Cheetah: Accelerating Dynamic Graph Mining with Grouping Updates

YI ZHANG[*†], Huazhong University of Science and Technology, China
XIAOMENG YI, ZhejiangLab, China
YU HUANG[*‡] and JINGRUI YUAN[*†], Huazhong University of Science and Technology, China
CHUANGYI GUI, Eastern Institute of Technology, China
DAN CHEN, National University of Singapore, Singapore
LONG ZHENG[*†], Huazhong University of Science and Technology, China
JIANHUI YUE, Michigan Technological University, USA
XIAOFEI LIAO[*†] and HAI JIN[*†], Huazhong University of Science and Technology, China
JINGLING XUE, University of New South Wales, Australia

Graph pattern mining is essential for deciphering complex networks. In the real world, graphs are dynamic and evolve over time, necessitating updates in mining patterns to reflect these changes. Traditional methods use fine-grained incremental computation to avoid full re-mining after each update, which improves speed but often overlooks potential gains from examining inter-update interactions holistically, thus missing out on overall efficiency improvements.

In this paper, we introduce Cheetah, a dynamic graph mining system that processes updates in a coarse-grained manner by leveraging *exploration domains*. These domains exploit the community structure of real-world graphs to uncover data reuse opportunities typically missed by existing approaches. Exploration domains, which encapsulate extensive portions of the graph relevant to updates, allow multiple updates to explore the same regions efficiently. Cheetah dynamically constructs these domains using a management module that identifies and maintains areas of redundancy as the graph changes. By grouping updates within these domains and employing a neighbor-centric expansion strategy, Cheetah minimizes redundant data accesses. Our evaluation of Cheetah across five real-world datasets shows it outperforms current leading systems by an average factor of 2.63×.

[*]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Huazhong University of Science and Technology (HUST), Wuhan, 430074, China
[†]Cluster and Grid Computing Lab, School of Computer Science and Technology, HUST, Wuhan, 430074, China
[‡]School of Software Engineering, HUST, Wuhan, 430074, China

This is a new article, not an extension of a conference paper.
This work is supported by the National Key Research and Development Program of China under Grant No. 2023YFB4503400.
Authors' Contact Information: Y. Zhang, Huazhong University of Science and Technology, Wuhan, Hubei, China; email: yizh@hust.edu.cn; X. Yi, Zhejiang Lab, Hangzhou, Zhejiang, China; email: xiaomeng.yi@zhejianglab.com; Y. Huang (corresponding author), J. Yuan, Huazhong University of Science and Technology, Wuhan, Hubei, China; emails: {yuh, jryuan}@hust.edu.cn; C. Gui, Eastern Institute of Technology, Ningbo, Zhejiang, China; email: cygui@eitech.edu.cn; D. Chen, National University of Singapore, Singapore; email: danchen@nus.edu.sg; L. Zheng, Huazhong University of Science and Technology, Wuhan, Hubei, China; email: longzh@hust.edu.cn; J. Yue, Michigan Technological University, Houghton, USA; email: jyue@mtu.edu; X. Liao, H. Jin, Huazhong University of Science and Technology, Wuhan, Hubei, China; emails: {xfliao, hjin}@hust.edu.cn; J. Xue, School of Computer Science and Engineering, University of New South Wales, Sydney, Australia; email: j.xue@unsw.edu.au.

ACM 1544-3566/2025/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

CCS Concepts: • **Computer systems organization → Special purpose systems**; **Parallel architectures**.

Additional Key Words and Phrases: Graph mining system, dynamic graph, incremental computing

ACM Reference Format:
Yi Zhang, Xiaomeng Yi, Yu Huang, Jingrui Yuan, Chuangyi Gui, Dan Chen, Long Zheng, Jianhui Yue, Xiaofei Liao, Hai Jin, and Jingling Xue. 2025. Cheetah: Accelerating Dynamic Graph Mining with Grouping Updates. *ACM Trans. Arch. Code Optim.* 37, 4, Article 111 (August 2025), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

As graphs increasingly represent real-world relationships, extensive research explores their potential through graph mining methods and systems. Graph mining applications include social network analysis [1–3], anti-fraud detection [4], and bioinformatics [5]. While current systems mostly handle static graphs [6–15], the real world often deals with dynamic graphs that frequently update vertices and edges. For instance, on Black Friday, e-commerce giants like Amazon process billions of transactions [16], forming dynamic graphs that track customer and seller interactions, essential for detecting fraud.

Recomputing all matches in the entire graph for every update is extremely costly. Since updates usually affect only a minor portion of the graph [17], many studies [18–21] advocate for incremental computation strategies that only modify the impacted results. These methods [18–21] often utilize fine-grained incremental techniques, initiating from the altered vertices to perform pattern matching by expansion. Each update triggers a sequence of expansion operations to integrate new matches or eliminate outdated results, yet these updates are processed independently.

In practical applications, tracking each update's individual changes is often unnecessary [22–26]. For example, e-commerce platforms prioritize identifying groups of ordering behaviors that indicate fraud, rather than pinpointing the specific order responsible for a new match discovery. This allows updates to be handled with coarser granularity. Although existing methods [18–21] process updates individually, enhancing performance, they often overlook the interactions between updates, leading to inefficiencies and redundant data accesses in dynamic graph mining. Our work investigates how to leverage this overlooked locality to enhance the efficiency of dynamic graph mining processes.

Updates in the graph are highly clustered; over 90% share at least one vertex with others, as detailed in Sec. 2.2. This clustering suggests a significant locality in dynamic graph mining, where most updates originate from similar locations and share exploration areas during incremental execution. To leverage this feature, we introduce the concept of *exploration domains*. These domains organize the graph into distinct components that cover the majority of the working set involved in updates. By grouping updates within these domains and executing them concurrently, we can reuse overlapping expansion paths, thereby reducing redundant data accesses and enhancing efficiency.

Adopting coarse-grained incremental computation in dynamic graph mining presents significant challenges. Firstly, expansion paths across different workloads change dynamically with the graph's evolution, necessitating their capture and maintenance in real time. This online upkeep of exploration domains can be costly, potentially offsetting the gains from reduced data processing. Secondly, the effectiveness of processing multiple updates simultaneously hinges on maximizing overlap in expansion paths. However, the actual traversals in the underlying graph for various updates may not align efficiently. This misalignment means that the expected benefits of shared exploration domains are not fully realized, as most processing efforts are diverted to managing divergent paths. Additionally, simultaneous access to these paths can lead to competition for cache resources, further degrading performance rather than enhancing it.

In this work, we introduce Cheetah, a dynamic graph mining system that groups updates and uses a coarse-grained execution strategy to leverage spatial locality. Cheetah's primary innovation

Algorithm 1: Execution of per-update mining

```
Input: $G$ – Graph, $P$ – Pattern, $u$ – Update
Output: $match\_set$ – Match set
$s \leftarrow u$ // Initial the extended subgraph $s$ as update $u$
Function Update_Mining($G, P, s$):
    $candidate\_list$.append(Access_Neighbors($G, s$))
    foreach vertex $v$ in $candidate\_list$ do
        // Check if $v$ can be extended to subgraph $s$
        if Break_Symmetry($G, s, v$) then
            add $v$ to $subgraph$ $s$
            if $len(s) < len(P)$ then
                Update_Mining($G, P, s$) // Recursive call
            else        // Subgraphs extended to pattern size
                if Filter($G, P, s$) then
                    $match\_set$.push(Process($G, P, s$))
                remove $v$ from $subgraph$ $s$
```

involves constructing exploration domains around regions of overlapping vertices from previous matches, enabling shared vertex access during path expansions with minimal overhead from incremental adjustments based on new matches. Moreover, Cheetah groups updates within these domains to ensure coherent execution and minimize interference from unrelated updates. It applies a neighbor-centric expansion method, focusing on extending vertices rather than subgraphs for optimal data reuse. Built on a distributed framework, Cheetah integrates advanced parallelism techniques to enhance scalability and optimize load distribution for complex pattern mining on dynamic graphs.

The key contributions of this paper are as follows:

- We identify the primary causes of memory inefficiency in current dynamic graph mining systems and highlight opportunities for data reuse to mitigate memory constraints.
- We introduce *exploration domains* to capture the overlapping nature of the expansion process among updates, facilitating shared expansion paths and reducing data accesses.
- We develop Cheetah, a system driven by exploration domains. It utilizes previous matching results to maintain these domains efficiently, then groups and executes updates in a neighbor-centric manner to enhance the reuse of expansion paths.
- We deploy Cheetah on a 16-node distributed platform and benchmark it against real-world workloads. Results demonstrate that Cheetah achieves a 2.95× reduction in memory access and a 2.63× performance increase compared to Tesseract [21].

2 BACKGROUND AND MOTIVATION

2.1 Dynamic Graph Mining

Mining dynamic graphs is increasingly relevant as it offers insights into evolving networks. We define a mining pattern by P, with a match m in graph G being a subgraph isomorphic to P, and the set of all matches as M. As updates are applied to G, changing it from G_0 to G_1 with updates ΔG, the match set also transitions from M_0 to M_1. A common approach is to re-mine the entire set M_1 from scratch in G_1. Yet, continuously recomputing all matches for each update introduces significant overhead, especially in environments where graphs frequently change [27].

Graph mining challenges are often localized and bounded, so updates usually affect only a close subset of matches [17]. Accordingly, many studies [18–21] have pursued incremental graph mining.

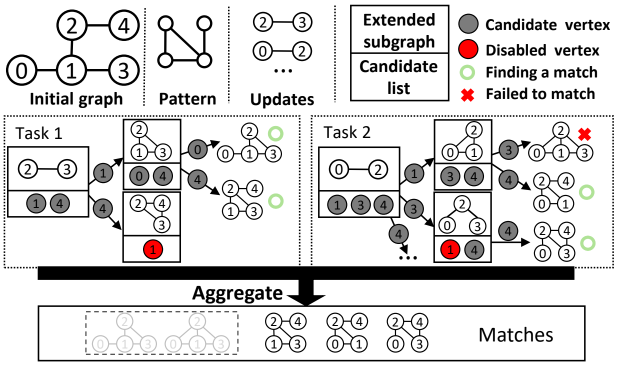


Fig. 1. An example of the per-update mining model showing match set changes after inserting edges: $\{(2,3),(0,2)\}$

These approaches begin at the update's point, expanding through neighboring vertices from this subgraph to identify all alterations in the match set. Alg. 1 explains this methodology, with the Update_Mining function incorporating four key operations: Access_Neighbor, Break_Symmetry, Filter, and Process.

When managing an extended subgraph *s*, Access_ Neighbor retrieves the neighbors of vertices in *s* to form a candidate list (line 3). Break_Symmetry ensures uniqueness by applying a deterministic rule (update canonicality, e.g., selecting the vertex with the smallest unexplored ID) to select vertices from this list (lines 4-6). If *s* is expandable (line 8), Update_Mining is recursively invoked. When *s* matches the pattern size, Filter discards non-viable subgraphs based on predefined conditions. Successful subgraphs then undergo final isomorphic checks and consolidation in Process (line 12). Notably, Access_Neighbor and Break_Symmetry involve extensive irregular vertex traversals, consuming a substantial portion of the processing time (up to 80% in Tesseract [21]).

Example. Fig. 1 demonstrates the per-update mining process after inserting two new edges, $(2,3)$ and $(0,2)$. Starting with edge $(2,3)$, neighbors of vertices 2 and 3 form the candidate list $\{1, 4\}$. Beginning with vertex 1, we extend to subgraph $(2,3,1)$. Since it is smaller than the pattern size, we continue extending to create matches $(2,3,1,0)$ and $(2,3,1,4)$. Next, selecting vertex 4 forms subgraph $(2,3,4)$, but extension with vertex 1 is avoided due to duplication, completing the mining for edge $(2,3)$. For edge $(0,2)$, extensions result in matches $(0,2,1,4)$ and $(0,2,3,4)$. The mining results are merged, and notably, the insertion of edge $(0,2)$ leads to the elimination of subgraph $(0,1,2,3)$ for not matching the updated pattern during aggregation.

2.2 Pitfalls of Per-Update Mining Process

Existing approaches [18–21] enhance individual update performance through an incremental computation paradigm but often overlook interactions between updates. In practice, tracking each change in detail is usually unnecessary; graph updates are more effectively managed at a coarser granularity. We contend that efficient dynamic graph mining systems should evaluate update interactions holistically rather than concentrating solely on individual changes.

Locality between Updates. We observe that updates in dynamic graphs exhibit strong *temporal locality*, with certain vertices appearing repeatedly within the same update batch. This pattern is typical in real-world scenarios, such as social networks, where active users frequently interact with multiple others within short periods, reflected in the updates. Fig. 2(a) illustrates this behavior across five real-world graphs with varying update batch sizes, showing that, on average, 95% of updated edges involve at least one recurring vertex. Since incremental computation typically begins

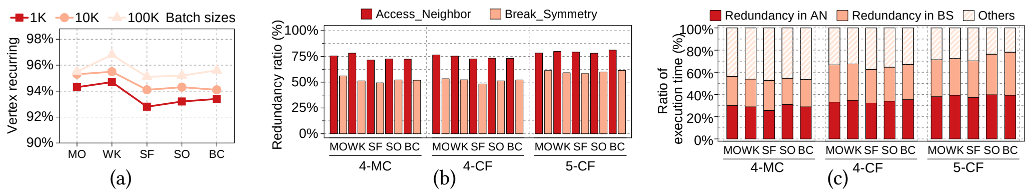


Fig. 2. (a) Proportion of recurring vertices in updates across varying batch sizes; (b) Redundancy ratios in Access_Neighbor and Break_Symmetry for 4-MC, 4-CF, and 5-CF on five real-world graphs (Table 1); and (c) Breakdown of execution times for operations in Tesseract, where AN represents Access_Neighbor and BS stands for Break_Symmetry

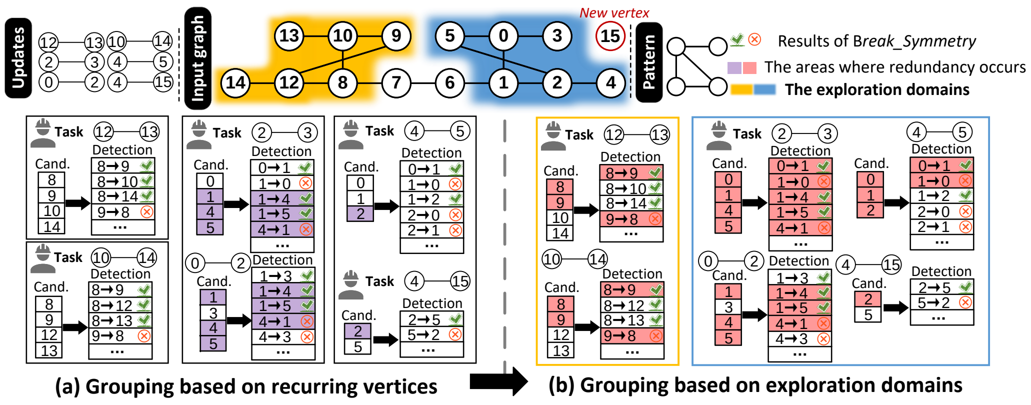

Updates
Input graph
New vertex
Pattern
Results of Break_Symmetry
The areas where redundancy occurs
The exploration domains
Task
Cand.
Detection
(a) Grouping based on recurring vertices
(b) Grouping based on exploration domains

Fig. 3. A motivating example illustrating redundant memory accesses across updates in dynamic graph mining, where (a) the black boxes represent groups of updates that are formed based on shared recurring vertices, and (b) the yellow and blue boxes represent groups of updates that are formed based on exploration domains

expansion from the updated vertex, recurring starting vertices suggest similar extension paths. This observation highlights the potential for reducing redundant accesses along identical paths by exploring the locality between updates, an aspect previously neglected in research.

We select the state-of-the-art system Tesseract [21] to explore inefficiencies in existing dynamic graph mining systems. A detailed analysis is conducted on redundant data accesses during the two most time-consuming stages, Access_Neighbor and Break_Symmetry. Fig. 2(b) shows the proportion of redundant accesses in these stages. For further details on the dataset and benchmark, refer to Sec. 7.1.

Redundancy in Access_Neighbor. In Access_Neighbor, redundant accesses occur due to repeated visits to neighbors of the same vertex, with these redundancies accounting for 78.72% of total accesses in this stage, as depicted in Fig. 2(b). During the incremental computation, each updated edge requires accessing its neighboring vertices to generate candidate lists for mining. The frequent occurrence of identical vertices in different updates, as shown in Fig. 2(a), increases the likelihood of revisiting the same edge lists, resulting in substantial redundancy in memory access.

Fig. 3 highlights this redundancy within Access_Neighbor. For instance, when mining for update (2, 3), neighbors of vertices 2 and 3 are accessed to form the candidate list $\{0, 1, 4, 5\}$. For a subsequent update (0, 2), the list is $\{1, 3, 4, 5\}$. The intersection of these lists is $\{1, 4, 5\}$, indicating redundant accesses to neighbors of the common vertex 2 during the mining processes for updates (2, 3) and (0, 2).

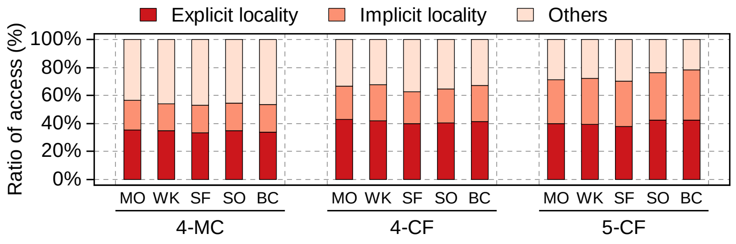


Fig. 4. Ratio of redundant access for explicit and implicit locality

Redundancy in Break_Symmetry. Redundancy in Break_Symmetry occurs from repeatedly sorting the same vertices in candidate lists across different updates, accounting for 62.32% of the stage's accesses, as shown in Fig. 2(b). To prevent duplicate subgraph extensions, Break_Symmetry sorts vertices to control their inclusion sequence. Due to frequent vertex recurrence in updates, overlapping candidate lists lead to unnecessary repeated sorting of intersecting vertices. For example, in the update (2, 3) depicted in Fig. 3, vertices 1 and 4 are sorted with vertex 1 first. This ordering is redundantly repeated for the update (0, 2), which also includes vertices 1 and 4 in the candidate list. Note that other stages are less significant in terms of data access costs and data reuse, as they do not involve accessing extended paths, thus presenting lower redundancy potential.
Remarks. Graph mining is memory-intensive [12, 28, 29], where excessive data accesses significantly impair performance. We analyze the execution times for three typical applications across five real-world graphs. Specifically, we track the total number of vertex accesses and redundant accesses to the same vertex in the Access_Neighbor and Break_Symmetry to quantify the memory redundancy ratio. Additionally, we use Intel VTune Profiler to analyze execution time and determine the proportion of redundancy. As depicted in Fig. 2(c), redundant memory accesses in Access_Neighbor and Break_Symmetry account for 34.95% and 29.84% of the total execution time, respectively. In more extreme cases, such as 5-*Clique Finding* (5-CF) on the BC graph, these figures escalate to 40.12% and 38.72%, respectively.

3 THE CHEETAH DESIGN

3.1 Identifying Inefficiencies and Outlining Challenges

Existing approaches utilize a fine-grained execution model that processes each update individually, with limited consideration for interactions between updates. This per-update approach often fails to capitalize on the potential locality benefits arising from overlaps in extension paths among updates. In this work, we introduce a coarse-grained execution policy that groups correlated updates and processes them simultaneously, aiming to better exploit these overlaps.

The basic strategy is to prioritize grouping updates that involve the same vertices, which is an intuitive and logical approach for several reasons. Firstly, a significant proportion of updates—up to 95% as shown in Fig. 2(a)—involve recurring vertices. Secondly, we define *expansion paths* as sequences of vertices traversed when an updated edge expands into a subgraph matching the target pattern size, and since incremental computations start from these updates, shared vertices suggest similar expansion paths. For instance, as depicted in Fig. 3(a), simultaneous execution of updates (2, 3) and (0, 2) facilitates shared utilization of vertices 1, 4, and 5. This coordination effectively prevents redundant operations, such as duplicative formations of the candidate list {1, 4, 5} in Access_Neighbor and redundant order verifications in Break_Symmetry.

While the basic grouping strategy focuses on the locality of the root vertex in the expansion paths (termed *explicit locality*), it overlooks the locality of vertices along these paths (termed *implicit locality*). This oversight means the full potential for locality between updates is not harnessed.

For instance, the update pairs $\{(2,3),(0,2)\}$ and $\{(4,5),(4,15)\}$ in Fig. 3(b) initially do not share vertices, but they intersect at vertices 0 and 1 along their expansion paths. This leads to redundant data accesses when forming the candidate list with $\{0,1\}$ in Access_Neighbor and sorting vertices 0 and 1 in Break_Symmetry. As depicted in Fig. 3(b), by merging the groups $\{(2,3),(0,2)\}$ and $\{(4,5),(4,15)\}$, vertices 0 and 1 are shared, which helps avoid redundant access. Fig. 4 provides a detailed analysis showing that the proportions of overall redundant data accesses due to explicit locality and implicit locality are 40.78% and 27.74%, respectively. Therefore, fully exploiting both types of locality is crucial to improving overall execution efficiency.

In this paper, we find that the inherent locality in dynamic graph mining is largely due to overlapping expansion paths, with most updates sharing similar exploration areas. We define *exploration domains* as distinct sets of vertices that contain these expansion paths. Additionally, we introduce a method of grouping updates driven by exploration domains, allowing for the full exploitation of both explicit and implicit localities among updates.

Challenges. Implementing exploration domain-driven grouping updates for dynamic graph processing presents several challenges. Firstly, expansion paths vary across workloads and evolve with the graph updates, necessitating that exploration domains, dependent on these paths, be constructed dynamically at runtime. Efficiently and precisely maintaining these domains on-the-fly is a complex task. Secondly, while updates are grouped based on exploration domains to maximize path overlap during execution, the actual graph traversals may not align well among different updates. This misalignment often leads to increased processing times as systems handle divergent paths, limiting the effective use of shared exploration domain localities. Moreover, this scenario can lead to cache resource competition among updates, potentially degrading overall performance.

3.2 Overview

Motivated by the findings above, we introduce Cheetah, an efficient dynamic graph mining system that groups edge updates with recurring vertices to eliminate redundant operations seen in traditional systems. Cheetah strategically groups updates using exploration domains derived from match sets, enhancing data locality with minimal redundancy by merging updates into groups and adopting a *neighbor-centric* extension approach.

Fig. 5 illustrates two key modules in Cheetah: the domain management module and the group-based mining module. The exploration-domain management module forms exploration domains from match sets of the initial graph and adjusts them as the graph evolves. Each exploration domain consists of disjoint vertex sets from the initial graph matches. This module also supports assigning updates to the corresponding exploration domains as the graph changes. The group-based mining module identifies edges within an exploration domain that share common recurring vertices and merges them into update-groups, reducing redundant neighbor accesses. Additionally, this module uses a neighbor-centric extension strategy on update-subgraphs to minimize symmetry break checks and implements topological pruning to eliminate unpromising extensions.

Fig. 5 shows Cheetah's workflow. Cheetah creates exploration domains from initial graph matches (**Step ❶**) and assigns updates to these domains to ensure that updates with the same expansion path are executed together (**Step ❷** and **Step ❸**). Details on utilizing exploration domains to assign updates are provided in Sec. 4.2. Cheetah schedules mining tasks to handle updates within exploration domains (**Step ❹**). A mining task merges updates into update-groups, generates update-subgraphs, and extends them (**Step ❺**). These extended subgraphs are filtered through isomorphism tests and matched subgraphs are stored in the match set (**Step ❻**). Unlike Tesseract [21], which performs per-update mining (**Step ❺**), Cheetah executes neighbor-centric extensions for grouped updates (**Step ❺**). With each new update batch, Cheetah updates the exploration domains (**Step ❼**).

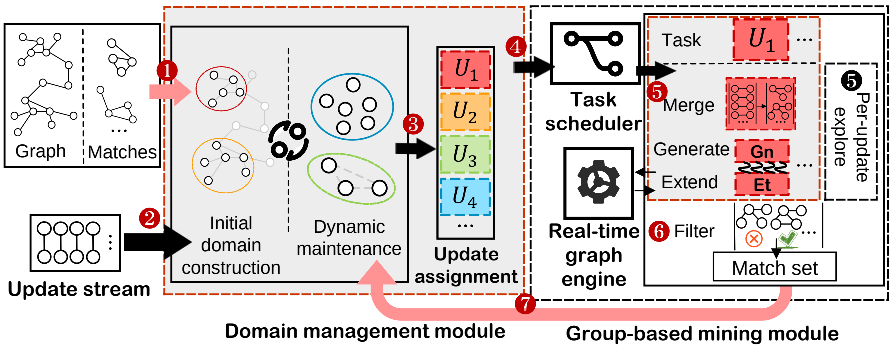


Fig. 5. Overview of Cheetah (with this work gray-shaded)

Discussion. While our batch-dynamic approach may introduce a slight increase in response time for individual updates, this effect is negligible in real-world high-throughput scenarios. For example, in social networks like Twitter, the edge update rate can reach 236K updates per second, whereas the incremental mining rate of state-of-the-art systems [21] is typically around 34K updates per second for executing 4-CF. This gap becomes even more pronounced for complex applications, where throughput is significantly lower. As a result, updates naturally accumulate while waiting for processing, making sequential processing inefficient.

To address this, our approach proactively processes pending updates in batches rather than sequentially. This strategy not only reduces processing latency for many updates by leveraging temporal locality but also eliminates redundant operations across updates. Consequently, our method achieves simultaneous improvements in both latency and throughput, making it highly effective for high-throughput environments.

Additionally, we provide a comprehensive latency analysis in Sec. 7.2, which demonstrates that our approach achieves a shorter average latency and lower latency variance compared to baseline methods. These results further validate the practicality and efficiency of our batch-dynamic approach in real-world applications.

4 EXPLORATION-DOMAIN MANAGEMENT

In this section, we analyze the construction of exploration domains and introduce the concept of *match connected component* (MCC) for their formation. We then describe the dynamic incremental maintenance strategy and the process of assigning updates to these exploration domains.

4.1 Constructing Exploration Domains

The most accurate way to obtain exploration domains is to record the entire mining process for batch updates, but this approach is impractical due to its high overhead and inability to prevent redundancy within the batch. Thus, we propose an approximation method for constructing exploration domains. We assume that exploration domains created by this method have two properties: (1) they should contain as many updated vertices as possible to link them with updates and detect redundancy, excluding vertices not accessed during mining to keep the domain size manageable, and (2) vertices that are close to each other in the latest graph should be in the same exploration domain, as mining typically involves accessing these vertices and their neighbors.

Correlations in real-world graphs show that vertices accessed in one batch of updates are often similar to those in the next [30, 31]. Additionally, vertices in the match set must have been accessed during mining to be included in the set. Thus, we can construct approximate exploration domains

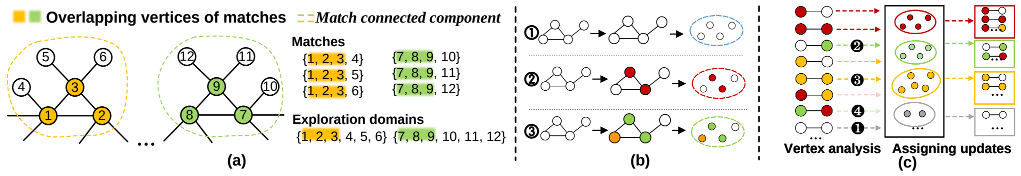


Fig. 6. (a) Constructing exploration domains via MCCs; (b) Cases of construction; (c) Update assignment

Algorithm 2: Assigning updates to exploration domains

Input: U – Updates, G – Graph, D – Exploration domains
Output: UD – Updates in exploration domains

```
Function Assign_Updates(U, D):
    foreach update u in U do
        // Initialize $v_0, v_1$ with the vertices in u
        vertex $\{v_0, v_1\} \leftarrow u$
        if $v_0, v_1$ not in any Domains then
            $d \leftarrow$ Create_NewDomain($\{v_0, v_1\}, D$)
            Assign($u, d$)
        if only one $v_i$ of $\{v_0, v_1\}$ is in Domains then
            $d \leftarrow$ Find_Domain($v_i, D$)
            Add_To_Domain($u, d$)
            Assign($u, d$)
        else // Both vertices are in domains
            $d_0 \leftarrow$ Find_Domain($v_0, D$)
            $d_1 \leftarrow$ Find_Domain($v_1, D$)
            if $d_0 == d_1$ then // $v_0, v_1$ are in the same domains
                Assign($u, d_0$)
            else // $v_0, v_1$ are in different domains
                $d \leftarrow$ Select_By_Recurring_Degree($u, D, G$)
                Assign($u, d$)
    $UD \leftarrow$ Get_Updates_From_Domains(D)
```

from the match set of the previous batch to detect redundancy in the next. We introduce the concept of MCC to facilitate this process.

Definition 1 (MCC). *Given all the vertices V and edges E in a match set, a connected subgraph $s = (V', E')$ is a match connected component if $\forall\, u \in (V - V')$ and $\forall\, v \in V'$, $(u, v) \notin E$.*

In what follows, we also refer to an MCC as a maximal connected subgraph containing vertices and edges from the match set. An MCC effectively represents an exploration domain for two main reasons. Firstly, all vertices in an MCC have been accessed during mining, as only accessed vertices appear in the match set. Secondly, the connected nature of an MCC ensures that neighboring vertices accessed during mining belong to the same domain. For all vertices $u \in V'$ in an MCC, there exits at least one vertex $v \in V'$ such that $(u, v) \in E$. This implies that there is at least one path between every pair of vertices in the MCC, ensuring the MCC itself is connected.

Fig. 6(a) illustrates constructing two exploration domains from two MCCs. Based on the vertices and edges in the matches, two MCCs are identified: $\{1, 2, 3, 4, 5, 6\}$ and $\{7, 8, 9, 10, 11, 12\}$. The vertices in each MCC form an exploration domain. The construction steps are as follows (shown in Fig. 6(b)). First, we examine each match to check if its vertices have been assigned to any domains. If none are assigned, we create a new domain and assign all the match's vertices to it, which is the case ①. If some vertices are already assigned to a domain, we assign the remaining vertices to the same domain, i.e., the case ②. If a match contains vertices from multiple domains, we merge these domains into one and assign the remaining vertices to it, like case ③.

Constructing exploration domains from MCCs offers several advantages. Firstly, it reveals redundancy by including recurring vertices from matches, indicating multiple accesses during mining. Secondly, it is efficient, as tracing the mining access process is difficult and costly. Thirdly, it reduces construction overhead by integrating domain construction with mining result production, thus concealing the overhead.

4.2 Assigning Updates to Exploration Domains

Since the initial domains are constructed from the match set, they may not contain all the vertices in the edge updates. For those edge updates containing these unclassified vertices, Cheetah temporarily puts these vertices into exploration domains and removes them from the domains after assigning, which ensures that all updates are included in exploration domains.

Given a set of exploration domains and an edge update in a batch, we assign the update by considering four scenarios, as shown in Fig. 6(c) and Alg. 2. If neither vertex of the edge is in any domain, we create a new domain and assign the update to it, like the case ❶ (lines 5-7). If only one vertex is in an existing domain, we add the other vertex to that domain and assign the update, which is the case ❷ (lines 8-11). If both vertices are in the same domain, we assign the update to that domain, i.e., the case ❸ (lines 15-16). If the vertices are in different domains, we assign the update to the domain containing the vertex with the higher recurring degree, like the case ❹ (lines 18-19). The recurring degree of a vertex v is calculated as:

$$R(v) = Dgr(v) \times Num(v) \tag{1}$$

where $Dgr(v)$ is the degree of vertex v and $Num(v)$ is the number of times vertex v is updated in the current batch.

Processing an update batch may introduce new matches to the match set and remove old ones, necessitating updates to existing exploration domains for accurate processing of future batches. The update process differentiates between additions and deletions. Newly added matches are incorporated using the initial graph's domain construction method. For removed matches, however, the corresponding vertices are eliminated from the match set, and an analysis of vertex connectivity in the updated graph could result in subdividing some exploration domains into smaller ones.

5 GROUP-BASED MINING

To exploit data locality in dynamic graph mining, we introduce group-based mining, processing updates within the same exploration domain collectively. This approach enhances neighbor reuse by implementing a neighbor cache for each exploration domain, significantly reducing redundant off-chip memory accesses. Furthermore, we promote reusing a canonical order to efficiently explore multiple subgraphs, thereby minimizing the overhead from symmetry breaking checks. We also suggest pruning ineffective exploration of subgraphs to further boost performance.

5.1 Group Building

We consolidate edge updates within an exploration domain into update-groups. Each update-group associated with a recurring vertex v includes v and its connected updates, forming a star topology. Constructing an update-group involves two steps. Initially, Cheetah identifies the recurring vertex set $\{c_1, c_2, ..., c_n\}$ from updates within the exploration domain. Subsequently, Cheetah assigns an edge update (v_1, v_2) to the update-group of the recurring vertex v, if v_1 or v_2 matches v. If both v_1 and v_2 are recurring vertices, the update is added to the group with higher recurring degree as per Eq. 1. For instance, Fig. 7(a) illustrates the update-group for vertex 2 comprising vertices $\{2, 0, 3, 7\}$. To prevent duplicate updates within groups, we ensure that each updated edge (v_1, v_2) adheres to the condition $v_1 < v_2$.

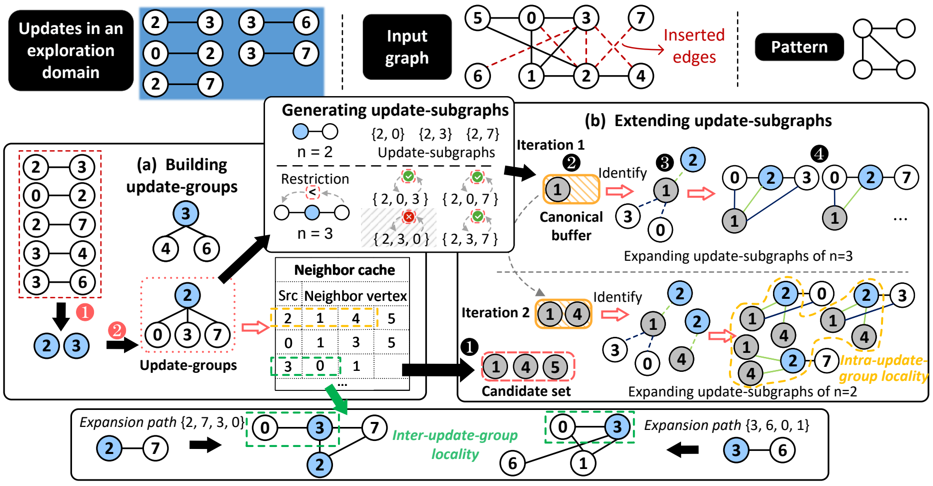

Updates in an exploration domain
Input graph
Inserted edges
Pattern
Generating update-subgraphs
n = 2
{2, 0} {2, 3} {2, 7}
Update-subgraphs
Restriction
n = 3
{2, 0, 3} {2, 0, 7}
{2, 3, 0} {2, 3, 7}
(a) Building update-groups
Update-groups
Neighbor cache
Src Neighbor vertex
(b) Extending update-subgraphs
Iteration 1
Canonical buffer
Identify
Expanding update-subgraphs of n=3
Iteration 2
Candidate set
Expanding update-subgraphs of n=2
Intra-update-group locality
Expansion path {2, 7, 3, 0}
Inter-update-group locality
Expansion path {3, 6, 0, 1}

Fig. 7. Group-based mining with the tailed triangle pattern (n: vertices per update-subgraph)

Intra-Update-Group Locality. To enhance the temporal locality of neighbor accesses, we introduce a neighbor cache for each update-group ug during its exploration. This cache retains the vertices within ug along with their neighbors. Fig. 7(a) displays the neighbor cache for the update-group $\{2, 0, 3, 7\}$. Specifically, for the group $\{(2, 3), (0, 2), (2, 7)\}$, the expanded subgraphs of updates $(2, 3)$, $(0, 2)$, and $(2, 7)$ are $\{2, 0, 1, 4\}$, $\{2, 3, 1, 4\}$, and $\{2, 7, 1, 4\}$, respectively. Since all these expansions originate from vertex 2, the neighbors of vertex 2 are frequently accessed and vertex 2 becomes a shared vertex among them. Thus, our neighbor cache effectively captures this temporal locality, denoted as *intra-update-group locality*. The size of the neighbor cache is small since an update-group encompasses only a limited number of vertices, and the cache stores solely the immediate neighbors of these vertices. Thus, a neighbor cache can be efficiently accommodated in a CPU cache, significantly reducing off-chip memory accesses to neighbors. In Tesseract, each update of $\{(2, 3), (0, 2), (2, 7)\}$ is processed independently, leading to multiple redundant accesses where vertex 2 is accessed three times. In contrast, Cheetah merges these updates into a single update-group, enabling the construction of a neighbor cache that ensures vertex 2 is accessed only once, thereby significantly reducing redundant memory operations.

Inter-Update-Group Locality. We can further leverage *inter-update-group locality* by exploring multiple update-groups in the same exploration domain simultaneously. Typically, in an MCC-formed domain (i.e., an exploration domain constructed using the vertices within an MCC), a recurring vertex $v1$ is likely to be a two- or three-hop neighbor of another recurring vertex $v2$ if both are in the same domain. When subgraphs extending from $v1$'s update-group and $v2$'s update-group are explored together, $v1$ appears in the subgraphs derived from $v2$, resulting in increased accesses to $v1$'s neighbors. Specifically, in the example shown in Fig. 7, vertex 3 is a neighbor of vertex 2, which causes the expansion subgraphs generated by update $(2, 7)$ and update $(3, 6)$ to share vertex 3 in expansions $\{2, 7, 3, 0\}$ and $\{3, 6, 0, 1\}$. This connectivity enhances the efficiency of accessing shared neighbor data across different update-groups.

5.2 Extending Subgraphs in Group

To efficiently extend subgraphs within groups, we initially generate update-subgraphs from an update-group and then extend these subgraphs through neighbor-centric extensions.

Generating Update-Subgraphs. Update-subgraphs are formed from the vertices and their edges within update-groups, with sizes ranging from 2 to $p-2$ for a p-size pattern. To construct a set of update-subgraphs for an update-group, Cheetah identifies a recurring vertex as the necessary vertex and selects additional vertices from the update-group.

Update-subgraphs generation incurs significant storage overhead for two main reasons. Firstly, the potential number of update-subgraphs is large, driven by the combination possibilities within update-groups. For a k-size update-group and a p-size pattern, the total number of possible update-subgraphs is $C_k^2 + C_k^3 + \ldots + C_k^p \ldots + C_k^k = 2^k - k - 1$, resulting in substantial memory usage. To mitigate this, we cap the maximum vertex count of update-subgraphs at p, curbing memory overhead and preventing the production of redundant intermediate results. Secondly, subgraph automorphism can lead to duplicate subgraphs if vertices are selected in different orders. For instance, both vertex selection orders $0, 3$ and $3, 0$ in the context of the recurring vertex 2 produce the same update-subgraph $(2, 0, 3)$, as depicted in Fig. 7(a). To prevent such duplicates, we implement the following two rules to control the order of vertex selection.

Definition 2. *Let $s = (v_1 \ldots v_k)$ be an update-subgraph in an update-group g, where v_1 is a recurring vertex in a mining pattern. A vertex v can be added to s if: (1) s has fewer vertices than the pattern size, and (2) v has the smallest ID among all vertices in g (excluding v_1) and has not been added to s before.*

Example. Fig. 7 illustrates the group-based mining process for the tailed triangle pattern, where n denotes the number of vertices in each update-subgraph. We examine the update-group $\{2, 0, 3, 7\}$, with subscripts indicating the sequence of update-subgraph formation. For the tailed triangle pattern, we focus on identifying subgraphs with fewer than four vertices. Since 2 is a recurring vertex, it appears in every update-subgraph, allowing the easy formation of $\{2, 0\}$, $\{2, 3\}$, and $\{2, 7\}$ for two-vertex subgraphs. For $n = 3$, expanding from vertex 0 allows further expansion to vertices 3 and 7. However, starting from vertex 3 prevents further expansion to vertex 0 as per Def. 2, ensuring each update-subgraph is uniquely generated.

Neighbor-Centric Extension. To reduce symmetry breaking overhead, Cheetah extends multiple update-subgraphs simultaneously using a uniform canonical order. This approach targets a set of update-subgraphs, S, each comprising $(p-k)$ vertices, where p denotes the pattern size. These subgraphs are augmented by adding k vertices, collectively called vs, chosen from the neighbors of vertices in the corresponding update-group g. The process involves verifying whether a vertex sequence in vs can extend a subgraph in a canonical manner. The resulting extensions, forming complete patterns of p vertices, are potential matches and subjected to pattern matching, as shown in **Step ❺** in Fig. 5.

All vertices in vs are linked to every subgraph in S and are added after a single check of vs's canonical order. vs is assembled such that its vertices have the smallest IDs among unexplored neighbors of g, and these are stored in a canonical buffer for reuse in extending subgraphs within S. This neighbor-centric extension strategy leverages common neighbors to enhance efficiency and prevent duplicates by adhering to the canonical order set by Tesseract [21]. The method efficiently carries out extensions for all viable k-size vs configurations using the $(p-k)$-size update-graphs in S, where the size of S ranges from 2 to $p-2$.

Example. Fig. 7(b) demonstrates the neighbor-centric extension using a straightforward example to mine the tailed triangle pattern. We form the candidate set from adjacent vertices in the neighbor cache as $\{1, 4, 5\}$ (excluding vertices in the update-group) (❶). In the first iteration, vertex 1 is added to the canonical buffer (❷). With the buffer containing vertex 1, we identify that vertex 1 is connected with vertices $2, 3, 0$ (❸) and then extend 3-vertex update-subgraphs with vertex 1 in the buffer, resulting in extensions $(2, 0, 3, 1)$, $(2, 0, 7, 1)$, and $(2, 3, 7, 1)$ (❹). In the second iteration, vertex

Algorithm 3: Execution of group-based mining

```
Input: $G$ – Graph, $g$ – Update-group, $P$ – Pattern, $p$ – Pattern size
Output: $match\_set$ – Match set
Function Group_Mining($G, P, g$):
    // Initial canonical buffer $buf$ and extended set $e\_set$
    $buf \leftarrow \emptyset, e\_set \leftarrow \emptyset$
    $S \leftarrow$ Update_Subgraphs_Generation($g, p$)
    $nc \leftarrow$ Access ($G, g$) // Get neighbor cache $nc$
    $CS \leftarrow$Get_Cacdidate_Set($nc$) // Get candidate vertices $CS$
    // Traverse candidate vertices to generate extended sets
    foreach vertex $v_i \in CS$ do
        $buf$.push($v_i$)
        Subgraphs_Extend ($G, S, CS, buf, e\_set, v_i, 1$ )
        $buf$.pop()
    $match\_set \leftarrow$ Process($G, P, e\_set$)
// Procedure of the neighbor-centric extension
Function Subgraphs_Extend($G, S, CS, buf, e\_set, v_i, iter$):
    // Generate canonical order for update-subgraphs
    if $iter == 1$ or Break_Symmetry($CS, buf$) then
        $p\_S \leftarrow$ Prune_Update_Subgraphs($G, S[p - iter], buf$)
    if $!empty(p\_S)$ then // Generate extended subgraphs
        $e\_set$.push(Connect_Update_Subgraphs($p\_S, buf, p - iter$))
    if $iter <= (p - 2)$ then // Add vertices to canonical buffer
        $tmp \leftarrow CS \cup N(v_i)$, $v_{next} \leftarrow tmp$.pop( )
        while $v_{next} \neq NULL$ do
            $buf$.push($v_{next}$)
            Subgraphs_Extend($G, S, CS, buf, e\_set, v_{next}, iter + 1$)
            $buf$.pop()
            $v_{next} \leftarrow tmp$.pop( )
    else // Exit under the order in $buf$ is non-canonical
        return
```

4 is added to the buffer from the candidate set, passing the symmetry check as vertex 4 has the smallest ID among the candidates. With two vertices in the buffer, we execute a neighbor-centric extension for 2-vertex update-subgraphs, yielding $(2, 0, 1, 4)$, $(2, 3, 1, 4)$, and $(2, 7, 1, 4)$. In Tesseract, the update set $\{(0, 2), (2, 3), (2, 7)\}$ are processed independently, causing the vertex pair $(1, 4)$ to undergo symmetry checks three times, once per update. In contrast, Cheetah groups these updates and our neighbor-centric extension performs the symmetry check only once, eliminating redundant operations and improving efficiency.

Topological Pruning. Traditional methods [11, 21] perform isomorphism checks on extended subgraphs. The neighbor-centric extension method avoids these checks for update-subgraphs that do not connect to all vertices in the canonical order buffer. Cheetah, on the other hand, conducts isomorphism checks on update-subgraphs connected to all vertices in this buffer, since subgraphs not connected to any buffer vertices cannot qualify as pattern matches. This approach, known as *topological pruning*, eliminates unfeasible extensions to further enhance performance.

5.3 Putting Them Together

Alg. 3 demonstrates the comprehensive procedure of group-based mining. Initially, Cheetah parallelly explores each update-group using GROUP_MINING (line 1). UPDATE_ SUBGRAPH_GENERATION (line 4) then generates the update-subgraphs for the update-group. Subsequently, Cheetah accesses the graph to fill the neighbor cache for update-groups and assembles the candidate set from vertices in the neighbor cache and their neighbors. The process progresses to neighbor-centric extensions, starting from a canonical order buffer buf containing a single vertex (line 8), and extends $(p-1)$ update-subgraphs through SUBGRAPHS_EXTEND.

SUBGRAPHS_EXTEND (line 10) iteratively extends, taking into account buf with vertices ranging from 2 to $p-2$, by recursively invoking itself, with parameter $iter$ denoting the number of vertices in buf. If buf encompasses multiple vertices, BREAK_SYMMETRY ensures that the vertex sequence in buf is canonical (line 16). When buf holds $iter$ vertices, SUBGRAPHS_EXTEND reviews the update-subgraphs with $p-iter$ vertices and applies PRUNE_UPDATE_SUBGRAPHS for topological pruning of these subgraphs (line 17). CONNECT_UPDATE_SUBGRAPH identifies and includes connected vertices for those in the canonical buffer, thus expanding the pruned update-subgraphs. These expanded subgraphs es are then added to the output subgraph set (line 19). SUBGRAPHS_EXTEND continues to recursively extend update-subgraphs across all feasible vertex counts (lines 22 to 26).

Finally, PROCESS conducts isomorphism testing and consolidation for extended subgraphs to renew the match set (line 12). When processing inserted updates, Cheetah checks if the subgraphs match the patterns in both the current and updated graphs. It is crucial to note that matches derived from inserted edges may need removal if edges are deleted, introducing some inefficiencies in the mining process. To address this, priority is given to deleted edges in an update set to pinpoint their matches, thereby reducing the wasteful use of computing resources.

Discussion. Although we introduce additional merging overhead of updates, it remains consistently low regardless of specific scenarios and graph types. The underlying reason is that the merging overhead per update is constant-time. Specifically, the merging operation only requires determining which domain the vertices of an updated edge belong to. Since this information is pre-stored, the operation requires only two lookup queries, resulting in an $O(1)$ computational cost. Additionally, since each vertex requires only 8 bits to store its domain information, the total storage overhead is merely V bytes, which is negligible compared to the graph data and intermediate results. Therefore, the merging overhead remains consistently low and does not scale significantly with different graph structures or scenarios.

6 IMPLEMENTATION

The high-level workflow of Cheetah includes two critical components for parallel computing: a `Manager` component and multiple `Worker` components. The `Manager` handles three main tasks: managing exploration domains, grouping updates according to these domains, and distributing the mining tasks of each group to the `Workers`. The `Workers` passively receive tasks from the `Manager` and submit the processed results, which helps dynamically maintain the exploration domains. This producer-consumer model is adaptable for both single-node and distributed platforms. On a single node, the model can be implemented using multiple processes. In a distributed setting, one node serves as the `Manager`, and the remaining nodes function as `Workers`.

Cheetah consists of approximately 5,000 lines of C++ code and runs on a distributed platform for parallel dynamic graph mining. Cheetah runs a single process on each node, utilizing MPI for inter-node data communication. Within each node, data is processed in parallel by OpenMP [32] threads. The `Manager` groups incoming updates and invokes MPI_ISEND to non-blockingly assign them to idle `Workers`. Each `Worker` independently executes the mining process and outputs changes in the match set, which are aggregated through the collective communication function MPI_REDUCE. Communication times between `Workers` and the `Manager` are significantly shorter than execution times, enabling pipe-lined processing and communication. Thus, communication overhead does not pose a bottleneck.

6.1 Parallel Execution of Grouping Updates

Execution Parallelism. In Cheetah, we utilize MPI and OpenMP to facilitate inter-node and intra-node parallel execution, respectively. Cheetah incorporates parallelisms at both the domain and update levels, tailored for inter-node and intra-node environments. Specifically, across multiple nodes, groups associated with different domains are executed in parallel, allowing the updates to be processed independently without data sharing between `Workers`. This independence is advantageous as grouping updates with potential data overlap within the same exploration domain minimizes data reuse across groups in different exploration domains. Thus, allocating distinct groups to separate `Workers` efficiently leverages inter-update locality without compromising performance. Within a single node, we enhance intra-domain mining parallelism by employing update-level parallelism, where updates within the same group are distributed across different threads of a `Worker`. This is enabled by the neighbor-centric extension approach, which allows for the independent extension of each vertex in an update.
Load Balance. The workload among `Workers` in Cheetah can vary significantly due to differences in domain sizes and the number of neighbors in edge updates, leading to uneven workload distribution. To improve load balancing, Cheetah implements two strategies: First, Cheetah employs dynamic task assignment to keep all `Workers` actively engaged and ensure an even distribution of workload. For domains that are large and contain many updates, Cheetah allocates multiple `Workers` to efficiently handle the mining tasks. Additionally, for `Workers` that are consistently busy due to the extensive number of neighbors in the update-group, Cheetah incorporates a work-stealing mechanism. This approach allows less busy `Workers` to take on executable updates from those that are overloaded, similar to strategies used in prior works like [31, 33]. This method helps in maintaining optimal productivity and load balance across all `Workers`.

6.2 Data Management

Cheetah uses MongoDB to manage dynamically evolving graph data. The graph is stored in an adjacency list format, with each vertex record containing a list of outgoing edges, including the destination vertex, timestamp, and relevant labels. Deleted edges are marked distinctly. In a single-node setup, the entire data graph is stored in main memory. In a distributed environment, the data graph is replicated across all nodes. Prior research [21] has shown that MongoDB efficiently handles updates without imposing performance constraints.
Exploration Domains Storage. We efficiently manage exploration domains using a union-find tree, a data structure commonly employed in graph clustering to organize data into disjoint sets where each element belongs to exactly one set. Specifically, each exploration domain is represented by a rooted tree, with each node containing one vertex. Additionally, by optimizing the index structure and refining the query tree, we enhance the query efficiency at runtime.
Ordered Neighbor Cache. When initially accessing a vertex's neighbors, we sort and store them in a neighbor cache, offering three advantages: 1) An ordered cache enables efficient construction of the vertex candidate list. 2) Restricting candidate list vertices to backward access minimizes inefficient comparisons during expansion. 3) Extensions from cached vertices are detected more efficiently. This sorting overhead occurs only once and is amortized by multiple update executions, which is common in practice [18–21].

7 EVALUATION

We begin by detailing the configuration of Cheetah, followed by an evaluation to demonstrate its superior performance.

Table 1. Real-world dynamic graph datasets

Dataset	#Vertices	#Edges	#BEdges	Type
sx-mathoverflow (MO)	0.02M	0.51M	0.24M	Interact graph
wiki-talk-temporal (WK)	1.14M	7.83M	3.31M	Interact graph
soc-flickr-growth (SF)	2.30M	33.13M	16.82M	Social network
sx-stackoverflow (SO)	2.60M	63.50M	36.23M	Interact graph
soc-bitcoin (BC)	24.58M	122.95M	64.49M	Social network

7.1 Experimental Setup

Benchmarks and Datasets. We assess Cheetah using four widely-used graph mining algorithms: *Triangle Counting* (TC), counting triangles within the graph; *Clique Finding* (k-CF), identifying complete subgraphs of size k (cliques); *Motif Counting* (k-MC), measuring the frequency of specific recurring subgraph patterns of size k; and *Frequent Subgraph Mining* (k-FSM), discovering frequent patterns constrained by the vertex count k. Our evaluation employs five real-world dynamic graph datasets, as detailed in Table 1. The "#BEdge" column lists the initial edge set size, and "#Edges" indicates the total edges after updates, reflecting real modifications from the datasets. The MO, WK, and SO datasets are interactive graphs, while the SF and BC datasets originate from social networks. We divide the streaming edge updates into multiple batches according to the specified batch size, ensuring continuity of updates within each batch. By default, a batch size of 100K updates is utilized to optimize the trade-off between throughput and latency, as recommended by [34]. Furthermore, we conduct experiments to evaluate the performance impact of varying batch sizes in Sec. 7.6. The threshold (s) for identifying frequent subgraphs in k-FSM is set at 2K. All graph structures in our experiments are uniformly undirected and unlabeled.

Configuration. Unless specified otherwise, our experimental setup involves a 16-node cluster connected by a 10Gbps network. Each node is equipped with two 14-core Intel Xeon E5-2680v4 processors, 256GB of memory, and a 512GB SSD. The system operates on a 64-bit Ubuntu 24.04 platform with kernel 6.8. Applications are compiled using GCC 13.2.0 with 'O3' optimization level. We utilize OpenMP 5.0 for multi-threading and MPI library 4.1.2 for distributed communications.

Baselines. We evaluate Cheetah against three dynamic graph mining systems, Tesseract [21] and Delta-BigJoin [20], using graph mining algorithms on five real-world graphs. Tesseract is recognized as the state-of-the-art system for supporting incremental dynamic graph mining, while Delta-BigJoin stands at the forefront of systems enabling subgraph mining on dynamic graphs, leveraging Timely Dataflow [35]. We also compare Cheetah with the state-of-the-art open-source graph mining system, GrpahPi [8]. Since GraphPi is limited to static graph mining, we extend its capabilities to support incremental mining by initiating the mining process from edge updates [36]. This enhanced version, termed GraphPi-D, allows for a fair and direct comparison with Cheetah in dynamic graph mining scenarios. Note that baseline systems support batching to reduce storage overhead and enhance parallelism. To ensure a fair comparison, updates come in batches in our experimental setups for all systems. Additionally, our evaluation excludes the time required for preprocessing, compilation, and applying updates to the graph structure, focusing solely on the time spent mining the results produced by updates.

7.2 Overall Results

We present the overall performance of Cheetah and offer a detailed analysis of the performance optimizations, particularly focusing on memory access.

Performance. Table 2 shows the execution times of Delta-BigJoin, GraphPi-D, Tesseract, and Cheetah for five real-world graphs using various algorithms, with Cheetah consistently outperforming the others. Overall, Cheetah outperforms Delta-BigJoin, GraphPi-D, and Tesseract by 17.69×, 2.58×, and 2.63×, respectively. Performance optimization varies by algorithm. Notably, for 4-CF, 4-MC,

Table 2. Performance of Delta-BigJoin, GraphPi-D, Tesseract, and Cheetah, where TO is short for "Time Out" denoting time over 3 hours

Algorithm	**System**	**Execution time (Secs)**				
		MO	WK	SF	SO	BC
TC	Delta-BigJoin	1.21	9.38	27.72	57.19	129.73
	GraphPi-D	0.89	8.74	25.66	55.49	119.11
	Tesseract	1.08	8.93	26.06	54.13	124.08
	Cheetah	0.87	8.17	23.86	50.54	106.97
4-CF	Delta-BigJoin	9.38	61.56	187.88	376.57	854.84
	GraphPi-D	3.73	22.34	89.25	376.57	341.37
	Tesseract	8.55	54.38	158.61	280.45	747.98
	Cheetah	4.21	21.39	61.35	95.07	291.14
4-MC	Delta-BigJoin	61.18	434.64	1894.65	3451.61	5985.09
	GraphPi-D	16.64	86.19	259.17	471.34	1093.33
	Tesseract	9.63	62.09	209.56	386.37	855.01
	Cheetah	4.59	22.56	86.57	116.61	309.65
5-CF	Delta-BigJoin	165.35	477.41	1254.50	1876.90	4839.85
	GraphPi-D	59.95	207.30	491.06	692.21	2484.56
	Tesseract	137.14	419.94	1208.87	1535.64	4203.73
	Cheetah	51.02	156.78	477.52	500.62	1586.75
4-FSM	Delta-BigJoin	787.87	5762.01	TO	TO	TO
	GraphPi-D	76.26	569.04	1526.10	2293.57	6375.07
	Tesseract	47.49	309.46	1055.91	1853.41	5128.25
	Cheetah	22.98	111.11	375.71	661.35	1547.43

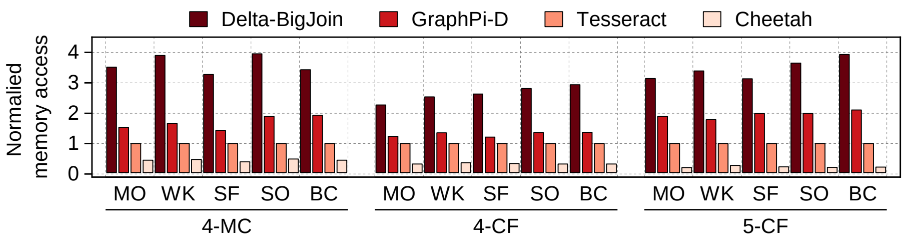


Fig. 8. Memory accesses of Delta-BigJoin, GraphPi-D, Tesseract, and Cheetah (normalized to Tesseract)

5-CF, and 4-FSM, Cheetah achieves average acceleration ratios of 2.53×, 2.66×, 2.72×, and 2.75× over Tesseract, showing significant performance improvements. However, acceleration for TC is the lowest, averaging only 1.12×. This is because triangle counting involves fewer vertices and limited vertex reuse. As the number of vertices in mined patterns increases slightly from 3 to 4, the memory access operations required during mining grow exponentially, offering significant opportunities to optimize redundant memory accesses. Additionally, our findings show that GraphPi-D performs similarly to Cheetah when executing TC, 4-CF, and 5-CF on the MO and WK datasets. However, in multi-pattern tasks, such as 4-MC and 4-FSM, GraphPi-D exhibits a performance gap of up to 5.12×. This discrepancy arises from redundant operations during the extension of updates for multiple patterns, which Cheetah mitigates by extending updates first and then applying isomorphic testing, thereby improving efficiency.

Cheetah's performance enhancement in processing these algorithms is primarily due to optimized memory access operations during graph mining of update batches. This optimization includes efficiently pinpointing likely accessed vertices through exploration domains and improving the

Table 3. Cache hit rate

System	App.	MO	WK	SF	SO	BC
Delta-BigJoin	4-MC	10.67%	15.69%	12.31%	11.02%	9.78%
	5-CF	7.56%	8.35%	6.89%	6.13%	5.57%
GraphPi-D	4-MC	20.99%	28.19%	22.37%	13.71%	9.424%
	5-CF	17.98%	20.32%	18.38%	13.47%	10.38%
Tesseract	4-MC	22.07%	34.31%	24.08%	18.72%	14.53%
	5-CF	23.57%	29.32%	18.18%	15.61%	13.28%
Cheetah	4-MC	71.13%	82.63%	75.71%	76.17%	80.12%
	5-CF	76.27%	88.73%	72.12%	75.88%	78.16%

Table 4. Performance trends of average latency, variance, P55, and P99 for 4-CF, 4-MC, and 5-CF on Twitter

System	Algorithm	Ave. (ms)	Var. (ms^2)	P50 (s)	P99 (s)
Delta-BigJoin	**4-CF**	7.91	15.16	1.51	4.58
GraphPi-D		6.16	11.51	1.39	3.12
Tesseract		7.05	8.11	1.43	3.25
Cheetah		2.76	2.58	0.61	1.93
Delta-BigJoin	**4-MC**	53.65	78.46	3.65	10.96
GraphPi-D		13.56	35.74	2.53	5.56
Tesseract		8.44	12.53	1.60	3.56
Cheetah		3.09	4.77	0.68	2.08
Delta-BigJoin	**5-CF**	62.98	118.25	8.23	21.81
GraphPi-D		48.35	55.65	5.07	11.45
Tesseract		52.61	50.73	5.51	13.23
Cheetah		21.32	11.97	2.28	6.11

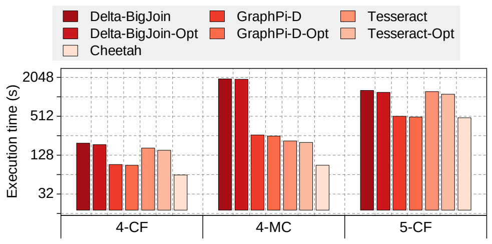


Fig. 9. Execution time of systems with optimizations for 4-CF, 4-MC, and 5-CF on SF

temporal locality of vertex information. Moreover, Cheetah minimizes redundant memory accesses by utilizing group-based mining to designate and identify specific mining operation domains for each vertex. This approach facilitates the reuse of intermediate results, enabling simultaneous processing of multiple update-subgraphs.

Memory Access Efficiency. Fig. 8 gives vertex memory access counts and Table 3 gives cache hit rate for 4-MC, 4-CF, and 5-CF across four real-world graphs, illustrating Cheetah's acceleration factors. For statistical purposes, we measure the memory access on a single node. The experimental results indicate that Tesseract outperforms Delta-BigJoin and GraphPi-D in terms of memory access efficiency. Furthermore, compared to Tesseract, Cheetah achieves significant reductions in off-chip memory access—55.37% for 4-MC, 65.13% for 4-CF, and 71.97% for 5-CF. These improvements are due to the implementation of a neighbor cache which enhances cache hit rates and reduces vertex accesses, moving away from Tesseract's per-update access strategy. As shown in Table 3, Cheetah's approach to leveraging access locality has led to cache hit rate improvements of up to 64.88% over Tesseract.

Latency Analysis. Table 4 presents the average latency, variance, P55, and P99 for 4-CF, 4-MC, and 5-CF on Twitter [37]. In our experiments, we conservatively set the update rate to 100K updates per second (higher rates further favor our approach). Cheetah achieves shorter average, P55, and P99 latencies by eliminating redundant computations, enhancing execution efficiency. It also maintains lower latency variance, ensuring consistent performance. These improvements stem from the proactive processing of pending updates, reducing waiting times. Overall, Cheetah delivers improved *Quality of Service* (QoS) through lower average latency and reduced variance.

Comparison with Simply Optimized Systems. We apply topological sorting to all baseline systems to evaluate the performance of 4-CF, 4-MC, and 5-CF on the SF dataset. In Fig. 9, Tesseract-Opt represents the optimized version of Tesseract, with similar optimizations applied to other systems. This simple optimization improved system performance by 1.21% to 9.26%. However, even with this enhancement, the optimized systems still exhibit a significant performance gap compared to Cheetah. This suggests that while such optimizations can provide some benefits, they may not fundamentally improve system execution efficiency.

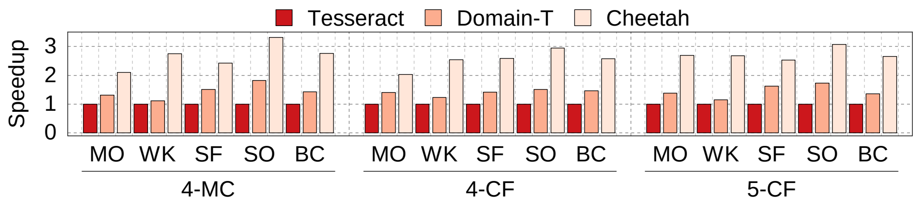


Fig. 10. Speedups of Tesseract, Domain-T, and Cheetah (normalized to Tesseract), where Domain-T is Tesseract with per-update mining restricted to exploration domains

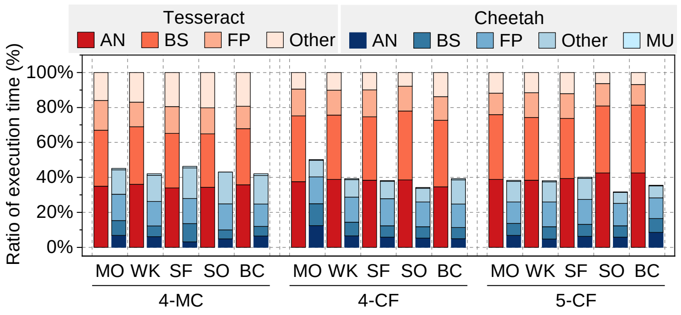
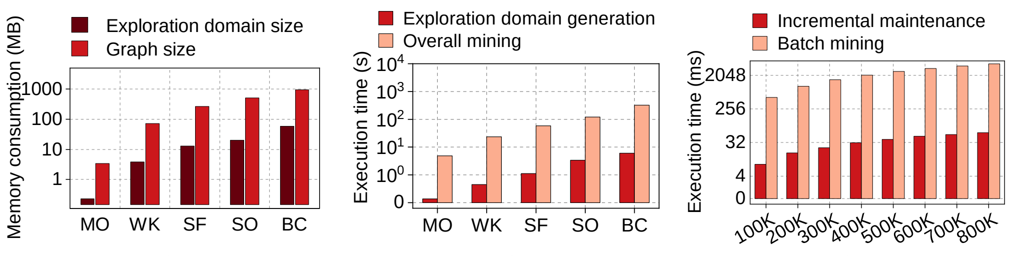


Fig. 11. Performance breakdown of Tesseract and Cheetah (normalized to Tesseract)

Upper Bound Analysis. Theoretically, if all redundant accesses are completely eliminated, the execution time could be reduced by approximately 68.53% (as profiled in Fig. 2(c)), implying a theoretical upper bound of performance improvement of around 3.17×. However, achieving the ideal implementation requires infinitely large memory capacity to cache all shared intermediate results, which is not feasible in practice. Our approach effectively eliminates approximately 93.49% of redundant accesses, resulting in an average performance improvement of 2.63× compared to the state-of-the-art system [21].

7.3 Effectiveness of Optimizations in Cheetah

Effectiveness of Domain Management. Fig. 10 demonstrates the effectiveness of Cheetah's domain management in reducing redundancy by comparing Domain-T and Tesseract. Domain-T, which uses Cheetah's domain management but applies Tesseract's per-update strategy to Cheetah's groups, achieves average speedups of 1.47× on 4-MC, 1.36× on 4-CF, and 1.52× on 5-CF, respectively. These improvements stem from sharing extension paths between adjacent updates within the exploration domain, minimizing redundant data accesses. However, this strategy mainly enhances data locality among consecutive updates and does not fully optimize redundant access reduction.
Effectiveness of Group-Based Mining. In Fig. 10, we compare Cheetah with Domain-T to demonstrate the impact of group-based mining on Cheetah's performance. The key distinction between Domain-T and Cheetah lies in how updates assigned to an exploration domain are processed. Domain-T handles each update independently, similar to Tesseract, whereas Cheetah employs a group-based mining strategy. When updates exhibit no locality or only strictly pairwise locality, Domain-T can achieve performance comparable to Cheetah. While in real-world graphs, as shown in Fig. 10, Cheetah is consistently superior across various graphs and applications. For 4-MC, 4-CF, and 5-CF, Cheetah achieves average speedups of 1.81×, 1.86×, and 1.79× over Domain-T, respectively. These improvements are driven by Cheetah's merged memory accesses and neighbor-centric extensions, which Domain-T's per-update execution does not optimize. Notably, Cheetah records the highest speedup on SO in real-world graphs, benefiting significantly from reduced redundant



(a) (b) (c)

Fig. 12. (a) Memory consumption for storing exploration domains across various graphs; (b) Execution times for generating exploration domains across various graphs; (c) Execution times for incremental maintenance of exploration domains on WK

memory accesses and enhanced duplicate vertex detection, which are influenced by graph topology and update sequence order.

7.4 Breakdown on Mining Time

Fig. 11 shows the time distribution for different mining phases during 4-MC, 4-CF, and 5-CF executions on Cheetah and Tesseract. For Cheetah, the phases are merging updates into update-groups, accessing neighbors to construct sorted neighbor caches, breaking symmetry and generating extended subgraphs, completing matches via pruning optimizations or isomorphism tests, and additional mining operations. For Tesseract, the phases are ACCESS_NEIGHBOR, BREAK_SYMMETRY, FILTER and PROCESS, and other operations.

Specifically, when comparing the execution of processing 5-CF on BC, the runtime ratio of accessing neighbors and expanding subgraphs on Cheetah significantly drops from 81.29% to 46.32%. This reduction is due to Cheetah's strategy of merging updates into update-groups, which reduces redundant accesses to recurring vertices. While this introduces some overhead, it constitutes less than 1.08% of the total execution time. Additionally, by grouping updates, Cheetah effectively uses vertex detection results to extend multiple subgraphs using a neighbor-centric approach, optimizing the use of intermediate extension results.

7.5 Overhead of Managing Exploration Domains

Fig. 12 shows the extra overhead for managing exploration domains of processing 4-MC in Cheetah. In Fig. 12(a), the memory overhead for storing exploration domains accounts for only 3.97% to 6.81% of the total graph size, indicating a minimal impact on overall memory usage. As MCCs are only used to construct the initial exploration domain, they do not need to be stored, which further reduces memory footprint. In Fig. 12(b), the overhead for generating exploration domains accounts for just 2.83% of the total mining time, a proportion that decreases with larger graph sizes as mining matches becomes increasingly more resource-intensive than domain generation. Moreover, this overhead is a one-time cost, which can be reduced by integrating it into the initial matching process. Fig. 12(c) contrasts the incremental maintenance overhead of exploration domains with the computation overhead for batch mining at different batch sizes on WK. The time spent on incremental maintenance is significantly less than that for batch mining; for example, while it takes 12.15s to mine 800k updates on WK, only 0.078s are devoted to incremental maintenance. We have effectively reduced performance overhead by overlapping incremental maintenance with domain-based mining.

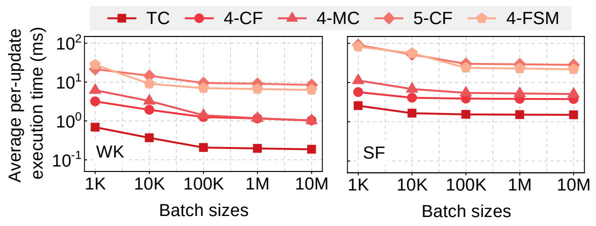


Fig. 13. Cheetah's average per-update execution times with varying batch size on WK and SF

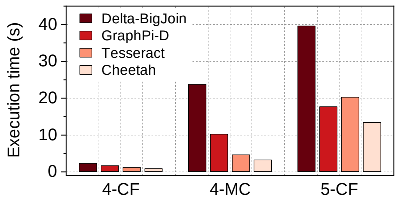


Fig. 14. Execution times for 4-CF, 4-MC, and 5-CF on the irregular graph

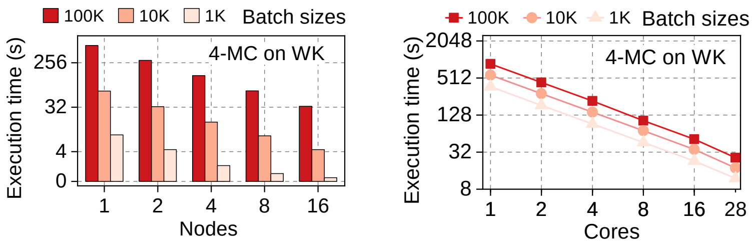


Fig. 15. Execution times with varying node (core) counts

7.6 Sensitivity to Batch Size

To evaluate the impact of batch size on performance, we select the WK dataset from interactive graphs and the SF dataset from social networks, measuring the average execution time per update as the number of edge updates per batch varies. As illustrated in Fig. 13, the longest execution times occur at a batch size of 1K, with 4-FSM on WK averaging 27ms and 5-CF on SF averaging 90ms. As the batch size increases, the average execution time per update decreases, primarily due to the identification and grouping of redundant updates, which enhances mining efficiency (as shown in Fig. 2(a)). Beyond the batch size of 100K, the execution times plateaus, indicating that updates are sufficiently grouped at this point, providing minimal additional performance gains, and further increases in batch size yield diminishing returns in reducing execution time.

7.7 Performance on Irregular Graphs

We conduct an extreme-case evaluation using NetworkX by generating a synthetic graph with 5K vertices and a connection probability of 0.05. We assess the performance of 4-CF, 4-MC, and 5-CF on this synthetic graph. Specifically, we first load 90% of the edges and treat the remaining 10% as insertion updates. Given the presence of locality among updates, our approach remains effective even for irregular graphs. As shown in Fig. 14, Cheetah outperforms Delta-BigJoin, GraphPi-D, and Tesseract by 4.28×, 2.12×, and 1.44×, respectively. While temporal locality in the synthetic irregular graph is weaker compared to real-world graphs, Cheetah effectively captures the remaining locality, reducing redundant accesses and improving overall system performance.

Additionally, our approach may not outperform existing systems in cases of highly irregular updates where no vertices are shared during subgraph expansion, as Cheetah's performance gains rely on exploiting locality in expansion paths among updates. However, such scenarios are rarely encountered in real-world applications, particularly in domains such as social networks, financial systems, and e-commerce, where graph updates inherently demonstrate strong locality [1, 30] due to the structural properties of the data.

7.8 Scalability

Fig. 15 illustrates Cheetah's scalability in performance across different node counts and update batch sizes. Although larger batch sizes increase execution times due to more matches being mined, they ultimately reduce average execution times by providing more opportunities to eliminate redundant memory accesses. Additionally, Cheetah demonstrates a near-linear negative correlation between runtime and the logarithm of both node and core counts, highlighting its efficient scalability. Cheetah achieves this scalability by balancing loads and minimizing access conflicts. It distributes updates from distinct domains across separate nodes, reducing domain interactions and the impact of access conflicts. Moreover, task-stealing mechanisms within nodes enhance execution efficiency by dynamically redistributing workloads to maintain high utilization and performance.

8 RELATED WORK

Most graph mining systems have focused on static graphs [6–15, 38, 39]. Arabesque [6] is the first distributed graph mining system using an embedding-centric model. RStream [7] and Kaleido [40] optimize the embedding-centric model for single-machine environments. Systems as AutoMine [9] and Peregrine [10] adopt a pattern-centric model to reduce memory overhead of storing intermediate results. Subgraph Morphing [13] and DecoMine [12] use compiler-based approaches to minimize redundancy when processing multiple patterns, enhancing efficiency. Contigra [15] further exploits the dependencies between tasks, enabling efficient mining with containment constraints. However, these systems are designed for static graphs, and they may not meet the responsiveness demands of dynamic graphs [41].

While recent research has begun to address the challenges of dynamic graph mining, the issue of redundant memory accesses between updates remains largely unexplored. Systems like TurboISO [18], Turboflux [19], and Delta-BigJoin [20] utilize relational methods to support subgraph queries by treating patterns as relational queries and generating matches through joining edge tables. These efforts [18–20, 42–44] address a specific subset of dynamic graph mining problems involving fixed subgraph queries. Tesseract [21] supports general graph mining by breaking down graph updates into individual mining tasks. PSMiner [36] introduces a pattern-aware accelerator that enhances the speed of set operations in pattern-centric mining by storing partial results, but it is not suitable for the embedding-centric execution model, which includes subgraph extension and isomorphic testing. In contrast, our proposed system, Cheetah, focuses on inter-update interactions and achieves enhanced performance optimizations while supporting existing techniques.

Graph processing, crucial for graph problems, has seen extensive research, particularly in dynamic graph processing to optimize response times for graph-based applications [26, 30, 34, 45–50]. These approaches use incremental techniques to reduce redundant computations by reusing previous results [22, 51], with some grouping updates into batches to further reduce computations [23, 34]. GraPU [45] speeds up batch-update monotonic algorithms by employing component-based classification and in-buffer precomputation. KickStarter [23] ensures accurate incremental computation for monotonic algorithms by using dependence trees. In contrast, GraphBolt [22] introduces a generalized incremental model tailored to support non-monotonic algorithms. However, they primarily focus on vertex-centric computations [52, 53], which are less effective for graph mining algorithms requiring subgraph enumeration [12].

9 CONCLUSION

In this work, we pinpoint the primary causes of memory inefficiency in dynamic graph mining and introduce an exploration domain-driven execution strategy to capitalize on data reuse opportunities overlooked in prior research. We also introduce Cheetah, which utilizes previous matching results and graph updates to efficiently maintain exploration domains. Cheetah groups updates and executes them using a neighbor-centric expansion method to maximize the reuse of expansion paths. Our evaluations show that Cheetah significantly outperforms the leading dynamic graph mining system Tesseract, achieving memory reductions and speedups ranging from 2.17× to 4.38× and 1.10× to 3.31×, respectively.

REFERENCES

[1] Wenfei Fan, Xin Wang, and Yinghui Wu. 2013. Diversified top-k graph pattern matching. In *Proceedings of the VLDB Endowment*, Vol. 6. 1510–1521.

[2] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time constrained cycle detection in large dynamic graphs. In *Proceedings of the VLDB Endowment*, Vol. 11. 1876–1888.

[3] Yuanjing Hao, Long Li, Liang Chang, and Tianlong Gu. 2024. MLDA: A multi-level k-degree anonymity scheme on directed social network graphs. *Frontiers of Computer Science* 18, 2 (2024), 182814:1–182814:16.

[4] Ebberth L. Paula, Marcelo Ladeira, Rommel N. Carvalho, and Thiago Marzagão. 2016. Deep learning anomaly detection as support fraud investigation in brazilian exports and anti-money laundering. In *Proceedings of the International Conference on Machine Learning and Applications*. 954–960.

[5] Alfred Hero and Bala Rajaratnam. 2016. Large-scale correlation mining for biomolecular network discovery. *Big Data Over Networks* (2016), 409.

[6] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for distributed graph mining. In *Proceedings of the Symposium on Operating Systems Principles*. 425–440.

[7] Kai Wang, Zhiqiang Zuo, John Thorpe, Tien Quang Nguyen, and Guoqing Harry Xu. 2018. RStream: Marrying relational algebra with streaming for efficient graph mining on a single machine. In *Proceedings of the Symposium on Operating Systems Design and Implementation*. 763–782.

[8] Tianhui Shi, Mingshu Zhai, Yi Xu, and Jidong Zhai. 2020. GraphPi: High performance graph pattern matching through effective redundancy elimination. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 100:1–100:14.

[9] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing high-level abstraction and high performance for graph mining. In *Proceedings of the Symposium on Operating Systems Principles*. 509–523.

[10] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A pattern-aware graph mining system. In *Proceedings of the European Conference on Computer Systems*. 13:1–13:16.

[11] Xuhao Chen, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2020. Pangolin: An efficient and flexible graph mining system on CPU and GPU. In *Proceedings of the VLDB Endowment*, Vol. 13. 1190–1205.

[12] Jingji Chen and Xuehai Qian. 2023. DecoMine: A compilation-based graph pattern mining system with pattern decomposition. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 47–61.

[13] Kasra Jamshidi, Harry Xu, and Keval Vora. 2023. Accelerating graph mining systems with subgraph morphing. In *Proceedings of the European Conference on Computer Systems*. 162–181.

[14] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana Stefan, Juan Gómez-Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-centric instruction set architecture for graph mining on processing-in-memory systems. In *Proceedings of the International Symposium on Microarchitecture*. 282–297.

[15] Joanna Che, Kasra Jamshidi, and Keval Vora. 2024. Contigra: Graph mining with containment constraints. In *Proceedings of the European Conference on Computer Systems*. 50–65.

[16] Saravanan Alagarsamy, K. Ganesh Varma, K.Harshitha, K. Hareesh, and K. Varshini. 2023. Predictive analytics for black Friday sales using machine learning technique. In *Proceedings of the International Conference on Intelligent Data Communication Technologies and Internet of Things*. 389–393.

[17] Wenfei Fan, Chunming Hu, and Chao Tian. 2017. Incremental graph computations: Doable and undoable. In *Proceedings of the SIGMOD International Conference on Management of Data*. 155–169.

[18] Wook Shin Han, Jinsoo Lee, and Jeong Hoon Lee. 2013. Turbo: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *Proceedings of the SIGMOD International Conference on Management of Data*. 337–348.

[19] Kyoungmin Kim, In Seo, Wook Shin Han, Jeong Hoon Lee, Sungpack Hong, Hassan Chafi, Hyungyu Shin, and Geonhwa Jeong. 2018. TurboFlux: Fast continuous subgraph matching system for streaming graph data. In *Proceedings of the SIGMOD International Conference on Management of Data*. 411–426.

[20] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed evaluation of subgraph queries using worst-case optimal and low-memory dataflows. In *Proceedings of the VLDB Endowment*, Vol. 11. 691–704.

[21] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: Distributed, general graph pattern mining on evolving graphs. In *Proceedings of the European Conference on Computer Systems*. 458–473.

[22] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-driven synchronous processing of streaming graphs. In *Proceedings of the European Conference on Computer Systems*. 25:1–25:16.

[23] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and accurate computations on streaming graphs via trimmed approximations. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 237–251.

[24] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. 2012. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the European Conference on Computer Systems*. 85–98.

[25] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A system for real-time iterative analysis over evolving data. In *Proceedings of the SIGMOD International Conference on Management of Data*. 417–430.

[26] Pourya Vaziri and Keval Vora. 2021. Controlling memory footprint of stateful streaming graph processing. In *Proceedings of the USENIX Annual Technical Conference*. 269–283.

[27] Jure Leskovec and Andrej Krevl. 2014. SNAP datasets: Stanford large network dataset collection. http://snap.stanford.edu/data

[28] Jingji Chen and Xuehai Qian. 2023. Khuzdul: Efficient and scalable distributed graph pattern mining engine. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 413–426.

[29] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: A pattern-aware accelerator for graph pattern mining. In *Proceedings of the Annual International Symposium on Computer Architecture*. 581–594.

[30] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A fast graph update library for FPGA-based dynamic graph processing. In *Proceedings of the International Symposium on Field Programmable Gate Arrays*. 149–159.

[31] Dan Chen, Chuangyi Gui, Yi Zhang, Hai Jin, Long Zheng, Yu Huang, and Xiaofei Liao. 2022. GraphFly: Efficient asynchronous streaming graphs processing via dependency-flow. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 45:1–45:14.

[32] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: An industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* 5, 1 (1998), 46–55.

[33] Pengcheng Yao, Long Zheng, Zhen Zeng, Yu Huang, Chuangyi Gui, Xiaofei Liao, Hai Jin, and Jingling Xue. 2020. A locality-aware energy-efficient accelerator for graph mining applications. In *Proceedings of the Annual International Symposium on Microarchitecture*. 895–907.

[34] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In *Proceedings of the SIGMOD International Conference on Management of Data*. 513–527.

[35] Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A timely dataflow system. In *Proceedings of the Symposium on Operating Systems Principles*. 439–455.

[36] Hao Qi, Yu Zhang, Ligang He, Kang Luo, Jun Huang, Haoyu Lu, Jin Zhao, and Hai Jin. 2023. PSMiner: A pattern-aware accelerator for high-performance streaming graph pattern mining. In *Proceedings of the Design Automation Conference*. 1–6.

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media. In *Proceedings of the International Conference on World Wide Web*. 591–600.

[38] Chuangyi Gui, Xiaofei Liao, Long Zheng, Pengcheng Yao, Qinggang Wang, and Hai Jin. 2021. SumPA: Efficient pattern-centric graph mining with pattern abstraction. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. 318–330.

[39] Jiajun Wu, Chengjian Sun, and Chenyang Yang. 2024. On the size generalizibility of graph neural networks for learning resource allocation. *Science China Information Sciences* 67, 4 (2024), 142301:1–142301:16.

[40] Cheng Zhao, Zhibin Zhang, Peng Xu, Tianqi Zheng, and Jiafeng Guo. 2020. Kaleido: An efficient out-of-core graph mining system on a single machine. In *Proceedings of the International Conference on Data Engineering*. 673–684.

[41] Philippe Fournier-Viger, Ganghuan He, Chao Cheng, Jiaxuan Li, Min Zhou, Jerry Chun-Wei Lin, and Unil Yun. 2020. A survey of pattern mining in dynamic graphs. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 10, 6 (2020), 1–30.

[42] Wenfei Fan, Jianzhong Li, Jizhou Luo, Zijing Tan, Xin Wang, and Yinghui Wu. 2011. Incremental graph pattern matching. In *Proceedings of the SIGMOD International Conference on Management of Data*. 925–936.

[43] Sutanay Choudhury, Lawrence Holder, George Chin, Khushbu Agarwal, and John Feo. 2015. A selectivity based approach to continuous pattern detection in streaming graphs. In *Proceedings of the International Conference on Extending Database Technology*. 157–168.

[44] Jun Gao, Chang Zhou, and Jeffrey Xu Yu. 2016. Toward continuous pattern detection over evolving large graph with snapshot isolation. *The VLDB Journal* 25, 2 (2016), 269–290.

[45] Feng Sheng, Qiang Cao, Haoran Cai, Jie Yao, and Changsheng Xie. 2018. GraPU: Accelerate streaming graph analysis through preprocessing buffered updates. In *Proceedings of the Symposium on Cloud Computing*. 301–312.

[46] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2016. Synergistic analysis of evolving graphs. *ACM Transactions on Architecture and Code Optimization* 13, 4 (2016), 32:1–32:27.

[47] Yiding Liu, Xingyao Zhang, Donglin Zhuang, Xin Fu, and Shuaiwen Song. 2022. DynamAP: Architectural support for dynamic graph traversal on the automata processor. *ACM Transactions on Architecture and Code Optimization* 19, 4 (2022), 60:1–60:26.

[48] Long Zheng, Bing Zhu, Pengcheng Yao, Yuhang Zhou, Chengao Pan, Wenju Zhao, Xiaofei Liao, Hai Jin, and Jingling Xue. 2025. PRAGA: A priority-aware hardware/software co-design for high-throughput graph processing acceleration. *ACM Transactions on Architecture and Code Optimization* 22, 1 (2025), 24:1–24:27.

[49] Chengying Huan, Yongchao Liu, Heng Zhang, Shuaiwen Song, Santosh Pandey, Shiyang Chen, Xiangfei Fang, Yue Jin, Baptiste Lepers, Yanjun Wu, and Hang Liu. 2024. *TEA+*: A novel temporal graph random walk engine with hybrid storage architecture. *ACM Transactions on Architecture and Code Optimization* 21, 2 (2024), 37:1–37:26.

[50] Tianming Zhang, Jie Zhao, Cibo Yu, Lu Chen, Yunjun Gao, Bin Cao, Jing Fan, and Ge Yu. 2025. Labeling-based centrality approaches for identifying critical edges on temporal graphs. *Frontiers of Computer Science* 19, 2 (2025), 192601:1–192601:16.

[51] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-aware incremental processing of streaming graphs. In *Proceedings of the European Conference on Computer Systems*. 83–98.

[52] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.

[53] Jin Zhao, Yu Zhang, Ligang He, Qikun Li, Xiang Zhang, Xinyu Jiang, Hui Yu, Xiaofei Liao, Hai Jin, Lin Gu, Haikun Liu, Bingsheng He, Ji Zhang, Xianzheng Song, Lin Wang, and Jun Zhou. 2023. GraphTune: An efficient dependency-aware substrate to alleviate irregularity in concurrent graph processing. *ACM Transactions on Architecture and Code Optimization* 20, 3 (2023), 37:1–37:24.