G-Thinkerq: A General Subgraph Querying System With a Unified Task-Based Programming Model

Lyuheng Yuan, Guimu Guo, Da Yan, *Senior Member, IEEE*, Saugat Adhikari, Jalal Khalil, Cheng Long, *Senior Member, IEEE*, and Lei Zou

***Abstract*—Given a large graph G, a subgraph query Q finds the set of all subgraphs of G that satisfy certain conditions specified by Q. Examples of subgraph queries including finding a community containing designated members to organize an event, and subgraph matching. To overcome the weakness of existing graph-parallel systems that underutilize CPU cores when finding subgraphs, our prior system, G-thinker, was proposed that adopts a novel think-like-a-task (TLAT) parallel programming model. However, G-thinker targets offline analytics and cannot support interactive online querying where users continually submit subgraph queries with different query contents. The challenges here are (i) how to maintain fairness that queries are answered in the order that they are received: a later query is processed only if earlier queries cannot saturate the available computation resources; (ii) how to track the progress of active queries (each with many tasks under computation) so that users can be timely notified as soon as a query completes; and (iii) how to maintain memory boundedness and high task concurrency as in G-thinker. In this article, we propose a novel TLAT programming framework, called G-thinkerQ, for answering online subgraph queries. G-thinkerQ inherits the memory boundedness and high task concurrency of G-thinker by organizing the tasks of each query using a "task capsule" structure, and designs a novel task-capsule list is to ensure fairness among queries. A novel lineage-based mechanism is also designed to keep track of when the last task of a query is completed. Parallel counterparts of the state-of-the-art algorithms for 4 recent advanced subgraph queries are implemented on G-thinkerQ to demonstrate its CPU-scalability.**

***Index Terms*—Query, graph, subgraph, parallel, task.**

Received 12 December 2022; revised 10 August 2024; accepted 26 January 2025. Date of current version 1 May 2025. The work of Lyuheng Yuan, Da Yan, Saugat Adhikari, and Jalal Khalil was supported in part by DOE ECRP Award under Grant DE-SC0025228, and in part by NSF under Grant OIA-2229394, Grant OAC-2414474, and Grant NSF OAC-2414185. The work of Guimu Guo was supported by NSF under Grant CRII SHF-2245792. The work of Cheng Long was supported by the Ministry of Education, Singapore, through Academic Research Fund under Grant Tier 1 Award (RG77/21). Recommended for acceptance by E.C. Dragut. *(Lyuheng Yuan, Guimu Guo and Da Yan contributed equally to this work.) (Corresponding author: Da Yan.)*

Lyuheng Yuan, Da Yan, and Saugat Adhikari are with the Department of Computer Science, Indiana University Bloomington, Bloomington, IN 47405 USA (e-mail: lyyuan@iu.edu; yanda@iu.edu; adhiksa@iu.edu).

Guimu Guo is with the Department of Computer Science, Rowan University, Glassboro, NJ 08028 USA (e-mail: guog@rowan.edu).

Jalal Khalil is with the Department of Computer Science and Information Technology, St. Cloud State University, St. Cloud, MN 56301 USA (e-mail: jalal.khalil@stcloudstate.edu).

Cheng Long is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798 (e-mail: c.long@ntu.edu.sg).

Lei Zou is with the Wangxuan Institute of Computer Technology, Peking University, Beijing 100871, China (e-mail: zoulei@pku.edu.cn).

The source code of G-thinkerQ is available at https://github.com/lyuheng/TthinkerQ.

Digital Object Identifier 10.1109/TKDE.2025.3537964

I. Introduction

GRAPH data are common in real applications, such as social networks, biological networks [71], and the Semantic Web. It is important to query these graphs to find subgraphs of interest, such as dense social communities or functional groups [32], [68], subgraph matching [51] and length-constrained cycle enumeration for fraudulent activity detection [45].

Graphs in modern applications can easily contain millions of vertices or more, and parallel infrastructure for their efficient processing has been popularly explored in the past decade. Pioneered by Google's Pregel [38], a number of systems have been proposed for simple iterative graph processing [24], [25], [33], [36], [37], [48], [55], [56], [57], [64]. They advocate a think-like-a-vertex (TLAV) programming model, where vertices communicate with each other by message passing along edges to update their states. Computation repeats in iterations until the vertex states converge. In distributed TLAV systems [24], [25], [36], [57], [64], the number of messages transmitted in an iteration is usually comparable to the number of edges, making the execution communication-bound. To avoid communication, single-machine TLAV systems [33], [48] emerge by streaming vertices and edges from disk to memory for batched state updates, but this leads the execution to be disk IO-bound. In fact, several works [23], [44], [59] have noticed that these TLAV frameworks are only efficient for iterative computations where each iteration has $O(n)$ cost and there are $O(\log n)$ iterations, giving a time complexity upper bound of $O(n \log n)$ where n is the input graph size (i.e., $|V| + |E|$). McSherry et al. [41] have also noticed that existing TLAV systems have performance comparable to and sometimes even slower than a single-threaded program.

However, many subgraph queries are compute-heavy and often NP-hard (e.g., dense subgraphs, subgraph matching), so existing IO-bound TLAV systems are simply not the right infrastructure to address them. Moreover, TLAV programs target problems that output one value for each vertex (e.g., PageRanks, single-source shortest paths), but algorithms for subgraph queries operate on subgraphs rather than individual vertices, so it is inconvenient to write TLAV programs to answer subgraph queries [61].

To overcome the latter problem, some pioneering systems began to explore a think-like-a-subgraph programming model,

such as Arabesque [52] and RStream [54]. Despite more intuitive programming interfaces, such interfaces are often too simple for users to properly adapt their existing serial algorithms with complicated but effective pruning rules for advanced subgraph mining problems, and the underlying execution engines are still IO-bound [61]. Unfortunately, these poor design decisions deeply impacted many recent new graph mining systems, and we will discuss more on this in Section III.

To fill the void of CPU-scalable programming frameworks, we proposed the think-like-a-task (TLAT) programming model [4], [62] which targets problems solvable by divide and conquer, oftentimes a recursive algorithm. In the specific domain of graph processing, such problems often operate on subgraphs as the basic units, such as dense subgraph mining and subgraph matching. These problems can be effectively parallelized due to their divisible nature: we can keep decomposing big tasks into smaller ones to create more parallelism opportunities. Examples of successful TLAT systems include G-thinker [19], [26], [27], [32], [61], [63] for dense subgraph mining and subgraph matching, PrefixFPM [18], [46], [65], [66] for frequent pattern mining in transaction databases, TreeServer [60] for training decision trees, T-FSM [70] for frequent subgraph pattern mining in a big graph, and T-DFS [69] for subgraph matching on GPUs.

In particular, G-thinker [61], [63] supports offline subgraph finding and its TLAT programming interface is not algorithm-invasive as is the case for Arabesque and RStream, allowing natural adaptation of serial backtracking mining programs with advanced search-space pruning techniques to maximize code and technique reuse. For example, G-thinker has been used for maximal quasi-clique mining that involves 7 categories of sophisticated search-space pruning techniques [26], [27], [32], which would be difficult to implement in Arabesque.

In this paper, we study subgraph queries which are different from offline subgraph finding. In particular, given a specific type of subgraph queries over an input graph G, the result subgraphs of a query Q are dependent on the content of Q such as the designated members that a dense community should contain, or the query graph for subgraph matching. We design a general parallel TLAT programming framework that can solve different types of subgraph queries. G-thinker is not suitable for online subgraph querying since **(1)** it needs to start a new G-thinker program for each individual subgraph query, which requires loading the same input graph (and building proper indices to speed up computation). While this problem can be solved by revising G-thinker to pre-load the input graph and pin it in memory, **(2)** queries are initiated as independent jobs that cannot see the progress of each other, so the execution order is totally left to the OS to schedule, and a query submitted earlier than another query may be evaluated later, breaking the fairness. **(3)** A third problem is that if many queries are initiated at the same time, their tasks could cause memory to be used up, and there is no coordinator that can control the number of queries that create tasks for evaluation, by holding the evaluation of later queries until resources become available. Optionally, (4) we can evaluate one query at a time using G-thinker, in which case if a query cannot saturate the CPU cores, the idle cores cannot be utilized to process subsequent queries first until the current query is fully evaluated.

To address the above problems, we develop a TLAT query engine called G-thinkerQ, which allows the input graph to be loaded (and indexed) once and reused for subsequent user queries, and which manages the submitted queries to ensure (1) high task concurrency, (2) memory boundedness, (3) fairness, and (4) timeliness of returning results. Our main contributions are:

- G-thinkerQ designs an intuitive TLAT programming interface for users to write parallel algorithms for various subgraph queries, by directly adapting from their serial algorithm counterparts with advanced pruning techniques.
- G-thinkerQ organizes tasks of a query with "task capsule" to guarantee memory boundedness and high task concurrency, and uses a novel task-capsule list to ensure fairness of query evaluation order while supporting memory-bounded concurrent evaluation of multiple queries: *tasks of a later query is processed only if tasks of earlier queries cannot saturate the available CPU cores.*
- A lightweight lineage-based mechanism is designed to keep track of when the last task of a query is completed, so as to return query results in time and to release resources.
- Recent serial algorithms for 4 advanced subgraph queries are parallelized on G-thinkerQ with excellent speedup. G-thinkerQ is on average 65.3× and 8.2× faster than Fractal and Peregrine for subgraph matching queries, 2.1×–8.1× faster than its predecessor offline G-thinker, and 3 to 5 orders of magnitude faster than vertex-centric querying engine Quegel for maximal-clique counting queries.

Paper Organization: Section II reviews the preliminaries of subgraph queries and TLAT paradigm, and Section III reviews the related work. Then, Section IV introduces the programming interface of G-thinkerQ and Section V describes the system design. Section VI briefly presents how to develop four graph querying applications on top of G-thinkerQ, and Section VII reports the experimental results. Finally, Section VIII concludes this paper.

II. Preliminaries

We now illustrate how the TLAT model can parallelize a recursive algorithm using a specific example: finding dense subgraphs.

Dense Subgraph Mining by Set-Enumeration Search: Given a graph $G = (V, E)$, the goal is to find all the dense subgraphs of G that satisfy certain graph density conditions. For a vertex set $S \subseteq V$, let us denote by $G(S)$ the subgraph of G induced by S. Finding valid dense subgraphs $G(S)$ is a problem with a giant search space, since the domain of S is V's power set, which can be organized as a set-enumeration search tree [35]. Many serial recursive mining algorithms conduct depth-first backtracking search on the set-enumeration tree, including the famous Bron-Kerbosch algorithm for mining maximal cliques [16] and k-plexes [74], and the QUICK algorithm for mining maximal γ-quasi-cliques [35].

Fig. 1 shows the set-enumeration tree T for a graph G with four vertices $\{a, b, c, d\}$ where we assume a vertex order $a < b <$

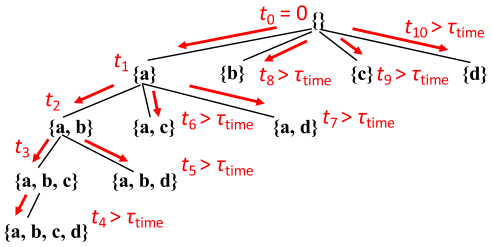


Fig. 1. Set-enumeration tree & timeout-based task decomposition.

Algorithm 1: *Recursive_Mine* $(S, ext(S), t_0)$.

1: Conduct problem-specific pruning over S and $ext(S)$
2: **if** the entire task $\langle S, ext(S)\rangle$ is pruned **do return**
3: **for each** vertex $v \in ext(S)$ **do**
4: create $S' = S \cup \{v\}$ and $ext(S')$
5: **if** $t_{cur} - t_0 \leq \tau_{time}$ **do**
recursive_mine$(S', ext(S'), t_0)$
6: **else** create task $\langle S', ext(S')\rangle$ and add it to system

$c < d$. Each tree node represents a vertex set S, and only those vertices larger than the largest vertex in S are used to extend S. For example, in Fig. 1, node $\{a, c\}$ can be extended with d but not b as $b < c$; in fact, $\{a, b, c\}$ is obtained by extending $\{a, b\}$ with c. Let us denote T_S as the subtree of T rooted at a node with set S. Then, T_S represents a search space for all possible dense subgraphs that contain all vertices in S. In other words, for any dense subgraph $G(S')$ found in T_S, we have $S' \supseteq S$. We represent the task of mining T_S as a pair $\langle S, ext(S)\rangle$, where S is the set of vertices assumed to be already included, and $ext(S) \subseteq (V - S)$ keeps those vertices that can extend S further into a valid dense subgraph.

As we shall see in Section VI, many vertices cannot form a valid dense subgraph together with S and can thus be safely pruned from $ext(S)$, making $ext(S)$ much smaller than $(V - S)$. Note that the task mining of T_S, i.e., $\langle S, ext(S)\rangle$, can be recursively decomposed into tasks $\langle S', ext(S')\rangle$ that mine the subtrees rooted at the children nodes S' of node S in T_S.

While a root task with $S = \emptyset$ mines dense subgraphs in an entire graph G offline, it is often interesting to find dense communities containing a set of query vertices $V' \subseteq V$ online [20], [21], [34], [68], i.e., by using $\langle V', ext(V')\rangle$ as the root task. This now becomes a subgraph query $Q = V'$.

Straggler Elimination by Task Decomposition: While different subgraph queries are independent and can be concurrently processed, we would also like each individual subgraph query to be concurrently evaluated by multiple threads when possible, both to fully utilize idle CPU cores and to reduce query latency.

Taking set-enumeration search for illustration, Algorithm 1 shows a typical recursive mining algorithm, where let us temporarily ignore time variables t_0, t_{cur}, and Line 6, and assume that Line 5 is simply *recursive_mine*$(S', ext(S'), t_0)$. Then this recursive algorithm simply mines the set-enumeration tree in depth-first order. A problem of this approach is that the set-enumeration tree T_S can be huge and processing it recursively by one thread is time-consuming. A natural solution is to decompose it into smaller tasks for concurrent processing, such as Level-1 tasks in Fig. 1 with $S = \{a\}, \{b\}, \{c\}, \{d\}$, respectively. However, this straightforward decomposition may suffer from the straggler problem, as subtree $T_{\{a\}}$ is apparently much larger than $T_{\{c\}}$ in Fig. 1. Even worse, as [26] demonstrates, when mining maximal γ-quasi-cliques, the processing cost of a task cannot be readily predicted from simple features such as vertex and edge numbers and density in $G(S \cup ext(S))$, since such exponential-time-complexity algorithms heavily rely on problem-specific pruning rules (c.f. Algorithm 1 Line 1) to be tractable in practice, and the timing when those rules are applicable changes dynamically during recursive mining depending on the vertex connections and cannot be effectively predicted other than conducting the actual divisible mining.

In the TLAT model, we can use a task timeout strategy to eliminate straggler tasks. Specifically, a task first records the task starting time t_0, and then runs *recursive_mine*$(S, ext(S), t_0)$ as in Algorithm 1, which recursively processes the set-enumeration tree T_S in depth-first order until the task running time exceeds a predefined task-timeout threshold τ_{time} (our default $\tau_{time} = 0.3$s which is tuned and found to work well generally for set-enumeration search), after which the search backtracks by creating new tasks for the remaining workloads rather than mining them recursively by the current thread (c.f. Algorithm 1 Line 6), so that they can be processed in parallel by idle threads. Fig. 1 illustrates how Algorithm 1 works with this timeout strategy: it recursively expands the set-enumeration tree in depth-first order, processing 3 nodes until entering $\{a, b, c, d\}$ for which we find the entry time t_4 times out; we then wrap $S' = \{a, b, c, d\}$ as a new task $\langle S', ext(S')\rangle$ to be added to the TLAT system, and backtrack the upper-level nodes to also add them as new tasks (due to timeout). Note that the new tasks are created at different granularities that are necessary and not over-decomposed (e.g., $\langle\{b\}, ext(\{b\})\rangle$). Clearly, this timeout strategy is applicable to a generic recursive algorithm. It guarantees that each task spends at least a duration of τ_{time} on the recursive computation before dividing the remaining workloads into new tasks (which incurs additional overhead for task creation and scheduling).

III. Related Work

Vertex-Centric Systems: As Section I has explained, TLAV frameworks are ill-suited for compute-heavy subgraph querying that we study here. A detailed review of TLAV systems is beyond our scope and please refer to [14], [28], [37], [40], [55], [67], [72].

While no work has ever considered the TLAT model for online querying, our Quegel system [58], [73] extended the TLAV model of Pregel [38] to answer iterative graph queries, but Quegel is not suitable for compute-heavy subgraph queries.

Graph-Centric Systems: Realizing this limitation, many graph-parallel systems were designed to directly operate on subgraphs. Different from TLAV systems, these systems adopt a think-like-a-graph (TLAG) programming model, but they still

suffer from IO-bound execution. Specifically, NScale [47] constructs candidate subgraphs using multiple rounds of MapReduce, leading to large amounts of data shuffling. Arabesque [52] and RStream [54] advocate a simple programming model that expands the set of subgraphs with i edges/vertices by one more adjacent edge/vertex, to construct subgraphs with $(i+1)$ edges/vertices for processing. New subgraphs that pass a filter-condition are further processed and then passed to the next iteration. For example, to find cliques, the filter-condition checks whether a subgraph g is a clique; if so, g is passed to the next iteration to grow larger cliques. Obviously, such breadth-first subgraph exploration strategy materializes a huge amount of intermediate subgraphs; such cost does not exist in serial depth-first backtracking algorithms such as the Bron-Kerbosch algorithm [16] for mining maximal cliques. Arabesque [52] is a distributed system, and RStream [54] eliminates communication by utilizing relational joins for evaluation in a single machine to achieve better performance, but it is not multi-core friendly.

Fractal [22] aims to overcome the above breadth-first subgraph exploration issue by treating each subgraph expansion (by a vertex/edge) as a basic unit for task scheduling, so that depth-first subgraph expansion can happen to avoid unnecessary subgraph materialization to keep memory usage bounded. However, it forces users to reformulate subgraph mining algorithms using its three primitives: extension, aggregation and filtering, which is not always easy for advanced subgraph mining problems requiring sophisticated indexing and pruning techniques such as those we study in Section VI. Peregrine [31] adopts a pattern-based programming model that treats graph patterns as first class constructs, which enables Peregrine to extract the semantics of patterns to guide its exploration. While the interface is suitable for pattern-based problems, it is difficult to write pattern-based algorithms for problems like mining quasi-cliques and hop-constrained s-t path enumeration which we study in Section VI.

In summary, existing TLAG systems attempt to hide parallel execution details from end users, by specifying various types of graph-centric programming models. They force users to write their graph mining problems using the specific interfaces, which may not be flexible enough to cover all kinds of graph queries; and even this is if possible, the cost of learning a new programming model and reformulating a serial algorithm counterpart with advanced pruning techniques accordingly is very high. Finally, the execution efficiency may still be unsatisfactory and IO-bound due to the need of subgraph materialization.

Task-Centric Systems: The TLAT model was originally proposed in [4], [62] as an abstract computation model for parallelizing compute-intensive problems. Then, G-thinker [61] was developed as the first truly CPU-scalable framework for subgraph finding designed based on the TLAT model, and it beats existing TLAG systems by up to two orders of magnitude in speed, and scales to graphs two orders of magnitude larger. Among the other TLAT systems, G-Miner [17] was based on an earlier prototype of G-thinker but has a poor task scheduling scheme. PrefixFPM [18], [65], [66] is a task-centric system to mine frequent patterns (sequences, subgraphs, subtrees and matrices) in transactional settings. TreeServer [60] is a task-centric system to train models based on many decision trees. T-FSM [70] is a task-centric system for frequent subgraph mining in a big graph. T-DFS [69] is a task-centric system for subgraph matching on GPUs.

However, none of the above TLAT systems target the querying workloads. While G-thinker is a distributed system, our G-thinkerQ framework is currently implemented as a prototype for a single-machine shared-memory environment, but it can be easily extended for distributed processing following a similar design as in G-thinker, by maintaining vertices of an input graph in a distributed key-value store for tasks to request the necessary data. Back to G-thinker, Guo et al. [26], [27] and Yan et al. [63] further integrated the timeout strategy to improve load balancing in G-thinker for the problems of maximal quasi-clique mining and maximum clique finding, respectively. Recently, Khalil et al. [32] redesigned G-thinker into a single-machine environment and called the resulting framework as T-thinker, which is more accessible to an average user without a distributed cluster. G-thinkerQ reuses the well-tuned task queueing structure of the offline T-thinker framework [32] but devises new fair multi-query task scheduling and query progress tracking techniques to address the challenges introduced in Section I.

IV. Programming Interface

Overview: We first introduce some concepts necessary to understand G-thinkerQ, including ***task, comper*** and ***worker***. Fig. 3 illustrates these concepts in G-thinkerQ's system architecture.

Specifically, G-thinkerQ follows a client-server architecture: users submit their queries to the client programs, which then send them to the server for evaluation. G-thinkerQ supports an arbitrary number of clients: a user may directly type each query in a client console, or submit a file containing a batch of queries. A user can submit a special query string "server_exit" to notify the server to terminate. The server will ignore all queries received after "server_exit," and terminate as soon as all queries received before "server_exit" finish their evaluation.

The server program consists of a master thread called ***worker*** and a number of computing threads called ***compers***. Initially, the worker loads the input graph, conducts initial graph pruning and indexing (if applicable), and then receives the incoming graph queries for processing by the compers. Each comper is a thread that keeps fetching the next task for processing if available, or sets its state to idle otherwise. The worker periodically checks if there are still tasks to be processed and if so, it wakes up idle compers to process them. The server terminates if after a "server_exit" message has been received, the worker finds that all compers become idle.

Each query is treated initially as a ***root task*** that gets fetched by an available comper to process, and a task may create new tasks of smaller workloads for concurrent processing (c.f. the timeout strategy illustrated by Fig. 1). Hence, each query may correspond to many tasks that are processed by different compers. As we shall see in Section V, G-thinkerQ provides a lineage-based mechanism to track when all the tasks of a query Q have finished so that users can be notified to check the results of Q (either printed on the console or written by compers on disk). As we

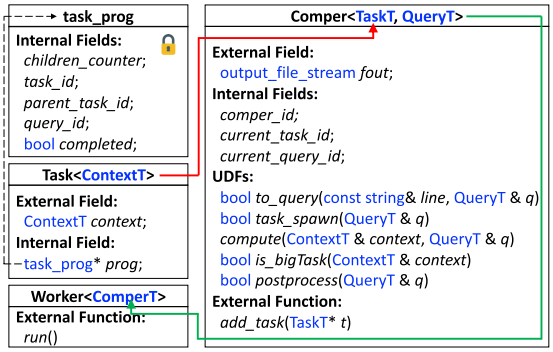


Fig. 2. Programming interface of G-thinkerQ.

shall see in Section VI, some types of queries may need multiple phases of evaluation, so when one phase ends, our API allows a query to create a new root task to start the next phase rather than returning results to users.

Programming Interface: G-thinkerQ is written in C++. It defines a set of base classes, each associated with some template arguments. To write an application program, a user only needs to properly specify the data types and implement user-defined functions (UDFs) according to application logic.

Fig. 2 summarizes these base classes, including their external fields and functions for users to use when writing their applications, and some important internal fields that are transparent to users. We list internal fields to help readers understand our system design in Section V, and application programmers can safely ignore them. As an overview, *a user only needs to specify the logic of two UDFs:* (1) *task_spawn*(q), which specifies how to create a root task from a user-provided query content q, and (2) *compute*$(context, q)$, which specifies how a task belonging to query q computes by reading from and writing to its context, the type of which is also flexible for users to specify. UDF *compute*(.) is exactly where users write their recursive algorithm (e.g., Algorithm 1 for set-enumeration search) usually directly adaptable from the serial counterpart with advanced pruning techniques, and if the task times out, one may call *add_task*(t) to add a child task t to the system. Clearly, our API (1) is general and intuitive for parallelizing any recursive algorithms with advanced pruning rules, and (2) does not require users to learn a new parallel programming model as in existing subgraph-centric systems such as Arabesque [52], RStream [54], Fractal [22] and Peregrine [31].

We next describe the API of Fig. 2 in detail. Specifically, Task<ContextT> defines the data type of a task. Each "Task" object keeps a user-specified field *context* to hold the content that a task needs during its computation, which is of a user-specified type <ContextT>. A task object t also keeps another progress object *prog* of type "task_prog" (transparent to programmers), which not only tracks t's information such as its own task ID and the corresponding query ID, but also other information necessary for lineage-based query progress tracking which we shall explain in detail in Section V.

Comper<TaskT, QueryT> in Fig. 2 is the class that implements a comper (i.e., computing thread). This is G-thinkerQ's most important base class since it provides 5 UDFs for users to specify the application logic. Notably, "Comper" is the only class for which users need to define a subclass to implement UDFs. The other base classes have no UDF so users only need to specify the template arguments and to rename the new type using "typedef" for ease of use. Users need to specify two types for the "Comper" class: (1) <TaskT> which is the user-specified "Task" class, and (2) <QueryT> which is a user-specified type for a query object. The first UDF *to_query*$(line, q)$ specifies how to parse a query string submitted by a user into a query object q of type <QueryT>. For example, if we want to find a dense subgraph containing vertices in vertex set $S = \{a, c\}$, then *line* can be a string "a c" and <QueryT> can be a vertex set to keep S. Besides the query content, <QueryT> may also keep other information global to a query to be shared by all its tasks, such as the set of best results currently found for pruning purposes when we search for the best result or top-k results. The UDF returns *true* if *line* is a valid query string, and *false* otherwise in which case the worker at the server rejects this query.

Once we get a query object q, inside UDF *task_spawn*(q) we then specify how to create a root task t_{root} out of q, and we call Comper's *add_task*(t_{root}) function to add it to the system to be scheduled for computation. We return *true* in UDF *task_spawn*(q) if a root task is successfully added, but in certain conditions q can be directly pruned or its results can be directly found without further evaluation, in which case we return *false* to signal the system to release the resources allocated for q and to notify users about the query outcome.

In G-thinkerQ, a comper calls *task_spawn*(q) to process a new query q if it cannot get any task from the existing task pool (see Section V). Note from Fig. 2 that each comper also maintains a file stream *fout* for users to write query results in UDFs when needed. Before a comper calls *task_spawn*(q) to create root task for a new query q, it first creates a directory $\mathcal{F}_q$ of files, one for use by each comper i, denoted by f_q^i. When a comper i processes a task that belongs to query q, it will first assign the output stream of f_q^i to *fout* before calling a UDF so that users can use *fout* to write results to f_q^i in the UDF. When a query q finishes evaluation, its results can be obtained by concatenating all files under $\mathcal{F}_q$. Note that users can write results using *fout* in UDF *task_spawn*(q) even if it does not generate a root task and returns *false*.

After a comper gets a task t, it calls the third UDF *compute*$(context, q)$ by passing in t's content and t's corresponding query object. This UDF is where a user implements the divide-and-conquer algorithm for subgraph querying (e.g., Algorithm 1). In this UDF, results can be written using *fout* and new tasks can be added by calling *add_task*(.). When a comper calls *add_task*(t), the function uses the fourth UDF *is_bigTask*$(context)$ to determine if the task t is a big one or a regular one using t's content $context$. As we shall see in Section V, big tasks have their dedicated task queues so that they are prioritized in scheduling (e.g., for early decomposition by the timeout strategy) to improve load balancing among compers. Unlike the 3 earlier UDFs that are pure virtual functions, *is_bigTask*(.) is a virtual function that returns *true* by default,

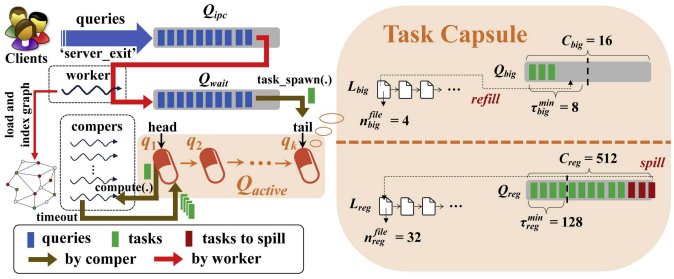


Fig. 3. System architecture diagram.

which basically treats all tasks as big ones if users do not override it with their own implementation.

The last UDF *postprocess*(q) is called once G-thinkerQ detects that all tasks associated with q have finished evaluation, in which case we say that one phase of q finishes; in this UDF, we then determine whether q requires another phase. If so, we call *add_task*(.) to add a new root task created based on q's latest content saved in the QueryT object, and return *true*; otherwise, the query finishes and we return *false* to signal the system to release resources for q and to notify results to users.

Once the subclass of "Comper" is properly defined, we then pass this type to the "Worker" base class, and call its *run*() function (see Fig. 2) to start the query server engine.

V. System Design

Architecture Overview: Fig. 3 overviews G-thinkerQ's system architecture. Specifically, queries are submitted by client programs into an interprocess message queue Q_{ipc} (implemented with Linux IPC), and the worker of the server program periodically probes Q_{ipc} and appends its queries to a query-waiting queue Q_{wait} where the compers may then fetch queries for processing when they become available.

Each query string gets assigned a query ID when the worker moves it from Q_{ipc} to Q_{wait}, which we simply use the next unassigned integer. Therefore, for two queries q_i and q_j with IDs $i < j$, we know that q_i is received before q_j. G-thinkerQ schedules q_i's tasks for computing before q_j's tasks, which respects the first-come, first-served principle. However, different queries have different workloads so they may finish out of order: for example, when all q_i's tasks are being processed by compers, the tasks of q_j could be fetched by other compers to process so that q_j finishes early before some tasks of q_i time out and generate new subtasks.

Fig. 3 shows a list of active queries $q_1, q_2, \ldots, q_k$ currently under evaluation by the compers. We denote this active query list by Q_{active}. Recall that Q_{wait} holds those pending queries waiting to be fetched for processing, so they are not yet active. To respect the time order of receiving queries, a comper always attempts to obtain the first available task starting from Q_{active}'s head for processing; when the comper fails to get any task from Q_{active}, it may dequeue a query q from Q_{wait} to attempt to spawn a root task for processing, which will move q to Q_{active}'s tail (i.e., after those active queries received earlier than q).

For each active query q_i in list Q_{active}, its corresponding element keeps (1) q_i's query ID (indicating its temporal priority), (2) q_i's query object which is set by UDF *to_query*(.), and (3) a memory-bounded container structure to keep the tasks of q_i, which we call a ***task capsule***. Fig. 3 shows the capsule structure on its right (details to introduce later). In a nutshell, a task capsule spills superfluous tasks resulted from decomposing big tasks to the disk for later processing to keep memory usage bounded, and it prioritizes big tasks for scheduling before regular tasks.

To keep the memory cost bounded, we allow a maximum capacity of C_{active} active queries ($C_{active} = 10$ by default) to be in Q_{active}. So after a comper fails to get any task from Q_{active}, it will go idle either directly if $|Q_{active}|$ is already C_{active}, or otherwise, it will fetch a new task from Q_{wait}, and go idle only if Q_{wait} is empty (the comper will be awakened later by the worker to process tasks when more tasks are detected).

In the sequel, we introduce some key designs of G-thinkerQ's task execution engine, including (1) task ID assignment, (2) task capsule's design, (3) how a comper obtains a task for processing, (4) lineage-based query progress tracking.

Task ID Assignment: G-thinkerQ assigns each task a unique ID so that a query can track its task pool without ambiguity (recall that Q_{active} has multiple active queries).

In G-thinkerQ, tasks are created by compers in three places. (1) In UDF *task_spawn*(q) when a comper failed to get any active task from Q_{active}, in which case it uses UDF *to_query*(.) to obtain a query object q from the dequeued query string in Q_{wait} and attempts to create a root task for q. (2) In UDF *compute*(.) where the current task is decomposed into smaller tasks. (3) In UDF *postprocess*(q) when query q's last task finishes its *compute*(.) call, where we may create a root task for the next phase of q. In the UDFs, we call *add_task*(t) to add each new task t, inside which t is assigned a new task ID.

Since tasks can be assigned IDs from different compers, one challenge is to ensure that these task IDs do not conflict with each other. For this purpose, each comper maintains a sequence no. n_{seq}. Whenever it adds a new task t, it associates t with a 64-bit task ID $id(t)$ which concatenates a 16-bit comper ID with the 48-bit n_{seq}, and n_{seq} is then incremented for use by the next task to be added by the same comper.

Task Capsule: In [32], a single-machine parallel counterpart of G-thinker was designed to perform TLAT computation for offline graph mining, called T-thinker, which uses a memory-bounded task container design. Our G-thinkerQ is different from T-thinker in that we now have multiple active queries, each requiring a memory-bounded task container. As Fig. 3 shows, each active query is associated with a task capsule, which essentially wraps the task containers of T-thinker [32]. We reuse this exact same design to hold the tasks for each individual query, since T-thinker's task containers have been tuned to ensure both high concurrency and memory efficiency. Below, we briefly introduce these containers in a capsule; please refer to Section 5.3 of [32] for more details.

Recall that UDF *Comper*::*is_bigTask*(.) determines if a task to add is big or not. It is desirable to schedule big tasks early, so that they can be computed and decomposed earlier to improve load balancing. Therefore, each capsule of a query q_i maintains two

queues Q_{big} and Q_{reg}, one for big tasks and the other for regular tasks (see Fig. 3). To be memory bounded, Q_{big} (resp. Q_{reg}) has a maximum task capacity. However, it is possible for a big task to generate many decomposed tasks to be inserted into Q_{big} and Q_{reg}, causing either queue to overflow. To keep the number of in-memory tasks bounded, if Q_{big} (resp. Q_{reg}) is full but a new task is to be inserted, we spill a batch of tasks at the end of Q_{big} (resp. Q_{reg}) as a file to local disk to make room. Note that tasks spilled from Q_{big} (resp. Q_{reg}) are written to the disk (and loaded back later) in batches to achieve serial disk IO. We use a file list $\mathcal{L}_{big}$ (resp. $\mathcal{L}_{reg}$) to track those files spilled from Q_{big} (resp. Q_{reg}) to be loaded back to Q_{big} (resp. Q_{reg}) later when it needs a task refill. Task spilling is automatically handled by *add_task*(t). Also, whenever a comper that checks Q_{big} (resp. Q_{reg}) for task fetching finds that there are less than τ_{big}^{min} (resp. τ_{reg}^{min}) tasks in Q_{big} (resp. Q_{reg}), it will refill tasks from a task file. Here, τ_{big}^{min} and τ_{reg}^{min} are tunable system parameters.

The Algorithm of a Comper: A comper always keeps fetching and processing the next task that it can obtain from Q_{active}, or if not available, a root task generated from Q_{wait}, until the moment when such a task cannot be found, in which case it sets its state to idle. An idle comper will be periodically awakened by the worker, which either signals it to process newly available tasks, or signals it to terminate the task probing loop if the server program is terminated by the "server_exit" message from a client. When a comper attempts to fetch a task, it first examines the list Q_{active} starting from its head so that the tasks of those queries that arrive early are prioritized in fetching. For each task capsule q_i in Q_{active}, we first try to obtain a big task. If $q_i.Q_{big}$ is empty, or it is being accessed by another comper (i.e., a try-lock failure), we then try to obtain a regular task by checking $q_i.Q_{reg}$. If $q_i.Q_{reg}$ is still empty, or try-lock fails over $q_i.Q_{reg}$ (as it is accessed by another comper), we then move on to the next task capsule q_j in Q_{active} and repeat the same process until a task is successfully fetched. As an exception, instead of try-locking $q_k.Q_{reg}$ of the last task capsule q_k in Q_{active}, a comper blocks on $q_k.Q_{reg}$ until it gets the lock, after which it dequeues a task from $q_k.Q_{reg}$ for processing. However, if $q_k.Q_{reg}$ is found to be empty (and there is no spilled task file for refill) but the capacity of Q_{active} permits another query, the comper will then try to spawn a root task from a query q dequeued from Q_{wait}, and to allocate resources for q such as (i) a task capsule to be added to Q_{active}'s tail and (ii) file output streams for writing results. Otherwise, the comper goes idle to be awakened by the worker in its next state probe.

Query Progress Tracking: One challenge remains: how to determine the timing when a query has finished its evaluation? The problem does not exist in an offline TLAT system like T-thinker [32], where a task object (including its associated content data such as its subgraph) is freed from memory as soon as it finishes computation, and the task will hence lose track of the children tasks that it creates. This is a good design since data of those processed tasks are no longer needed and hence timely garbage collected, while the main thread can easily determine that an offline job finishes if it finds that all computing threads become idle during a periodic probe. In contrast, G-thinkerQ needs to standby all the time (unless notified to terminate by message "server_exit"), and to answer queries coming on demand. As a result, G-thinkerQ needs a mechanism to know when a query finishes its evaluation so that it can notify users with the query results in time, and release the resources occupied by q.

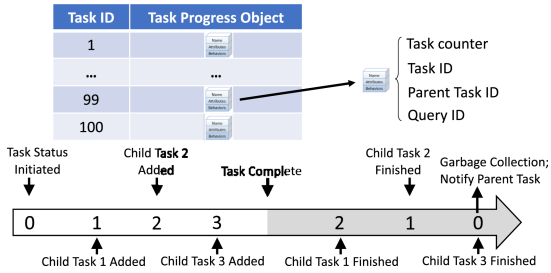


Fig. 4. Query progress bookkeeping by task lineage tracking.

We propose a task lineage tracking approach which associates each task object t with a task progress object $t.prog$ of type *task_prog*, as shown in Fig. 2. Note that $t.prog$ is an internal field used by the G-thinkerQ system code but transparent to application developers. It is automatically created when a user calls *add_task*(t), and it tracks whether t and all its descendant tasks have finished, after which it is automatically deleted. Note that (1) the progress object of q's root task tracks whether q has finished one phase of evaluation; and (2) $t.prog$ can exist after t is deleted, since it needs to wait till all descendant tasks have finished (before it is deleted).

For this reason, a task object t only maintains a pointer to the actual progress object (see Fig. 2), and the set of all progress objects are actually maintained in a concurrent hash table $\mathcal{T}_{prog}$ shown on the top of Fig. 4, where we can retrieve the progress object (of type "task_prog*") of a task object t using its task ID (even after t is deleted). Unlike a task object t that keeps a large amount of data $t.context$ for task processing, a task progress object only keeps lightweight status fields such as a child-task counter, task ID, parent task's ID, the associated query ID, and a flag indicating whether task t itself has finished (see Figs. 2 and 4), so the memory cost of keeping $\mathcal{T}_{prog}$ for task lineage tracking is low.

The life cycle of a task progress object is shown on the bottom of Fig. 4. Specifically, when a task t (and its progress object) is created, the child-task counter is initialized as 0. Whenever a child task is created by t, the counter is incremented. When t finishes its call of *compute*(.), we set the complete-flag of its progress object. Also, when a child-task finishes, the counter is decremented. When the counter is decremented to 0 and complete-flag is also set, we delete t's progress object and remove its entry from $\mathcal{T}_{prog}$, and update the progress object of its parent task in $\mathcal{T}_{prog}$ by decrementing its child-task counter; if the parent task's counter becomes 0, the same process propagates to the parent of this parent task, and this propagation may continue all the way up to the root progress object in which case q is considered as have finished its current phase. Note that this end-condition propagation may also be triggered by

Algorithm 2: *Backtrack*(Task_Prog* $prog$, QueryT & q).

```
1:  prog->completed ← true
2:  if prog->children_counter > 0 return
3:  Repeat
4:    erase table entry T_prog[prog->task_id]; delete prog
5:    if prog->parent_task_id = −1 break //root task
6:    prog ← T_prog[prog->parent_task_id]
7:    decrement prog->children_counter by 1
8:    if !prog->completed or prog->children_counter > 0
9:      return
10: if postprocess(q) = false
11:   get q's capsule from Q_active and remove it from
      Q_active
12:   delete its task containers and close its output streams
13:   notify client about the query results
```

the completion of task t itself, if all child-tasks finish before t completes (or if no child-task is created).

Algorithm 2 describes the above backtracking process, and a comper calls this member function *backtrack*($t.prog$, q) right after it finishes *compute*($t.context$, q). Note that a comper first needs to obtain t for computation from its corresponding query's capsule in Q_{active}, so query object q is directly taken from this capsule and passed to *compute*(.) and *backtrack*(.).

Specifically, since t has finished *compute*(.), Line 1 first sets the complete-flag for t. If there are still unfinished child-tasks, the function returns in Line 2. Otherwise, t and all its descendant tasks have finished so Line 4 deletes t's progress object. If t is not a root task (checked by Line 5), Line 6 updates $prog$ to be the progress object of t's parent task, and Line 7 decrements the child-counter of $prog$ to reflect t's completion. If this parent task and its descendants are also complete (checked in Line 8), backtracking continues upwards until we reach q's root task (whose parent task ID is assigned as a special value -1). If the root task is reached (c.f., "break" in Line 5), then q has finished one phase so Line 10 calls UDF *postprocess*(q) to determine if another phase is needed (which also produces a new root task for q if so). If *postprocess*(q) returns *false*, q is complete so Lines 11–13 release the resources allocated for q.

Other Technical Details: While Line 7 of Algorithm 2 handles the decrement of child-task counter, the counter increment is automatically handled by *add_task*(t') which is called inside UDF *compute*($t.context$, q). Here, t' is a child task of t, and we simply increment $t.prog$->*children_counter* by 1.

Also, $t.prog$ can be simultaneously updated and checked in Lines 1–2 by a comper that computes t, and in Lines 7–9 by a comper that computes a descendant task of t and backtracks upwards. Therefore, we add a lock for each $t.prog$ to protect Lines 1–2 and Lines 7–9 as two critical sections. Otherwise, if t's comper executes Line 1 to mark t as complete, and another comper running the last child task of t executes Line 7 to decrement t's child-task counter to 0, then both compers will backtrack from t upwards (as the conditions in both Line 2 and Line 8 are *false*), causing 'double deletion.' Also note that in UDF *compute*($t.context$, q), whenever we call *add_task*(t'), the function should create $t'.prog$ with its parent task ID set as t's task ID, and increment the child-task counter of $t.prog$ by 1. In contrast, in UDFs *task_spawn*(q) and *postprocess*(q), *add_task*(t_r) adds the root task t_r so the parent task ID of $t_r.prog$ should be set as -1 (as required by Line 5 of Algorithm 2). Therefore, before calling a UDF, a comper sets a flag indicating the UDF context so that when *add_task*(.) is called therein, it knows which of the above 2 branches to take automatically based on this UDF-context flag.

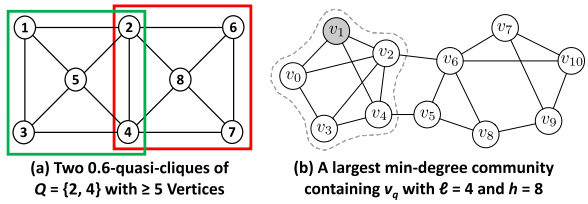


Fig. 5. Illustration of dense subgraph queries.

Finally, since all compers repeatedly access Q_{active} to obtain a task starting from its head, to ensure high concurrency of accessing Q_{active}, we protect it with a read-write lock rather than a mutex. This is because compers check Q_{active} frequently which is read-only, and the updates to Q_{active} only happen infrequently either (i) when a comper cannot find any task from Q_{active} so it dequeues a new query from Q_{wait} for processing, or (ii) when a comper backtracks and finds that a query finishes (c.f. Line 11 of Algorithm 2); in either of the two cases, we write-lock Q_{active}.

VI. Applications

This section illustrates the use of G-thinkerQ by implementing 4 applications on top, which are the very recently studied subgraph queries proposed in top venues [32], [45], [51], [68].

Maximal Quasi-Cliques [32]: Given a minimum degree threshold $\gamma \in [0, 1]$, a γ-quasi-clique is a subgraph $g = (V_g, E_g)$ where each vertex v connects to at least γ fraction of the other vertices in g, i.e., $deg(v) \geq \lceil \gamma \cdot (|V_g| - 1) \rceil$. Following [32], we request a result subgraph to be maximal and to contain at least τ_{size} vertices. We additionally require a result subgraph to contain all vertices in a query set Q, which essentially starts with $S = Q$ in Algorithm 1. Fig. 5(a) gives a query with 2 results shown in red and green.

Here, <QueryT> is a triplet (γ, τ_{size}, Q), and we assume $\gamma \geq \gamma^{min}, \tau_{size} \geq \tau_{size}^{min}$ for all queries, where γ^{min} and τ_{size}^{min} are user-defined density and size lower bound for queries. It is easy to see that a result subgraph cannot contain a vertex v with $deg(v) < \lceil \gamma^{min} \cdot (\tau_{size}^{min} - 1) \rceil \triangleq k^{min}$, so after the input graph G is loaded, we will first shrink it into its k^{min}-core before calling *Worker*::*run*(), which takes linear time [39].

As shown in [42], a result quasi-clique has a diameter upper bound $UB(\gamma)$, and $UB(\gamma) = 2$ when $\gamma \geq 0.5$ as we assume here. Therefore, in *task_spawn*(q), we return *false* without creating a root task if any two vertices in Q are > 2 hops away, or any vertex in Q has degree $< \lceil \gamma \cdot (\tau_{size} - 1) \rceil \triangleq k$. Otherwise, we create the root task by setting $S = Q$ and including into $ext(S)$ only those vertices of G that are within 2 hops to every

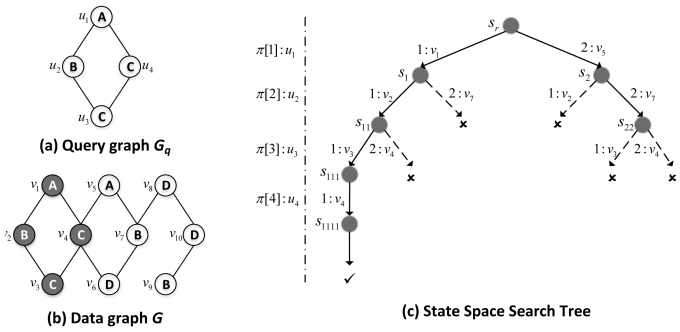


Fig. 6. Illustration of subgraph matching (from [50]).

Algorithm 3: *Subgraph_Match*$(G_q = (V_q, E_q), G = (V, E))$.

1: generate matching order $\pi = [u_1, u_2, \ldots, u_{|V_q|}]$ $(u_i \in V_q)$
2: *enumerate*$(\emptyset, 1, \pi, G_q, G)$
Procedure *enumerate*(S, i, π, G_q, G)
3: $C_S(u_i) \leftarrow$ viable vertex candidates in G to match u_i
4: **for each** $v \in C_S(u_i)$
5: Append S with (u_i, v)
6: **if** $|S| = k$ **then** output S
7: **else** *enumerate*$(S, i+1, \pi, G_q, G)$
8: Pop (u_i, v) from S

vertex in Q. The root task also materializes the induced subgraph $G(Q \cup ext(Q))$ (actually its k-core) to mine upon to avoid scanning the long adjacency lists of G.

In UDF *compute*(.), a task runs the algorithm of [32] which follows Algorithm 1's framework but with 7 categories of pruning rules to aggressively reduce the search space. UDF *is_bigTask*(.) determines if a task $\langle S, ext(S)\rangle$ is big by checking if $|ext(S)| \geq \tau_{split}$, since $|ext(S)|$ upper-bounds the number of levels to continue search in the set-enumeration tree. Here, τ_{split} is a user-defined parameter tuned to be 200 by default for set-enumeration search. Finally, this query has only one phase so we use the default *postprocess*(q), which returns *false* to terminate the query.

Size-Bounded Community [68]: This query finds a subgraph with the largest min-degree among all connected subgraphs g that contain the query vertex v_q and have $\ell \leq |V_g| \leq h$. For example, Fig. 5(b) shows a toy graph upon which we issue query $(v_q = v_1, \ell = 4, h = 8)$ and finds the best community $\{v_0, v_1, v_2, v_3, v_4\}$ which has minimum degree 3. This query is also answered by set-enumeration search [68] but we only find one best subgraph which we denote by H hereafter.

We define <QueryT > to include not only $q = (v_q, \ell, h)$, but also the current best result subgraph H and its associated min-degree d_{min} for pruning unpromising search space. Since a query could be processed by multiple tasks, < QueryT> keeps a read-write lock to protect H and d_{min} since every task needs to access them to check pruning conditions, but only when a better subgraph is found will they be updated which happens not very frequently.

In [68], H is initialized by a fast algorithm that greedily selects vertices around v_q to maximize subgraph min-degree for effective pruning. Let $cn(v)$ be the core number of a vertex v, then a subgraph better than H must have every vertex with degree $> d_{min}$, so we shrink G into its $(d_{min}+1)$-core.

In UDF *task_spawn*(q), we first conduct the above preprocessing. Note that the best subgraph has a min-degree upper-bounded by $UB = \min\{cn(v_q), h-1\}$ since v_q cannot be contained in a subgraph with min-degree $> cn(q)$. If the min-degree lower bound d_{min} also equals UB, then the initial H is already the best and we simply output H and return *false* without creating a root task. Otherwise, we create and add a root task $\langle S, ext(S)\rangle$ with $S = \{v_q\}$ and return *true*.

UDF *compute*(.) runs the algorithm of [68] which follows Algorithm 1's framework but with advanced pruning, where Line 3 now adopts the smarter domination-based branching technique proposed in [68] to reduce the recursion tree fanout.

Finally, UDF *is_bigTask*(.) is implemented exactly as in our maximal quasi-cliques application, and UDF *postprocess*(q) outputs H and then returns *false* to end the query evaluation.

Subgraph Matching [51]: Algorithm 3 sketches Ullmann's recursive algorithm [53] for subgraph matching, which, given a query graph G_q, retrieves all subgraphs of a data graph G that are isomorphic to G_q. There, we match data vertices to query vertices $u_1, u_2, \ldots, u_k$ one at a time, with S recording the current partial match. The search process can be depicted by a state space search tree [50] in Fig. 6, where the rightmost path gives a recursion path $S = [(u_1, v_5), (u_2, v_7), (u_3, v_4)]$. This path failed since there is no edge (v_7, v_4) to match query edge (u_2, u_3).

A number of improvements have been proposed on top of Ullmann's algorithm. They are summarized by [51] with the best combination recommended and followed by us.

Given G_q, first consider UDF *task_spawn*(G_q). We first create an auxiliary data structure $\mathcal{A}_q$ as proposed by DP-iso [29], which maintains a list $C(u_i)$ of data vertex candidates for each query vertex u_i, and which maintains those edges of E between candidates in $C(u_j)$ and those in $C(u_k)$ if $(u_j, u_k) \in E_q$. The index $\mathcal{A}_q$ is fast to build and effectively filters out unpromising vertices v from each $C(u_i)$ simply based on the local neighborhood of v in G. Subgraph enumeration in Algorithm 3 is directly conducted on the smaller $\mathcal{A}_q$ rather than G, and the recursion fanout in Line 4 is significantly reduced. We also generate query-vertex matching order π following GraphQL [30], which picks u_i to be a neighbor of at least one vertex in $\{u_1, u_2, \ldots, u_{i-1}\}$ in G_Q that has the smallest $|C(u_i)|$ to reduce fanout in the upper levels of recursion. Both G_q and the computed $\mathcal{A}_q$ and π are kept in <QueryT> for use by G_q's enumeration tasks, and we create and add a root task with $\langle S, i\rangle = \langle \emptyset, 1\rangle$ as in Line 2.

Note that if we find $C(u_i) = \emptyset$ after constructing $\mathcal{A}_q$, *task_spawn*(G_q) returns *false* immediately without computing π and creating root task, as there is no subgraph matching G_q.

Then, UDF *compute*(.) runs Algorithm 3 to compute each task $\langle S, i\rangle$, but creates new tasks $\langle S \oplus (u_i, v), i+1\rangle$ rather than recursively calling *enumerate*(.) in Line 7 if timeout occurs, where $\oplus$ denotes element appending.

Algorithm 4: *Path_Enumerate*(s, t, k, G).

1: stack $P = \emptyset$, bool $visited[] \leftarrow \{false\}$
2: *DFS*$(s, t, k, P, visited, G)$
Procedure *DFS*$(s, t, k, P, visited, G)$
3: $visited[s] \leftarrow true$, $P.push(s)$
4: **if** $s = t$ **then** report P
5: **else if** $|P| \leq k$ **then**
6: **for each** $v \in N(s)$ with $visited[v] = false$
7: *DFS*$(v, t, k, P, visited, G)$
8: $P.pop()$, $visited[s] \leftarrow false$

Let us denote $B^{\pi}(u_i) = \{u_j \mid j < i \wedge (u_j, u_i) \in E_q\}$ as the backward neighbors of u_i in G_q given order π, and denote $N(v)$ as the neighbors of v in G. For each query edge $(u_j, u_i) \in E_q$ and each $v \in C(u_i)$, let us denote v's neighbors that can match u_j by $\mathcal{A}_{u_i}^{u_j}(v) = N(v) \cap C(u_j)$, which is saved in index $\mathcal{A}_q$ as the matched data edges. Then, Line 3 basically computes $C_S(u_i) = \cap_{u_j \in B^{\pi}(u_i)} \mathcal{A}_{u_i}^{u_j}(S[u_j])$ to ensure that all backward edges are matched, where $S[u_j]$ denotes the data vertex matched to u_j in S. Since set intersections are heavily used (over 80% of the total time), we adopt the hybrid set intersection method of [51], which leverages the SIMD parallelism supported by modern processors (AVX expands integer commands to 256 bits).

While Algorithm 3 is conceptually simple as a recursive function, it is inefficient since each time a recursion layer is added to the program stack (which takes time). To be efficient, we translate Algorithm 3 into an iterative-style implementation which is equivalent in logic but does not have any recursive function call.

Hop-Constrained s-t Path Enumeration [45]: Given two distinct vertices s and t in G, and a hop constraint k, this query outputs all paths from s to t with length at most k. This query can be used to detect money laundering and e-commerce merchant fraud [49]. Algorithm 4 shows the recursive depth-first path enumeration algorithm which tracks the current path by P, where Line 5 prevents P from growing beyond length k, and Line 4 outputs a path P whenever t is reached and stops further growing from P. We can also increment a path counter in Line 4 to count the number of s-t paths.

To parallelize it in G-thinkerQ, <QueryT> keeps $q = (s, t, k)$ as well as the path counter. We actually maintain an array of counters, one for each comper to increment to avoid counter locking overhead, and the total path count can be reported by summing these counters in UDF *postprocess*(q).

Each task runs *DFS*$(v, t, k, P, visited, G)$ so <ContextT> keeps $(v, P, visited)$ needed for path enumeration. UDF *compute*(.) runs Algorithm 4 but creates new tasks rather than recursively calling *DFS*(.) in Line 7 if timeout occurs.

UDF *is_bigTask*(.) checks for a task $(v, P, visited)$ if the outdegree of $v > \tau_{deg}$, where τ_{deg} is a user-defined threshold.

Besides this basic version, we also implemented the hot point indexing (HPI) method [45], which indexes all vertices with degree $>$ a threshold τ_{hot} (called hot points) and their connectivity in a graph G_{hot}, where each edge (v_i, v_j) with weight ℓ corresponds to a length-ℓ path from hot point (a.k.a. **HP**) v_i to HP v_j without any other HP in the middle. When we run DFS on G, we stop expansion at HPs to avoid large search fanout, and use G_{hot} to complete the paths with HPs.

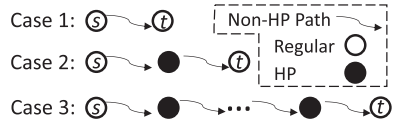


Fig. 7. Illustration of path cases when s and t are not hot.

Given a query (s, t, k), assuming that neither s nor t is an HP, then there are 3 cases for an s-t path P as illustrated in Fig. 7. (1) P has no HP, in which case k-hop DFS from s is sufficient. (2) P has one HP, so we conduct k-hop DFS from s to obtain a left table of paths ending at HPs, denoted by $\mathcal{L}$, and conduct k-hop reverse DFS from t over the reverse graph of G (i.e., along the in-neighbors) to obtain a right path-table $\mathcal{R}$ similarly. We then join those paths of $\mathcal{L}$ and $\mathcal{R}$ that share the same ending points to obtain Case 2 s-t paths with length $\leq k$. (3) P has more than one HP. Let a left (resp. right) path be $s \rightsquigarrow v_i$ (resp. $v_j \rightsquigarrow t$), then we basically obtain P by joining 3 path tables $s \rightsquigarrow v_i$, $v_i \rightsquigarrow v_j$ and $v_j \rightsquigarrow t$, where $v_i \rightsquigarrow v_j$ is obtained by DFS from v_i on G_{hot}.

Correspondingly, our G-thinkerQ program has 4 phases: (1) k-hop DFS from s to output Case-1 paths and compute $\mathcal{L}$; (2) k-hop reverse DFS from t to compute $\mathcal{R}$; (3) binary joining $\mathcal{L}$ and $\mathcal{R}$ to get Case-2 paths; (4) three-way joining $\mathcal{L}$, $\{v_i \rightsquigarrow v_j\}$, $\mathcal{R}$ to get Case-3 paths. Note that for cases when s or t is an HP, we basically skip Phase (1) or (2), respectively, while if both s and t are HPs, we only conduct Phase (4).

We keep the HP case type in <QueryT > which is properly initialized in *to_query*(.). We also keep the phase number of each query q in < QueryT>, which is properly advanced in *postprocess*(q). Both HP case type and phase number are used in UDFs to properly create and compute tasks by condition branching. We also keep $\mathcal{L}$ and $\mathcal{R}$ in <QueryT>, implemented as concurrent hash tables with an HP (e.g., v_i) as the key and the corresponding paths (i.e., paths ending at v_i) as value. Note that since DFS could time out and be completed by multiple comper threads, $\mathcal{L}$ and $\mathcal{R}$ have to be thread-safe for path insertion.

The root task for Phase (3) basically conducts hash join over $\mathcal{L}[v_i]$ and $\mathcal{R}[v_i]$ for each shared HP v_i. A coarse-grained implementation lets the root task create join-tasks $\langle \mathcal{L}[v_i], \mathcal{R}[v_i] \rangle$ for concurrent processing, but tasks may have imbalanced workloads. A fine-grained implementation lets the root task scan $(\mathcal{L}[v_i], \mathcal{R}[v_i])$ pairs to wrap them into join-tasks where each join-task $\{(L_j, R_j)\}$ has $cost = \sum_j (|L_j| \times |R_j|) \leq B$, with the batch size threshold tuned to be 500,000 by default. Specifically, if $|\mathcal{L}[v_i]| \leq |\mathcal{R}[v_i]|$, we create join mini-batches (L_j, R_j) with $L_j = \mathcal{L}[v_i]$ and R_j being a sub-table of paths in $\mathcal{R}[v_i]$; the case when $|\mathcal{L}[v_i]| > |\mathcal{R}[v_i]|$ is symmetric. Mini-batches are packed into batches with cost close to B in a streaming manner, and each batch creates a new join-task.

Phase (4) performs $|\mathcal{L}|$ single-source multi-destination DFS's over G_{hot}, each from an HP $v_i \in \mathcal{L}$ and whenever reaching an HP $v_j \in \mathcal{R}$, we conduct join over $(\mathcal{L}[v_i], P, \mathcal{R}[v_j])$. The DFS length restriction (for P) is properly determined; for example,

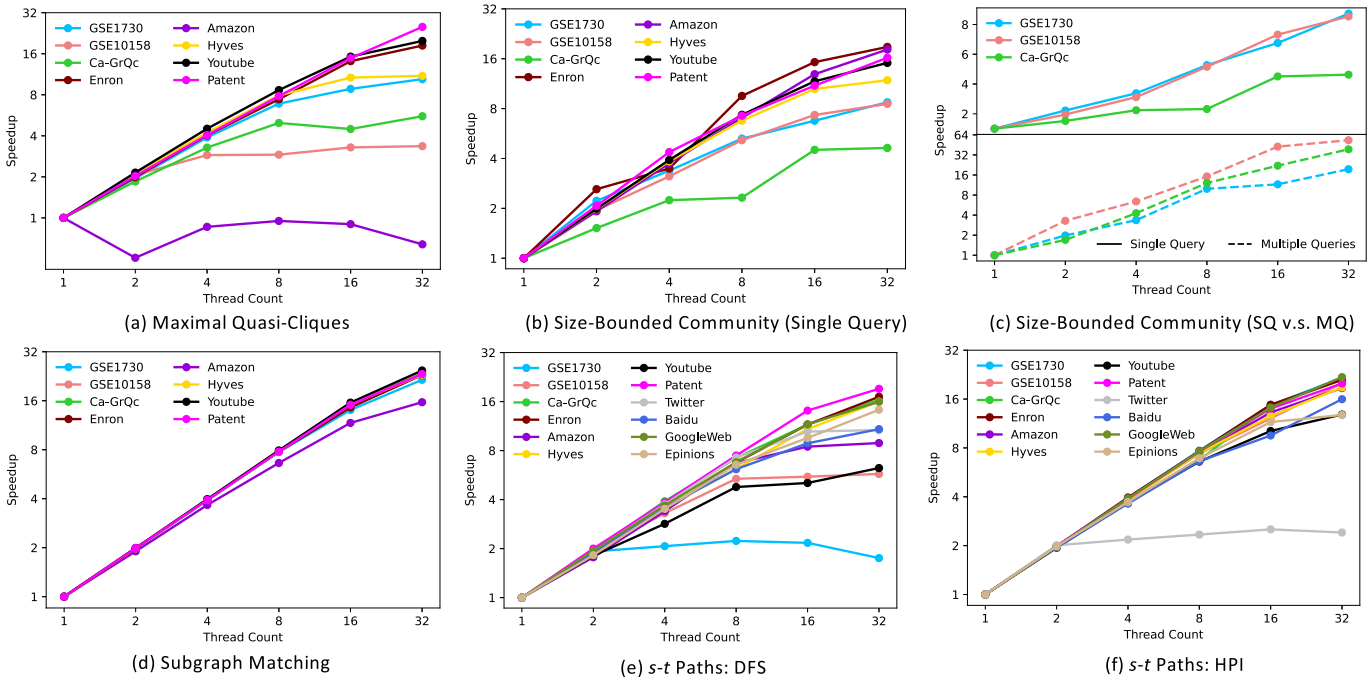


Fig. 8. Speedup ratio w.r.t. number of compers.

if neither s nor t is an HP, it is set as $(k - min\_path\_length(\mathcal{L}) - min\_path\_length(\mathcal{R}))$. The root task generates and adds DFS tasks each from an HP v_i, and each DFS task may time out to generate new DFS tasks. Similar to Phase (3), our fine-grained implementation splits the workloads of joins $(\mathcal{L}[v_i], P, \mathcal{R}[v_j])$ into individual join-tasks $\{(L_j, P, R_j)\}$ with $cost \leq B$. In contrast, our coarse-grained implementation lets DFS tasks perform joins immediately.

Finally, for the HPI implementation, *is_bigTask*(.) uses the default one that treats all tasks as regular, since DFS-tasks now have limited fanouts and join-tasks have bounded cost.

TABLE I
GRAPHS (8 UNDIRECTED, 4 DIRECTED)

Data	$\|V\|$	$\|E\|$	$\|E\|/\|V\|$	Max Outdeg	Max Indeg
CX_GSE1730	998	5,096	5.11	197	
CX_GSE10158	1,621	7,079	4.37	110	
Ca-GrQc	5,242	14,496	2.77	81	
Enron	36,692	183,831	5.01	1,383	
Amazon	334,863	925,872	2.76	549	
Hyves	1,402,673	2,777,419	1.98	31,883	
YouTube	1,134,890	2,297,624	2.63	28,754	
Patent	3,774,768	16,518,947	4.38	793	
Epinions	75,879	508,837	6.71	1,801	3,035
Twitter	465,017	834,797	1.80	500	199
GoogleWeb	875,713	5,105,039	5.83	456	6,326
Baidu	2,141,300	17,794,839	8.31	2,596	97,950

VII. EXPERIMENTS

This section studies how much G-thinkerQ speeds up our 4 subgraph queries introduced in Section VI, all of which are defined for undirected graphs while s-t path enumeration also supports directed graphs. Table I summarizes the 12 biological, communication and social networks we used for evaluation: *CX_GSE1730* [6], *CX_GSE10158* [5], *Ca-GrQc* [3], *Enron* [7], *Amazon* [1], *Hyves* [10], *YouTube* [13], *Patent* [11], *Epinions* [8], *Twitter* [12], *GoogleWeb* [9], and *Baidu* [2]. The first 8 are undirected and remaining 4 are directed. Experiments were run on a server with an IBM POWER8 CPU (32 cores, 3491MHz). All reported experiments were run with the tuned default system and algorithm parameters previously described, each repeated for 3 times with the average reported. Our source code has been released at https://github.com/lyuheng/TthinkerQ.

Maximal Quasi-Cliques: Recall that we load the input graph and shrink it into its k-core using initial parameters $(\gamma^{min}, \tau_{size}^{min})$. We use $(\gamma^{min}, \tau_{size}^{min})$ close to (γ, τ_{size}) tuned in [32] to ensure there exist results and the runtime is tractable.

To generate queries on a graph, we first run [32]'s program with $(\gamma, \tau_{size}) = (\gamma^{min}, \tau_{size}^{min})$ to mine maximal quasi-cliques, and let their vertices be V_{qcq}. We then generate random queries (γ, τ_{size}, Q) by only sampling vertices from V_{qcq} into Q to ensure there are results, and by uniformly sampling γ from $\gamma^{min} + [0, 0.05)$ and τ_{size} from $\tau_{size}^{min} + \{0, 1, 2\}$. For a query batch, we generate 20% queries for each length $|Q| = 1, 2, 3, 4, 5$. Table II(a) shows the $(\gamma^{min}, \tau_{size}^{min})$, runtime and speedup ratio when we process a batch of 100 queries with 32 compers. Fig. 8(a) further shows the speedup ratio when we vary the # of compers. We can see that good speedup is achieved for query batches that need time to run, such as *YouTube*, *Patent* and *Enron*, while speedup is limited for those that are fast to run (in fact a slowdown on *Amazon* that takes sub-second) since task scheduling and query tracking overheads increase with the # of compers.

Size-Bounded Community: Following [68], we select vertices with core number >5 as v_q, and try size bounds ℓ and h with a difference from 3 to 10. Many queries run for a long time

TABLE II
PARAMETERS & TIME AND SPEEDUP WITH 32 COMPERS (TIME UNIT: SECOND)

Data	(a) Maximal Quasi-Cliques					(b) Size-Bounded Community					(c) Subgraph Matching				
	γ^{min}	τ_{size}^{min}	Time	Speedup	#Results	v_q	ℓ	h	Time	Speedup	$\|V_Q\|$	$\|\Sigma\|$	#Queries	Time	Speedup
CX_GSE1730	30	0.85	3.45	10.42	270,698	25	33	39	13.68	8.73	8	5	20	9.14	21.52
CX_GSE10158	29	0.77	1.05	3.36	91,716	31	32	42	5.66	8.54	16	9	20	12.63	23.35
Ca-GrQc	10	0.8	1.28	5.57	1,190,563	492	38	44	3.13	4.63	8	4	20	79.01	23.48
Enron	23	0.84	9.87	18.37	627	27	18	21	60.08	18.85	16	24	5	14.64	23.02
Amazon	12	0.5	0.14	0.64	262	33247	31	34	268.45	18.19	16	4	80	27.99	15.67
Hyves	22	0.9	3.76	11.02	5,000	9	21	24	23.91	11.87	16	42	20	56.07	23.12
YouTube	18	0.9	135.43	19.87	2,957	15	15	18	54.24	15.1	24	70	5	119.12	24.54
Patent	20	0.9	205.52	25.19	11,076	30471	18	21	65.64	16.22	32	32	8	37.06	23.34

and [68] sets a time limit of two hours, after which it simply returns the current best H. We, in contrast, tried different v_q till some query can finish within a reasonable amount of time so that we can report speedup with the # of compers.

Table II(b) shows the 32-comper runtime and speedup ratio for the first randomly sampled query (v_q, ℓ, h) we found to be able to finish in reasonable time on each dataset. Fig. 8(b) further shows the speedup ratio when we vary the number of compers. We observe good speedup for queries that need time to run, such as *Enron*, *Amazon* and *Patent*. The speedup is limited for datasets *CX_GSE1730*, *CX_GSE10158* and *Ca-GrQc* that are fast to run, since Table II(b) reports single-query parallelism where parallel child-tasks are only generated when a parent task times out after running for $\tau_{time} = 0.3$ s, so compers are not saturated by tasks when computation begins.

The speedup ratio can be significantly improved if we have more queries as illustrated in Fig. 8(c), where we ran 10, 10, 32 queries on *CX_GSE1730*, *CX_GSE10158* and *Ca-GrQc*, respectively. This is because there are more cross-query parallelism, and the combined task workloads of these queries are able to keep all compers busy most of the time.

Subgraph Matching: The graphs in Table I have no labels, so we randomly assign each vertex a label from a label set Σ.

Following [29], [51], to ensure that a query graph $G_q = (V_q, E_q)$ can find matches in G, we generate G_q by a random walk starting from a random vertex until getting the specified # of vertices, and we extract the induced subgraph as G_q.

We tried different sizes $|V_q|$, $|\Sigma|$ and different # of queries in a batch, and we picked the combination such that the query batch can finish within reasonable time to report in Table II(c) regarding 32-comper runtime and speedup ratio. Fig. 8(d) further shows the speedup ratio when we vary the number of compers. We observe good speedup ratio on all datasets.

Recall from Section III that two graph-centric systems, Fractal [22] and Peregrine [31], can also answer subgraph matching queries efficiently. Fractal [22] treats each subgraph expansion as a basic unit for depth-first task scheduling, while Peregrine adopts a pattern-based model to extract the semantics of patterns to guide its exploration. We, therefore, compare G-thinkerQ with Fractal and Peregrine using subgraph matching.

Note that our other applications are more advanced than what can be formulated using the simple APIs of Fractal and Peregrine, while [61] and [22] have demonstrated that depth-first systems such as G-thinker and Fractal beats breadth-first systems such as Arabesque and RStream by a large margin. Moreover, [31] shows that Peregrine can further beat Fractal.

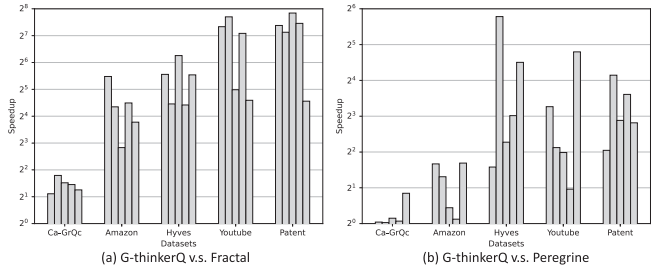


Fig. 9. Speedup ratio w.r.t. fractal and peregrine on 5 queries.

Among the 8 undirected graphs in Table I, the last four are significantly larger and all with a high maximum degree reflecting that there is at least a very dense region in each graph. We find that the speedup ratio of G-thinkerQ over Fractal and Peregrine is the most significant on such graphs. Due to page limit, we use these 4 large graphs to report the speedup ratio, while only include Ca-GrQc as a representative of the other graphs that are smaller (so the speedup ratio of G-thinker is not as significant).

Fig. 9(a) shows the runtime speedup of G-thinkerQ over Fractal for five random-walk-generated queries (represented by the five bars) on each of the five datasets, where the runtime of G-thinkerQ has been normalized as $2^0 = 1$. Here, we use $|V_Q| = 8$ since when running Fractal on query graphs with more vertices, Fractal either reports ArrayIndexOutOfBoundsException or gives 0 results, though Fractal gives correct results when there are ≤ 8 vertices. With this small $|V_Q|$, queries would be very fast to evaluate using the values of $|\Sigma|$ in Fig. 2(c), so we reduce $|\Sigma|$ to 3, 2, 8, 16, 8 on the 5 datasets, respectively, to properly increase the query workloads. As Fig. 9(a) shows, the speedup of G-thinkerQ becomes more significant as the graph size increases. G-thinkerQ can be two orders of magnitude faster than Fractal, since Fractal does not allow indexing so it enumerates subgraphs on the big input graph rather than a small auxiliary data structure $\mathcal{A}_q$, and it performs additional isomorphism checking. Moreover, Fractal has to match each query graph one by one, but G-thinkerQ can achieve further speedup if we run the 5 queries as a batch.

Similarly, Fig. 9(b) reports the runtime speedup of G-thinkerQ over Peregrine for five random-walk-generated queries (represented by the five bars) on each dataset, where the runtime of G-thinkerQ has been normalized as $2^0 = 1$. We observe similar speedup trend as in Fig. 9(a) but the speedup ratio is not as large since Peregrine is in general faster than Fractal [31].

TABLE III
TIME & SPEEDUP WITH 32 COMPERS

Data	k	τ_{hot}	#Queries	HPI (Fine-Grained)		DFS	
				Time	Speedup	Time	Speedup
CX_GSE1730	4	60	10	24.51	21.69	0.3	1.74
CX_GSE10158	4	40	100	7.26	20.03	0.69	5.74
Ca-GrQc	5	40	100	50.25	21.02	3.37	16.41
Enron	4	500	10	90.18	21.04	2.51	17.12
Amazon	6	100	100	14.26	18.7	1.37	8.9
Hyves	4	800	100	17.96	18.67	7.02	16
YouTube	3	3000	100	3.8	12.85	1.17	6.25
Patent	5	400	100	12.57	19.97	8.48	19.17
Epinions	4	250	10	58.97	12.77	1.01	14.25
Twitter	8	100	1000	5.68	2.41	8.99	10.69
Baidu	4	600	10	32.03	15.97	1.25	10.83
GoogleWeb	6	100	100	102.82	21.84	4.62	15.96

Hop-Constrained s-t Path Enumeration: Following [43], [49], we generate each query (s, t, k) where source s is randomly sampled from V and target vertex t is a vertex reached by a k-step random walk from s, so that we make sure there exist result paths. We select the number of queries in a batch, k and τ_{hot} on each dataset so that the query batch can finish in reasonable time for both HPI (fine-grained) and DFS.

Table III reports the 32-comper runtime and speedup ratio. Fig. 8(e) and (f) further show the speedup ratio of DFS and HPI, respectively, when we vary the number of compers.

We see that HPI has good speedup on all datasets except for the very sparse graph *Twitter*, since the path join operations in Phases (3) and (4) usually constitute the majority of computation workloads according to our observation, which can be effectively decomposed into join-tasks of cost $\leq B = 500,000$ for parallel processing on denser graphs. While the speedup of DFS is lower than HPI as shown in Table III, the querying time is actually much shorter on all graphs except for *Twitter* which is very sparse. This might be surprising since HPI is an indexing-based method, but our program profiling shows that path joining with the help of G_{hot} is oftentimes much more expensive than direct DFS unless a graph is very sparse. In fact, HPI was originally proposed for querying the sparse Alibaba transactional network [45], and this weakness of HPI was also observed by later works [43], [49] which propose new techniques and better join plans to overcome this weakness. We use HPI as an application to illustrate multi-phase algorithm programming in G-thinkerQ, and we plan to implement more recent solutions [43], [49] as future work.

Recall that we also proposed a coarse-grained HPI baseline that joins $\langle \mathcal{L}[v_i], \mathcal{R}[v_i] \rangle$ and $(\mathcal{L}[v_i], P, \mathcal{R}[v_j])$ for HPs v_i, v_j directly rather than decomposing large ones into smaller join-tasks of cost $\leq B = 500,000$. We find that coarse-grained HPI always suffers from the straggler problem since some $\mathcal{L}[v_i]$ and/or $\mathcal{R}[v_j]$ can be very large and the joins involving them can cause the handling compers to become stragglers. To illustrate, Fig. 10 compares the scalability on *Baidu* and *CX_GSE1730* where we see that coarse-grained HPI is much more expensive (note that runtime is in log scale) and does not reduce much after the # of compers reaches 16.

Memory Consumption: One advantage of G-thinkerQ is that it inherits the task capsule design of [32] that keeps the memory usage by tasks bounded. In fact, memory is never an issue in all our experiments, so we have omitted them due to the space limitation. As an illustration, Table IV reports the total memory usage, and memory usage by tasks only (by deducting the memory usage after input graph is loaded) when we process 100 quasi-clique queries on the large *Patent* graph, where we can see that most memory is occupied for storing the input graph, and tasks only use 8.36× memory when we increase # of compers from 1 to 32.

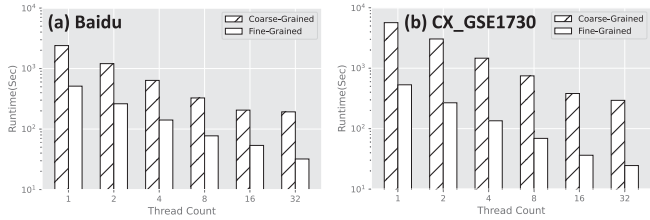


Fig. 10. Coarse- v.s. fine-grained HPI methods.

TABLE IV
MEMORY USAGE FOR MAXIMAL QUASI-CLIQUES ON *PATENT*

Thread #	Runtime (s)	Memory (MB)	Mem w/o G	Task Mem Ratio
1	5177.52	2353	61	1
2	2553.86	2381	93	1.52
4	1277.54	2424	110	1.8
8	662.49	2501	184	3.02
16	348.36	2675	341	5.59
32	205.52	3029	510	8.36

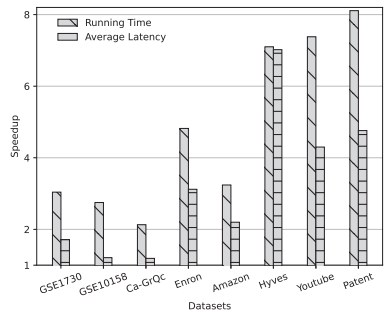


Fig. 11. Time and latency speedup of g-thinkerQ v.s. G-thinker.

Comparison with G-thinker: Compared with answering queries with G-thinker [61] one by one, G-thinkerQ supports inter-query parallelism so that tasks of a later query can be processed if tasks of earlier queries cannot saturate the available CPU cores. We demonstrate how effective G-thinkerQ's inter-query parallelism is, by running a batch of 10 quasi-clique queries with both G-thinkerQ and G-thinker on each dataset. Parameters (γ, τ_{size}) are set as (30, 0.85), (29, 0.77), (10, 0.8), (23, 0.84), (12, 0.5), (23, 0.91), (18, 0.9) and (20, 0.9), respectively, for the 8 undirected graphs in Table I, but the initial vertex of each query is randomly selected. We consider two metrics: running time (to finish the query batch) and average latency, where the latency is the time a user waits to obtain the result of a query.

Fig. 11 show the comparison results where the running time and average latency of G-thinkerQ has been normalized as 1, and graph loading time is not counted. We can see that G-thinkerQ is 2.1×–8.1× faster than G-thinker thanks to its inter-query

TABLE V
SPEEDUP OF G-THINKERQ V.S. QUEGEL

	Amazon					Patent				
	Q1	Q2	Q3	Q4	Q5	Q1	Q2	Q3	Q4	Q5
Quegel	35.4	31.3	33.9	25.9	24.4	139,117	131,177	78,490	125,596	124,908
Ours	0.05	0.04	0.03	0.03	0.02	1.9	1.6	3.0	1.9	1.5

parallelism. In terms of average latency, it is also $1.2\times$–$7.0\times$ shorter for G-thinkerQ thanks to our effective task scheduling and query progress tracking designs.

Comparison with Quegel: Quegel [58] is a vertex-centric system to answer iterative graph queries online. Since our four studied graph queries operate on subgraphs involving vertices more than one hop away, they are not suitable for implementation as a vertex-centric program. Therefore, in order to compare G-thinkerQ with Quegel, we use the application of maximal-clique enumeration to find all maximal cliques that contain an initial query vertex, for which a vertex-centric algorithm is available (see Algorithm 3 of [15]) and we implement it for Quegel. In G-thinkerQ, we implement the Bron-Kerbosch algorithm [16] which is recursive and follows the set-enumeration framework illustrated in Fig. 1.

In the vertex-centric algorithm of [15], when extending a set $S = \{v_1, v_2, \ldots, v_{|S|}\}$ with a candidate $v' \in ext(S)$, S needs to be sent from $v_{|S|}$ to v' to process the extended set $S' = \{v_1, \ldots, v_{|S|}, v'\}$. Even though we run Quegel on the same server with 32 threads (rather than in a distributed cluster to save communication), a lot of intermediate sets are materialized creating excessive computing and memory overheads.

We generate 5 queries with non-trivial workload for each graph G. Specifically, we first use G-thinker [61] to find the maximum clique Q_{max} in G, and then generate each query by selecting a random vertex in Q_{max} as the initial query vertex. Table V reports the query running time of both systems on Amazon ($|Q_{max}| = 7$) and Patent ($|Q_{max}| = 11$), while we omit the other graphs (whose $|Q_{max}|$ is much larger) since Quegel runs out of memory when processing queries on them. As Table V shows, G-thinkerQ is 3 to 5 orders of magnitude faster than Quegel.

VIII. CONCLUSION

We proposed G-thinkerQ, a general subgraph querying system with a unified task-based programming model. G-thinkerQ allows the input graph to be loaded (and indexed) once and reused for subsequent user queries, and it manages the submitted queries to ensure (1) high task concurrency, (2) memory boundedness, (3) fairness, and (4) timeliness of returning results. Specifically, G-thinkerQ provides an intuitive TLAT programming interface that allows users to write parallel algorithms for various subgraph queries by directly adapting from their serial algorithm counterparts with advanced pruning techniques, and it uses a novel task-capsule list to ensure fairness of query evaluation order while supporting memory-bounded concurrent evaluation of multiple queries. A lightweight lineage-based mechanism is also designed to keep track of when the last task of a query is completed, so as to return query results in time and to release resources. Recent serial algorithms for 4 advanced subgraph queries are parallelized on G-thinkerQ with excellent speedup.

REFERENCES

[1] Amazon. [Online]. Available: https://snap.stanford.edu/data/com-Amazon.html
[2] Baidu. [Online]. Available: http://konect.cc/networks/zhishi-baidu-internallink/
[3] Ca-GrQc. [Online]. Available: https://snap.stanford.edu/data/ca-GrQc.html
[4] CCC Great Innovative Idea on T-thinker. [Online]. Available: https://cra.org/ccc/great-innovative-ideas/t-thinker-a-task-centric-framework-to-revolutionize-big-data-systems-research/
[5] CX_GSE10158. [Online]. Available: https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE10158
[6] CX_GSE1730. [Online]. Available: https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE1730
[7] Enron. [Online]. Available: https://snap.stanford.edu/data/email-Enron.html
[8] Epinions. [Online]. Available: http://konect.cc/networks/soc-Epinions1/
[9] GoogleWeb. [Online]. Available: https://snap.stanford.edu/data/web-Google.html
[10] Hyves. [Online]. Available: http://konect.cc/networks/hyves/
[11] Patent. [Online]. Available: https://graphchallenge.s3.amazonaws.com/synthetic/gc6/U1a.tsv
[12] Twitter. [Online]. Available: http://konect.cc/networks/munmun_twitter_social/
[13] YouTube. [Online]. Available: https://snap.stanford.edu/data/cit-Patents.html
[14] K. Ammar and M. T. Özsu, "Experimental analysis of distributed graph systems," in *Proc. VLDB Endowment*, vol. 11, no. 10, pp. 1151–1164, 2018.
[15] A. Brighen, H. Slimani, A. Rezgui, and H. Kheddouci, "Listing all maximal cliques in large graphs on vertex-centric model," *J. Supercomput.*, vol. 75, no. 8, pp. 4918–4946, 2019.
[16] C. Bron and J. Kerbosch, "Finding all cliques of an undirected graph (algorithm 457)," *Commun. ACM*, vol. 16, no. 9, pp. 575–576, 1973.
[17] H. Chen, M. Liu, Y. Zhao, X. Yan, D. Yan, and J. Cheng, "G-miner: An efficient task-oriented graph mining system," in *Proc. 13th Eur. Conf.*, 2018, pp. 32:1–32:12.
[18] J. Cheng et al., "Mining order-preserving submatrices under data uncertainty: A possible-world approach and efficient approximation methods," *ACM Trans. Database Syst.*, vol. 47, pp. 1–57, 2022.
[19] Q. Cheng et al., "Efficient enumeration of large maximal k-plexes," in *Proc. Annu. Int. Conf. Extending Database Technol.*, 2025, pp. 53–65.
[20] Y. H. Chou, E. T. Wang, and A. L. P. Chen, "Finding maximal quasi-cliques containing a target vertex in a graph," in *Proc. 4th Int. Conf. Data Manage. Technol. Appl.*, 2015, pp. 5–15.
[21] P. Conde-Cespedes, B. Ngonmang, and E. Viennet, "An efficient method for mining the maximal α-quasi-clique-community of a given node in complex networks," *Social Netw. Anal. Mining*, vol. 8, no. 1, 2018, Art. no. 20.
[22] V. V. dos Santos Dias, C. H. C. Teixeira, D. Guedes, W. Meira Jr., and S. Parthasarathy, "Fractal: A general-purpose graph pattern mining system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2019, pp. 1357–1374.
[23] W. Fan, F. Geerts, and F. Neven, "Making queries tractable on big data with preprocessing," in *Proc. VLDB Endowment*, vol. 6, no. 9, pp. 685–696, 2013.
[24] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Symp. Operating Syst. Des. Implementation*, Hollywood, CA, USA, 2012, pp. 17–30, 2012.

[25] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 14 Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[26] G. Guo, D. Yan, M. T. Özsu, Z. Jiang, and J. Khalil, "Scalable mining of maximal quasi-cliques: An algorithm-system codesign approach," in *Proc. VLDB Endow.*, vol. 14, no. 4, pp. 573–585, 2020.

[27] G. Guo et al., "Maximal directed quasi-clique mining," in *Proc. Int. Conf. Data Eng.*, 2022, pp. 1900–1913.

[28] M. Han, K. Daudjee, K. Ammar, M. T. Özsu, X. Wang, and T. Jin, "An experimental comparison of pregel-like graph processing systems," in *Proc. VLDB Endow.*, vol. 7, no. 12, pp. 1047–1058, 2014.

[29] M. Han, H. Kim, G. Gu, K. Park, and W. Han, "Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together," in *Proc. ACM SIGMOD Int. Conf. Manage. Data Conf.*, 2019, pp. 1429–1446.

[30] H. He and A. K. Singh, "Graphs-at-a-time: Query language and access methods for graph databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data Conf.*, 2008, pp. 405–418.

[31] K. Jamshidi, R. Mahadasa, and K. Vora, "Peregrine: A pattern-aware graph mining system," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 13:1–13:16.

[32] J. Khalil, D. Yan, G. Guo, and L. Yuan, "Parallel mining of large maximal quasi-cliques," *VLDB J.*, vol. 31, pp. 649–674, 2021.

[33] A. Kyrola, G. E. Blelloch, and C. Guestrin, "Graphchi: Large-scale graph computation on just a PC," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 31–46.

[34] P. Lee and L. V. S. Lakshmanan, "Query-driven maximum quasi-clique search," in *Proc. SIAM Int. Conf. Data Mining*, 2016, pp. 522–530.

[35] G. Liu and L. Wong, "Effective pruning techniques for mining quasi-cliques," in *Proc. Joint Eur. Conf. Mach. Learn. Knowl. Discov. Databases*, Springer, 2008, pp. 33–49.

[36] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," in *Proc. VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[37] Y. Lu, J. Cheng, D. Yan, and H. Wu, "Large-scale distributed graph computing systems: An experimental evaluation," in *Proc. VLDB Endow.*, vol. 8, no. 3, pp. 281–292, 2014.

[38] G. Malewicz et al., "Pregel: A system for large-scale graph processing," in *Proc. SIGMOD Conf.*, 2010, pp. 135–146.

[39] F. D. Malliaros, A. N. Papadopoulos, and M. Vazirgiannis, "Core decomposition in graphs: Concepts, algorithms and applications," in *Proc. Annu. Int. Conf. Extending Database Technol.*, 2016, pp. 720–721.

[40] R. R. McCune, T. Weninger, and G. Madey, "Thinking like a vertex: A survey of vertex-centric frameworks for large-scale distributed graph processing," *ACM Comput. Surv.*, vol. 48, no. 2, pp. 25:1–25:39, 2015.

[41] F. McSherry, M. Isard, and D. G. Murray, "Scalability! but at what cost?," in *Proc. 15th Workshop Hot Top. Operating Syst.*, 2015, Art. no. 14.

[42] J. Pei, D. Jiang, and A. Zhang, "On mining cross-graph quasi-cliques," in *Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2005, pp. 228–238.

[43] Y. Peng, Y. Zhang, X. Lin, W. Zhang, L. Qin, and J. Zhou, "Hop-constrained s-t simple path enumeration: Towards bridging theory and practice," in *Proc. VLDB Endow.*, vol. 13, no. 4, pp. 463–476, 2019.

[44] L. Qin, J. X. Yu, L. Chang, H. Cheng, C. Zhang, and X. Lin, "Scalable big graph processing in MapReduce," in *Proc. Int. Conf. Manage. Data*, Snowbird, UT, USA, 2014, pp. 827–838.

[45] X. Qiu et al., "Real-time constrained cycle detection in large dynamic graphs," in *Proc. VLDB Endow.*, vol. 11, no. 12, pp. 1876–1888, 2018.

[46] W. Qu, D. Yan, G. Guo, X. Wang, L. Zou, and Y. Zhou, "Parallel mining of frequent subtree patterns," in *Proc. Softw. Found. Data Interoperability Large Scale Graph Data Analytics*, Springer, 2020, pp. 18–32.

[47] A. Quamar, A. Deshpande, and J. Lin, "NScale: Neighborhood-centric large-scale graph analytics in the cloud," *VLDB J.*, vol. 25, pp. 125–150, 2014.

[48] A. Roy, I. Mihailovic, and W. Zwaenepoel, "X-stream: Edge-centric graph processing using streaming partitions," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 472–488.

[49] S. Sun, Y. Chen, B. He, and B. Hooi, "Pathenum: Towards real-time hop-constrained s-t path enumeration," in *Proc. ACM SIGMOD Int. Conf. Manage. Data Conf.*, 2021, pp. 1758–1770.

[50] S. Sun and Q. Luo, "Parallelizing recursive backtracking based subgraph matching on a single machine," in *Proc. IEEE 24th Int. Conf. Parallel Distrib. Syst.*, 2018, pp. 42–50.

[51] S. Sun and Q. Luo, "In-memory subgraph matching: An in-depth study," in *Proc. ACM SIGMOD Int. Conf. Manage. Data Conf.*, 2020, pp. 1083–1098.

[52] C. H. C. Teixeira, A. J. Fonseca, M. Serafini, G. Siganos, M. J. Zaki, and A. Aboulnaga, "Arabesque: A system for distributed graph mining," in *Proc. 25th Symp. Operating Syst. Princ.*, 2015, pp. 425–440.

[53] J. R. Ullmann, "An algorithm for subgraph isomorphism," *J. ACM*, vol. 23, no. 1, pp. 31–42, 1976.

[54] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu, "RStream: Marrying relational algebra with streaming for efficient graph mining on A single machine," in *Proc. 14 Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 763–782.

[55] D. Yan, Y. Bu, Y. Tian, and A. Deshpande, "Big graph analytics platforms," *Found. Trends Databases*, vol. 7, no. 1/2, pp. 1–195, 2017.

[56] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Blogel: A block-centric framework for distributed computation on real-world graphs," in *Proc. VLDB Endow.*, vol. 7, no. 14, pp. 1981–1992, 2014.

[57] D. Yan, J. Cheng, Y. Lu, and W. Ng, "Effective techniques for message reduction and load balancing in distributed graph computation," in *Proc. 24th Int. Conf. World Wide Web*, Florence, Italy, May 18–22, 2015, pp. 1307–1317.

[58] D. Yan et al., "A general-purpose query-centric framework for querying big graphs," in *Proc. VLDB Endow.*, vol. 9, no. 7, pp. 564–575, 2016.

[59] D. Yan, J. Cheng, K. Xing, Y. Lu, W. Ng, and Y. Bu, "Pregel algorithms for graph connectivity problems with performance guarantees," in *Proc. VLDB Endowment*, vol. 7, no. 14, pp. 1821–1832, 2014.

[60] D. Yan, M. M. R. Chowdhury, G. Guo, J. Khalil, Z. Jiang, and S. K. Prasad, "Distributed task-based training of tree models," in *Proc. IEEE Int. Conf. Data Eng.*, 2022, pp. 2237–2249.

[61] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, W. Ku, and J. C. S. Lui, "G-thinker: A distributed framework for mining subgraphs in a big graph," in *Proc. Int. Conf. Data Eng.*, 2020, pp. 1369–1380.

[62] D. Yan, G. Guo, M. M. R. Chowdhury, M. T. Özsu, J. C. S. Lui, and W. Tan, "T-thinker: A task-centric distributed framework for compute-intensive divide-and-conquer algorithms," in *Proc. 24th Symp. Princ. Pract. Parallel Program.*, 2019, pp. 411–412.

[63] D. Yan, G. Guo, J. Khalil, M. T. Özsu, W.-S. Ku, and J. Lui, "G-thinker: A general distributed framework for finding qualified subgraphs in a big graph with load balancing," *VLDB J.*, vol. 31, pp. 287–320, 2021.

[64] D. Yan et al., "GraphD: Distributed vertex-centric graph processing beyond the memory limit," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 1, pp. 99–114, Jan. 2018.

[65] D. Yan, W. Qu, G. Guo, and X. Wang, "PrefixFPM: A parallel framework for general-purpose frequent pattern mining," in *Proc. Int. Conf. Data Eng.*, 2020, pp. 1938–1941.

[66] D. Yan, W. Qu, G. Guo, X. Wang, and Y. Zhou, "PrefixFPM: A parallel framework for general-purpose mining of frequent and closed patterns," *VLDB J.*, vol. 31, pp. 253–286, 2021.

[67] D. Yan, Y. Tian, and J. Cheng, "Systems for big graph analytics, "2017. [Online]. Available: https://dblp.uni-trier.de/rec/series/sbcs/YanTC17.html?view=bibtex

[68] K. Yao and L. Chang, "Efficient size-bounded community search over large networks," in *Proc. VLDB Endow.*, vol. 14, no. 8, pp. 1441–1453, 2021.

[69] L. Yuan, D. Yan, J. Han, A. Ahmad, Y. Zhou, and Z. Jiang, "Faster depth-first subgraph matching on GPUs," in *Proc. IEEE 40th Int. Conf. Data Eng.*, 2024, pp. 3151–3163.

[70] L. Yuan et al., "T-FSM: A task-based system for massively parallel frequent subgraph pattern mining from a big graph," in *Proc. ACM Manag. Data*, vol. 1, no. 1, pp. 74:1–74:26, 2023.

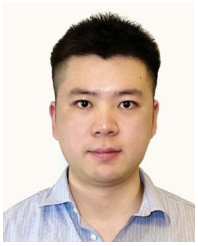
[71] Z. Yue, D. Yan, G. Guo, and J. Y. Chen, "Biological network mining," in *Proc. Model. Transcriptional Regulation*, Springer, 2021, pp. 139–151.

[72] Q. Zhang, H. Chen, D. Yan, J. Cheng, B. T. Loo, and P. V. Bangalore, "Architectural implications on the performance and cost of graph analytics systems," in *Proc. Symp. Cloud Comput.*, 2017, pp. 40–51.

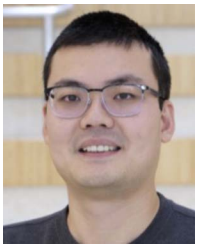
[73] Q. Zhang, D. Yan, and J. Cheng, "Quegel: A general-purpose system for querying big graphs," in *Proc. ACM SIGMOD Int. Conf. Manage. Data Conf.*, 2016, pp. 2189–2192.

[74] Y. Zhou, J. Xu, Z. Guo, M. Xiao, and Y. Jin, "Enumerating maximal *K*-plexes with worst-case time guarantee," in *Proc. Conf. Assoc. Adv. Artif. Intell.*, 2020, pp. 2442–2449.

Lyuheng Yuan received the bachelor's degree from the Hefei University of Technology, China and the master's degree from the University of Pennsylvania. He is currently working toward the PhD degree in the Department of Computer Science with Indiana University Bloomington. He was a winner of ICDE 2024 Best Demonstration Runner-Up Award, and a Blazer Fellow at the University of Alabama at Birmingham. His research interests lie in parallel and distributed systems, and graph mining.



Guimu Guo received the master's degree from Tongji University, Shanghai and the PhD degree in computer science from the University of Alabama at Birmingham (UAB). He is an assistant professor of computer science with Rowan University. He was a winner of Alabama EPSCoR Graduate Research Scholar Program (GRSP) Award in Rounds 15 and 16, UAB College of Arts and Sciences Outstanding PhD Student Award, and NSF CRII Award. His current interests include parallel and distributed computing, graph mining and GPGPU.



Da Yan (Senior Member, IEEE) received the BS degree in computer science from Fudan Universityi, in 2009 and the PhD degree in computer science from the Hong Kong University of Science and Technology, in 2014. He is an associate professor of computer science with Indiana University Bloomington. He was the sole winner of Hong Kong 2015 Young Scientist Award in Physical/Mathematical Science, a DOE Early Career Research Program (ECRP) awardee in 2023, and a senior member of ACM and IEEE. His research interests include database systems, data mining and machine learning.

Saugat Adhikari received the BE degree in computer engineering from Kathmandu University, Nepal, in 2018. He is currently working toward the PhD degree in the Department of Computer Science, Indiana University Bloomington. He was a winner of the Radiance Technologies Innovation Bowl 2022-2023 ($25,000), and a Blazer Fellow with the University of Alabama at Birmingham. His research interests include spatial data mining, graph mining, and deep learning.

Jalal Khalil received the PhD degree in computer science from the University of Alabama at Birmingham (UAB), in 2023. He is an assistant professor of Software Engineering with St. Cloud State University. Prior to joining UAB, in 2019, he worked in the industry for eight years as a software and cloud engineer. He was a winner of ICDE 2024 Best Demonstration Runner-Up Award. His research interests include graph mining, geospatial data science, and traffic simulation.

Cheng Long (Senior Member, IEEE) received the BEng degree from the South China University of Technology, China, in 2010 and the PhD degree from the Hong Kong University of Science and Technology, Hong Kong, in 2015. He is currently an associate professor with the College of Computing and Data Science, Nanyang Technological University. From 2016 to 2018, he worked as a lecturer with Queen's University Belfast, U.K. His research interests are broadly in data management and data mining.

Lei Zou is a professor in Wangxuan Institute of Computer Technology of Peking University. He is also a faculty member in National Engineering Laboratory for Big Data Analysis and Applications (Peking University) and the Center for Data Science of Peking University. His research interests include graph databases and semantic data management.