Schema-agnostic Blocking for Streaming Data

Tiago Brasileiro Araújo
Federal University of Campina Grande
Campina Grande, Brazil
tiagobrasileiro@copin.ufcg.edu.br

Kostas Stefanidis
Tampere University
Tampere, Finland
konstantinos.stefanidis@tuni.fi

Carlos Eduardo Santos Pires
Federal University of Campina Grande
Campina Grande, Brazil
cesp@dsc.ufcg.edu.br

Jyrki Nummenmaa
Tampere University
Tampere, Finland
jyrki.nummenmaa@tuni.fi

Thiago Pereira da Nóbrega
State University of Paraíba
Campina Grande, Brazil
thiagonobrega@uepb.edu.br

ABSTRACT

Currently, a wide number of information systems produce a large amount of data continuously. Since these sources may have overlapping knowledge, the Entity Resolution (ER) task emerges as a fundamental step to integrate multiple knowledge bases or identify similarities between entities. Considering the quadratic cost of the ER task, blocking techniques are often used to improve efficiency. Such techniques face two main challenges related to data volume (i.e., large data sources) and variety (i.e., heterogeneous data). Besides these challenges, blocking techniques also face two other ones: streaming data and incremental processing. To address these four challenges simultaneously, we propose PI-Block, a novel incremental schema-agnostic blocking technique that utilizes parallelism (through distributed computational infrastructure) to enhance blocking efficiency. In our experimental evaluation, we use four real-world data source pairs, and highlight that PI-Block achieves better results regarding efficiency and effectiveness compared to the state-of-the-art technique.

CCS CONCEPTS

• **Information systems** → **Entity resolution**; *Semi-structured data*;

KEYWORDS

Entity resolution, streaming data, heterogeneous data, metablocking, incremental processing.

ACM Reference Format:
Tiago Brasileiro Araújo, Kostas Stefanidis, Carlos Eduardo Santos Pires, Jyrki Nummenmaa, and Thiago Pereira da Nóbrega. 2020. Schema-agnostic Blocking for Streaming Data. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3375776

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3375776

1 INTRODUCTION

With the growing use of information, systems are producing a large amount of data continuously. The data provided by these different applications may have overlapping knowledge. For instance, numerous data provided by a sensor network may generate mass similar data. To address this context, Entity Resolution (ER) emerges as a fundamental step to integrate multiple knowledge bases or identify similarities between entities [4]. ER is a very common task in data processing and data integration areas, where different entity profiles, usually described under different schemas, are mapped to the same real-world object [6]. Formally, ER identifies records (the entity profiles) from several data sources (the entity collections) that refer to the same real-world entity. In the context of heterogeneous data, ER faces with two well-known data quality challenges: *volume*, as it handles a growing number of entities; and *variety*, since different formats and schemes are used to represent the entity profiles [4]. To deal with volume, blocking and parallel computing are applied [2, 7]. Blocking groups similar entities into blocks and perform comparisons within each block. Thus, blocking avoids the huge number of comparisons to be performed when the comparisons are guided by the Cartesian product. ER in parallel aims to minimize the overall execution time of the task by distributing the computational cost (i.e., the comparisons between entities) among the resources of a computational infrastructure [1].

The challenge of variety is related to the difficulty of performing the blocking task since heterogeneous data hardly share the same schema and compromises the blocks generation. Therefore, traditional (i.e., schema-based) blocking techniques (e.g., sorted neighborhood and adaptive window) do not present satisfactory effectiveness [13]. In turn, the variety challenge is addressed by schema-agnostic blocking techniques [4]. Among them, Metablocking emerges as the most promising approach [13]: blocks form a weighted graph and pruning criteria are applied to remove edges with weight below a threshold, aiming to discard comparisons between entities with few chances of being considered a match.

Furthermore, it is possible to detach two additional challenges faced by the ER task: streaming data and incremental processing [3, 5, 10, 14]. Streaming data is commonly related to dynamic data sources (e.g., from Web Systems, Social Media, sensors), whose data is sent continuously. Therefore, we assume that not all data, from all data sources, are available at once. For this reason, we have to process (i.e., match) the entities as they appear, also considering

the incremental behavior (in other words, entities already matched previously). Regarding incremental ER, new challenges need to be considered, such as i) *how to manage the entities already processed since they can be infinite?* and ii) *how to execute efficiently the ER task considering the whole stream of entities?* [3, 8].

Notice that the challenges are strengthened when the ER task is considered in the context of heterogeneous data, streaming data and incremental processing, simultaneously. To this end, we propose the Parallel-based Incremental Blocking (PI-Block) technique, a promising schema-agnostic blocking technique capable to incrementally process entity profiles. To our knowledge, there is a lack of blocking techniques for addressing all challenges emerged in this scenario. In this sense, we propose a time window strategy that discards old entities, based on a time threshold. Thus, the PI-Block technique aims to deal with streaming and incremental data efficiently using a distributed computational infrastructure. Overall, the contributions of our work are the following: i) we propose a novel incremental and parallel-based schema-agnostic blocking technique, called PI-Block, that deals with streaming and incremental data; ii) we introduce a parallel-based workflow to perform entity blocking: PI-Block reduces the number of parallel steps of Metablocking, achieving efficiency gains without decreasing effectiveness; iii) we present strategies to avoid unnecessary entity comparisons during the blocking step; and iv) we propose a time window strategy to discard entities already processed based on the time they were sent, avoiding problems related to excessive memory consumption. PI-Block is evaluated against the state-of-the-art technique, regarding efficiency and effectiveness, using four pairs of real data sources.

2 PROBLEM DESCRIPTION

PI-Block receives as input data provided by two data sources D_1 and D_2. Since data is sent incrementally, the ER task processing is divided into a set of increments $I = \{i_1, i_2, i_3, ..., i_{|I|}\}$, such that $|I|$ is the number of increments. Moreover, consider that each increment is associated with a predetermined time interval τ. Thus, the time between each increment should be $T(i_{|I|}) - T(i_{|I|-1}) = \tau$. For each increment $i \in I$, each data source D sends an entity collection $E_i = \{e_1, e_2, e_3, ..., e_{|E_i|}\}$, such that $|E_i|$ is the number of entities. Since the entities can follow different schemes, each entity $e \in E_i$ has a specific attribute set and a value associated to each attribute, denoted by $A_e = \{\langle a_1, v_1\rangle, \langle a_2, v_2\rangle, \langle a_3, v_3\rangle, ..., \langle a_{|A_e|}, v_{|A_e|}\rangle\}$, such that $|A_e|$ is the amount of attributes associated with e. Moreover, in order to generate the entity blocks, tokens are extracted from the attribute values. Namely, all tokens associated to an entity e are grouped into a set Λ_e, i.e., $\Lambda_e = \bigcup(\Gamma(v) \mid \langle a, v\rangle \in A_e)$, such that $\Gamma(v)$ is a function to extract the tokens from the attribute value v.

To generate the entity blocks, a similarity graph $G(X, L)$ is created, in which each $e \in E_i$ is mapped to a vertex $x \in X$ and a non-directional edge $l \in L$ is added. Each l is represented by a triple $\langle x_1, x_2, \rho\rangle$, such that x_1 and x_2 are vertices of G and ρ is the similarity value between the vertices. Thus, the similarity value between two vertices (i.e., entities) x_1 and x_2 is denoted by $\rho = \Phi(x_1, x_2)$. Then, the similarity value is given by the average of common tokens between x_1 and x_2, $\Phi(x_1, x_2) = \frac{|\Lambda_{x_1} \cap \Lambda_{x_2}|}{max(|\Lambda_{x_1}|, |\Lambda_{x_2}|)}$.

In ER, a blocking technique aims to group the vertices of G into a set of blocks denoted by $B_G = \{b_1, b_2, ..., b_{|B_G|}\}$. However, a pruning criterion $\Theta(G)$ is applied to remove redundant comparisons, resulting in a pruned graph G'. The vertices of G' are grouped into blocks $B'_{G'} = \{b'_1, b'_2, ..., b'_{|B'_{G'}|}\}$, s.t. $\forall b' \in B'_{G'}(b' = \{x_1, x_2, ..., x_{|b'|}\}), \forall\langle x_1, x_2\rangle \in b' : \exists\langle x_1, x_2, \rho\rangle \in L$ and $\rho \geq \theta$, where θ is a threshold defined by a pruning criterion $\Theta(G)$. Intuitively, each data increment, denoted by ΔE_i, also affects G. Thus, we denote the increments over G by ΔG_i. Let $\{\Delta G_1, \Delta G_2, ..., \Delta G_{|I|}\}$ be a set of $|I|$ data increments on G. Each ΔG_i is directly associated with an entity collection E_i, which represents the entities in the increment $i \in I$. The computation of B_G, for each ΔG_i, is performed on a parallel distributed computing infrastructure, composed by multiple nodes (e.g., computers or virtual machines). In this context, $N = \{n_1, n_2, ..., n_{|N|}\}$ is the set of nodes used to compute B_G. The execution time using a single node $n \in N$ is denoted by $T^n_{\Delta G_i}(B_G)$, while the time using the whole computing infrastructure N is denoted by $T^N_{\Delta G_i}(B_G)$.

Since blocking is performed in parallel over the infrastructure N, the whole execution time is given by the execution time of the node that demanded the highest time to execute the task for a specific increment ΔG_i: $T^N_{\Delta G_i}(B_G) = max(T^n_{\Delta G_i}(B_G)), n \in N$. Assuming now the streaming behavior, where each increment arrives in each τ time interval, it is necessary to determine a restriction of execution time to process each increment, given by $T^N_{\Delta G_i}(B_G) \leq \tau$. This restriction aims to prevent the blocking execution time from overcoming the time interval of each data increment. To achieve this restriction, blocking must be performed as quickly as possible. As stated previously, one possible solution to minimize the execution time of the blocking step is to execute it in parallel over a distributed infrastructure.

3 STREAMING METABLOCKING

The state-of-the-art blocking techniques do not work properly in scenarios involving incremental and streaming data since they were not conceived to deal with these situations. In this sense, we developed three blocking techniques capable to deal with streaming and incremental data: Streaming Metablocking, PI-Block, and PI-Block-windowed. The Streaming Metablocking technique is based on the same state-of-the-art parallel workflow proposed in [6]. However, Streaming Metablocking was adapted to take into account challenges involving streaming and incremental data. Since the technique considers the incremental behavior, we need to update the blocks in order to consider the new entities that are coming. Using a brute force strategy, after the arrival of a new increment, Streaming Metablocking needs to rearrange all blocks, including blocks that did not suffer any update. Clearly, this strategy is costly in terms of efficiency since it performs a huge number of unnecessary comparisons that have already been performed. To avoid this kind of unnecessary comparisons, Streaming Metablocking applies a store structure provided by Spark Streaming that considers only the data being updated in the current increment. Therefore, Streaming Metablocking only considers the blocks that suffered at least one update for the current increment. The reduction on the number of comparisons helps to minimize the computational cost

of generating the blocking graph and performing the pruning step since the technique evaluates fewer number of entity pairs.

4 PI-BLOCK TECHNIQUE

Metablocking was not originally conceived to deal with incremental and streaming scenarios. This fact motivated us to propose the PI-Block technique, a schema-agnostic blocking technique able to deal with the challenges related to these scenarios efficiently. Unlike the previous parallel-based Metablocking approaches [6], PI-Block uses a different workflow in order to reduce the number of MapReduce jobs and, consequently, minimize the execution time of the task as a whole. In turn, the PI-Block workflow is composed of two MapReduce jobs (see Figure 1), two jobs less when compared with the Parallel Metablocking proposed in [6]. Moreover, it is important to highlight that we improved the workflow proposed in [6] so that the reduction on the number of MapReduce jobs does not impact negatively on the effectiveness results. In other words, the novel workflow does not modify the generated blocks. Figure 1 will be used throughout this section to illustrate the PI-Block workflow, which is divided into three steps: token extraction, blocking generation, and pruning.

Token Extraction Step. In this step, tokens are extracted from data. Each token will be used as a blocking key. Initially, for each increment, blocking receives a pair of entity collections E_{D_1} and E_{D_2} provided by D_1 and D_2. The tokens are extracted from the attribute values of each entity. For each entity e from E_{D_1} and E_{D_2}, all tokens Λ_e associated with e are extracted and stored. This set of tokens Λ_e will be used in the following step to determine the similarity between the entities. Each token in Λ_e will be used as a blocking key and included in the map of blocks B following the format $\langle t, \langle e, \Lambda_e, D\rangle\rangle$, such that t is the blocking key, e represents the entity, Λ_e the set of tokens (i.e., blocking keys) and D the data source that provided e.

In Figure 1, there are two sets of data that represent the entities provided by two distinct increments. For the first increment (top of the figure), D_1 provides entities e_1 and e_2, while D_2 provides entities e_3 and e_4. In the token extraction step, the tokens A, B and C are extracted from entity e_1. For example, in a real-world scenario, the entity e_1 can be represented by $e_1 = \{\langle name, Steve\ Jobs\rangle \langle nationality, American\rangle\}$. Thus, the tokens A, B and C represent the attribute values "*Steve*", "*Jobs*" and "*American*", respectively. From the extracted tokens, all entities sharing the same token are grouped in the same block. Thus, each token is used as a blocking key. For instance, block b_1 is related to token A and contains entities e_1 and e_4 since both entities share the token A. Moreover, in this step, the entities are arranged in the format $\langle e, B\rangle$ such that e represents a specific entity and B denotes the set of blocks that contain entity e.

Blocking Generation Step. In this step, the weight graph is generated to define the level of similarity between entities. Initially, the blocks B generated in the previous step are received as input. For each blocking key k in B, the entities stored in the same block are compared. Thus, the entities provided from different data sources are compared to define the similarity ρ between them. The similarity is defined based on the number of co-occurring blocks (i.e., similar blocking keys) between the entities. After defining the similarity between the entities, the entity pairs are inserted into the graph G, such that the similarity ρ represents the weight of the edge that links the entity pair. The blocks generated in this step are stored in memory to maintain them available for the next increments. In this sense, new entity blocks will be included or merged with the entity blocks previously stored. The blocking generation step is the most costly (in terms of computational costs) in the workflow since the comparison between the entities is performed in this step.

For instance, in Figure 1, block b_1 contains entities e_1 and e_4. Therefore, these entities must be compared to determine the similarity between them. The similarity between them is 1 since they co-occur in all blocks in which each one is contained. On the other hand, in the second increment (bottom of the figure), block b_1 receives entities e_5 and e_7. Thus, in the second increment, block b_1 contains entities e_1, e_4, e_5 and e_7 since all of them share token A. For this reason, entities e_1, e_4, e_5 and e_7 must be compared[1] with each other to determine the similarity between them. However, it is important to detach that entities e_1 and e_4 were already compared in the first increment and, consequently, they must not be compared twice. This would be considered an unnecessary comparison.

There are three types of unnecessary comparisons at the blocking generation step: i) since there is overlapping between the blocks, an entity pair can be compared in several blocks (i.e., more than once) unnecessarily; ii) during the incremental process, some blocks may not be updated. In this sense, entities contained in blocks that did not suffer any updates must not be compared again since this would demand time and memory consumption unnecessarily; iii) updated blocks also contains entity pairs that have already been compared in previous iterations. Therefore, these entity pairs should not be compared again in future increments.

To avoid unnecessary comparisons, three strategies are applied. For type i), the Marked Common Block Index (MaCoBI) [1, 6] condition is used. For each entity pair $\langle e_i, e_j\rangle$, the blocks (i.e., blocking keys) associated with the entities that have an identifier lower than the identifier of the block in question are added to a set of marked blocks (MB). The MaCoBI condition is satisfied when there are no block identifiers in common between the entities e_i and e_j, i.e., $MB_i \cap MB_j = \emptyset$. For type ii), only new blocks and blocks that suffered any update will be considered. To this end, an update-oriented structure is applied to store the blocks previously generated, ensuring that only blocks that have been updated will be taken into account. For type iii), entities previously compared (in previous increments) are marked. Thus, an entity pair must only be compared if at least one of the entities is not marked as already compared. This strategy prevents entity pairs already compared in previous increments from being compared again.

To better understand how PI-Block avoids the three types of unnecessary comparisons, we will consider the second iteration (bottom of Figure 1). For type i), the pair $\langle e_1, e_4\rangle$ should be compared in blocks b_1, b_2 and b_3. To avoid these unnecessary comparisons, the entity pair $\langle e_1, e_4\rangle$ will not be compared in blocks b_2 and b_3 since it does not satisfy the MaCoBi condition. For type ii), only the

[1] Following the restriction that only entities from different data sources should be compared.

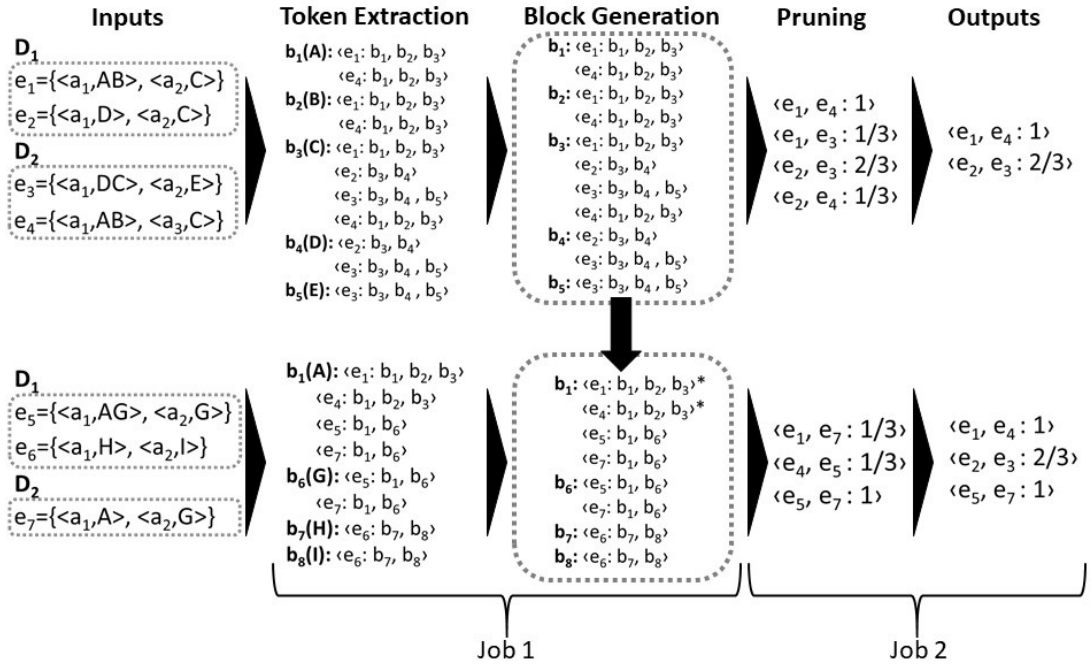

Figure 1: PI-Block workflow.

updated block b_1 and the new blocks b_6, b_7 and b_8 are considered. For type iii), at block b_1, e_1 and e_4 are marked (with the symbol *) since they were already compared in the first increment. Thus, during the comparisons of the entities contained in block b_1, the pair $\langle e_1, e_4 \rangle$ will not be compared again.

Pruning Step. After the comparison between entities, the pruning criterion is applied to discard entity pairs with low similarity values. In the pruning step, a pruning criterion is applied to generate the set of high-quality blocks B'. Regarding the pruning criterion, the works [6, 13] propose different pruning algorithms that can be applied in this step. Particularly, in this work, we apply the WNP-based pruning algorithm [13] since it has achieved better results than its competitors [6, 13]. The WNP algorithm applies the vertex-centric pruning algorithm with a local weight threshold that is given by the average edge weight of each neighborhood. Thus, for each vertex in G, the WNP algorithm calculates the sum of the edge weights and the average of the edge weights. The average of the edge weights is applied as the local pruning threshold. Therefore, the neighborhood entities whose edge weight is greater than the local threshold are inserted in B'. The other entities (i.e., edge weight is lower than the local threshold) are discarded.

Furthermore, we highlight that only the entity pairs compared in the previous step will be considered, implying a significant saving in the computational processing of the pruning step. For example, in the first increment (top of Figure 1), the pairs $\langle e_1, e_4 \rangle$ and $\langle e_2, e_3 \rangle$ should be compared in the ER task since they are considered promising pairs (i.e., with high chances of being considered similar). In the second increment (bottom of Figure 1), only the pair $\langle e_5, e_7 \rangle$ is considered a promising pair (should be compared in the ER task).

5 PI-BLOCK-WINDOWED TECHNIQUE

Considering the incremental challenges, the PI-Block technique faces limitations related to resource consumption (e.g., memory). Since PI-Block stores the blocks previously generated to block the entities incrementally, the consumption of memory may increase infinitely as the increments are processed. This behavior directly results in memory-intensive consumption or problems related to memory overflow. The time window strategy is applied during the blocking generation step since in this step the generated blocks are stored in a data structure, to be used during the next increments. Then, the proposed strategy applies a time window to maintain the entities in the data structure for a certain time interval, preventing excessive memory consumption. However, it is worth mentioning that the application of a time window may affect negatively the effectiveness results since this strategy discards entities which exceed the window time interval. Thus, similar entities cannot be compared because they are not sent at the same time interval.

Considering some entities used in the example depicted in Figure 1, we will describe the PI-Block-windowed technique by means of Figure 2. In this example, three increments are sent at three different times (i.e., T_1, T_2 and T_3). Moreover, the size of the time window (i.e., the time threshold) is given by the time interval of two increments. In the first increment (i.e., T_1), PI-Block receives entities e_1 and e_3. After the blocking generation, blocks b_1, b_2, b_3, b_4 and b_5 are generated. For the second increment (i.e., T_2), PI-Block receives entities e_2 and e_4. Considering the blocks already stored, the entity e_2 is added to blocks b_3 and b_4 and the entity e_4 is added to b_1, b_2 and b_3. Since the time threshold is two increments, the entities provided by the first increment should be discarded. Therefore, for the third increment (i.e., T_3), the entities e_1 and e_3 are removed from

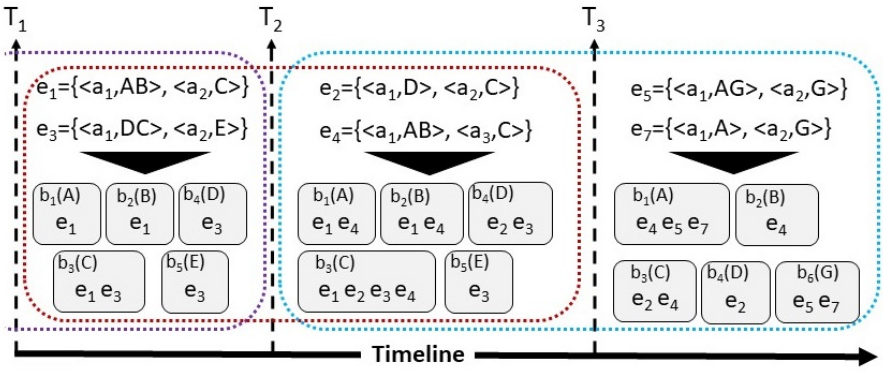

Figure 2: The time window strategy applied to PI-Block.

Table 1: Data sources characteristics.

Pairs of Datasets	$\|D_1\|$	$\|D_2\|$	$\|M\|$	$\|A_1\|$	$\|A_2\|$
Abt-Buy	1,076	1,076	1,076	3	3
Amazon-GP	1,354	3,039	1,104	4	4
DBLP-ACM	2,616	2,294	2,224	4	4
IMDB-DBpedia	27,615	23,182	22,863	4	7

the blocks. Furthermore, it is important to detach that block b_5 is discarded since all entities contained in this block were removed. Finally, entities e_5 and e_7 are inserted into block b_1 and a new block b_6 is generated with the entities e_5 and e_7.

6 EXPERIMENTS

In this section, we evaluate PI-Block[2] and Streaming Metablocking in terms of effectiveness and efficiency. We run our experiments on a cluster infrastructure with 13 nodes (one master and 12 slaves), each one with one core. Each node has an Intel(R) Xeon(R) 1.0GHz CPU, 6GB memory, runs the 64-bit Debian GNU/Linux OS with a 64-bit JVM and Apache Spark 2.0.1[3]. In our evaluation, four real-world pairs of data sources[4] were used. Table 1 shows the amount of entities (D) and attributes (A) contained in each dataset, and the number of duplicates (i.e., matches - M) present in each pair of data sources.

To simulate the streaming behavior on the data, a data streaming sender was implemented. This data streaming sender reads the entities from the data sources and sends the entities to the Kafka producer. The Kafka producer is responsible to provide the data, in a continuous way, to be consumed by the PI-Block technique for each τ time interval (i.e., increment).

To measure the effectiveness of blocking, three quality metrics have been applied: i) Pair Completeness (PC) - similar to recall - estimates the portion of matches that were identified, denoted by $PC = \frac{|M(B')|}{|M(D_1,D_2)|}$, where $|M(B')|$ is the amount of duplicate entities in the set of pruned blocks B' and $|M(D_1, D_2)|$ is the amount of duplicate entities between the data sources D_1 and D_2; ii) Pair Quality (PQ) - similar to precision - estimates the portion of executed comparisons that result in matches, denoted by $PQ = \frac{|M(B')|}{||B'||}$, where $||B'||$ is the amount of comparisons to be performed in the pruned blocks; iii) Reduction Ratio (RR) - estimates the portion of comparisons that are avoided in B' (i.e., $||B'||$) with respect to the comparisons guided by Cartesian product (i.e., $|D_1| \cdot |D_2|$) - is defined by $RR = 1 - \frac{||B'||}{|D_1|\cdot|D_2|}$. PC, PQ and RR take values in the interval [0, 1], with higher values indicating a better result. However, PQ commonly presents low values since it considers all comparisons to be performed (in all blocks) [4, 13]. In terms of efficiency, we measure the whole execution time (i.e., including all steps) of PI-Block considering the execution of all increments. In addition, we evaluate the memory consumption of the distributed infrastructure. Thus, we calculate the average of memory consumed (in percentage) by the nodes that compose the cluster.

To compare PI-Block against Streaming Metablocking, we developed two scenarios of incremental inputs. First, we evaluate both techniques in a scenario where the increment size is the same for all increments. To this end, we set the number of entities per increment as 10% of the whole data source. Thus, for each data source, there are 10 increments containing 10% of entities from the data source. In the second scenario, the increment size varies during the timeline, similar to many real-world streaming data sources. Then, we randomly define the percentage of entities from each data source to be sent in each of the 6 increments. Namely, these percentage values for D_1 are 23%, 8%, 19%, 15%, 22%, 13%, and for D_2 are 12%, 26%, 11%, 20%, 7%, 24%.

For PI-Block-windowed, we vary the size of the time window in order to evaluate the impact of the window size on the PI-Block technique. Thus, we apply the notation $\alpha \cdot \tau$ to determine the window size, such that α determines the number of time intervals τ (i.e., increments) covered by the window.

Efficiency. In this experiment, we evaluate the efficiency of the PI-Block, PI-Block-windowed and Streaming Metablocking techniques. The execution times are given by the average of five executions of each blocking technique. Figures 3 and 4 illustrate the results of the comparative analysis between the PI-Block, PI-Block-windowed and Metablocking techniques for the fixed and varying incremental size scenarios, respectively.

[2]https://github.com/brasileiroaraujo/Streaming/
[3]https://spark.apache.org/
[4]Available in the project's repository.

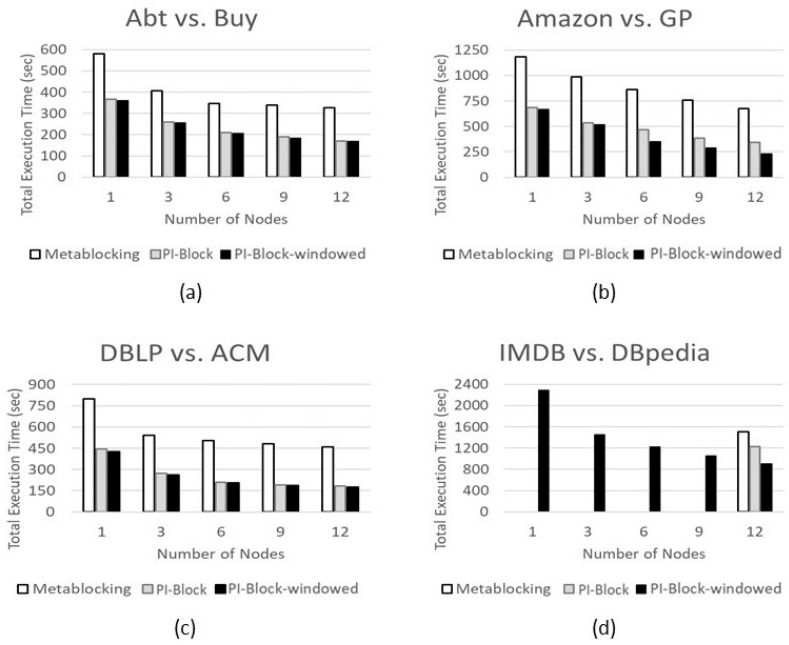

Figure 3: Fixed incremental size scenario: execution time of PI-Block, PI-Block-windowed and Metablocking techniques.

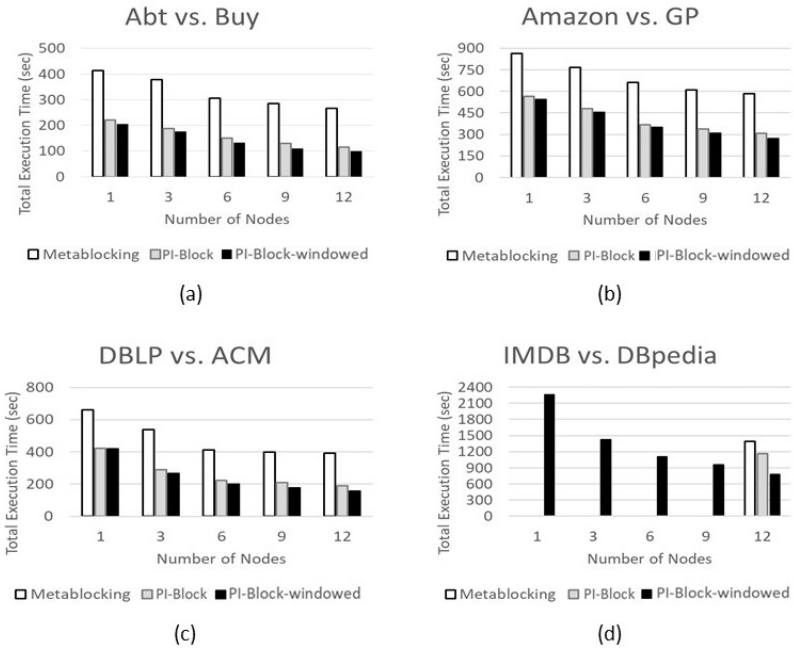

Figure 4: Varying incremental size scenario: execution time of PI-Block, PI-Block-windowed and Metablocking techniques.

We evaluate the execution time (in seconds) varying the number of nodes (one up to 12 nodes) in the distributed infrastructure. For the four data source pairs, depicted in Figure 3 (a-d) and Figure 4 (a-d), it is possible to detach that both PI-Block techniques (with and without the time window strategy) outperformed the Metablocking for all scenarios. This result is directly related to the novel workflow proposed in this work, which requires two MapReduce jobs less than the Metablocking workflow. The new workflow provides a reducing in the execution time of 46% and 45%, on average, for the fixed and varying incremental size scenarios, respectively. By comparing PI-Block-windowed against PI-Block, it is possible to notice a small decreasing in the execution time, since the former applies the time

window strategy (described in Section 5) and, therefore, takes into account a fewer number of entities to be compared.

Regarding IMDB-DBpedia (illustrated in Figures 3 (d) and 4 (d)), PI-Block (without time window) and Metablocking are not able to be executed when less than 12 nodes are used by the distributed infrastructure. It occurs since these techniques consider all the entities sent in all increments. Therefore, the data structure that stores the blocks previously generated requires a large amount of memory of the distributed infrastructure. For this reason, PI-Block and Metablocking have enough memory to be executed only when 12 nodes are used. This limitation leads us to propose the PI-Block-windowed technique which handles a higher amount of entities efficiently. For this data source pair, the maximum window size applied was $4 \cdot \tau$ since the application of bigger window sizes exceeds the memory consumption for one node.

We also evaluate the memory consumption for the fixed incremental size scenario. We vary the number of nodes used by the distributed infrastructure, as depicted in Figure 5. For DBLP-ACM (Figure 5 (a)), it is possible to note that PI-Block-windowed consumes less memory than PI-Block and Metablocking for all number of nodes. PI-Block-windowed consumes less memory due to the application of the time window strategy, which maintains a fewer number of entities on memory. For the pair IMDB-DBpedia (Figure 5 (b)), the memory consumption for PI-Block and Metablocking achieve (on average) around 95% of all available memory in the nodes, when 12 nodes are applied. On the other hand, PI-Block-windowed (with a window size of $4 \cdot \tau$) consumes around 47% of all available memory in the nodes. However, as the number of nodes decreases (consequently, the amount of total memory available decreases), the average memory consumption increases.

Effectiveness. Regarding effectiveness results, Tables 2 and 3 illustrate the PC, PQ and RR metrics for PI-Block, PI-Block-windowed and Streaming Metablocking techniques. For the PI-Block-windowed technique, the effectiveness results are shown for different window sizes, $2 \cdot \tau$ to $8 \cdot \tau$ for the fixed incremental size scenario and $2 \cdot \tau$ to $4 \cdot \tau$ for the varying incremental size scenario. If PI-Block and Metablocking techniques receive the same input and apply the same pruning criterion, they will generate the same output. For this reason, notice that the effectiveness results are also the same for both techniques. Related to the incremental size scenarios (i.e., fixed and varying), PI-Block and Metablocking present the same effectiveness results for both scenarios since the last pruned graph (i.e., after process all increments) produces the same blocks for the fixed and varying scenarios.

For PC, PI-Block and Metablocking present promising results for both incremental size scenarios, achieving more than 96% for all data source pairs. However, since PI-Block-windowed considers only the entities sent between a time interval according to the window size, PC is directly affected. Intuitively, the larger the window size, the better the PC value. This behavior was confirmed for both incremental size scenarios, as illustrated in Tables 2 and 3. For all data source pairs, PC tends to be low for small window sizes and higher for large window sizes. For instance, in the fixed incremental size scenario, PI-Block-windowed achieves PC values above 0.70 with a window size of $6 \cdot \tau$ for all data source pairs. In the varying incremental size scenario, PI-Block-windowed achieves PC values over than 0.75 with a window size of $4 \cdot \tau$ for all data source pairs.

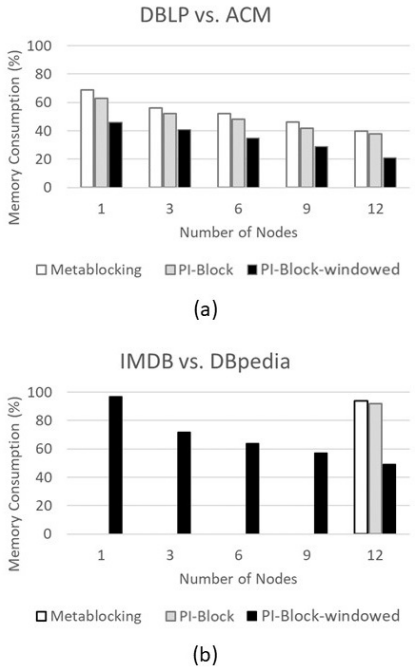

Figure 5: Memory consumption of PI-Block, PI-Block-windowed and Metablocking techniques for (a) DBLP-ACM and (b) IMDB-DBpedia data sources.

In terms of PQ, the blocking techniques achieve different values according to each data source pair. PQ is directly related to the nature of the data sources (e.g., content, number of entities, number of attributes, and entropy of attribute values) that can interfere with the accuracy of the generated blocks [4]. However, PQ is different from Precision commonly used to evaluate the results of ER. PQ evaluates the accuracy of generated blocks. Thus, PQ values achieved by the PI-Block and Metablocking techniques, as depicted in Tables 2 and 3, are satisfactory results for blocking [1, 2, 6, 13].

RR estimates the relative decrease in the number of comparisons conveyed by blocking techniques. Tables 2 and 3 show the RR values for the four data source pairs. RR is fundamental for measuring the efficiency gains of ER since it directly estimates the percentage of comparisons that are avoided after blocking. PI-Block presents promising results in terms of RR, achieving RR values higher than 0.75 for all data source pairs. For IMDB-DBpedia, PI-Block presents an RR value around 0.90. Therefore, PI-Block is able to enhance the efficiency of ER since it reduces up to 90% the number of comparisons to be executed in the ER task.

7 RELATED WORK

Several blocking techniques in stand-alone [2, 13] or parallel mode [1, 6] have been proposed to deal with heterogeneous data. In terms of incremental blocking techniques for relational data sources, [5, 8, 11] propose approaches capable of blocking entities in an incremental way. Thus, these works propose an incremental workflow to the blocking task, considering the evolutionary behavior of data

Table 2: Fixed incremental size scenario: effectiveness results of PI-Block, PI-Block-windowed and Metablocking techniques.

	PI-Block-windowed												PI-Block/Metablocking		
	$2 \cdot \tau$			$4 \cdot \tau$			$6 \cdot \tau$			$8 \cdot \tau$			-		
Data Sources	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**
Abt-Buy	0.72	0.0190	0.96	0.87	0.0125	0.93	0.92	0.0098	0.92	0.96	0.0092	0.91	0.99	0.0091	0,90
Amazon-GP	0.17	$4 \cdot 10^{-4}$	0.91	0.34	$5 \cdot 10^{-4}$	0.84	0.71	$8 \cdot 10^{-4}$	0.82	0.92	$9 \cdot 10^{-4}$	0.80	0.98	0.0010	0,79
DBLP-ACM	0.37	0.0015	0.92	0.60	0.0015	0.86	0.81	0.0016	0.81	0.94	0.0016	0.78	0.97	0.0017	0,76
IMDB-DBpedia	0.27	$3 \cdot 10^{-4}$	0.96	0.58	$3 \cdot 10^{-4}$	0.93	0.85	$3 \cdot 10^{-4}$	0.92	0.95	$3 \cdot 10^{-4}$	0.91	0.98	$3 \cdot 10^{-4}$	0,89

Table 3: Varying incremental size scenario: effectiveness results of PI-Block, PI-Block-windowed and Metablocking techniques.

	PI-Block-windowed						PI-Block/Metablocking		
	$2 \cdot \tau$			$4 \cdot \tau$			-		
Data Source Pair	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**	**PC**	**PQ**	**RR**
Abt-Buy	0.84	0.0135	0.94	0.94	0.0095	0.91	0.99	0.0091	0,90
Amazon-GP	0.21	$4.36 \cdot 10^{-4}$	0.87	0.79	$9.06 \cdot 10^{-4}$	0.81	0.98	0.0010	0,79
DBLP-ACM	0.30	0.00161	0.93	0.76	0.00168	0.84	0.97	0.0017	0,76
IMDB-DBpedia	0.44	$3.19 \cdot 10^{-4}$	0.95	0.86	$3.15 \cdot 10^{-4}$	0.90	0.98	$3.16 \cdot 10^{-4}$	0,89

sources to perform the blocking. However, these works do not deal with heterogeneous and streaming data.

Related to other incremental tasks that present useful strategies for ER, we can detach the works [9, 15] that propose incremental models to address the tasks of Name Disambiguation and Dynamic Graph Processing, respectively. More specifically in the ER context, [12] proposes an incremental approach to perform ER on Social Media data sources. Although such sources commonly provide heterogeneous data, this work ignores the challenges related to such kind of data. To this end, the workflow proposed in this work generates an intermediate schema so that the extracted data from the data sources follow such schema. Thus, [12] differs from our technique since it does not consider the heterogeneous data challenges and does not apply or propose blocking techniques to support ER. Some ER approaches that deal with streaming data, e.g., [10, 14], do not consider incremental processing and therefore discard the previously processed data. Thus, none of them deal simultaneously with the three challenges (i.e., heterogeneous data, streaming data and incremental processing) addressed by our work. However, these works (i.e., [10, 14]) apply Spark Streaming and Kafka platforms, supporting us to apply this kind of platform for streaming data.

8 SUMMARY

Blocking techniques are widely applied to ER approaches as a preprocessing step in order to avoid the quadratic cost of the ER task. In this context, heterogeneous data, streaming data and incremental processing emerge as the major challenges faced by blocking techniques, resulting in a lack of techniques that address all these challenges [2, 4, 5]. In this sense, we propose PI-Block, a novel incremental schema-agnostic blocking technique that utilizes parallelism to enhance blocking efficiency. PI-Block is able to deal with streaming and incremental scenarios as well as minimize the challenges related to both scenarios. Based on the experimental results, we can highlight that PI-Block presents better results regarding efficiency than the state-of-the-art technique (i.e., Streaming Metablocking) without negative impacts on the effectiveness.

REFERENCES

[1] Tiago Brasileiro Araújo, Carlos Eduardo Santos Pires, and Thiago Pereira da Nóbrega. 2017. Spark-based Streamlined Metablocking. In *ISCC*.

[2] Tiago Brasileiro Araújo, Carlos Eduardo Santos Pires, Demetrio Gomes Mestre, Thiago Pereira da Nóbrega, Dimas Cassimiro do Nascimento, and Kostas Stefanidis. 2019. A noise tolerant and schema-agnostic blocking technique for entity resolution. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. ACM, 422–430.

[3] Tiago Brasileiro Araújo, Kostas Stefanidis, Carlos Eduardo Santos Pires, Jyrki Nummenmaa, and Thiago Pereira da Nóbrega. 2019. Incremental Blocking for Entity Resolution over Web Streaming Data. In *IEEE/WIC/ACM International Conference on Web Intelligence*. ACM, 332–336.

[4] Vassilis Christophides, Vasilis Efthymiou, and Kostas Stefanidis. 2015. Entity Resolution in the Web of Data. *Synthesis Lectures on the Semantic Web* (2015).

[5] Dimas Cassimiro do Nascimento, Carlos Eduardo Santos Pires, and Demetrio Gomes Mestre. 2018. Heuristic-based approaches for speeding up incremental record linkage. *Journal of Systems and Software* 137 (2018), 335–354.

[6] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2017. Parallel meta-blocking for scaling entity resolution over big heterogeneous data. *Information Systems* 65 (2017), 137–157.

[7] V. Efthymiou, G. Papadakis, K. Stefanidis, and V. Christophides. 2019. MinoanER: Schema-Agnostic, Non-Iterative, Massively Parallel Resolution of Web Entities. In *EDBT*.

[8] Anja Gruenheid, Xin Luna Dong, and Divesh Srivastava. 2014. Incremental record linkage. *Proceedings of the VLDB Endowment* 7, 9 (2014), 697–708.

[9] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. 2016. iGraph: an incremental data processing system for dynamic graph. *Frontiers of Computer Science* 10, 3 (2016), 462–476.

[10] Kun Ma and Bo Yang. 2017. Stream-based live entity resolution approach with adaptive duplicate count strategy. *International Journal of Web and Grid Services* 13, 3 (2017), 351–373.

[11] Markus Nentwig and Erhard Rahm. 2018. Incremental Clustering on Linked Data. In *IEEE International Conference on Data Mining Workshop (ICDMW)*. IEEE. (to appear).

[12] Bernd Opitz, Timo Sztyler, Michael Jess, Florian Knip, Christian Bikar, Bernd Pfister, and Ansgar Scherp. 2014. An Approach for Incremental Entity Resolution at the Example of Social Media Data.. In *AIMashup@ ESWC*.

[13] George Papadakis, George Papastefanatos, Themis Palpanas, and Manolis Koubarakis. 2016. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking. EDBT.

[14] Xiangnan Ren and Olivier Curé. 2017. Strider: A hybrid adaptive distributed RDF stream processing engine. In *International Semantic Web Conference*.

[15] Alan Filipe Santana, Marcos André Gonçalves, Alberto HF Laender, and Anderson A Ferreira. 2017. Incremental author name disambiguation by exploiting domain-specific heuristics. *Journal of the Association for Information Science and Technology* 68, 4 (2017), 931–945.