QueryER: A Framework for Fast Analysis-Aware Deduplication over Dirty Data

Giorgos Alexiou
galexiou@athenarc.gr
ATHENA Research Center &
National Technical University
Athens
Athens, Greece

George Papastefanatos
gpapas@athenarc.gr
ATHENA Research Center
Athens, Greece

Vassilis Stamatopoulos
bstam@athenarc.gr
ATHENA Research Center
Athens, Greece

Georgia Koutrika
georgia@athenarc.gr
ATHENA Research Center
Athens, Greece

Nectarios Koziris
nkoziris@cslab.ece.ntua.gr
National Technical University
Athens
Athens, Greece

ABSTRACT

This work investigates the challenge of accurately and efficiently answering complex SPJ (Select-Project-Join) queries issued directly on top of dirty data. We introduce QueryER, a novel framework that seamlessly integrates Entity Resolution (ER) into traditional query processing. QueryER performs analysis-aware deduplication by incorporating ER operators into the query plan, enabling efficient entity resolution over dirty data containing duplicate entries with minimal pre-processing time and no manual preparation overhead. Our comprehensive experimental evaluation, conducted using both real-world and synthetic datasets, demonstrates that QueryER adapts to the workload, exhibits sublinear scalability, and consistently achieves high recall performance, outperforming baseline approaches. The results attest to the robustness of QueryER and its suitability for data exploration and analysis workflows. This work opens up new possibilities for more efficient query processing over dirty data, particularly in the realm of data analysis and exploration.

KEYWORDS

query processing, entity resolution, data integration, data exploration, data quality

1 INTRODUCTION

Analysis-aware data processing pertains to an exploratory data analysis type where users apply data integration techniques, such as data augmentation or cleaning [46], during query execution. Such approaches extend SQL by incorporating operators to rectify inconsistent or missing data [17]. *Analysis-aware Entity Resolution* (ER) is a specific instance that augments query results by resolving duplicate entities (records of the same real-world entity) from the underlying tables [1, 3, 4]. Opposite to typical data integration settings that employ ER techniques as a preprocessing step, analysis-aware ER targets interactive scenarios, in which users are interested in subsets of the data and wish to minimize the time-to-analysis by operating directly on duplicate data instead of conducting ER on the entire dataset (batch ER process). This need is prevalent in data aggregators and data lake environments, which present heterogeneous, rapidly evolving, and often overlapping information from multiple sources.

Related state of the art solutions include progressive and analysis-aware ER. *Progressive ER* [36, 41] aims at reducing the time-to-analysis by delivering partially resolved results, which are progressively refined over time. However, its main objective is to speed up the typical batch ER process rather than addressing on demand exploratory scenarios, as it disregards user queries and applies ER to the entire dataset rather than the subset of the data the user is interested in. *Analysis-aware* approaches [1, 3, 4, 42] drive ER-related tasks (e.g., blocking, entity matching) based on a user query. However, they suffer from performance limitations, as they decouple ER operators from the SQL ones; e.g., they apply ER tasks on top of the query results, or as in the case of progressive ER, they evaluate SQL queries on top of (progressive) ER results. For the same reason, they offer support for simple SP (Select-Project) or SPG queries [1, 3, 42], failing to exploit the full capabilities of SQL, such as SPJ (Select-Project-Join) queries or requiring pre-processing steps for more complex analysis (e.g., SPJ queries) [4]. Essentially, *current approaches are not tightly integrated into SQL engines*, leading to significant challenges such as reduced compatibility with existing data infrastructure and an inability to exploit the benefits of query planning for enhanced performance optimization.

In this paper, we seamlessly integrate Entity Resolution (ER) into query engines for on-demand exploratory analysis, standing apart from full-dataset batch ER. This streamlines analysis on multi-sourced, overlapping datasets while reducing overheads like schema matching. We address crucial challenges that involve: (i) *the design and introduction of novel ER-specific query operators, primed for seamless integration within query evaluation pipelines*; (ii) *the establishment of enhanced semantics to bolster both the efficiency and accuracy of entity resolution within SQL queries*; and (iii) *the computation and subsequent incorporation of ER operators' cost into query planning and optimization.* The hallmark of our work lies in the optimized amalgamation of these ER components within the query execution process, delivering substantial improvements in computational efficiency, precision, and overall cost-effectiveness in the area of data exploration and analysis.

In this study, we tackle the outlined challenges through the introduction of *QueryER*, a framework that seamlessly integrates ER operations into the planning and execution of SPJ queries. To accomplish this, we put forth *three innovative (ER-specific) query*

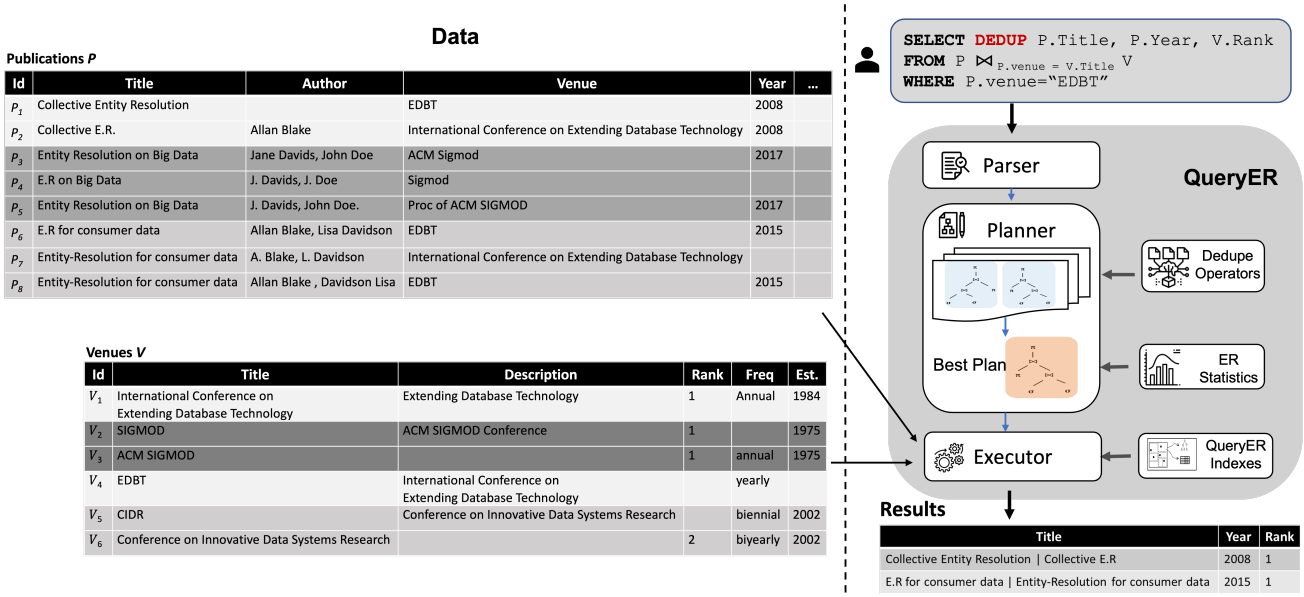

Figure 1: Tabular collections Publications and Venues with data from multiple sources, having duplicates and missing values. On the right, QueryER's architecture overview and an example user query for evaluating results on both tables.

operators that: *(a)* detect and resolve duplicates within a table via a schema-agnostic resolution approach, thereby eliminating configuration overhead; *(b)* perform joins across two or more tables containing duplicate entities; and *(c)* group/merge deduplicated entities into a single representation. Subsequently, we devise a method for integrating these operators into the query execution process. On the ER front, the operators leverage Blocking [6, 9] and Meta-Blocking [35] techniques to eliminate duplicates while minimizing cleaning overhead. Traditionally employed in an end-to-end offline setting, these techniques are recognized for delivering high recall levels [31, 33]. To the best of our knowledge, this is the first study to weave Meta-Blocking techniques into a query engine. Given that our experiments (Table 4) confirmed the dominating cost in query execution to be pairwise entity comparisons, we propose a *cost-based planner* aimed at reducing the number of comparisons among alternative query plans. We have implemented our concepts into a custom SQL query engine using Apache Calcite[1], a framework renowned for building query engines with a diverse range of data sources and extensible customization options. The performance and scalability of our approach were evaluated using real and synthetic data. In summary, the **main contributions** of this paper are:

- The *QueryER* framework, a novel solution for analysis-aware entity resolution over duplicate data with *minimal preprocessing* time and *no configuration overhead.*
- Three novel *operators*, specifically optimized for query execution, that implement *core ER operations* (Blocking/Meta-Blocking, matching, and result grouping) within a query plan pipeline.
- A *cost-based planner* for the efficient execution of joins over data with duplicates.
- A comprehensive experimental evaluation of the proposed approach using real and synthetic datasets.

Outline. Section 2 introduces our motivating example. Section 3 offers a system overview. Section 4 presents the basic concepts and problem formulation. In Sections 5 and 6, we discuss new operators and query planning & evaluation methods, respectively. Section 7 reviews related work, while Section 8 provides the experimental evaluation. Finally, Section 9 concludes the paper.

2 MOTIVATING EXAMPLE

To contextualize our work, consider a data scientist employed by a scholarly data aggregator, such as the Open Academic Graph[2] or Openaire[3], who conducts a range of analyses, including impact assessment and citation analysis. The aggregator gathers information from various publishers, open archives, and data repositories, mapping it to a common schema and categorizing it by type (e.g., publications, venues). Since records can appear in multiple repositories, duplicates are common. As the aggregation is irregular, minimizing time-to-analysis is vital, avoiding batch deduplication with each new source harvested. Therefore, users need to query dirty data on the fly, necessitating duplicate resolution in the results.

A part of the collected information about publications P and venues V is illustrated in Fig. 1. Sets $[P_1, P_2]$, $[P_3, P_4, P_5]$ and $[P_6, P_7, P_8]$ indicate matching records with varying attributes, like author names or missing years, demonstrating data inconsistencies across sources. Similarly, $[V_1, V_4]$, $[V_2, V_3]$ and $[V_5, V_6]$ are sets of matching venues. The user wants to analyze the underlying data and identify publications from "EDBT" conferences along with the venue rank, resulting in the resolution and fusion of any duplicate entries into single records in the results. The user query would be: `SELECT P.Title, P.Year, V.Rank FROM P INNER JOIN V ON P.venue = V.title WHERE P.venue="EDBT"` (execution plan is depicted in Fig. 2).

The query first performs a table scan in P (we assume for simplicity no indexes exist), selects $[P_1, P_6, P_8]$, retrieves V_4 via the join with V, and outputs the projected attributes of joins. However, it omits entities P_2, P_7 and V_1 - duplicates of $P1$, $P6/P8$, and $V4$ respectively - which means that the user misses the *Title* for P_2, the *Year* for P_7, and the *Rank* for V_4. To meet her analysis

[1] calcite.apache.org

[2] https://www.microsoft.com/en-us/research/project/open-academic-graph/

[3] Openaire https://www.openaire.eu

requirements, the user expects the results at the bottom right of Fig. 1, where duplicates are identified and grouped into a single record; for instance, missing values (null) are replaced by existing ones, and contradictory values are all presented to the user. This is an indicative grouping method, and other methods may apply. Currently, batch or progressive ER techniques would be needed to deduplicate both tables pre-query, which is time-consuming and redundant, as only a subset of the data affects the query. Progressive ER, although providing faster partially-resolved results, it is less suitable for exploratory scenarios, as it does not consider user queries and applies ER indiscriminately rather than focusing on the data subset relevant to the user. An ideal solution should (*a*) identify the records selected by the user's WHERE clause, (*b*) deduplicate them against other database records, (*c*) join resolved records, and (*d*) fuse and project the results meaningfully. However, conventional SQL operators do not inherently support such operations, except for the selection in the first step. To remedy this, we integrate *Entity Resolution* into *Query Processing* via operators that incorporate traditional ER techniques into the query planning pipeline. While our motivating example, which serves as the running example throughout this paper, focuses on a scholarly data aggregator, the utility of QueryER is not limited. For instance, during the data preparation phase of ML pipelines, QueryER can swiftly clean and deduplicate features in an analysis-aware fashion. This enables the immediate availability of unique, high-quality training datasets, streamlining ML model development and simplifying the tedious and time-consuming tasks of feature exploration and selection. As a result, data scientists can focus more on critical tasks like model creation and refinement, leading to more efficient and effective outcomes.

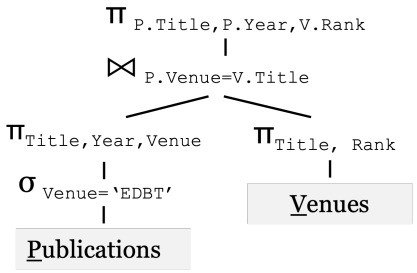

Figure 2: Plan for the query in the motivating example

3 SYSTEM OVERVIEW

QueryER integrates ER into SPJ query execution, enabling users to query dirty data without pre-cleaning. It delivers results equivalent to querying a batch ER deduplicated database but reduces cleaning overhead by deduplicating only data selected by user queries. Additionally, the cost-based planner optimizes operator placement using ER-related statistics to minimize pairwise comparisons. Finally, duplicates are indexed, avoiding redundant ER for similar queries. An overview of our solution is shown in Fig. 1. The user employs SQL syntax to query tables containing duplicates. The `DEDUP` keyword in the beginning of the `SELECT` clause indicates that the results should be resolved for duplicates before the final projection; otherwise, standard SQL semantics apply. Upon parsing the input, the *Query Parser*, generates an abstract query representation that is used by the *Query Planner* to create an initial plan without incorporating the *ER operators*. The plan is then transformed through the insertion and substitution of *ER operators* into a set of alternative query plans. The planner utilizes relational and ER-specific statistics (e.g. selectivity, estimated comparisons, etc.) to devise the best plan. Post-planning, the *Query Executor* retrieves data, executes the optimized query operators, and projects deduplicated results to the user. The execution process, especially the role of the *Deduplicate Operator*, is detailed in Sec. 5.1, demonstrating how entities are processed through an advanced operator pipeline (Fig. 3) that optimizes the deduplication process.

Additionally, QueryER utilizes three in-memory indices per table for ER-related statistics: (*a*) the *Table Block Index* (*TBI*), (*b*) the *Inverse Table Block Index* (*ITBI*), and (*c*) the *Link Index* (*LI*), built during table initialization and maintained in-memory, with *LI* starting off empty. These indices record ER-specific statistics for efficient planning. This approach enhances system efficiency, circumventing the need for on-the-fly indexing during queries. Further insights are in Sections 5 and 6, with initialization times detailed in Sec. 8.1.

4 PRELIMINARIES

This section provides preliminary concepts and the problem formulation. The notations are summarized in Table 1.

Entity Collections and duplicates. Let $D = \{E_1, E_2, ..., E_n\}$ denote entity collections with entities e_i described by attributes $A^E = \{a_1^E, a_2^E, ..., a_k^E\}$. Each entity, identifiable by $e_{id} \in A^E$, corresponds to records in data files (e.g. csv, parquet) or tables. Omitting PKs and FKs enhances compatibility across formats, as these are not uniformly present across datasets. Entities e_i and $e_j \in E$, representing the same real-world object, are *duplicates* ($e_i \equiv e_j$). Joins reflect relations without forming new entities. A collection E is *dirty* if it has duplicates, as seen with $e_{P_1} \equiv e_{P_2}$ in the *Publications* table, Fig. 1.

Entity Resolution (ER). Entity Resolution (ER) identifies and links different representations of the same real-world object [16]. In the context of homogeneous entity collections, where duplicates exist only within individual tables, this crucial process is termed Deduplication [8, 9]. Formally, when working on a dirty collection E, ER produces a set of matches (*linkset*), represented as L_E, which indicates pairs of duplicates in E. The key components of the QueryER's Operators include various ER techniques such as *Blocking*, *Meta-Blocking*, *Matching*, and *Grouping*.

Blocking. Blocking is widely used to scale ER [6, 9] by limiting comparisons to similar entities. The basic concept is the block $b = (e_1, e_2, ...e_{|b|})$, which is identified by a unique Blocking Key (*BK*) and groups entities based on key similarity/equality (e.g., tokens, n-grams etc.). ER restricts pair-wise comparisons between the entities in b instead of all entities in E. A set of blocks is called block collection B, with size $|B|$ denoting the number of blocks it contains, and cardinality denoting the total number of comparisons it involves: $||B|| = \Sigma_{b_i \in B}||b_i||$, where $||b_i||$ is a block's cardinality.

Meta-Blocking. Meta-blocking refines a Block Collection B by discarding redundant and non-matching comparisons while preserving matching ones [35]. It comprises: i) *Block-refinement*, and ii) *Comparison-refinement* methods [33]. From the former, we use *Block Purging (BP)* and *Block Filtering (BF)* [37]. *BP* purges oversized blocks lacking discriminativeness with a block size threshold, while *BF* keeps entities in their n smallest blocks based on a set percentage. In the latter category, *Edge Pruning (EP)* [37] reduces unnecessary comparisons. EP: (i) transforms B into a

Table 1: Notation for Concepts

Symbol	Description
E, e_i	An Entity collection (e.g., a table), a single entity
B, b_i,	A Block collection, a single block
$\|b_i\|$, $\|\|b_i\|\|$	Size (#entities) and cardinality (#comparisons) of b_i
DQ	A Dedupe query
BQ	A Batch Approach Query operating on an E_G
E_σ, $\|E_\sigma\|$	Entities evaluated by a DQ, Size (#entities) of E_σ
$\overline{E}_\sigma$	Duplicate Entities of E_σ
L_E	A collection of matching pairs (e_i,e_j) in E
E_G	Deduplicated grouped entities ($E_\sigma \cup \overline{E}_\sigma$, L_E)
R_G	Result-set of DQ
TBI	Table Block index for entity collection E
$ITBI$	Inverse Table Block index for E
LI	A Link Index for E
QBI	A Query Block Index
$EQBI$	An enriched Query Block Index
$EQBI'$	An EQBI after Meta Blocking has been applied

blocking graph, with entities as nodes and co-occurring pairs as edges, and (ii) weights edges, pruning those below a minimum score. The Meta-Blocking phase operates during query execution, focusing on query-relevant data, maximizing efficiency by skipping unneeded processing.

Entity Matching & Grouping. We treat entity matching and grouping as orthogonal tasks to blocking, adhering to the best practices in literature [8, 9, 31, 33]. In this context, we presume that two duplicate entities, $e_i \equiv e_j$, are detected in B if and only if they co-occur in at least one of its blocks. The performance of Entity Resolution (ER) relies largely on the accuracy of the similarity methods employed for entity comparison (e.g., Jaccard, etc). Upon identifying duplicate entities, the final stage of an ER task merges these duplicates into a single representation. Given a dirty collection E and a linkset L_E, a grouping function produces a set of *deduplicated grouped entities* E_G. This step, relying on the fusion technique used, can employ methods like surrogate key-based grouping or fusing entities' attribute values using domain-specific rules[11]. QueryER remains agnostic to the matching and grouping techniques, offering the flexibility to incorporate a wide array of conventional or advanced methods (as explained in the comparison execution step of Section 5) based on the specific requirements of the data scenario.
Next, we define the two types of queries that are applied on a deduplicated (cleaned) or on a dirty set of entities.

Batch Approach Query (*BQ*). A *BQ* is a SPJ query, which operates on a set of *deduplicated grouped entities* E_G, and returns the result-set R_G. *BQ* corresponds to the case where the underlying collections have been deduplicated via a (batch or progressive) ER process before the user starts the analysis.

Dedupe Query (*DQ*). In contrast, a Dedupe Query *DQ* is also a SPJ query, which operates directly on **dirty** *entity collections* $E \in D$, and returns a result-set, R_G. A *DQ* is *equivalent* to *BQ*, when both employ the same condition expressions in E and E_G, respectively. We consider conjunctive and disjunctive queries where a condition expression can be of the form: α_k^E op `constant` (op can be =,>,<, IN, etc) or $E_1.\alpha_x = E_2.\alpha_y$ (equijoins). We denote by E_σ the set of entities evaluated by *DQ* after all conditions in the `WHERE` clause are evaluated over every E, and by $\overline{E}_\sigma$ the set of entities not evaluated by *DQ* but which are duplicates of E_σ in E. *DQ* identifies the E_σ, $\overline{E}_\sigma$, resolves the duplicates, and produces a set of *deduplicated grouped entities* $E_G = <E_\sigma \cup \overline{E}_\sigma, L_E>$, which is used to return R_G. *DQ* is the type of query we evaluate and optimize in QueryER.

In the context of our example, executing a *DQ* on the datasets publications (*P*) and venues (*V*) for 'EDBT' conferences, initial selection yields entities [*P*1, *P*6, *P*8] from (*P*) and [*V*4] from (*V*), constituting the set E_σ that meets the query's WHERE clause. Crucially, *DQ* extends to identify $\overline{E}\sigma$, the entities not evaluated but which are duplicates of E_σ. This includes [*P*2, *P*7] as duplicates of [*P*1, *P*6/*P*8] and [*V*1] as a duplicate of [*V*4], which are initially overlooked. By resolving these duplicates, *DQ* produces an E_G set, ensuring the inclusion of all relevant data.

Problem Statetement. Our problem is a *query optimization problem* of *DQ*, i.e., optimized execution of *DQ*, while maintaining *correctness*. The *correctness* criterion requires that the entities returned by *DQ* over $E \in D$ are the same with the entities returned by *BQ* over the *deduplicated grouped entities* E_G, i.e., $DQ_{R_G} \equiv BQ_{R_G}$.

To tackle this, we introduce new ER operators detailed in the following section. Our approach efficiently executes the Dedupe Query (*DQ*), while accounting for increased complexity from ER methods, all within the constraints of query-time.

5 DEDUPE QUERY OPERATORS

This section unveils three innovative query operators that form the cornerstone of the *Dedupe Query* implementation: (*a*) Deduplicate, (*b*) Deduplicate-Join and (*c*) Group-Entities.

5.1 Deduplicate Operator

The *Deduplicate operator* is a relational operator that constitutes the key concept of ER integration into query processing. It processes a set of entities $E_\sigma \in E$ derived from the user query (Sec. 4), and finds their duplicates in E. The operator achieves its goal by encapsulating several distinct operations in its pipeline, namely: (*i*) Blocking, (*ii*) Block-Join, (*iii*) Meta-Blocking, and (*iv*) Comparison-Execution as well as relational operations (e.g. Table Scan). The input of this relational operator is a selection E_σ, and the final output is a set of *deduplicated grouped entities* E_G.

The operator utilizes three in-memory hash indices for managing block collections per table. The *Table Block Index TBI* maps blocks to record IDs, sorted in ascending order by block size, while the *Inverse Table Block Index ITBI* acts as its inverse mapping. Additionally, the *Link Index LI* stores each entity's linkset. All these indices are established during table initialization and retained in memory. The *Link Index LI* is initially empty, storing each entity's linkset, and is populated with links resolved by each query. This index, created during table initialization and kept in memory, enables skipping of redundant matching operations for previously matched entities in E_σ. New detected pairs (L_E) are also added to *LI* for subsequent queries, enhancing efficiency as more queries are executed during a user session (see Fig.8/Sec.8). The operations of the Deduplicate Operator are presented next. Fig.3 shows an example for the operator pipeline..

(i) Query Blocking processes the entities in E_σ derived from the user query that are not in *LI*, and constructs an in-memory hash mapping, *Query Block Index QBI*. Both the *QBI* and *TBI* are built using the same blocking function, and given that $E_\sigma \subseteq E$, it follows that $|QBI| \leq |TBI|$. We employ a schema-agnostic configuration of *Token Blocking* [33]. In this method, blocks are

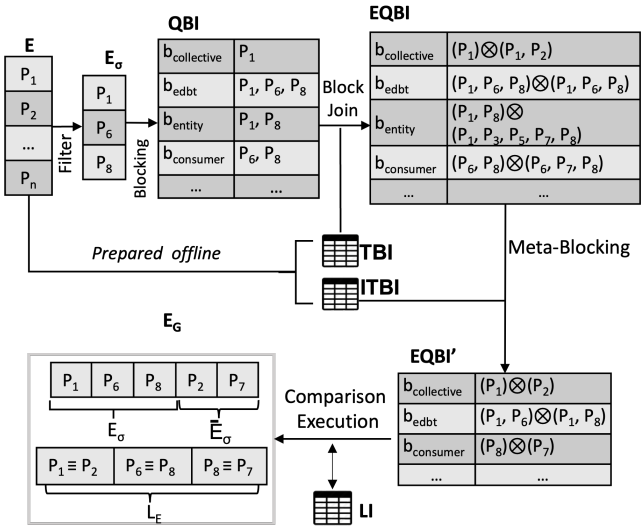

Figure 3: Deduplicate Operator Pipeline. Entities E are the publications in Fig. 1. Symbol ⊗ denotes the inner-product between QBI entity sets and Block-Join results.

formed using tokens extracted from every attribute of a specific entity e, serving as Blocking Keys (BKs) to group records. For example, the `WHERE` clause of the query of Sec. 2 selects entities P_1, P_6, and P_8 from table P. Implementation of *Token Blocking* on these entities generates multiple blocks (Fig.3), i.e., $b_{Collective} = P_1$, $b_{edbt} = P_1, P_6, P_8$, $b_{entity} = P_1, P_8$, $b_{consumer} = P_6, P_8$, etc. (due to space constraints, only four blocks are displayed). Notably, our strategy is not confined to a singular blocking method. Alternative blocking techniques can be employed based on the dataset's specific demands and traits.

(ii) Block-Join performs a *hash-join* between QBI and TBI on their shared Blocking Keys (BKs). This augments the QBI blocks with entities from E present in TBI blocks. The resulting hash mapping, denoted as $EQBI$, captures the dirty subsets of entities that approximate the user's query, potentially including false-positives but not the opposite.

(iii) Meta-Blocking refines $EQBI$ to curtail unnecessary comparisons stemming from Block-Join, ensuring efficiency without compromising effectiveness. We sequentially employ *Block Purging* (BP), *Block Filtering* (BF), and *Edge Pruning* (EP). This sequence is crucial as BP and BF, operating at the block level, mitigate the computational burden for EP, which operates at the finer comparison level [33]. For BP, we establish a global threshold t (computed once offline from TBI) as the first individual size of block that has the same $CC = \frac{\text{blockAssignments}}{\text{totalComparisons}}$ value with the next (smaller) one; blocks are purged when $|b_i| > t$ [38]. BF, utilizes $ITBI$, where for each entity the associated blocks are pre-sorted by size. For every entity $e_i \in E\sigma$ such that $e_i \Rightarrow e_i \in ITBI$, the entity is retained in its initial smallest n blocks, where $n = p \cdot |B|$ and p is a predefined value within the range $[0, 1]$ [38]. For EP, we construct the blocking graph using $EQBI$ and determine edge weights based on the Jaccard similarity of their associated block indices. Essentially, we focus on counting the common blocks between two entities to ascertain similarity. To decide if two entities warrant comparison, we define a threshold using their block associations from $ITBI$. If the common blocks between two entities reach $a \times 100\%$, with a predefined in the interval (0,1], of the blocks of the entity with the least associations, they are considered for comparison [33]. This threshold ensures we base our comparisons on significant block overlaps, concentrating on entities with substantial similarities in $ITBI$. The final outcome is the refined $EQBI$ denoted $EQBI'$.

(iv) Comparison-Execution performs the surviving comparisons from Meta-Blocking between the entities in E_σ and $EQBI'$. This process identifies duplicates exclusively within the initial selection E_σ, hence significantly reducing the number of comparisons. Embracing a schema-agnostic approach, we perform pairwise comparisons of all corresponding attributes. While this method does not prioritize attributes with higher potential duplicate likelihood (e.g., identifiers), it simplifies the process by removing the need for user configuration. Importantly, QueryER ensures that no entity is compared with itself and no comparison is repeated. Being *matcher-agnostic*, QueryER can integrate a wide array of matching techniques, from conventional schema-based methods to advanced Machine Learning models. This versatility enables sophisticated ML algorithms or pre-trained classifiers, providing robust performance across diverse data scenarios. By assuming transitivity in matches and using the union-find algorithm[5] to form disjoint-sets of matching entities, we effectively avoid additional comparisons. The outcome is a set of deduplicated grouped entities (E_G).

DQ Correctness. We define the correctness of a Dedupe Query (DQ) relative to an equivalent Batch Approach Query (BQ) as follows: A Dedupe Query DQ executed over an Entity collection E producing DQ_{R_G} is considered correct if $DQ_{R_G} \equiv BQ_{R_G}$, where a Batch Approach Query BQ is executed over E_G yielding BQ_{R_G} (see Sec. 4/Problem Statement). This equivalence is critical for validating the effectiveness of our deduplication process. It rests on the deterministic functioning of the following ER processes (detailed above in paragraphs (i) to (iv)):

(i) *Blocking and Block-Join*: The initial step involves segregating entities into blocks using a predefined blocking function, which is consistent across both BQ and DQ. The Block-Join operation then merges these blocks based on matching Blocking Keys, ensuring that any entity $e \in E_\sigma$ is grouped identically in both $EQBI$ and TBI, denoted as $B_{EQBI}(e) \equiv B_{TBI}(e)$. This identity is also verified by the inverse index, as $ITBI(e) \equiv B_{EQBI}(e) \equiv B_{TBI}(e)$. This step is crucial for maintaining the integrity of the entity resolution process, as it guarantees that subsequent operations, such as Meta-Blocking and Comparison-Execution, act on a consistent set of entities.

(ii) *Meta-Blocking*: The three Meta-Blocking methods (detailed above in (iii) Meta-Blocking) are designed to prune blocks (Block Purging BP), entities (Block Filtering BF), and comparisons (Entity Pruning EP) within blocks. The BP threshold, precomputed based on TBI, ensures that a block purged from TBI due to BP will similarly be excluded in DQ on $EQBI$. The BF method filters entities from blocks using a predefined filtering value. Given $B_{EQBI}(e) \equiv B_{TBI}(e)$, this guarantees that an entity e is consistently included in the same blocks for both BQ and DQ processing. Additionally, EP relies on identical blocking graphs for each entity $e \in E_\sigma$ in both approaches, further validated by $B_{EQBI}(e) \equiv B_{TBI}(e)$. By employing the blocking graph and $ITBI$, comparison weights and thresholds are established based on common block associations and a predefined value α, which, as previously mentioned, determines the minimum proportion of common blocks required for two entities to be considered for comparison [33]. Threshold α, set in the interval (0,1], ensures that pruning criteria are consistently applied in both the DQ and the BQ, maintaining the integrity of the deduplication process.

(iii) *Comparison-Execution*: This step utilizes a deterministic matching function that applies the same criteria and thresholds across both batch and dedupe processing. The consistency of this function is vital; it ensures that any pair of entities identified as duplicates in one process (batch or dedupe) will be recognized as such in the other. This determinism is the cornerstone of asserting the equivalence of DQ_{R_G} and BQ_{R_G}, as it guarantees that the outcome of the deduplication process is reliable across different data processing paradigms.

The meticulous design and execution of these ER processes underpin the robustness of our approach, demonstrating that *DQ* is effective and correct relative to *BQ*. This equivalence is not only a testament to the accuracy of the process but also highlights the adaptability of our framework to diverse data scenarios.

Cost Analysis. *Deduplicate operator's* cost encompasses I/O, *QBI* construction, Block-Join, Meta-Blocking, and Comparison-Execution. I/O considerations involve (i) the initial E_σ set determined by the query and (ii) entities in $EQBI'$ after Meta-Blocking. All indices are hash-based and reside in-memory, ensuring $O(1)$ access time. The computational and space costs are distributed among the *Deduplicate Operator* components and the in-memory indices, emphasizing on the ER operations.

- *QBI* is created by iterating over all attributes A^E of E_σ entities; for homogeneous datasets, the cost of *QBI* is mainly determined by $|E_\sigma|$ and $|A^E|$, i.e., $O(|E_\sigma|\times|A^E|)$.
- The cost of the Block-Join and the block-refinement methods is determined by the number of blocks in *TBI* and *QBI*. For Block-Join, it is $3(|QBI|+|TBI|)$, since we perform a hash-join and the blocks are traversed only once each. The cost of *Block Purging* (BP) is $O(|EQBI|)$. The cost of *Block Filtering* (BF) is $O(|EQBI|\times|b_i|)$, since for each block $b_i \in EQBI$ we iterate over all its entities.
- *Edge Pruning* (EP) and Comparison-Execution operate on the entity pairs within each block. To estimate the number of comparisons within a block $b_i \in EQBI$, we compare the intersection of entities in E_σ and b_i. Given that each entity comparison is done *only once* and self-comparisons are avoided, the comparisons are: $|E_\sigma \cap b_i|\times(|b_i|$ - $(|E_\sigma \cap b_i|+1)/2)$ and for the entire *EQBI* is $\Sigma_{b_i \in EQBI}\ |E_\sigma \cap b_i|\times(|b_i|$ - $(|E_\sigma \cap b_i|+1)/2)$. The resolution function's (e.g., Jaro-Winkler[22]) cost is multiplied by this number to compute comparison execution cost. In practice, the total number is even smaller, as multi-block comparisons are executed once, and we compute the linksets of entities in E_σ not in Link Index *LI*. Hence, the dominant comparison execution cost tends to decrease significantly with each query.
- The space complexities are closely tied to the core operations. The *TBI* has a complexity of $O(|E| \times |A^E|)$, accommodating entities and their attributes, while *ITBI* simplifies to $O(|E|)$, mapping entity IDs to tokens (*BKs*). The *LI*'s complexity, $O(d)$, depends on the count of distinct duplicate pairs. Post Block-Join, *QBI* is dismantled, and *EQBI* combines *QBI*'s structure with *TBI*, yielding a worst-case complexity of $O(|TBI|)$.

5.2 Deduplicate-Join Operator

Echoing relational algebra's join operation with an emphasis on data quality, this operator processes two entity sets by first fully deduplicating one upfront. It then performs a relational join between the deduplicated set and the second, initially dirty entity set to identify joining entities. These entities are subsequently deduplicated against the second set, culminating in a second join operation between two fully deduplicated sets to capture all possible join combinations. This method, alongside the consistent application of the *Deduplicate Operator*, underpins *DQ Correctness*, with detailed procedures outlined in Algorithm 1 and Procedure 1, accounting for the position of the dirty set within the query-tree.

(i) Dirty-Right. It takes as input two entity sets (`Left`, `Right`), the join type, and the related attributes from the query plan. The left set is deduplicated, assigned to LE_G, and projects necessary attributes (*line 2*). The right set is assigned to E_σ (*line 3*). A relational join $LE_G \bowtie E_\sigma$ filters non-joining entities, forming E'_σ (*line 4*), capturing all join combinations. From these, entities originating from the right set are deduplicated against the entire right set, resulting in RE_G (*line 5*). Finally, Procedure 1 joins the resolved sets (*line 11*), creating the output. Both joins consider resolved entities and their link-set to ensure comprehensive joins.

(ii) Dirty-Left. This mirrors the Dirty-Right approach but with roles of `Left` and `Right` sets reversed.

The *Deduplicate-Join Procedure* (Procedure 1) accepts the LE_G and RE_G sets and returns their join, JE_G. Beginning at *line 4*, it iterates through LE_G. If an entity has not been *visited*, it retrieves its duplicates from the L_E and flags it as *visited* (*lines 6-7*). For each duplicate, it identifies joining entities in RE_G, and their duplicates (*lines 8-12*). After identifying all joining entities from both tables, it performs the Cartesian product of these sets (*line 14*), generating the new JE_G set.

The *DQ* Correctness is satisfied since the join of Procedure 1 is always performed on two deduplicated sets (recall the definition of correctness in Section 4) and thus the operator always produces consistent output, i.e., E_G, crucial for multi-join query plans.

Algorithm 1: Deduplicate-Join Operator

Input: $Left$, $Right$, $JoinType$, $AttrsToProject$
Result: Joined E_G

1 **if** *JoinType is DIRTY-RIGHT* **then**
2 $\quad LE_G \leftarrow$ Project(Deduplicate($Left$), $AttrsToProject$)
3 $\quad E_\sigma \leftarrow Right$
4 $\quad E'_\sigma \leftarrow discardRight(E_\sigma \bowtie LE_G)$
5 $\quad RE_G \leftarrow$ Project(Deduplicate(E'_σ), $AttrsToProject$)
6 **else if** *JoinType is DIRTY-LEFT* **then**
7 $\quad RE_G \leftarrow$ Project(Deduplicate($Right$), $AttrsToProject$)
8 $\quad E_\sigma \leftarrow Left$
9 $\quad E'_\sigma \leftarrow discardLeft(E_\sigma \bowtie RE_G)$
10 $\quad LE_G \leftarrow$ Project(Deduplicate(E'_σ), $AttrsToProject$)
11 **return** *DeduplicateJoinProcedure(LE_G , RE_G)*

Cost Analysis. The *Deduplicate-Join Operator* comprises the *Deduplicate Operator* and two hash-joins: (i) $E_G \bowtie E_\sigma$ between a clean and a dirty entity set, and (ii) $LE_G \bowtie RE_G$ between the two clean sets. *Deduplicate-Join Operator's* cost is presented in Sec. 5.1. The hash-joins costs are: $3(|LE_G|+|RE_G|)$ and $3(|E_G|+|E_\sigma|)$, respectively, ensuring efficient processing of large datasets.

5.3 Group-Entities Operator

The *Group-Entities Operator* consolidates *deduplicated grouped entities* E_G into a single record per entity, yielding a result-set R_G, before the final *Project*. It acts as an aggregate function, grouping all attribute values $\forall e_i \equiv e_j$ by concatenation. While we do not focus on data merging techniques for fusing matching entities,

Procedure 1: Deduplicate-Join Procedure

```
1  Function DeduplicateJoinProcedure(LE_G, RE_G)
2      JE_G // Joined E_G
3      visited = set()
4      for e ∈ LE_G do
5          if e ∉ visited then
6              E_left ← e ∪ LE_G.L_E.get(e)
7              visited.addAll(E_left)
8              for e_l ∈ E_left do
9                  E_joined ← e_l ⋈ RE_G
10                 for e_r ∈ E_joined do
11                     E_right ← e_r ∪ RE_G.L_E.get(e_r)
12                 end
13             end
14             JE_G.add(E_left × E_right)
15         end
16     end
17     return (JE_G)
```

we group them for a simplified presentation of the final projection. For instance, given attributes "EDBT" and "International Conference on Extending Database Technology" in matching entities, a "hyper-entity" with [EDBT | International Conference on Extending Database Technology] is created. Still, any other aggregate/fusion function (e.g., VOTE [42] or group_concat) could be used on E_G, to generate R_G.

6 DQ PLANNING & EVALUATION

This section outlines strategies for planning and evaluating *Dedupe* queries, presenting two solutions: a naive approach using fixed plans, and an advanced method optimizing comparison execution cost through ideal plan selection.

6.1 Execution based on fixed plans

A straightforward way to answer the SPJ query of the motivating example (Fig. 2) is to first deduplicate the entire tables P and V, and then join the two clean record sets to answer the query. This is the equivalent of placing the *Deduplicate Operator* directly above the table scan (Fig. 4a). However, this approach, like the batch, is costly as it cleans the entire table before filtering (e.g., p.venue="EDBT"). An obvious plan enhancement places the *Deduplicate Operator* above the filter on the left tree branch (Fig. 4b) hence reducing the number of entities $|E_\sigma|$ that will initially feed the *Deduplicate Operator*. Next, we observe that cleaning the entire V is not needed, as only a few venue records will eventually join with the publications, selected and cleaned by the left branch. Eliminating non-joining venues before cleaning can further decrease computational costs.

6.2 Comparison-Based Optimization

Considering the observations above, we introduce a *cost-based* method to significantly enhance query performance by minimizing unnecessary comparisons prior to the computationally demanding *Comparison-Execution*.

6.2.1 Query Planning. The *Comparison-based Optimization* begins with the optimal non ER-enabled query plan (Fig. 2). This

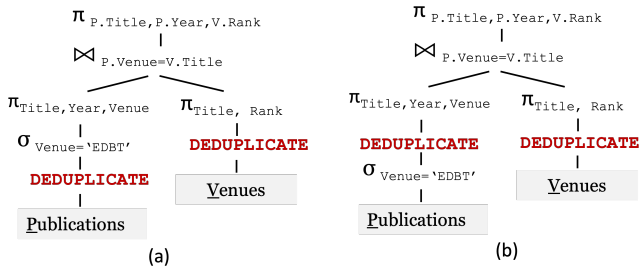

Figure 4: Two naive Dedupe query evaluation plans: Left applies DEDUPLICATE operator to both tables before predicates selection, right applies it only to filtered query records.

plan is then transformed into one similar to Fig. 5a or 5b (depending on statistics) by inserting the *Deduplicate Operator*, *Group-Entities Operator* and replacing the initial Join Operator with the *Deduplicate-Join*. The objective is to eliminate unnecessary comparisons before *Comparison-Execution* while maintaining correctness and optimizing *Join Ordering* when multiple joins are involved. Query Planning works as follows:

(i) Statistics Computation. First, we compute the query statistics; i.e., the estimated number of comparisons and the estimated number of join predicates. We use the indexes *TBI*, *ITBI* and *LI* (Sec. 3). *ITBI* is a hash index mapping record IDs to blocks.

We estimate comparisons considering that condition expressions in the WHERE clause identify Blocking Keys (*BK*) in the *TBI*. In particular, we create a set of *BKs* based on the condition expressions in the WHERE clause which is a subset of *BKs* in *TBI*. Then, for each key, we get the corresponding block $b_i \in TBI$. Based on the operators *AND*, *OR* of the clause, we derive the estimated selected set $E_s \approx E_\sigma$. For all entities e from E_s that are not in *LI* ($e \in E_s \backslash LI$), we retrieve the related blocks from *ITBI*, thereby forming a block collection *SB* to approximate *EQBI*. We then apply the block purging and filtering algorithms (discussed in Sec.5.1/Meta-Blocking) to approximate post-meta-blocking comparisons (*EQBI'*). The final number of estimated comparisons for this table is $C = \Sigma_{SB_{b_i} \in SB} |E_s \cap SB_{b_i}| \cdot (|SB_{b_i}| - (|E_s \cap SB_{b_i}| + 1)/2)$. This estimation helps to determine which table yields the largest number of comparisons. Since the cost of estimating the output of the *Edge Pruning* (discussed in Sec.5.1) is very high, we terminate our calculations at the filtering step, where a safe conclusion about the inequality can be drawn.

To estimate Join predicate numbers, we gauge the size of E_G pre- and post-join. An offline pre-cleaned sample informs the duplication factor df during initialization, approximating duplicates' prevalence. For example, if a table's sample $|S| = |E_\sigma| = 800$ entities expands to 1000 entities post-deduplication, we infer a 20% duplication rate (see Table 6 for Q-errors on stats). We also precompute the join percentage between every table pair to anticipate post-join E_G size reduction. For example, a 20% join between P and 50% of V implies a corresponding decrease in their E_G sizes.

(ii) Plan Creation. The planner identifies optimal operator placement considering *DQ* Performance. For *SP* queries, the *Deduplicate Operator* is added atop the *Filter*, reducing the number $|E_\sigma|$ of entities initially feeding the *Operator*. For SPJ queries, one query tree branch must be deduplicated before the Join for *Correctness*. The planner uses statistics to place the *Deduplicate Operator* on the branch yielding the fewest comparisons. For instance, based on the example dataset in Fig.1, initial cleaning of

Figure 5: Comparative DEDUPLICATE-JOIN plans from Fig.4b. The selection process aims for minimal comparisons, with the left plan optimized for select publication queries. Projection is not pushed down prior to deduplication, preserving data essential for accurate duplication. The right plan prioritizes deduplicating Venues before joining with Publications. Detailed explanation in Sec.6.2

table V yields 15 comparisons, while cleaning table P (post filter) yields 18, Therefore, the planner opts for the plan depicted in Fig. 5b over Fig. 5a. This approach reduces total comparisons after *Deduplicate-Join* because the *Operator* forms a *QBI* for the dirty entity-set from entities joining with the deduplicated set. Based on this, the planner selects the appropriate *Deduplicate-Join Operator* type, optimizes join order using statistics to minimize I/O and memory usage, and positions the *Group-Entities Operator* to produce grouped results R_G.

6.2.2 Query Execution. The plan executes sequentially as designated by the planner, using the Iterator Interface to pass operator outputs to their parent nodes. As Fig. 5b demonstrates execution begins with the *Deduplicate Operator*, after scanning table V. Subsequently, the *Project Operator* is applied, in line with our schema-agnostic approach to ensure essential data for deduplication is retained (see Sec.5.1). Following this, table P's *Filter Operator* selects entities per query requirements, preceding the *Deduplicate-Join Operator*. After the *Join*, *Group-Entities Operator* assembles matched pairs into *GroupedEntities* for each duplicate set. Finally, the Projection returns requested attributes.

7 RELATED WORK

Entity Resolution. Entity Resolution (ER) is a well-studied field in database research [21, 25, 34], with typical ER approaches using blocking methods to scale to large datasets by comparing similar entities. Blocking enhances precision at the cost of recall. Traditional methods cater to structured data abide by specific schemas (schema-based blocking) [9], but struggle with highly heterogeneous data. To address this, schema-agnostic blocking has been introduced, treating each token from every entity value as a blocking key, overcoming heterogeneity but generating overlapping blocks that cause redundant comparisons [31, 35]. Meta-blocking mitigates this by assessing block-to-entity relationships, removing unnecessary comparisons [35, 38]. MinoanER [13] uses schema-agnostic techniques for ER in heterogeneous data but depends on batch processing. In contrast, QueryER operates efficiently during query-time.

Progressive ER. Progressive data integration, first introduced by Madhavan et al. [27], consolidates Web data as much as possible within limited resources and time. It has been applied to ER [47] and schema mapping [40]. In the context of ER, progressive methods prioritize matches based on likelihood, ordering records by similarity for progressive comparison within a window, neighborhood, or sampling of blocks [14, 15, 23, 36, 41, 47]. These dynamic approaches yield incremental entity resolutions and partial outcomes until completion. However, they are unsuitable for SQL SP/SPJ queries due to potential inaccuracies from applying resolution functions on partially identified matches, leading to variable results throughout the process and adding complexity to execution.

Analysis-aware processing. Recent years have seen the integration of Entity Resolution with Query Processing for SQL-like queries on erroneous data. Yet, many such methods fall short in dealing with broader SPJ queries [2], or lack optimization for selection queries, including range queries [7]. Some techniques [1], only cater to basic SP queries on single entity collections, bypassing planning and optimization. Other strategies address probabilistic databases for aggregation queries on single [20, 39, 44] and multiple tables [19]. Approaches like Sample-and-clean [46] rectify aggregate queries on dirty data, while CleanDB [18] opts for an all-inclusive cleaning in a distributed environment. Daisy [17] focuses on managing integrity constraints with probabilistic repairs for SPJ queries, differing from QueryER's emphasis on ER techniques. To the best of our knowledge, the closest works to ours are *QuERy* [4], *QDA* [3], and *BrewER* [42]. *QuERy* manages SPJ queries over dirty data using blocking and sketch summaries. It constructs a query tree for clean records and sketches, leading to data cleaning if a sketch passes predicates. Despite the absence of source code or datasets for direct comparison, *QuERy*'s execution time exceeds 100 seconds for low selectivity SPJ queries on ($C_{|80070|} \bowtie M_{|1237|}$) (reported in Fig. 19 in [4]), while QueryER performs similar queries under 100 seconds, even for complex ones (e.g., $OAGP_{2M} \bowtie OAGV_{130K}$ in Fig.10). QueryER's efficiency stems from using Meta-blocking and block-join to reduce overhead and retrieve potential dirty subsets, avoiding repeated sampling and frequent cleaning. On the other hand, *QDA* [3] simplifies cleaning for SP queries by identifying entities that satisfy selection predicates without full resolution. Yet, it's less equipped for handling the wider SPJ queries and has limited resolution function support. *BrewER* [42] progressively evaluates SP aggregation queries, employing block merging and transitive closure calculations, and approximating ORDER BY clause values for progressive entity emission. Both *BrewER* and QueryER ensure the correctness of the results and are anagnostic to the matching and blocking methods. However, *BrewER* and *QDA* target simple SP aggregation queries with a focus on duplicate analytics.

ER Pipelines. Traditional ER pipelines typically prioritize offline data processing, employing blocking and progressive outputs to balance performance with accuracy. QueryER sets itself apart by transforming state-of-the-art ER techniques into query operators within the query execution pipeline, improving performance through strategic query planning and comparison optimization. This marks a significant shift away from the sole reliance on offline processing inherent in batch and progressive methods. Additionally, QueryER efficiently executes and refines joins, leveraging a cost-optimized planner, to outperform many analysis-aware systems.

8 EXPERIMENTAL EVALUATION

We evaluated the effectiveness, the efficiency and the scalability of our approach on several real and synthetic datasets.

8.1 Experimental Setup

We have implemented QueryER using Apache Calcite framework[1] in Java version 17. The experiments were performed on a desktop computer with Intel i7 (3.4GHz) and 64GB of RAM. All measurements were repeated 10 times and the average value is reported. All resources, including a link to an online demo, are available in the provided artifact repository[4].

Datasets. Our experimental analysis involves the following datasets (Table 2 summarizes their technical characteristics[5]):

(*a*) *Real datasets*: Widely used in literature[13, 24], *DBLP-Scholar*[24] (**DSD**), contains (single-table) bibliographic records from DBLP and Google-Scholar. We also utilize the Open Academic Graph[43, 45] (**OAG**), a knowledge graph combining MAG[6] and AMiner's[7] billion-scale academic graphs. This includes Paper (**OAGP**) and Venue (**OAGV**) data, featuring 5 different size variations (**OAGP200K-2M**) specifically created for scalability testing (ground-truth provided). (*b*) *Real datasets modified to include duplicates*: The *Organisations* (**OAO**) and *Projects* (**OAP**) datasets are obtained from the OpenAIRE project [28]. They contain records for research organizations and the corresponding projects that they participate in. Both datasets have been modified using *febrl* [10] to include 10% duplicate records, providing a more realistic scenario for experimentation.
(*c*) *Synthetic datasets*: The datasets (**PPL200K-2M**) were also created using *febrl*. Duplicate-free people records were produced based on real-world frequency tables, adding an extra attribute to assign organisations (ids from OAO) to person. Duplicates were generated based on real-world error characteristics, resulting in datasets with 40% duplicate records. Each record can have up to 3 duplicates and 4 modifications, with a maximum of 2 modifications per attribute.

Evaluation Measures. Performance is assessed by: (*a*) *Pair Completeness (PC)*, which gauges recall of $EQBI'E\sigma$ against the ground-truth (GT), as $PC = \mathcal{D}(EQBI'E\sigma)/GT(EQBI'_{E\sigma})$, representing the rate of duplicate occurrence within blocks; (*b*) *Total execution time (TT)* for assessing overall workload durations; (*c*) *Comparisons*, total executed comparisons from queries; (*d*) *Pair Quality (PQ)*, precision post-*Meta-Blocking*, as $PQ = GT(L_E)/Comparisons$; (*e*) *Q-Error* [29], measuring deviations between predicted and actual values.

Competitors and Configurations. Our experiments are divided into SP and SPJ classes based on query types. *In the SP class*, we evaluated QueryER against BrewER[42] and QDA[3] - which only support SP queries -, assessed scalability, the impact of *LI* and Meta-Blocking configurations on performance. As auxiliary structures without alternative configurations, *TBI* and *ITBI* were not directly assessed.

In the SPJ class, we evaluated the performance of our *cost-based planner* against two naive fixed-query-plan solutions. We did not evaluate Meta-Blocking separately, as it is integrated in the *Deduplicate Operator*, called within the *Deduplicate-Join Operator* (Sec. 5.2). Likewise, *LI* is independent of the query class and was tested only once. For all configurations, we used fixed

[4] github.com/VisualFacts/queryER
[5] A more detailed table is also provided at github.com/VisualFacts/queryER
[6] academic.microsoft.com
[7] www.aminer.cn

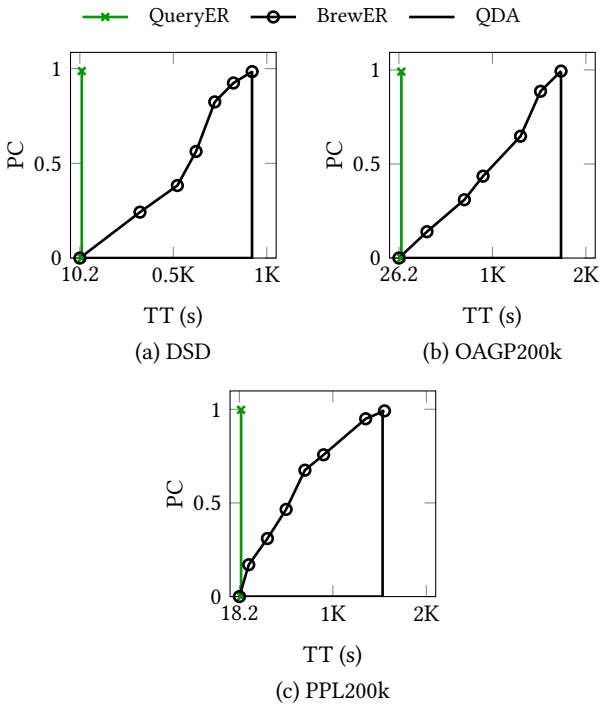

Figure 6: Recall (PC) vs. time (TT) between QueryER, BrewER and QDA in real & synthetic datasets. QueryER achieves the same PC in less than 30s in all three datasets, being two orders of magnitude (100X) faster than competitors.

Token-Blocking and Meta-Blocking, along with the *Jaro-Winkler* similarity function. Recall that our approach is *matcher-agnostic* permitting alternative methods (e.g., pre-trained classifiers) to classify entity pairs as duplicates [12, 26, 30, 32]. Appropriate synthetic and real datasets were chosen per experiment, considering their technical characteristics.

Initialization. QueryER's one-off initialization phase establishes relational and ER-specific statistics alongside index creation. Table 3 presents our benchmarking for all datasets (Table 2). With median block index times typically under two minutes, the system transitions quickly to readiness. The modest increase from the 25^{th} to 75^{th} percentiles indicates a consistent performance, and the low standard deviation reflects this initialization's stability across varying dataset sizes. This efficiency demonstrates our capacity for effectively managing large datasets for optimized query execution.

Table 2: Datasets Characteristics: $|E|$:# records, $|L_E|$:# duplicates, $|A|$: # attributes, $|TBI|$: # blocks in TBI

E	$\|E\|$	$\|\mathbf{L_E}\|$	$\|A\|$	$\|TBI\|$
DSD	66879	5347	4	88K
OAGV	130K	29841	5	55K
OAGP200K-2M	200K-2M	5K-300K	18	110K-360K
OAO	55464	5464	3	22K
OAP	500K	58074	8	170K
PPL200K-2M	200K-2M	64K-645K	12	160K-850K

8.2 SP Class: Performance of QueryER

In this experiment, we assess QueryER's performance and scalability in terms of *execution time* (TT) needed to return query results. We employ *QDA*[3] and *BrewER*[42] as baselines, which

only support SP queries. We apply QueryER's *blocking* and *meta-blocking* configuration to all experiments, as both baselines lack built-in block-producing mechanisms. Providing post-meta-blocking block collections to *QDA* and *BrewER*, ensures a fair comparison under identical conditions and maintains the maximum *PC* for each query across all approaches. For *BrewER*, progressive recall is obtained during execution after the emission of each entity.

Evaluation Workload. We evaluate SP *conjunctive* and *disjunctive* queries with two selection predicates for each dataset, creating two batches (*conjunctive* / *disjunctive*) of 20 queries each and reporting averages. Each batch consists of 10 high-selectivity and 10 low-selectivity queries out of 50 randomly generated ones. This setup covers a broad range of selectivity and explores performance under various data distribution scenarios. We utilized this setup to ensure fairness in the competition with BrewER and QDA both of which were evaluated in a similar scenario. The generator used for our query evaluation is provided in our Github repo. We used real *DSD*, *OAGP*200*K* and synthetic datasets *PPL*200*K*. Larger datasets were excluded due to scalability limitations of the baseline approaches[8].

Results: Fig. 6 illustrates the performance of the three approaches in terms of maximum (progressive for *BrewER*) *PC* against *TT*. QueryER and *QDA* display a step curve, as they perform all pairwise comparisons before emitting the results. QueryER consistently demonstrates higher performance ($\approx 100X$ faster than its competitors). Since we used the same blocking and meta-blocking configuration for all approaches, the performance difference between QueryER and the other approaches, *QDA* and *BrewER*, can be attributed to their transitive closure handling. *QDA* and *BrewER* use iterative transitive closure computation and graph traversal in merged blocks, to perform comparisons and assess duplicates, a process whose complexity grows cubically with block size, $O(|b|^3)$, impeding scalability. This laborious process aims to fully resolve each entity's duplicates before proceeding. Both approaches perform the same number of comparisons if given enough time. In contrast, QueryER leverages the transitive nature of matches during comparison execution to avoid unnecessary pairwise comparisons. It uses the union-find[5] to form disjoint-sets of matching entities and checks if the entities in a comparison have already matched with a common entity, thus reducing computational overhead.

Table 3: Pre-processing time percentiles based on all datasets.

	10%	25%	50%	75%	90%	std.
Index Creation (s)	2.88	10.16	31.01	83.32	113.91	44.8
Relational Stats Calc. (s)	0.15	0.69	1.07	1.55	1.8	0.65
E.R Stats Calc. (s)	1.53	6.99	27.99	76.86	99.05	40.42

Table 4: Execution time issuing Q_{80} (high selectivity) on *DSD* and *OAP*.

E	TT(s)	Deduplicate Operator			Group-Entities	Other
		Block-Join	Meta-Blocking	Comp.-Exec.		
DSD	6.2	7%	5%	82%	3%	3%
OAP	422.5	5%	7%	83%	1%	4%

[8]For instance, it took over 9000 sec. for *BrewER* and *QDA* to evaluate queries on *OAGP*500*K* dataset.

Table 5: *M-B* configurations on *PPL*1*M* and *OAGP*1*M*, to examine both low (Q_5) and high (Q_{80}) selectivity scenarios.

Query	Method	Time	PC	PQ	F1
Q_5	ALL	88.15 / 120.14	0.99 / 0.98	0.01 / 0.01	0.02 / 0.02
Q_5	BP + BF	429.2 / 457.32	0.99 / 0.99	0.001 / 0.001	0.002 / 0.002
Q_5	BP + EP	> 30 MIN	> 30 MIN	N/A	N/A
Q_{80}	ALL	305.12 / 352.51	0.99 / 0.98	0.06 / 0.034	0.11 / 0.07
Q_{80}	BP + BF	980.72 / 802.12	0.99 / 0.99	0.01 / 0.009	0.02 / 0.02
Q_{80}	BP + EP	> 30 MIN	> 30 MIN	N/A	N/A

8.3 SP Class: Scalability of QueryER

In these experiments, we assess the scalability of QueryER and the impact of *LI* and different *meta-blocking* configurations. We use SP queries on real (*OAGP*) and synthetic (*PPL*) datasets with increasing $|E|$ (500K-2M). We also analyze *Deduplicate* and *Group-Entities* operators' execution time on real datasets (*DSD* & *OAP*).

Scalability: We evaluate QueryER's scalability over increasing dataset sizes with fixed selectivity on both real ($OAGP200K - 2M$) and synthetic ($PPL200K - 2M$) datasets. We employ Q_{MOD}: *MOD(id, 10) <1* in the *WHERE* clause for both datasets. The MOD(id,n) <1 condition, a SQL function that calculates the remainder of the integer attribute 'id' divided by n, ensures a random set of entities with a fixed selectivity, irrespective of the increasing number of entities in the dataset. Fig. 7 shows the *TT* and the executed comparisons over an increasing dataset size with fixed selectivity. This comprehensive experiment showcases QueryER's scalability as the dataset size increases. Both metrics indicate that QueryER's scaling is sub-linear as $|E|$ increases, keeping the number of comparisons in the same order of magnitude across the dataset size range, without a proportional increase when $|E|$ doubles.

Meta-Blocking Configurations: We examined *Meta-Blocking* configurations on real (*OAGP*1*M*) and synthetic (*PPL*1*M*) datasets using Q_{MOD} queries Q_5 and Q_{80} for selectivities of $\approx$ 5% and $\approx$ 80% respectively. Table 5 contrasts three configurations: (i) *ALL* (all methods), (ii) *BP+BF* (*Block Purging* and *Block Filtering*), and (iii) *BP+EP* (*Block Purging* and *Edge Pruning*). *ALL* yields high recall (PC) and the best precision (PQ) and F1 scores, especially in high selectivity cases. The *ALL* method's balanced precision-recall trade-off, combined with its faster execution times, validate its effectiveness for QueryER's use, outweighing configurations with longer runtimes and lower precision and F1 scores. Other combinations took over an hour to finish and were dismissed.

Impact of LI: We assess the impact of the index *LI* on QueryER scalability by executing four overlapping range-queries ($QR1 - QR4$) on the *OAGP2M* dataset. Each query contains the E_σ of the previous plus 30% more entities, starting with *QR*1 which has $|E_\sigma|$ = 760K. For comparison purposes, we used the *BQ* including the offline deduplication time. Fig. 8 depicts how QueryER's performance is affected by using the *LI*. As shown, the *TT* of each approach diverges from the others with each issued query, exhibiting opposite trends. The *TT* of the "Without *LI*" approach increases sub-linearly, approaching the *TT* of the *BQ*. In contrast, by utilizing *LI* and incrementally cleaning the *E*, the *TT* of the "With *LI*" approach decreases similarly and approaches 0.

Time breakdown: We analyze the execution time of the *Deduplicate* and *Group-Entities* operators on a real (*DSD*) and synthetic (*OAP*) dataset using the high-selectivity SP query (Q_{80}). Table 4 presents the results. The total time comprises *Deduplicate* operator's pipeline: (i) *Block-Join*, (ii) *Meta-Blocking*, (iii) *Comparison-Execution*, as well as (iv) *Group-Entities* operator and (v) other operations (e.g. Table-Scan). The results indicate that, *Comparison-Execution* dominates the total time on large datasets with high

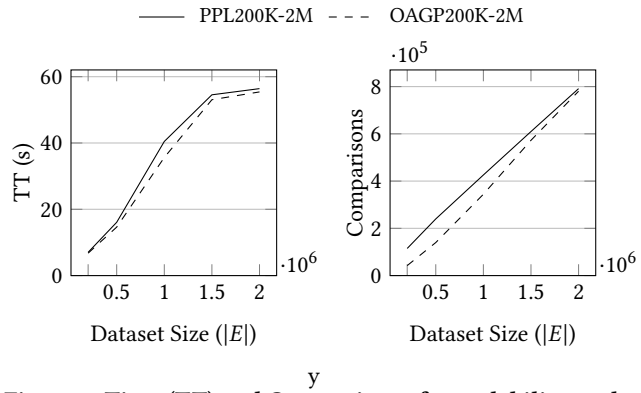

Figure 7: Time (TT) and Comparisons for scalability evaluation with increasing dataset size and fixed selectivity (Q_{MOD}).

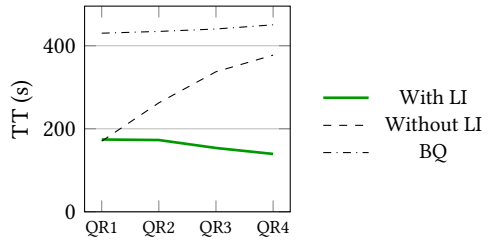

Figure 8: Time (TT) for consecutive queries (QR1-QR4) with and without utilizing *LI* on *OAGP2M* dataset.

selectivity, as expected, due to the high-cost distance functions. Based on this, we can conclude that *Comparison-Execution* is the dominant factor in the overall approach.

8.4 SPJ Class: Cost-based planner Evaluation

In this section, we assess the performance and scalability of our *cost-based planner* against two naive fixed-query-plan solutions: (**BQ**), cleaning the entire table before filtering (like Fig.4a) and (**NES**), placing the *Deduplicate Operator* above the filter on one query tree branch (like Fig.4b). Our proposed *Advanced ER Solution* (**AES**) minimizes comparison execution costs by selecting the plan with the fewest comparisons, deemed optimal. A thorough exposition of the process is provided in Sec. 6.

Evaluation Workload. We have designed a series of SPJ queries on both real and synthetic datasets. All queries involved joins between two tables, with fixed selectivity (100%) on one side, and varying (low-high) selectivity on the other, with averages reported. For performance evaluation, we synthesized queries as described in Sec. 8.2, using the dataset combinations: (i) PPL2M ⋈ OAO, (ii) OAGP2M ⋈ OAGV, (iii) OAP ⋈ OAO, (iv) OAGP2M ⋈ OAGV. With query sets (i) and (ii) yielding the lowest selectivity, while (iii) and (iv) the highest. For scalability assessment, we maintained fixed selectivity (≈15%) while ensuring random selection on one side and kept the other side at (100%), across increasing dataset sizes $|E|$. For this assessment, we used the following dataset combinations: (v) PPL200K-2M ⋈ OAO, and (vi) OAGP200K-2M ⋈ OAGV.

Performance: Fig. 9, displays the TT, indicating also the planning time for *AES*, (top) and executed comparisons (bottom) for *AES*, *NES* and *BQ* across dataset combinations. As anticipated, *AES* outperforms both *NES* and *BQ* in terms of TT and executed comparisons. This is attributed to the use of query's statistics and the best placement of ER operators (Sec.6.2.1), which results in a reduction of the overall number of executed comparisons. The reduction of the difference exhibited by *NES* and *BQ* in Fig. 9 (c) and (d), where the highest selectivity occurs, can be attributed to the large number of selected entities in the query, similar to what was observed in Fig. 8, where the "Without *LI*" approach increases sub-linearly, approaching the TT of *BQ*. In contrast, the effectiveness of *AES*, which primarily focuses on cleaning the table with the fewest comparisons first, is evident in its distinct performance from *NES* and *BQ* in such queries. When table sizes are similar, the performance of *AES* is primarily influenced by the *join-percentage*, rather than dataset size. Additionally, the planning phase constitutes only 1.80% to 7.41% of *AES*'s total execution time (TT), highlighting its efficiency.

An interesting observation in Fig. 9 (b) and (d) is that the TT of *AES*, unlike the one of *NES*, in certain cases does not strongly correlate with the executed comparisons. This can occur when the join-percentage between tables is small, resulting in a small E'_σ for the left joining table, formed from entities joining with the right table's E_G (see Sec. 5.2). In such cases, the small join-percentage (≈5%) results in a small E'_σ for *OAGP2M* formed from entities joining with *OAGV*'s E_G. This matters because the number of executed comparisons is related to the size of the E_σ, and in this instance, the TT is dominated by the blocking/meta-blocking operations. Specifically, the time breakdown for the different deduplication operations is as follows: Blocking 10%, Block-join 3%, Planning 7%, Block Purging 0.5%, Block Filtering 0.5%, Edge Pruning 75%, Comparison-Execution 4%. Contradicting Table 4, these findings emphasize the importance of cleaning the table with fewest comparisons first for optimal performance.

Scalability: Fig. 10 displays the TT and the executed comparisons for *AES* and *NES* for PPL200K-2M ⋈ OAO (top), and OAGP200K-2M ⋈ OAGV (bottom) dataset combinations. Both solutions exhibit sub-linear scaling with increasing dataset size, with *AES* outperforming *NES*. This observation can be more readily discerned in the comparisons plots, where the order of magnitude for the comparisons remains consistent despite doubling the dataset size. Interestingly, in this case, not only does the dataset size increase, but so does the original number of entities ($|E_\sigma|$) for each dataset. As the number of comparisons significantly depends on $|E_\sigma|$, this could affect performance, but the scaling remains sub-linear despite this.

Statistics. Table 6 evaluates our cost-based planner's Q-errors, revealing deviations between predicted and actual values. The planner effectively predicts duplication factor (df) and dynamically calculates *Comparisons* and E_G size statistics for each query (see Sec. 6.2.1), as shown by the low values and the minimal variance in Q-error percentiles. This consistent accuracy, demonstrated by low standard deviations, underscores *AES*'s reliability and enhances execution efficiency, reinforcing its analytical superiority over *NES* across varying dataset complexities.

Table 6: Computed Statistics Estimation Q-Error percentiles based on all datasets used in SPJ Class.

	10%	25%	50%	75%	90%	std.
Duplicate Factor	1.03	1.07	1.15	1.39	1.46	0.19
Comparisons	1.23	1.27	1.52	7.87	7.88	3.21
E_G **size**	1.45	1.48	1.67	2.25	3.75	1.25

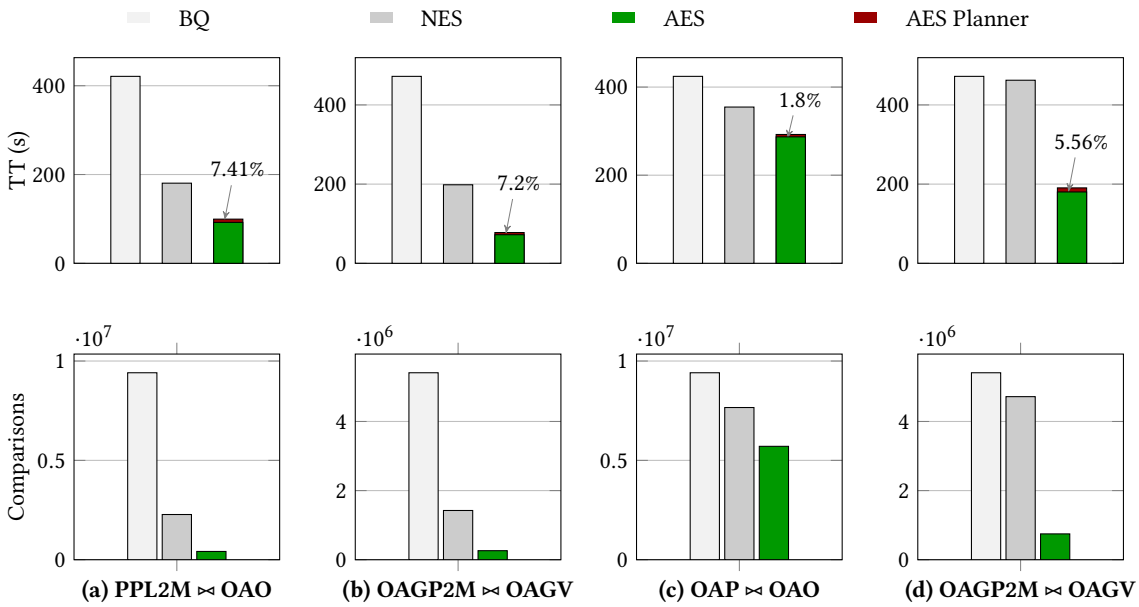

Figure 9: BQ vs NES vs AES on TT (top) and Comparisons (bottom). (a), (b) lowest selectivity on PPL2M⋈OAO and OAGP2M ⋈ OAGV respectively. (c), (d) highest selectivity on OAP ⋈ OAO and OAGP2M ⋈ OAGV, respectively.

8.5 Summary

Our extensive experimental study underscores QueryER's robust performance. Notably, its efficacy escalates inversely with selectivity ($|E_\sigma|$), consistently surpassing the baseline. In the SPJ class, QueryER shines in joining tables with low join-percentage and pronounced table size disparities - a scenario where a *Batch Approach* would falter. This reinforces QueryER's aptitude for data exploration and analysis. For smaller $|E_\sigma|$ instances, Meta-Blocking, particularly *EP*, primarily influences total time (*TT*), despite seeming discrepancies with Table 4 results; *ALL* configuration remains the most efficient upon closer examination of Table 5. QueryER consistently outperforms the baseline ($\approx 100X$ faster) across diverse real and synthetic datasets, upholding PC levels with a minimum of 97% and a mean of 98%. The scalability and consistently high recall levels, demonstrated across a diverse range of datasets with varying characteristics, attest to its robustness and adaptability. Lastly, *LI* further enhances the performance, enabling incremental deduplication beneficial for data exploration and analysis scenarios involving consecutive, often overlapping, queries.

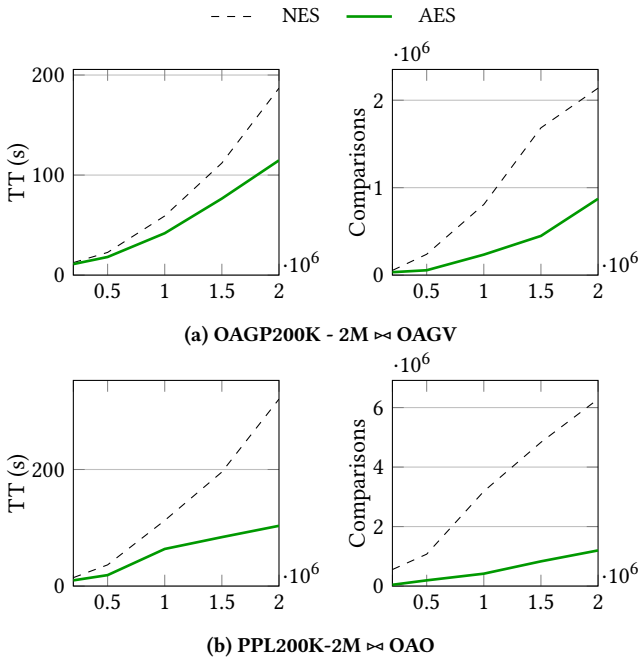

Figure 10: Time (TT) and Comparisons vs. increasing dataset size ($|E|$) for scalability evaluation of NES and AES.

9 CONCLUSIONS AND FUTURE WORK

In this work, we investigated the problem of integrating Entity Resolution into Query Processing. We developed QueryER, an innovative SQL engine that enables efficient, analysis-aware ER over dirty data, with minimal pre-processing and no manual preparation overhead. Our experimental evaluation demonstrates its sub-linear scalability and consistently high recall performance, outperforming baseline approaches. This study has revealed several new research paths. Extending QueryER to tackle other query classes, such as aggregation and analytical queries, presents a valuable direction for exploration. Another promising direction involves evaluating performance and recall with ML-based blocking and entity matching methods. Given QueryER modular design and agnosticism to specific methods, these techniques could potentially augment its matching effectiveness further. Finally, we plan to scale out our implementation to a distributed and parallel environment, further enhancing the performance and applicability of QueryER.

Acknowledgements. This work was partially supported by ExtremeXP (EU Horizon program, GA: 101093164) and EOSC Beyond projects (GA: 101131875).

REFERENCES

[1] G. Alexiou and G. Papastefanatos. 2019. Query Driven Entity Resolution in Data Lakes. In *Springer, Cham*, Vol. International Workshop on Information Search, Integration, and Personalization. 117–130.

[2] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. 2013. Query-driven approach to entity resolution. In *Proceedings of the VLDB Endowment*. 1846–1857.

[3] H. Altwaijry, D. V. Kalashnikov, and S. Mehrotra. 2017. QDA: A Query-driven Approach to Entity Resolution. In *IEEE Trans. Knowl. Data Eng.* 402–417.

[4] H. Altwaijry, S. Mehrotra, , and D. V. Kalashnikov. 2015. Query: A framework for integrating entity resolution with query processing. In *Proceedings of the VLDB Endowment*, Vol. 9(3). 120–131.

[5] R. J. Anderson and H. Woll. 1991. Wait-free parallel algorithms for the union-find problem. In *In Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 370–380.

[6] R. Baxter, P. Christen, and T. Churches. 2003. A comparison of fast blocking methods for record linkage. In *Workshop on Data Cleaning, Record Linkage and Object Consolidation*. 25–27.

[7] I. Bhattacharya, , and L Getoor. 2007. Query-time entity resolution. booktitle of Artificial Intelligence Research. 621–657.

[8] P. Christen. 2012. Data Matching. Data-centric systems and applications. In *Springer*.

[9] P. Christen. 2012. A survey of indexing techniques for scalable record linkage and deduplication. In *IEEE Trans. Knowl. Data Eng.* 1537– 1555.

[10] P. Christen and T. Churches. 2002. Febrl-Freely extensible biomedical record linkage.

[11] X. Dong, E. Gabrilovich, G. Heitz, W. Horn, K. Murphy, S. Sun, and W. Zhang. 2014. From Data Fusion to Knowledge Fusion. In *Proc. VLDB Endow.*, Vol. 7. 881–892.

[12] M. Ebraheem, S. Thirumuruganathan, S.R. Joty, M. Ouzzani, and N. Tang. 2018. Distributed Representations of Tuples for Entity Resolution., In Proc. VLDB Endow. 11, 1454–1467.

[13] V. Efthymiou, , K. Stefanidis, , and V. Christophides. 2016. Minoan ER: progressive entity resolution in the web of data.. In *EDBT*. 221–232.

[14] D. Firmani, B. Saha, and D. Srivastava. 2016. Online Entity Resolution using an Oracle.. In *Proc. VLDB Endow.*, Vol. 9. 384–395.

[15] S. Galhotra, D. Firmani, B. Saha, and D. Srivastava. 2021. Efficient and Effective ER with Progressive Blocking.. In *VLDB J.*, Vol. 30. 537–557.

[16] L. Getoor, , and A. Machanavajjhala. 2012. Entity resolution: Theory, practice and open challenges. In *Proceedings of the VLDB Endowment*. 2018–2019.

[17] S. Giannakopoulou, M. Karpathiotakis, and A. Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 20)*. 805–815. https://doi.org/10.1145/3318464.3389775

[18] S. Giannakopoulou, M. Karpathiotakis, B.C.D. Gaidioz, and A. Ailamaki. 2017. An overview of microsoft academic service (mas) and applications. In *Proceedings of the VLDB Endowment,10(CONF)*.

[19] E. Ioannou, , and M. Garofalakis. 2015. Query analytics over probabilistic databases with unmerged duplicates. In *IEEE Trans. Knowl. Data Eng.*, Vol. 27(8). 2245–2260.

[20] E. Ioannou, W. Nejdl, C. Niederée, and G. Velegrakis. 2010. On-the-fly entity-aware query processing in the presence of linkage.. In *Proceedings of the VLDB Endowment*, Vol. 3(12). 429–438.

[21] P. Ipeirotis, Verykios V. S., and A. K. Elmagarmid. 2007. Duplicate record detection: A survey. In *IEEE Trans. Knowl. Data Eng.* 1–16.

[22] M. A. Jaro. 1989. Advances in record-linkage methodology as applied to matching the 1985 census of Tampa, Florida, In Journal of the American Statistical Association. 84(406), 414–420.

[23] D. Karapiperis, A. Gkoulalas-Divanis, and V.S. Verykios. 2021. MultiBlock: A Scalable Iterative Approach for Progressive Entity Resolution. In *In 2021 IEEE International Conference on Big Data (Big Data)*. 219–228.

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

[25] M. Lenzerini. 2002. Data integration: A theoretical perspective.. In *IEEE Trans. Knowl. Data Eng.* 233–236.

[26] Y. Li, J. Li, Y. Suhara, A. Doan, and W. C. Tan. 2020. Deep Entity Matching with Pre-trained Language Models. In *Proc. VLDB Endow*, Vol. 14. 50–60.

[27] J. Madhavan, S. Cohen, X. L. Dong, A. Y. Halevy, S. R. Jeffery, D. Ko, and C. Yu. 2007. Web-scale Data Integration: You Can Afford to Pay as You Go.. In *CIDR.*, Vol. 30. 342–350.

[28] Paolo Manghi et al. 2019. *OpenAIRE Research Graph Dump.* https://doi.org/10.5281/zenodo.3516918

[29] G. Moerkotte, T. Neumann, and G. Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors, In Proceedings of the VLDB Endowment. 2(1), 982–993.

[30] S. Mudgal, H. Li, T. Rekatsinas, A. Doan, Y. Park, G. Krishnan, R. Deep, E. Arcaute, and V. Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration.. In *In SIGMOD Conference. ACM*. 19–34.

[31] G. Papadakis, G. Alexiou, G. Papastefanatos, , and G. Koutrika. 2015. Schema-agnostic vs schema-based configurations for blocking methods on homogeneous data. In *Proceedings of the VLDB Endowment*. 312–323.

[32] G. Papadakis, V. Efthymiou, E. Thanos, O. Hassanzadeh, and P. Christen. 2023. An analysis of one-to-one matching algorithms for entity resolution. In *The VLDB Journal*. 1–32.

[33] G. Papadakis, E. Ioannou, T. Palpanas, C. Niederee, and W. Nejdl. 2013. A blocking framework for entity resolution in highly heterogeneous information space. In *IEEE Trans. Knowl. Data Eng.* 2665– 268.

[34] G. Papadakis, E. Ioannou, E. Thanos, and T. Palpanas. 2021. The Four Generations of Entity Resolution. Synthesis Lectures on Data Management,. In *16(2)*. 1–170.

[35] G. Papadakis, G. Koutrika, T. Palpanas, , and W. Nejdl. 2013. Meta-blocking: Taking entity resolution to the next level. In *IEEE Trans. Knowl. Data Eng.* 1946–1960.

[36] G. Papadakis, G. Mandilaras, L. Gagliardelli, G. Simonini, E. Thanos, G. Giannakopoulos, S. Bergamaschi, T. Palpanas, and M. Koubarakis. 2020. Three-dimensional entity resolution with JedAI.. In *Information Systems*. 101565.

[37] G. Papadakis, G. Papastefanatos, T. Palpanas, and M. Koubarakis. 2016. Scaling entity resolution to large, heterogeneous data with enhanced meta-blocking.. In *EDBT*. 221–232.

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

[39] A. Periklis, A. Fuxman, and R.J. Miller. 2006. Clean answers over dirty databases: A probabilistic approach.. In *In 22nd International Conference on Data Engineering*. 30–30.

[40] A. D. Sarma, X. Dong, and A. Y. Halevy. 2008. Bootstrapping Pay-as- you-go Data Integration Systems.. In *In SIGMOD Conference. ACM*. 861–874.

[41] G. Simonini, G. Papadakis, T. Palpanas, and S. Bergamaschi. 2019. Schema-agnostic progressive entity resolution.. In *IEEE Trans. Knowl. Data Eng.* 1208–1221.

[42] G. Simonini, L. Zecchini, S. Bergamaschi, and F. Naumann. 2022. Entity resolution on-demand.. In *Proceedings of the VLDB Endowment*. 1506–1518.

[43] A. Sinha, Z. Shen, Y. Song, H. Ma, D. Eide, B. J. Hsu, and K Wang. 2015. Cleanm: An optimizable query language for unified scale-out data cleaning. In *Proceedings of the 24th international conference on world wide web*. 243–246.

[44] Y. Sismanis, L. Wang, A. Fuxman, P. J. Haas, and B. Reinwald. 2009. Resolution-aware query answering for business intelligence. In *In 25th International Conference on Data Engineering*. 976–987.

[45] J. Tang, J. Zhang, L. Yao, J. Li, L. Zhang, and Z. Su. 2008. Arnetminer: extraction and mining of academic social networks. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*. 990–998.

[46] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. 2014. A sample-and-clean framework for fast and accurate query processing on dirty data.. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 469–480.

[47] S. E. Whang, D. Marmaros, and H. Garcia-Molina. 2013. Pay-as-you-go Entity Resolution.. In *IEEE Trans. Knowl. Data Eng.*, Vol. 25. 1111– 1124.