Rule-Based Graph Cleaning with GPUs on a Single Machine

WENCHAO BAI, Southeast University, China
WENFEI FAN, Shenzhen Institute of Computing Sciences, China, University of Edinburgh, United Kingdom, and Beihang University, China
SHUHAO LIU, Shenzhen Institute of Computing Sciences, China
KEHAN PANG, Beihang University, China
XIAOKE ZHU, Beihang University, China
JIAHUI JIN*, Southeast University, China

This paper studies cost-effective graph cleaning with a single machine. We adopt a rule-based method that may embed machine learning models as predicates in the rules. Graph cleaning with the rules involves rule discovery, error detection and correction. These tasks are both computation-heavy and I/O-intensive as they repeatedly invoke costly graph pattern matching, and produce a large volume of intermediate results, among other things. In light of these, no existing single-machine system is able to carry out these tasks even on not-too-large graphs, even using GPUs. Thus we develop MiniClean, a single-machine system for cleaning large graphs. It proposes (1) a workflow that better fits a single machine by pipelining CPU, GPU and I/O operations; (2) memory footprint reduction with bundled processing and data compression; and (3) a multi-mode parallel model for SIMD, pipelined and independent parallelism, and their scheduling to maximize CPU–GPU synergy. Using real-life graphs, we empirically verify that MiniClean outperforms the SOTA single-machine systems by at least 65.34× and multi-machine systems with 32 nodes by at least 8.09×.

CCS Concepts: • **Information systems → Data cleaning**; **Graph-based database models**.

Additional Key Words and Phrases: Graph cleaning rules, single machine systems, GPU

ACM Reference Format:
Wenchao Bai, Wenfei Fan, Shuhao Liu, Kehan Pang, Xiaoke Zhu, and Jiahui Jin. 2025. Rule-Based Graph Cleaning with GPUs on a Single Machine. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 166 (June 2025), 31 pages. https://doi.org/10.1145/3725303

1 Introduction

A variety of single-machine systems have been developed for graph analytics, for in-memory tasks when graphs can be loaded entirely into main memory [41, 56, 65, 76, 90, 92, 94], or out-of-core processing of graphs that are too large to fit into the main memory of a machine at once [9, 50, 51, 54, 57, 72, 83, 93, 95]. These systems efficiently support common graph analytic queries such as connected components (CC), PageRank (PR), single-source shortest path (SSSP),

*Corresponding author

Authors' Contact Information: Wenchao Bai, wbai@seu.edu.cn, Southeast University, China; Wenfei Fan, wenfei@inf.ed.ac.uk, Shenzhen Institute of Computing Sciences, China and University of Edinburgh, United Kingdom and Beihang University, China; Shuhao Liu, shuhao@sics.ac.cn, Shenzhen Institute of Computing Sciences, China; Kehan Pang, pangkehan@buaa.edu.cn, Beihang University, China; Xiaoke Zhu, zhuxk@buaa.edu.cn, Beihang University, China; Jiahui Jin, jjin@seu.edu.cn, Southeast University, China.

ACM 2836-6573/2025/6-ART166
https://doi.org/10.1145/3725303

minimum spanning tree (MST) and random walk (RW).

How would these systems behave when it comes to computation problems that are more complex and are widely practiced?

Our industry collaborators brought up such a problem to us. They aim to clean graphs using a single machine. Given a graph G, it is to detect and correct errors in G, such as duplicates (*e.g.,* different vertices denoting the same person) and conflicts (contradictory facts such as a stadium that was torn down before it was built). They have already deployed a single-machine system for graph association analysis in battery manufacturing production lines [29], and they want to enrich the system with the functionality of graph cleaning. The need for single-machine graph cleaning is also evident in industrial Internet-of-Things and edge computing (for cleaning sensor data at collection [15, 84]) because of: (a) edge locations' physical constraints (*e.g.,* limited power and space) that make cluster maintenance impractical, (b) prohibitive costs of cloud alternatives (network bandwidth at $0.05–0.09 per GB [1]), and (c) industrial data privacy requirements that favor local processing. These scenarios benefit from single-machine systems through reduced operational costs, simplified deployment, and enhanced data security.

Our collaborators adopt a rule-based method utilizing Graph Cleaning Rules (GCRs) [22]. A GCR has the form of $Q[x_0, y_0](X \rightarrow p_0)$, where Q is a graph pattern that identifies relevant entities, and $X \rightarrow p_0$ is a dependency that discloses the correlations, interactions and associations of these entities (see Section 2). GCRs may embed machine learning (ML) models as predicates, and unify ML prediction and logic reasoning to catch and fix errors. Cleaning a graph G with GCRs involves discovering GCRs from (samples of) G, and detecting and correcting errors in G with the mined rules.

Unfortunately, the practitioners find that none of the existing single-machine graph systems can carry out the three tasks, even on not-very-large graphs and with GPUs. Below are a few reasons.

(1) Computation-heavy. Rule discovery, error detection and correction need to repeatedly enumerate matches of graph pattern Q, a task that is far more costly than CC, PR, SSSP, MST and RW. Even when GPUs are available, their memory capacity is often insufficient for large graphs G. Moreover, to make effective use of the computing resources and reduce their idling, nontrivial scheduling is a must for pipelining and balancing I/O, CPU/GPU operations, and data transfers between CPU and GPU memories.

(2) Excessive intermediate results. Rule validation may generate a large amount of intermediate results, *e.g.,* pattern matches of various GCRs. Simply swapping the intermediate results between memory and secondary storage incurs excessive I/O; it may even crash some systems that require intermediate results to fit in memory [50, 54, 93]. Thus, the I/O strategies of existing systems no longer suffice. Even with high-throughput NVMe SSDs, I/O remains a bottleneck due to repetitive loading and evictions of intermediate results. One needs a more sophisticated strategy for memory management.

(3) Parallel model. The existing graph systems typically adopt a fixed parallel model, *e.g., vertex-centric* (VC) [59, 65, 76], *edge-centric* (EC) [57, 72, 95], or a hybrid [96] of VC and graph-centric (GC) [35]. However, as will be seen in Section 6, for the three graph cleaning tasks, a single parallel model is no longer able to cope with CPU–GPU collaboration. CPUs, though with limited parallelism, can handle flexible control flows with diverse data dependencies in irregular graph structures. In contrast, GPUs, with massive Single Instruction, Multiple Data (SIMD) parallelism, require parallel algorithms based on regular data structures and control flows. These call for a multi-mode parallel model to maximize resource utilization.

MiniClean. In response to the practical need, this paper develops MiniClean, a single-machine system for cost-effective graph cleaning. It can scale to large graphs with billions of vertices and edges, outperforming multi-machine systems with 32 nodes by 8.09–52.20×. It is unique in the following.

(1) Parallel graph cleaning (Section 3). MiniClean supports full-fledged graph cleaning with GCRs, *i.e.,* rule discovery, error detection and correction. It proposes a two-stage workflow for match enumeration in a pipeline of I/O, CPU and GPU operations, which better fits the shared-memory architecture of a single machine. To handle large intermediate results that exceed memory capacity, it treats NVMe SSDs as memory extensions and main memory as cache.

(2) Host-side optimizations (Section 4). To reduce repetitive computations, intermediate results, and data transfer between CPUs and GPUs, MiniClean develops and deploys a combination of strategies, including (a) a bundling strategy that groups similar pattern matching tasks, (b) a compression technique for intermediate data based on conditional succinct tables, and (c) an adaptive strategy for recursive bundling to balance CPU and GPU workloads.

(3) A multi-mode parallel model (Section 5). MiniClean integrates (a) pipelined parallelism, which pipelines two stages of the match enumeration workflow on CPUs and GPUs, using adaptive bundling to balance their workloads; (b) SIMD parallelism, in which GPU threads execute the same operation on different data elements concurrently within an enumeration task; and (c) independent parallelism, which carries out separate enumeration tasks on a GPU simultaneously. To fully utilize the computing resources, it proposes a strategy to schedule task execution, minimizing data I/O and transfers. We show that it is NP-complete to find an optimal schedule; nonetheless, we develop an effective heuristic solution.

(4) Performance (Section 6). Using real-life graphs, we experimentally find the following. (1) MiniClean is able to clean billion-scale graphs within 4.67 h. In contrast, the state-of-the-art (SOTA) single-machine systems cannot conduct rule discovery and error correction even in small graphs, and cannot even do simpler error detection in large graphs within 8 h. For simpler error detection in small graphs, MiniClean is faster by 65.34× than the SOTA hybrid MiniGraph, while both in-memory and out-of-core system still run out-of-memory. (2) Compared to SOTA multi-machine systems, MiniClean outperforms 32-node clusters by 9.22×, 20.42× and 8.09× in rule discovery, error detection and error correction, respectively. (3) It is a combination of strategies that make MiniClean capable of cleaning large graphs, such as tasking bundling, succinct tables, multi-mode parallelism and scheduling, which speed up the performance by up to 2.47×, 13.06×, 1.77× and 1.45×, respectively.

Organization. The rest of the paper is organized as follows.
- A review of GCRs and GCR-based graph cleaning (Section 2).
- The parallel workflow and architecture of MiniClean (Section 3).
- Optimization techniques for host-side computation (Section 4).
- The multi-mode parallel model and its schedule (Section 5).
- An experimental study of effectiveness and efficiency (Section 6),
- Related work (Section 7) and topics for future work (Section 8).

2 Graph Cleaning with GCRs

This section reviews notations (Section 2.1), GCRs (Section 2.2) and multi-machine parallel algorithms for graph cleaning (Section 2.3).

2.1 Preliminaries

We start with basic notations. Assume two countably infinite alphabets, denoted by Λ and Υ, for labels and attributes, respectively.

Graphs. A directed labeled graphs is $G = (V, E, L, F_A)$, where (a) V is a finite set of vertices; (b)

Table 1. Notations and glossaries.

Notation / Term	Description
$Q[x_0, \bar{x}] = (V_Q, E_Q, L_Q, \mu)$	A star pattern, whose center is x_0.
$\mathcal{Q}[x_0, y_0] = \langle Q_x[x_0, \bar{x}], Q_y[y_0, \bar{y}] \rangle$	A dual star pattern, with two centers at x_0 and y_0.
$\varphi = \mathcal{Q}[x_0, y_0](X \to p_0)$	A graph cleaning rule (GCR).
Scattered match	A data structure used for efficient matching enumeration of star patterns (§ 2.3).
Mono-star component $\omega = Q[x_0, \bar{x}](X, F_Q)$	Decomposition of a GCR, used for star matching in Stage 1 of the workflow (§ 3).
Mono-star bundle $\psi = \langle \omega_*, \omega_+ \rangle$	Composition of mono-star components/bundles, used for bundled processing (§ 4.1).
Star candidate	A match of a mono-star bundle (§ 3).

$E \subseteq V \times \Lambda \times V$ is a finite set of edges, in which $e = (v, l, v')$ is an edge labeled $l \in \Lambda$ from vertex v to v'; (c) each vertex $v \in V$ has label $L(v)$ from Λ, and carries a tuple $F_A(v) = (A_1, \ldots, A_n)$ of *attributes* of a finite arity, where $A_i \in \Upsilon$ and $A_i \neq A_j$ if $i \neq j$. Vertices may carry different attributes, which is not constrained by a schema like relational databases.

We assume the existence of a special attribute id at each vertex v, denoting its vertex identity, such that for any two vertices v and v' in G, if v.id = v'.id, then v and v' refer to the same entity.

Paths. A *path ρ from a vertex* v_0 in G is a list $\rho = (v_0, l_0, v_1, \ldots, v_n)$ such that (v_{i-1}, l_{i-1}, v_i) is an edge in G ($i \in [1, n]$). We consider *simple* paths on which each vertex appears at most once. A vertex v is called a *child* of u if there exists an edge (u, l, v) in E.

Star patterns. A *star pattern* is $Q[x_0, \bar{x}] = (V_Q, E_Q, L_Q, \mu)$, where (1) V_Q (resp. E_Q) is a set of pattern vertices (resp. edges) as defined above; (2) L_Q assigns a label of Λ to each vertex in V_Q; (3) $\bar{x}$ is a list of distinct variables, and μ is a bijective mapping from $\bar{x}$ to the vertices of Q; (4) x_0 is a designated variable in $\bar{x}$, referred to as the *center* of Q; and (5) for each $z \in \bar{x}$, there exists a single path from x_0 to z, z has at most one child, except x_0. For variables $z \in \bar{x}$, we use $\mu(z)$ and z interchangeably if it is clear in the context.

Intuitively, $Q[x_0, \bar{x}]$ has the shape of a star with center x_0. The center x_0 denotes an entity of interest, and it links to a set of characteristic features without children (*leaf* vertices), each via a path.

Dual patterns. A *dual pattern* is $\mathcal{Q}[x_0, y_0] = \langle Q_x[x_0, \bar{x}], Q_y[y_0, \bar{y}] \rangle$, where $Q_x[x_0, \bar{x}]$ and $Q_y[y_0, \bar{y}]$ are disjoint star patterns. *i.e.,* Q_x and Q_y have no common vertices. Intuitively, $\mathcal{Q}$ is a pair of patterns to represent two entities x_0 and y_0 with (possibly) heterogeneous structures in a schemaless graph. The star patterns specify features (property vertices) for pairwise comparison between x_0 and y_0.

Homomorphic mapping. A *homomorphic mapping h* from graph $G = (V_G, E_G, L_G, F_A^G)$ to graph $H = (V_H, E_H, L_H, F_A^H)$ is a function $h : V_G \to V_H$ such that (a) for each vertex $u \in V_G$, $L_G(u) = L_H(h(u))$, and (b) for each edge (u, l, u') in E_G, $(h(u), l, h(u'))$ is an edge in E_H.

Pattern matches. A *match of a star pattern Q in a graph G* is a homomorphic mapping h from the pattern vertices in Q to G.

Similarly, a match of a dual pattern $\mathcal{Q}[x_0, y_0] = \langle Q_x, Q_y \rangle$ in graph G is a homomorphic mapping h from the pattern vertices in $V_{Q_x} \cup V_{Q_y}$ to V of G. Note that $h(x_0)$ and $h(y_0)$ are possibly disconnected vertices in G. These allow us to compare entities that are disconnected and may have different topological structures.

It has been shown that it is in PTIME to check the existence of matches of a star-shaped dual pattern [22]. In contrast, graph pattern matching is NP-complete for generic graph patterns (cf. [39]).

The notations and glossaries are summarized in Table 1.

2.2 Graph Cleaning Rules

We next review GCRs [22], starting with their predicates.

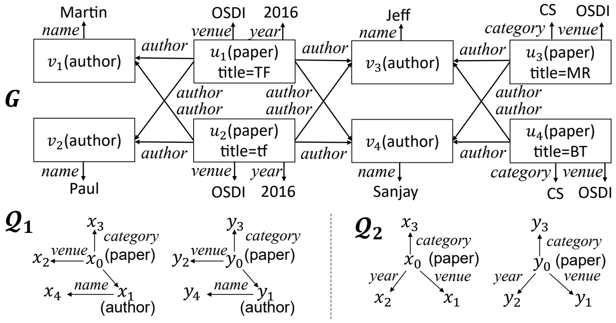


Fig. 1. Graph G and the dual-star patterns of two GCRs φ_1 and φ_2.

Predicates. A *predicate* of pattern $Q[x_0, y_0]$=$\langle Q_x[x_0, \bar{x}], Q_y[y_0, \bar{y}]\rangle$ is

$$p ::= x.A \oplus y.B \mid z.A \oplus c \mid \mathcal{M}(x.\bar{A}, y.\bar{B}),$$

where $\oplus$ is one of $=, \neq, <, \leq, >, \geq$; $x \in \bar{x}$ and $y \in \bar{y}$ are variables in star patterns Q_x and Q_y, respectively, and $z \in \bar{x} \cup \bar{y}$; c is a constant; A and B are attributes in Υ; and $x.\bar{A}$ is a list of attributes at x.

Here $\mathcal{M}(x.\bar{A}, y.\bar{B})$ is an ML model that returns true iff $\mathcal{M}$ predicts true at $(x.\bar{A}, y.\bar{B})$. In principle, $\mathcal{M}$ can be any ML model that returns a Boolean value (*e.g.,* $\mathcal{M} \geq \theta$ for a predefined bound θ).

We refer to (a) $\mathcal{M}(x.\bar{A}, y.\bar{B})$ as *ML predicate*, (b) $x.A \oplus y.B$ and $\mathcal{M}(x.\bar{A}, y.\bar{B})$ as *binary predicates*, and (c) $z.A \oplus c$ as a *unary predicate*. Note that x and y are variables for vertices in Q_x and Q_y, respectively, to pairwisely compare features of centers x_0 and y_0.

GCRs. A *graph cleaning rule* (GCR) φ has the following form:

$$Q[x_0, y_0](X \rightarrow p_0),$$

where $Q[x_0, y_0]$ is a dual pattern, X is a conjunction of predicates of Q, and p_0 is a predicate of Q. Moreover, in precondition X, (1) binary predicates are defined on either two leaves or the centers x and y in the two stars of Q, respectively; and (2) each leaf may carry at most one such predicate but multiple constant predicates $z.A \oplus c$.

We refer to Q and $X \rightarrow p_0$ as the *pattern* and *dependency* of φ, and X and p_0 as *precondition* and *consequence* of φ, respectively.

Intuitively, restrictions (1) and (2) above are to strike a balance between expressive power and efficiency. Indeed, it is in PTIME to detect and correct errors with GCRs [22]. We separate Q and X to (i) visualize the topology of x_0 and y_0 by Q, and (ii) speed up evaluation by leveraging the locality of pattern matching.

Example 1: Figure 1 showcases two GCRs for conflict resolution (CR) and entity resolution (ER), respectively, in a citation network.

(1) $\varphi_1 = Q_1[x_0, y_0](x_2.\text{val} = y_2.\text{val} \wedge x_4.\text{val} = y_4.\text{val} \wedge y_3.\text{val} = \text{CS} \rightarrow x_3.\text{val} = \text{CS})$. It says that paper x_0's category should be CS if (a) it shares a coauthor with another paper y_0; (b) x_0 and y_0 are published in the same venue; and (c) y_0 is a CS paper;

(2) $\varphi_2 = Q_2[x_0, y_0](\mathcal{M}_s(x_0.\text{title}, y_0.\text{title}) \wedge x_1.\text{val} = y_1.\text{val} \wedge x_2.\text{val} = y_2.\text{val} \wedge x_3.\text{val} = y_3.\text{val} \rightarrow x_0.\text{id} = y_0.\text{id})$. It suggests x_0 and y_0 be the same paper if (a) their titles roughly match (checked by a similarity-checking model $\mathcal{M}_s$); (b) they are published in the same venue in the same year; and (c) they belong to the same category. □

Semantics. We apply GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ to graphs G. Let h denote a match of the dual pattern Q in G. A match h *satisfies* a predicate p, denoted by $h \models p$, if one of following conditions is satisfied: (a) when p is $x.A \oplus y.B$, vertex $h(x)$ (resp. $h(y)$) carries attribute A (resp. B), and

Algorithm 1: Parallel GCR discovery.

```
Input: A sample graph $G$, thresholds for support $\sigma$ and confidence $\delta$, $\Sigma = \emptyset$.
Output: A cover $\Sigma_c$ of discovered rules.
1 while $\Sigma_i$ := GenLevelGCRs($\Sigma, \sigma, \delta$).Next() is not empty do
2     for each $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ in $\Sigma_i$ do in parallel
3         $H$ := EnumCandidateMatches ($Q, X, G, G$);
4         pos, neg := Filter($H, p_0$).count, Filter($H, \neg p_0$).count;
5         if pos $\geq \sigma$ and $\frac{\text{pos}}{\text{pos+neg}} \geq \delta$ then $\Sigma := \Sigma \cup \{\varphi\}$;
6 return a cover $\Sigma_c$ of $\Sigma$;
```

Algorithm 2: Parallel error detection/correction.

```
Input: Airty graph $G$, ground truth $\Gamma$ in $G$, a set $\Sigma$ of GCRs.
Output: A set $\mathcal{F}$ of errors/fixes in $G$.
1 $\Delta\Gamma := \Gamma$; $\mathcal{F} = \emptyset$;        /* $\Delta\Gamma$ keeps track of changes to ground truth $\Gamma$. */
2 for each $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ in $\Sigma$ do in parallel
3     $H$ := EnumCandidateMatches($Q, X, G, \Gamma$); cfl := Filter($H, \neg p_0$);
4     if error detection then $\mathcal{F} := \mathcal{F} \cup$ cfl;
5     if error correction then fix cfl in $G$; add fixes to $\mathcal{F}$ and $\Delta\Gamma$;
6 if $\Delta\Gamma$ not empty then $\Gamma := \Gamma \cup \Delta\Gamma$; $\Delta\Gamma = \emptyset$ goto line 2; else return $\mathcal{F}$;
```

$h(x).A \oplus h(y).B$; (b) when p is $z.A \oplus c$, attribute A exists at $h(z)$ and $h(z).A \oplus c$; and (c) when p is $\mathcal{M}(x.\bar{A}, y.\bar{B})$, the ML model $\mathcal{M}$ predicts true at $(x.\bar{A}, y.\bar{B})$.

For a set X of predicates, we write $h \models X$ if h satisfies *all* the predicates in X. We refer to a match h of Q in G as a *candidate match* of φ if $h \models X$. We write $h(\bar{x}) \models X \rightarrow p$ if $h \models X$ implies $h \models p$.

A graph G *satisfies* GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$, denoted by $G \models \varphi$, if for all matches h of $Q[x_0, y_0]$ in G, $h \models X \rightarrow p_0$.

GCRs support all the primitives of relational data cleaning rules such as conditional functional dependencies (CFDs) [24], denial constraints (DCs) [11] and matching dependencies (MDs) [23]. Here we extend the GCRs of [22] with comparison operators $<, \leq, >, \geq$.

2.3 Parallel Graph Cleaning with GCRs

We next outline algorithms for GCR-based graph cleaning [22].

Rule discovery. Given graph G, this task is to mine a cover Σ_c of GCRs from a sample G_s of G. It discovers a set Σ in which each GCR has support and confidence above predefined thresholds σ and δ, respectively. It returns the cover Σ_c of Σ, which is a minimum set of non-redundant GCRs that is "equivalent to" Σ, *i.e.*, each GCR in Σ can be entailed by Σ_c via logic implication, and vice versa.

As shown in Algorithm 1, parallel discovery works in rounds [22]. Each round generates a batch of unverified GCRs level-wise, with pruning based on the monotonicity of support [22] (line 1). It enumerates all candidate matches H of each GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ (line 3) to calculate its support and confidence (line 5) by counting candidate matches in H that satisfy or contradict p_0 (line 4). If both reach their thresholds (σ and δ), φ is added to Σ (line 5). Finally, a cover Σ_c of Σ is computed and returned (line 6).

Error detection. This is to apply the mined Σ to G, and catches violations of the GCRs. For a GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$, a *violation* is $h(p_0)$, where h is a valuation of φ in G such that $h \models X$ but $h \not\models p_0$, and $h(p_0)$ is p_0 in which each variable z is instantiated with vertex $h(z)$. MiniClean returns all such violations as errors, including duplicates (u.id = v.id) and conflicts ($u.A \neq v.B$).

Algorithm 2 shows the parallel error detection algorithm of [22]. Starting from a set of ground truth Γ (validated facts of the form $u.A \oplus v.B$ and $u.A \oplus c$), it applies each GCR φ in Σ in parallel, invoking the same procedure for enumerating candidate matches H as in rule discovery, except that the precondition X of φ is validated with the facts in Γ (line 3). It then checks each candidate match h in H against G, and records the violation in cfl if $h \not\models p_0$ (line 3). Finally, it returns the set $\mathcal{F}$ consisting of all violations (line 4).

Error correction. To fix duplicates and conflicts in G, it chases G with the GCRs in Σ [32]. It accumulates ground truth in a set Γ, and references Γ in the process. The process is Church-Rosser [7], *i.e.,* it converges at the same fixes no matter what rules in Σ are used and in what order they are applied. It returns the fixed graph G as output, and extends Γ with the fixes for subsequent corrections.

Error correction has a workflow similar to error detection (Algorithm 2), except that (a) it fixes identified conflicts in G in-place, rather than simply keeping a record of them; (b) it recursively applies the corrections to G, *i.e.,* a chase (the goto statement in line 6), by maintaining the ground truth updates $\Delta\Gamma$ (*i.e.,* $h(p_0)$ for each valid match h in H that makes a correction), until no more errors are found (line 5–6); and (c) it returns the set of fixes $\mathcal{F}$.

Example 2: We apply GCRs φ_1 and φ_2 in Example 1 to correct errors in graph G (Figure 1). (a) Initially, GCR φ_1 resolves a conflict with papers u_1 and u_2. Their categories are updated to CS because they share a coauthor v_4 and a venue with a CS paper u_4. (b) After the corrections are applied, GCR φ_2 identifies and merges the duplicate papers u_1 and u_2 in the next chase round. □

Match enumeration. At the heart of all graph cleaning tasks is the enumeration of all candidate matches for a GCR φ. As shown in Algorithms 1–2, prior algorithms [22] perform match enumeration by invoking a PTIME procedure EnumCandidateMatches for star patterns, which is built upon the notion of *scattered matches* [33].

Given a path pattern $\langle p_0, l_0, \ldots, p_l\rangle$ and a graph G, the scattered matches of a node p_i ($0 \leq i \leq l$) include all vertices v in G such that (a) v matches p_i; and (b) there exists an edge between v and a scattered match of p_{i+1} for $i < l$ with matching edge label l_i. Based on this notion, it enumerates the matches of the source p_0 by dynamic programming. Starting from the sink p_l, it finds the scattered matches of each node one after another along the path in reverse, until p_0 is reached. It returns only matches of source p_0 and leaf p_l. Scattered matches can thus be done in $O(|G|l)$ time.

Procedure EnumCandidateMatches extends this idea to dual-star patterns Q and precondition X as follows: (a) each star pattern is trivially decomposed into a set of paths from the center to each leaf; (b) unary predicates in X are treated as additional constraints to filter scattered matches; and (c) a binary predicates across stars on nodes p_m and q_n are handled by maintaining valid scattered matches of $\langle p_m, q_n\rangle$, where p_m and q_n are either centers or leaves of the two stars. With these strategies, candidate matches of a GCR φ can be enumerated by assembling the paths matches at the center pairs. The entire procedure takes $O((|Q| + |X|)^2|G|^2)$ time.

3 MiniClean: A Single Machine System

This section presents an overview of system MiniClean.

Challenges. As suggested by Algorithms 1–2, match enumeration dominates the computational cost of the graph cleaning tasks. For each GCR $\varphi = Q[x_0, y_0](X \to p_0)$, it requires pattern matching of dual-star pattern Q in graph G and checking the precondition X in an integrated process. The process is repeatedly invoked for different GCRs. Existing approach [22, 33], *i.e.,* EnumCandidateMatches via scattered matches, is designed for multi-machine systems. It partitions G into fragments, loads the fragments to memory of different machines at once, and enumerates matches of Q in the fragments with different machines in parallel. However, a single machine cannot afford the resources. To process big graphs G and make effective use of GPU, it encounters the following challenges.

Algorithm 3: Two-stage match enumeration workflow.

```
Input: A set $\Sigma$ of GCRs, a graph $G$, ground truth $\Gamma$.
Output: A set $H_i$ of candidate matches for each $\varphi_i$ in $\Sigma$.
1  $\Psi$ := BreakAndBundle($\Sigma$);                /* $\Psi$ is a set of mono-star bundles. */
2  foreach mono-star bundle $\psi$ in $\Psi$ do $C_\psi$ := MatchStar($\psi, G, \Gamma$);
3  for GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ in $\Sigma$ do in parallel
4      $H_\varphi$ := EnumCrossStar ($\varphi, C_{\psi_x}, C_{\psi_y}$);
5  return $\bigcup_{\varphi \in \Sigma} H_\varphi$;

Procedure EnumCrossStar ($\varphi = Q[x_0, y_0](X \rightarrow p_0), C_x, C_y$):
       /* $C_x, C_y$: star candidates for mono-star bundles derived from $\varphi$. */
6      return $(C_x \times C_y)$.bucketize($X$).filter($X$);
```

(1) Irregular data access. Finding scattered matches requires frequent access to the graph structure, leading to irregular access patterns and control flows. Such workload cannot be easily accelerated by GPUs, underutilizing their massive SIMD parallelism.

(2) Repetitive computation. Graph cleaning applies a set Σ of GCRs to graph G. EnumCandidateMatches matches each GCR in Σ in separate to simplify synchronization. This, however, can lead to repetitive computation *w.r.t.* matching common substructures, which are regular among GCRs since Σ is mined in a level-wise process. A single machine cannot afford the redundant computations.

(3) Memory usage. Maintaining scattered matches introduces excessive intermediate results, *e.g.,* all valid combinations of candidate path pairs. It typically exceeds the memory capacity of a single machine, leading to either frequent disk swaps or memory overflows.

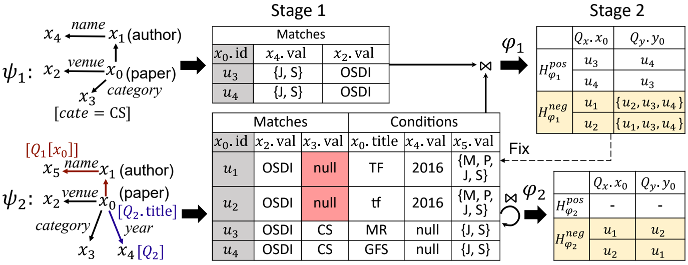
(4) Rigid parallelism. As opposed to enumeration of a single pattern [43, 80, 86, 88, 89], we have to match different patterns of various GCRs. At each fragment, EnumCandidateMatches exploits independent parallelism, by enumerating the scattered matches of different path patterns in parallel. This does not fully utilize the hardware concurrency of GPUs. Worse yet, it incurs even heavier memory footprint, since parallel tasks maintain their intermediate results in memory at the same time, multiplying the space overhead.

Workflow overview. To overcome these challenges, MiniClean adopts a set of strategies to optimize match enumeration. Departing from algorithms of [22, 33], it (a) decouples and reassembles the two star patterns in two stages, allowing heavy-duty work to take regular data structures and be accelerated on GPUs; (b) bundles similar stars together for grouped match enumeration, mitigating repetitive computation (Section 4.1); (c) compresses intermediate data to reduce memory footprint (Section 4.2); and (d) adopts a multi-mode parallel model to maximize CPU–GPU synergy (Section 5).

More specifically, as shown in Algorithm 3. MiniClean adopts a two-stage workflow to enumerate matches for a set of GCRs.

Stage 1: Star matching. Given a set Σ of GCRs, graph G, and ground truth Γ, this stage reorganizes Σ into a set Ψ of *mono-star bundles* (line 1), and enumerates *star candidates* of each bundle in Γ (line 2).

The reorganization (line 1) starts with decomposing each GCR in Σ into two *mono-star components*. That is, it breaks GCR $\varphi = Q[x_0, y_0](X \rightarrow p_0)$ into $\omega_x = Q_x[x_0, \bar{x}](X_x, F_{Q_x})$ and $\omega_y = Q_y[y_0, \bar{y}](X_y, F_{Q_y})$, where (a) $Q_x[x_0, \bar{x}]$ (resp. $Q_y[y_0, \bar{y}]$) is the left (resp. right) star of $Q[x_0, y_0]$; (b) X_x (resp. X_y) is the conjunctions of all unary predicates defined on $\bar{x}$ (resp. $\bar{y}$); and (c) each vertex v in V_{Q_x} (resp. V_{Q_y}) carries a tuple $F_{Q_x}(v)$ (resp. $F_{Q_y}(v)$) of variable attributes associated with p_0 or binary predicates of X; their values will be materialized during the following match enumeration.



Matches		
x_0.id	x_4.val	x_2.val
u_3	{J, S}	OSDI
u_4	{J, S}	OSDI

Matches			Conditions		
x_0.id	x_2.val	x_3.val	x_0.title	x_4.val	x_5.val
u_1	OSDI	null	TF	2016	{M, P, J, S}
u_2	OSDI	null	tf	2016	{M, P, J, S}
u_3	OSDI	CS	MR	null	{J, S}
u_4	OSDI	CS	GFS	null	{J, S}

	$Q_x.x_0$	$Q_y.y_0$
$H^{pos}_{\varphi_1}$	u_3	u_4
	u_4	u_3
$H^{neg}_{\varphi_1}$	u_1	$\{u_2, u_3, u_4\}$
	u_2	$\{u_1, u_3, u_4\}$

	$Q_x.x_0$	$Q_y.y_0$
$H^{pos}_{\varphi_2}$	-	-
$H^{neg}_{\varphi_2}$	u_1	u_2
	u_2	u_1

Fig. 2. The two-stage workflow of match enumeration.

Intuitively, each mono-star component preserves all information local to a star pattern of φ, while keeping track of cross-star associations via F_{Q_x} and F_{Q_y}. Then, "similar" mono-star components are grouped into a composite mono-star structure, termed a *mono-star bundle*, using the bundling techniques proposed in Section 4.1.

For each mono-star bundle ψ in Ψ, MiniClean performs match enumeration for ψ using CPUs (line 2), materializing *star candidates*, *i.e.*, matches of a mono-star bundle that satisfy its associated unary predicates with validated facts in Γ. MiniClean adapts the algorithm of [33] to mono-star bundles via scattered matches, in $O((|Q_\psi|+|X_\psi|)|G|)$ time. The final result is a set of star candidates, compressed in a table-like structure (see Section 4.2) to facilitate efficient parallelism in the next stage.

Stage 2: Cross-star enumeration. Using star candidates from Stage 1, this stage completes the match enumeration process by assembling the star candidates for both stars of each GCR with GPUs.

For a given GCR φ, MiniClean retrieves the star candidates C_{ψ_x} and C_{ψ_y}, for the star patterns Q_x and Q_y of φ (line 4), respectively. It then scans each pair of candidates in the Cartesian product $C_{\psi_x} \times C_{\psi_y}$, checking whether the binary predicates are satisfied (line 6).

The two-stage workflow introduces the following benefits.

Reduced intermediate results. It makes more efficient use of the limited memory capacity. It decouples the dual patterns of a GCR, thus avoiding the need to maintain combinations of candidate path *pairs*. As will be seen in Section 4.2, the materialized star candidates take at most $O(|Q||V|^2)$ space for each GCR, as opposed to $O(|Q|^2|V|^4)$ space by EnumCandidateMatches of [22, 33]. Moreover, it substantially speeds up the computation as will be seen in Section 6.

Pipelining. The workflow optimizes performance by exploring the strengths of CPU and GPU for different stages of the workflow.

Star matching is executed on the host CPU, utilizing all cores. This stage accesses irregular graph data, which is difficult for GPUs due to their SIMD architecture. CPUs are better suited for managing these. Moreover, star matching generates relatively large sets of star candidates, which the CPU can better manage by leveraging its access to larger main memory and SSDs. This is hard for GPUs, which have limited memory and no direct access to external storage.

MiniClean offloads cross-star enumeration to the GPU. Before this stage, the star candidates are organized into table-like structures, well-suited for GPU processing. The stage involves checking the Cartesian product of star candidates, an expensive yet highly parallel task at which GPUs excel due to their massive hardware concurrency with SIMD, substantially speeding up the process.

MiniClean offers efficient pipelined execution in this way, improving resource utilization and system performance. Its bundling strategy (Section 4.3) balances the workload between the two stages.

Example 3: Using GCRs $\Sigma = \{\varphi_1, \varphi_2\}$ from Example 1, Figure 2 depicts the two-stage workflow of match enumeration. For simplicity, some optimization strategies are omitted.

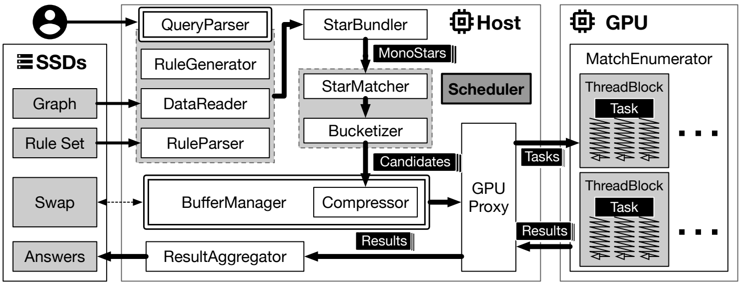


Fig. 3. The pipelined architecture of MiniClean.

(1) GCRs φ_1 and φ_2 have 4 star patterns. Based on similarity, Stage 1 produces two mono-star bundles: (a) ψ_1 with the right star of φ_1, (b) ψ_2, a bundle of the left star of φ_1 and both of φ_2. Then for each bundle, it finds all star candidates in G, materialized in a table.

(2) Stage 2 assembles candidates for φ_1 and φ_2 with ψ_1 and ψ_2. For φ_1, it checks all pairs of star candidates of ψ_1 and ψ_2, filtered with X_1. For φ_2, it checks the self join of candidates for ψ_2, filtered by X_2. □

Pipelined architecture. MiniClean takes as input a graph G and thresholds σ and δ. It discovers a cover Σ_c of the set Σ of GCRs that have support above σ and confidence above δ, from a sample of G. Then upon users' request, it detects and/or corrects the errors in G by applying the GCRs in Σ_c. As illustrated in Figure 3, MiniClean is built around a pipeline that effectively orchestrates the two-stage workflow for match enumeration. It uses a central Scheduler to manage memory buffer, overlap I/O operations and CPU/GPU computations, maximizing resource utilization (detailed in Section 5). The architecture organizes the following pipeline stages.

(1) Job configuration. QueryParser takes user input, prepares data, and configures jobs for match enumeration. It has (a) DataReader, which loads input graphs G from storage; (b) RuleGenerator, which generates unverified GCRs level-wise, and (c) RuleParser, which parses GCRs mined offline for error detection and correction.

(2) Star matching. Given a match enumeration job with a set Σ of GCRs, RuleBundler transforms Σ into a set Ψ of mono-star bundles. For each bundle ψ in Ψ, StarMatcher identifies all star candidates C_ψ. The candidates are then passed to Bucketizer, which materializes them into buckets based on specific attribute values. It groups matches with similar attribute values to speed up similarity checking and subsequent verification for binary predicates.

(3) Candidate management. BufferManager manages materialized star candidates in buckets, for (a) compressing candidates within each bucket, (b) managing memory by transparently swapping data between the host memory and SSDs, treating the former as cache with schedule-based evictions; and (c) packing the compressed buckets into discrete tasks, preparing them for GPU processing.

(4) Cross-star enumeration. With star candidates organized, tasks are delegated to the GPU for parallel processing. GPUProxy manages the GPUs, where MatchEnumerator performs Stage 2 of the workflow. Results for each task are collected for the final response.

(5) Response. Finally, ResultAggregator compiles the results of candidate matches of GCRs and synthesizes them as a coherent output.

4 Optimizing Star Matching

This section presents star bundling (Section 4.1), candidate compression (Section 4.2), and adaptive bundling strategies (Section 4.3) for optimizing star matching and balancing CPU/GPU workloads.

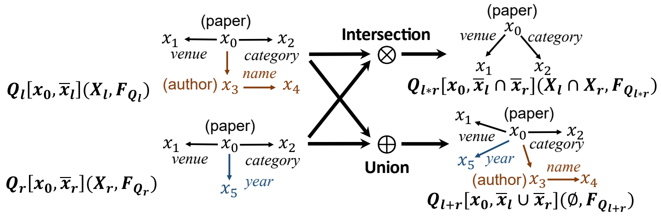


Fig. 4. Bundling of mono-star components $\varphi_1.Q_y$ and $\varphi_2.Q_y$.

4.1 Bundled Processing

We start with bundled matching of mono-star bundles to reduce the cost of candidate enumeration for a group of star patterns.

Motivation. Star matching takes as input a set Ψ of mono-star components and enumerates star candidates for each. It is costly. Worse yet, it cannot be accelerated on a GPU due to the irregular memory access and the data-dependent control flow on a graph G. For a mono-star component $Q[x_0, \bar{x}](X, F_Q)$ in Ψ, it takes $O((|Q| + |X|)|G|)$ time to enumerate the star candidates of ψ in G, even via scattered matches. While it is hard to further speed up single star matching, can we do better in handling a large set Ψ at once?

Strawman 1: Independent matching. With multiple machines, Algorithms 1–2 separately match each mono-star from Ψ. However, this solution does not consider common substructures, *e.g.*, path patterns shared by the mono-stars. Common substructures are regular in practice, since the mono-star components in Ψ are derived from GCRs mined in a level-wise manner. This leads to (a) *redundant computation* since matching common substructures yields repetitive work, and (b) *excessive CPU–GPU transfers*, since candidates for the common substructures inflict repetitive data transfers.

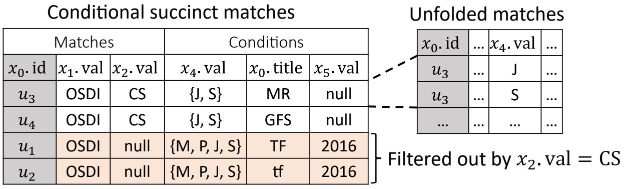
These suggest that we leverage common substructures shared by the mono-star components in Ψ. Below is an attempt.

Strawman 2: Extracting common substructures. One may want to (a) extract common substructures (*e.g.*, paths) of mono-star components, (b) materialize their matches, and (c) assemble the matches into star candidates for each ω in Ψ by joining at the central vertex.

In principle, this approach adapts the concept of *interleaved execution* from multi-query optimization [49, 71, 73] and adapts it to star matching; however, it is surprisingly expensive. The common substructures, especially shared paths, often have a large number of matches, since they relax the constraints of the original mono-star components. This leads to heavy memory footprint to match a single star pattern, and worse yet, the materialization and assembly of these matches introduce excessive overhead *w.r.t.* joining and filtering. Such inefficiencies are exacerbated in practice. To enumerate matches for star Q_y of Q_1 in Figure 1, Strawman 2 takes 10.7s, with 6.7s for assembling alone. In contrast, matching via scattered matches takes 2.6s, using ≤ 63.2% of peak memory.

Our solution. In light of these, we propose bundled processing to reduce the materialization of candidate matches and repetitive computations caused by common substructures. Instead of breaking down mono-star components in Ψ into smaller pieces, MiniClean groups "similar" ones into a larger bundle and matches it as a whole.

Bundling two stars. It is possible to bundle two mono-star components $\omega_l = Q_l[x_0, \bar{x}_l](X_l, F_{Q_l})$ and $\omega_r = Q_r[x_0, \bar{x}_r](X_r, F_{Q_r})$ into a mono-star bundle ψ if their center x_0 share the same label. Intuitively, it is to "join" stars ω_l and ω_r at the center x_0, while keeping track of both their common paths and distinct ones.



Conditional succinct matches

Matches			Conditions		
x_0. id	x_1. val	x_2. val	x_4. val	x_0. title	x_5. val
u_3	OSDI	CS	{J, S}	MR	null
u_4	OSDI	CS	{J, S}	GFS	null
u_1	OSDI	null	{M, P, J, S}	TF	2016
u_2	OSDI	null	{M, P, J, S}	tf	2016

Filtered out by x_2. val = CS

Unfolded matches

x_0. id	...	x_4. val	...
u_3	...	J	...
u_3	...	S	...
...	...	...	...

Fig. 5. Conditional succinct table.

Formally, the bundle ψ of mono-stars ω_l and ω_r is a tuple $\langle \omega_l \otimes \omega_r, \omega_l \oplus \omega_r \rangle$. Here (a) $\omega_l \otimes \omega_r = Q_{l*r}[x_0, \bar{x}_l \cap \bar{x}_r](X_l \cap X_r, F_{Q_{l*r}})$ represents the *common substructure* of ω_l and ω_r, where pattern Q_{l*r} is the maximum common induced subgraph of Q_l and Q_r, and $F_{Q_{l*r}}$ includes all variable attributes shared by F_{Q_l} and F_{Q_r}; (b) $\omega_l \oplus \omega_r = Q_{l+r}[x_0, \bar{x}_l \cup \bar{x}_r](\emptyset, F_{Q_{l+r}})$ represents the *complete structure* of ω_l and ω_r. Its pattern Q_{l+r} is the join of Q_l and Q_r at the common center x_0, with the paths in Q_{l*r} deduplicated; and (c) $F_{Q_{l+r}}(v) = F_{Q_l}(v) \cup F_{Q_r}(v)$ carries attributes in either F_{Q_l} or F_{Q_r}.

By matching the mono-star bundle ψ against graph G, MiniClean avoids repetitive computation on the common substructure $\omega_l \otimes \omega_r$. Moreover, the complete structure preserves all useful properties. Compared to independent matching, its overall computational workload is reduced by a factor of $\Theta(\frac{|Q_{l*r}|+|X_l \cap X_r|}{|Q_l|+|Q_r|+|X_l|+|X_r|})$. Moreover, this approach produces a single shared set of star candidates for the mono-star bundles, significantly reducing both the size of intermediate results and the volume of CPU–GPU data transfers.

Example 4: Continuing with Example 1, Figure 4 illustrates the bundle ψ_2 of mono-star components $\varphi_1.Q_y$ and $\varphi_2.Q_y$. Here we relabel the two as $Q_l[x_0, \bar{x}_l](X_l, F_{Q_l})$ and $Q_r[x_0, \bar{x}_r](X_r, F_{Q_r})$, where predicates $X_l = \{x_2.\text{val} = \text{CS}\}$, $X_r = \emptyset$, attributes $F_{Q_l} = \{x_1.\text{val}, x_2.\text{val}, x_4.\text{val}\}$, and $F_{Q_r} = \{x_0.\text{title}, x_1.\text{val}, x_2.\text{val}, x_5.\text{val}\}$.

The bundle ψ_2 has (1) $Q_{l*r}[x_0, \bar{x}_l \cap \bar{x}_r](X_l \cap X_r, F_{Q_{l*r}})$ as the common substructure, where $X_l \cap X_r$ includes $x_2.\text{val} = \text{CS}$ and $F_{Q_{l*r}} = \{x_1.\text{val}, x_2.\text{val}\}$; and (2) $Q_{l+r}[x_0, \bar{x}_l \cup \bar{x}_r](\emptyset, F_{Q_{l+r}})$ is the complete structure, where $F_{Q_{l+r}} = \{x_0.\text{title}, x_1.\text{val}, x_2.\text{val}, x_4.\text{val}, x_5.\text{val}\}$. □

Recursive bundling. Given a large set of mono-star components Ψ, MiniClean performs bundling recursively, by selecting two mono-stars based on a similarity measure, removing both from Ψ, bundling the pair and adding their bundle back to Ψ. The process iterates until no more mono-stars can be bundled or it is suppressed by Scheduler for load balancing (see Section 4.3). Recursive bundling is a simple extension to bundling two mono-stars, *i.e.,* bundling $\psi_l = \langle \omega_l^*, \omega_l^+ \rangle$ and $\psi_r = \langle \omega_r^*, \omega_r^+ \rangle$ yields $\psi_{l,r} = \langle \omega_l^* \otimes \omega_r^*, \omega_l^+ \oplus \omega_r^+ \rangle$.

4.2 Star Candidate Compression

We next introduce a compressed data structure for materializing bucketized star candidates, to reduce the size of intermediate results. For clarity, we focus on a single bucket in this discussion.

Background. Matching a mono-star bundle generates a large number of star candidates. A naive approach to materializing them is a large table, where each row represents a complete match for the star pattern. It takes $O(|V|^{|Q|})$ space since the rows consist of the Cartesian product of all scattered matches of leaf nodes. To reduce the intermediate results, we develop a more compact representation.

Succinct table. We propose a join-free succinct table for star candidates, compressing a long table losslessly. It organizes the star candidates around the central node of the star pattern, as follows.

(1) *Rows.* Each row is indexed by a unique vertex v in G, where v is a match for the center x_0 of the star pattern. In other words, a row keyed by v encodes all star candidates centered at v.

(2) *Columns.* In a row keyed by v, each column encodes a complete list of the matches of a specific leaf. The columns only include matches of leaves for cross-star checking, *i.e.,* leaf IDs and marked attributes (Section 3) in consequence p_0 or binary predicates.

As opposed to join-free structures [2, 62, 78, 85], this succinct table does not require additional metadata to be lossless. It leverages two properties of star candidates: (a) star candidates are combinations of path matches centered on a common vertex, and (b) the marked attributes only exist at the center or leaf nodes. Thus, its size is in $O(|V|^2|Q|)$ by grouping path matches by their shared center. It makes the buckets compact enough to fit in limited GPU memory without requiring too fragmented bucketization. Moreover, such tables can be efficiently constructed. The algorithm of scattered matches naturally groups leaf matches by center vertices, and directly inserts appropriate cells into the table.

Conditioning on bundled stars. An additional complication is that we must differentiate the star candidates for each individual star in the bundle. Adapting conditional tables [6, 8, 42, 46, 79], we extend the succinct table with condition columns. The conditional structure efficiently differentiates the star candidates for each mono-star component while avoiding unnecessary materialization.

More specifically, consider the bundle $\psi = \langle \omega_1 \otimes \omega_2, \omega_1 \oplus \omega_2 \rangle$ of mono-stars ω_1 and ω_2. The conditioned succinct table includes (a) *matches*, which are stored in the columns *w.r.t.* $\omega_1 \otimes \omega_2$, and (b) *conditions*, which are stored in additional columns for $(\omega_1 \oplus \omega_2) \setminus (\omega_1 \otimes \omega_2)$, *i.e.,* the distinct substructures of ω_1 and ω_2. If no matches exist for a center v, its corresponding entry is set to null.

Direct candidate retrieval. To recover the star candidates for a mono-star component, *e.g.,* ω_1, we can filter rows by the extended conditions. For each row, one only needs to check the condition columns for ω_1, *i.e.,* all columns that represent the materialized attributes of ω_1 distinct from ω_2. If any value is null, the row does not contribute to a valid match for ω_1, and it can be discarded. If a valid match exists, it is part of the star candidates for ω_1.

We only access relevant matches without recomputing or performing joins, thus supporting fast access to star candidates.

Example 5: Continuing with Example 4, Figure 5 illustrates the conditional succinct table for bundle ψ_2. Its rows are indexed by matches of center x_0 and its columns store: (1) attributes from $F_{Q_{l*r}}$, including x_1.val and x_2.val; and (2) attributes from $F_{Q_{l+r}}$ including x_4.val, x_0.title and x_5.val. The matches of $\varphi_2.Q_x$ can be retrieved by filtering out rows that violate X_1, *i.e.,* x_2.val = CS. □

4.3 Balancing CPU and GPU Workloads

In recursive bundling, it is not always beneficial to bundle as many mono-stars as possible, since bundling shifts the workload from CPU to GPU. Here we discuss this tradeoff and introduce an adaptive bundling strategy to guide the bundling process.

Trade-off in bundling. While bundled processing reduces redundant computations in Stage 1 (star matching on the CPU) via a shared conditional succinct table, it introduces added complexity to Stage 2 (cross-star enumeration on the GPU) for differentiating and recovering the star candidates of each individual mono-star.

Such a trade-off is illustrated in Example 4. Without bundling, the CPU must match 4 mono-stars of GCRs φ_1 and φ_2 separately, incurring redundant computation on the mono-stars of φ_2. The GPU can then directly process these matches. In contrast, matching bundles ψ_1 and ψ_2 yield two succinct tables with 2 and 4 rows, reducing the CPU workload. However, it comes at the cost of additional star candidate recovery work on the GPU, as shown in Example 5.

To optimize workload balance, we propose an adaptive bundling strategy that adjusts the bundling process based on real-time pipeline conditions. This strategy addresses two key questions: (a) should bundling continue, and if so, (b) which pair of mono-stars should be prioritized for bundling? We start with similarity measures that quantify the gains and costs of bundled processing.

Similarity measures. Recursive bundling is conducted by quantifying the similarity between mono-star components. On the one hand, when two components have larger common substructures, bundling them reduces more redundant computations. On the other hand, it introduces overhead when retrieving matches from their shared, materialized star candidates. Hence we introduce two metrics to evaluate both shared and distinct substructures.

Reward. Consider two mono-stars $\omega_1 = Q_1[x_0, \bar{x}_1](X_1, F_{Q_1})$ and $\omega_2 = Q_2[x_0, \bar{x}_2](X_2, F_{Q_2})$ with common substructure $\omega_1 \otimes \omega_2 = Q_{1*2}[x_0, \bar{x}_1 \cap \bar{x}_2](X_1 \cap X_2, F_{Q_{1*2}})$. The *reward* R measures the computational cost saved by matching Q_{1*2} as follows:

$$R(\omega_1, \omega_2) = (|Q_{1*2}| + |X_1 \cap X_2|)|G|.$$

Penalty. The *penalty* measures the additional computational overhead incurred by retrieving individual matches from bundled candidates. The conditional succinct table allows candidate retrieval in a linear scan of all rows. Given the bundle ψ of mono-stars ω_1 and ω_2, the *penalty* P is proportional to the table size:

$$P(\omega_1, \omega_2) = \alpha|C(Q_{1*2}[x_0, \bar{x}_1](X_1 \cap X_2, F_{Q_{1*2}}))|,$$

where α is a constant factor to account for the accelerated filtering scan on the GPU, and $|C(\cdot)|$ denotes the number of star candidates for a given mono-star bundle, which can be efficiently and accurately obtained using cardinality estimation techniques [68].

The reward and penalty quantify the tradeoff between the computational costs of the two workflow stages, executed on the host CPU and GPU, respectively. Intuitively, the reward captures the benefit of bundling by quantifying redundant computation that can be avoided. The larger the common substructure, the greater the benefit. Conversely, the penalty reflects the additional overhead introduced. It increases with the size of the conditional succinct table, *i.e.*, the number of star candidates. Next, we leverage them to balance workloads, thereby improving the pipelined processing.

Adaptive bundling. MiniClean employs an adaptive bundling strategy that dynamically monitors GPU task backlogs and adjusts bundling decisions based on real-time workload conditions.

Its GPUProxy maintains a queue of tasks awaiting GPU processing. It continuously tracks the size of this queue, measured by the total number of star candidates pending GPU processing. Intuitively, the queue size reflects the workload imbalance between the CPU and GPU. A large queue indicates that the GPU is becoming a bottleneck, while a small/empty queue implies GPU underutilization.

Based on the real-time queue size τ, MiniClean operates in three modes, governed by two thresholds, τ_1 and τ_2, where $0 < \tau_1 < \tau_2$:

(1) Greedy mode ($\tau \leq \tau_1$). When the CPU generates star candidates at a rate slower than the GPU can handle, MiniClean aggressively bundles tasks. In this mode, it selects mono-star pairs with the highest reward (R) to maximize the reduction in CPU workload.

(2) Balanced mode ($\tau_1 < \tau \leq \tau_2$). Given a relatively balanced pipeline, MiniClean prioritizes bundling based on the reward–penalty ratio (R/P), optimizing overall system throughput by balancing the trade-offs between bundling benefits and associated overhead.

(3) Disabled mode ($\tau > \tau_2$). When the GPU becomes overloaded, bundling is disabled. Unprocessed mono-star components are processed independently to allow the GPU to catch up, and prevent unnecessary overhead on the system bottleneck.

At a high level, this adaptive strategy dynamically adjusts the bundling process based on the current workload distribution among the CPU and GPU, minimizing the risk of overloading the GPU while allowing the CPU to offload tasks when necessary. The thresholds τ_1 and τ_2 are determined via profiling (see Section 6).

5 Hybrid Parallel Model

This section studies a multi-mode parallel model and its scheduling.

5.1 Multi-mode parallelism

MiniClean employs multiple parallel modes to maximize resource utilization.

Pipelined parallelism. MiniClean pipelines two stages of match enumeration across CPU and GPU, overlapping the two to improve throughput and "canceling" the cost of SSD data swapping and CPU–GPU data transfers. Unlike prior GPU-accelerated systems that offload nearly all computation to GPU and use CPU only for lightweight tasks (*e.g.*, kernel management, result aggregation), it exploits the strengths of both in different stages (Section 3).

SIMD parallelism. MiniClean exploits SIMD parallelism by having GPU threads to concurrently process different data components. By the use of succinct tables (see Section 4.2), a regular data structure for cross-star enumeration to be easily parallelized across GPU threads, it fully utilizes all threads without leaving any idle.

Independent parallelism. MiniClean also enables independent parallelism by running multiple cross-star enumeration tasks on a GPU simultaneously, each processing a pair of star candidate buckets. Since the buckets have variable sizes, a task on small buckets has limited SIMD parallelism, leading to many idle threads if it monopolizes the GPU. As we will see in Section 6, allowing multiple tasks to share the GPU resources substantially improves GPU utilization.

Example 6: Consider the workflow in Example 3. MiniClean exploits (1) pipelined parallelism by attempting to overlap Stage 1 and Stage 2 operations; (2) SIMD parallelism by allowing GPU threads to process different rows of a conditional succinct table; and (3) independent parallelism by processing both tables on the GPU concurrently. The three modes are combined to maximize resource utilization, subject to memory and data placement constraints. □

The scheduling problem. While the model mitigates resource idling, it introduces challenges to scheduling tasks across the pipeline to fully realize the potential of hybrid parallelism.

Problem Statement. Given a set A of m star matching tasks on the CPU and a set B of n cross-star enumeration tasks on the GPU, where each task b in B depends on the results of two tasks $D_l(b)$ and $D_r(b)$ in A (*i.e.,* the mono-star bundles for b's left and right patterns, respectively), it is to find an optimal schedule $\mathcal{S}^* = \langle \bar{\alpha}, \bar{\beta}, \bar{p}(t) \rangle$, where $\alpha_i \in \bar{\alpha}, i \in [1, m]$ (resp. $\beta_j \in \bar{\beta}, j \in [1, n]$) denotes the completion time of task a_i (resp. b_j), and $p_i(t) \in \bar{p}(t), i \in [1, m]$ is for the placement flags of star candidates of α_i at time t. We assume *w.l.o.g.* that the tasks in A and B are sorted in the chronological order of their completion times. Its objective function is

$$\mathcal{S}^* = \arg\min_{\mathcal{S}} \beta_m. \tag{1}$$

It is to minimize the *makespan* of a schedule $\mathcal{S}$, *i.e.,* the completion time β_m of the last cross-star enumeration task b_m. A *valid* schedule is subject to constraints: (1) the completion α_i of task $a_i \in A$ must not be earlier than the total execution time of tasks $a_1, \ldots, a_i$; (2) the star candidates of tasks $D_l(b_j)$ and $D_r(b_j)$ must reside in the GPU memory between the time duration $[\beta_j - c(b_j), \beta_j]$, where $c(b_j)$ is cost of task b_j on the GPU; (3) at any time point t, the total size of host memory (resp. GPU memory)-residing star candidates must not exceed the host memory (resp. GPU memory) capacity; and (4) any change of data placement between $\bar{p}(t)$ and $\bar{p}(t + \Delta t)$ must respect

Algorithm 4: Greedy scheduling algorithm of MiniClean.

```
Input: A set $\Sigma$ of pending GCRs, a graph $G$, and a state DAG $S$ at time $t$.
Output: The next scheduled GCR $\varphi$, and an updated state DAG $S'$.
$\varphi_{\text{target}}$ := null; $S'$ = null; $c = +\infty$;
foreach GCR $\varphi$ in $\Sigma$ do
    $\bar{\text{S}}_\varphi := \text{SimSchedule}(S, \varphi)$;          /* Compute all valid updated statuses. */
    foreach Updated status $S_\varphi$ in $\bar{\text{S}}_\varphi$ when GCR $\varphi$ is scheduled next do
        $c_\varphi := \max\{C_{\text{IO}}(S_\varphi), C_{\text{trans}}(S_\varphi)\}$;
        if $[c_\varphi < c]$ or $[c_\varphi = c$ and $C_{\text{GCR}}(S', \varphi_{\text{target}}) > C_{\text{GCR}}(S_\varphi, \varphi)]$ then $\varphi_{\text{target}} := \varphi$; $S' := S'_\varphi$; $c := c_\varphi$;
return $\varphi_{\text{target}}, S'$;
```

the SSD and GPU transfer bandwidth constraints.

Its decision problem, denoted by DSP, is to decide, given the input and a deadline B, whether there exists a valid schedule $\mathcal{S}$ with makespan at most B. The problem is intractable.

Theorem 1: DSP *is NP-complete.* □

Proof sketch: DSP is in NP since one can guess a schedule and check in PTIME whether it is valid and its makespan is at most B. We show that it is NP-complete by reduction from the partition problem (see [5] for a proof), which is NP-complete (cf. [39]). □

5.2 A Scheduling Strategy

Theorem 1 suggests that an optimal schedule is intractable to find. A particular complication is that task execution requires loading data into limited host or GPU memory, which often leads to data eviction and can complicate the scheduling of subsequent tasks.

We propose a system that makes *locally optimal* scheduling decisions by modeling states. Based on the current state, its Scheduler selects the next tasks to schedule while considering whether there is sufficient memory, either host or GPU, to accommodate the required data. If not, it evicts other data to free space.

Computation state. At any time t, the state of computation is modeled as a directed acyclic graph (DAG) S_t. The DAG extends traditional dependency graphs by capturing dependencies on both computation and data placement. It models the pipelined execution with disk I/O, host–GPU data transfers, and a two-stage workflow.

In S_t, each node represents a *data status*, encoded as a data–placement tuple. It indicates whether a piece of data is materialized in the SSD, the host memory, or GPU memory. Edges between nodes represent computation tasks and/or data loading, with associated weights denoting the costs of materializing the data downstream.

The DAG captures the system state at time t. From S_t, we can derive the system's minimum cost from time t onward: (a) I/O $C_{\text{IO}}(S_t)$, (b) data transfer $C_{\text{trans}}(S_t)$, (c) host computation $C_{\text{cpu}}(S_t)$, and (d) enumeration latency $C_{\text{GCR}}(S_t, \varphi)$ for a GCR φ. We can also estimate the host and GPU memory usage, denoted by $M_{\text{cpu}}(S_t)$ and $M_{\text{gpu}}(S_t)$, respectively. Further details are deferred to [5].

Scheduling strategy. At any time t, we greedily decide the next tasks and manage data eviction. We select one pending GCR φ from Σ as the target, and prioritize all tasks on which φ depends. If the host or GPU memory has no room for the required data materialization, we selectively evict other unused data to make room.

Since determining the global makespan of the remaining computation is intractable, we use *scheduling overhead* as a local metric. The overhead indicates the added cost to the system bottleneck

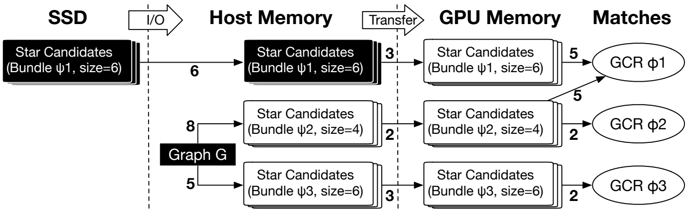

SSD
I/O
Host Memory
Transfer
GPU Memory
Matches
Star Candidates (Bundle ψ1, size=6)
6
Star Candidates (Bundle ψ1, size=6)
3
Star Candidates (Bundle ψ1, size=6)
5
GCR ϕ1
5
8
Star Candidates (Bundle ψ2, size=4)
2
Star Candidates (Bundle ψ2, size=4)
2
GCR ϕ2
Graph G
5
Star Candidates (Bundle ψ3, size=6)
3
Star Candidates (Bundle ψ3, size=6)
2
GCR ϕ3

Fig. 6. State DAG, where dark nodes denote materialized data.

when scheduling φ. These costs stem from data evictions from the host or GPU memory, incurring extra I/O and data transfer for pending tasks. The overhead helps us decide on resource reallocation.

Given state DAG S_t and GCR φ to schedule, we estimate the scheduling overhead as follows. (a) Data materialization: Mark all data statuses associated with φ as materialized. (b) Memory feasibility: Check whether the host/GPU memory can accommodate the materialization. (c) Eviction plans: Generate updated states $\bar{\mathrm{S}}_\varphi$ based on data eviction strategies under memory constraints. (d) Bottleneck cost: For each updated state S_φ in $\bar{\mathrm{S}}_\varphi$, calculate the bottleneck cost c_φ, *i.e.,* the maximum I/O and data transfer cost.

Algorithm 4 outlines our scheduling algorithm. It iterates over all pending GCRs in Σ and simulates the impact of scheduling each one (line 2). The goal is to find the GCR φ_{target} and corresponding updated state S' that minimize the scheduling overhead. (1) For each GCR φ, we compute all valid updated states $\bar{\mathrm{S}}_\varphi$ (line 3). (2) For each updated state S_φ in $\bar{\mathrm{S}}_\varphi$, we compute the bottleneck cost c_φ (line 5). (3) The GCR φ with the lowest cost is selected as the target φ_{target}, and its updated state S' is recorded (line 6).

Intuitively, the strategy aims to reduce the frequency of forced evictions, which typically increases overhead. By focusing on GCRs that minimize bottleneck costs, we exploit data locality and reduce the cost of data reloads. If multiple GCRs have the same overhead, we prioritize the GCR φ with the lowest remaining enumeration cost $C_{\text{GCR}}(S_t, \varphi)$. We favor tasks closer to completion (line 6), ensuring that the pipeline remains filled with tasks ready for execution. In this way, we balance scheduling efficiency with resource management, to continue processing tasks without unnecessary delays.

Example 7: Continuing with Example 3, Figure 6 depicts a state DAG S_t for GCRs φ_1 and φ_2, and another GCR φ_3. At time t, the star candidates for bundle ψ_1 have been materialized in the host memory, while bundles ψ_2 and ψ_3 have not been computed yet.

With S_t, our scheduling strategy tests each pending GCR (line 2). (1) The bottleneck cost c_{φ_1} of φ_1 is always higher than that of φ_2, due to their shared critical path. Thus, φ_1 is not scheduled first due to the positive condition at line 6 with φ_2. (2) When the host memory capacity is 11, it schedules φ_2, because it incurs no overhead, while scheduling φ_3 would lead to eviction of ψ_1's star candidates, increasing the I/O cost to 10. (3) If the host memory capacity is 13, it schedules φ_3, since neither φ_2 nor φ_3 incurs scheduling overhead, and φ_3 has lower remaining cost (line 6). □

Interaction with adaptive bundling. Bundling decisions may directly alter the topology of the state DAG, creating a tight coupling between the scheduling and bundling. This coupling adds complication. To address this, we decouple the process into three steps: (1) recursively bundle the mono-stars in Ψ based on the current mode (see Section 4.3); (b) apply the scheduling strategy to the resulting state DAG; and (c) remove the scheduled mono-stars from Ψ and restore the remaining ones to their original, unbundled form.

This decoupling accommodates interaction between bundling and scheduling, ensuring that the system remains adaptive. Our experiments (Exp-3, Section 6) verify its effectiveness in practice.

Table 2. Graph cleaning benchmark: dataset details.

Graph	Versions	$\lvert V \rvert$	$\lvert E \rvert$	#Labels	#Attributes	#Duplicates Real/Synthetic	#Conflicts Real/Synthetic
BioGRID	4.4.235	4.0M	26.5M	10	38	0.08M / 0.28M	0 / 0.27M
IMDB	23.06–24.06	5.2M	48.2M	4	19	15K / 0.52M	0 / 0.52M
SemScholar	23.10–24.08	0.16B	0.75B	10	20	0.21M / 0	0.06M / 0

Remarks. (1) Our scheduling strategy avoids the need to estimate computational costs. Instead, it adopts scheduling overhead, which depends solely on the materialized sizes of star candidates, and can be assessed by cardinality estimation techniques [68]. (2) Given a set Σ of pending GCRs, the scheduling algorithm runs in $O(|\Sigma|^2)$ time. Since Σ is small in practice, the overhead is negligible and the algorithm can be executed frequently as the state DAG is updated.

6 Experimental Study

This section presents our graph cleaning datasets (Section 6.1) and the evaluation of MiniClean against various baselines (Section 6.2).

6.1 Graph Cleaning Datasets

To test real-world graph cleaning tasks, we constructed three real-life graph datasets, as summarized in Table 2. All graphs are created from open-source datasets from different application domains.

(1) BioGRID [67] is a protein-protein interaction (PPI) network; its vertices denote interactions, proteins and genes, and edges represent their relationships and annotations. Since it integrates data from multiple sources, duplicates are expected, including synonyms.

(2) IMDB [3] is a movie database; its vertices denote movies, actors, directors and genres. Its edges model relationships like casting (actor to movie), directing (director to movie), and genre classification (movie to genre). Errors include duplicates for movies or people.

(3) SemScholar [4] is a billion-scale graph. Its vertices represent papers, authors and venues, and edges denote citations, authorship and affiliations. Its automatic data crawling introduces errors such as duplicate authors/papers, and conflicting affiliations or venues.

Graph construction. We construct each of the datasets as follows.

(1) Graph extraction. BioGRID and IMDB are distributed across multiple tables; we converted the tables to graphs using the standard RDB2Graph method. SemScholar is inherently a graph; we accessed its data through its Web service APIs, ensuring accurate extraction. The resulting graphs are property graphs, where vertices are enriched with key-value pairs representing various attributes.

(2) Real errors. While all three datasets have duplicates, real conflicts are rare in BioGRID and IMDB since both are curated and have been under maintenance for years by dedicated teams. SemScholar has a large number of attribute updates. We manually flagged ones with semantic errors as conflicts, excluding evolving facts.

(3) Synthetic errors. To test various data cleaning methods, we injected synthetic errors into the graphs following [40, 64], controlled by noise ratio $\beta\% = 10\%$ and duplication ratio $\gamma\% = 10\%$. By default, we picked 3% of the data in the original graphs as ground truth Γ_0.

(4) Training/testing split. We split each dataset into training and testing sets for embedded ML models, with the ratio of 80/20.

6.2 Experimental Evaluation

Using the graphs, we evaluated MiniClean versus (1) SOTA single-machine systems for graph cleaning, (2) multi-machine graph cleaning systems to verify the cost-effectiveness; and (3) its own variants to assess the effectiveness of our proposed system optimizations.

Experimental setup. We start with the experimental settings.

Baselines. Single-machine graph baselines include (1) in-memory CoroGraph [94], (2) out-of-core Blaze [50], and (3) hybrid system MiniGraph [96]. We implemented the GCR mining and error detection/correction algorithms of [22, 33]. Since CoroGraph and Blaze do not support property graphs, we treat labels and properties as status variables in these systems. We also tested (4) HyperBlocker [97], a GPU-accelerated ER system, adapted to graph data.

To evaluate the relative efficiency and accuracy against ML-based solutions, we tested (5) Ditto [53] for ER and (6) KGClean [40] for CR, each was trained in their default settings using the designated training sets, *e.g.,* with the use of negative examples.

We also compared with multi-machine baselines: (7) GCRClean, which uses GCRs [22]; and (8) GARClean, which uses more complex GARs [27] with the discovery and deduction algorithms from [21, 27]. Both systems are built on GraphScope [25], a leading multi-machine graph system.

For ablation tests, we evaluated variants of MiniClean: (9) noGPU, which disables GPU acceleration; (10) noCompress, which materializes star candidates without compression; (11) noBundle, which skips mono-star bundling (see Section 4.1); (12) allBundle, which performs bundle greedily, not adaptively (see Section 4.3); (13) noPipelinedPar, which disables pipelined parallelism; (14) noIndPar, which disables independent parallelism (see Section 5); and (15) randSchedule, which executes tasks in a random order.

ML models. MiniClean, GCRClean and GARClean employ the following pre-trained ML models as predicates: Jaccard similarity [47] and SimCSE [38] for textual data similarity checking and clustering, and DeepER [19] for ER. The rule discovery algorithm of GCRClean (resp. GARClean) adopts an LSTM-based model for path (resp. pattern) selection. We trained each model as 2-layer models following [63] in their default configurations, using the training splits.

Testbeds. Our primary testbed for single-machine systems and the ML-based ER/CR solutions is an enterprise-grade GPU server. It is powered by 2× Intel Xeon Gold 6254 @3.10GHz CPUs, each of which has 16 physical cores with hyperthreading. It is also equipped with 64 GB of DDR4 RAM and 4× NVIDIA Tesla V100 32GB GPUs. It has 4× 2TB Samsung 990Pro NVMe SSDs, each of which has an average throughput of 7.4 MB/s and 6.9 GB/s for sequential read and write, respectively. Unless stated otherwise, we used all available CPUs and a single GPU for acceleration.

We deployed multi-machine GCRClean and GARClean on a GraphScope cluster with up to 32 machines, where each is powered by an Intel Xeon @2.2GHz 12-core CPU and 128GB DDR4 RAM.

Configurations. For rule discovery, we set $\sigma = 10^5$, $\delta = 0.9$ and $\alpha = 5$ as the default thresholds for support, confidence and the number of predicates, respectively. We set an upper bound $|V_Q| \leq 10$ for the pattern size, where V_Q denotes the vertex set of pattern Q. Since it takes long to mine a complete set of rules, we curtailed our evaluation to the first 100 rules mined for all methods. For error detection and correction, we used a rule set of size $|\Sigma| = 100$ by default.

We profiled MiniClean to determine appropriate threshold values for the GPU task queue, *i.e.,* τ_1 and τ_2 (Section 4.3). They were tuned based on observed CPU and GPU utilization, ensuring high resource utilization even under transient workload changes. We present the average of each experiment over 5 repetitions. We report results over some datasets; the other results are consistent.

Experimental results. We next report our findings.

Table 3. Error detection efficiency with single-machine systems.

Graph & Metric	BioGRID		IMDB		SemScholar	
	Time (s)	I/O (GB)	Time (s)	I/O (GB)	Time (s)	I/O (GB)
MiniClean	259.6	18.97	325.5	23.82	2993.7	92.54
CoroGraph/Blaze	OOM	OOM	OOM	OOM	OOM	OOM
MiniGraph	65.34×	10.73×	TO	TO	TO	TO
HyperBlocker	11.29×	5.74×	OOM (GPU)	OOM (GPU)	OOM (GPU)	OOM (GPU)

◇ "OOM" denotes out-of-memory, "TO" denotes timeout after 8h.

Table 4. Efficiency and accuracy of MiniClean vs. ML models.

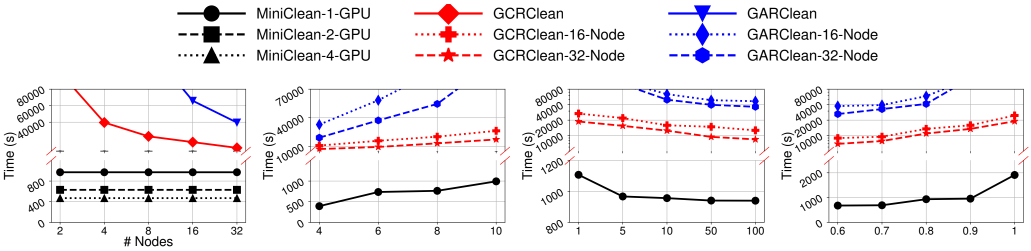
System	BioGRID			IMDB			SemScholar		
	Time	ER-F1(%) all (real)	CR-F1(%) synthetic	Time	ER-F1(%) all (real)	CR-F1(%) synthetic	Time	ER-F1(%) real	CR-F1(%) real
MiniClean	312.7s	98.7 (97.5)	97.2	409.1s	99.9 (94.0)	80.4	4089.1s	94.6	70.6
Ditto	7.7×	91.2 (91.0)	N/A	8.0×	95.6 (90.8)	N/A	4.3×	90.3	N/A
KGClean	11.9×	N/A	54.9	64.2×	N/A	20.0	11.9×	N/A	29.8

Exp-1: Comparison with single-machine systems. We first evaluated MiniClean vs. single-machine systems. We find that rule discovery and error correction are beyond reach of all baselines, which are either out-of-memory (OOM) or timeout (TO) after 8h. In contrast, MiniClean completes these within 4.67 h and 1.14 h on the billion-scale SemScholar, respectively. Thus we only report the efficiency of error detection in Table 3, using rules mined by MiniClean. We also tested MiniClean against ML-based solutions.

Error detection. Table 3 reports the cost of error detection, including time and I/O. We find the following. (1) In-memory CoroGraph and out-of-core Blaze run out-of-memory on all datasets, since both require the intermediate data to fit into the memory. (2) Over small BioGRID, MiniClean beats MiniGraph by 65.34× and generates 90.68% less I/O, benefiting from reduced synchronization and intermediate data size. Moreover, it is 11.29× faster than HyperBlocker, despite both using GPUs. The speedup stems from (a) more efficient use of GPU resources via task bundling and multi-mode parallelism, and (b) reduced I/O via data compression. (3) Over IMDB, MiniGraph cannot finish the computation within 8 h, while MiniClean detects errors in 325.5s. HyperBlocker easily runs out of GPU memory due to the sheer size of intermediate results. (4) No baseline can handle SemScholar, a large graph; in contrast, MiniClean completes the task in 49.9 min. These justify the practicality of a specialized graph cleaning solution, which exhibits reasonable efficiency and cost effectiveness to large real-world graphs.

Accuracy. For error correction over small BioGRID and IMDB, we report the accuracy of MiniClean for ER and CR versus ML-based solutions. We used test sets that contain a small percentage (2.5–3.5%) of ground truth verified by domain experts, which served as starting points for our error correction processes. The accuracy is measured using the F1-score $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$, where precision (resp. recall) is the ratio of detected real errors to all the rule violations (resp. all real errors).

As shown in Table 4, (1) for ER, MiniClean's F1-score is up to 7.5% higher than Ditto in BioGRID and IMDB, GCRs unify logic reasoning and ML, reducing false positives (FPs) by 76.2% and false negatives (FNs) by 78.7% of ML predictions. (2) For CR, MiniClean is 42.3–60.4% more accurate than KGClean. (3) For real ER errors, the F1-score of MiniClean is up to 6.5% higher than Ditto. It reduces FPs (resp. FNs) in large SemScholar by 79.2% (resp. 28.6%). For real CR errors, MiniClean is 40.8% more accurate than KGClean. This is because with GCRs, it can leverage the ground truth to detect errors with more certainty than ML models; moreover, it conducts ER and CR in the same process, which interact with each other. (4) MiniClean is not only 11.9–64.2× faster, but also more accurate than ML models. Moreover, its rule mining time is less than 30% of the model training.



(a) Scalability, BioGRID. (b) Varying $|Q|$, BioGRID. (c) Varying $\sigma(\times 10^4)$, BioGRID. (d) Varying δ, BioGRID.

Fig. 7. Mining efficiency of MiniClean vs. multi-machine systems.

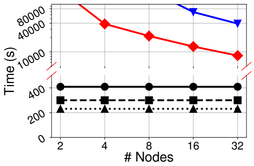
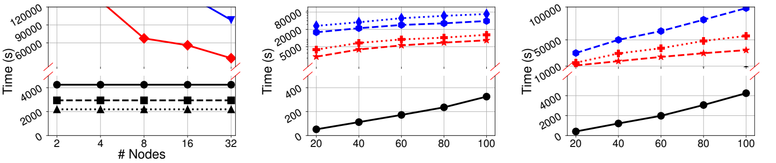


Fig. 8. Detection scalability, IMDB.



(a) Scalability, SemScholar. (b) Varying $|\Sigma|$, IMDB. (c) Varying $|\Sigma|$, SemScholar.

Fig. 9. Correction efficiency of MiniClean vs. multi-machine systems.

This affirms the practicality of MiniClean.

Exp-2: Comparison with multi-machine systems. We next evaluated the efficiency of MiniClean vs. multi-machine systems. By default, we compare MiniClean against 32-node clusters.

Rule discovery: efficiency. As shown in Figure 7a on BioGRID, (1) with a single GPU, MiniClean completes rule discovery in 970.9s. It is 9.22× and 40.78× faster than a 32-node GCRClean and GARClean cluster, respectively. This is because MiniClean can efficiently leverage the massively parallel computation power of GPUs in a shared-memory environment, while multi-machine systems typically incur significant communication costs and suffer from straggler nodes. Moreover, MiniClean costs only 0.66% monetarily based on AWS pricing [1], verifying the cost-effectiveness of single-machine systems. (2) Using more GPUs significantly speeds up MiniClean. When equipped with 2 GPUs (resp. 4 GPUs), it becomes 1.54× (resp. 2.06×) faster, beating a 32-node GCRClean cluster by 14.21× (resp. 19.07×). These further justify the practicality of MiniClean.

Rule discovery: varying $|Q|$. Figure 7b reports the impact of pattern size $|Q|$ on rule mining efficiency over BioGRID. As expected, all systems get slower as $|Q|$ increases. MiniClean is relatively more sensitive to $|Q|$ than GCRClean. When $|Q|$ varies from 4–10, MiniClean is slowed by 2.53×, whereas GCRClean takes 2.40× longer. This is because MiniClean employs task bundling. It is more effective for a smaller $|Q|$, since a common substructure is more likely among a set of smaller patterns, especially in rules generated for mining.

Rule discovery: varying σ and δ. On IMDB, Figures 7c–7d show the impact of varying support and confidence thresholds on the cost of rule discovery. (1) As σ increases, all systems run faster due to the anti-monotonicity of support in GCRs and GARs, which reduces the search space. MiniClean is less sensitive to changes in σ than the multi-machine baselines. (2) Higher δ leads to longer time for rule discovery by all systems, since more GCRs have to be checked to find 100 rules that have high enough confidence. This is evidenced by the 2.0× slowdown of MiniClean when δ varies from 0.9 to 1, since rules with absolute certainties are much rarer. (3) In all settings of σ and δ, MiniClean consistently outperforms the competitors tested, *e.g.*, 32-machine GCRClean and GARClean by 8.99–16.98× and 38.67–52.20×, respectively.

Table 5. Ablation studies for match enumeration. CPU and GPU utilizations are measured in relative terms.

Dataset	Metric	MiniClean	noGPU	noCompress	noBundle	allBundle	noPipelinedPar	noIndPar	randSchedule
BioGRID (1 GPU)	Time	**259.56 s**	36.37×	12.62×	2.47×	1.11×	1.65×	1.33×	1.45×
	Match Size	18.97 GB	2.42×	5.74×	2.42×	**0.64**×	0.79×	1.07×	1.18×
	Mem-GPU	20.87 GB	N/A	5.22×	2.20×	**0.70**×	0.85×	1.05×	4.17×
	CPU Util	1	**+100.9%**	-38.6%	-23.7%	-42.8%	-43.8%	-13.8%	-29.7%
	GPU Util	1	N/A	**+90.9%**	-62.6%	**+58.3%**	-37.1%	-28.7%	-35.1%
IMDB (1 GPU)	Time	**325.54 s**	37.57×	13.06×	2.02×	1.19×	1.77×	1.15×	1.24×
	Match Size	23.82 GB	2.06×	5.71×	2.06×	**0.58**×	0.91×	1.25×	1.21×
	Mem-GPU	27.56 GB	N/A	4.93×	1.78×	**0.62**×	0.96×	1.15×	4.63×
	CPU Util	1	**+117.0%**	-12.1%	-18.8%	-74.6%	-49.4%	-19.4%	-14.5%
	GPU Util	1	N/A	**+120.6%**	-40.7%	**+157.9%**	-49.1%	-39.9%	-31.4%
SemScholar (2 GPUs)	Time	**2993.68 s**	Timeout	Timeout	1.35×	Timeout	1.26×	1.12×	1.20×
	Match Size	92.54 GB	Timeout	Timeout	1.33×	Timeout	0.93×	1.02×	1.08×
	Mem-GPU	103.57 GB	Timeout	Timeout	1.20×	Timeout	0.95×	1.13×	2.32×
	CPU Util	1	Timeout	Timeout	-13.0%	Timeout	-30.2%	-11.0%	-10.7%
	GPU Util	1	Timeout	Timeout	-32.9%	Timeout	-25.3%	-29.1%	-15.9%

Error detection: efficiency. Figure 8 shows the efficiency of error detection over IMDB. (1) MiniClean consistently beats 32-node GCRClean cluster by 20.42×, and GARClean cluster by 95.82×. (2) Like for rule discovery, using more GPUs, MiniClean substantially speeds up error detection. Using 2 and 4 GPUs, it is 1.36× and 1.73× faster, respectively. 4 GPU-MiniClean can detect all errors in 231.0s.

Error correction: efficiency. As shown in Figure 8a over SemScholar, the error correction efficiency has a trend similar to detection. MiniClean is 8.09–37.64× faster than the baselines in a 32-node setting; it gets a speedup by 1.94× using 4 GPUs.

Error correction: varying $|\Sigma|$. We tested the impact of the size $|\Sigma|$ of the rule set on the efficiency of error correction. As shown in Figures 8b–8c, all rule-based systems get slower as $|\Sigma|$ increases, as expected. This said, MiniClean's performance decline is sub-linear; it remains much faster than GCRClean and GARClean, because its task bundling (Section 4.1) reduces redundant computations.

Exp-3: Ablation studies. We conducted ablation studies to evaluate the impact of various optimizations of MiniClean. These help pinpoint the contributions of specific features to the match enumeration efficiency, as shown in Table 5, which details system metrics.

Computing power of GPUs. Compared to noGPU that disables GPU acceleration and uses CPUs only for parallel computation, MiniClean is at least 36.37× faster in match enumeration over different graphs. This indicates that the algorithms of MiniClean make effective use of massive SIMD parallelism of GPUs (Section 2).

Impact of data compression. Succinct tables for GPU join operations accelerate match enumeration by > 12.62× on BioGRID and IMDB. Although uncompressed star candidates in noCompress utilizes GPU better for the uniform layout, they incur > 4.93× data transfers and underutilize the CPU. Without using succinct tables, noCompress cannot handle large SemScholar, since the data structure has a compression ratio around 17.5%, significantly reducing data transfer. These justify the need for data compression.

Impact of bundled processing. MiniClean is 1.35–2.47× faster than noBundle that skips mono-star bundling and matches each star in separate, since it reduces data transfers by up to 2.20× and has a 32.9–62.6% higher GPU utilization. Note that noBundle has a lower CPU utilization since its 1.33–2.42× larger star candidates generate more I/O, which may block CPU operations. Moreover, MiniClean beats allBundle by up to 1.19× on small graphs, and allBundle cannot handle SemScholar, as its greedy bundling incurs heavy and unbalanced GPU workloads (+58.3–157.9%

utilization), especially on large graphs. These justify the need for adaptive star bundling.

Effectiveness of hybrid parallelism. Hybrid parallelism in MiniClean improves resource utilization and helps overlap computation with I/O operations. As shown in Table 5, disabling either pipelined or independent parallelism substantially hampers performance. MiniClean beats noPipelinedPar by 1.26–1.77× because pipelined processing improves CPU utilization by 30.2–49.4% and GPU utilization by 25.3–49.1%. Without independent parallelism, noIndPar is slowed by 1.12–1.33×, due mainly to the underutilized GPU (-28.7–39.9%). These verify the effectiveness of hybrid parallelism.

Effectiveness of scheduling. As compared to randSchedule that processes tasks in a random order, MiniClean is 1.20–1.45× faster. This is because its scheduling strategy minimizes resource idling, as evidenced by the reduced data transfer (2.32–4.63× lower), improved CPU utilization (+10.7–29.7%) and GPU utilization (+15.9–35.1%). These underscore the effectiveness of scheduling strategy.

Summary. We find the following. (1) No single-machine systems are able to support rule discovery and error correction even in small graphs. Moreover, error detection in large graphs is beyond the reach of such systems. In contrast, MiniClean completes rule discovery, error detection and error correction in billion-scale graphs within 4.67 h, 0.83 h and 1.14 h, respectively. For error detection in small graphs, it outperforms MiniGraph by 65.34×, while both in-memory CoroGraph and out-of-core Blaze run out-of-memory. (2) Using a single GPU, MiniClean outperforms a 32-node GCRClean cluster by at least 9.22×, 20.42× and 8.09× in rule discovery, error detection and error correction, respectively. (3) For real-life error detection, MiniClean is up to 6.5% and 40.8% more accurate than ML-based ER and CR models, respectively. (4) The capability of MiniClean for cleaning large graphs comes from a combination of strategies, such as bundled processing, succinct tables, multi-mode parallelism and scheduling, which speed up the performance by up to 2.47×, 13.06×, 1.77× and 1.45× respectively. (5) MiniClean scales well with GPUs. Using 4 GPUs, it is 2.06× faster than using 1 GPU and beats a 32-node cluster by 19.07× in rule discovery.

7 Related Work

We categorize the related work as follows.

Graph cleaning systems. Various rule-based systems have been developed for graph cleaning. They employ different graph dependencies for, *e.g.,* RDFs [10, 18, 20, 44] and property graphs [22, 27, 30–34]. Among these, earlier work [27, 30–32, 34] adopt generic graph patterns and thus require EXPTIME for enumeration. This is impractical for cleaning large graphs, despite recent efforts in parallelizing algorithms for rule discovery [14, 21, 26, 28, 36, 37, 66, 70]. In contrast, MiniClean uses GCRs [22], which may embed ML models as predicates and allow deep ER and CR via chase. Moreover, it is in PTIME to detect and correct errors with GCRs, since they employ a dual pattern to specify characteristic features of disconnected entities. Efficient multi-machine parallel algorithms have been developed [22, 33] for GCR mining, error detection and correction.

As opposed to multi-machine solutions, MiniClean is the first single-machine system capable of cleaning billion-scale graphs. Using a single machine with GPUs, it addresses unique challenges not previously encountered, such as the limited computation and memory capacity, and the heterogeneous parallel architecture with CPU and GPU. To tackle these, it develops a two-stage workflow that accelerates heavy-duty tasks on GPUs, departing from prior algorithms for GCRs [22, 33]. Moreover, it proposes a combination of optimization strategies that are tailored to a single machine with limited resources, providing a cost-effective solution.

Besides graph dependencies, ML-based graph cleaning adopts graph embeddings [40, 91], edit histories [81], language models [13], or user annotations [12] to catch and fix errors. However, they are less accurate (see Section 6), because an ML model (a) targets either ER or CR, but not both in a unified recursive process, while the two interact with each other for overall data quality [33], and (b) cannot reliably leverage ground truth to make certain fixes [32].

Single-machine graph systems. General-purpose graph analytics systems aim to support custom graph algorithms, simplifying user programming with a high-level graph abstraction. These include in-memory ([41, 56, 65, 76, 90, 92, 94]), semi-external ([50, 54, 93]), and out-of-core [9, 51, 57, 72, 83, 95, 96] systems. While they support graph cleaning in theory, most are unable to manage the excessive intermediate results of match enumeration. The only exception is MiniGraph [96]; however, it can only detect errors in small graphs, and requires partitioning the graph into small subgraphs, incurring substantial overhead for among-subgraph synchronization.

Closer to this work are specialized graph mining systems. To find matches of a general pattern in a graph, CPU-based solutions [17, 48, 60, 61, 75] require all auxiliary data to fit into the main memory, and cannot scale to large graphs. Recent GPU-accelerated systems [16, 43, 45, 80, 86–89] offload pattern matching jobs entirely to the SIMD architecture of GPUs, by transforming the computation into *e.g.,* set operations. Such techniques are limited to counting queries only and are not applicable to match enumeration.

In contrast, MiniClean is specialized for graph cleaning with GCRs, for tasks distinct from simple graph analytics queries. As opposed to existing graph mining systems, it (a) is tailored for dual-star patterns rather than general patterns, enabling unique optimizations, *e.g.,* star bundling and candidate compression; (b) manages large intermediate results that exceed the memory capacity by treating SSDs as extensions; (c) goes beyond match enumeration, supporting more complex rule validation and error correction; and (d) balances workload between the CPU and GPU adaptively to maximize their synergy, rather than allocating tasks statically.

Conditional tables. Designed to model the uncertain data [46], a conditional table [6, 8, 42, 46, 79] is a relation where each tuple is annotated with a condition over some variables. MiniClean extends the concept to represent the materialized bundled star candidates. Our conditional succinct table leverages the filtering capability of annotations, indexing the pattern-level conditions alongside the predicate-level conditions. Moreover, it integrates the nested data representation [2, 62, 78, 85] for a compressed data layout. Capitalizing on the unique properties of GCR, *e.g.,* dual-star patterns with center/leaf-only predicates, it requires neither additional meta-information [2, 62] nor complex unnesting operations [78, 85].

8 Conclusion

To the best of our knowledge, MiniClean is the first system for graph cleaning with a single machine. It supports rule discovery, error detection, and error correction, while no SOTA single-machine systems can mine rules or correct errors even in small graphs. To make a single-machine system capable of cleaning large graphs, MiniClean develops and deploys a combination of techniques, such as (a) a pipeline of CPU and GPU operations to their synergy, (b) a bundling strategy to reduce redundant matching, (c) conditional succinct tables to reduce memory footprint, (d) a multi-mode parallel model to maximize resource utilization, and (e) task scheduling to overlap I/O and CPU/GPU operations. None of the techniques alone suffices for graph cleaning with a single machine. Our empirical study has verified that MiniClean is promising in practice.

Topics for future work include (a) ML-aided scheduling to optimize parallel processing, and (b) the capacity of a single machine for other complex computational problems such as fraud detection.

Acknowledgments

This work is supported by China NSFC under Grant No. 62232004.

References

[1] 2024. Amazon AWS Pricing. *https://aws.amazon.com/pricing.*

[2] 2024. Apache Parquet. *https://parquet.apache.org/.*

[3] 2024. IMDB graph dataset. *https://www.cs.toronto.edu/ oktie/linkedmdb/linkedmdb-latest-dump.zip.*

[4] 2024. SemanticScholar academic graph. *https://www.semanticscholar.org/.*

[5] 2025. Online full version. https://shuhaoliu.github.io/assets/papers/wenchao-sigmod25-miniclean-full.pdf.

[6] Serge Abiteboul and Oliver M Duschka. 1998. Complexity of answering queries using materialized views. In *PODS*. 254–263.

[7] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.

[8] Serge Abiteboul, Paris Kanellakis, and Gösta Grahne. 1991. On the representation and querying of sets of possible worlds. *Theoretical Computer Science* 78, 1 (1991), 159–187.

[9] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing out all the value of loaded data: An out-of-core graph processing system with reduced disk I/O. In *USENIX ATC*. 125–137.

[10] Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. 2010. Constraints in RDF. In *SDKB*.

[11] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. 1999. Consistent Query Answers in Inconsistent Databases. In *PODS*. 68–79.

[12] Abdallah Arioua and Angela Bonifati. 2018. User-guided Repairing of Inconsistent Knowledge Bases. In *EDBT*.

[13] Hiba Arnaout, Trung-Kien Tran, Daria Stepanova, Mohamed Hassan Gad-Elrab, Simon Razniewski, and Gerhard Weikum. 2022. Utilizing language model probes for knowledge graph repair. In *Wiki Workshop 2022*.

[14] Lihan Chen, Sihang Jiang, Jingping Liu, Chao Wang, Sheng Zhang, Chenhao Xie, Jiaqing Liang, Yanghua Xiao, and Rui Song. 2022. Rule mining over knowledge graphs via reinforcement learning. *Knowl. Based Syst.* 242 (2022).

[15] Mao Chen, Wen Chen, and Yongqi Zhu. 2021. A Novel Big Data Cleaning Algorithm Based On Edge Computing In Industrial Internet of Things. In *International Conference on Advances in Computer Technology, Information Science and Communication (CTISC)*. 194–198.

[16] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *USENIX OSDI*.

[17] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, Loc Hoang, and Keshav Pingali. 2021. Sandslash: A two-level framework for efficient graph pattern mining. In *International Conference on Supercomputing*.

[18] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of Constraints in RDFS. In *AMW*.

[19] Muhammad Ebraheem, Saravanan Thirumuruganathan, Shafiq R. Joty, Mourad Ouzzani, and Nan Tang. 2018. Distributed Representations of Tuples for Entity Resolution. *PVLDB* 11, 11 (2018), 1454–1467.

[20] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. 2015. Keys for graphs. *PVLDB* 8, 12 (2015), 1590–1601.

[21] Wenfei Fan, Wenzhi Fu, Ruochun Jin, Ping Lu, and Chao Tian. 2022. Discovering Association Rules from Big Graphs. *PVLDB* 15, 7 (2022), 1479–1492.

[22] Wenfei Fan, Wenzhi Fu, Ruochin Jin, Ping Lu, and Chao Tian. 2023. Making It Tractable to Catch Duplicates and Conflicts in Graphs. *Proc. ACM Manag. Data* 1, 1 (2023), 86:1–86:28.

[23] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.

[24] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional Functional Dependencies for Capturing Data Inconsistencies. *ACM Trans. Database Syst.* 33, 1 (2008), 6:1–6:48.

[25] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, Youyang Yao, Qiang Yin, Wenyuan Yu, Kai Zeng, Kun Zhao, Jingren Zhou, Diwen Zhu, and Rong Zhu. 2021. GraphScope: A Unified Engine For Big Graph Processing. *PVLDB* 14, 12 (2021), 2879–2892.

[26] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2020. Discovering Graph Functional Dependencies. *ACM Trans. Database Syst.* 45, 3 (2020), 15:1–15:42.

[27] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Capturing Associations in Graphs. *PVLDB* 13, 11 (2020), 1863–1876.

[28] Wenfei Fan, Ruochun Jin, Ping Lu, Chao Tian, and Ruiqi Xu. 2022. Towards Event Prediction in Temporal Graphs. *PVLDB* 15, 9 (2022), 1861–1874.

[29] Wenfei Fan and Shuhao Liu. 2024. Fishing Fort: A System for Graph Analytics with ML Prediction and Logic Deduction. In *The Provenance of Elegance in Computation - Essays Dedicated to Val Tannen*, Vol. 119. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 6:1–6:18.

[30] Wenfei Fan, Xueli Liu, Ping Lu, and Chao Tian. 2020. Catching Numeric Inconsistencies in Graphs. *ACM Trans. Database Syst.* 45, 2 (2020), 9:1–9:47.

[31] Wenfei Fan and Ping Lu. 2019. Dependencies for Graphs. *ACM Trans. Database Syst.* 44, 2 (2019), 5:1–5:40.

[32] Wenfei Fan, Ping Lu, Chao Tian, and Jingren Zhou. 2019. Deducing Certain Fixes to Graphs. *PVLDB* 12, 7 (2019), 752–765.

[33] Wenfei Fan, Kehan Pang, Ping Lu, and Chao Tian. 2025. Making It Tractable to to Detect and Correct Errors in Graphs.

ACM Trans. on Database Systems (2025).

[34] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *SIGMOD*. ACM, 1843–1857.

[35] Wenfei Fan, Wenyuan Yu, Jingbo Xu, Jingren Zhou, XiaoJian Luo, Qiang Yin, Ping Lu, Yang Cao, and Ruiqi Xu. 2018. Parallelizing Sequential Graph Computations. *ACM Trans. Database Syst.* 43, 18 (2018).

[36] Luis Antonio Galárraga, Christina Teflioudi, Katja Hose, and Fabian Suchanek. 2013. AMIE: Association rule mining under incomplete evidence in ontological knowledge bases. In *WWW*.

[37] Kun Gao, Katsumi Inoue, Yongzhi Cao, and Hanpin Wang. 2022. Learning First-Order Rules with Differentiable Logic Program Semantics. *CoRR* (2022). https://doi.org/10.48550/arXiv.2204.13570

[38] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *EMNLP*. 6894–6910.

[39] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company.

[40] Congcong Ge, Yunjun Gao, Honghui Weng, Chong Zhang, Xiaoye Miao, and Baihua Zheng. 2020. KGClean: An Embedding Powered Knowledge Graph Cleaning Framework. *CoRR* abs/2004.14478 (2020).

[41] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *PVLDB* 13, 8 (2020), 1304–1318.

[42] Gösta Grahne and Alberto O Mendelzon. 1999. Tableau techniques for querying information sources through global schemas. In *ICDT*. Springer, 332–347.

[43] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-accelerated subgraph enumeration on partitioned graphs. In *SIGMOD*.

[44] Jelle Hellings, Marc Gyssens, Jan Paredaens, and Yuqing Wu. 2016. Implication and axiomatization of functional and constant constraints. *Ann. Math. Artif. Intell.* 76, 3-4 (2016), 251–279.

[45] Lin Hu, Lei Zou, and M Tamer Özsu. 2023. GAMMA: A Graph Pattern Mining Framework for Large Graphs on GPU. In *ICDE*. IEEE, 273–286.

[46] Tomasz Imieliński and Witold Lipski Jr. 1984. Incomplete information in relational databases. *JACM* 31, 4 (1984), 761–791.

[47] Paul Jaccard. 1901. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bull Soc Vaudoise Sci Nat* 37 (1901), 547–579.

[48] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating graph mining systems with subgraph morphing. In *EuroSys*. 162–181.

[49] Tarun Kathuria and S Sudarshan. 2017. Efficient and provable multi-query optimization. In *PODS*. 53–67.

[50] Juno Kim and Steven Swanson. 2022. Blaze: Fast graph processing on fast SSDs. In *SC*. 1–15.

[51] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-scale graph computation on just a PC. In *OSDI*.

[52] Zhuohang Lai, Xibo Sun, Qiong Luo, and Xiaolong Xie. 2022. Accelerating multi-way joins on the GPU. *The VLDB Journal* (2022), 1–25.

[53] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep entity matching with pre-trained language models. *arXiv preprint arXiv:2004.00584* (2020).

[54] Hang Liu and H Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *FAST*. 285–300.

[55] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *Proceedings of the 2022 International Conference on Management of Data*. 1017–1032.

[56] Lingxiao Ma, Zhi Yang, Han Chen, Jilong Xue, and Yafei Dai. 2017. Garaph: Efficient GPU-accelerated Graph Processing on a Single Machine with Balanced Replication. In *USENIX ATC*.

[57] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woon-Hak Kang, Mohan Kumar, and Taesoo Kim. 2017. Mosaic: Processing a Trillion-Edge Graph on a Single Machine. In *EuroSys*. ACM.

[58] Vasilis Mageirakos, Riccardo Mancini, Srinivas Karthik, Bikash Chandra, and Anastasia Ailamaki. 2022. Efficient gpu-accelerated join optimization for complex queries. In *ICDE*. IEEE, 3190–3193.

[59] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A system for large-scale graph processing. In *SIGMOD*. 135–146.

[60] Daniel Mawhirter, Sam Reinehr, Connor Holmes, Tongping Liu, and Bo Wu. 2021. GraphZero: A High-Performance Subgraph Matching System. *SIGOPS Operation Systems Review* 55, 1 (June 2021), 21–37.

[61] Daniel Mawhirter and Bo Wu. 2019. Automine: Harmonizing high-level abstraction and high performance for graph mining. In *SOSP*.

[62] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive analysis of web-scale datasets. *PVLDB* 3, 1-2 (2010), 330–339.

[63] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. 2017. Regularizing and optimizing LSTM language models.

arXiv preprint arXiv:1708.02182 (2017).

[64] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD*. 19–34.

[65] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A lightweight infrastructure for graph analytics. In *SOSP*.

[66] Stefano Ortona, Venkata Vamsikrishna Meduri, and Paolo Papotti. 2018. Robust Discovery of Positive and Negative Rules in Knowledge Bases. In *ICDE*. 1168–1179.

[67] Rose Oughtred, Jennifer Rust, Christie Chang, Bobby-Joe Breitkreutz, Chris Stark, Andrew Willems, Lorrie Boucher, Genie Leung, Nadine Kolas, Frederick Zhang, Sonam Dolma, Jasmin Coulombe-Huntington, Andrew Chatr-Aryamontri, Kara Dolinski, and Mike Tyers. 2021. The BioGRID database: A comprehensive biomedical resource of curated protein, genetic, and chemical interactions. *Protein Science* 30, 1 (2021), 187–200.

[68] Yeonsu Park, Seongyun Ko, Sourav S Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A framework for performance benchmarking of cardinality estimation techniques for subgraph matching. In *SIGMOD*. 1099–1114.

[69] Johns Paul, hengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. *Proc. ACM on Manag. Data*, 1413–1425.

[70] Meng Qu, Junkun Chen, Louis-Pascal A. C. Xhonneux, Yoshua Bengio, and Jian Tang. 2021. RNNLogic: Learning Logic Rules for Reasoning on Knowledge Graphs. In *ICLR*.

[71] Xuguang Ren and Junhu Wang. 2016. Multi-query optimization for subgraph isomorphism search. *PVLDB* 10, 3 (2016), 121–132.

[72] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP*. ACM, 472–488.

[73] Prasan Roy and S. Sudarshan. 2018. Multi-query Optimization. In *Encyclopedia of Database Systems, Second Edition*. Springer, 2425–2429.

[74] Ran Rui, Hao Li, and Yi-Cheng Tu. 2020. Efficient join algorithms for large database tables in a multi-GPU environment. In *PVLDB*, Vol. 14. NIH Public Access, 708.

[75] Tianhui Shi, Jidong Zhai, Haojie Wang, Qiqian Chen, Mingshu Zhai, Zixu Hao, Haoyu Yang, and Wenguang Chen. 2023. GraphSet: High Performance Graph Mining through Equivalent Set Transformations. In *SC*.

[76] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *SIGPLAN*. 135–146.

[77] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-conscious hash-joins on gpus. In *ICDE*. IEEE, 698–709.

[78] Jaclyn Smith, Michael Benedikt, Milos Nikolic, and Amir Shaikhha. 2020. Scalable Querying of Nested Data. *PVLDB* 14, 3 (2020), 445–457.

[79] Dan Suciu, Dan Olteanu, Christopher Ré, and Christopher Ellis Koch. 2011. Probabilistic Databases. *Synthesis Lectures on Data Management* 3 (2011), 1–180.

[80] Xibo Sun and Qiong Luo. 2023. Efficient GPU-Accelerated Subgraph Matching. *Proc. ACM Manag. Data* 1, 2 (2023), 1–26.

[81] Thomas Pellissier Tanon and Fabian M. Suchanek. 2021. Neural Knowledge Base Repairs. In *ESWC*.

[82] Lasse Thostrup, Gloria Doci, Nils Boeschen, Manisha Luthra, and Carsten Binnig. 2023. Distributed GPU Joins on Fast RDMA-capable Networks. *Proc. ACM on Manag. Data* 1, 1 (2023), 1–26.

[83] Keval Vora, Guoqing Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*. 507–522.

[84] Tian Wang, Haoxiong Ke, Xi Zheng, Kun Wang, Arun Kumar Sangaiah, and Anfeng Liu. 2020. Big Data Cleaning Based on Mobile Edge Computing in Industrial Sensor-Cloud. *IEEE Transactions on Industrial Informatics* 16, 2 (2020), 1321–1329.

[85] Zhiyi Wang and Shimin Chen. 2017. Exploiting common patterns for tree-structured data. In *SIGMOD*. 883–896.

[86] Yihua Wei and Peng Jiang. 2022. STMatch: Accelerating graph pattern matching on GPU with stack-based loop optimizations. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–13.

[87] Lizhi Xiang, Arif Khan, Edoardo Serra, Mahantesh Halappanavar, and Aravind Sukumaran-Rajam. 2021. cuTS: Scaling subgraph isomorphism on distributed multi-GPU systems using trie based data structure. In *International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[88] Li Zeng, Lei Zou, and M Tamer Özsu. 2022. SGSI–A Scalable GPU-friendly Subgraph Isomorphism Algorithm. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[89] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *ICDE*.

IEEE, 1249–1260.

[90] Kaiyuan Zhang, Rong Chen, and Haibo Chen. 2015. NUMA-aware graph-structured analytics. In *PPoPP*. 183–193.

[91] Qinggang Zhang, Junnan Dong, Keyu Duan, Xiao Huang, Yezi Liu, and Linchuan Xu. 2022. Contrastive Knowledge Graph Error Detection. In *CIKM*.

[92] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. 2018. GraphIt: A high-performance graph DSL. *OOPSLA* 2 (2018), 1–30.

[93] Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing Billion-Node graphs on an array of commodity SSDs. In *FAST*. 45–58.

[94] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2023. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *PVLDB* 17, 4 (2023), 891–903.

[95] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*. 375–386.

[96] Xiaoke Zhu, Yang Liu, Shuhao Liu, and Wenfei Fan. 2023. MiniGraph: Querying Big Graphs with a Single Machine. *PVLDB* 16, 9 (2023), 2172–2185.

[97] Xiaoke Zhu, Min Xie, Ting Deng, and Qi Zhang. 2025. HyperBlocker: Accelerating Rule-based Blocking in Entity Resolution using GPUs. *PVLDB* 18, 2 (2025), 308–321.

A Proof of Theorem 1

Problem statement. We study the decision problem of the scheduling problem, which is denoted by DSP and stated as follows.

Input. (1) A set $A = \{a_1, a_2, \ldots, a_{2n}\}$ of star matching tasks on the CPU, where task a_j has a cost $C_A(a_j, p)$ using p cores ($j \in [1, 2n]$). (2) A set $B = \{b_1, \ldots, b_n\}$ of cross-star enumeration tasks on the GPU, where each task b_i depends on two tasks a_{2i-1} and a_{2i} in A and has a cost function $C_B(b_i)$ for $i \in [1, n]$. (3) An m-core machine whose (a) host memory and GPU memory capacity are η_1 and η_2, respectively; (b) disk read bandwidth is γ_1; and (c) host–GPU data transfer bandwidth is γ_2. (4) A deadline D for the completion of all tasks.

Question. Does there exist a valid schedule $\mathcal{S} = \langle \bar{\alpha}, \bar{\beta}, \bar{p}(t) \rangle$, whose makespan is bounded by D, *i.e.,* $\max_{i \in [1,n]}\{\beta_i\} \leq D$?

Proof of the NP-completeness. We verify the upper bound of DSP by developing an NP algorithm. The algorithm works as follows: (1) first guess a schedule $\mathcal{S} = (\bar{\alpha}, \bar{\beta}, \bar{p}(t))$; and (2) then check whether the makespan $\max_{i \in [1,n]}\{\beta_i\} \leq D$; if so, return `true`. The algorithm is clearly in NP since verification in step (2) can be done in polynomial time (PTIME); hence so is DSP.

We verify the lower bound of DSP by reduction from the partition problem, which is known to be NP-complete (cf. [39]). The partition problem is to decide, given a finite set Λ of positive integers, whether there exists a subset $\Lambda' \subseteq \Lambda$, such that Λ' and its complement $\bar{\Lambda'} = \Lambda \setminus \Lambda'$ have an equal sum, *i.e.,* $\sum_{x_i \in \Lambda'} x_i = \sum_{y_j \in \bar{\Lambda'}} y$.

Given a set of $2n$ positive integers $\Lambda = \{x_1, x_2, \ldots, x_{2n}\}$, we construct a schedule $\mathcal{S}$ for (1) a set $A = \{a_1, a_2, \ldots, a_{2n}\}$ of star matching tasks and their cost function $C_A(a_j, p)$; (2) a set $B = \{b_1, \ldots, b_n\}$ of cross-star enumeration task and their cost function $C_B(b_i, p)$; (3) a positive integer m, four positive numbers η_1, η_2, γ_1 and γ_2; and (4) a deadline D to ensure the following property: Λ can be partitioned into Λ and $\bar{\Lambda'}$ of equal sum if and only if $\max_{i \in [1,n]}\{\beta_i\} \leq D$. Without loss of generality, we assume that $\Lambda' = \{x_1, x_2, \ldots, x_k\}$ and $\bar{\Lambda'} = \{x_{k+1}, x_{k+2}, \ldots, x_{2n}\}$.

The construction details are presented as follows.

(1) The cost function C_A is defined such that for all $j \in [1, 2n]$ and any positive integer p, $C_A(a_j, p) = x_i$. More specifically, (a) graph G consists of a collection $\{P_1, P_2, \ldots, P_{2n}\}$ of disconnected paths. The vertices on path P_j are all labeled L_j (for $j \in [1, 2n]$), except that the sink vertex is labeled L_0. (b) For each $j \in [1, 2n]$, the mono-star pattern $Q_j(X_j)$ has a single path of length $x_j - |X_j|$. The leaf node of Q_j has label L_0, while the rest of the pattern nodes have label L_j.

Therefore, finding matches of mono-star pattern Q_j in the constructed graph G can be completed in $x_j - |X_j|$ steps, and checking constant predicates in X_j requires $|X_j|$ steps. One can easily verify that the total cost of finding star candidates of $Q_j(X_j)$ is x_j, which cannot be accelerated by using more CPU cores.

(2) The cost function C_B is defined such that for all $i \in [1, n]$ and any positive integer p, $C_B(b_i, p) = 1$. This is because the cross-star enumeration task b_i takes as input the result of tasks a_{2i-1} and a_{2i}, each of which has only one star candidate. Task b_i only needs to check the Cartesian product of the two star candidates, a total of one pair of matches. It is obvious to see that the cost of b_i is 1, and it cannot be accelerated by using more GPU threads, either.

(3) We set $m = 2$, $\eta_1 = 2n + |G| + 10$, $\eta_2 = 2n$ and $\gamma_1 = \gamma_2 = 2$. Intuitively, the machine has two CPU cores (*e.g.,* Core 0 and Core 1). It has host memory sufficient to store graph G and all star candidates of star matching tasks $\{a_1, a_2, \ldots, a_{2n}\}$. Its GPU memory is also large enough to store all star candidates. Moreover, its host–GPU data transfer bandwidth is 2, such that the star candidates

of each star matching task can be transferred to the GPU in 0.5 time unit.

(4) Schedule $\mathcal{S}$ assigns a single thread to execute all star matching tasks on the CPU. For the completion times $\bar{\alpha}$ of star matching tasks, $\mathcal{S}$ sets the start time $\alpha_1 = x_1$, $\alpha_{k+1} = x_{k+1}$, and $\alpha_j = \alpha_{j-1} + x_j$ for all $1 < j \leq k$ and $k + 1 < j \leq 2n$. In other words, tasks $a_1, a_2, \ldots, a_k$ are executed on Core 0 sequentially without leaving any gap, where a_1 starts at time 0; tasks $a_{k+1}, a_{k+2}, \ldots, a_{2n}$ are executed on Core 1 sequentially without leaving any gap, where a_{k+1} starts at time 0.

For the completion times $\bar{\beta}$ of cross-star enumeration tasks, schedule $\mathcal{S}$ assigns $\beta_i = \max\{\alpha_{2i-1}, \alpha_{2i}\} + 2$ for all $i \in [1, n]$. Intuitively, each enumeration task b_i starts as soon as the star candidates of a_{2i-1} and a_{2i} are both computed and transferred to the GPU.

(4) Finally, we set deadline $D = \frac{1}{2} \sum_{x \in \Lambda} x + 2$. Note that we assume *w.l.o.g.* that B is a positive integer and thus $\sum_{x \in \Lambda} x = 2D - 4$.

We next verify the correctness of our reduction.

($\Rightarrow$) Suppose that Λ can be partitioned into Λ' and $\bar{\Lambda}'$ of equal sum. Then $\sum_{i \in [1,k]} x_i = \sum_{i \in [k+1,2n]} x_i = D - 2$. The makespan of schedule $\mathcal{S}$ can be calculated as follows, meeting the deadline D:

$$\begin{aligned} \max_{i \in [1,n]} \{\beta_i\} &= \max_{j \in [1,2n]} \{\alpha_j\} + 2 \\ &= \max\{\alpha_k, \alpha_{2n}\} + 2 \\ &= \max\{D - 2, D - 2\} + 2 = D. \end{aligned}$$

($\Leftarrow$) Suppose that the makespan of schedule $\mathcal{S}$ can meet deadline D, *i.e.*, $\max_{i \in [1,n]}\{\beta_i\} = \max_{j \in [1,2n]}\{\alpha_j\} + 2 \leq D$. We have that $\max\{\alpha_k, \alpha_{2n}\} \leq D - 2$. As a consequence, one can deduce:

$$\Sigma_{j=1}^{k} x_j \leq D - 2, \tag{2}$$

$$\Sigma_{j=k+1}^{2n} x_j \leq D - 2. \tag{3}$$

Here we construct $\Lambda' = \{x_1, x_2, \ldots, a_k\}$ and its complement $\bar{\Lambda}' = \{x_{k+1}, x_{k+2}, \ldots, x_{2n}\}$. From Equations 2 and 3, we have that $\Sigma_{x \in \Lambda'} x \leq D - 2$ and $\Sigma_{x \in \bar{\Lambda}'} x \leq D - 2$, respectively.

Provided that D is defined such that $\sum_{x \in \Lambda} x = 2D - 4$, we have $\Sigma_{x \in \Lambda'} x = \Sigma_{x \in \bar{\Lambda}'} x = D - 2$. In other words, Λ' and its complement $\bar{\Lambda}'$ have an equal sum $D - 2$. This completes the reduction. □

B Reliability of GCR-based Graph Cleaning

It has been shown that rules Σ_c learned from a sample G_s of graph G provide recall and support guarantees [21], *i.e.*, at least $\gamma\%$ of rules on G can be learned from G_s, and each rule learned from G_s can be applied for at least σ^* times on G. Here γ and σ^* are determined by sampling ratio $\frac{|G_s|}{|G|}$. That is, each rule discovered from G_s makes sense on G, and most rules on G can be learned from G_s. For noisy real-life graphs G, we set the confidence threshold δ high enough such that the mined rules are reliable. We also require domain expert verification to further validate rule correctness.

C GPU Utilization in MiniClean vs. Prior Work

GPUs have been used to accelerate computation-intensive pattern matching [16, 43, 45, 80, 86–89]. Despite moderate speedup over CPU-based systems, they underutilize the computation power of GPUs due to warp divergence, where irregular GPU memory access limits SIMD parallelism. To tackle this, recent work employs GPU-accelerated join operations for pattern matching, proposing hardware-conscious optimizations [55, 74, 77], load balancing strategies [52, 69, 82], and join order optimizations [58, 97]. However, these systems incur data transfers of large intermediate results and degrade performance on large graphs.

MiniClean employs a unique pipelined CPU–GPU architecture specialized for graph cleaning with GCRs, which involves both pattern matching and rule validation. As opposed to offloading most computation to GPUs, it decouples star matching and cross-star enumeration, such that both CPUs and GPUs can be utilized effectively. Moreover, the pipelined architecture enables MiniClean to dynamically balance I/O, CPU, data transfers, and GPU operations to maximize their synergy. Together with other tailored optimization techniques, it improves performance by adapting to both computation-heavy and intermediate data-heavy workloads.

D Details of Scheduling Strategy

State DAG construction. At any time point t, we model the state of computation and the dependency of tasks in MiniClean as a directed acyclic graph (DAG) S_t. Each node in S_t represents a *data status*, *i.e.,* a data–placement pair where the data can be either input graph G, buckets of star candidates, or the final enumerated matches for a GCR, and the placement can be in the SSD, the host or GPU memory. A node has two states to indicate whether the data is materialized at the corresponding placement at time t. An edge denotes a dependency between data statuses, whose weight is the cost of materializing the downstream data status given the upstream data status. Apparently, all GCR nodes are the source nodes in DAG S_t because they have no dependent data statuses; all sink nodes must be in the materialized state.

Costs and memory usage modeling. Intuitively, a state DAG $S(t)$ captures the system snapshot at time t, modeling the progress of match enumeration computation and intermediate data placement. Given $S(t)$, we define cost and memory usage functions, useful for quantifying the effectiveness and validity of scheduling decisions.

(1) The I/O cost C_{IO} (resp. data transfer cost C_{trans}) is calculated by summing up the weights of all edges that connect an un-materialized host-memory (resp. GPU-memory) node to its upstream on-SSD (resp. host-memory) node. We say that the bottleneck cost $\hat{C}$ is the maximum of C_{IO} and C_{trans}.

(2) The host computation C_{cpu} is measured by the weight sum of all edges that connect an unmaterialized host node to graph G.

(3) The host and GPU memory usage (M_{cpu} and M_{gpu}) at time t is the total data sizes of all nodes materialized at host and GPU.

(4) The remaining cost $C_{\text{GCR}}(S(t), \varphi)$ for enumerating matches of a GCR φ is estimated by traversing the DAG from the source node of φ to the sinks, and calculating the total weight along the critical path.

E Limitation Analysis

Our ablation studies also suggest that certain optimizations may not be as effective over extremely skewed graphs. SemScholar is a representative, where task bundling (resp. independent parallelism) only contributes to a 1.35× (resp. 1.12×) speedup. This is because the skewed citation relationships in SemScholar yield a large volume of candidates during enumeration, overloading the GPU. Thus, the adaptive bundling strategy largely degrades to noBundle to avoid shifting more workload, and independent parallelism becomes less effective. Nevertheless, MiniClean can still handle such large-scale graphs, beyond the capabilities of prior systems.

Received October 2024; revised January 2025; accepted February 2025