End-to-end Task Based Parallelization for Entity Resolution on Dynamic Data

1st Leonardo Gazzarri
University of Stuttgart, Germany
leonardo.gazzarri@ipvs.uni-stuttgart.de

2nd Melanie Herschel
National University of Singapore, Singapore
University of Stuttgart, Germany
melanie.herschel@ipvs.uni-stuttgart.de

***Abstract*—Entity resolution (ER) is the problem of finding which digital representations of entities correspond to the same real-world entity. In many Big Data scenarios, in addition to the problems of volume and variety that are commonly addressed in ER, data is continuously generated, which requires novel solutions to address the velocity problem.**

This paper presents a framework for *end-to-end ER* that *incrementally* and *efficiently* produces results as heterogeneous data streams in. These characteristics are achieved by proposing a novel *functional model* for ER on incremental or streaming data, and adopting *task-based parallelization*. Our evaluation demonstrates that even without parallelization, our framework outperforms state-of-the-art (batch) ER in terms of runtime and quality. We also validate that it can achieve high throughput and low latency on streaming data, paving the way to real-time ER.

***Index Terms*—entity resolution, streaming data, parallelization**

I. Introduction

Entity resolution (ER) is the problem of identifying so called *matches*, i.e., digital representations of entities that correspond to the same real-world entity. The problem is well-known in data cleaning and integration [6]. It finds further application in the Internet domain where entities need to be resolved in large volumes of heterogeneous and semi-structured data. A classical example of large-scale ER targeting these *heterogeneous data* are meta-engines matching product descriptions from many web shops to provide price comparisons [16].

Figure 1 illustrates a typical ER pipeline for heterogeneous data. It divides ER into *data reading*, *blocking*, *comparison*, and *classification*. Essentially, data reading standardizes entity descriptions, e.g., applying word stemming, consistently using same abbreviations, etc. The standardized entity descriptions are then clustered into blocks during blocking. The idea of blocking is to reduce the quadratic complexity of comparing all entity descriptions to each other during the classification step. Indeed, when entity descriptions are divided into a *block collection* as a result of blocking, only those within a same block are paired together for further processing. Comparison assigns to each pair of considered entity descriptions a similarity score. Based on these, classification determines if the pairs of entity descriptions are matches or non-matches.

When processing heterogeneous data, the blocking step actually divides into *block building*, *block cleaning*, and *comparison cleaning* [20], as illustrated in the bottom part of Figure 1. Block building constructs an initial block collection.

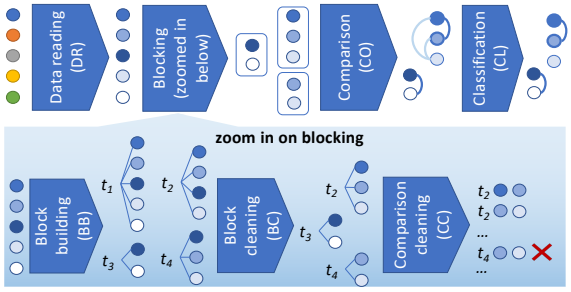

Fig. 1. Traditional workflow of ER in heterogeneous domain.

Blocks or entity descriptions within blocks are pruned during block cleaning. From these blocks, pairwise comparisons are generated and subjected to comparison cleaning to further reduce the number of pairs entering the comparison step.

Existing methods that implement (parts of) the ER pipeline primarily focus on batch processing large volumes of heterogeneous data. This means that they assume complete (static) datasets to run the ER pipeline once. In this paper, we study the problem of ER when dealing with large volumes of *dynamic data*. We consider both the case of (1) *incremental ER* where a finite dataset changes periodically so we aim at maintaining the full ER result incrementally and (2) *streaming ER* where a possibly infinite stream of entity descriptions is processed.

Solutions for incremental and streaming ER over large volumes of heterogeneous data have many potential practical applications [6], [11]. For instance, the previously mentioned meta-engines clearly handle dynamic data. Another application arises when digitizing and streamlining data across multiple phases of manufacturing or building processes, such as in digital design and construction for the building industry. In this context, ER needs to be applied among frequently changing or newly added representations of architectural designs, prefabricated building components, or construction site characteristics.

Clearly, there is a need for ER that supports dynamic data, a gap this paper fills by presenting an end-to-end framework for ER over dynamic data. Its definition targets efficiency and quality for incremental ER, and additionally optimizes latency and throughput for streaming ER.

To improve the efficiency when performing ER on large volumes of data, different solutions explore parallelizing ER. These mostly focus on data parallel solutions that parallelize steps of the ER pipeline [7], [13], [14], [17]. These solutions

all target batch processing. To the best of our knowledge, the only work that parallelizes the single step of blocking when processing increments of data is [1]. Overall, end-to-end ER of dynamic data using task parallelism is an open problem. In general, task-based parallel approaches allow different segments of the stream to be executed in parallel, typically improving runtime and throughput by keeping low latency.

Contributions. We propose a task-based parallel framework for end-to-end ER targeting heterogeneous data and supporting incremental and streaming ER. Achieving task-parallelism requires us to pipeline entity descriptions through the existing and necessary steps of ER summarized in Figure 1. To this end, we model each ER step as a function. The ER pipeline is then defined as a function combination. Functions apply on a single entity description at a time. That is, our first contribution is a functional model for ER on dynamic data. Next, we investigate how to incorporate task parallelism into an ER framework for dynamic data that conforms to our model. In particular, this requires the definition of alternative algorithms to those traditionally implementing block cleaning and comparison cleaning, as these do not fit the parallelization paradigm. It further requires careful balancing of the different (parallel) steps of ER to mitigate bottlenecks. Extensive evaluation validates that our framework successfully allows efficient and effective ER on dynamic data, reaching close to real-time latency and high throughput. It further scales to large sizes of ER problems where it has comparable or improved ER quality while running up to 100 times faster than state-of-the-art approaches.

This paper has its roots in a vision paper [10]. The present paper significantly extends this vision paper, which did not focus on streaming data, had no implementation and evaluation, did not formalize a clear functional model for ER on dynamic data, and did not investigate several adaptations of algorithms to dynamic data and optimizations for good overall performance.

Structure. Section II discusses preliminaries on the state-of-the-art ER pipeline shown in Figure 1 and summarizes relevant related work. Section III describes our functional model for ER on dynamic data. Section IV describes the task-parallel framework. Section V presents our implementation and experimental evaluation. We conclude with Section VI.

II. Preliminaries and Related Work

This section first presents the different ER steps that compose the pipeline depicted in Figure 1. It then reviews related work.

A. General ER pipeline

Data reading (DR) consists in retrieving data from one or multiple sources, typically applying some preprocessing to ease the subsequent steps (e.g., tokenization or data standardization). Depending on the number and quality of sources input to DR, different variants of the ER problem arise: (1) dirty ER considers that any of the input sources may itself include matches, whereas (2) clean-clean ER assumes that matches can only arise across sources. Figure 2(a) illustrates dirty ER data in various formats that could coexist in a data lake integrating heterogeneous data from the building sector. We assume that

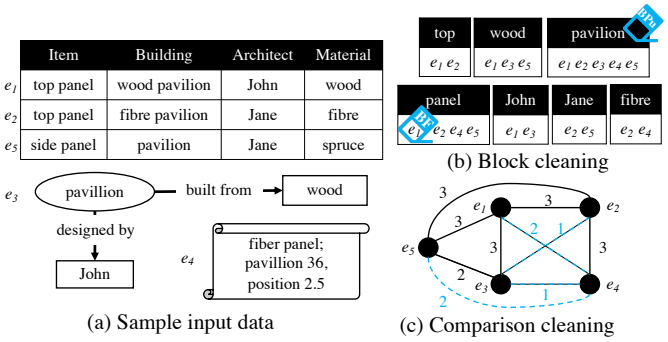

Fig. 2. Illustrative example for different ER steps

standardization transforms e_4's "fiber" token to British English and changes e_5's material to the more general term "wood".

Block building (BB) aims at reducing the total number of comparisons by grouping pairs of representations in blocks according to a set of features (as surveyed in [3], [20]), to then only compare entity representations that fall in a same block. When entity representations highly vary in their structure (e.g., no fixed schema and thousands of attributes that may be scarcely used), classical attribute-based blocking used for relational data does not apply. A simple and commonly used method to deal with this high schema-heterogeneity is *token blocking* [20] that creates a block b_k for each token k appearing in the preprocessed values of at least two entities. Essentially, this gives each pair of entity descriptions that share at least one standardized token the chance to be compared, despite missing or contradictory data. Figure 2(b) (disregarding the blue annotations) shows the block collection after applying token blocking on the full dataset comprising e_1 to e_5 of Figure 2(a). Note that previous standardization allows e_4 (resp. e_5) to appear in the block with token "fibre" (resp. "wood").

Even in this small example, the main problem of token blocking shows: indeed, "naive" ER would simply compare all five entity representations shown in Figure 2(a) with each other, resulting in 6 pairwise comparisons (assuming a symmetric similarity measure). Using the block collection depicted in Figure 2(b) would require 23 comparisons. In real ER scenarios, the total number of comparisons after token blocking (or any other method that use schema-free keys for block building) easily exceeds the theoretical worst-case quadratic number of comparisons by orders of magnitude [9]. To re-establish the rationale of using blocking, i,e., significantly reduce the number of pairwise comparisons, further post-processing methods are integrated as part of the blocking step of the ER pipeline. These methods pursue the following goals: (1) avoid *redundant* comparisons, for instance compare e_1 and e_3 at most once, instead of the three times the block collection of Figure 2(b) would require and (2) reduce *superfluous* comparisons that are unlikely to result in a match (e.g., e_2 and e_3).

Block cleaning (BC). Among the block post-processing methods, block cleaning prunes the block collection to reduce the number of comparisons. Two main approaches for BC are *block purging* and *block filtering*.

Block purging (BPu) removes blocks that are considered

oversized and thus too general, yielding many superfluous and also likely redundant comparisons. Oversized blocks are identified using a parameter r ($0 < r < 1$) and b_{max}, largest block in the block collection B. More specifically, BPu removes all the blocks $b \in B$ with size $|b| > r \cdot |b_{max}|$.

Block filtering (BF) removes entity descriptions from blocks. For an entity description e_i, it first retrieves the set of blocks B_{e_i} within B that e_i has been assigned to during block building. Given a parameter s, where $0 < s < 1$, BF retains e_i in the $\lfloor s \cdot |B_{e_i}| \rfloor$ smallest blocks, deleting it from the remaining larger blocks. The rationale is to remove e_i from blocks where it is more likely that the comparisons are superfluous.

As indicated by the blue annotations in Figure 2(b), BPu would for instance remove the block defined by token "pavilion", assuming $r = 0.8$. Subsequently applying BF removes e_1 from the block "panel", assuming, $s = 0.8$ (other descriptions are removed too, for conciseness and better readability we do not report them).

Comparison cleaning (CC). The second post-processing step to further reduce the number of pairwise comparisons resulting from a block collection is comparison cleaning. For this step, *meta-blocking* is commonly used [18], [19]. The ideas underlying meta-blocking are illustrated in Figure 2(c). First it builds a blocking graph from a block collection, where each node represents an entity description e_i and an edge (e_i, e_j) between two nodes exists only if e_i and e_j co-occur in at least one block. Each edge is weighted according to a *weighting scheme*. Then, meta-blocking prunes the low-weighted edges according to a *pruning scheme*. In [18], five general weighting schemes, as well as four pruning schemes are proposed. For the graph shown in Figure 2(c), we consider the block collection given in Figure 2 (without performing BC) as input and we use the *Common Blocks Scheme (CBS)* as weighting scheme. It simply counts the number of blocks in which e_i and e_j co-occur. Each edge in the graph is considered a pairwise comparison. So clearly, at this stage, building the graph has already pruned redundant comparisons. To also prune superfluous comparisons, further edges are pruned from the graph. The dashed blue edges in Figure 2(c) illustrate which edges the *Weight Node Pruning* (WNP) scheme removes. In essence, given a node e_i, it first computes a local threshold defined as the average edge weight of its adjacent edges. It then retains only the adjacent edges having a weight higher than the local threshold.

Comparison (CO). The result of the blocking step, which comprises BB, BC, and CC, is used to form the pairs of entity descriptions input to the comparison step. In this step, each pair of entity descriptions (e.g., (e_1, e_2), (e_1, e_3), (e_1, e_5), (e_2, e_4), (e_2, e_5) and (e_3, e_5)) is compared based on a similarity measure.

Classification (CL). The computed similarities form the basis for the final classification of entity descriptions as matches or non-matches, which is the task of the classification step. A common strategy is to classify pairs as matches if their similarity is above a given threshold. In our example, we assume that both (e_1, e_3) and (e_2, e_4) are identified as matches.

B. Related Work

We now summarize work related to scaling ER through parallelization and seminal work in ER for dynamic data.

ER parallelization. To scale to large volumes of data, parallelism in ER has been explored, as surveyed in [2]. In particular, data parallel solutions (map-reduce based) have been proposed for meta-blocking [8], [17]. These parallelization strategies do not naturally support incremental or streaming ER. Therefore, we shift to a task-based parallel solution able to process a high number of entity descriptions in a given time.

To the best of our knowledge, besides our vision paper [10] previously mentioned, only [25] explores task-based parallelization in ER. [25] focuses only on relational data and does not integrate blocking techniques necessary for heterogeneous data. That is, this paper is the first to propose a practical end-to-end parallel framework for ER able to process dynamic data.

ER for dynamic data. To cope with dynamic data, approaches have been proposed to perform (parts of) ER incrementally [11], [12], [22]–[24]. However, they target relational data and do not trivially extend to ER on heterogeneous data, in particular to the different blocking steps it requires. Once matching pairs are found, approaches to incrementally cluster these to form groups of descriptions of the same entity have been proposed [5], [11], [24]. We consider these complementary to our approach, as they typically consume pairs as output by our framework.

The research most similar to ours is a schema-agnostic blocking technique, named PI-Block [1]. it is based on meta-blocking and supports both heterogeneous data and incremental processing, leveraging parallelization using Apache Spark. PI-Block operates on large increments of data and does not apply to "real-time" stream processing. In contrast to PI-Block, we provide an end-to-end ER approach for dynamic data rather than focusing on a single step. We further consider the timeliness of results to effectively support streaming data, where each input entity description should be processed as fast as possible.

III. A FUNCTIONAL MODEL FOR ER ON DYNAMIC DATA

This section presents the formalization of a functional model for ER on dynamic and heterogeneous data. We have seen in the introduction that two variants of ER are often distinguished, namely dirty ER and clean-clean ER. We also target two types of dynamic data: incremental data considers a finite dataset that is incrementally updated, whereas streaming data is a possibly endless stream of individual entity descriptions.

The discussion of our functional model first elaborates on dirty ER and incremental data in Section III-A. We then extend the functional model for clean-clean ER and streaming data in Sections III-B and III-C, respectively. Table I summarizes notations that we frequently use in the paper.

A. Model for incremental dirty ER

As input, we consider an entity description e_i of a dataset $D = \{e_1, ..., e_n\}$. We denote with p_i the profile of entity description e_i that is the result of standardization as performed by the data reading step. An entity description e_i (and profile p_i) is uniquely identified by i that can be an identifier or a URI.

TABLE I
SUMMARY OF NOTATIONS

Notation	Meaning
e_i, p_i	An entity description and its standardized profile
$D = \{e_1, \ldots, e_n\}$	A finite set of entity descriptions
$D_S = [e_1, e_2, \ldots]$	A possibly infinite stream of entity descriptions
K_i	the set of blocking keys of e_i (p_i)
b_k	A block grouping all profiles that share blocking key k
$B = \{b_{k_1}, \ldots b_{k_n}\}$	A block collection
$\|\|B'_i\|\|$	The number of pairwise comparisons B would result in
$c_{ij} = \langle i, p_i, j, p_j \rangle$	A pair of profiles p_i and p_j with their identifiers i and j
M	a set of profile pairs c_{ij} considered to be matches
$\sigma_i = \langle M_i, B_i \rangle$	State maintained when processing e_i

We further define a state $\sigma_i = \langle M_i, B_i \rangle$ where M_i is the set of discovered matches found before processing e_i, while B_i is the block collection built before processing e_i. We denote by f_{er} the function returning an updated state σ_{i+1} based on e_i and σ_i; that is $\sigma_{i+1} = f_{er}(e_i, \sigma_i)$. The state σ_{i+1} updates σ_i with a new set of discovered matches $M_{i+1} \supseteq M_i$ and a new blocking collection B_{i+1}. Let σ_1 be an initial state. It can either be empty or be filled with the state resulting from applying ER on another dataset, which D is updating. Then, we define an incremental ER computation by the fold computation of f_{er} for each e_i in D, that is: $\sigma_{n+1} = f_{er}(e_n, ...f_{er}(e_2, f_{er}(e_1, \sigma_1))...)$.

We model f_{er} as the following combination of functions, that adapt the different steps of the general ER pipeline depicted in Figure 1 to support dynamic data:

$$f_{er}(e_i, \sigma_i) = (f_{cl} \circ f_{co} \circ f_{cc} \circ f_{cg} \circ f_{bc} \circ f_{bb} \circ f_{dr})(e_i, \sigma_i)$$

All functions take as input a single tuple, denoted within $\langle \ldots \rangle$, and return a single tuple. Their input tuple generally includes a state $\sigma_i = \langle M_i, B_i \rangle$ and the output tuple either comprises the identity of the state σ_i or a new updated state where M_i or B_i may have changed depending on the function. In this case, we denote the changed state as σ'_i and its changed components as M'_i or B'_i. We now discuss the individual functions.

Data reading: Let f_{dr} be a function returning a tuple $\langle i, p_i, K_i, \sigma_i \rangle$ given as input $\langle e_i, \sigma_i \rangle$. In addition to the unchanged state σ_i, the tuple specifies a unique identifier i, the standardized representation p_i of e_i, and the set of associated blocking keys K_i resulting from tokenizing the values in p_i.

Block building: We denote by f_{bb} the function returning a tuple $\langle i, p_i, K_i, \sigma'_i \rangle$, given as input a tuple $\langle i, p_i, K_i, \sigma_i \rangle$. The updated state consists in $\sigma'_i = \langle M_i, B'_i \rangle$, where B'_i updates all the blocks $b_k \in B_i$ where $k \in K_i$ by adding the tuple $\langle i, p_i \rangle$.

Block cleaning: We denote by f_{bc} the function returning a tuple $\langle i, p_i, K_i, \sigma'_i \rangle$ from input $\langle i, p_i, K_i, \sigma_i \rangle$. The new state consists in $\sigma'_i = \langle M_i, B'_i \rangle$ where B'_i is the "cleaned version" of the block collection B_i. Essentially, block cleaning removes tuples of the form $\langle i, p_i \rangle$ from B_i, targeting a significant reduction of $||B_i||$, the number of comparisons block collection B_i would result in. That is, block cleaning achieves $||B'_i|| \ll ||B_i||$. This property models both block purging and block filtering, the main methods used for block cleaning.

Comparison generation: While block cleaning considers individual entity descriptions assigned to blocks, comparison cleaning reasons on pairs of entity descriptions. Therefore, we introduce the comparison generation function, denoted f_{cg}. It takes as input $\langle i, p_i, K_i, \sigma_i \rangle$ and returns $\langle C_i, \sigma_i \rangle$, where C_i is a sequence of pairwise comparisons. Each comparison $c_{ij} \in C_i$ is a tuple $c_{ij} = \langle i, p_i, j, p_j \rangle$, constructed as follows. Essentially, for the "newly incoming" profile p_i identified by i and that matches the blocking keys K_i, we retrieve each $k \in K_i$ to then lookup the full blocks of the block collection with that key, i.e., $b_k \in B$. We then retrieve tuples from these blocks as $\langle j, p_j \rangle \in b_k$ and emit the pair $c_{ij} = \langle i, p_i, j, p_j \rangle$.

Comparison cleaning: We denote by f_{cc} the function that reduces the comparisons compared to the list of comparisons provided by the input in form of a tuple $\langle C_i, \sigma_i \rangle$. That is, it returns $\langle C'_i, \sigma_i \rangle$ such that the number of pairwise comparisons is significantly reduced, i.e., $|C'_i| \ll |C_i|$. This property models the same rationale as meta-blocking.

Comparison: We denote by f_{co} the function returning a set S_i as part of its output $\langle S_i, \sigma_i \rangle$, upon receiving $\langle C_i, \sigma_i \rangle$ as input. Essentially, for all $c_{ij} \in C_i$, we add an s_i to S_i, where $s_i = \langle c_{ij}, sim_{ij} \rangle$ and sim_{ij} is a similarity value measuring the similarity of the profiles in c_{ij}.

Classification: As last function, we introduce f_{cl}, the function that returns the final result of entity resolution (i.e., of f_{er}) by returning σ_{i+1}, which is updated based on the input $\langle S_i, \sigma_i \rangle$. The new state is defined as $\sigma_{i+1} = \langle M_{i+1}, B_{i+1} \rangle$. M_{i+1} updates M with new matches found by classifying each $s_i \in S_i$ as match or non-match. B_{i+1} is the same as in the function's input σ_i. It has been previously updated in the ER pipeline during block building and block cleaning.

B. Modeling clean-clean ER

The above model for incremental dirty ER can be easily specialized for clean-clean ER. First, we introduce a function $f_{combine}$ that takes as input two datasets D_x and D_y that are assumed to be clean. The function combines the two datasets in a dataset $D_{x,y}$ where each entity description e_i from dataset D_x (or D_y) is mapped as e_i^x (or e_i^y). In this way, we can use the same functions we defined before, just using both i and the dataset identifier combined in a tuple $\langle i, x \rangle$ (or $\langle i, y \rangle$) as identifier. The only exception is f_{cg}, which is adapted to generate comparisons only among profiles with a dataset identifier different to the dataset identifier of the input tuple.

C. Extending to streaming data

We define a streaming ER computation as a higher order function. It takes as input (1) a data stream of entity descriptions $D_S = [e_1, e_2, ..., e_n]$ possibly unbounded with $n \to +\infty$ and entity descriptions arriving in sequence at times $t_1 < t_2 < ... < t_n$ and (2) the function f_{er} with an initial state σ_1. The output of this higher order function is a stream $[M_1, M_2, ..., M_n]$, possibly unbounded with $n \to +\infty$, which represents a sequence of matches returned at times $t'_1 < t'_2 < ... < t'_n$ such that $t_i < t'_i$ and $\langle M_i, B_i \rangle = \sigma_i = f_{er}(e_{i-1}, \sigma_{i-1})$ is computed at t'_i.

In summary, this section covered a functional model of the general ER pipeline that updates a global state of blocks and found matches when a new entity e_i is processed. This

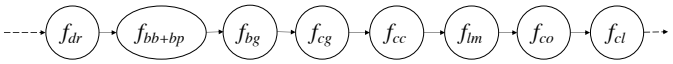

Fig. 3. Entity resolution pipeline

inherently supports incremental and streaming entity resolution and is amenable to both dirty ER and clean-clean ER. It further enables us to introduce task-based parallelization, as we describe in the next section.

IV. Task-based parallelization for ER

The functional model for ER on dynamic and heterogeneous data described in the previous section is suitable to be pipelined in multiple parallel stages. This allows different and independent tasks or segments of the stream to execute in parallel to improve the overall performance. In our context, tasks are entity descriptions, which can be individually processed as our functional model considers an individual e_i (or its corresponding p_i) in the functions' input. In this section, we study more closely how to map the functions of our model to multiple parallel processes or stages.

In Section IV-A, we discuss a mapping of the functional model to a pipeline of processes running in parallel, which mostly follows our functional model. As we shall see, this pipeline framework incurs some bottlenecks. To resolve these bottlenecks, we present an optimized framework in Section IV-B that carefully allocates computing resources to different steps of the ER pipeline, such that all ER steps achieve comparable latency.

A. Pipeline framework

Figure 3 shows a pipeline that maps all functions of the functional model discussed in Section III to a separate stage, with the exception of block building and block cleaning. These are split differently, resulting in two functions f_{bb+bp} and f_{bg} further introduced below. We further introduce a function f_{lm} for load management. This pipeline mapping is the result of the following design choices.

1) General design choices: In principle, we could faithfully follow the functional model to implement our framework. However, we opt for some adaptations, explained as follows.

Avoiding shared state. Our functional model passes $\sigma_i = \langle M_i, B_i \rangle$ across the different steps in a pure functional style. However, in our framework, for efficiency reasons, we decide to "locally" maintain the state as part of a blocking function f_{bb+bp}, that exclusively summarizes the blocking steps that modify the state. Consequently, we "outsource" our stream-enabled variant of block filtering, called block ghosting, to a separate stage f_{bg}. This separation gives us more opportunity for task parallelization by also replicating f_{bg} to run different and independent tasks in parallel. Indeed, for functions changing the state, the more suited choice is data parallelism by partitioning the state. We further assume that f_{cl} does not update any state and just delivers the results, i.e., the matches found that involve the processed e_i, to a consumer or stores the result on disk. We further maintain a state for f_{lm}, described next.

Profile maintenance. We reduce the size of the block collection maintained by f_{bb+bp} by only storing identifiers $\langle j \rangle$ rather than both the identifier and the corresponding profile, i.e., $\langle j, p_j \rangle$. We recover the profiles once needed, meaning prior to the comparison stage. To this end, f_{lm} maintains a profile map PM. PM serves as an index to lookup a full profile p_j of an entity e_j identified by j, which has been determined to require comparison to the currently processed entity description.

2) Implementing individual stages: We now discuss details on the implementation of individual stages. The algorithms underlying data reading, comparison generation, comparison, and classification simply adapt existing solutions from batch ER processing, incorporating the design choices described above. The algorithm implementing f_{lm} performs a simple lookup of a profile to implement. Due to space constraints, our discussion focuses on the remaining steps that need more adaptation, i.e., blocking using f_{bb+bp} and f_{bg} and comparison cleaning f_{cc}.

Block building (part of Algorithm 1) is the first part of the algorithm for f_{bb+bp}. It only requires to add each key among the keys associated with e_i, that is, each $k \in K_i$, to the corresponding block b_k of the block collection B_i. This corresponds to the first two lines of the for-loop in Algorithm 1. We assume that $B_i.getOrInit(b_k)$ returns an empty set if no block exists for key k.

Figure 4 shows how the block collection local to f_{bb+bp} evolves as the entities e_1 through e_5 of our previous example are incrementally processed. At first, data reading for e_1 yields a standardized and tokenized representation p_1 of attribute values of e_1, and the associated set of blocking keys is $K_1 = \{top, panel, wood, pavillion, John\}$. Thus, block building results in B_1 having 5 blocks, each referring to e_1. As further entities are processed, state B_i gradually grows. In comparison to the block collection of Figure 2(b), once e_5 is processed, we continue to maintain blocks of size 1 (e.g., block "Jane") as they may grow in the future.

Let us now focus on block cleaning. As we have seen in Section IV-A, block cleaning integrates both block purging and block filtering. For dynamic data where the final sizes of blocks are a priori unknown, we cannot leverage existing techniques that rely on maximum or minimum size blocks. Therefore, in our framework, we propose *block pruning* that similarly to block purging removes complete blocks from the block collection, and *block ghosting* as a variant of block filtering where individual profiles in a block are ignored.

Block pruning (part of Algorithm 1) relies on a parameter $\alpha > 1$. It can be set based on the estimated size of a dataset to be processed, or to a user-defined maximum admissible block size. We further maintain a blacklist $\overline{K}$ of key values that have been encountered when processing entity descriptions e_h, $h < i$ and already pruned as the corresponding blocks have grown too large. In the algorithm, after adding i to b_k, we verify if b_k is oversized by checking if its size $|b_k| \geq \alpha$. An oversized block b_k is removed from the block collection B_i and the corresponding k is added to $\overline{K}$. This blacklist is consulted when new keys are processed, i.e., in the for loop, we check if a key $k \notin \overline{K}$ to proceed. The non-oversized blocks to which i

Algorithm 1: f_{bb+bp}

Input: $\langle i, p_i, K_i \rangle$
Output: $\langle i, p_i, K_i \rangle$
B_i : the block collection
α: a parameter
B_{e_i} : a subset of B_i, initially empty
$\overline{K}$: blacklist of keys
foreach $k \in K_i \wedge k \notin \overline{K}$ **do**
  $b_k \leftarrow B_i.getOrInit(k)$
  $b_k.add(\langle i \rangle)$
  if $|b_k| \geq \alpha$ **then**
    $\overline{K} \leftarrow \overline{K} \cup \{k\}$
    $B_i.remove(k)$
  else
    $B_i.put(k, b_k)$
    $B_{e_i}.put(k, b_k)$
return $\langle i, p_i, K_i, removeSingletons(B_{e_i}) \rangle$

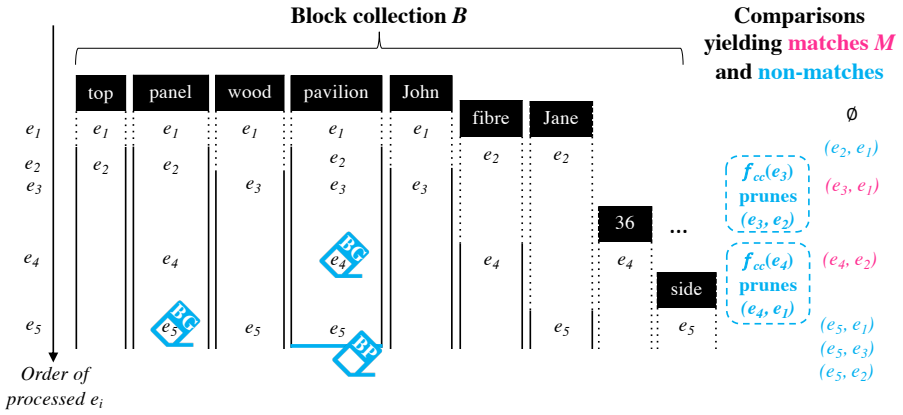

Fig. 4. Example execution of incrementally processing e_1 to e_5 using pipelined framework.

Algorithm 2: f_{bg}

Input: $\langle i, p_i, K_i, B_{e_i} \rangle$
Output: $\langle i, p_i, K'_i, B'_{e_i} \rangle$
β: a parameter
$b_{min} \leftarrow$ the minimum size block in B_{e_i}
foreach $k \in K_i$ **do**
  $b_k \leftarrow B_{e_i}.get(k)$
  if $|b_k| > \frac{|b_{min}|}{\beta}$ **then**
    $K_i \leftarrow K_i \setminus \{k\}$
    $B_{e_i}.remove(k)$
return $\langle i, p_i, K_i, B_{e_i} \rangle$

Algorithm 3: f_{cc}

Input: $\langle C_i \rangle$
Output: $\langle C'_i \rangle$
$G \leftarrow \emptyset$, a map maintaining a count for each c_{ij}
foreach $c_{ij} \in C_i$ **do**
  $newCount \leftarrow G.get(c_{ij}) + 1$
  $G.put(c_{ij}, newCount)$
$avg \leftarrow$ get average of counts in G
$C'_i \leftarrow \emptyset$
foreach $\langle c_{ij}, count \rangle \in G$ **do**
  if $count \geq avg$ **then**
    $C'_i \leftarrow C'_i \cup \{c_{ij}\}$
return $\langle C'_i \rangle$

is added are updated both in the global block collection B_i and a temporary block collection B_{e_i} that only includes the blocks relevant to processing e_i. The function $removeSingletons$ removes blocks of size 1 of B_{e_i} before, which becomes part of the output of f_{bb+bp}.

In the example shown in Figure 4, we assume $\alpha = 5$. When processing e_1 through e_4, blocks continuously grow but obviously, none reaches size 5. When processing e_5, data reading gives $K_5 = \{panel, wood, pavilion, Jane, side\}$. Adding e_5 to the "pavilion" block makes it oversized. Thus, "pavilion" is added to the blacklist $\overline{K}$ to ensure the block is discontinued. This prunes four comparisons of e_5 to other entity descriptions. After removing the singleton block "side", block building passes on only 3 blocks as B_{e_5} for further processing.

Block ghosting (Algorithm 2) In a dynamic ER scenario where B_i changes over time, it is not wise to actually remove individual profiles from each block, especially in the early phases of the resolution process. Indeed, the distribution of block sizes may change and shift the decision of whether to generate comparisons for an entity description from one block or another. Therefore, block ghosting retains all entity descriptions (more precisely, their identifiers) in blocks, but adapts the set of keys K_i to only include keys of blocks to be considered by the next stage, i.e., comparison generation. This effectively ignores blocks whose keys have been removed from K_i. To determine the keys to be ignored, block ghosting first finds the smallest block b_{min} present in B_{e_i}. For each key $k \in K_i$, we then check if the corresponding block is significantly larger than b_{min}, where significant is determined by the parameter β, $0 < \beta < 1$. Intuitively, β indicates at which point we discard comparisons to e_i because it is in a block too general and likely to generate many superfluous comparisons. Currently, β is set statically. Changing it dynamically is an interesting avenue for future research. If b_k is considered too large, it is removed from K_i. In f_{cg}, we will then only generate comparisons c_{ij} from blocks with a key contained in K_i.

In our example and assuming $\beta = 0.6$, block ghosting prunes nothing when incrementally processing e_1 through e_3. When reaching e_4, among the three blocks in B_{e_4}, the smallest block size is 2 (e.g., block "fibre"). This discards the "pavilion" block, which has reached size 4 at this point, because $4 > \frac{2}{0.6}$, effectively pruning comparisons (e_4, e_1), (e_4, e_2), and (e_4, e_3).

When processing e_i, comparison generation f_{cg} simply builds, for each remaining block in B_{e_i} all pairwise comparisons c_{ij} between e_i and any e_j present in a same block. The collection of all such comparisons, denoted C_i is passed on to comparison cleaning. For instance, after block cleaning for e_4 as described above, comparison generation creates $C_4 = \{(e_4, e_1), (e_4, e_2), (e_4, e_2)\}$, where the first two comparisons originate from the "panel" block and the last one from the "fibre" block.

Comparison cleaning (Algorithm 3) prunes pairwise comparisons, for which we propose a variant for dynamic data of the CBS weighting scheme and the WNP pruning scheme introduced in Section IV-A. Essentially, we group the tuples in C_i by i (that is fixed) and j and count the number of tuples in each group. The grouping effectively prunes redundant comparisons. To further prune superfluous comparisons, we determine a threshold avg, computed as the average count per group. We then only keep comparisons with a count equal to or larger than avg, as these co-occur in more blocks and thus are more likely to be actual matches. Note that this approach does not rely on a blocking graph and only depends on the comparisons C_i generated by the previous stage.

In our example, when processing C_4 given above, compari-

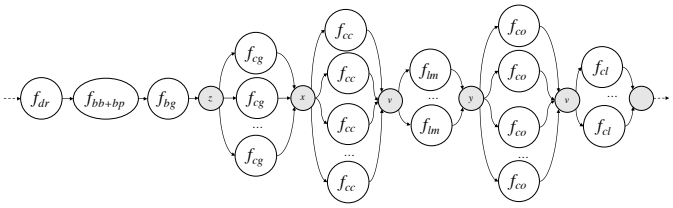

Fig. 5. Parallelization overview of optimized framework

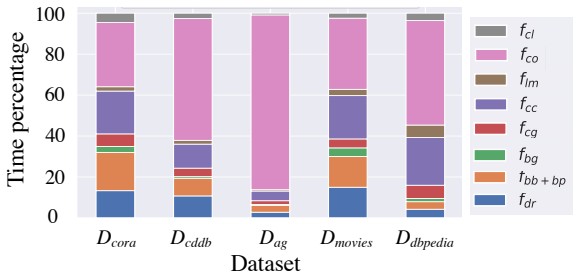

Fig. 6. Computation bottlenecks ($\beta = 0.05$, $\alpha = 0.005|D|$ for $D_{dbpedia}$, otherwise $\alpha = 0.05|D|$.

son cleaning creates the following groups and associated count: $(e_4, e_1) \rightarrow 1$ and $(e_4, e_2) \rightarrow 2$. Only (e_4, e_2) has a count higher than the average count of 1.5. So f_{cc} relays only one comparison $C'_4 = \{(e_4, e_2)\}$.

3) Discussion: When implementing the pipeline for ER on dynamic data as described above, we can support both incremental ER and streaming ER. However, theoretical speedup is limited by the number of the stages. Indeed, assuming that all stages are busy processing an entity description at the same time, we can at most process eight entity descriptions simultaneously. This assumes that each stage uses comparable time to process an entity description. However, this optimal performance is not achieved due to some stages being slower than others, i.e., they are bottleneck stages. As experiments show, the pipeline solution, even with careful design choices, is slowed down by the bottleneck stages of comparisons, comparison cleaning, and comparison generation, yielding only a moderate speedup of at most 2 compared to a sequential execution that does not incorporate task-paralellism through pipelining. The optimized framework described in the next section studies and addresses these bottlenecks.

B. Optimized framework

To identify bottleneck stages, we resort to both a theoretical and an experimental analysis of time required by the algorithms underlying each stage.

Theoretical analysis. We estimate similar and low runtime for f_{dr}, f_{bb+bp}, and f_{bg} as they all iterate over a similar number of tokens or token keys K_i. Opposed to that, comparison generation has a complexity of $O(|K_i| \times \alpha)$. This is also true for comparison cleaning, however, we expect it to run twice as long. The next stage is load management f_{lm}. Assuming comparison cleaning is effective in reducing the number of comparisons (experiments show a reduction by an order of magnitude), f_{lm} performs a simple lookup for far less pairs than originally generated and input to comparison cleaning. It is fair to assume that this will take less time than f_{cg} and f_{cc}. Opposed to that, while f_{co} also processes the same (reduced) number of pairs as f_{lm}, we have to take into account the potentially complex similarity measure that compares two (possibly large) profiles with each other. Depending on the size of profiles, similarity measure used, and the effectiveness of pruning comparisons through blocking, f_{co} may or may not be more time consuming than f_{cc} or f_{cg}. As for f_{cl}, it should be faster than f_{cg}. While both have comparable theoretical worst case complexity, effective pruning should keep f_{cl} below f_{cg}.

From the above discussion, we expect the main bottlenecks to be f_{cc}, f_{co}, and f_{cg}, while f_{lm} and f_{cl} are expected to be more efficient and comparable in runtime. The fastest stages are likely f_{dr}, f_{bb+bp}, and f_{bg}. From this observation, we now formulate the problem of assigning processing resources, i.e., P processes, for parallel execution such that all stages complete in similar times. Intuitively, this requires us to allocate most resources to parallelize the bottleneck stages, while we can discard parallelization for the cheapest stages.

This amounts to solving the following problem. Let T_f be the "target" runtime for completing each stage, which is equal to the (comparable) time of stages not requiring parallelization, i.e., $T_f = T_{dr} = T_{bb+bp} = T_{bg}$. We want to assign x processes to f_{cc} such that $\frac{T_{cc}}{x} = T_f$. Analogously, we assign y and z processes to f_{co} and f_{cg} such that $\frac{T_{co}}{y} = T_f$ and $\frac{T_{cg}}{z} = T_f$. The number of processes assigned to the comparable f_{lm} and f_{cl} can be set to v such that $T_{lm} = T_{cl} = v \times T_f$. Having in total P processes at our disposal, and requiring at least one process for each stage, we have $P = 1 + 1 + 1 + z + x + v + y + v = 3 + 2v + x + y + z$, such that $v < z$, $x > z$, and $y > z$.

The optimized framework thus no longer resembles a pipeline of successive stages. Instead, it follows the general scheme of parallelizing different stages by different degrees (v, x, y, and z) as illustrated in Figure 5.

Experimental analysis. Given the setup and data we describe in detail in Section V, we further conduct an experiment to measure the actual times per stage. For different datasets, Figure 6 shows how much each stage contributes to the overall runtime for processing the full dataset when using the pipeline in Figure 3 using parameter settings offering good efficiency-quality balance as established in Section V.

We see that in practice, f_{co} and f_{cc} are the main bottlenecks, which is in accord with our theoretical analysis. They are followed by f_{cg} if we consider $D_{dbpedia}$, the biggest dataset, which again matches our theoretical analysis. On smaller datasets however, f_{bb+bp} is the third most expensive. One explanation may be that for the smaller datasets, processing a potentially large number of tokens in $|K_i|$ is actually more expensive than generating the moderate number of comparisons in f_{cg} due to generally small block sizes and effective pruning. Also, absolute runtimes on the small datasets are low (between 600 ms and 15 s), while rising to 54 minutes on $D_{dbpedia}$.

Considering $D_{dbpedia}$, all phases except f_{co} and f_{cc} have comparable times, which are between 100 and 200s. We further

TABLE II
DATASETS CHARACTERISTICS

	Name	Type	Number of entity descriptions	Number of matches	Average Number of name-value pairs per profile
D_{cora}	cora	dirty ER	1.29k	17.1k	5.5
D_{cddb}	cddb	dirty ER	9.76k	299	17.8
D_{ag}	amazon-google	dirty ER	4.39k	1,10k	3.3
D_{movies}	movies	clean-clean ER	27.6k - 23.1k	22.8k	5.6
$D_{dbpedia}$	dbpedia	clean-clean ER	1.19M - 2.16M	892k	14.2

observe that $T_{co} \approx 2 \cdot T_{cc}$ and that $T_{cc} \approx 3 \cdot T_{cg}$. Based on this analysis, we set the number of parallel processes run for the different phases. For instance, when using $P = 15$, we set $v = 1$, $x = 3$, $y = 6$ and $z = 1$.

V. EVALUATION

In this section, we evaluate experimentally our solutions for ER on dynamic data. (1) We first study how our adapted techniques for block cleaning and comparison cleaning, which reduce the number of expensive pairwise comparisons to be performed, compare to existing techniques that do not support dynamic data. We then show how this affects the overall performance of our pipeline, compared to the overall performance of existing batch ER. For a fair comparison, we run all approaches in a batch setting where the task is to perform ER on a fixed dataset once. We also do not yet leverage parallelism, effectively implementing a sequential ER pipeline that implements the algorithms presented in Section IV-A. (2) Next, we evaluate ER performance in an incremental setting, where we compare our approach to adapted batch solutions and a pipeline incorporating PI-Block [1]. (3) Next, we focus on the effect of parallelization by studying speedup. (4) Finally, for streaming data, we evaluate the throughput and latency.

Datasets. Table II summarizes characteristics of the datasets we used. They have been extensively used in the ER literature (e.g., in [11], [15], [26]). While the datasets suited for dirty ER do not exhibit significant schema heterogeneity (D_{cora}, D_{cddb}, D_{ag}), the large datasets for clean-clean ER are heterogeneous and semi-structured (D_{movies}, $D_{dbpedia}$). A ground truth file of matches accompanies each datasets[1].

Metrics. We evaluate incremental ER solutions in terms of quality and runtime efficiency. We quantify quality using *pair completeness (PC)*, an established measure defined as the number of matches that can be detected after BC and CC, divided by the total number of matches $|M_D|$ in the input dataset D. To assess the effectiveness of pruning techniques, we rely on the number of comparisons generated by a block collection (i.e., the cardinality of the block collection $||B||$) after BB, BC, or CC. To evaluate the benefits of parallelization, we rely on *speedup* $sp(n) = \frac{RT(SEQ)}{RT(n)}$ that puts in relation the time for sequential ER to complete, denoted $RT(SEQ)$ with the resolution time $RT(n)$ of the parallel implementation with parallelism degree of n. For ER on streaming data, we measure the *latency* defined as the time to process a single input entity description end to end and the *throughput* defined as the number of entity descriptions processed per second.

Baselines. For comparison to state-of-the-art solutions, we first leverage the methods available via the JedAI framework [21]. These allow us to build an ER pipeline for batch processing that includes token blocking, block purging relying on a parameter r, block filtering configured using parameter s, meta-blocking for comparison cleaning, pairwise comparison (employing Jaccard similarity), and classification via lookup in the ground truth data (thereby assuming a perfect classifier). Such a workflow represents the state-of-the-art to perform batch ER for larger volumes of heterogeneous data [4]. Given that our work focuses on efficiently supporting dynamic data, we use a standard configuration for the steps mostly affecting ER quality without significantly changing the runtime (i.e., comparison using Jaccard and classification using lookup). Opposed to that, we test different parametrizations for block purging where we vary $r \in \{0.05, 0.005\}$ and block filtering, where we vary $s \in \{0.1, 0.5, 0.8\}$. For meta-blocking, we use CBS as weighting scheme, as it is closest to the weighting scheme used in our approach, whereas we vary the pruning scheme, using either WEP, WNP, Reciprocal WNP (RWNP), CEP, CNP, and Reciprocal CNP (RCNP) [18], [19]. Considering efficiency-intensive applications we follow the suggestion of [19] and we consider also the combinations of Reciprocal WNP with JS scheme (RWNP+JS) for clean-clean ER and Reciprocal CNP with ARCS scheme (RWNP+ARCS) for dirty ER.

For incremental ER, we consider three baselines: (1) the best batch ER configuration that recomputes the blocking steps over the whole collected data for each increment without recomputing again previously processed comparisons; (2) an ER pipeline that integrates the incremental blocking solution PI-Block (substituting f_{bb+bp}, f_{bg}, f_{cg}, and f_{cc}; (3) a degraded version of our ER framework that skips block cleaning, because PI-Block essentially performs comparison cleaning.

Implementation. We implemented our methods as well as baseline pipelines in Scala. Baselines use methods provided by the JedAI framework (version 3.0) that is implemented in Java 8. The only exception is our re-implementation of PI-Block. We reimplemented it because it was originally developed in Apache Spark and as acknowledged in the original paper (and verified by us), this implementation is unable to complete, e.g, on the moderately sized dataset D_{movies}, without the use of substantial computing resources (in their case a distributed infrastructure with at least 12 nodes) as it requires a large amount of memory. For parallelization and support of streaming data, we leverage the Akka Streams API, which allows programmers to build scalable applications on streaming data. All experiments were performed on an OpenStack virtualized server with Ubuntu 18.04 (16 processors at 2.30GHz, 50GB RAM).

A. Comparative evaluation: Batch setting

We first consider a sequential implementation of our functions by comparing a non-parallel version of our pipeline of Figure 3 to different baseline configurations for batched ER.

[1]Available at: https://github.com/scify/JedAIToolkit

TABLE III
COMPARATIVE EVALUATION OF THE NUMBER OF COMPARISONS RESULTING FROM BLOCK CLEANING

	block purging + block filtering							block pruning + block ghosting					
r	0.05			0.005			α	0.05 $\times\|D\|$			0.005 $\times\|D\|$		
s	0.1	0.5	0.8	0.1	0.5	0.8	β	0.1	0.05	0.01	0.1	0.05	0.01
(a) Parameter configurations for baseline block cleaning (left) and stream-enabled block cleaning (right)													
D_{cora}	2.68E+03	3.94E+04	8.95E+04	4.30E+01	7.94E+02	1.32E+03		3.01E+05	3.44E+05	3.54E+05	9.92E+03	9.92E+03	9.92E+03
D_{cddb}	2.05E+04	8.11E+05	4.48E+06	8.58E+03	8.58E+03	4.63E+05		1.57E+06	3.39E+06	1.61E+07	1.38E+06	1.38E+06	2.36E+06
D_{ag}	1.61E+04	5.21E+05	2.02E+06	3.25E+03	4.12E+04	1.16E+05		1.43E+06	3.01E+06	8.50E+06	5.81E+05	5.86E+05	5.86E+05
D_{movies}	1.97E+05	8.32E+06	4.92E+07	1.11E+05	2.41E+06	9.72E+06		7.85E+06	1.51E+07	5.14E+07	4.47E+06	7.28E+06	1.18E+07
$D_{dbpedia}$	//	//	//	1.15E+07	1.92E+09	1.21E+10		//	//	//	1.29E+09	3.25E+09	1.16E+10
(b) Number of pairwise comparisons that would result from the cleaned block collection for different parameter configurations													

This series of experiments validates that, in a batch setting, our approach, which incorporates alternative blocking techniques, is comparable or even better than well-established algorithms for batched ER, both in terms of efficiency and quality.

We start with a study of how block cleaning, as implemented in our framework using block pruning (BP) and block ghosting (BG), performs compared to batched block cleaning that is based on block purging (BPu) and block filtering (BF). We use the baseline configurations described above. For our BP and BG algorithms, we vary the parameters $\alpha \in \{0.05 \times |D|, 0.005 \times |D|\}$ and $\beta \in \{0.1, 0.05, 0.01\}$.

Table III reports the results in terms of pairwise comparisons that would be performed when using the cleaned block collection as returned by block cleaning. For $D_{dbpedia}$ we consider only the more aggressive (in terms of pruning) configurations of $r = 0.005$ (respectively $\alpha = 0.005 \times |D|$) for block purging (respectively block pruning). The other configurations would take too long.

From the numbers reported in Table III, we first observe that the most aggressive baseline configuration ($r = 0.005$ and $s = 0.01$) is more effective in pruning comparisons than the most aggressive configuration for stream-enabled block cleaning ($\alpha = 0.005 \times |D|$ and $\beta = 0.1$). Indeed, the difference in number of pairwise comparisons is regularly more than two orders of magnitude. As we move towards less aggressive configurations, we observe that this gap between baseline solutions for batch ER and our approach diminishes to less than an order of magnitude. Nevertheless, our block cleaning for streaming ER is generally less effective in pruning pairwise comparisons than its counterpart in batch ER.

We now evaluate the effectiveness of the next step that prunes pairwise comparisons, i.e., comparison cleaning. We consider different baselines for CC as described above. We refer to our CC method as I-WNP (for incremental weighted node pruning). Figure 7 plots the number $||B||$ of pairwise comparisons resulting from (different configurations of) block cleaning on the x-axis, and the number of pairwise comparisons after CC on the y-axis, denoted as $||B'||$. We report results for D_{cddb} in Figure 7(a), which are representative of results obtained on the other datasets with the exception of $D_{dbpedia}$, which we therefore report in Figure 7(b).

Overall, on most datasets, we observe a trend similar to the trend seen in Figure 7(a). That is, most baseline approaches

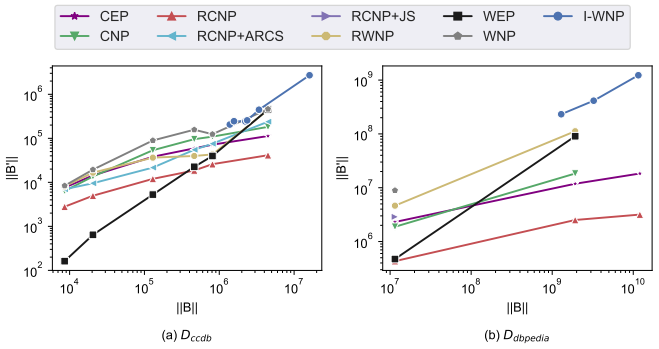

Fig. 7. Effectiveness of comparison cleaning

reduce the number of pairwise comparisons between one and two orders of magnitude. Our I-WNP is at the more conservative end of the spectrum by reducing the number of comparisons by roughly one order of magnitude. The picture changes when we look at the largest dataset $D_{dbpedia}$. First, note that Figure 7(b) reports results for less configurations (i.e., less points), as some failed after hours of computation as they ran out of memory. For the configurations still running, we see that RCNP can achieve a reduction of comparisons of up to three orders of magnitude, while I-WNP maintains its one order of magnitude. We conclude that the effectiveness of our comparison cleaning method I-WNP is stable and reduces the number of comparisons by an order of magnitude. This is comparable to some baseline approaches, while it is significantly less than the best baseline CC techniques.

However, the number of comparisons to be performed after BC and CC is not yet an indicator of the overall end-to-end performance of ER. In particular, the generally expensive comparison step CO still lies ahead. Remember that the primary goal of putting so much effort into BB, BC, and CC is to perform less expensive comparisons. So an interesting question is how the effectiveness and cost of reducing comparisons balances with the cost of CO. Furthermore, the reduction in performed comparisons needs to be put in perspective to the quality of the returned result. In practice, the most aggressive parametrizations obtaining the highest pruning do not yield good quality results. To study the end-to-end performance of different configurations of our approach with respect to different baseline configurations, we measure the overall runtime (RT)

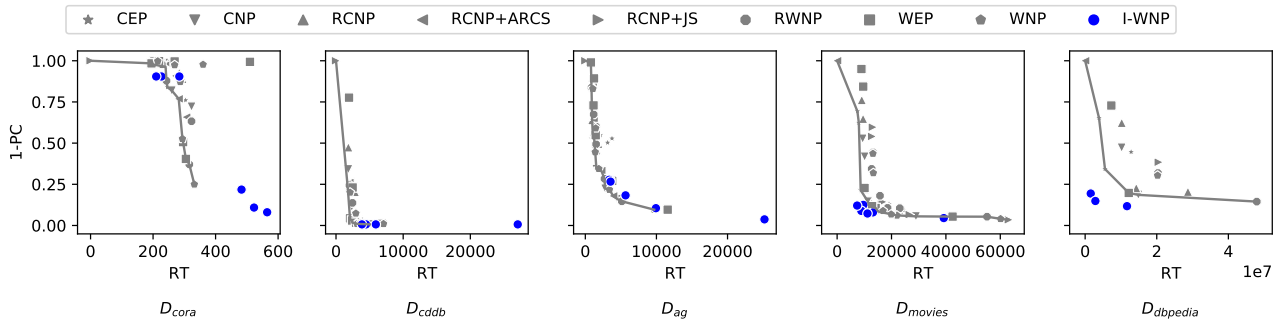

Fig. 8. I-WNP configurations against the Pareto frontier of the baseline configurations to evaluate the tradeoff between overall ER runtime and quality.

of each configuration as well as the quality of the ER result as pair completeness (PC). We use PC instead of recall and precision, because, as mentioned before, we assume a perfect classifier. Then, PC equals recall while precision is 1.

Figure 8 plots the runtime (in ms) against $1 - PC$ (such that both axis represent better results with smaller values) for each tested configuration and dataset. Grey points represent results for baseline end-to-end ER for different CC approaches and varying BC configurations (left in Table III). Blue points summarize results for I-WNP coupled with the different configurations of BP and BG (right in Table III). In each graph, we trace the Pareto frontier of the baseline configurations, in other words the best compromises between RT and $1 - PC$.

On all datasets, we see that at least one configuration of our end-to-end ER solution with I-WNP is on or "ahead" of the Pareto frontier. This shows that even in batch mode, our solution is either comparable or better than the state-of-the-art. From these results, we conclude that our solution actually designed for ER on dynamic data is comparable or outperforms batch ER solutions in terms of overall runtime and quality, despite the individual functions used to reduce the number of comparisons (BC and CC) being less effective.

To understand how we obtain this overall performance, we further investigate the performance of different ER steps using both the baselines and our solution. For the representative datasets D_{cddb} and $D_{dbpedia}$, Figure 9 reports (1) the time BT that summarizes the time needed for data reading, block building, and block cleaning (plus comparison generation and load managing for our solution), (2) the time for the subsequent comparison cleaning (CCT), and (3) the overall runtime RT, which adds to the time BT and CCT the time for pairwise comparisons and classification. All results (on the y-axis) are reported with respect to the number of comparisons reported in Table III after applying different block cleaning configurations. The shaded area within the dashed lines in the graphs puts the results in perspective with the results of Figure 8 on overall ER quality. The area covers the optimal configurations on the Pareto frontier for the baselines, and the configurations of our solution that are comparable or better than these baselines.

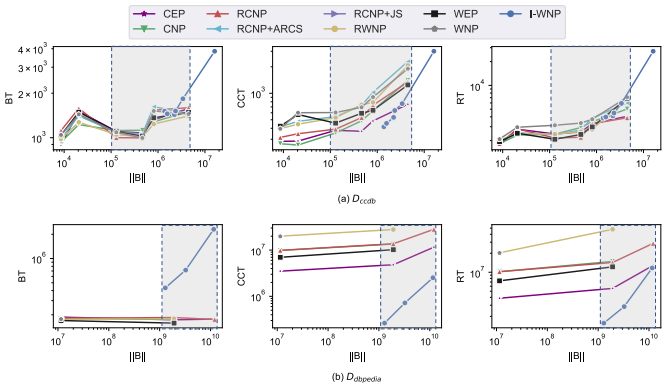

Fig. 9. Runtime for blocking steps (BT), comparison cleaning (CCT), and end-to-end ER (RT) for both D_{ccdb} and $D_{dbpedia}$.

As mentioned before, the overall runtime RT is comparable for all approaches on smaller datasets such as D_{cddb} for comparable quality. On this dataset, we see that while BT is comparable for all configurations, our approach compensates its reduced "pruning power" by being significantly faster during comparison cleaning. The reason lies in the low scalability that meta-blocking has, since it needs to materialize and process an increasingly large graph, resulting in the main computation bottleneck of batch ER. This effect is exacerbated when looking at the results of $D_{dbpedia}$, the largest dataset. Here, we actually see that even though BT for our approach is significantly higher than baselines, our comparison cleaning method outperforms all baseline CC approaches. In the baseline batch ER pipelines, we observe that CC is more than 10 times slower than its predecessor steps summarized by BT, whereas CC is actually faster or comparable to blocking when using the solutions proposed in this paper to run sequential ER. This gain is not lost during comparison and classification, as the overall runtime RT on $D_{dbpedia}$ shows. That is, our approach outperforms all others in terms of runtime (and quality, cf. Figure 8).

B. Comparative evaluation: Incremental setting

For incremental processing, we divide a dataset into a varying number of equally sized partitions to be incrementally processed. We then compare the following approaches: I-WNP is the best configuration of our solution (the blue point closest to the origin in Figure 8); Batch denotes the best configuration of batch ER; PI-Block is the ER pipeline integrating PI-Block; and

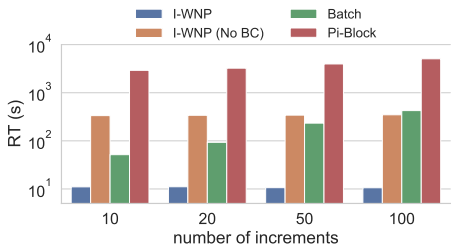

Fig. 10. Incremental ER runtimes for D_{movies}.

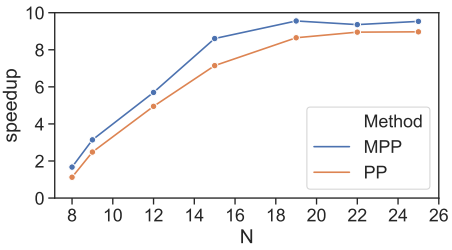

Fig. 11. Speedup on $D_{dbpedia}$.

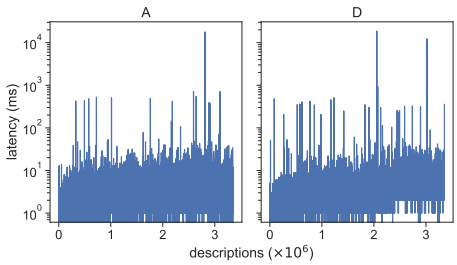

Fig. 12. Latency per entity description for source rates of (A) 5000 and (D) 100000 descriptions/s.

I-WNP (No BC) excludes block cleaning from our approach, as PI-Block does not feature this blocking step.

Figure 10 reports the runtime used to process all increments (number varies on x-axis) of D_{movies}. It is the sum of end-to-end time needed to process each increment, starting from the moment when it arrives. We also measure quality in terms of PC. For approaches using both BC and CC, PC is consistently around 0.90 whereas those limiting to CC exhibit PC ≈ 0.97.

Clearly, I-WNP outperforms all baselines in an incremental setting in terms of runtime. Independently of the number of increments, its runtime is stable (around 11s). Opposed to that, runtime of Batch to process D_{movies} increases with the number of increments. The results also show the importance of applying both block cleaning and comparison cleaning, as the baselines without block cleaning, i.e., PI-Block and I-WNP (No BC), perform worst. Note that the best blocking time using the original PI-Block implementation based on Spark reported in [1] is around 1200 s when using 12 nodes and 10 increments. It could not run with less nodes because the internal state needs to be partitioned across many nodes before fitting in available memory). In contrast, our approach is 100 times faster while using less resources. Therefore, we do not consider PI-Block in the next section, where we focus on how parallelization affects the performance of our solution.

C. Parallel evaluation

We implement and configure the parallel optimized framework, subsequently referred to as parallel pipeline (PP) as discussed in Section IV. We further implement a variant of PP that leverages micro-batching when passing data between stages. This variant, called micro-batched parallel pipeline (MPP) introduces an aggregation stage before each stage of the framework. Its task is to collect multiple profiles that are then passed on as a micro-batch to be processed sequentially. We configure these aggregation stages to generate micro-batches of at most 100 profiles in at most 10 milliseconds. We report experiments on the largest dataset $D_{dbpedia}$.

In terms of quality, the sequential implementation SEQ as well as PP and MPP reach the same high $PC = 0.85$. As for runtime, SEQ takes 54 minutes, PP runs in 360 s, and MPP finishes after 340 s. Clearly, both approaches that use parallelization are roughly 10 times faster than the non-parallel solution, MPP being the fastest.

For PP and MPP, we further study the speedup with respect to SEQ when we vary the number of processes from 8 (meaning no parallelization as each function requires at least one process) to 25, where processes are distributed among the bottleneck stages using the previously mentioned proportions. Figure 11 reports the results.

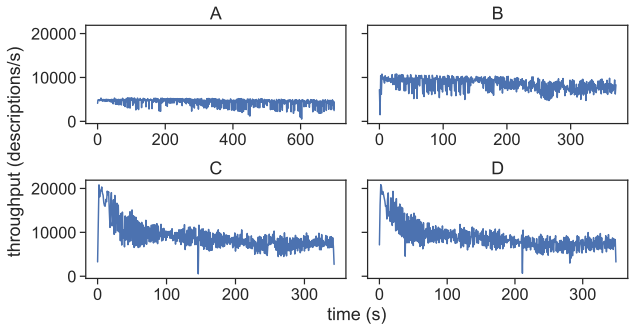

Fig. 13. Output throughput over time for source rates of (A) 5000, (B) 10000, (C) 50000, and (D) 100000 descriptions/s.

First, we observe that when using 8 processes, our results for PP are consistent with the results reported for our non-optimized pipeline that suffers from communication overhead and bottleneck stages. Indeed, the speedup for PP is then only 1.12. Looking at the performance of MPP, we see that micro-batching attenuates the effect of non-negligible communication overhead and we reach a speedup of 1.67. MPP consistently outperforms PP also when we increase the number of processes. The peak speedup is around 8 for PP and 9.5 for MPP. Note that this peak is reached when using 19 processes (as a reminder, we have a 16 core machine). We explain the stagnation of speedup thereafter by the saturation of computing resources.

In summary, we see that parallelization can significantly speed up batched and non-parallel ER (we reached a factor of 100). However, the allocation of processing power to individual functions needs to be carefully balanced to avoid bottleneck stages, which is integrated in our optimized framework. Micro-batching further improves the performance.

D. Streaming evaluation

Our final set of experiments focuses on dynamic data. We validate that our framework indeed supports streaming data and study both throughput and latency. We report results when using PP configured as described in Section IV-B and using 25 processes. We simulate a source sending a stream of entity descriptions at given rate. We vary the

rate from (A) 5000 descriptions/s, (B) 10000 descriptions/s, (C) 50000 descriptions/s, and (D) 100000 descriptions/s. Entity descriptions are retrieved from $D_{dbpedia}$.

Figure 12 reports the latency (only extreme cases A and D for conciseness, B and C are similar). The latency is measured per entity description for a stream of 3 million entity descriptions. We observe that the latency is quite robust to different input stream rates, indicating that no significant bottleneck builds up. However there are latency peaks that may degrade overall performance, which we will further investigate in the future.

Figure 13 shows the output throughput of our optimized framework in terms of descriptions processed end-to-end per second, given the four different stream rates. We first observe that when the source generates descriptions at a rate that is lower than the effective execution rate of the framework, the maximum output throughput is the same as the maximum input throughput (case A). We notice some drops in throughput, which we attribute to segments of the streams that are particularly CPU-intensive as we measured high latency peaks (see below).When the source generates descriptions at a rate that is comparable to the effective average execution rate within the framework (determined as $\frac{RT(SEQ)}{|D_{dbpedia}|}$), the throughput is approximately stable over time (case B). When the source generates profiles at a rate higher than the effective execution rate of the framework, the output throughput is high at the beginning when the system is not overloaded, and then tends to stabilize (case C, D).

To summarize, the above results demonstrate that our optimized framework enables "real-time" ER processing of dynamic data, as latency is typically in the range of 10 to 100 ms. While throughput can match even high incoming stream rates during early stages of processing, the throughput tends to stabilize around a system-dependent throughput, in our cases around 7500 to 8000 descriptions per second.

VI. Conclusion

This paper introduced a novel framework for end-to-end ER of dynamic and heterogeneous data. We described in detail a functional model, which forms the theoretical foundation for task parallel ER processing. We discussed how to optimally map the functional model into a parallel ER framework to avoid that bottlenecks arise. As experiments validated, this framework allows to efficiently and effectively perform ER, improving the efficiency of state-of-the-art approaches by up to two orders of magnitude. For dynamic data, our experiments showed that good speedup can be reached through well designed parallelization, ultimately resulting in high throughput and low latency that pave the way to real-time ER. Ideas for future research include devising a self-tuning framework and studying end-to-end ER with service guarantees.

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

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

[3] P. Christen. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE TKDE*, 24(9):1537–1555, 2012.

[4] V. Christophides, V. Efthymiou, T. Palpanas, G. Papadakis, and K. Stefanidis. End-to-end entity resolution for big data: A survey. *CoRR*, abs/1905.06397, 2019.

[5] D. C. do Nascimento, C. E. S. Pires, and D. G. Mestre. Heuristic-based approaches for speeding up incremental record linkage. *J. Syst. Softw.*, 137:335–354, 2018.

[6] X. L. Dong and D. Srivastava. Big data integration. In *IEEE ICDE*, pages 1245–1248, 2013.

[7] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*, pages 411–420, 2015.

[8] V. Efthymiou, G. Papadakis, G. Papastefanatos, K. Stefanidis, and T. Palpanas. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Inf. Syst.*, 65:137–157, 2017.

[9] V. Efthymiou, K. Stefanidis, and V. Christophides. Benchmarking blocking algorithms for web entities. *IEEE Trans. Big Data*, 6(2):382–395, 2020.

[10] L. Gazzarri and M. Herschel. Towards task-based parallelization for entity resolution. *SICS Softw.-Intensive Cyber Phys. Syst.*, 35(1):31–38, 2020.

[11] A. Gruenheid, X. L. Dong, and D. Srivastava. Incremental record linkage. *PVLDB*, 7(9):697–708, 2014.

[12] D. Karapiperis, A. Gkoulalas-Divanis, and V. Verykios. Summarization algorithms for record linkage. In *EDBT*, 2018.

[13] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *IEEE ICDE*, pages 618–629, 2012.

[14] L. Kolb, A. Thor, and E. Rahm. Don't match twice: redundancy-free similarity computation with mapreduce. In *Data Analytics in the Cloud*, pages 1–5, 2013.

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

[16] H. Köpcke, A. Thor, S. Thomas, and E. Rahm. Tailoring entity resolution for matching product offers. In *EDBT*, 2012.

[17] G. Papadakis, K. Bereta, T. Palpanas, and M. Koubarakis. Multi-core meta-blocking for big linked data. In *SEMANTiCS*, pages 33–40, 2017.

[18] G. Papadakis, G. Koutrika, T. Palpanas, and W. Nejdl. Meta-blocking: Taking entity resolutionto the next level. *IEEE TKDE*, 26(8):1946–1960, 2014.

[19] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. Boosting the efficiency of large-scale entity resolution with enhanced meta-blocking. *Big Data Research*, 6:43–63, 2016.

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

[21] G. Papadakis, L. Tsekouras, E. Thanos, G. Giannakopoulos, T. Palpanas, and M. Koubarakis. The return of jedai: end-to-end entity resolution for structured and semi-structured data. *PVLDB*, 11(12):1950–1953, 2018.

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

[23] B. Ramadan, P. Christen, H. Liang, R. W. Gayler, and D. Hawking. Dynamic similarity-aware inverted indexing for real-time entity resolution. In *PAKDD Workshops*, pages 47–58, 2013.

[24] A. Saeedi, E. Peukert, and E. Rahm. Incremental multi-source entity resolution for knowledge graph completion. In *ESWC*, pages 393–408, 2020.

[25] W. Santos, T. Teixeira, C. Machado, W. Meira Jr, R. Ferreira, D. Guedes, and A. S. Da Silva. A scalable parallel deduplication algorithm. In *Symposium on Computer Architecture and High Performance Computing*, pages 79–86, 2007.

[26] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi. Schema-agnostic progressive entity resolution. *IEEE TKDE*, 31(6):1208–1221, 2018.