Progressive Entity Resolution over Incremental Data

Leonardo Gazzarri
University of Stuttgart
Stuttgart, Germany
leonardo.gazzarri@ipvs.uni-stuttgart.de

Melanie Herschel
University of Stuttgart
Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

ABSTRACT

Entity Resolution (ER) algorithms identify entity profiles corresponding to the same real-world entity among one or multiple data sets. Modern challenges for ER are posed by volume, variety, and velocity that characterize Big Data. While progressive ER aims to efficiently solve the problem under time constraints by prioritizing useful work over superfluous work, incremental ER aims to incrementally produce results as new data increments come in.

This paper presents algorithms that combine these two approaches in the context of streaming and heterogeneous data. The overall goal is to maximize the chances to spot duplicates to a given entity profile in a moment closest to its arrival time (early quality), without relying on any schema information, while being sufficiently efficient to process large volumes of fast streaming data without compromising the eventual quality (by cutting too many corners for efficiency). Experiments validate that our algorithms are the first to support incremental and progressive ER and, compared to state-of-the-art incremental approaches, improve early quality, eventual quality, and system efficiency by progressively and adaptively performing the unexecuted comparisons that are more likely to match when waiting for the next stream input increment.

1 INTRODUCTION

Entity Resolution (ER) approaches identify which different *profiles* stored within data sources refer to the same real-world entity. It is a fundamental step in data cleaning and data integration. While it has been studied widely in the last decades [8], the Big Data era has raised additional challenges [11]: (1) data integration applications may operate on huge volumes of data collected from the Web. (2) Data are continuously generated, requiring applications to work incrementally. (3) Data may exhibit considerable diversity even among profiles referring to the same entity. To perform ER in this setting, schema-agnostic ER solutions have been proposed. As surveyed in [9], these can be classified in three broad categories.

Batch ER produces as output a set of duplicates, either within a single input dataset (Dirty ER) or across multiple clean datasets (Clean-Clean ER). Each duplicate identifies a pair of profiles referring to the same entity. The process of finding duplicates is often time consuming due to large datasets and an instrinsic quadratic number of expensive comparisons. To scale to large amount of data, batch ER often adopts *blocking*, which assigns profiles to blocks and then only compares profiles of a same block.

Progressive ER targets ER applications that benefit from finding duplicates early, e.g., because they have a maximal time budget (and the number of duplicates found in that time should be maximized). The main idea is to execute the comparisons of profile

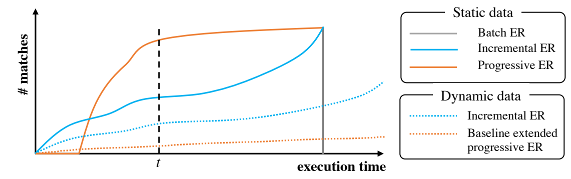

Figure 1: Matches found over time by different ER approaches over static and dynamic data (sketched behavior).

pairs that are most likely to be matches as early as possible. While both batch and progressive ER operate on static datasets, i.e., datasets fully available upfront and that do not change over time, the latter focuses on minimizing the average time between two consecutive duplicate detections rather than reducing the overall time to finish processing a static dataset [30]. To achieve this behavior, progressive ER algorithms typically start off with a pre-analysis step. This step divides into *blocking* (as in batch ER) and *prioritization* that ranks pairwise comparisons and defines the order of execution.

Incremental ER operates on data increments, i.e., sets of entity profiles arriving at different time instants. Considering a non empty data increment ΔD and a previously processed dataset D, the goal of incremental ER is to find new duplicates by exploiting previous computations from processing D. Incremental ER approaches are designed to run much faster than running batch ER on the "new" dataset $D' = D \uplus \Delta D$, while reaching comparable quality [18]. Incremental blocking approaches select a small number of new comparisons involving at least one profile from ΔD. This selection is done independently from the input rate of increments or the rate at which selected comparisons are further processed, which, as we will see in experiments, leads to a fluctuating performance.

Figure 1 sketches the ideal behavior of the three approaches when a finite dataset is available a priori (static data). It plots the typical number of discovered matches over time. Let us consider time t that precedes the time needed to process the full dataset (at the rightmost vertical solid grey line, when batch ER terminates and returns all the matches at the same time). At time t, progressive ER should give a better *partial* result than both batch and incremental ER, assuming t comes after completion of its pre-analysis step (flat line at the beginning).

When the data are not available upfront, increments stream in at a possibly varying rate (dynamic data). To keep up with increment input rates, incremental ER either slows down the upstream to reduce the load or prunes comparisons. This may cause slow stream consumption or lost matches: While the former strategy delays the computation and thus the delivery of the results, the latter loses matching results when comparisons yielding matches get pruned. The degradation in performance is sketched as dotted blue line in Figure 1, which we also observe experimentally (see, e.g., Figures 2 and 7). This is a point where the prioritization of progressive ER would come in handy. The idea here is to

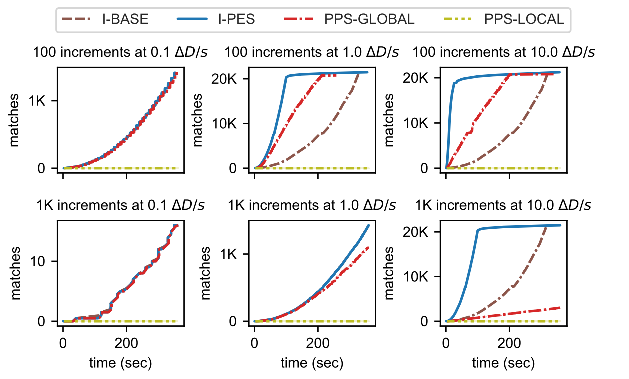

Figure 2: Two progressive ER algorithms in incremental setting (PPS-global and PPS-local), one incremental ER algorithm (I-BASE) and one PIER algorithm (I-PES) when processing slow streams (left), fast streams (right), short streams (top), long streams (bottom) on the movies dataset.

avoid pruning by prioritizing the most promising comparisons and delaying the least promising ones. However, straightforward adaptations of progressive ER suffer from the reassessment of the prioritization for every new increment (sketched in Figure 1 and observed in Figures 2 and 8).

We propose *progressive and incremental ER (PIER)* solutions to efficiently apply the progressive ER idea on incremental data to solve the issues presented above. To further illustrate the aforementioned limitations of the state-of-the-art and the performance of our contributions, in Figure 2, we compare one of our PIER algorithms (I-PES) against the schema-agnostic algorithms I-BASE [17] for incremental ER and two baseline adaptations of a progressive algorithm called PPS [36] to incremental data processing: PPS-GLOBAL considers the complete stream data to generate and prioritize comparisons, and PPS-LOCAL considers only the last increment of data for generating and prioritizing comparisons. Clearly, PPS-LOCAL performs poorly in all settings, barely finding any matches because it does not consider inter-increment comparisons. PPS-GLOBAL works well for slow streams ($0.1\Delta D/s$), but with faster streams (1 to $10\Delta D/s$) its performance degrades, especially for longer streams with 1000 of increments. The reason for this is that the prioritization is reassessed for each incoming increment - this assessment takes longer as more data are globally considered and needs to be repeated more often for longer streams. The incremental (but not progressive) I-BASE algorithm eventually finds the most matches, but does not exhibit a good progressive behavior where we want to find matches early. Our newly proposed I-PES algorithm outperforms all state-of-the-art algorithms.

Supporting progressive ER over incremental data offers some added value over existing ER solutions in a variety of practical applications. One example are anti-crime applications that use ER to identify potentially financial illicit activities[1][2]. These applications may benefit from PIER. Indeed, the earlier the illicit is detected (ideally, as soon as the data increment documenting some illicit activity arrives), the better, since follow-up problems or crimes may thus be prevented. We also experience the need for PIER in adaptive building and construction [23]. Here, matching data across architectural design, pre-fabrication, and monitoring data on the construction site (the latter two contributing incremental and streaming data) requires incremental processing. At the same time, matches found early based on progressive behavior allow for timely responses based on identified matches, which can be a factor in improving overall productivity. For instance, performing ER across parts of an architectural design (modelled in semi-structured formats such as IFC [5] or BHOM[3]), products on the construction site with properties extracted from monitoring data (imagery, point-clouds, sensors), and skill-based task descriptions and status of both on-site an prefabrication machines (expressed in another semi-structured format called AutomationML [12]) can help adaptively adjust the pre-fabrication (e.g., positioning for pre-drilling holes) of design parts (e.g., part of a wall) based on on-site factors (e.g., product alignment). In both these examples, data arrive incrementally and the earlier a match is found, the better (e.g., for faster crime discovery or better resource utilization). Furthermore, both examples may suffer from missed duplicates due to the comparison selection strategy of existing incremental ER approaches that allows faster processing of an increment at the price of not performing all promising comparisons. Note that this problem can be addressed by employing two ER pipelines: one online that processes newly arriving profiles incrementally, and one offline that runs on a timely basis for more accurate results. In this setting, our PIER algorithms improve the performance of the online ER pipeline, maximizing the benefit of finding duplicates early.

Contributions. We present a family of PIER algorithms. These are the first algorithms that progressively solve ER for dynamic data, which covers both incremental and streaming data. Thereby, we address the previously described shortcomings of existing approaches that either put unnecessary work into not so promising comparisons or spend too much time and resources in determining promising comparisons. Our algorithms stand on the shoulders of progressive strategies presented in [36] and extend the ER framework for dynamic data proposed in [17]. Our algorithms go beyond limitations of existing incremental ER approaches that uniquely rely on (incremental) blocking to choose the comparisons to execute for each input increment. Indeed, our methods (1) incrementally build a "global comparison index" of the best comparisons involving the profiles found, (2) progressively execute comparisons from this index until the next increment arrives, and (3) adaptively reassess the number of comparisons to generate for an increment based on how fast the overall ER pipeline can process the generated pairs. Our algorithms do not assume a fixed schema of incoming profiles and thus qualify as schema-agnostic ER (in line with, e.g., in [3, 24, 29, 36]). Experiments validate that our algorithms are the first to support incremental and progressive ER and, compared to state-of-the-art incremental approaches, improve early quality, eventual quality, and system efficiency by progressively and adaptively performing the unexecuted comparisons that are more likely to match when waiting for the next increment.

Structure. Section 2 provides the formal background and reviews relevant related work. Section 3 formalizes our novel PIER problem and describes the framework embedding our algorithms. The next three sections discuss different PIER algorithms. We discuss experiments in Section 7 and conclude in Section 8.

[1]https://www.financialtechnologytoday.com/are-entity-resolution-solutions-the-weapon-to-fight-financial-crimes
[2]https://www.consultancy.uk/news/19454/venture-arm-of-accenture-invests-in-financial-crime-specialist-quantexa

[3]https://github.com/BHoM

2 PRELIMINARIES AND RELATED WORK

In this section, we develop notation and we formally define batch ER (Section 2.1), progressive ER (Section 2.2), and incremental ER (Section 2.3). We review further related work in Section 2.4.

2.1 Batch ER

ER consists in finding the duplicates, aka matches, in a dataset, where each duplicate is a pair of profiles that refer to the same real-world entity. We denote by F_{batch} the batch ER method that obtains a set of duplicates M_D from a set of profiles D; that is $M_D = F_{batch}(D)$. For example, from a set of entity profiles $D = \{p_1, p_2, ..., p_{100}\}$ the output $M_D = \{(p_1, p_2), (p_6, p_{100}), (p_{20}, p_{63})\}$ indicates that, e.g., profiles p_6 and p_{100} refer to the same real-world entity (i.e., they are a match).

To determine if profiles p_i and p_j are a match or not, a comparison candidate $c_{i,j} = (p_i, p_j)$ is created and then evaluated by match function M, which may obtain $true$ if they are a match or $false$ otherwise. For example, $M(c_{6,100}) = true$ and $M(c_{1,6}) = false$. In practice, the function M employs a similarity function to assess the similarity of the two profiles (e.g., Jaccard or Edit Distance) and based on a similarity threshold, it classifies the pair as a match ($true$) or non-match ($false$). While the exact choice of the similarity function and threshold are not the focus of this paper (and are generally highly dependent on the actual application), different choices come with significantly different computational complexities, which influence the performance of F_{batch}.

Without further optimizations, the number of comparisons over a dataset D including n profiles is in $O(n^2)$. In practice, to significantly reduce the number of comparisons and scale to large datasets, blocking is commonly used. Blocking takes as input the dataset D and returns a block collection B_D. Essentially, input profiles are placed in the same block according to some criteria. A profile may be placed in multiple blocks as well. We denote by $B_D(p_i)$ all blocks of block collection B_D containing profile p_i. After blocking, only pairs of profiles within a same block are considered for comparison.

Considering heterogeneous data, i.e., data where profiles do not conform to a single structured schema, blocking further divides into three steps: block building, block cleaning, and comparison cleaning [29]. Block building techniques construct an initial block collection. Block cleaning techniques restructure the block collection by removing blocks or individual profiles from blocks. From these blocks, pairwise comparisons are generated and further pruned by a comparison cleaning technique.

2.2 Progressive ER

The main idea of progressive ER is to order the comparison candidates by their match-likelihood. Then, a matching function M is applied first on the pairs (comparisons) that are more likely to match. Notice that progressive ER still applies on batch data, but the goal is to discover the matches as early as possible. We denote by F_{prio} the progressive ER algorithm that obtains a set of duplicates $F_{prio}(D)$ from a set of profiles D. The idea behind progressive ER is to have the best possible partial result under fixed time/cost budget constraints. For instance, considering a time budget of t, we would like to get a set of duplicates $F_{prio}(D)[t]$ having more duplicates than its batch counterpart $F_{batch}(D)[t]$.

Definition 1 (Progressive ER [30, 39]). *Let D be a set of profiles. In comparison to a batch ER algorithm F_{batch}, a progressive ER algorithm F_{prio} satisfies the following two conditions:*

- Improved early quality: *Let t be an arbitrary target time smaller than the overall runtime of F_{batch} on D. Then, F_{prio} discovers more duplicates at time t than F_{batch}, i.e., $|F_{prio}(D)[t]| > |F_{batch}(D)[t]|$.*
- Same eventual quality: *When finished, the two algorithms discover the same duplicates, $F_{batch}(D) = F_{prio}(D)$.*

To satisfy the second condition, both F_{prio} and F_{batch} use the same blocking method (that selects the same pairs of profiles to compare) and the same match method (that for a given pair of profiles gives the same result $true$ or $false$). To satisfy the first condition, after blocking, two phases are needed. In the *initialization* phase, the algorithm builds the data structures needed to define the comparison order for the pairs based on their match-likelihood. In the *emission* phase, the best remaining pair is retrieved to be compared. The initialization phase is executed once in batch mode (corresponds to first flat part of curve in Figure 1), while the emission phase is executed each time a new comparison is requested or until depletion.

2.3 Incremental ER

The main idea of incremental ER is to find new duplicates whenever a new data increment comes as input, without recomputing from scratch the data structures needed for processing (e.g., the block collection), without repeating already executed comparisons, and without reconsidering again the already discovered duplicates. We denote by F_{incr} the incremental ER algorithm that obtains a set of duplicates $F_{incr}(D, \Delta D, B_D, M_D)$ from a set of profiles D and a new data increment ΔD, reusing the block collection B_D and the set of duplicates M_D from a dataset D. The idea behind incremental ER is to compute a set of duplicates $F_{incr}(D, \Delta D, B_D, M_D)$ with approximately the same quality of its batch counterpart $F_{batch}(D \uplus \Delta D)$, but much faster. To measure time efficiency, we introduce a function $T(\cdot)$ that takes as input a computation of an algorithm and returns its runtime. We define incremental ER, similarly as [18]:

Definition 2 (Incremental ER). *Let D be a set of profiles and ΔD an increment to it. In comparison to a batch ER algorithm F_{batch}, an incremental ER algorithm F_{incr} satisfies the following two properties:*

- *When $|\Delta D| << |D|$, computing F_{incr} should be much faster than F_{batch}, $T(F_{incr}(D, \Delta D, B_D, M_D)) << T(F_{batch}(D \uplus \Delta D))$.*
- *When finished, the two algorithms discover approximately the same duplicates, $F_{incr}(D, \Delta D, B_D, M_D) \approx F_{batch}(D \uplus \Delta D)$.*

Opposed to ER algorithms operating in batch mode, incremental ER needs to perform blocking incrementally whenever a new data increment arrives. For this reason, F_{incr} takes as parameter the existing block collection B_D that has previously been computed for D. Moreover, in order to not repeat work that already led to discover matches, the set of duplicates M_D is needed as parameter as well. Note that in the rest of this paper, to simplify notation, we will omit the parameters B_D and M_D by writing $F_{incr}(D, \Delta D)$ instead of $F_{incr}(D, \Delta D, B_D, M_D)$.

Let D_n be the dataset composed by the bag union of n data increments $\Delta D_1, \Delta D_2, ...\Delta D_n$. The application of F_{incr} over the n increments is denoted by

$$\overline{F}_{incr}(D_n) = \biguplus_{i=1}^{n} F_{incr}(D_{i-1}, \Delta D_i)$$

Notice that for $i = 1$, the initial dataset D_0, the block collection and the set of duplicates are empty. For $i > 1$, the block collection and the set of duplicates in the input are computed by the application of F_{incr} on the previous data increment ΔD_{i-1}.

Incremental ER applications need a minimum of memory to handle long streams of data and keep the necessary data structures in memory. They also should exhibit high throughput to process an increment before the next increment arrives.

2.4 Related Work

As surveyed in [4, 6, 8, 9, 28] a lot of work has been proposed for ER, ranging from methods for relational data, over parallelization approaches to scale ER to large data, to the application of deep learning approaches to ER in the matching step.

Progressive ER methods aim to prioritize useful work emitting possible matches as soon as possible. State-of-the-art approaches are mostly for relational data [1, 2, 30, 39]. Progressive methods based on an oracle feedback find application in crowdsourcing ER [14, 37, 38]. This type of methods re-adjust the processing order in function of the matcher's answer and differ from the first that define a static order independently of the matching step. An advanced blocking method named pBlocking [15] progressively refines the blocking results leveraging intermediate ER results from oracle-based methods such as [14].

Incremental ER is a challenging task, albeit necessary to perform ER in the presence of dynamic data. Several approaches to perform parts of incremental ER have been proposed [18, 19, 32–34]. All of these approaches require domain or schema knowledge, making them suitable only for relational data, and they do not trivially extend to highly heterogeneous data (e.g., Web data).

Schema mapping specifies a relationship between source and target schemas. Schema mapping has been widely studied, e.g., in [10, 20, 21, 31]. However, in the context of incremental ER and/or progressive ER, if the input data are highly heterogeneous (e.g., Web Data), schema mapping solutions may be extremely time consuming and unsuitable for processing ER under time constraints. Schema-agnostic solutions are therefore necessary in these settings and to the best of our knowledge the only work addressing progressive schema-agnostic ER is [36], while the only works addressing schema-agnostic incremental entity resolution are [3, 17].

Four schema-agnostic methods for progressive ER have been proposed in [36]. Local Schema-Agnostic PSN (LS-PSN) and Global Schema-Agnostic PSN (GS-PSN) are both non-trivial schema-agnostic adaptations of PSN [39]. Progressive Block Scheduling (PBS) sorts the blocks by size and the blocks are processed starting from the smallest. For each block, a weighting scheme of Meta-blocking is used to rank the comparisons and to define the order of emission. Progressive Profile Scheduling (PPS) uses a weighting scheme of Meta-blocking [25] to compute the likelihood of a profile to have one or more duplicates. Meta-blocking techniques [13, 25, 27, 35] build a *block graph* where the nodes are the profiles in D and an edge between p_i and p_j exists if they share at least on block. A variety of weighting schemes exist to give a weight to each edge, as well a variety of pruning techniques exist to remove the low-weighted edges. After applying Meta-blocking, only pairs of profiles that are directly connected by an edge are considered for comparison. In PPS, the profiles are processed starting from the profile with highest likelihood to the one with the lowest likelihood. For each profile, the top-k best non redundant comparisons involving that profile are emitted and then executed. All these approaches are not applicable to incremental data.

PI-Block [3] is a schema-agnostic blocking technique also based on Meta-blocking that supports both heterogeneous data and incremental processing. The framework presented in [17], which integrates specific techniques for incremental block cleaning and for incremental comparison cleaning, is the only solution that allows incremental ER for heterogeneous and possibly highly dynamic data such as high velocity streams.

3 PIER PROBLEM DEFINITION

This section first formalizes the problem of incremental and progressive ER, or PIER for short. We then describe the framework that we define to address this problem.

3.1 Problem definition

Considering dynamic data, let D be a (possibly empty) data set and S be a sequence of increments ΔD_1, ΔD_2, ..., ΔD_n coming at different times t_1, t_2, ..., t_n where $t_i < t_j$ for $i < j$. It is usually desirable to process the increment ΔD_i before ΔD_{i+1} arrives, i.e., the time to process ΔD_i (service time) should be less than or equal to the time $\delta t_i = t_{i+1} - t_i$ (interarrival time). In other words, it is desirable to process the increment ΔD_i under a time constraint, i.e., the *time budget* δt_i. If the desired time property does not hold, i.e., in scenarios where $\delta t_i < T(F_{incr}(D_{i-1}, \Delta D_i)))$, bottlenecks arise.

To run in this ideal setting, the question is how to select the work to be done in this limited time. Classical incremental ER usually limits itself to comparisons involving at least one profile of the current increment, i.e., for a given increment ΔD_i, they only generate pairwise comparisons involving at least one profile $p_j \in \Delta D_i$. A particular incremental ER algorithm F_{incr} determines the generated comparisons through blocking and possibly with a further refinement according to some match-likelihood criteria. This approach naturally limits the idea of progressive ER where the most promising comparisons come first, which may not be in the current increment - we may not have had time to process more promising ones of the previous increments. Clearly, this requires a more global view of promising comparisons while ensuring throughput and a strategy to adapt to varying interarrival times to best allocate work.

Our goal is to devise an ER method that, *considering dynamic input*, is able to incrementally find duplicates such that the incremental output will more quickly approximate the result for the current set of profiles. We define a progressive incremental ER method as follows.

Definition 3 (Progressive incremental ER). *Let the dataset $D_n = \Delta D_1 \uplus \ldots \uplus \Delta D_n$ be the dataset combining data increments $\Delta D_1, \ldots, \Delta D_n$ occurring at respective times $t_1, \ldots, t_n$. In comparison to a batch algorithm F_{batch}, a PIER algorithm F_{pier} satisfies the following four conditions:*

- Improved early quality: *Let t be an arbitrary target time such that $t_1 < t << T(F_{batch}(D_n))$. Then, $|\overline{F}_{pier}(D_n)[t]| > |F_{batch}(D_n)[t]|$.*
- Comparable eventual quality: *For sufficiently large t, the two algorithms discover approximately the same duplicates, i.e., $\overline{F}_{pier}(D_n) \approx F_{batch}(D_n)$.*
- Incrementality: *Let ΔD_i be an increment, $i > 1$. Then, it should hold (especially for large i) that*

$$T\left(F_{pier}\left(D_{i-1}, \Delta D_i\right)\right) < T\left(F_{batch}\left(D_i\right)\right)$$

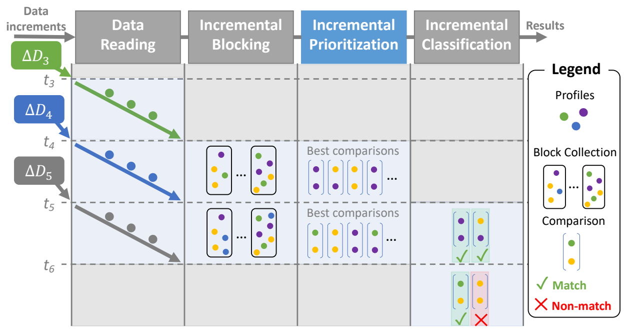

Figure 3: Framework for incremental progressive ER: temporal progression of five increments ΔD_1, ΔD_2, ΔD_3, ΔD_4, ΔD_5. Details in gray areas are omitted for simplicity. Profile colors indicate from which increment they originate.

- Globality: *Let ΔD_i be the increment that has become available most recently at a time $t, t_1 < t < t_{i+1}$. Then, up to time t_{i+1}, F_{pier} processes the best remaining pairs among profiles in $\Delta D_1 \uplus \ldots \uplus \Delta D_i$.*

The condition of improved early quality is adapted from the corresponding condition for progressive ER (see Definition 1). For the second condition on result quality, we follow the definition of incremental ER (see Definition 2) rather than requiring the exact same results as progressive ER (in batch mode), because we cannot expect an algorithm for progressive and incremental ER to perform the exact same comparisons (only in a different order) than a batch ER algorithm. Indeed, incremental processing typically requires "cutting corners" to counteract bottlenecks and keep internal data structures manageable in order to ensure the incrementality property (also adapted from Definition 2). The novel condition of globality formalizes that PIER needs to prioritize comparisons taking all profiles encountered so far into account. This means that executed comparisons for F_{pier} in the interarrival time δt_i may not involve profiles in ΔD_i as these are deemed unlikely to match. This is an advantage for both early and eventual quality, because the comparison candidates of (likely) duplicates that have not been considered yet (e.g., as fully processing the increments including their profiles would have taken longer than the interarrival time for these increments) get a chance to be found early, respectively at all. Meanwhile, this also allows to better balance the load of processing the full data. As long as there exist comparisons potentially yielding matches among all data, comparisons are continuously performed, irrespective to which increment their profiles belong to, which reduces idle time compared to existing incremental ER methods such as [17]. Indeed, if $T(F_{incr}(D_{i-1}, \Delta D_i)) << \delta t_i$, the system goes idle waiting for the next increment without performing useful work (this is somewhat visible in Figure 2 where the idle time corresponds to "steps" in the curve for slow streams). If data increments arrive too quickly, i.e., $T(F_{incr}(D_{i-1}, \Delta D_i)) > \delta t_i$, incremental ER delays the processing of the next increment and it will be slower to consume the entire stream (the reason why the incremental approach takes longer to reach the "final quality" for faster streams in Figure 2). Opposed to that, globality allows to put comparisons temporarily on hold when a new increment arrives.

3.2 Framework

Our framework for progressive and incremental ER for unstructured data, like any other ER framework, integrates functions for data reading, blocking, pair generation, and classification. Given that we intend to process incremental data, we rely on incremental versions of individual steps. In this work, we reuse components for incremental blocking and incremental classification of unstructured data proposed in [17], which correspond to the most recent state-of-the-art. However, instead of using comparison cleaning to select non-redundant pairs from a block collection, we introduce incremental prioritization, which implements novel techniques for the improved early quality and the globality conditions. This paper focuses on implementations of this component, nevertheless, to clarify how it integrates ER processing, we briefly summarize all components. To illustrate how the components work in the incremental setting, we use Figure 3. It shows the incremental processing over time of several increments at the different components (listed at the top). At time t_3, the first increments ΔD_1 and ΔD_2 have already "flowed through" the initial steps as ΔD_3 comes in.

Data Reading receives data increments ΔD_i at different points in time, denoted t_i. It is responsible for emitting profiles that describe real-world entities that are part of ΔD_i, which typically involves some data scrubbing or standardization. In Figure 3, at time t_3, the data reading component has received ΔD_3 and it emits tokenized entity profiles. The data reading step interfaces with the sources and is also responsible for managing the flow of the system, i.e., if the system is slow and the input rate is too fast, it buffers the increments or it slows down the sources.

Incremental Blocking takes as input profiles and outputs both the block collection it incrementally maintains and the last received data increment. To maintain the block collection, we first tokenize the values of input profiles and add a profile p to each block that corresponds to a token present in p. Complete blocks or profiles in blocks can then further be pruned, e.g., oversized blocks yielding an excessive number of comparisons are removed by block pruning [17]. Figure 3 illustrates that Incremental Blocking has finished the blocking process for ΔD_1, ΔD_2, and ΔD_3 at time t_5 and thus outputs the resulting block collection and ΔD_3.

As we shall see, when there is spare time as no further data increment arises, we integrate techniques to trigger the consideration of further pairs for prioritization. For this reason, Incremental Blocking periodically emits the block collection and an empty data increment to the next component, thus triggering the generation of the comparisons from older data.

Incremental Comparison Prioritization. We introduce this novel component to the ER pipeline to support progressive and incremental ER, therefore, describe further details here. Algorithm 1 summarizes its general processing steps. It takes as input a data increment and a (possibly empty) block collection, as received from the Incremental Blocking component. Eventually, it returns a list of comparisons to be compared next, i.e., that is passed as a batch to the incremental classification step. The algorithm can be set to a specific incremental prioritization strategy *IncrPrioritization*. This paper proposes three different implementations for *IncrPrioritization*, which are the subject of subsequent sections, where we discuss comparison-centric (Section 4), block-centric (Section 5), and entity-centric prioritization approaches (Section 6). The common goal of all these approaches is to maintain and update a global index *CmpIndex*,

Algorithm 1: Progressive incremental algorithm

```
Input: Increment $\Delta D$, block collection $B_{D \uplus \Delta D}$
Output: List of comparisons $CmpList$
Strategy: $IncrPrioritization$
1  $IncrPrioritization.updateCmpIndex(B_{D \uplus \Delta D}, \Delta D)$;
2  $CmpIndex \leftarrow IncrPrioritization.getCmpIndex()$;
3  $CmpList \leftarrow \emptyset$;
4  $K \leftarrow findK()$;
5  while $CmpIndex \neq \emptyset$ and $K > 0$ do
6      $c \leftarrow CmpIndex.dequeue()$;
7      $CmpList \leftarrow CmpList \cup \{c\}$;
8      $K \leftarrow K - 1$;
9  return $CmpList$;
```

which handles the global best comparisons and offers an interface with two operations: *dequeue()* retrieves and removes the best comparison from the index, while *enqueue($c_{i,j}$)* inserts a weighted comparison $c_{i,j}$ to the index. First, the algorithm updates the global index *CmpIndex* implemented by the specific *IncrPrioritization* strategy (line 1). Then, the algorithm retrieves *CmpIndex* (line 2). Next, as long as *CmpIndex* is non empty and we have not reached a previously determined maximum number K of comparisons to output (more on this later), the algorithm fills *CmpList* with the next best comparison (lines 5-8).The number K of comparisons returned is chosen dynamically according to the rate of the different components in order to promote a faster stream consumption if it can occur. In particular, a slow matcher implies lower K, a fast matcher implies higher K. We implement *findK()* by computing the input and processing rates as the average of their latest measurements. If the average input rate is lower than the system service rate, usually determined by the matcher (typical bottleneck in ER), it increases K (i.e., the matcher can perform more work). Otherwise, it decreases K (i.e., the matcher has to perform less work to facilitate a faster stream consumption).

In Figure 3, starting from t_5, the Incremental Prioritization component adds comparisons involving profiles of ΔD_3 to the *CmpIndex*, which defines an order of comparison execution. With K determined to be 2 in this example, it returns two comparisons in order of priority.

Incremental Classification. The last framework component receives a list of comparisons, which are processed in received order. More precisely, each comparison yields a classification result corresponding to the two compared profiles being duplicates or not. In our example of Figure 3, at time t_5, it classifies only pairs involving profiles of ΔD_1 and ΔD_2 because Incremental Prioritization for ΔD_3 is ongoing, while ΔD_4 is being processed by the Incremental Blocking component.

4 COMPARISON-CENTRIC PRIORITIZATION

The general idea of comparison-centric approaches is to produce a list of profile comparisons sorted in descending order of matching likelihood [9]. To measure the matching likelihood of a pair of entity profiles $c_{x,y} = (p_x, p_y)$ (i.e., a comparison) we use a weighting scheme $w(c_{x,y})$. In this paper, we consider the *Common Blocks Scheme (CBS)* as weighting scheme of meta-blocking [25], as it is the fastest to compute among the proposed alternatives while its adaptation to an incremental setting exhibited good overall performance [17]. CBS is defined as the number of blocks that p_x and p_y have in common, i.e., $w(c_{x,y}) = |B(p_x) \cap B(p_y)|$.

Algorithm 2: I-PCS.updateCmpIndex

```
Input: (i) Block collection $B_{D \uplus \Delta D}$, (ii) Increment $\Delta D$
Parameters: Parameter $\beta$
Global: Index $CmpIndex$
1  $CmpList \leftarrow \emptyset$;
2  foreach $p_x \in \Delta D$ do
3      $C_x \leftarrow \emptyset$;
4      $B_x \leftarrow B_{D \uplus \Delta D}(p_x)$;
5      $B_x \leftarrow BlockGhosting(B_x, \beta)$;
6      foreach $p_y \in b \wedge b \in B_x \wedge x \neq y$ do
7          $C_x \leftarrow C_x \cup \{c_{x,y}\}$;
8      $C_x \leftarrow$I-WNP$(C_x)$;
9      $CmpList \leftarrow CmpList \cup C_x$;
10 if $\Delta D = \emptyset \wedge CmpIndex = \emptyset$ then
11     $CmpList \leftarrow GetComparisons(B_{D \uplus \Delta D})$;
12 foreach $c_{x,y} \in CmpList$ do
13     $CmpIndex.enqueue(c_{x,y})$;
```

The comparison-centric approach for PIER that we propose is called *Incremental Progressive Comparison Scheduling (I-PCS)*. I-PCS uses a bounded priority queue to store the generated comparisons and its efficiency relies completely on the quality of the weighting scheme used, in our case an approximation of CBS. Experiments show that relying on this alone leads to poor performances, especially when matching is expensive. We propose a more effective approach addressing this problem in Section 6.

The *I-PCS* algorithm for maintaining and updating the global index *CmpIndex*, introduced in Section 3.2, is outlined in Algorithm 2. Here *CmpIndex* is a bounded priority queue returning as first element the comparison with highest weight. First, all the new comparisons that can be generated by a new profile p_x are inserted in a local comparison list C_x (lines 1- 7). The comparisons are generated from B_x, the set of blocks that include p_x, after the application of block ghosting [17], an incremental block cleaning method, that removes from B_x the least representative blocks for p_x. Block ghosting relies on a parameter β and it removes all the blocks $b \in B_x$ such that $b > \frac{|b_{min}|}{\beta}$ where b_{min} is the smallest block in B_x. Then, I-PCS calls the *I-WNP* algorithm proposed in [17] to update the weight of comparisons. In a nutshell, I-WNP is an incremental comparison cleaning algorithm that removes comparisons with low weight (below the average for the input comparison list) and weighs the remaining comparisons according to a weighting scheme, e.g., CBS. Applying I-WNP returns a refactored, weighted comparison list that will update the local list *CmpList* containing all the weighted comparisons for the increment (line 9). Finally, with this local list, the algorithm updates the *CmpIndex* (lines 12-13).

As we have introduced in Section 3.2, the blocking step periodically sends messages with empty increments if no new data are available from its input. Note that in case of an empty increment, the local comparison list *CmpList* will be empty and not used to update *CmpIndex*. However, if *CmpIndex* is non empty, Algorithm 1 will still be able to get the best remaining comparisons. If both the data increment ΔD and the *CmpIndex* are empty, I-PCS updates *CmpList* with the function $GetComparisons(B_{D \uplus \Delta D})$, that for each call takes comparisons from a block $b \in B_{D \uplus \Delta D}$,

Algorithm 3: I-PBS.updateCmpIndex

```
Input: (i) Block collection $B_{D \uplus \Delta D}$, (ii) Increment $\Delta D$
Global: (i) Index CmpIndex, (ii) Cardinality index $CI$,
        (iii) Profile index $PI$, (iv) Comparison filter $CF$
1  foreach profile $p_x \in \Delta D$ do
2      $B_x \leftarrow B_{D \uplus \Delta D}(p_x)$;
3      foreach $b \in B_x$ do
4          $CI(b) \leftarrow CI(b) + |b| - 1$;
5          $PI(b) \leftarrow PI(b) \uplus \{p_x\}$;
6  $b_{min} \leftarrow$ block in $CI$ with the smallest number of
       associated comparisons;
7  if $CmpIndex \neq \emptyset$ then
8      $\langle bsize, weight \rangle \leftarrow CmpIndex.top.weight$;
9  if $CmpIndex = \emptyset \lor bsize < |b_{min}|$ then
10     foreach $c_{x,y} \in \{(p_x, p_y) \mid p_x \in PI(b_{min}) \land x < y)\}$
         do
11         if $\neg CF.contains(c_{x,y})$ then
12             $CF.add(c_{x,y})$;
13             $c_{x,y}.weight = \langle |b_{min}|, w(c_{x,y}) \rangle$;
14             $CmpIndex.enqueue(c_{x,y})$;
       /* Reset the indexes                                  */
15     $CI(b_{min}) \leftarrow +\infty$;
16     $PI(b_{min}) \leftarrow \emptyset$;
```

from the smallest to the biggest. The idea of this behavior is based on continuing the computation even if the index becomes empty and the time budget is not yet exhausted.

5 BLOCK-CENTRIC PRIORITIZATION

Our second algorithm for comparison prioritization is based on the ordering of blocks. The general idea for such a block-centric prioritization is to sort the block collection in a descending order of likelihood that blocks contain duplicates [9]. Here, we propose an incremental version of *Progressive Block Scheduling (PBS)* [36]. PBS is based on the hypothesis that the smaller a block in a block collection, the more likely it contains duplicates. Based on this hypothesis, PBS sorts the blocks from the smallest to the largest. Then, it defines the processing order of comparisons inside each block according to a chosen schema-agnostic weighting scheme of Meta-blocking [25]. As we did before for comparison-centric prioritization, we consider CBS as weighting scheme.

Algorithm 3 provides pseudo-code for our *Incremental Progressive Block Scheduling (I-PBS)* algorithm. I-PBS relies on two globally maintained indexes: (1) a cardinality index CI that maps block identifiers b to the number of comparisons that can be generated by the unexecuted profiles in b, (2) a profile index PI that maps block identifiers to a list of unexecuted entity profiles. These indexes are respectively initializated such that each block identifier is associated with $+\infty$ and $\emptyset$, respectively. I-PBS accesses and updates the global index *CmpIndex* that we introduced in Section 3.2 in Algorithm 1. Again, *CmpIndex* is a bounded priority queue, but the weights it considers are pairs of values as opposed to single values considered in I-PCS. For each profile $p_x \in \Delta D$, the algorithm retrieves B_x, the set of blocks containing p_x (line 2). Then, for each block $b \in B_x$, it updates CI by adding to $CI(b)$, i.e., the entry in CI having as key the block identifier of b, the number of the new and unexecuted comparisons involving p_x. The algorithm also updates PI by adding to $PI(b)$ (analogous to $CI(b)$) the unexecuted profile p_x (lines 4- 5). Note that in case a block identifier b is not yet in PI (CI), the index automatically creates an entry $PI(b)$ ($CI(b)$) initialized to $\emptyset$ ($+\infty$). Next, I-PBS selects the block denoted b_{min}. This is the block yielding the least number of unexecuted comparisons, i.e., the block where $CI(b_{min})$ is minimal (line 6). Note that $CI(b_{min}) \leq ||b_{min}||$, where $||b_{min}||$ denotes the number of comparisons that can be generated from b_{min}. Then, if *CmpIndex* is non empty, the algorithm retrieves the current top comparison and its weight as a pair $\langle bsize, weight \rangle$ from *CmpIndex* (line 8). The value *bsize* is the size of the comparison's generating block when it has been inserted in *CmpIndex*, while the *weight* is a computation of CBS (see lines 13- 14). The *CmpIndex* prioritizes first on *bsize*, then on *weight*. Then, only if *Cmpindex* is empty or $bsize < |b_{min}|$ (why is explained later), the algorithm adds to *CmpIndex* all the non-redundant comparisons $c_{x,y}$ that can be generated based on b_{min} and where p_x is an unexecuted profile for that block as determined by PI (lines 10-14). A filter CF implemented as a scalable Bloom filter is used to check if a comparison $c_{x,y}$ is redundant or not, as shown in [16]. To indicate that b_{min} has been processed, it resets the entries in CI and PI for b_{min} (lines 15-16).

As mentioned before, I-PBS updates *CmpIndex* only when the comparisons generated in an earlier iteration have been exhausted or if the top weighted comparison in *CmpIndex* comes from a block with smaller size than b_{min}. We justify this choice as it avoids the comparison list becoming indefinitely large and to prefer comparisons that originated from smaller blocks. The *weight* of a comparison is used to prioritize comparisons with higher *weight* if they originated from the same block. This means that the first emitted unexecuted comparison $c_{x,y}$ for a block is the one with the higher $w(c_{x,y})$.

6 ENTITY-CENTRIC PRIORITIZATION

The main limitation of the comparison-centric approach is that its effectiveness and efficiency are not independent from the weighting method we have chosen. For example, *CBS* will prioritize comparisons between two entities sharing many tokens. It is possible that for many of these comparisons, these entities do not represent the same real-world entity but their descriptions are just very long. As we will see in Section 7 this may have very negative impact in performances, especially when using expensive matching functions.

Our third prioritization strategy considers entity-centric prioritization. In general, the main idea of entity-centric prioritization approaches is to provide a list of profiles that are sorted according to their duplication likelihood. Therefore, the comparisons for each profile are emitted following the order given by the list.

Here, we propose an incremental entity-centric prioritization approach that is inspired by *Progressive Profile Scheduling (PPS)* proposed in [36]. *PPS* employs a meta-blocking graph to compute the duplication likelihood of each profile (by aggregating the weights of its incident edges) and it creates a sorted profile list. During the construction of the sorted profile list, it also builds a comparison list where the top comparisons for each profile are stored and sorted. During the emission phase, *PPS* first emits the top comparisons from the comparison list, then it emits a bounded number of comparisons for each profile following the order given by the sorted profile list. PPS does not extend to the incremental setting because the incremental building, maintaining,

```
Algorithm 4: I-PES.updateCmpIndex
Input: (i) Block collection $B_{D \uplus \Delta D}$, (ii) Increment $\Delta D$
Parameters: Parameter $\beta$
Global: (i) Index $CmpIndex = \langle EntityQueue, E_{PQ}, PQ \rangle$,
        (ii) total number of generated comparisons $Count$,
        (iii) total sum of comparisons' weights $Total$
/* repeat lines 1-11 of Algorithm 2                    */
foreach $c_{x,y} \in CmpList$ do
    $w_{x,y} \leftarrow c_{x,y}.weight$;
    $\langle Total, Count \rangle \leftarrow \langle Total + w_{x,y}, Count + 1 \rangle$;
    if $E_{PQ}(p_x).top.weight < w_{x,y}$ then
        $E_{PQ}(p_x).enqueue(c_{x,y})$;
        $EntityQueue.enqueue(\langle p_x, w_{x,y} \rangle)$
    else if $E_{PQ}(p_y).top.weight < w_{x,y}$ then
        $E_{PQ}(p_y).enqueue(c_{x,y})$;
        $EntityQueue.enqueue(\langle p_y, w_{x,y} \rangle)$
    else if $w_{x,y} > \frac{Total}{Count}$ then
        $i \leftarrow \arg\min(|E_{PQ}(p_i)| \wedge i \in (x, y))$;
        $insert(c_{x,y}, p_i, E_{PQ}(p_i))$;
    else
        $PQ.enqueue(c_{x,y})$
    $CmpIndex \leftarrow \langle EntityQueue, E_{PQ}, PQ \rangle$
```

and updating of the meta-blocking graph is very costly, as shown in [17]. Moreover, the comparison list needs to be computed for every update.

Our algorithm, named *Incremental Progressive Entity Scheduling (I-PES)*, proposes a similar strategy in the incremental setting. However *I-PES*, as opposed to its batch counterpart, does not employ a meta-blocking graph for ranking the comparisons and it implements the profile list as a priority queue that can change over time. For this purpose, we employ three data structures: (1) an entity index E_{PQ} that maps each entity identifier p_x to a priority queue PQ_x containing weighted comparisons C_x, (2) an entity priority queue *EntityQueue* that stores tuples in the form $\langle p_x, w_x \rangle$ where w_x is the weight of the top comparison in PQ_x at the time of the tuple insertion, and (3) a bounded priority queue PQ that keeps low-weighted comparisons. These three data structures will constitute the *CmpIndex* for I-PES.

The *I-PES* approach is described in Algorithm 4. As we have seen in Algorithm 2, a weighted comparison list *CmpList* is created by using *I-WNP*. Then, if $w_{x,y}$ is higher than the weight of the current top comparison in $E_{PQ}(p_x)$, we update $E_{PQ}(p_x)$ with $c_{x,y}$ (line 5) and *EntityQueue* with $\langle p_x, w_{x,y} \rangle$ (line 6). Analogously, we update $E_{PQ}(p_y)$ in lines 7 -9. If $w_{x,y}$ is the lower of both the top comparisons, but higher than the average weight considering all the inserted comparisons, computed as $\frac{Total}{Count}$, first we select p_i between p_x or p_y such that the size of the priority queue $|E_{PQ}(p_i)|$ associated to p_i is the minimum (line 11). Then, we call the function $insert(c_{x,y}, p_i, E_{PQ}(p_i))$ (line 12). This function will insert $c_{x,y}$ in $E_{PQ}(e)$ only if its weight $w_{x,y}$ is higher than the average weight of all the comparisons involving e that have been inserted in $E_{PQ}(e)$. The reason why we introduced this double pruning is to reduce the memory overhead and remove further superfluous comparisons. The performance of the pruning depends on the weighting scheme used and thus on the token distribution. Experiments demonstrate good performances for I-PES across all real-world datasets we considered. Finally, if $w_{x,y}$ is lower than the total average weight $\frac{Total}{Count}$, it is inserted in a bounded priority queue *PQ*.

As said before, here the *CmpIndex* is essentially a wrapper for *EntityQueue*, E_{PQ}, and *PQ*. The operation $CmpIndex.dequeue()$, first retrieves (and removes) the top $\langle e, weight \rangle$ from *EntityQueue*, then it retrieves (and removes) the best comparison from $E_{PQ}(e)$.

If the *EntityQueue* becomes empty, for each entry e in E_{PQ} we add the tuple $\langle e, E_{PQ}(e).top.weight \rangle$ in *EntityQueue*. If the *EntityQueue* is smaller than K (see Algorithm 1) the missing comparisons are taken from *PQ*.

Table 1: Datasets characteristics

	Name	#Profiles	#Matches
D_{da}	dblp-acm	2.62k - 2.29k	2.22k
D_{movies}	movies	27.6k - 23.1k	22.8k
D_{2M}	synthetic	2M	1.7M
$D_{dbpedia}$	dbpedia	1.19M - 2.16M	892k

7 EVALUATION

In this section, we experimentally evaluate our PIER algorithms. First, we describe the experimental setup, then we make a detailed comparative analysis between our PIER algorithms and state of the art approaches. Our evaluation consider both a static setting and a dynamic setting, for which PIER algorithms are best suited.

7.1 Setup

Implementation. We implemented our methods in Scala and using the Akka Streams framework[4] (note however that our methods can be implemented in other data stream processing systems as well). The code is publicly available[5]. Baselines for progressive ER use methods of the JedAI framework (v3.1, Java 8)[6]. The baseline for incremental ER is implemented in Scala [17]. We run experiments on an OpenStack virtualized server with Ubuntu 18.04 (16 processors @ 2.30GHz, 50GB RAM).

Datasets and configurations. Table 1 summarizes the characteristics of the datasets we used. The datasets D_{da}, D_{movies}, and $D_{dbpedia}$ have been used extensively in related ER literature (e.g., in [17, 22, 26, 36]). These datasets include real-world collections suitable for Clean-Clean ER. In addition to these standard benchmark datasets, we use D_{2M}, an artificial dataset including census data which has been generated based on Febrl [7, 26]. This dataset is suitable for Dirty ER. All the datasets with ground truth files are available in a Mendeley repository[7] used to assess JedAI performance [26]. Observe that D_{ag}, D_{da} are quite small datasets, D_{movies} has moderate size, while D_{2M} and $D_{dbpedia}$ are by far the largest datasets.

While the PIER algorithms are general and independent from the match function used, they depend on how fast comparisons can be performed subsequently in the matching step (see Figure 3). We thus test them in two alternative pipeline configurations that use different match functions. The first match function employs a cheap similarity metric, namely the Jaccard similarity (JS), resulting in a matching step that is fast in consuming the pairs emitted by prioritization. Thus, in Algorithm 1, K is comparatively large.

[4] https://doc.akka.io/docs/akka/current/stream/index.html
[5] https://github.com/UniStuttgart-DataEngineering/progressive-incremental-er
[6] https://github.com/scify/JedAIToolkit
[7] https://data.mendeley.com/datasets/4whpm32y47/7

The second match function relies on the more expensive edit distance (ED). This simulates system behavior with small K.

Experiments and metrics. Our evaluation is related to the properties that an incremental and progressive ER algorithm F_{pier} should have in comparison to a batch algorithm F_{batch} (see Definition 3), i.e., early quality, eventual quality, incrementality, and globality. For the properties relating to quality, we quantify quality according to Pair Completeness (PC). PC is an established measure defined as the number of matches emitted by the blocking step (or prioritization step) divided by the size of the set of all the existing matches. A higher PC value indicates a better performance of a method. To study the progress of an algorithm, we consider both its progress over time or progress after number of comparisons.

Given that the four properties of the PIER problem are a "summa" of the properties we find in the definitions of progressive ER and incremental ER, we directly evaluate our solutions in each setting for comparative evaluation to state-of-the-art solutions.

Progressive setting. In progressive ER, we are interested in how fast matches are emitted over time, in other words the effectiveness over time. To study this, we plot the evolution of PC (vertical axis) with respect to the time (horizontal axis). We study how our methods compare to the progressive baselines "at their best", i.e., when these run on static data. The comparison focuses on improved early quality and eventual quality, given that these are the two relevant criteria for progressive ER (see Definition 1).

Incremental setting. In incremental ER, both incrementality and eventual quality are of concern. In this setting, we evaluate how the PIER algorithms compare to the state-of-the-art approach for incremental ER in incrementally processing a dataset. Questions we study are how fast the algorithms consume the streaming data and if their matching quality is *eventually* good. We further validate that straightforward adaptations of progressive ER algorithms to the incremental setting yield unsatisfactory results, by studying their performance in incremental settings where we vary the input rate of increments to be processed.

Baselines. In the progressive setting, for comparison to state-of-the-art solutions, we consider the two best methods for schema-agnostic progressive ER described in [36], namely PPS and PBS. For the incremental setting, we consider the best state-of-the-art approach described in [17], denoted as I-BASE (Incremental Baseline). We further consider simple adaptations of PPS and PBS to the incremental setting. As we have seen in the introduction when discussing Figure 2, the only reasonable adaptation are the GLOBAL variants, which we implemented here. We test all algorithms as part of ER pipelines with different similarity measures (JS and ED) to measure their behavior for different match overheads. Each algorithm supports both Dirty ER and Clean-Clean ER (the latter only considering pairs with a profile from each source).

7.2 Progressive setting

7.2.1 Experiment 1: progressive ER vs PIER. Here, we compare PPS and PBS with our algorithms in a progressive setting where all profiles in a dataset are available from the start (static dataset). This puts batch progressive ER solutions in their ideal situation, as they can consider all data to determine an optimal comparison order. Our algorithms, opposed to PPS and PBS, process data incrementally. Therefore, we divide the datasets into multiple increments that are processed sequentially one after the other. When not mentioned otherwise, we split D_{da} and D_{movies} into 1000 increments, while for D_{2M} and $D_{dbpedia}$, we create 20000 and 30000 increments, respectively. We choose these configurations to simulate long streams (a dedicated experiment studies the effect of varying the number of increments). In the progressive setting considered in this section, we study the evolution of PC over time and over number of comparisons performed. The evolution of PC over time gives insights into the efficiency of the different methods as well as the quality over time. Studying PC over performed comparisons provides further details on how much effort an algorithm wastes on comparisons not leading to matches. Our analysis focuses on the early quality and eventual quality of different approaches, which are, according to Definition 3, the main characteristics of progressive ER.

In Figure 4, we plot PC evolving over time for all our methods and baselines implementing progressive ER on the four datasets of Table 1. The first line shows results when JS applies during matching, while the results summarized in the second line use ED for comparisons. The plots set a time-budget of 5 minutes for the small and medium datasets, while for the larger D_{2M} and $D_{dbpedia}$ datasets, we set a maximum time-budget of 80 minutes. Figure 5 puts the number of comparisons on the x-axis and reports results for all algorithms until completion (so no time budget). Based on these results, we make the following observations.

PPS vs I-PES. As we have seen in Section 6, I-PES is based on similar ideas as PPS. In all experiments, we observe in Figure 5 that their eventual quality is comparable. However, they may reach the eventual quality at very different points in time. While on the small datasets D_{da} and D_{movies}, we see in Figure 4 that the performance over time is comparable, PPS requires an excessive time for initialization on the larger datasets (not even visible for $D_{dbpedia}$ as it took longer than 80 minutes). Consequently, on the large datasets, even in a static context, I-PES is clearly preferable over PPS in terms of early quality, while the difference (mere seconds) is negligible on smaller datasets. Note that this behavior is not impaired by the fact that I-PES may need to perform significantly more comparisons to reach the PC of PPS. For instance, in Figure 5 for $D_{dbpedia}$ using ED, we see that the line of I-PES is clearly below the line of PPS, still, we get a reasonable result after 80 minutes while for PPS, in our setting, the initialization alone takes more than 4 hours.

PBS vs. I-PBS. Next, we focus on the relative performance of the related algorithms PBS and I-PBS. When comparisons are fast (i.e., when using JS), we observe a comparable eventual quality for both algorithms, I-PBS being only at slight disadvantage. However, PBS requires much less preprocessing time than PPS on the large datasets, so its early quality is best. This does not come as a surprise, as, opposed to the incremental methods, considering the full dataset upfront finds a better global comparison prioritization compared to the incremental methods, that perform a varying number K of comparisons (see Algorithm 1). As a reminder, K adapts to the rate that pairs can be processed by the subsequent matcher. Thus, K is sufficiently large when using the cheap JS to still obtain satisfactory results. However, the results obtained when using the expensive ED for comparisons draw a different picture, as K may become too small and, for each increment, only few pairs even get a chance to be compared. We observe this effect both on D_{movie} and $D_{dbpedia}$. So overall, as the system gets "throttled" based on the matching step, I-PBS may become inadequate for progressive ER on a static dataset. Furthermore, while the number of comparisons are approximately the same

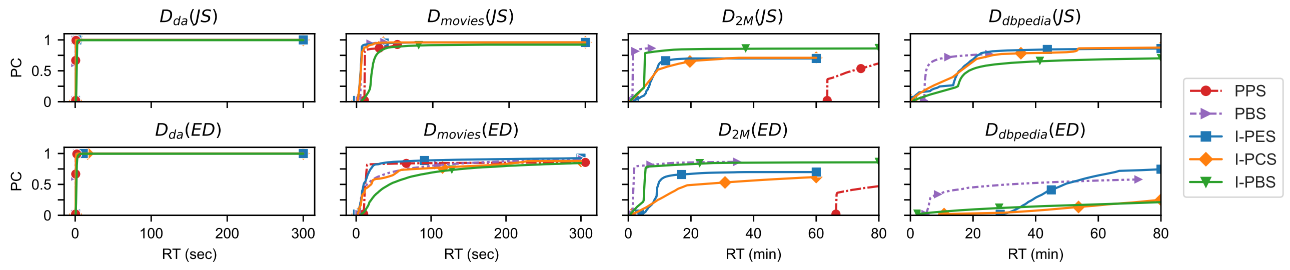
Figure 4: PC over time for D_{da}, D_{movies}, D_{2M} and $D_{dbpedia}$ in a batch setting.

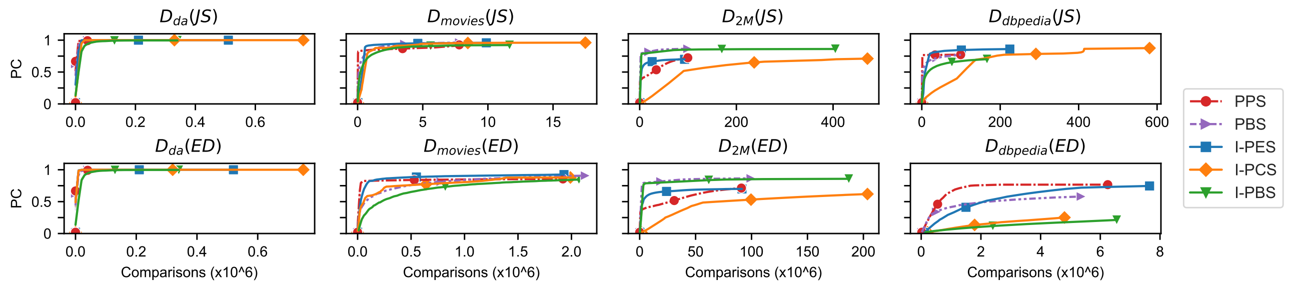
Figure 5: PC per emitted comparison for D_{da}, D_{movies}, D_{2M} and $D_{dbpedia}$ in a batch setting.

for both algorithms (see Figure 5), the effort spent on those is less well spent by I-PBS compared to PBS. The reason for this is that the effectiveness of I-PBS relies on the update frequency of *CmpIndex* (see Algorithm 3). When the index is empty, the unexecuted comparisons from the current smallest block are inserted into *CmpIndex*. Using a slow matcher implies consuming the comparisons in *CmpIndex* slower, missing the opportunity to execute better comparisons earlier.

I-PCS. Finally, our comparison-based approach I-PCS exhibits an acceptable eventual quality when using JS, in the sense that it ultimately reaches the same eventual quality as a baseline progressive ER method. Its early quality in these settings is also comparable to I-PES. We notice however that, compared to I-PES, it performs far more comparisons to reach this similar PC-over-time performance. One reason for this is that I-PCS's effectiveness relies on the rank method used, in this case CBS. Using CBS, pairs of profiles that share many tokens are prioritized. However, we observe that a lot of these pairs are just non-matches with long entity representations, resulting in more expensive and useless match computation. When considering the settings using ED, analogously to our observations concerning I-PBS, the performance of I-PCS degrades both in terms of early quality as well as eventual quality.

7.2.2 *Experiment 2: Influence of increment size.*

In this experiment, we vary the size of increments on our largest datasets. We expect that as we increase the size of increments, we can get a better order that comes closer to the "optimal" order of the progressive baselines. However, larger increments require more maintenance effort for the data structures the PIER algorithms use.

In Figure 6, considering $D_{dbpedia}$ and ED, we report results for 30000 increments of approximately 100 profiles ($|D_{dbpedia}/30000$) as well as 300 increments of size 10000. PC over time results are shown left, while PC over number of comparisons is shown on the right. The plots compare the behavior of I-PES and I-PBS to the behavior of their progressive baseline counterparts. Focusing on the PC over number of comparisons, we clearly see for I-PBS that with smaller number of larger increments, the

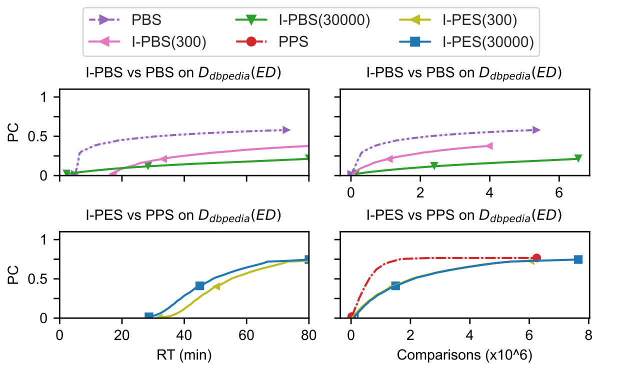
Figure 6: Influence of batch size on $D_{dbpedia}$. Alg(x) denotes algorithm *Alg* (either I-PBS or I-PES) on x increments.

comparison order determined becomes better, thus getting closer to the performance of PBS. This effect is not clearly visible for I-PES compared to PPS. The reason is that PPS needs 4 hours to find the good comparison order, while I-PES "does its best" with a budget of 80 minutes, where in this experiment, no significant improvements are reached for larger increments. Considering the evolution of PC over time, we observe that the price for a potentially better eventual quality is a longer pre-analysis on large increments.

7.2.3 *Summary for progressive setting.*

When performing progressive and incremental ER in a static setting where the full dataset is given upfront, our methods, when working with increments, perform differently depending on the match function used and the number of created increments. Overall, the I-PES method was the most useful of the incremental methods on the considered datasets, as it exhibits high eventual quality and adequate progression of early quality. Only exception is the D_{2M} dataset where I-PBS is superior. The reason here relies on the fact that D_{2M} is relational data (short and non-heterogeneous values), involving census data artificially generated. This implies that the

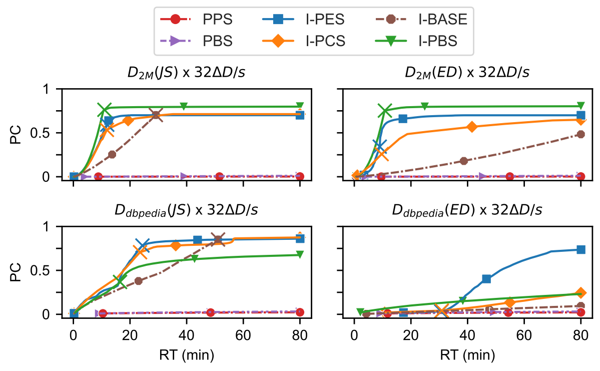

Figure 7: PC over time for D_{2M} and $D_{dbpedia}$ in an incremental setting with a fast stream ($32\Delta D/s$). If present, the symbol $\times$ on a line indicates the point where the stream has been fully consumed.

smallest blocks are highly informative blocks, thus rewarding block-centric approaches.

7.3 Incremental setting

So far, we have evaluated our PIER approaches in a setting they are not designed for, on static datasets. In this section, we now consider their performance in an incremental setting. Note that for space constraints, we only report results on the two large datasets.

7.3.1 Experiment 3: Comparative evaluation. We consider as baselines the adaptations of the progressive ER approaches PPS and PBS. Additionally, we compare our approaches to the incremental (but not progressive) approach I-BASE. To create increments, we split datasets into a given number of equi-sized increments (as before, 20000 and 30000 for D_{2M} and $D_{dbpedia}$, respectively). This experiment uses a fast rate at which the increments stream in, i.e., $32\Delta D/s$. We again assume a time budget of 80 minutes. In Figure 7, we report only the results for D_{2M} and $D_{dbpedia}$, for which we make the following observations.

Progressive ER adaptations. Clearly, the straightforward adaptations of PPS and PBS to the incremental setting are both not suited. Their PC remains close to 0 over the first 80 minutes. As explained in the motivation, this behavior is caused by the relatively high overhead times of PBS and especially PPS required to reorganize internal data structures needed to enable the progressive behavior.

Incremental baseline I-BASE. When focusing on I-BASE with JS, I-BASE gets the same eventual PC as I-PES and I-PCS. However, in terms of early quality, it lags behind all PIER algorithms. When looking at the results when using ED, we observe that the performance of I-BASE (as well as I-PCS and I-PBS, discussed later) significantly degrades - the effect being much more visible on $D_{dbpedia}$ than D_{2M}. For I-BASE, we observe that the stream can no longer be consumed within 80 minutes. The reason for this is that while the PIER algorithms are adaptive and select useful work according to the input rate and the system's response, I-BASE generates the same number of comparisons for a given input increment and existing block collection, independently of the input rate or the system's response. For this reason, when using ED, I-BASE gets stuck in useless match computations, delaying further processing.

PIER algorithms. Looking at the relative performance of the PIER algorithms, I-PES outperforms both I-PCS and I-PBS in terms of eventual quality for both JS and ED on all the datasets of Table 1. Only exception is the dataset D_{2M}, involving census data where smallest blocks are highly informative, where I-PBS performs better. On JS, I-PCS almost matches the early and eventual quality of I-PES, and the evolution of I-PBS is similar. However, when ED is used on a real dataset, the prioritization of I-PBS and I-PCS, guided mainly by CBS, degrades the comparison order by giving high priority not only to matches, but also to profile pairs with long representations. These useless comparisons are very expensive when ED is used, contributing to a drastic drop in performance.

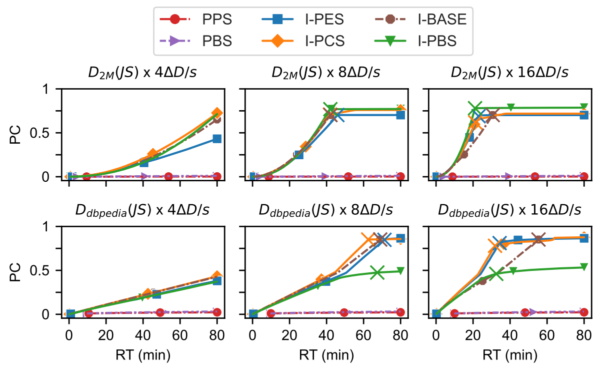

Figure 8: PC over time for D_{2M} and $D_{dbpedia}$ in an incremental setting. Four streams with four different rates are considered. If present, the symbol $\times$ indicates the point where the stream has been fully consumed.

7.3.2 Experiment 4: Varying the increment input rate. This experiment reuses the same setup as Experiment 3, however, we vary the input rate. We report results for D_{2M} and $D_{dbpedia}$ for input rates of $4\Delta D/s$, $8\Delta D/s$, and $16\Delta D/s$. We summarize the results in Figure 8. From these, we make the following observations.

Progressive ER adaptations. Even for the slowest input rate, we do not observe good performance for PPS and PBS. This validates that in larger incremental settings, they do not apply.

Incremental ER approaches. We observe that for JS, the benefit of using the PIER approaches on early quality as opposed to I-BASE increases when the input rate increases. On the slower streams (4 and 8 $\Delta D/s$), I-BASE can keep up with processing the profiles streaming in, reaching comparable performance to the PIER algorithms. But as the input rate increases, I-BASE can eventually not keep up and "stagnates". Opposed to that, the PIER algorithms are adaptive to different input rates. Of course, for slow streams where all (scarce) matches are found before the next increment arrives, PC only increases slowly for all approaches, while a clear benefit on early quality becomes evident on faster streams. Considering the results employing ED, this effect is also visible, even though all approaches suffer from the expensive ED, leading to a much slower increase of PC over time. I-PES on the slowest stream for D_{2M} suffers of large overheads when its *EntityQueue* becomes empty too many times.

7.3.3 Summary for incremental setting. Clearly, in an incremental setting, our PIER algorithms outperform any of the considered baselines in terms of early quality as well as eventual quality (validated for all experiments that did not time out). Given that they can compete with I-BASE that was validated in terms of incrementality in [17], we can further conclude that they satisfy incrementality as well. As for globality, it is built into the prioritization algorithms, contributing to improved performance with respect to state of the art incremental solutions by allowing a more "global" prioritization of comparisons. Among the PIER algorithms, our evaluation indicates that I-PES is in general the best algorithm, by being less sensitive to a particular weighting scheme or matching function.

8 CONCLUSION

We introduced the PIER problem, i.e., progressive and incremental entity resolution. Focusing on heterogeneous data, after showing how to extend an existing schema-agnostic incremental ER pipeline with a prioritization component necessary for progressive behavior, we presented three algorithms to implement this prioritization component. Experiments on several benchmark datasets validate that our approaches are the only ones to work in an incremental setting, while offering both improved early quality and earlier eventual quality compared to state-of-the-art incremental approaches. Among the three alternatives presented, the entity-centric approach I-PES appears to be the method of choice, especially for long and fast streams, as it manages to compensate poor performance of weighting schemes (used by all algorithms) when these are not optimally chosen for a dataset. Future work includes the integration of a heuristic for determining the best appropriate method to use for the given data.

ACKNOWLEDGMENTS

Partially supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2120/1 - 390831618. Partially funded by the Ministry of Science of Baden-Württemberg, Germany, for the Doctoral Program 'Services Computing'.

REFERENCES

[1] Y. Altowim, D.V. Kalashnikov, and S. Mehrotra. 2014. Progressive approach to relational entity resolution. *Proceedings of the VLDB Endowment (PVLDB)* 7, 11 (2014), 999–1010.

[2] Y. Altowim and S. Mehrotra. 2017. Parallel progressive approach to entity resolution using mapreduce. In *IEEE International Conference on Data Engineering (ICDE)*. 909–920.

[3] T.B. Araújo, K. Stefanidis, C.E. Santos Pires, J. Nummenmaa, and T.P. da Nóbrega. 2020. Schema-agnostic blocking for streaming data. In *ACM Symposium on Applied Computing*. 412–419.

[4] N. Barlaug and J.A. Gulla. 2021. Neural Networks for Entity Matching: A Survey. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 15, 3 (2021), 1–37.

[5] A. Borrmann, J. Beetz, C. Koch, T. Liebich, and S. Muhic. 2018. Industry foundation classes: A standardized data model for the vendor-neutral exchange of digital building models. In *Building Information Modeling*. Springer, 81–126.

[6] X. Chen, E. Schallehn, and G. Saake. 2018. Cloud-scale entity resolution: current state and open challenges. *Open Journal of Big Data (OJBD)* 4, 1 (2018), 30–51.

[7] P. Christen. 2008. Febrl- an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)*. 1065–1068.

[8] P. Christen. 2011. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 24, 9 (2011), 1537–1555.

[9] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. 2020. An overview of end-to-end entity resolution for big data. *Comput. Surveys* 53, 6 (2020), 1–42.

[10] D. De Una, N. Rümmele, G. Gange, P. Schachte, and P.J. Stuckey. 2018. Machine Learning and Constraint Programming for Relational-To-Ontology Schema Mapping.. In *International Joint Conference on Artificial Intelligence (IJCAI)*, Vol. 2018. 27th.

[11] X.L. Dong and D. Srivastava. 2015. *Big Data Integration*. Morgan & Claypool Publishers.

[12] R. Drath, A. Luder, J. Peschke, and L. Hundt. 2008. AutomationML-the glue for seamless automation engineering. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 616–623.

[13] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Information Systems* 65 (2017), 137–157.

[14] D. Firmani, B. Saha, and D. Srivastava. 2016. Online entity resolution using an oracle. *Proceedings of the VLDB Endowment (PVLDB)* 9, 5 (2016), 384–395.

[15] S. Galhotra, D. Firmani, B. Saha, and D. Srivastava. 2021. Efficient and effective ER with progressive blocking. *The VLDB Journal* 30, 4 (2021), 537–557.

[16] L. Gazzarri and M. Herschel. 2020. Boosting Blocking Performance in Entity Resolution Pipelines: Comparison Cleaning using Bloom Filters. In *International Conference on Extending Database Technology (EDBT)*. 419–422.

[17] L. Gazzarri and M. Herschel. 2021. End-to-end Task Based Parallelization for Entity Resolution on Dynamic Data. In *IEEE International Conference on Data Engineering (ICDE)*. 1248–1259.

[18] A. Gruenheid, X.L. Dong, and D. Srivastava. 2014. Incremental record linkage. *Proceedings of the VLDB Endowment (PVLDB)* 7, 9 (2014), 697–708.

[19] D. Karapiperis, A. Gkoulalas-Divanis, and V.S. Verykios. 2018. Summarization Algorithms for Record Linkage.. In *International Conference on Extending Database Technology (EDBT)*. 73–84.

[20] A. Kimmig, A. Memory, and L. Getoor. 2017. A collective, probabilistic approach to schema mapping. In *IEEE International Conference on Data Engineering (ICDE)*. 921–932.

[21] A. Kimmig, A. Memory, R.J. Miller, and L. Getoor. 2018. A collective, probabilistic approach to schema mapping using diverse noisy evidence. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 8 (2018), 1426–1439.

[22] H. Köpcke, A. Thor, and E. Rahm. 2010. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment (PVLDB)* 3, 1-2 (2010), 484–493.

[23] Nico Lässig, Melanie Herschel, Alexander Reichle, Carsten Ellwein, and Alexander Verl. 2022. The ArchIBALD data integration platform: Bridging fragmented processes in the building industry (to appear). In *CAISE Forum*.

[24] G. Papadakis, G. Alexiou, G. Papastefanatos, and G. Koutrika. 2015. Schema-Agnostic vs Schema-Based Configurations for Blocking Methods on Homogeneous Data. *Proceedings of the VLDB Endowment (PVLDB)* 9, 4 (2015), 312–323.

[25] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. 2013. Meta-blocking: Taking entity resolutionto the next level. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 26, 8 (2013), 1946–1960.

[26] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. 2020. Three-dimensional entity resolution with jedai. *Information Systems* 93 (2020), 101565.

[27] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. 2016. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking.. In *International Conference on Extending Database Technology (EDBT)*. 221–232.

[28] G. Papadakis, D. Skoutas, E. Thanos, and T. Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *Comput. Surveys* 53, 2 (2020), 1–42.

[29] G. Papadakis, J. Svirsky, A. Gal, and T. Palpanas. 2016. Comparative analysis of approximate blocking techniques for entity resolution. *Proceedings of the VLDB Endowment (PVLDB)* 9, 9 (2016), 684–695.

[30] T. Papenbrock, A. Heise, and F. Naumann. 2014. Progressive duplicate detection. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 27, 5 (2014), 1316–1329.

[31] L. Qian, M.J. Cafarella, and H.V. Jagadish. 2012. Sample-driven schema mapping. In *ACM International Conference on Management of Data (SIGMOD)*. 73–84.

[32] B. Ramadan, P. Christen, H. Liang, and R.W. Gayler. 2015. Dynamic sorted neighborhood indexing for real-time entity resolution. *Journal of Data and Information Quality (JDIQ)* 6, 4 (2015), 1–29.

[33] B. Ramadan, P. Christen, H. Liang, R.W. Gayler, and D. Hawking. 2013. Dynamic similarity-aware inverted indexing for real-time entity resolution. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*. 47–58.

[34] W. Ren, X. Lian, and K. Ghazinour. 2021. Online Topic-Aware Entity Resolution Over Incomplete Data Streams. In *ACM International Conference on Management of Data (SIGMOD)*. 1478–1490.

[35] G. Simonini, S. Bergamaschi, and H.V. Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *Proceedings of the VLDB Endowment (PVLDB)* 9, 12 (2016), 1173–1184.

[36] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi. 2019. Schema-Agnostic Progressive Entity Resolution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 31, 6 (2019), 1208–1221.

[37] N. Vesdapunt, K. Bellare, and N. Dalvi. 2014. Crowdsourcing algorithms for entity resolution. *Proceedings of the VLDB Endowment (PVLDB)* 7, 12 (2014), 1071–1082.

[38] J. Wang, T. Kraska, M.J. Franklin, and J. Feng. 2012. CrowdER: Crowdsourcing Entity Resolution. *Proceedings of the VLDB Endowment (PVLDB)* 5, 11 (2012), 1483–1494.

[39] S.E. Whang, D. Marmaros, and H. Garcia-Molina. 2012. Pay-as-you-go entity resolution. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 25, 5 (2012), 1111–1124.