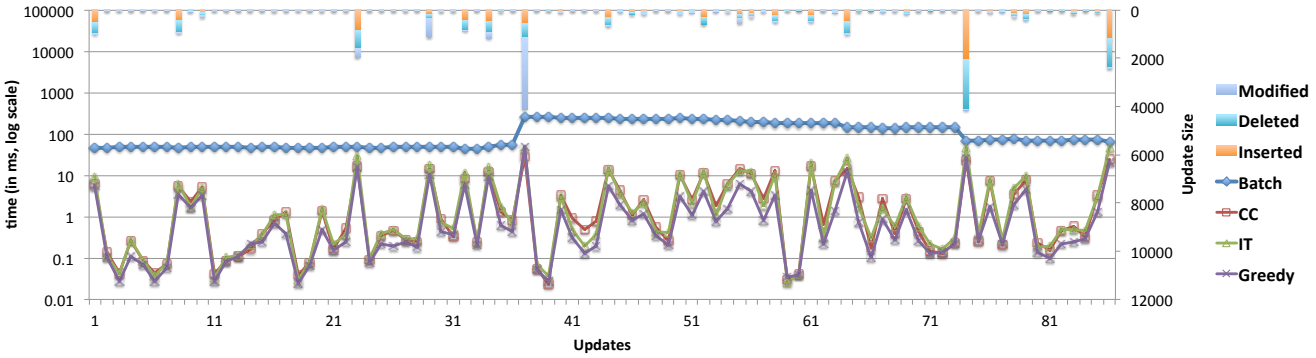
We present experimental results on two real-world data sets with different characteristics. The experimental results show the promise of the incremental algorithms, especially showing that GREEDY typically has the highest efficiency and the highest quality compared with batch linkage and other incremental linkage algorithms.

5.1 Experiment setup

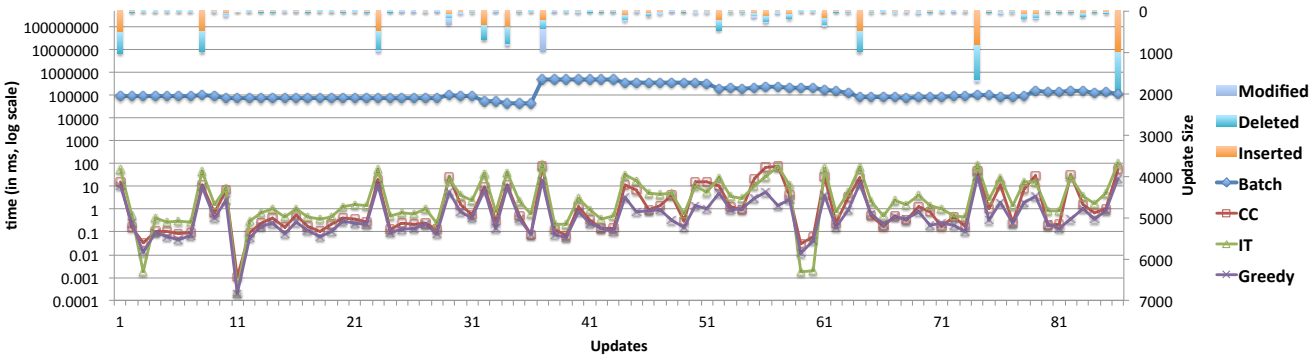
Data sets: We experimented on two data sets. The first data set, *Biz*, contains 87 snapshots of business records in the San Francisco area; we took the first snapshot as the original data set, and computed an increment for each later snapshot. We evaluated correlation clustering on the full data set and evaluated DB-index clustering on a subset with 20% of the data, because batch linkage for DB-index took a very long time. We next discuss our observations on the full data set and summarize for both data sets in Table 2. The first snapshot of *Biz* contains 5K records. The increments contain on average 120 `Inserts`, 118 `Deletes`, and 59 `Changes`, and the maximum number of operations in an increment is 4120. The top part of Figure 8(a) (with the Y-axis on the right side of the figure) shows a break down of the updates for each increment. We applied the *Monge-Elkan* [3] string similarity for pairwise similarity computation and ignored edges with a similarity below .7. The similarity graphs are fairly sparse: (1) each node is connected to 2.7 nodes on average and to 38 nodes at most; (2) there are 1832 connected subgraphs, each containing 2.7 nodes on average, 84 nodes and 73 clusters at most. We do not have a gold standard for this data set; from the results of correlation clustering over all iterations, we observed that the clusters are typically small and sparsely connected: (1) there are 2769 clusters on average in each similarity graph, each containing 1.8 nodes on average and 38 nodes at most; (2) each cluster is directly connected to 1.1 clusters on average and 37 clusters at most.

The second data set, *Cora*[3], contains 997 publication records. We applied two string similarity metrics on this data set: *Jaccard*

[3]http://www.cs.umass.edu/mccallum/data/cora-refs.tar.gz



(a) Correlation clustering on *Biz* (5K).



(b) DB-index clustering on *Biz* (1K).

Figure 8: Comparison of various algorithms under CONT on *Biz*.

and *Monge-Elkan* [3]. The similarity graph is very dense for both metrics, and even denser for the latter metric. Below we discuss only for the Jaccard metric and we summarize for both metrics in Table 2. (1) Each node is connected to 34 nodes on average and to 139 nodes at most. (2) There are 87 connected subgraphs, each containing 12 nodes on average and 795 nodes at most; (3) each cluster is directly connected to 16 clusters on average and 70 clusters at most. (4) according to the gold standard, there are 191 clusters, each containing 5.2 records on average and 102 records at most. On this data set we have a single snapshot and we generate the increment 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.*, 10-th) increment, we only add back the previously removed (512) records. We repeated 100 times and took the average.

We note that these two data sets have different features: *Biz* contains a large number of records but the connections between the records are quite sparse; *Cora* contains only a small number of records but the similarity graph is well connected.

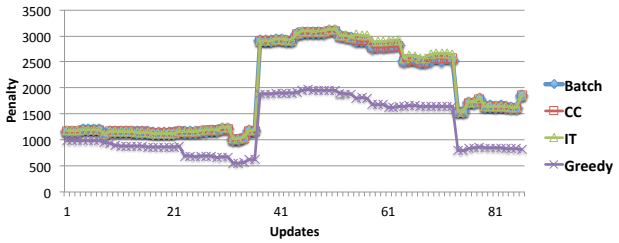
Implementations: For each of correlation clustering and DB-index clustering, we compared four algorithms: BATCH, the baseline, applies the batch linkage algorithm (CAUTIOUS [1] for correlation clustering and the hill climbing algorithm in [6] for DB-index clustering); CC applies CONNECTED; IT applies ITERATIVE; and GREEDY applies the greedy algorithm. 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 more likely to use CONT for updates and periodically apply batch linkage. In CAUTIOUS, we used parameter $\delta = .1$ [1]; in DB-INDEX, we set $\alpha = .05$ and $\beta = .001$. We implemented in Java, and experimented on a Linux machine with four Intel Xeon(R) X3360 cores (2.83GHz, cache 6MB).

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 when 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 and (2) the *F-measure* when 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 *F-measure* is computed as $\frac{2 \cdot precision \cdot recall}{precision + recall}$.

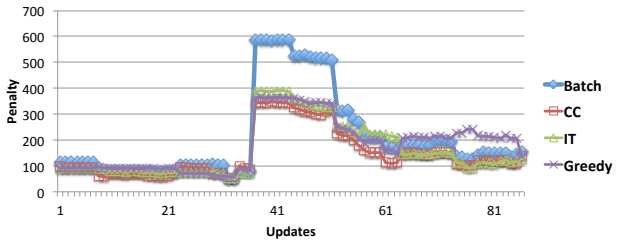
5.2 Experiments on *Biz*

Overview: Figure 8 (with the Y-axis on the left side of the figure) shows the execution time for each increment for the four methods under CONT; Figure 9 shows the penalty of the results under CONT; and Table 3 gives a summary.

We have three observations on correlation clustering. First, all three incremental linkage algorithms significantly improve over BATCH; indeed, for 69% of the increments GREEDY reduced linkage time by 2-3 orders of magnitude. Second, among the three incremental algorithms, GREEDY has the smallest execution time: it reduced execution time by 32% over CC and by 47% over IT; it is faster than CC on 88% of the increments and than IT on 97% of the increments. This is because each iteration builds upon the previous one and greedily explores Merge, Split, and Move operations. Third, CC has comparable penalty to BATCH (on average 1949 vs. 1941) and IT has slightly higher penalty (2% higher than BATCH); this is because we applied CAUTIOUS for correlation clustering, which often returns sub-optimal clustering and may return different results each time we run it. On the other hand, GREEDY has much



(a) Correlation clustering on *Biz (5K)*.



(b) DB-index clustering on *Biz (1K)*.

Figure 9: Penalty for CONT on *Biz*.

Table 3: Comparison of various algorithms on *Biz*. Highest performance is highlighted in bold.

Method			Time (ms)	Impro.	Penalty
Corr Clust. (5k)		BATCH	10205	-	1941
	CONT	CC	333	96.7%	1949
		IT	431	95.8%	1979
		GREEDY	228	**97.8%**	**1239**
	RESET	CC	331	96.8%	1947
		IT	431	95.8%	1949
		GREEDY	248	**97.6%**	**1833**
DB-Index (1k)		BATCH	3.8 hr	-	211
	CONT	CC	673	99.9995%	**147**
		IT	1060	99.9992%	162
		GREEDY	189	**99.9999%**	182

lower penalty (36% lower than BATCH) (note that under RESET the penalty is higher, since it cannot benefit from the improvement on the previous increments); this shows that whereas both GREEDY and CAUTIOUS output sub-optimal results, GREEDY often obtains a higher quality than CAUTIOUS. In addition, as we show soon (Table 5), GREEDY has a lot of merges and splits on existing clusters, showing that the ability to fix previous errors is important to obtain high quality of linkage.

For DB-index clustering, the batch algorithm, which is essentially a hill-climbing algorithm, is very expensive as it looks for the local optimal solution at each step; in total it took 3.8 hours on all increments. The advantage of GREEDY is even more pronounced here: on average it improved over BATCH by 4 orders of magnitude and improved over CC by 72%. However, GREEDY did not have the lowest penalty: as we have shown, DB-index lacks the many desired properties as we described in Defn 3.2; thus, the results are more random and there is not a consistent pattern over all increments. However, we observed that the incremental algorithms obtain a lower penalty on average than the batch algorithm.

IT vs. CC: We now compare CC and IT in more detail. Table 4 compares them on the top-5 and bottom-5 increments in terms of size, and Table 5 shows detailed statistics for CONT and correlation clustering. (1) Because of the sparsity of the graph, IT converged in only 2.0 iterations on average and 3 iterations at most; actually, for 86% of the increments, it converged in 2 iterations. (2) It did end up examining smaller subgraphs on average; however, for 35% of the increments it examined the same subgraph as CC, and even

Table 4: Comparison on top-5 and bottom-5 increments on *Biz* (Correlation Clustering (5k), DB-index (1k), CONT).

	Corr. Clust. (ms)				DB-index (ms)			
	Size	CC	IT	GREEDY	Size	CC	IT	GREEDY
Top	4120	**23**	47	27	1999	54.7	101	**21.4**
	4108	**27**	45	49	1696	43.5	87	**27.1**
	2395	25	46	**21**	1023	15.5	58	**10.5**
	1982	17	30	**16**	1001	18.7	54.6	**11.3**
	1172	12	16	**9**	995	11.9	45.5	**9.5**
Bott.	2	.042	.032	**.028**	2	**.078**	.62	.079
	4	.042	**.039**	**.039**	4	.065	**.002**	.04
	7	**.133**	.163	.138	6	**.174**	1.04	.232
	10	.152	.195	**.101**	10	.209	.817	**.15**
	15	.654	.301	**.221**	13	.243	.761	**.158**

Table 5: Details on *Biz* (Corr. Clust. CONT, averaged).

	CC	IT	GREEDY
time (ms)	333	431	227
#Iterations	1	2.0	2.8
#Nodes	526	488	500
#Edges	1001	932	945
#Total-nodes	-	722	1469
#Total-edges	-	1431	6346
#Examined-Merge	-	-	173
#Real-Merge	-	-	95
#Examined-Split	-	-	56
#Real-Split	-	-	17
#Examined-Move	-	-	12
#Real-Split	-	-	0

for the rest of the increments, the subgraphs it examined were not significantly smaller (on average 11% fewer nodes and 10% fewer edges). (3) Because of its iterative nature, it may examine the same part of a subgraph multiple times when there are more than one iteration. If we take into consideration the number of times it examined each node and edge, for 78% of the increments it ended up examining more nodes and edges in total than CC; on average it examined 37% more nodes and 43% more edges. (4) When IT examined fewer nodes and edges in total, it can be significantly faster than CC: for 14% of the increments it examined fewer nodes and is 49% faster than CC; however, for the rest of the increments, it is 33% slower than CC. (5) As shown in Table 4, for correlation clustering, IT is faster on smaller increments, where the execution time is low, but slower on larger increments, where the execution time is high, so its total execution time is 29% higher than CC; for DB-index clustering, IT is slower most of the time because theobjective function is expensive to compute.

GREEDY: We now compare GREEDY with CC and IT; again, Table 4 and Table 5 show the statistics. We have three observations. First, although GREEDY iterates at the cluster granularity, it does not have many more iterations than IT, which iterates at the subgraph granularity; indeed, on average it finished in only 2.8 iterations. Second, we observe that GREEDY also ended up examining nearly the same number of nodes and edges as CC, and turned out to examine more nodes and edges in total than IT; however, because each iteration is based on the previous iteration, GREEDY does not have many wasted efforts and is much faster than both CC and IT: taking correlation clustering as an example, for 3 out of the 5 largest increments, it was the fastest and saved 15% execution time over CC, and for 4 out of the 5 smallest increments, it was the fastest and saved 35% execution time over CC. This suggests that operating on clusters rather than subgraphs is highly beneficial. Third, on average GREEDY examined merging 173 times and conducted merging 95 times (55%); it examined splitting 56 times and conducted splitting 17 times (30%); it examined moving 12

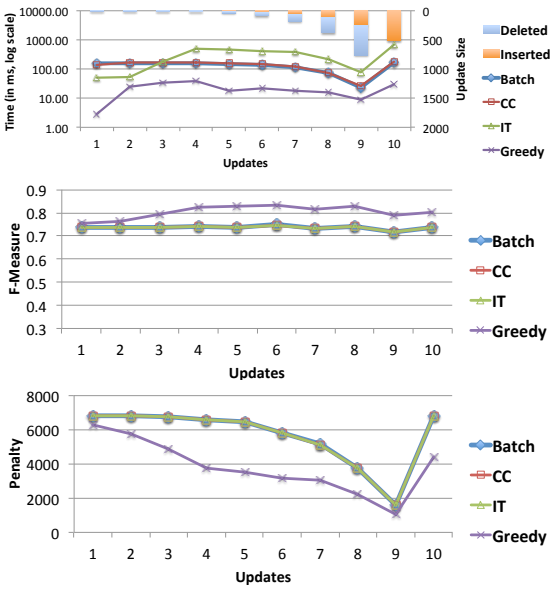


Figure 10: Experimental results on *Cora* (JACCARD).

times but did not conduct any moving. This justifies the order in which we consider these three operations; also 10.5% of merging and all of splitting is to fix previous errors, showing the benefit of our approach.

5.3 Experiments on *Cora*

Figure 10 shows the execution time and quality of our proposed methods using *Jaccard* similarity and correlation clustering on *Cora*. We observed the same trend on *Monge-Elkan* and on DB-index clustering. Most of our observations are consistent with the observations on *Biz*. Here we highlight several differences or new observations. First, we observe that because of the high connectivity of this data set, CC finished in nearly the same time as BATCH; indeed, recall that the number of subgraphs is only 87, so CC ended up examining 80% on average and 96% at most of the graph. Second, same as on *Biz*, IT is slower than CC when it examines nearly the same subgraph; because of the high connectivity of this subgraph, this happens even more often and IT is faster than CC only for the first two increments, where each increment contains a very small number of inserts and deletes. Third, GREEDY continues to be much better than any other algorithm: on average 84% faster than BATCH, 9% higher F-measure, and 33% lower penalty (note the correlation between penalty and F-measure). Fourth, the execution time for BATCH and CC decreased from the first increment to the ninth increment, and then increased at the tenth increment, because the total size of the graph first decreases and then increases. The execution time for IT and GREEDY increased for the first three increments because for iterative approaches, when the size of the increment is small, the size of the increment determines the execution time, but when the size of the increment is large, the size of the resulting data set determines the execution time as the algorithms are likely to examine the whole graph or a large part of the graph. Finally, we note that even in the last round, where the number of inserted nodes is the same as the number of existing nodes, GREEDY was still 82% faster than BATCH, showing the high effectiveness of GREEDY.

6. RELATED WORK

Record linkage has been extensively studied in the literature (surveyed in [5, 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 [10, 11]; however, they focused on evolving matching rules and discussed evolving data only very briefly. We have compared with their work in detail in Section 3.1.

For batch clustering, there are typically 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. Second, for records in the same block, it computes pairwise similarity. Third, it clusters the records based on pairwise similarity, 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. Since performing blocking and pairwise similarity computation incrementally is fairly straight-forward when the previous results are available, this paper focuses on incremental clustering.

There have been many clustering algorithms proposed, and we can classify them into two categories: *hierarchical clustering* and *objective-function based clustering*. Hierarchical clustering for record linkage mainly contains agglomerative clustering such as SWOOSH [2]. Indeed, most of the agglomerative clustering algorithms are general incremental, so according to [11], we simply need to apply the batch algorithm on the previous clustering results and the singleton clusters for the inserted nodes. Our paper considers objective-function based clustering and focuses on those that do not require *a priori* knowledge of the number of clusters and thus are suitable for record linkage, where the number of entities in the data is typically unknown. We designed algorithms that are widely applicable; we proved optimality of MONOCONNECTED and ITERATIVE when the objective function satisfies a natural set of desired features, and we show empirically the effectiveness of our algorithms instantiated for correlation clustering and DB-index clustering on real-world data sets. Finally, we note that incremental correlation clustering has been studied for the case where (1) one vertex is added each time, and (2) already identified clusters need to be preserved [9]. In contrast, our algorithms allow leveraging new evidence from updates for fixing previous clustering errors.

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 the new evidence from the updates to fix previous linkage errors. We conducted experiments on two real-world data sets showing the high efficiency and quality of our algorithms. 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] O. Benjelloun, H. Garcia-Molina, D. Menestrina, Q. Su, S. E. Whang, and J. Widom. Swoosh: A generic approach to entity resolution. Technical Report 2005-5, Stanford InfoLab, 2005.

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

[4] 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.

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

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

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

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

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

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

[11] S. Whang and H. Garcia-Molina. Incremental entity resolution on rules and data. *VLDBJ*, 2013.