HyperBlocker: Accelerating Rule-based Blocking in Entity Resolution using GPUs

Xiaoke Zhu
Beihang University, China
zhuxk@buaa.edu.cn

Min Xie*
Shenzhen Institute of Computing Sciences, China
xiemin@sics.ac.cn

Ting Deng
Beihang University, China
dengting@act.buaa.edu.cn

Qi Zhang
Meta Platforms, USA
qizhang@meta.com

ABSTRACT

This paper studies rule-based blocking in Entity Resolution (ER). We propose HyperBlocker, a GPU-accelerated system for blocking in ER. As opposed to previous blocking algorithms and parallel blocking solvers, HyperBlocker employs a pipelined architecture to overlap data transfer and GPU operations, and improve parallelism by effectively collaborating GPUs. It generates a data-aware and rule-aware execution plan on CPUs, for specifying how rules are evaluated, and develops a number of hardware-aware optimizations to achieve massive parallelism across multiple GPUs. Using real-life and synthetic datasets, we show that HyperBlocker is at least 6.8× and 9.1× faster than prior CPU-powered distributed systems and GPU-based ER solvers, respectively. Better still, by combining HyperBlocker with the state-of-the-art ER matcher, we speed up the overall ER process by at least 30% with comparable accuracy.

PVLDB Artifact Availability:
The source code, data, or other artifacts have been made available at https://github.com/SICS-Fundamental-Research-Center/HyperBlocker.

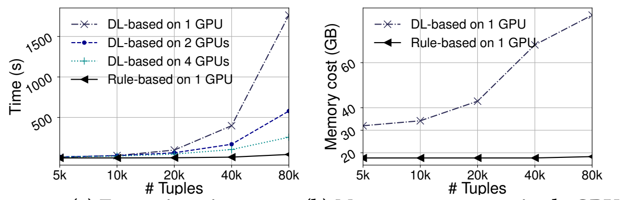
1 INTRODUCTION

Entity resolution (ER), also known as record linkage, data deduplication, merge/purge and record matching, is to identify tuples that refer to the same real-world entity. It is a routine operation in many data cleaning and integration tasks, such as detecting duplicate commodities [33] and finding duplicate customers [22].

Recently, with the rising popularity of deep learning (DL) models, research efforts have been spent on applying DL techniques to ER. Although these DL-based approaches have shown impressive accuracy, they also come with high training/inference costs, due to the large number of parameters. Despite the effort to reduce parameters, the growth in the size of DL models is still an inevitable trend, leading to the increasing time for making matching decisions.

In the worst case, ER solutions have to spend quadratic time to examine all pairs of tuples. As reported by Thomson Reuters, an ER project can take 3-6 months, mainly due to the scale of data [20]. To accelerate, most ER solutions divide ER into two phases: (a) a blocking phase, where a blocker discards unqualified pairs that are guaranteed to refer to distinct entities, and (b) a matching phase, where a matcher compares the remaining pairs to finally decide whether they are *matched*, *i.e.,* refer to the same entity. The blocking phase is particularly useful when dealing with large data and "is the crucial part of ER with respect to time efficiency and scalability" [64].

To cope with the volume of big data, considerable research has been conducted on blocking techniques. As surveyed in [49, 64], we can divide blocking methods into *rule-based* [20, 34, 37, 42, 63] or *DL-based* [24, 38, 73, 76], both have their strengths and limitations.



(a) Execution time **(b) Memory cost on a single GPU**

Figure 1: DL-based blocking vs. rule-based blocking

DL-based blocking methods typically utilize pre-trained DL models to generate embeddings for tuples and discard tuple pairs with similarity scores below a predefined threshold. While DL-based blocking can enhance ER by parallelizing computation and leveraging GPU acceleration [39], it often comes with long runtime and high memory costs. To justify this, we conducted a detailed analysis on DeepBlocker [73], the state-of-the-art (SOTA) DL-based blocker in Figure 1. We picked a rule-based blocker (a prototype of our method) with comparable accuracy with DeepBlocker and compared it with DeepBlocker in terms of runtime and memory. The evaluation was conducted on a machine equipped with V100 GPUs using the Songs dataset [58], varying the number of tuples. When running on one GPU, the runtime of DeepBlocker increases substantially when the number of tuples exceeds 40k. Worse still, it consumes excessive memory due to the large embeddings and intermediate results generated during similarity computation. Although the runtime of DeepBlocker can be reduced by using more GPUs, the issue remains, *e.g.,* even with four GPUs in Figure 1(a), DeepBlocker is still slower than the rule-based blocker that runs on one GPU.

In contrast, rule-based blocking methods demonstrate potential for achieving scalability by leveraging multiple *blocking rules*. Each rule employs various comparisons with logical operators such as AND, OR, and NOT to discard unqualified tuple pairs. Heuristic methods are also generally classified into this category [49]. For instance, a blocking rule for books may state "If titles match and the number of pages match, then the two books match" [44]. We refer to the comparisons in this rule as *equality comparisons*, as they require exact equality. Another example, referred to as *similarity comparisons*, is presented in [63], which adopts the Jaccard similarity to determine whether a pair of tuples requires further matching. Rule-based approaches complement DL-based approaches by providing flexibility, explainability, and scalability in the blocking process [17]. Moreover, by incorporating domain knowledge into blocking rules, these approaches can readily adapt to different domains.

Example 1: As a critical step for data consistency, an e-commerce company (*e.g.,* Amazon [6]) conducts ER for products, to enhance operations for *e.g.,* product listings and inventory management.

To identify duplicate products, the blocking rule φ_1 may fit.

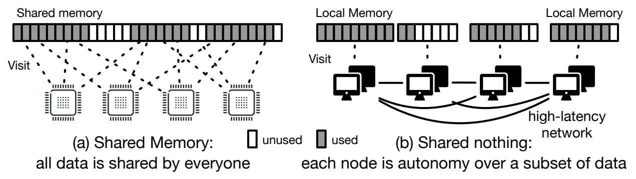

Shared memory
Visit
(a) Shared Memory: all data is shared by everyone
unused
used
Local Memory
Local Memory
Visit
high-latency network
(b) Shared nothing: each node is autonomy over a subset of data

Figure 2: Shared memory vs shared nothing architectures

φ_1: Two products are potentially matched if (a) they have same color and price, (b) they are sold at same store, (c) their names are similar.

Here φ_1 is a conjunction of attribute-wise comparisons, where both equality (parts (a) and (b)) and similarity comparisons (part (c)) are involved. In Section 2, we will formally define φ_1. □

Rule-based blocking in ER has attracted a lot of attention. A variety of rule-based blockers are in place (surveyed in [49, 64]). Unfortunately, most of them are designed for CPU-based (shared nothing) architectures, leading to unsatisfactory performance. Specifically, in a shared nothing architecture as shown in Figure 2(b) (adopted by many MapReduce-based systems), data is partitioned and spread across a set of processing units. Each processing unit conducts blocking independently with its own local memory, leading to potential scalability issues due to skewed partitions and increasing communication cost when tuples in different partitions should be paired. The shared memory architecture (Figure 2(a)) is the opposite: all data is accessible from all processing units, allowing for efficient data sharing, collaboration between processing units and dynamic workload scheduling. GPUs are typically based on shared memory architectures, offering promising opportunities to achieve blocking parallelism. However, unlike DL-based blocking approaches, few rule-based methods support the massive parallelism offered by GPUs, despite their greater potential in scalability (see Figure 1).

To make practical use of rule-based blocking, several questions have to be answered. Can we parallelize it under a share memory architecture, utilizing massive parallelism of a GPU? Can we explore characteristics of GPUs and CPUs, to effectively collaborate them?

HyperBlocker. To answer these, we develop HyperBlocker, a GPU-accelerated system for rule-based blocking in Entity Resolution. As proof of concept, we adopt matching dependencies (MDs) [27] for rule-based blocking. As a class of rules developed for record matching, MDs are defined as a conjunction of (similarity) predicates and support both equality and similarity comparisons. Compared with prior works, HyperBlocker has the following unique features.

(1) A pipelined architecture. HyperBlocker adopts an architecture that pipelines the memory access from/to CPUs for data transfer, and operations on GPUs for rule-based blocking. In this way, the data transfer and the computation on GPUs can be overlapped, so as to "cancel" the excessive data transfer cost.

(2) Execution plan generation on CPUs. To effectively filter unqualified tuple pairs, blocking must be optimized for the underlying data (resp. blocking rules); in this case, we say that the blocking is *data-aware* (resp. *rule-aware*). To our knowledge, no prior methods, neither on CPUs nor on GPUs, have considered data/rule-awareness for their execution models. HyperBlocker designs an effective execution plan generator to warrant efficient rule-based blocking.

(3) Hardware-aware parallelism on GPUs. Due to different characteristics of CPUs and GPUs, a naive approach that applies existing CPU-based blocking on GPUs makes substantial processing capacity untapped. We develop a variety of GPU-based parallelism strategies, designated for rule-based blocking, by exploiting the hardware characteristics of GPUs, to achieve massive parallelism.

(4) Multi-GPUs collaboration. It is already hard to offload tasks on CPUs. This problem is even exacerbated under multi-GPUs, due to the complexities of task decomposition, (inter-GPU) resource management, and workload balancing. HyperBlocker provides effective partitioning and scheduling strategy to scale with multiple GPUs.

With these attractive features, HyperBlocker is capable of conducting rule-based blocking with shorter time and less memory, as shown in Figure 1; more sophisticated experiments are shown later.

Contribution & organization. After reviewing background in Section 2, we present HyperBlocker as follows: (1) its unique architecture and system overview (Section 3); (2) the rule/data-aware execution plan generator (Section 4); (3) the hardware-aware parallelism and the task scheduling strategy across GPUs (Section 5); and (4) an experimental study (Section 6). Section 7 presents related work.

Using real-life and synthetic datasets, we find the following: (a) HyperBlocker speedups prior distributed ER blocking systems and GPU baselines by at least 6.8× and 9.1×, respectively. (b) Combining HyperBlocker with the SOTA ER matcher saves at least 30% time with comparable accuracy. (c) HyperBlocker is scalable, *e.g.,* it can process 36M tuples in 1604s. (d) While promising, DL-based blocking methods are not always the best. By carefully optimizing rule-based blocking on GPUs, we share valuable lessons/insights about when rule-based approaches can beat DL-based ones and vice versa.

2 PRELIMINARIES

We first review notations (more in [7]) for ER, blocking, and GPUs.

Relations. Consider a schema $R = (\text{eid}, A_1, \ldots, A_n)$, where A_i is an attribute ($i \in [1, n]$), and eid is an entity id, such that each tuple of R represents an entity. A relation D of R is a set of tuples of schema R.

Entity resolution (ER). Given a relation D, ER is to identify all tuple pairs in D that refer to the same real-life entity. It returns a set of tuple pairs (t_1, t_2) of D that are identified as *matches*. If t_1 does not match t_2, (t_1, t_2) is referred to as a *mismatch*.

Most existing methods typically conduct ER in three steps:

(1) Data partitioning. The tuples in relation D are divided into multiple disjoint data partitions, namely $P_1, P_2, \ldots, P_m$, so that tuples of similar entities are put into the same data partition.

(2) Blocking. Each tuple pair (t_1, t_2) from the same data partition P (*i.e.,* $(t_1, t_2) \in P \times P$) is a potential match that requires further verification. To reduce cost, a blocking method $\mathcal{A}_{\text{block}}$ (*i.e.,* blocker) is often adopted to filter out those pairs that are definitely mismatches *efficiently*, instead of directly examining every tuple pair. Denote the resulting set of remaining tuple pairs obtained from partition P by $\text{Ca}(P) = \{(t_1, t_2) \in P \times P \mid (t_1, t_2) \text{ is not filtered by } \mathcal{A}_{\text{block}}\}$.

(3) Matching. For the remaining tuple pairs in $\text{Ca}(P)$ of each data partition P, an accurate (but expensive) matcher will be applied, to make the final decision of matches/mismatches.

Our scope: blocking. Note that in some works, both steps (1) and

Table 1: A relation D of schema Products, where the dash ("-") denotes a missing value.

eid	pno	pname	price	sname	description	color	saddress
e_1	t_1	Apple Mac Air	\$909	Comp. World	Apple MacBook Air (13-inch, 8GB RAM, 256GB SSD)	Gray	9 Barton Grove, McCulloughmouth
e_2	t_2	ThinkPad	-	Smith's Tech	ThinkPad E15, 15.6-inch full HD IPS display, Intel Core i5-1235U processor, (16GB) RAM | 512GB PCIe SSD)	Gray	Seg Plaza, Hua qiang North Road
e_2	t_3	ThinkPad	\$849	Smith's Tech	Lenovo E15 Business ThinkPad, 15.6-inch full HD IPS display, 12 generation Intel Core i5, 16GB RAM, 512GB SSD	Gray	Seg Plaza, Hua qiang North Road
e_1	t_4	MacBook Air	\$909	Comp. World	Apple 2022 MacBook Air M2 chip 13-inch,8 GB RAM,256 GB SSD storage gray	Gray	-
e_1	t_5	MacBook Air	\$909	Comp. World	-	Gray	Barton Grove, McCulloughmouth

(2) are called blocking. To avoid ambiguity, we follow [73] and distinguish partitioning from blocking. We mainly focus on *blocking*, *i.e.,*

- *Input*: A relation D of the tuples of schema R, where the tuples in D are divided into m partitions $P_1, \ldots, P_m$.
- *Output*: The set $\mathsf{Ca}(P_i)$ of candidate tuple pairs on each P_i.

Although our work can be applied on data partitions generated by *any* existing method, we optimize over multiple data partitions, by exploiting designated GPU acceleration techniques (Section 5.3).

While blocking focuses more on efficiency and matching focuses more on accuracy, they can be used without each other, *e.g.,* one can directly employ rules [27] for ER or apply an ER matcher [51] on the Cartesian product of the entire partition. When blocking is used alone on a given partition P, all tuple pairs in $\mathsf{Ca}(P)$ are identified as matches. In Section 6, we will test HyperBlocker with or without a matcher, to elaborate the trade-off between efficiency and accuracy.

Rule-based blocking. We study rule-based blocking in this paper, due to its efficiency and explainability remarked earlier. We review a class of matching dependencies (MDs), originally proposed in [27].

Predicates. Predicates over schema R are defined as follows:

$$p ::= t.A = c \mid t.A = s.B \mid t.A \approx s.B$$

where t and s are tuple variables denoting tuples of R, A and B are attributes of R and c is a constant; $t.A = s.B$ and $t.A = c$ compare the equality on *compatible* values, *e.g.,* t.eid = s.eid says that (t, s) is a potential match; $t.A \approx s.B$ compares the *similarity* of $t.A$ and $s.B$. Here any similarity measure, whether symmetric or asymmetric, can be used as $\approx$, *e.g.,* edit distance or KL divergence, such that $t.A \approx s.B$ is true if $t.A$ and $s.B$ are "similar" enough w.r.t. a threshold.

Rules. A (bi-variable) *matching dependency* (MD) over R is:

$$\varphi = X \rightarrow l,$$

where X is a conjunction of predicates over R with two tuple variables t and s, and l is t.eid = s.eid. We refer to X as the *precondition* of φ, and l as the *consequence* of φ, respectively.

Example 2: Consider a (simplified) e-commence database with self-explained schema Products (eid, pno, pname, price, sname (store name), description, color, saddress (store address)). Below are some examples MDs, where the rule in Example 1 is written as φ_1.

(1) φ_1 : t.color = s.color $\wedge$ t.price = s.price $\wedge$ t.sname = s.sname $\wedge$ t.pname $\approx_{\mathsf{ED}}$ s.pname $\rightarrow$ t.eid = s.eid, where $\approx_{\mathsf{ED}}$ measures the edit distance. As stated before, φ_1 identifies two products, by their colors, prices, product names and the stores sold.

(2) φ_2 : t.sname = s.sname $\wedge$ t.description $\approx_{\mathsf{JD}}$ s.description $\rightarrow$ t.eid = s.eid, where $\approx_{\mathsf{JD}}$ measures the Jaccard distance. The MD says that if two products are sold in the store and have a similar description, then they are identified as a potential match.

(3) φ_3 : t.saddress $\approx_{\mathsf{ED}}$ s.saddress$\wedge t$.description $\approx_{\mathsf{JD}}$ s.description $\rightarrow$ t.eid = s.eid. It gives another condition for identifying two products, *i.e.,* the two products with similar descriptions are sold from stores with similar addresses are potentially matched. □

Semantics. A *valuation* of tuple variables of an MD φ in D, or simply *a valuation of* φ, is a mapping h that instantiates the two variables t and s with tuples in D. A valuation h *satisfies* a predicate p over R, written as $h \models p$, if the following is satisfied: (1) if p is $t.A = c$ or $t.A = s.B$, then it is interpreted as in tuple relational calculus following the standard semantics of first-order logic [14]; and (2) if p is $t.A \approx s.B$, then $h(t).A \approx h(s).B$ returns true. Given a conjunction X of predicates, we say $h \models X$ if for *all* predicates p in X, $h \models p$.

Blocking. Rule-based blocking employs a set Δ of MDs. Given a partition P, a pair $(t_1, t_2) \in P \times P$ is in $\mathsf{Ca}(P)$ *iff* there exists an MD $\varphi : X \rightarrow l$ in Δ such that the valuation $h(t_1, t_2)$ of φ that instantiates variables t and s with tuples t_1 and t_2 satisfies the precondition of φ; we call such φ as a *witness* at (t_1, t_2), since it indicates that (t_1, t_2) is a potential match. Otherwise, (t_1, t_2) will be filtered.

Example 3: Continuing with Example 2, consider D in Table 1 and a valuation $h(t_1, t_4)$ that instantiates variables t and s with tuples t_1 and t_4 in D. Since $h(t_1, t_4)$ satisfies the precondition of φ_1, φ_1 is a witness at (t_1, t_4). Similarly, one can verify that φ_1 is not a witness at (t_2, t_3) since $h(t_2, t_3) \not\models t$.price = s.price (due to value missing). □

Discovery of MDs. MDs can be considered as a special case of entity enhancing rules (REEs) [28, 29]. We can readily apply the discovery algorithms for REEs, *e.g.,* [28, 29], to discover MDs (details omitted).

GPU hardware. As general processors for high-performance computation, GPUs offer the following benefits compared with CPUs.

First, GPUs provide massive parallelism by programming with CUDA (Compute Unified Device Architecture) [53]. A GPU has multiple SMs (Streaming Multiprocessors), where each SM accommodates multiple processing units. *e.g.,* V100 has 80 SMs, each with 64 CUDA cores. SMs handle the parallel execution of CUDA cores. In CUDA programming, CUDA cores are conceptually organized into TBs (Thread Blocks) and physically grouped into thread warps, each comprising subgroups of 32 threads. This hierarchical organization allows thousands of threads running simultaneously on GPUs.

Second, GPUs utilize the DMA (Direct Memory Access) technology, which enables direct data transfer between GPU memory and system memory. This not only reduces CPU overhead but also allows the GPU to handle multiple data streams simultaneously. However, the number of PCIe lanes determines the maximum number of streams that can transfer data simultaneously (*e.g.,* 16 PCIe lanes for V100). When multiple partitions perform data transfers over a PCIe lane, only one can utilize the lane at a time.

Third, GPUs adopt *SIMT (Single Instruction, Multiple Threads)* execution, where each SIMT lane is an individual unit that is responsible for executing a thread under a single instruction. *Thread divergence* can adversely affect the performance and it typically occurs in conditional statements (*e.g., if-else*), where some lanes take one execution path while the others take a different path. However, GPUs must execute different execution paths sequentially, rather than in parallel, resulting in underutilization of GPU resources.

3 HYPERBLOCKER: SYSTEM OVERVIEW

In this section, we present the overview of HyperBlocker, a GPU-accelerated system for rule-based blocking that optimizes the efficiency by considering rules, underlying data, and hardware simultaneously. In the literature, GPUs and CPUs are usually referred to as devices and hosts, respectively. We also follow this terminology.

Challenges. Existing parallel blocking methods typically rely on multiple CPU-powered machines under the shared nothing architecture, to achieve data partition-based parallelism. They reduce the runtime by using more machines, which, however, is not always feasible due to the increasing communication cost. Moreover, it is hard to strike for a good balance between parallelism and recall. Although a more fine-grained data partitioning strategy can improve parallelism, it inevitably affects the recall since a match (t_1, t_2) may be missed if t_1 and t_2 are distributed to different machines.

In light of these, HyperBlocker focus on parallel blocking under the shared memory architecture. This not only opens up new possibilities for optimizations but also introduces new challenges.

(1) Execution plan for efficient blocking. The efficiency of blocking depends heavily on how much and how fast we can filter mismatches. Therefore, a good execution plan that specifies how rules are evaluated is crucial. However, most existing solvers fail to consider the properties of rules/data during blocking. This motivates us to design a different method to get an effective execution plan.

(2) Hardware-aware parallelism. Existing solvers often adopt data partition-based parallelism on CPU-powered machines. However, when GPUs are involved, CPU-based techniques no longer suffice, since GPUs have radically different characteristics (Section 2). Novel GPU-based parallelism for blocking is required to improve GPU utilization, *e.g.*, by reducing thread wait stalls and thread divergence.

(3) Multi-GPUs collaboration. Existing parallel blocking solvers focus on minimizing communication cost across all workers under the shared nothing architecture [20, 22]. This objective no longer applies in multi-GPUs scenarios, where unique task decomposition, (inter-GPU) resource management and scheduling challenges arise.

Novelty. The ultimate goal of HyperBlocker is to generate the set $\mathsf{Ca}(P)$ of potential matches on each data partition P. To achieve this, we implement three novel components as follows:

(1) Execution plan generator (EPG) (Section 4). We develop a data-aware and rule-aware generator to generate execution plans. Here we say the execution plan is data-aware, since it considers the distribution and skewness of data so as to decide which predicates are evaluated first; similarly, the execution plan is rule-aware, since its evaluation order is optimized for the underlying rules.

(2) Parallelism optimizer (Section 5). We implement a specialized optimizer that exploits the hierarchical structure of GPUs, to optimize the power of GPUs by utilizing thread blocks (TBs) and warps. With this optimizer, blocking can be effectively parallelized on GPUs.

(3) Resource scheduler (Section 5). To achieve optimal performance over multiple GPUs, a partitioning strategy and a resource scheduler are developed to manage the resources, and balance the workload across multiple GPUs, minimizing idle time and resource waste.

Architecture. The architecture of HyperBlocker is shown in Figure 3. Taken a relation D of tuples and a set Δ of MDs discovered offline as input, HyperBlocker divides the tuples in D into m disjoint partitions and asynchronously processes these data partitions in a pipelined manner, so that the execution at devices and the data transfer can be overlapped. Such architecture helps to mitigate the excessive I/O costs, reduce idle time, and improve hardware utilization.

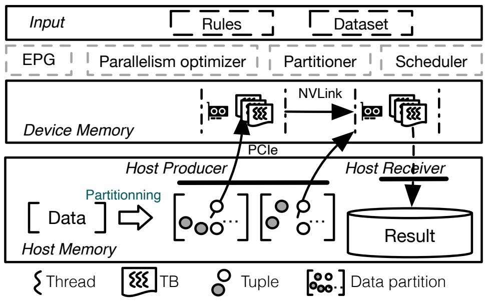


Figure 3: The pipelined architecture of HyperBlocker

Workflow. More specifically, HyperBlocker works in five steps:

(1) *Data partitioning.* HyperBlocker divides the tuples in D into m partitions, to allow parallel processing asynchronously.

(2) *Execution plan generation.* Given the set Δ of MDs discovered offline, an execution plan that specifies in what order the rules (and the predicates in rules) should be evaluated is generated at the host.

(3) *Host scheduling.* The blocking on each partition forms a computational task and the host dynamically assigns tasks to the queue(s) of available devices without interrupting their ongoing execution, minimizing the idle time of devices and improving resource utilization.

(4) *Device execution.* When a device receives the task assigned, it conducts the rule-based blocking on the corresponding data partition, following the execution plan generated in Step (2).

(5) *Result retrieval.* Once a task is completed on a device, the host will pull/collect the result (i.e., $\mathsf{Ca}(P)$) from the device.

To facilitate processing, HyperBlocker has two additional components: HostProducer and HostReceiver, where the former manages Steps (1), (2), and (3) and the latter handles Step (5). Steps (3) (4) (5) in HyperBlocker work asynchronously in a pipeline manner, to enable continuous execution without unnecessary waiting.

4 EPG: EXECUTION PLAN GENERATOR

Given the set Δ of MDs and a partition P of D, a naive approach to compute $\mathsf{Ca}(P)$ is to evaluate each MD in Δ for all pairs in P. That is, to decide whether (t_1, t_2) is in $\mathsf{Ca}(P)$, we perform $O(\sum_{\varphi \in \Delta} |\varphi|)$ predicate evaluation, where $|\varphi|$ is the number of predicates in φ. Worse still, there are $O(|\Delta|!|\varphi|!)$ possible ways to evaluate all MDs in Δ, since both MDs in Δ and predicates in each φ can be evaluated in arbitrary orders. However, not all orders are equally efficient.

Example 4: In Example 3, φ_1 is a witness at (t_1, t_4) while φ_3 is not. If we first evaluate φ_1 for (t_1, t_4), (t_1, t_4) is identified as a potential match and there is no need to evaluate φ_3. Moreover, when evaluating φ_1 for another pair (t_2, t_3), we can conclude that φ_1 is not a witness at (t_2, t_3), as soon as we find $h(t_2, t_3) \not\models t.\mathsf{price} = s.\mathsf{price}$. □

Challenges. Given the huge number of possible evaluation orders, it is non-trivial to define a good one, for three reasons:

(1) Rule priority. Recall that in rule-based blocking, as long as there exists a witness at (t_1, t_2), (t_1, t_2) will be considered as a potential match. This motivates us to prioritize the rules in Δ so that promising ones can be evaluated early; once a witness is found, the evaluation of the remaining rules can be skipped.

(2) Reusing computation. MDs may have common predicates. To avoid evaluating a predicate repeatedly, we reuse previous results whenever possible, *e.g.,* given (t_1, t_2), $\varphi_1 : p \wedge X_1 \to l$ and $\varphi_2 : p \wedge X_2 \to l$, if φ_1 is not a witness at (t_1, t_2) since $h(t_1, t_2) \not\models p$, neither is φ_2.

(3) Predicate ordering. Given (t_1, t_2) and $\varphi : X \to l$, φ is not a witness at (t_1, t_2) if we find the first p in X such that $h(t_1, t_2) \not\models p$. However, to decide which p in X is evaluated first, we have to consider both the evaluation cost of p and the possibility of p being satisfied.

Novelty. To tackle these challenges, EPG in HyperBlocker gives a lightweight solution, by generating an execution plan to make the overall evaluation cost of Δ on P as small as possible. Its novelty includes (a) a new notion of execution tree to allow efficient reuse of common predicates, (b) a rule-aware scoring strategy, so as to decide which MDs in Δ are evaluated with higher priority, and (c) a data-aware predicate ordering scheme, to strike for a balance between the cost and the effect of evaluating a predicate.

Below we first give the formal definition of execution plans and then show how EPG generates a good execution plan.

4.1 Execution plan

An execution plan specifies how rules and predicates in Δ are evaluated. Although an execution plan can be represented in different ways, we represent it as an *execution tree*, denoted by $\mathcal{T}$ in this paper for its conciseness and simplicity (an example is given in Figure 4, to be explained in more detail later). (1) A node in $\mathcal{T}$ is denoted by N, where the root is denoted by N_0. (2) A *path* ρ *from the root* is a list $\rho = (N_0, N_1, \ldots, N_L)$ such that (N_{i-1}, N_i) is an edge of $\mathcal{T}$ for $i \in [1, L]$; the length of ρ is L, *i.e.,* the number of edges on ρ. (3) We refer to N_2 as a *child node* of N_1 if (N_1, N_2) is an edge in $\mathcal{T}$, and as a *descendant* of N_1 if there exists a path from N_1 to N_2; conversely, we refer to N_1 as a *parent node* (resp. predecessor) of N_2. (4) Each edge e represents a predicate p and is associated with a score, denoted by $\text{score}(e)$, indicating the priority of e. (5) A node is called a *leaf* if it has no children and $\mathcal{T}$ has $|\Delta|$ leaves, where each leaf is associated with a rule $\varphi : X \to l$ in Δ; the length of the path from the root to the leaf is $|X|$ (*i.e.,* the number of predicates in X) and for each predicate in X, it appears exactly once in an edge on the path. (6) The leaves of two MDs may have common predecessors, in addition to the root; intuitively, this means that the MDs have common predicates. With a slight abuse of notation, we also denote an execution plan by $\mathcal{T}$.

Evaluating an execution plan. For each pair (t_1, t_2), it is evaluated by exploring $\mathcal{T}$ via depth-first search (DFS), starting at the root. At each internal node N of $\mathcal{T}$, we pick a child N_c such that the edge (N, N_c), whose associated predicate is p, has the highest score among all children of N. Then we check whether $h(t_1, t_2) \models p$. If it is the case, we move to N_c and process N_c similarly. Otherwise, we check whether N still has other unexplored children and we process them similarly, according to the decreasing order of scores. If all children of N are explored, we return to the parent N_p of N and repeat the process. The evaluation completes if we reach a leaf of $\mathcal{T}$. Suppose the rule associated with this leaf is $\varphi : X \to l$. This means $h(t_1, t_2)$ satisfies all predicates in X, along the path from the root to that leaf, and thus $h(t_1, t_2) \models X$, *i.e.,*we find a witness at (t_1, t_2).

Example 5: Consider an execution tree $\mathcal{T}$ in Figure 4(b), which depicts MDs in Example 2. For simplicity, we denote a predicate $t.A = s.A$ (resp. $t.A \approx s.A$) by $p_A^{=}$ (resp. $p_A^{\approx}$) and the score associated with each edge is labeled. DFS starts at the root, which has two children. It first explores the edge labeled $p_{\text{sname}}^{=}$ since its score is higher. When DFS completes, MDs φ_2, φ_1 and φ_3 are checked in order. □

4.2 Execution plan generation

Taking the set Δ of MDs as input, EPG in HyperBlocker returns an execution plan $\mathcal{T}$ in the following two major steps:

(1) We order all predicates appeared in Δ, by estimating their evaluation costs via a shallow model and quantifying their probabilities of being satisfied, by investigating the underlying data distribution.

(2) Based on the predicate ordering, we build an execution tree $\mathcal{T}$ by iterating MDs in Δ. Moreover, we compute a score for each edge in $\mathcal{T}$, by considering the probability of finding a witness, *i.e.,* reaching a leaf, if we explore $\mathcal{T}$ following this edge.

Note that plan generation can be regarded as a pre-processing step for blocking, *i.e.,* once a plan is generated, it is applied in *all* partitions of D; as will be shown in Section 6, plan generation is fast and thus, will not be the bottleneck. Below we present these two steps. For simplicity, we assume *w.l.o.g.* that D is itself a partition.

Predicate ordering. Denote by $\mathcal{P}$ the set of all predicates appeared in Δ. Intuitively, not all predicates in $\mathcal{P}$ are equally potent for evaluation, *e.g.,* although textual attributes (*e.g.,* description) are often more informative in identifying entities than categorical ones (*e.g.,* color), the former comparison is more expensive. A simple strategy is to order predicates by only considering their attribute types and operators (*e.g.,* always evaluate equality first). However, the time/effect of evaluating a predicate for distinct tuple pairs can be different. Without taking the underlying data into account, it can lead to poor ordering. Motivated by this, we order the predicates in $\mathcal{P}$ by their "cost-effectiveness" on the input data D. Similar trade-off also appears in other scenarios, *e.g.,* query optimization in RDBMS [57].

For simplicity, below we consider a predicate p that compares A-values of two tuples, *i.e.,* $t.A = s.A$ or $t.A \approx s.A$ (simply $p_A^{=}$ or $p_A^{\approx}$). All discussion extends to other predicate types, *e.g.,* $t.A = s.B$.

Evaluation cost. We measure the evaluation cost of a predicate p by the time for evaluating p; a predicate that can be evaluated quickly should be checked first. Given a predicate p in $\mathcal{P}$ and a relation D, the evaluation cost of p on D, denoted by $\text{cost}(p, D)$, is:

$$\text{cost}(p, D) = \sum_{(t_1, t_2) \in D \times D} T_p(t_1, t_2).$$

where $T_p(t_1, t_2)$ denotes the actual time for checking $h(t_1, t_2) \models p$.

Note that it can be costly to iterate all tuple pairs in D to compute the exact evaluation cost of p on D, *e.g.,* on average, it takes more than 100s to compute $\text{cost}(p, D)$ on a dataset with 2M tuples (see more in [7]). Motivated by this, below we train a shallow NNs, denoted by $\mathcal{N}$, (*i.e.,* a small feed-forward neural network [45]) to estimate the exact $T_p(t_1, t_2)$, since it has been proven effective in approximating a continuous function on a closed interval [71].

Shallow NNs. The inputs of $\mathcal{N}$ are two tuples t_1 and t_2, and a predicate p, that compares the A-values of t_1 and t_2. It first encodes the attribute type and the A-value of t_1 into an embedding $\vec{t_1}$; similarly for $\vec{t_2}$. The embeddings are then fed to a feed-forward neural network, which outputs the estimated time for evaluating p on (t_1, t_2). We train $\mathcal{N}$ offline with training data sampled from historical logs, so that the training data follows the same distribution as D (see [7]).

Estimated cost. Based on $\mathcal{N}$, the estimated cost of p on D is

$$\hat{\text{cost}}(p, D) = \text{norm}\Big(\sum_{(t_1,t_2)\in D\times D} \mathcal{N}(p, t_1, t_2)\Big),$$

where $\text{norm}(\cdot)$ normalizes the estimated cost in the range (0,1].

Remark. As will be verified in Section 6, although $\hat{\text{cost}}(p, D)$ is computed by iterating tuple pairs in D (*i.e.,* quadratic cost), it is much faster than computing $\text{cost}(p, D)$. Better still, the predicate orderings derived from $\hat{\text{cost}}(p, D)$ is close to that derived from $\text{cost}(p, D)$.

Probability of being satisfied. We measure the effectiveness of predicate p by its possibility of being satisfied, *i.e.,* given the attribute A compared in p, we quantify how likely t_1 and t_2 have distinct/dissimilar values on A. If t_1 and t_2 do so with a high probability, p is less likely to be satisfied; such predicate should be evaluated first since it concludes that an MD involving p is not a witness early.

To achieve this, we investigate the data distribution in D. Specifically, we use LSH [16] to hash the A-values of all tuples into k buckets, so that similar/same values are hashed into the same bucket with a high probability, where k is a predefined parameter.

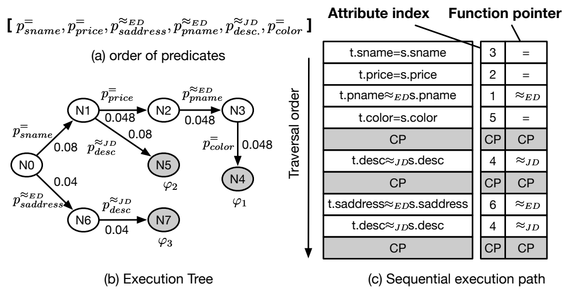
Denote the number of tuples hashed to the i-th bucket by b_i. Intuitively, the evenness of hashing results reflects the probability of p being satisfied. If all tuples are hashed into the same bucket, it means that the A-values of all tuples are similar and thus p (which compares the A-values) is likely to be satisfied by many pairs (t_1, t_2); such predicates should be evaluated with low-priority. Motivated by this, the probability of p being satisfied on D, denoted by $\text{sp}(p, D)$, is estimated by measuring the evenness of hashing, *i.e.,*

$$\text{sp}(p, D) = \text{norm}\Big(\sqrt{\frac{1}{k}\sum_{i=1}^{k}(b_i - \frac{|D|}{k})^2}\Big).$$

Ordering scheme. Putting these together, we can order all the predicates p in $\mathcal{P}$ by the cost-effectiveness, defined to be $\frac{1-\text{sp}(p,D)}{\hat{\text{cost}}(p,D)}$. Intuitively, hard-to-satisfied predicates will be evaluated first, since they are more likely to fail a rule, while costly predicates will be penalized, to strike a balance between the cost and the effectiveness.

Example 6: Consider two predicates $p^{=}_{\text{color}}$ and $p^{\approx\text{ED}}_{\text{pname}}$ in φ_1. On the one hand, since $p^{=}_{\text{color}}$ is an equality comparison while $p^{\approx\text{ED}}_{\text{pname}}$ computes the edit distance, $p^{\approx\text{ED}}_{\text{pname}}$ is more costly to evaluate, *e.g.,* $\hat{\text{cost}}(p^{=}_{\text{color}}, D) = 0.1 < \hat{\text{cost}}(p^{\approx\text{ED}}_{\text{pname}}, D) = 0.6$. On the other hand, since all tuples in D have the same color (and satisfy $p^{=}_{\text{color}}$), we have $\text{sp}(p^{=}_{\text{color}}, D) = 1$; similarly, let $\text{sp}(p^{\approx\text{ED}}_{\text{pname}}, D) = 0.4$. Then the cost-effectiveness of $p^{=}_{\text{color}}$ and $p^{\approx\text{ED}}_{\text{pname}}$ are $\frac{1-1}{0.1} = 0$ and $\frac{1-0.2}{1} = 0.8$, respectively, and $p^{\approx\text{ED}}_{\text{pname}}$ is ordered before $p^{=}_{\text{color}}$ (see Figure 4(a)). $\square$

Constructing an execution tree. We initialize the execution tree $\mathcal{T}$ with a single root node N_0. Then based on the predicate ordering, we progressively construct $\mathcal{T}$ by processing the MDs in Δ one by one. For each MD $\varphi : X \to l \in \Delta$, we assume the predicates in X are sorted in the descending order of their cost-effectiveness, *i.e.,* if X is $p_1 \wedge p_2 \wedge \ldots \wedge p_{|X|}$, then $\frac{1-\text{sp}(p_i,D)}{\hat{\text{cost}}(p_i,D)} > \frac{1-\text{sp}(p_j,D)}{\hat{\text{cost}}(p_j,D)}$ for $1 \le i < j \le |X|$. We traverse $\mathcal{T}$, starting from the root, and process the predicates in X, starting from p_1. Suppose that the traversal is at a node N and the predicate we are processing is p_i. We check the children of N. If there exists a child node N_c of N such that the edge (N, N_c) represents p_i, we move to this child and process the next predicate p_{i+1} in X. Otherwise, we create a new child node N_c for N such that the edge (N, N_c) represents p_i, move to this new child and process the next predicate p_{i+1} in X. The traversal process continues until all predicates in X are processed and we set the current node we reach as a leaf node, whose associated rule is φ.



	Attribute index	Function pointer
t.sname=s.sname	3	=
t.price=s.price	2	=
t.pname$\approx_{ED}$s.pname	1	$\approx_{ED}$
t.color=s.color	5	=
CP	CP	CP
t.desc$\approx_{JD}$s.desc	4	$\approx_{JD}$
CP	CP	CP
t.saddress$\approx_{ED}$s.saddress	6	$\approx_{ED}$
t.desc$\approx_{JD}$s.desc	4	$\approx_{JD}$
CP	CP	CP

Figure 4: Execution tree

Example 7: The predicate ordering is shown in Figure 4(a). Assume that we have processed φ_1 and created path $(N_0, N_1, N_2, N_3, N_4)$ in $\mathcal{T}$ in Figure 4(b). Then we show how $\varphi_2 : p^{=}_{\text{sname}} \wedge p^{\approx\text{JD}}_{\text{description}} \to l$ is processed. We start from the root and process $p^{=}_{\text{sname}}$. Since there is a child N_1 of root labeled $p^{=}_{\text{sname}}$, we move to N_1 and process $p^{\approx\text{JD}}_{\text{description}}$. Since there is no child of N_1 labeled $p^{\approx\text{JD}}_{\text{description}}$, we create a new N_5 and label (N_1, N_5) as $p^{\approx\text{JD}}_{\text{description}}$. Since all predicates in φ_2 are processed, N_5 is a leaf node, whose associated rule is φ_2. $\square$

Intuitively, given (t_1, t_2) and $\varphi \in \Delta$, if φ is more likely to be a witness at (t_1, t_2), it should be evaluated earlier. Motivated by this, we compute the probability for $\varphi : X \to l$ to be a witness on D as:

$$\text{wp}(\varphi, D) = \prod_{p\in X} \text{sp}(p, D),$$

if we assume the satisfaction of predicates as independent events; intuitively, if all predicates in X are satisfied, φ is a witness. If this does not hold, we can reuse historical logs and estimate $\text{wp}(\varphi, D)$, to be the proportion of historical pairs such that φ is a witness. Since the evaluation of MDs in Δ is guided by edge scores during DFS on $\mathcal{T}$, below we define the score of a given edge e based on $\text{wp}(\varphi, D)$.

Edge score. For each MD φ, we denote by ρ_φ the path of $\mathcal{T}$ from root to the leaf whose associated MD is φ. We compute the set of MDs φ in Δ such that the given edge e is part of ρ_φ and denote it by Ψ_e, *i.e.,* $\Psi_e = \{\varphi \in \Delta \mid e \text{ is part of } \rho_\varphi\}$. The score of edge e is $\text{score}(e) = \max_{\rho_\varphi \in \Psi_e} \text{wp}(\varphi, D)$. This said, edges leading to promising MDs will have high scores and thus, will be explored early via DFS on $\mathcal{T}$.

Example 8: Let $\text{sp}(p^{=}_{\text{sname}}, D) = 0.4$ and $\text{sp}(p^{\approx\text{JD}}_{\text{description}}, D) = 0.2$. Then $\text{wp}(\varphi_2, D) = 0.4 \times 0.2 = 0.08$. Assume that we also compute $\text{wp}(\varphi_1, D) = 0.048$. Then the score of edge $e = (N_0, N_1)$ is $\max\{\text{wp}(\varphi_1, D), \text{wp}(\varphi_2, D)\} = 0.08$, since e is part of both ρ_{φ_1} and ρ_{φ_2}. $\square$

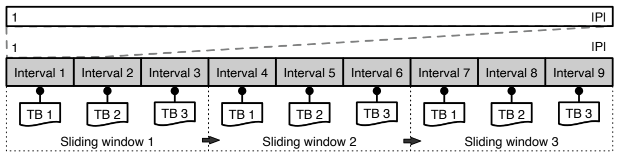


Figure 5: Parallel sliding windows

Complexity. It takes EPG $O(c_{\text{unit}}|\mathcal{P}| + |\mathcal{P}|\log(|\mathcal{P}|) + |\varphi||\Delta|)$ time to generate the execution plan, where c_{unit} is the unit time for computing the cost-effectiveness of a predicate. This is because the predicate ordering can be obtained in $O(c_{\text{unit}}|\mathcal{P}| + |\mathcal{P}|\log(|\mathcal{P}|))$ time and the tree can be constructed in $(|\varphi||\Delta|)$ time, by scanning Δ once.

Remark. Note that as a by-product of ensuring the predicate ordering and DFS tree traversal, we can reuse the evaluation results of common "prefix" predicates (*i.e.,* common predecessors in $\mathcal{T}$).

Example 9: We evaluate $\mathcal{T}$ in Figure 4(b) for (t_1, t_5) in D. After evaluating $p^{=}_{\text{sname}}$, we find that $h(t_1, t_5) \not\models p^{\approx\text{JD}}_{\text{description}}$ and thus we cannot move to N_5. Then DFS will return back to N_1 and continue to check unexplored children of N_1 (*i.e.,* N_2). In this way, the common "prefix" predicate $p^{=}_{\text{sname}}$ of φ_1 and φ_2 is only evaluated once. □

5 OPTIMIZATIONS AND SCHEDULING

As remarked earlier, GPUs adopt SIMT execution, where a thread is idle if other threads take longer (*i.e.,* thread divergence). Below are sources of divergence (some are specific to rule-based blocking).

- *Conditional statements.* GPUs may execute different paths in conditional statements (Section 2), *e.g.,* one pair may be quickly identified as a potential match if the first MD checked is its witness, while another is found as a mismatch until all MDs are iterated.
- *Data-dependent execution.* The execution depends on the data being processed, *e.g.,* even for the same predicate, the evaluation time on different tuples is different (*e.g.,* long vs. short text).
- *Imbalanced workloads.* If the workload assigned to each thread is not evenly distributed, some may complete faster than others.

While thread divergence is a general issue in GPU-programming, rule-based blocking offers some unique opportunities to mitigate it, *e.g.,* the evaluations of distinct pairs are often *independent* tasks, making it possible to (a) assign approximately equal tasks to threads, to enable *workload balancing*, and (b) "steal" tasks from other threads, to cope with different *execution paths* and *data-dependent execution*. Below we present a family of hardware-aware optimization and scheduling techniques that exploit GPU characteristics for massive parallelism. Our novelty includes: (a) efficient device execution of an execution plan (Section 5.1), (b) strategies to mitigate divergence (Section 5.2), and (c) collaboration of multiple GPUs (Section 5.3). More optimization strategies are presented in [7].

5.1 Execution plan on GPUs

The execution plan $\mathcal{T}$, initially generated on CPUs, will undergo the evaluation on GPUs in a DFS manner. However, DFS tree traversal is typically recursively implemented, which is not efficient on GPUs. It may exacerbate divergence since each call adds a recursive function to the stack and incurs message payloads (see Section 6). Code transformation may help alleviate such issues. Moreover, although we can reuse "prefix" predicates via DFS, some predicates may still be evaluated repeatedly, *e.g.,* $p^{\approx\text{JD}}_{\text{description}}$ in φ_2 and φ_3. Optimized structures are required to harness the power of GPUs.

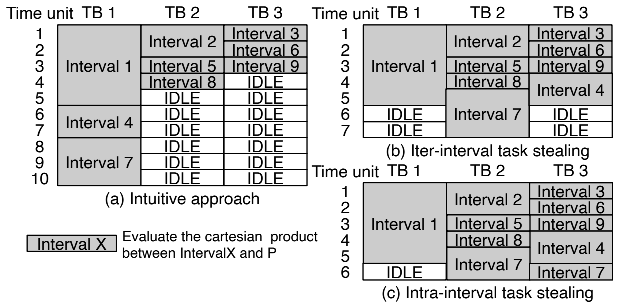


Figure 6: Task-stealing

Tree traversal on GPUs. Note that upon completion of the tree construction, the evaluation order is fixed. Thus the DFS traversal of the tree on CPUs can be translated to a *sequential execution path*, which is an ordered list of predicates, on GPUs (see Figure 4(c) for the sequential execution path of the tree in Example 5).

We maintain two structures for each predicate p in the execution path: an index buffer and a function pointer buffer, which store the indices of attributes compared in p, and the function pointer of the comparison operator in p, respectively, *e.g.,* for predicate $p^{=}_{\text{sname}}$: t.sname = s.sname, its comparison operator is "=" and its attribute index is 3 since sname is the 3rd attribute in schema Products. In addition, at the end of each rule, we set a checkpoint (CP). When a GPU thread encounters a CP, it knows that the undergoing tuple pair satisfies a rule and it can skip the subsequent computation.

Reusing computation. To avoid repeated evaluation, we additionally maintain a bitmap for all predicates on GPUs. The bit of a predicate p is set to true if p has been evaluated. If this is the case, we can directly reuse previous results. This bitmap can also be used for symmetric predicates (*i.e.,* $h(t_1, t_2) \models p$ iff $h(t_2, t_1) \models p$).

Note that to be general, we do not make the assumption that a witness φ at (t_1, t_2) is also a witness at (t_2, t_1), due to, *e.g.,* asymmetric similarity comparison (Section 2). However, we can extend HyperBlocker if such assumption holds, by maintaining a bitmap to avoid repeated evaluation for (t_2, t_1) if (t_1, t_2) is already evaluated.

5.2 Divergence mitigation strategies

To further mitigate divergence, we propose two GPU-oriented strategies, namely *parallel sliding windows (PSW)* and *task-stealing*.

Parallel sliding windows (PSW). Given a partition P, PSW processes it with only a few index jumps; it also helps GPUs evenly distribute workloads across SMs. Specifically, PSW works in 3 steps:

(1) We divide P into $\frac{|P|}{n_t}$ intervals, where each interval consists of n_t tuples. These intervals are processed with a fixed-size window, which slides the intervals from left to right. Within each window, we assign an interval to a Thread Block (TB) with warps of 32 threads; each thread in the TB is responsible for a tuple t_i in the interval.

(2) Assume that a thread is responsible for tuple t_i. Then this thread compares t_i with all the other tuples, say t_j, in P according to the execution plan $\mathcal{T}$ and decides whether (t_i, t_j) is a potential match.

(3) When all threads of a TB finish, this TB writes the results back to the host memory and it will move on to process the next interval in the next sliding window until the window reaches the end.

Note that in total, it requires $\frac{|P|}{n_t n_w}$ sequential index jumps for each TB, where n_w is the size of the sliding window.

Example 10: As shown in Figure 5, a data partition P is divided 9 intervals and the size of the sliding window is 3 (*i.e.,* $n_w = 3$). Interval 1 is assigned to TB1, where each thread in TB1 will compare a tuple in Interval 1 with all other tuples in P. When all threads of TB1 finish evaluation, TB1 moves on to process Interval 4. $\square$

Remark. Note that PSW achieves *tuple-level parallelism* (*i.e.,* tuples are evaluated in parallel; each thread is responsible for a tuple). As shown in [46], the sliding window-like method ensures that each TB (resp. each thread) is assigned roughly equal intervals (resp. tuples).

There are other design choices for parallelism, *e.g., rule-level, predicate-level* and *token-level* parallelisms, so that the entire warp is assigned to a tuple pair and rules, predicates and tokens in text are evaluated in parallel, respectively (one thread per rule/predicate/-token, see [7] for more). However, these designs may exacerbate divergence, due to, *e.g.,* synchronization cost (*e.g.,* one thread has to wait for results from other threads to make a final decision), or incur unnecessary computation, *e.g.,* when a thread finds a witness φ at (t_1, t_2) (resp. a predicate p in φ such that $h(t_1, t_2) \not\models p$), other threads may still check other rules in Δ (resp. other predicates in φ). As an evidence, when testing on a dataset with 6M tuple pairs, tuple-level parallelism is 7.3× faster than predicate-level one.

Task-stealing. Although each TB will process roughly equal intervals, the execution time of different intervals is not the same, due to conditional statements and data-dependent execution remarked earlier. This said, the workloads of all TBs can still be imbalanced.

Example 11: Continuing Example 10, three TBs process 9 intervals in Figure 6(a). Even though each TB is assigned 3 intervals, the execution times can still skew, *e.g.,* the total time units required by TB1 and TB3 are 10 and 3, respectively, *i.e.,* TB3 is idle for 7 time units. $\square$

Below we introduce both the inter-interval and intra-interval task-stealing strategies to further balance the workloads.

Inter-interval task-stealing. It is commonly observed that the execution times of some TBs are longer than the others. In this case, a large number of TBs are idle, waiting for the slowest TB.

In light of this, we employ an inter-interval task-stealing strategy to allow idle TBs to work on not-yet-processed intervals.

Specifically, we maintain a bitmap in global memory, where each bit indicates the status of an interval, so that TBs can steal not-yet-processed intervals from each other. Each TB processes intervals in two stages: (a) It first processes its assigned intervals one by one. Whenever a TB starts to process an interval, the bitmap is checked. If the bit of the interval is false (*i.e.,* not yet processed), it processes this interval and sets the bit true. (b) If this TB is idle after finishing all assigned intervals, it traverses the bitmap to steal a not-yet-processed interval, by setting the corresponding bit true and processing that interval. Other TBs will skip an interval if it has been stolen.

Example 12: In Example 11, TB3 finishes its assigned intervals after 3 time units. Then it checks the bitmap and steals Interval 4; similarly for TB2. Compared with the time in Figure 6(a), the total time units are reduced from 10 to 7 after stealing in Figure 6(b). $\square$

Intra-interval task-stealing. Recall that a thread for t_i will compare t_i with other tuples in P. Since the evaluation of distinct pairs is independent, we can even steal tasks from executing intervals. To facilitate this, we maintain two integers start and end, initialized to 1 and $|P|$, respectively, indicating the remaining range of tuples to be compared with t_i. Then this thread starts to evaluate (t_i, t_{start}). Upon completion, it sets start = start+1 and moves on to the next pair (t_i, t_{start}). When start = end, this thread finishes all evaluation for t_i.

Based on this, the intra-interval task-stealing works as follows. If TB$_a$ finishes all assigned intervals and there are no not-yet-processed intervals, it finds an executing TB$_b$ and iterates all threads in TB$_b$, so that the i-th thread in TB$_a$ steals half workload (*i.e.,* half pairs to be compared) from the i-th thread in TB$_b$. Assume the integers maintained for the i-th thread in TB$_b$ (resp. TB$_a$) are start$_b$ and end$_b$ (resp. start$_a$ and end$_a$). We set start$_a$ = start$_b$ + $\frac{\text{start}_b+\text{end}_b}{2}$, end$_a$ = end$_b$, and end$_b$ = start$_b$ + $\frac{\text{start}_b+\text{end}_b}{2} - 1$, *i.e.,* the latter half of tuples remained to be compared is stolen from each thread in TB$_b$.

Example 13: Continuing Example 12, when TB3 finishes Interval 4 stolen from TB1 in Figure 6(b), it finds no not-yet-processed intervals. However, since TB2 is still evaluating Interval 7, TB3 steals half remaining workload from it, saving 1 more time unit (Figure 6(c)). $\square$

5.3 GPU collaboration

A GPU server nowadays usually has multiple GPUs connected via NVLink [48] or PCIe. Scaling blocking to multiple GPUs is beneficial for jointly utilizing the computation and storage powers of GPUs.

In pursuit of this, one can split data evenly so that each GPU handles exactly one [62], or assign multiple partitions to each GPU in a round-robin manner [66]. These, however, do not work well since (a) workload can be imbalanced due to skewed execution times of partitions (see statistic in [7]), (b) pending partitions may wait when multiple partitions compete for limited PCIe bandwidth or CUDA cores (see Section 2) and (c) they independently conduct blocking on partitions and do not effectively handle scenarios where t_i and t_j reside on different partitions, resulting in elevated false-negative rates. To address this issue, one can duplicate tuples in multiple partitions [20, 22], but it incurs both memory and data transfer costs.

In light of these, we present a collaborative approach integrating partitioning and scheduling strategies, where the former aims at minimizing data redundancy while reducing false negatives and the latter prioritizes load balancing and minimizes resource contention.

Data Partitioning. A typical method for data partitioning computes a hash key for each tuple based on some attributes and tuples with same hash key are grouped together. Instead of sacrificing the accuracy (*e.g.,* using only one hash function) or unnecessarily duplicating tuples, HyperBlocker applies s hash functions to obtain s partition-keys, where s is the number of children N_c of the root node N_0 in the execution tree $\mathcal{T}$; each hash function is constructed from the predicate p associated with an edge (N_0, N_c). For example, if p is t.sname = s.sname, we hash tuples in D based on their values in attribute sname. The benefits are two-fold: (1) According to the construction of $\mathcal{T}$, the hash functions we constructed are often discriminative and might be shared by rules, *i.e.,* we can achieve good hashing with a few hashing functions. (2) We can assign each tuple a branch ID, indicating the hash function used. Only tuple pairs that share the same hash function are compared, thereby reducing redundant computations incurred by multiple hash functions.

Scheduling. HyperBlocker adopts a two-step scheduling strategy.

Table 2: Datasets

Dataset	Domain	#Tuples	Max #Pairs	#GT Pairs	#Attrs	#Rules	#Partitions
Fodors-Zagat	restaurant	866	1.8×10^4	112	6	1	1
DBLP-ACM	citation	4591	6.0×10^6	2294	4	10	8
DBLP-Scholar	citation	66881	1.7×10^8	5348	4	10	8
IMDB	movie	1.5M	8.1×10^{10}	0.2M+	6	10	128
Songs	music	0.5M	2.7×10^{11}	1.2M	8	10	128
NCV	vote	2M	1.0×10^{12}	0.5M+	5	10	512
TPCH	synthetic	4M	1.6×10^{13}	#	8	30	512
TFACC	traffic	10M	1.0×10^{14}	#	16	50	1024
TFACC$_{\text{large}}$	traffic	36M	1.3×10^{15}	#	16	50	1024

Initially, data partitions and GPUs are hashed to random locations on a unit circle [56]. If a partition P_i is assigned to an *ineligible* GPU (where there is no idle core or available PCIe bandwidth), it is rerouted to the nearest available GPU in a clockwise direction.

Remark. If data partitioning is done by a hashing function from a similarity predicate p, it is possible that $h(t_1, t_2) \models p$ but t_1 and t_2 reside on different partitions, leading to potential false negatives in blocking. In this case, a CUDA kernel [8] with local data P_i can optionally "pull" partition P_j from another kernel and evaluate $\mathcal{T}$ across P_i and P_j. The pull operation retrieves data from locations outside P_i, depending on whether P_i and P_j reside on the same GPU. If P_i and P_j reside on the same GPU, the pull operation is executed directly without any data transfer. Otherwise, the pull operation for P_j can be carried out using `cudaMemcpyPeer()` to take the advantages of high bandwidth and low latency provided by NVLink.

6 EXPERIMENTAL STUDY

We evaluated HyperBlocker for its accuracy-efficiency and scalability on both real-life and synthetic datasets. We also analyzed its performance, by conducting sensitivity tests and ablation studies.

Experimental setup. We start with the experimental setting.

Datasets. We used eight real-world public datasets in Table 2, which are widely adopted ER benchmarks and real-life datasets [3, 4, 10]. We also generated a synthetic dataset TPCH using TPCH dbgen [5]. Real-world datasets (except TFACC and TFACC$_{\text{large}}$) have labeled matches or mismatches as ground truths (GT). For datasets without ground truths, we assume the original datasets were correct, and randomly duplicated tuples as noises [30]. The training data consists of 50% of ground truths and 50% of randomly selected noise.

Baselines. As remarked in Section 2, although HyperBlocker is designed as a blocker, it can be used with or without a matcher. Thus, below we not only compared HyperBlocker against widely used blockers but also integrated ER solutions (*i.e.,* blocker + matcher).

We compared three distributed ER systems: (1) Dedoop [1, 42], (2) SparkER [13, 32], (3) DisDedup [9, 20], where DisDedup is the SOTA CPU-based parallel ER system, designed to minimize communication and computation costs; Dedoop focuses on optimizing computation cost; SparkER integrates Blast blocking [68] on Spark [12].

We also compared four GPU-based baselines: (4) DeepBlocker [73], (5) GPUDet [31], (6) Ditto [2, 51], (7) DeepBlocker$_{\text{Ditto}}$, where DeepBlocker is the SOTA DL-based blocker, GPUDet implements well-known similarity algorithms for tuple pair comparison, Ditto is the SOTA matcher, and DeepBlocker$_{\text{Ditto}}$ uses DeepBlocker as the blocker and Ditto as the matcher, respectively. Note that Ditto takes tuple pairs as input, instead of relations/partitions as other methods. Due to the high cost of Ditto, it is infeasible to feed the Cartesian product of data to Ditto. Thus, for each tuple in ground truths, we

Table 3: Comparison with the SOTA DL-based blocker

Method	Metric	Dataset		
		Fodors-Zagat	DBLP-Scholar	DBLP-ACM
DeepBlocker	Rec (%)	100 (+0)	98 (+5)	98 (+4)
	CSSR (‱)	15.1 (+14.5)	2.3 (+1.1)	2.2 (+1.8)
	Time (s)	6.1 (122×)	72.8 (11.0×)	8.0 (10.0×)
	Host Mem. cost (GB)	9.9 (49.5×)	14.0 (23.3×)	10.3 (34.3×)
	Device Mem. cost (GB)	0.9 (1.8×)	1.1 (1.6×)	0.9 (1.5×)
HyperBlocker	Rec (%)	100	93	94
	CSSR (‱)	0.6	1.2	0.4
	Time (s)	0.05	6.6	0.8
	Host Mem. cost (GB)	0.2	0.6	0.3
	Device Mem. cost (GB)	0.5	0.7	0.6

adopted a similarity-join method Faiss [39] to get the top-2 nearest neighbors, as a preprocessing step of Ditto. Denote the resulting baseline by Ditto$_{\text{top2}}$. Since Faiss often serves as a key component of many ER solutions [24, 73], we also compared Faiss in [7].

Besides, we also implemented several variants: (1) HyperBlocker, the basic blocker with all optimizations. (2) HyperBlocker$_{\text{Ditto}}$, an improved version that uses HyperBlocker as the blocker and Ditto as the matcher, respectively. Note that HyperBlocker$_{\text{Ditto}}$ is particularly compared against Ditto$_{\text{top2}}$ to show how we speed up the overall ER. (3) HyperBlocker$_{\text{noEPG}}$, a variant without EPG (Section 4). (4) HyperBlocker$_{\text{noHO}}$ that disables all hardware optimizations (Section 5). We also compared more designated variants in Exp 3-4.

Rules. We mined MDs using [70] and the number of MDs is shown in Table 2. We checked the MDs manually to ensure correctness.

Measurements. Following typical ER settings, we measured the performance of each method (blocker, matcher, or the combination of the two) in terms of the runtime and the F1-score, defined as F1-score = $\frac{2 \times \text{Prec} \times \text{Rec}}{\text{Prec} + \text{Rec}}$. Here Prec is the ratio of correctly identified tuple pairs to all identified pairs and Rec is the ratio of correctly identified tuple pairs to all pairs that refer to the same real-world entity. All methods aim to achieve high Rec, Prec and F1-scores. Following [73], we also report the candidate set size ratio (CSSR), defined as $\frac{|\text{Ca}(P)|}{|P| \times |P|}$, when comparing HyperBlocker with DeepBlocker, to show the portion of tuple pairs that require further comparison by the matcher, *i.e.,* the smaller the CSSR, the better the blocker.

Environment. We run experiments on a Ubuntu 20.04.1 LTS machine powered with 2 Intel Xeon Gold 6148 CPU @ 2.40GHz, 4TB Intel P4600 PCIe NVMe SSD, 128GB memory, and 8 Nvidia Tesla V100 GPUs with the widely adopted hybrid cube-mesh topology (see more in [61]). The programs were compiled with CUDA-11.0 and GCC 7.3.0 with -O3 compiler. DisDedup, SparkER, and Dedoop were run on a cluster of 30 HPC servers, powered with 2.40GHz Intel Xeon Gold CPU, 4TB Intel P4600 SSD, 128GB memory.

Default parameters. Unless stated explicitly, we used the following parameters, which are best-tuned on each dataset (*i.e.,* a parameter may be set differently on different datasets). The maximum number of predicates in an MD is 10. The number m of data partitions is given in Table 2. The sizes of intervals and sliding windows, namely n_t and n_w, are 256 and 1024, respectively. For the offline model $\mathcal{N}$, we adopted a regression model with 3 hidden layers, with 2, 6, and 1 neurons, respectively. We used ReLU [59] as the activation function and Adam [41] as the optimizer. We used one GPU by default.

Experimental results. For lack of space, we report our findings on some datasets as follows; consistent on others datasets (more in [7]).

Exp-1: Motivation Study. We motivate our study by comparing HyperBlocker, our rule-based blocker, with the SOTA DL-based

Table 4: Accuracy & runtime on benchmarks where "*" denotes that integrating HyperBlocker with Ditto does not improve the F1-score and thus we report the result of HyperBlocker, and "/" denotes that the F1-score cannot be computed within 3 hours.

Method	Backend	Category	DBLP-ACM		IMDB		Songs		NCV	
			F1-score	Time (s)	F1-score	Time (s)	F1-score	Time (s)	F1-score	Time (s)
SparkER	CPU	Blocker	0.77 (-0.17)	11.0 (13.8×)	0.31 (-0.65)	242.9 (6.8×)	0.08 (-0.72)	203.4 (15.2×)	0.26 (-0.66)	229.3 (49.8×)
GPUDet	GPU	Blocker	0.92 (-0.02)	20.1 (25.1×)	0.94 (-0.02)	323.8 (9.1×)	0.80 (+0)	404.8 (30.2×)	0.90 (-0.02)	1252.6 (272.3×)
DeepBlocker	GPU	Blocker	0.98 (+0.04)	8.3 (10.4×)	/	>3h	/	>3h	/	>3h
$\text{HyperBlocker}_{\text{noEPG}}$	GPU	Blocker	0.94 (+0)	9.9 (12.4×)	/	>3h	0.80 (+0)	1904.1 (142×)	0.92 (+0)	2408.6 (523.6×)
$\text{HyperBlocker}_{\text{noHO}}$	GPU	Blocker	0.94 (+0)	9.5 (11.9×)	0.96 (+0)	472.6 (13.2×)	0.80 (+0)	45.0 (3.4×)	0.92 (+0)	35.9 (7.8×)
HyperBlocker	GPU	Blocker	0.94	0.8	0.96	35.7	0.80	13.4	0.92	4.6
Dedoop	CPU	Blocker+Matcher	0.90 (-0.08)	59.4 (9.4×)	0.67 (-0.29)	534.0 (15.0×)	0.80 (-0.08)	7643.4 (6.5×)	/	>3h
DisDedup	CPU	Blocker+Matcher	0.45 (-0.53)	94.0 (14.9×)	0.67 (-0.29)	644.0 (18.0×)	0.06 (-0.82)	917.0 (0.8×)	/	>3h
$\text{Ditto}_{\text{top2}}$	GPU	Blocker+Matcher	0.98 (+0)	9.0 (1.4×)	0.79 (-0.17)	6741.2 (188.8×)	0.88 (+0)	2308.6 (2.0×)	0.97 (+0.03)	381.8 (2.1×)
$\text{DeepBlocker}_{\text{Ditto}}$	GPU	Blocker+Matcher	0.99 (+0.01)	12.4 (2.0×)	/	>3h	/	>3h	/	>3h
$\text{HyperBlocker}_{\text{Ditto}}$	GPU	Blocker+Matcher	0.98	6.3	*0.96	*35.7	0.88	1179.0	0.94	180.6

blocker DeepBlocker (Table 3), where the bracket next to a metric of DeepBlocker gives its difference or improvement factor to ours.

DL-based blocking vs. rule-based blocking. We report recall, CSSR, runtime, and (both host and device) memory for both methods. Consistent with [73], for DeepBlocker, each tuple was paired with the top-K similar tuples to form initial candidate pairs, where $K = 5$ on all datasets (except DBLP-Scholar where $K = 150$). As remarked in Section 1, both have strengths. (1) HyperBlocker effectively reduces the number of pairs to further compare while maintaining high Rec (>93%), *e.g.,* its average CSSR is 5.8‱ less than DeepBlocker. (2) HyperBlocker is at least 10× faster. (3) HyperBlocker consumes less memory than DeepBlocker, *e.g.,* the host memory it consumes is at least 23.3× less than DeepBlocker. (4) Note that the Rec of HyperBlocker is slightly lower than DeepBlocker, which is acceptable given its convincing speedup and memory saving, since the primary goal of a blocker is to improve the efficiency and scalability of ER, not to improve the accuracy of ER (the goal of a matcher).

Exp-2: Accuracy-efficiency. We first report the F1-scores and runtime of all blockers and integrated ER solutions (*i.e.,* blocker + matcher) in Table 4. Here DeepBlocker pairs each tuple with its top-2 tuples as initial candidate pairs. For all blockers, the bracket next to each F1-score (resp. time) gives the difference (resp. speedup) in F1-score (resp. time) to HyperBlocker (marked yellow). For a fair comparison, the brackets of each integrated ER solution, give the improvement compared with $\text{HyperBlocker}_{\text{Ditto}}$ (marked yellow).

Accuracy. We mainly analyze the F1-scores of HyperBlocker, which are consistently above 0.8 over all datasets. Besides, we find:

(1) HyperBlocker outperforms CPU-based distributed solutions, *e.g.,* it achieves up to 0.29, 0.74, and 0.72 improvement in F1-score against Dedoop, DisDedup, and SparkER, respectively, even though the former two are integrated with matchers. This is because these existing distributed solutions exploit data partition-based parallelism only, which may lead to potential false negatives (see Section 3).

(2) Compared with the four GPU-based baselines, HyperBlocker has comparable accuracy. In particular, it even beats $\text{Ditto}_{\text{top2}}$, the SOTA matcher, by 0.17 F1-score in IMDB. This shows that even without a matcher, HyperBlocker alone is already accurate in certain cases. Moreover, DeepBlocker and $\text{DeepBlocker}_{\text{Ditto}}$ struggle to handle large datasets. When facing million-scale data, they cannot finish in 3 hours. This again motivates the need for rule-based alternatives.

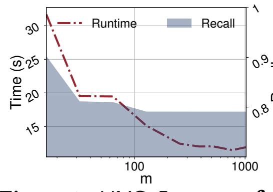


Figure 7: NVC: **Impact of** m

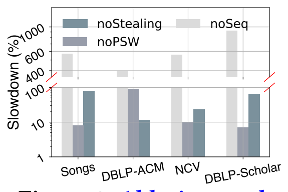


Figure 8: Ablation study

(3) Combing HyperBlocker with Ditto, $\text{HyperBlocker}_{\text{Ditto}}$ further boosts the accuracy, achieving the best F1-score in Songs. Nevertheless, DL-based solutions still have the best F1-scores in other cases, justifying that none of them can dominate the other in all cases.

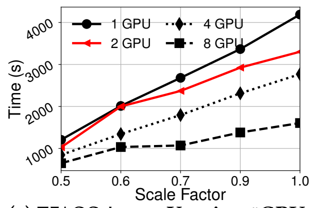
(4) $\text{HyperBlocker}_{\text{noEPG}}$ and $\text{HyperBlocker}_{\text{noHO}}$ are as accurate as HyperBlocker, since they only differ in the optimizations.

Runtime. We next report the runtime. (1) HyperBlocker runs substantially faster than all baselines, *e.g.,* it is at least 6.8×, 9.1×, 10.4×, 15.0×, 18.0×, 11.3× and 15.5× faster than SparkER, GPUDet, DeepBlocker, Dedoop, DisDedup, Ditto, and $\text{DeepBlocker}_{\text{Ditto}}$ respectively. (2) $\text{HyperBlocker}_{\text{Ditto}}$ is slower than HyperBlocker as expected since it performs additional matching. Nonetheless, $\text{HyperBlocker}_{\text{Ditto}}$ is at least 1.4× (resp. 2.0×) faster than $\text{Ditto}_{\text{top2}}$ (resp. $\text{DeepBlocker}_{\text{Ditto}}$). Given its comparable F1-score, we substantiate our claim (Section 1) that blocking is a crucial part of the overall ER process. (3) HyperBlocker is at least 12.4× and 3.4× faster than $\text{HyperBlocker}_{\text{noEPG}}$ and $\text{HyperBlocker}_{\text{noHO}}$, respectively, verifying the usefulness of execution plans and hardware optimizations.

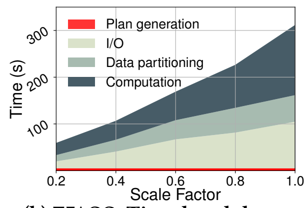
Impact of m. Figure 7 reports how the number m of data partitions affects the recall (the right y-axis) and the runtime (the left y-axis) on NVC. As shown there, both metrics of HyperBlocker decreases with increasing m. This is because when there are more partitions, both the number of pairwise comparisons and the candidate matches that can be identified in each partition are reduced.

Exp-3: Scalability. We tested our scalability under multi-GPUs scenarios. The default number of GPUs is 4 in this set of experiments.

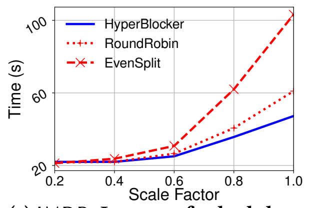
Varying |D|/#GPUs. We varied the scale factor of D in $\text{TFACC}_{\text{large}}$ and tested HyperBlocker with different numbers of GPUs in Figure 9(a). HyperBlocker scales well with data sizes, *e.g.,* with 8 GPUs, it takes 1604s to process 36M tuples; this is not feasible for both CPU- and GPU-based baselines. When the number of GPUs changes from 1 to 8, HyperBlocker is 2.6× faster, since HyperBlocker mainly accelerates the operations on GPUs, while other parts of the system (*e.g.,*



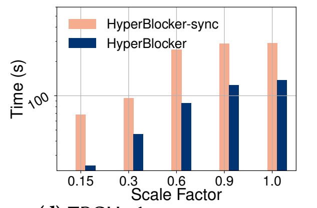
(a) TFACC-Large: **Varying #GPUs**



(b) TFACC: **Time breakdown**

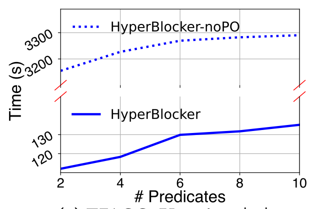


(c) IMDB: **Impact of schedulers**

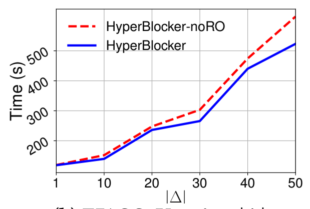


(d) TPCH: **Async. vs sync.**

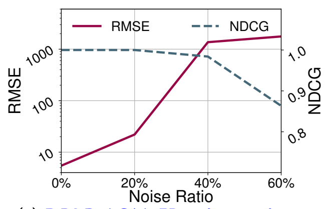
Figure 9: Scalability test of HyperBlocker



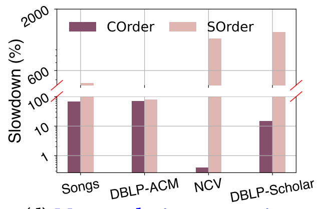
(a) TFACC: **Varying** $|\varphi|$



(b) TFACC: **Varying** $|\Delta|$



(c) DBLP-ACM: **Varying noise%**



(d) **More ordering strategies**

Figure 10: Experiments on EPG

I/O and data partitioning) may also limit the overall performance.

Breakdown. We break down the time of HyperBlocker on TFACC into 4 parts in Figure 9(b): (1) I/O, which loads data from disk to host memory; (2) data partition, which partitions data; (3) plan generation, which constructs the execution plan; (4) computation, that transfers data from the host to devices and computes matches. As shown there, plan generation (the bottom red line) is fast, and GPU operations do not dominate, consistent with findings in [35].

Impact of task schedulers. We tested the impact of task schedulers, by comparing HyperBlocker with two variants, that uses EvenSplit and RoundRobin for scheduling (Section 5.3), respectively. We varied the dataset size from 20% to 100% in IMDB in Figure 9(c). HyperBlocker works better than the two, *e.g.,* when the scale factor is 100%, HyperBlocker is 1.3× and 2.2× faster than RoundRobin and EvenSplit, respectively, since both variants may limit CUDA's ability of dynamically scheduling tasks. HyperBlocker also shows better scalability, whereas the others exhibit a steep rise in time.

Asynchronous vs. synchronous. The (asynchronous) pipelined architecture (Section 3) plays a pivotal role in HyperBlocker. To justify this, we compared HyperBlocker with HyperBlocker$_{\text{sync}}$, a variant without this architecture, *i.e.,* HyperBlocker$_{\text{sync}}$ initiates the next task after the prior one is fully submitted to devices, in Figure 9(d). HyperBlocker is at least 2.1× faster than HyperBlocker$_{\text{sync}}$, due to the synchronization cost. In contrast, by asynchronously overlapping the execution at devices and the data transfer in HyperBlocker, we reduce both idle time and unnecessary waiting.

Exp-4: Tests on EPG (Section 4). We evaluated EPG (and its offline model $\mathcal{N}$) and justified the need of effective evaluation orders.

Varying $|\varphi|$. We tested the number $|\varphi|$ of predicates in each MD φ against HyperBlocker$_{\text{noPO}}$ that evaluates predicates in a random order in TFACC (Figure 10(a)). (1) HyperBlocker takes longer with larger $|\varphi|$, as expected. (2) HyperBlocker is feasible in practice, *e.g.,* when $|\varphi| = 10$, it only takes 135.2s. (3) On average, HyperBlocker shows 32.5× speedup to HyperBlocker$_{\text{noPO}}$. This justifies the importance of predicate ordering in efficient rule-based blocking.

Varying $|\Delta|$. We evaluated the impact of the number $|\Delta|$ of MDs in Δ in Figure 10(b), where HyperBlocker takes longer with more rules, *e.g.,* it takes 523.2s when $|\Delta| = 50$, and consistently beats HyperBlocker$_{\text{noRO}}$, a variant that evaluates rules in a random order.

Shallow model $\mathcal{N}$. We evaluated the performance of $\mathcal{N}$ in EPG by (1) its sensitivity to noises, (1) the resulting predicate ordering, compared with the "ground truth" ordering derived from actual costs, and (3) the speedup of estimating the actual costs using $\mathcal{N}$.

(1) Given a noise ratio $\beta\%$, we injected $\beta\%$ noises to training data of $\mathcal{N}$, to disturb its distribution, and report RMSE (Root Mean Squared Error), a widely used metric for regression, in Figure 10(c) (the left y-axis). The RMSE of $\mathcal{N}$ does not degrade much when $\beta\% = 20\%$. However, when $\beta\%$ continues to increase, $\mathcal{N}$ becomes inaccurate. A case study about resulting orderings under different $\beta\%$ is given in [7].

(2) We compared the predicate ordering estimated via $\mathcal{N}$ with the ground truth one using NDCG (Normalized Discounted Cumulative Gain [74]), a widely used metric for evaluating ranking, in Figure 10(c) (the right y-axis). The result shows that the two orderings are close (*i.e.,* NDCG is high), even when the noise ratio is 40%.

(3) The average time for computing the actual cost of a predicate is 0.8s on DBLP-ACM, as opposed to 0.007s by $\mathcal{N}$ (see more in [7]).

More ordering strategies. To justify the needs of both the cost and effectiveness in ordering, we compared two more strategies using designated MDs: (1) COrder, that prioritizes cheap predicates (*e.g.,* always evaluate equality first, Section 4) and (2) SOrder, that prioritizes selective predicates. To better visualize the effects on datasets with different scales, we report their slowdown percentages (instead of the runtime) in Figure 10(d). SOrder (resp. COrder) is on average slowed by 733.6% (resp. 38.2%) compared with HyperBlocker. This said, HyperBlocker strikes a balance between the two strategies.

Exp-4: Tests on hardware optimizations (Section 5). Finally, we conducted an ablation study on hardware optimizations and report the runtime statistics. We compared three baselines: (1) noSeq, that recursively implements DFS without sequential execution paths, (2) noPSW that assigns continuous intervals to each TB without parallel sliding windows, and (3) noStealing, where GPUs automatically schedule a new TB whenever one is done, without task stealing. To better visualize the effect, below we used D as a single partition.

Ablation study. We show the slowdown percentages compared to

Table 5: Runtime info (↑: higher is better vs. ↓: lower is better)

Method	↓ Wait stalls (in terms of clock cycles)	↑ Branch efficiency	↑ Average number of active threads per warp
noSeq	4.25	89.9%	14.45
noPSW	13.79	96.3%	25.59
noStealing	4.11	96.2%	27.62
HyperBlocker	4.07	96.4%	28.21

HyperBlocker in Figure 8. We find: (1) noSeq is much slower than HyperBlocker, since recursive DFS is not efficient on GPUs. (2) noStealing and noPSW are on average 43.1% and 28.8% slower than HyperBlocker, respectively, justifying the use of both optimizations.

Runtime statistic. We adopted NSight [11], a profiling tool provided by NVIDIA, and report *wait stalls* (*i.e.*, the number of clock cycles that the kernel spent on waiting), *branch efficiency* (*i.e.*, the ratio of correctly predicted branch instructions), and the *average number active threads per warp* in TFACC (Table 5). HyperBlocker performs the best in all metrics. The reasons are twofold: (1) while divergence is sometimes unavoidable, a recursive DFS exacerbates it (*e.g.*, due to stacking), leading to more idle threads; and (2) the workloads can be imbalanced, *e.g.*, without parallel sliding windows, noPSW incurs a larger number of wait stalls compared with HyperBlocker.

Summary. We find the following. (1) HyperBlocker outperforms prior blockers and integrated ER solutions, with its novel pipelined architecture, execution plan, and hardware-aware optimizations on GPUs. It is at least 6.8×, 9.1×, 10.4×, 15.0×, 18.0×, 11.3× and 15.5× faster than SparkER, GPUDet, DeepBlocker, Dedoop, DisDedup, Ditto, and DeepBlocker$_{\text{Ditto}}$ respectively. (2) By combining HyperBlocker with Ditto, we save at least 30% of time with comparable accuracy. (3) HyperBlocker beats all its variants (except HyperBlocker$_{\text{Ditto}}$) in both runtime and accuracy, justifying the usefulness of various optimizations used: (a) EPG specifies an effective evaluation order, improving the runtime by at least 12.4× and (b) the hardware-level optimizations on GPUs speedups blocking by at least 3.4×. (3) HyperBlocker scales well with various parameters, *e.g.*, it completes blocking in 1604s on TFACC$_{\text{large}}$ with 36M tuples.

7 RELATED WORK

We categorize the related work in the literature as follows.

Blocking algorithms. There has been a host of work on the blocking algorithms, classified as follows: (1) Rule-based [20, 34, 37, 42, 63], *e.g.*, [34] creates data partitions and then refines candidate pairs in every partition, by removing mismatches with similarity measures or length/count filtering [54]. (2) DL-based [24, 38, 73, 76], which cast the generation of candidate matches into a binary classification problem, where each tuple pair is labeled "likely match" or "unlikely match", *e.g.*, [73] adopts similarity search to generate candidate matches for each tuple based on its top-K probable matches in an embedding space. DL-based blocking and rule-based blocking share the same goal, but are different in their approaches, where the former focuses on learning the distributed representations of tuples, while the latter emphasizes explicit logical reasoning.

Although we study rule-based blocking, we are not to develop another blocking algorithm. Instead, we provide a GPU-accelerated blocking solution. As a testbed, we use MDs as our blocking rules, which subsume many existing rules [44, 63] as special cases.

Parallel blocking solvers. Several parallel blocking systems have been proposed, *e.g.*, [15, 20, 21, 26, 30, 32, 42, 43, 65, 72], mostly under MapReduce [20, 32, 42] or MPC [22, 30, 72], which aim at scaling to large data with a cluster of machines. DisDedup [20] uses a triangle distribution strategy to minimize both comparisons and communication over Spark[12]. Minoan [26] runs on top of Spark and applies parallel meta blocking [25] to minimize its overall runtime.

This work differs as follows. Unlike MapReduce-based systems, which split data at the coordinator and execute tasks on workers, HyperBlocker focuses on collaborating GPUs and CPUs, to promote better resource utilization. HyperBlocker is designed for the shared memory architecture of GPUs and is fine-tuned to exploit GPU hardware for rule-based blocking. To the best of our knowledge, incorporating both the variety of GPU and CPU characteristics has not been considered in prior parallel blocking solutions.

GPU-accelerated techniques. GPUs have been used extensively to speed up the training of DL tasks. Recent works exploit GPUs to accelerate data processing, *e.g.*, GPU-based query answering [23, 36, 69] and similarity join [39, 52, 60]. Closer to this work are [39, 52] which leverage GPUs for similarity join, since blocking can be regarded as a similarity join problem under the assumption that two tuples refer to the same entity if their similarity is high. Similarity join is often served as a preprocessing step of ER.

In contrast, HyperBlocker aims at expediting rule-based blocking, addressing challenges in rule-based optimization that are not incurred in similarity join. The closest work is GPUDet [31], which employs GPUs to expedite similarity measures. HyperBlocker differs from GPUDet, in its data/rule-aware execution plan designated for rule evaluation, beyond similarity measures. It also incorporates hardware-aware optimizations for improving GPU utilization.

Query optimizations. Also related are prior methods for query optimization in RDBMS [40, 47, 50, 55, 57, 67], which utilize either sampling, statistics, or profiling to generate query execution plans, via cost estimation and cardinality estimation. Although EPG in HyperBlocker shares a similar goal, it aims to provide a lightweight solution for efficient rule-based blocking, where it suffices to find one witness from a set of rules, instead of minimizing the cost of executing one or multiple SQL queries, which can be measured by the memory consumption, CPU utilization, I/O operations or the size of intermediate results, which are not the main focus of blocking.

8 CONCLUSION

The novelty of HyperBlocker consists of (1) a pipelined architecture that overlaps the data transfer from/to CPUs and the operations on GPUs; (2) a data-aware and rule-aware execution plan generator on CPUs, that specifies how rules are evaluated; (3) a variety of hardware-aware optimization strategies that achieve massive parallelism, by exploiting GPU characteristics; and (4) partitioning and scheduling strategies to achieve workload balancing across multiple GPUs. Our experimental study has verified that HyperBlocker is much faster than existing CPU-powered distributed systems and GPU-based ER solvers, while maintaining comparable accuracy.

One future topic is to study how to accelerate the evaluation of more sophisticated rules, *e.g.*, REEs [28, 29] which are multi-variable rules defined across multiple relations, using GPUs. Another opportunity is to design a different execution plan on each partition.

REFERENCES

[1] 2021. Dedoop Source Code. https://dbs.uni-leipzig.de/dedoop.
[2] 2021. Ditto Source Code. https://github.com/megagonlabs/ditto.
[3] 2021. ER Benchmark Dataset. https://dbs.uni-leipzig.de/de/research/projects/object_matching/benchmark_datasets_for_entity_resolution.
[4] 2021. Magellan Dataset. https://sites.google.com/site/anhaidgroup/projects/data.
[5] 2021. TPCH. http://www.tpc.org/tpch/.
[6] 2023. Amazon Duplicate Product Listings. https://www.amazowl.com/amazon-frustration-free-packaging-2-2-2/.
[7] 2024. Code, datasets and full version. https://github.com/SICS-Fundamental-Research-Center/HyperBlocker.
[8] 2024. CUDA C Programming Guide. https://docs.nvidia.com/cuda/cuda-c-programming-guide/.
[9] 2024. DisDedup Source Code. https://github.com/david-siqi-liu/sparklyclean.
[10] 2024. MOT Tests and Results. https://ckan.publishing.service.gov.uk/dataset.
[11] 2024. NSight Compute Documentation. https://docs.nvidia.com/nsight-compute/.
[12] 2024. Spark. https://spark.apache.org.
[13] 2024. Sparker Source Code. https://github.com/Gaglia88/sparker.
[14] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley.
[15] Yasser Altowim and Sharad Mehrotra. 2017. Parallel Progressive Approach to Entity Resolution Using MapReduce. In *ICDE*.
[16] Alexandr Andoni and Piotr Indyk. 2008. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM* 51, 1 (2008).
[17] Nils Barlaug. 2023. ShallowBlocker: Improving Set Similarity Joins for Blocking. *arXiv preprint arXiv:2312.15835* (2023).
[18] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomás Mikolov. 2017. Enriching Word Vectors with Subword Information. *Trans. Assoc. Comput. Linguistics* 5 (2017), 135–146.
[19] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An Efficient and Flexible Graph Mining System on CPU and GPU. *PVLDB* (2020).
[20] Xu Chu, Ihab F Ilyas, and Paraschos Koutris. 2016. Distributed data deduplication. *PVLDB* 9, 11 (2016), 864–875.
[21] Sanjib Das, Paul Suganthan GC, AnHai Doan, Jeffrey F Naughton, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, Vijay Raghavendra, and Youngchoon Park. 2017. Falcon: Scaling up hands-off crowdsourced entity matching to build cloud services. In *SIGMOD*. 1431–1446.
[22] Ting Deng, Wenfei Fan, Ping Lu, Xiaomeng Luo, Xiaoke Zhu, and Wanhe An. 2022. Deep and collective entity resolution in parallel. In *ICDE*. 2060–2072.
[23] Gregory Frederick Diamos, Haicheng Wu, Jin Wang, Ashwin Sanjay Lele, and Sudhakar Yalamanchili. 2013. In *PPoPP*. 301–302.
[24] 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.
[25] Vasilis Efthymiou, George Papadakis, George Papastefanatos, Kostas Stefanidis, and Themis Palpanas. 2015. Parallel meta-blocking: Realizing scalable entity resolution over large, heterogeneous data. In *IEEE Big Data*. 411–420.
[26] Vasilis Efthymiou, George Papadakis, Kostas Stefanidis, and Vassilis Christophides. 2019. MinoanER: Schema-Agnostic, Non-Iterative, Massively Parallel Resolution of Web Entities. In *EDBT*.
[27] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *The VLDB Journal* 20 (2011), 495–520.
[28] Wenfei Fan, Ziyan Han, Yaoshu Wang, and Min Xie. 2022. Parallel Rule Discovery from Large Datasets by Sampling. In *SIGMOD*. 384–398.
[29] Wenfei Fan, Ziyan Han, Yaoshu Wang, and Min Xie. 2023. Discovering Top-k Rules using Subjective and Objective Criteria. In *SIGMOD*.
[30] Wenfei Fan, Chao Tian, Yanghao Wang, and Qiang Yin. 2021. Parallel discrepancy detection and incremental detection. *PVLDB* 14, 8 (2021), 1351–1364.
[31] Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. 2013. Duplicate detection on GPUs. *HPI Future SOC Lab* 70, 3 (2013).
[32] Luca Gagliardelli, Giovanni Simonini, Domenico Beneventano, and Sonia Bergamaschi. 2019. SparkER: Scaling Entity Resolution in Spark. In *EDBT* .
[33] Lei Gao, Pengpeng Zhao, Victor S. Sheng, Zhixu Li, An Liu, Jian Wu, and Zhiming Cui. 2015. EPEMS: An Entity Matching System for E-Commerce Products. In *Web Technologies and Applications*, Reynold Cheng, Bin Cui, Zhenjie Zhang, Ruichu Cai, and Jia Xu (Eds.). Springer International Publishing, Cham, 871–874.
[34] Lifang Gu and Rohan Baxter. 2004. Adaptive filtering for efficient record linkage. In *SDM*. 477–481.
[35] Wentian Guo, Yuchen Li, Mo Sha, Bingsheng He, Xiaokui Xiao, and Kian-Lee Tan. 2020. GPU-Accelerated Subgraph Enumeration on Partitioned Graphs. In *SIGMOD*.
[36] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K Govindaraju, Qiong Luo, and Pedro V Sander. 2009. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)* 34, 4 (2009), 1–39.
[37] Robert Isele, Anja Jentzsch, and Christian Bizer. 2011. Efficient multidimensional blocking for link discovery without losing recall.. In *WebDB*. 1–6.
[38] Delaram Javdani, Hossein Rahmani, Milad Allahgholi, and Fatemeh Karimkhani. 2019. Deepblock: A novel blocking approach for entity resolution using deep learning. In *ICWR*. 41–44.
[39] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2021. Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data (TBD)* 7, 3 (2021), 535–547.
[40] Tarun Kathuria and S Sudarshan. 2017. Efficient and provable multi-query optimization. In *PODS*. 53–67.
[41] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR* .
[42] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Dedoop: Efficient Deduplication with Hadoop. *PVLDB* 5, 12 (2012), 1878–1881.
[43] Lars Kolb, Andreas Thor, and Erhard Rahm. 2012. Load balancing for MapReduce-based entity resolution. In *ICDE*.
[44] Pradap Konda, Sanjib Das, AnHai Doan, Adel Ardalan, Jeffrey R Ballard, Han Li, Fatemah Panahi, Haojun Zhang, Jeff Naughton, Shishir Prasad, et al. 2016. Magellan: toward building entity matching management systems over data science stacks. *PVLDB* 9, 13 (2016), 1581–1584.
[45] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *SIGMOD*. 489–504.
[46] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale graph computation on just a PC. In *USENIX OSDI*. 31–46.
[47] Per-Ake Larson, Wolfgang Lehner, Jingren Zhou, and Peter Zabback. 2007. Cardinality estimation using sample views with quality assurance. In *SIGMOD*. 175–186.
[48] Ang Li, Shuaiwen Leon Song, Jieyang Chen, Jiajia Li, Xu Liu, Nathan R Tallent, and Kevin J Barker. 2019. Evaluating modern gpu interconnect: Pcie, nvlink, nv-sli, nvswitch and gpudirect. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 31, 1 (2019), 94–110.
[49] Bo-Han Li, Yi Liu, An-Man Zhang, Wen-Huan Wang, and Shuo Wan. 2020. A survey on blocking technology of entity resolution. *Journal of Computer Science and Technology* 35 (2020), 769–793.
[50] Guoliang Li, Jian He, Dong Deng, and Jian Li. 2015. Efficient similarity join and search on multi-attribute data. In *SIGMOD*. 1137–1151.
[51] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *PVLDB* 14, 1 (2020), 50–60.
[52] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. 2008. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*, Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen (Eds.). 1111–1120.
[53] Jonas Lippuner. 2019. *NVIDIA CUDA*. Technical Report. Los Alamos National Lab.(LANL), Los Alamos, NM (United States).
[54] Willi Mann and Nikolaus Augsten. 2014. PEL: Position-Enhanced Length Filter for Set Similarity Joins. In *Grundlagen von Datenbanken*.
[55] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul23. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019).
[56] Vahab S. Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent Hashing with Bounded Loads. In *SODA* . 587–604.
[57] Guido Moerkotte. [n.d.]. Building query compilers. ([n. d.]).
[58] 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*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.).
[59] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In *ICML*. 807–814.
[60] Negin Nematollahi, Mohammad Sadrosadati, Hajar Falahati, Marzieh Barkhordar, Mario Paulo Drumond, Hamid Sarbazi-Azad, and Babak Falsafi. 2020. Efficient nearest-neighbor data sharing in GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)* 18, 1 (2020), 1–26.
[61] NVIDIA. 2024. NVIDIA V100 TENSOR CORE GPU. https://www.nvidia.com/en-us/data-center/v100/.
[62] Santosh Pandey, Xiaoye Sherry Li, Aydin Buluc, Jiejun Xu, and Hang Liu. 2019. H-INDEX: Hash-Indexing for Parallel Triangle Counting on GPUs. In *IEEE High Performance Extreme Computing Conference*. 1–7.
[63] George Papadakis, Ekaterini Ioannou, Claudia Niederée, Themis Palpanas, and Wolfgang Nejdl. 2011. To compare or not to compare: making entity resolution more efficient. In *Proceedings of the international workshop on semantic web information management*. 1–7.
[64] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2020. Blocking and filtering techniques for entity resolution: A survey. *ACM Computing Surveys (CSUR)* 53, 2 (2020), 1–42.
[65] Vibhor Rastogi, Nilesh Dalvi, and Minos Garofalakis. 2011. Large-Scale Collective Entity Matching. *PVLDB* 4, 4 (2011).
[66] Ryan A Rossi and Rong Zhou. 2016. Leveraging multiple gpus and cpus for graphlet counting in large networks. In *CIKM*. 1783–1792.
[67] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*. 249–260.

[68] Giovanni Simonini, Sonia Bergamaschi, and HV Jagadish. 2016. BLAST: a loosely schema-aware meta-blocking approach for entity resolution. *PVLDB* 9, 12 (2016), 1173–1184.

[69] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *ICDE*. 698–709.

[70] Shaoxu Song and Lei Chen. 2013. Efficient discovery of similarity constraints for matching dependencies. *Data & Knowledge Engineering* 87 (2013), 146–166.

[71] Marshall Harvey Stone. 1937. Applications of the theory of Boolean rings to general topology. *Trans. Amer. Math. Soc.* 41, 3 (1937), 375–481.

[72] Yufei Tao. 2018. Massively Parallel Entity Matching with Linear Classification in Low Dimensional Space. In *ICDT*.

[73] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep learning for blocking in entity matching: a design space exploration. *PVLDB* 14, 11 (2021), 2459–2472.

[74] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, and Tie-Yan Liu. 2013. A theoretical analysis of NDCG type ranking measures. In *Conference on learning theory*. PMLR, 25–54.

[75] Li Zeng, Lei Zou, M Tamer Özsu, Lin Hu, and Fan Zhang. 2020. GSI: GPU-friendly subgraph isomorphism. In *ICDE*. 1249–1260.

[76] Wei Zhang, Hao Wei, Bunyamin Sisman, Xin Luna Dong, Christos Faloutsos, and Davd Page. 2020. Autoblock: A hands-off blocking framework for entity matching. In *WSDM*.

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

A NOTATION TABLE

We summarize the frequently used notations in Table 6.

Table 6: Summary of frequently used notations

Notations	Definition
R	A relation schema $(\text{eid}, A_1, \ldots, A_n)$
D/P	A set of tuples / a partition of D
m	The number of data partitions of D
$\text{Ca}(P)$	A set of potential matches from P
φ/Δ	A matching dependency (MD) / a set of MDs
$p/X/\mathcal{P}$	A predicate / a precondition / all predicates appeared in Δ
h and $h(t_1, t_2)$	A valuation of $\varphi : X \rightarrow l$ that instantiates variables t and s with tuples t_1 and t_2; φ is a witness at (t_1, t_2) if $h \models X$
$\mathcal{T}/N/\rho$	An execution plan (tree) / a node / a path of $\mathcal{T}$
$e/\text{score}(e)$	An edge (N_1, N_2) of $\mathcal{T}$ / the score of e
$\text{cost}(p, D)$ $(\hat{\text{cost}}(p, D))$	The (estimated) evaluation cost of p on D
$\text{sp}(p, D)$	The probability of p being satisfied on D
$\text{wp}(p, D)$	The probability of φ being a witness
n_t/n_w	The size of intervals / the size of window

B TRAINING OF SHALLOW MODEL

To train $\mathcal{N}$, we collect training data from historical logs, where each training instance is in form (x, y); here x is triplet (p, t_1, t_2) and y is its label, *i.e.,* the measured time of evaluating p at (t_1, t_2). We train $\mathcal{N}$ offline, using stochastic gradient descent with the mean square error loss. While the training time is not our focus, $\mathcal{N}$ converges quickly in a few pass [45], since it has few parameters.

C MORE OPTIMIZATIONS

HyperBlocker is implemented in 4K+ lines of C++/CUDA code with the pipelined architecture. It also offers the following optimizations.

C.1 Parallel write conflict

When thousands of threads on a GPU are executing in parallel, each thread may produce an uncertain number of results. Therefore, these threads will compete for GPU memory and it is hard for them to decide the position where they should write the results. This is called *parallel write conflict.* Many GPU-based systems, *e.g.,* Pangolin [19], address this by a costly two-round procedure, *i.e.,* they scan the threads twice to (a) record the number of results produced by each thread and (b) write results. Other methods, *e.g.,* GSI [75], estimate the maximum size of results of each thread for space pre-allocation, which, however, leads to low memory utilization.

To alleviate this, we maintain a local buffer for each TB at shared memory, so that only threads in this TB will compete for this buffer. Moreover, the local buffer is divided into two parts. Initially, all threads in a TB use their synchronization primitives to compute write offsets and write to the first part. When the first part is full, this TB will compete with other TBs and flush its buffered results to GPU memory, by computing the global offset with an atomic increment operation; meanwhile, all threads in this TB start to buffer in the second part of the local buffer, and the cycle repeats.

C.2 Data transfer between host and devices

It is widely recognized that data transfer between CPUs and GPUs is a bottleneck in CPUs/GPUs computation [35]. Fortunately, the use of CUDA streams allows to conduct data transfer and GPU execution simultaneously, to "cancel" the excessive data shipment cost.

Recall that relation D is divided into multiple partitions $P_1, \ldots,$ P_m. HyperBlocker extends [77] and iteratively processes partitions in a pipelined manner. Specifically, the out-of-device processing of each partition P is divided into three steps: (1) Read P into device memory. (2) Split P into intervals and process each interval. (3) Write the result $\text{Ca}(P)$ back to host memory. Then HyperBlocker conducts the evaluation on in-device partitions while loading pending ones from and writing results back to the host memory.

C.3 More choices for parallelism

There are indeed other design choices for parallelism:

(1) *Rule-level parallelism*, where the entire warp is assigned to each tuple pair and rules are evaluated in parallel, *i.e.,* each thread in the warp is responsible for the evaluation of a rule;

(2) *Predicate-level parallelism*, where the entire warp is assigned to each tuple pair and rules are evaluated sequentially. Within each rule, predicates are evaluated in parallel and each thread is responsible for evaluating a predicate. The warp moves to the next rule only after all predicates of the current rule are evaluated.

(3) *Token-level parallelism*, that only works for texts. Each text comparison utilizes multiple threads for parallel processing, *e.g.,* each thread compares the equality of a character/token of two texts.

However, for rule-level parallelism, there can be unnecessary computation, *e.g.,* even a thread finds a witness φ at (t_1, t_2), other threads may be still checking other rules in Δ, which is not needed, since one witness suffices to decide (t_1, t_2) makes a potential match.

Similarly, in predicate-level parallelism, even a thread finds a a predicate p in φ such that $h(t_1, t_2) \not\models p$, other threads may still be checking other predicates in φ, which is also not needed, since $h(t_1, t_2) \not\models p$ suffices to conclude that φ is not a witness at (t_1, t_2). Worse still, it treats all predicates equally potent for evaluation and may incur additional synchronization cost. Consider a rule $\varphi : X \rightarrow l$, where X consists of multiple predicates and each predicate is evaluated by a thread. Given a tuple pair (t_1, t_2), its evaluation times on different predicates can be different. In other words, even a thread finds a predicate p in φ such that $h(t_1, t_2) \models p$, it has to wait for results from other threads to make a final decision, since φ is a witness at (t_1, t_2) only if $h(t_1, t_2)$ satisfies all predicates in X. As an evidence, when testing on a dataset with 6M tuple pairs, tuple-level parallelism is 7.3× faster than predicate-level one.

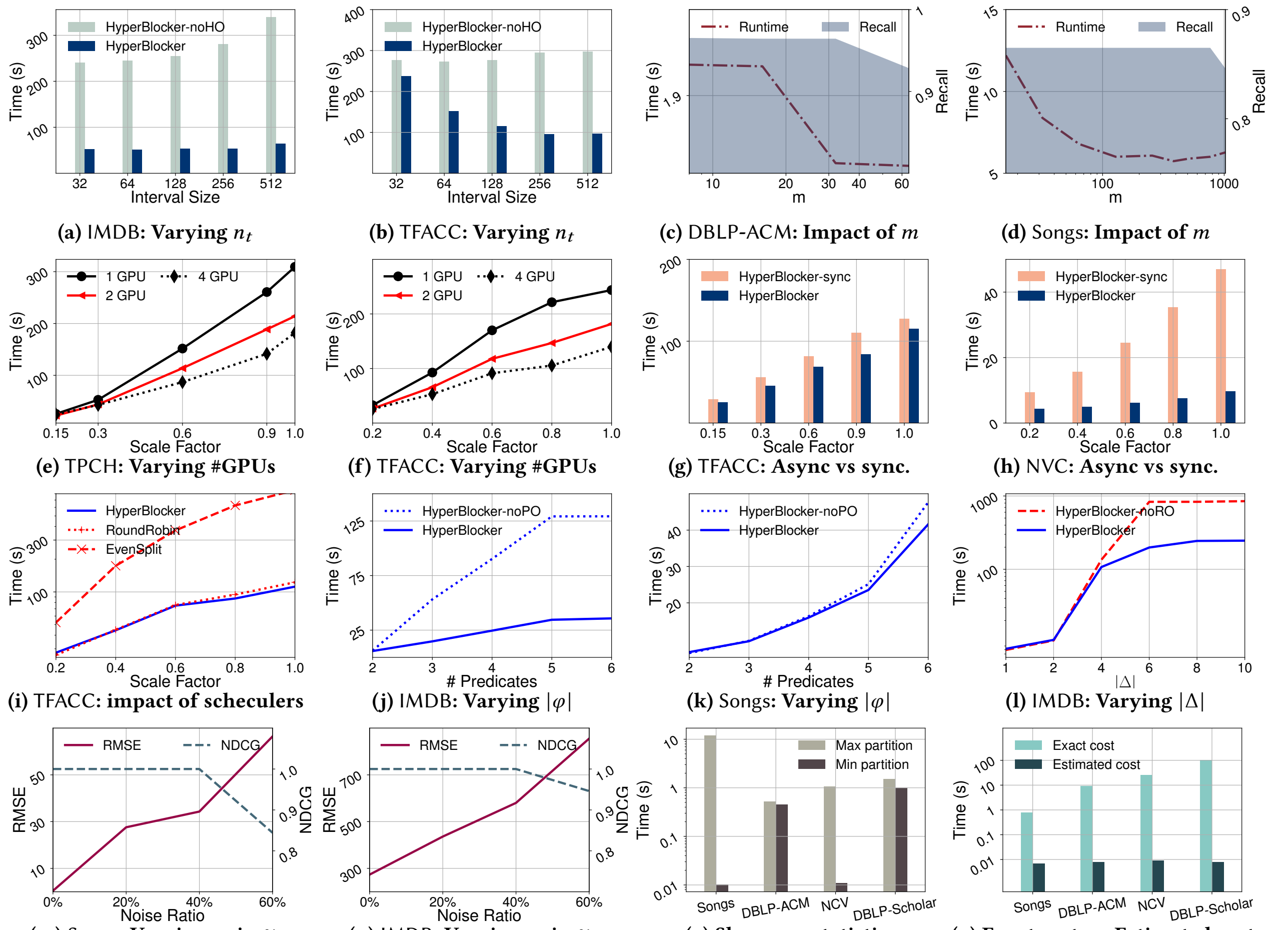
Token-level parallelism also inherits the synchronization issues and worse still, it only works for textual attributes.

D ADDITIONAL EXPERIMENTS

Below we report additional experimental results.

Comparison with similarity join. Recall that Faiss often serves as a key component of many ER solutions [24, 73], we compared Faiss with HyperBlocker as follows. We sampled 50,863 tuples from Songs, where HyperBlocker conducts rule-based blocking by taking these tuples as a single partition, while Faiss converts each tuple to an embedding using FastText [18] and gets its top-K nearest neighbors, varying K from 1 to 100. As shown in Table 7, HyperBlocker is more accurate and 3.65× faster than Faiss on average.

Varying n_t. In Figures 11(a)-(b), we tested the impact of interval size n_t on HyperBlocker and HyperBlocker$_{\text{noHO}}$, that disables all



(a) IMDB: **Varying n_t** **(b)** TFACC: **Varying n_t** **(c)** DBLP-ACM: **Impact of m** **(d)** Songs: **Impact of m**

(e) TPCH: **Varying #GPUs** **(f)** TFACC: **Varying #GPUs** **(g)** TFACC: **Async vs sync.** **(h)** NVC: **Async vs sync.**

(i) TFACC: **impact of scheculers** **(j)** IMDB: **Varying $|\varphi|$** **(k)** Songs: **Varying $|\varphi|$** **(l)** IMDB: **Varying $|\Delta|$**

(m) Songs: **Varying noise%** **(n)** IMDB: **Varying noise%** **(o) Skewness statistic** **(p) Exact cost vs. Estimated cost**

Figure 11: Additional experimental results

Table 7: Faiss **vs.** HyperBlocker

Method	Metric	Top-1	Top-10	Top-50	Top-100
Faiss	Prec (%)	52.0 (+44.6)	69.0 (-27.6)	73.6 (-23.0)	76.0 (-20.6)
	Time (s)	7.65 (3.34×)	8.58 (3.74×)	8.58 (3.74×)	8.64 (3.77×)
HyperBlocker	Prec (%)	96.6			
	Time (s)	2.29			

hardware-level optimizations (Section 5), varying n_t from 32 to 512. As shown there, on TFACC, HyperBlocker achieves the best performance when the interval size is 256, striking a balance between task granularity and parallelism. HyperBlocker runs up to 3.1× faster than HyperBlocker$_{\text{noHO}}$, verifying the usefulness of hardware optimizations on GPUs (Section 5). The results on IMDB are consistent.

Experiments with more datasets. Figures 11(c)-(n) report more experimental results that are not reported in Section 6, due to the space limit. These experiments were conducted under the same settings as their counter part in Section 6, using different datasets. The trends are consistent and thus, the detailed analysis is omitted.

Skewness on partitions. Figure 11(o) reports the maximum and minimum execution times over all partitions using the default numbers of partitions (Table 2). As shown there, workloads can be imbalanced due to skewed execution times of partitions, *e.g.*, the minimum execution time of a partition on Songs is 0.01s, as opposed to 11.91s for the maximum one, leading to performance degradation. The reason for the skewed execution times are twofold: there can be skewness in the sizes of partitions and moreover, the complexity/execution depends on the underlying data, *e.g.*, partitions containing long texts typically require more time compared to those with short texts. To address this, HyperBlocker not only adopts a collaborative approach integrating effective partitioning and scheduling (Section 5.3), but also advocates asynchronous processing (see Figure 9(d)), to improve resource utilization and overall throughput.

Time comparison for computing the exact vs. estimated evaluation cost. We compared the time for computing the exact and estimated evaluation costs, denoted by $\text{cost}(p, D)$ and $\hat{\text{cost}}(p, D)$, respectively, in Figure 11(p). Here $\text{cost}(p, D)$ is computed by summing up the actual times for evaluating a predicate p overall tuple pairs in D, while $\hat{\text{cost}}(p, D)$ is computed by replacing the actual times by the inference times of the shallow model $\mathcal{N}$. We compute $\text{cost}(p, D)$ and $\hat{\text{cost}}(p, D)$ by considering all predicates that appeared in the default rule set Δ and report the average computational time per predicate.

As shown there, computing the estimated evaluation cost is much faster than computing the exact evaluation cost, *e.g.*, on NCV, it only takes 0.008s to compute $\hat{\text{cost}}(p, D)$ on average, as opposed to 104.3s for $\text{cost}(p, D)$. Given the closeness of predicate orderings derived from both $\text{cost}(p, D)$ and $\hat{\text{cost}}(p, D)$ (see Figure 10(c)), we justify the necessity of the use of $\mathcal{N}$ to achieve speedup.

Case study on predicate orderings. Recall that in Figure 10(c), we report the performance of $\mathcal{N}$ by varying the noise ratio β%, and we show that with more noises, $\mathcal{N}$ is less accurate, as expected. However, one may notice that even if the predictions of $\mathcal{N}$ have a larger RMSE, it does not necessarily affect the resulting predicate orderings, as long as the relative orders of predicates are the same.

To show this, below we report a case study on the resulting predicate orderings under different noise ratios on DBLP-ACM using four predicates, namely $p_1 : t.\text{year} = s.\text{year}$, $p_2 : t.\text{authors} \approx_{\text{ED}} s.\text{authors}$, $p_3 : t.\text{venue} \approx_{\text{JD}} s.\text{venue}$, and $p_4 : t.\text{title} \approx_{\text{JD}} s.\text{title}$. When $\beta\% = 0\%$, the predicate ordering estimated via $\mathcal{N}$ is O1: p_1, p_2, p_3, p_4, which is the same as the ground truth ordering derived from the actual costs. This ordering is unchanged even if we injected 20% noises to the training data of $\mathcal{N}$. However, when we injected 40% noises, the predicate ordering is changed to O2: p_1, p_3, p_2, p_4, *i.e.,* only the relative order of p_2 and p_3 is swapped. When we further increased the noise ratio to 60%, the predicate ordering becomes more messy, *i.e.,* O3: p_2, p_3, p_1, p_4.

Case study on single vs. multi-execution plans. As remarked in Section 4, plan generation in HyperBlocker can be regarded as a pre-processing step for blocking, *i.e.,* once a plan is generated, it is applied in *all* partitions of D. However, it is also possible to group tuples into different partitions/clusters and generate an execution tree for each partition/cluster. We report a carefully designed case study, comparing two methods: (1) HyperBlocker$_{\text{same}}$ that uses the same execution plan for all partitions and (2) HyperBlocker$_{\text{mult}}$ that generates a different execution plan for each partition. Here we used DBLP-Scholar and split it into two partitions P_1 and P_2, where P_1 (resp. P_2) contains tuples with null (resp. non-null) values on year. We designed two blocking rules, namely φ_a and φ_b:

- $\varphi_a : t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \wedge t.\text{year} = s.\text{year} \to t.\text{eid} = s.\text{eid}$; and
- $\varphi_b : t.\text{year} = s.\text{year} \wedge t.\text{authors} \approx_{\text{JD}} s.\text{authors} \wedge t.\text{title} \approx_{\text{JD}} s.\text{title} \to t.\text{eid} = s.\text{eid}$.

For simplicity, we assume each execution plan consists of exactly one rule, leading to two execution plans, namely $\mathcal{T}_1 = \{\varphi_a\}$ and $\mathcal{T}_2 = \{\varphi_b\}$, where the predicates in each rule are evaluated in the order they appear above (*e.g.,* $t.\text{year} = s.\text{year}$ is evaluated first in $\mathcal{T}_2$). Moreover, HyperBlocker$_{\text{same}}$ used φ_a for both partitions, while HyperBlocker$_{\text{mult}}$ used φ_a for P_1 and used φ_a for P_2.

By the results, HyperBlocker$_{\text{mult}}$ accelerates HyperBlocker$_{\text{same}}$ by 32.7%, since HyperBlocker$_{\text{same}}$ adopts a poor execution plan: while distinct years are good indicators of mismatches and years are cheap to compare, $\mathcal{T}_1$ compares the values on years at the end of plan. In contrast, by applying $\mathcal{T}_2$, that compares years first, on P_2, we can quickly discard tuple pairs with distinct years (but this is not possible for P_1, where all tuples have the same value on years).

This case study showcases that it is indeed possible to improve the overall performance using a different execution plan for each partition, compared with a global execution plan for all partitions. However, after conducting tests on other datasets using the general setting, we find that the performance gain is not remarkable. This is because the global execution plan generated by EPG is not as poor as $\mathcal{T}_1$ and it works sufficiently well for most partitions. Moreover, the effectiveness of multiple execution plans also depends on the data distributions in different partitions, *e.g.,* in DBLP-ACM, the data distributions do not vary much across different partitions, so that a single execution plan suffices for reasonably good performance. Nevertheless, we plan to further optimize HyperBlocker over multiple execution plans as a promising future direction.