HUGE: An Efficient and Scalable Subgraph Enumeration System

Zhengyi Yang
The University of New South Wales
zyang@cse.unsw.edu.au

Longbin Lai
Alibaba Group
longbin.lailb@alibaba-inc.com

Xuemin Lin
The University of New South Wales
lxue@cse.unsw.edu.au

Kongzhang Hao
The University of New South Wales
khao@cse.unsw.edu.au

Wenjie Zhang
The University of New South Wales
zhangw@cse.unsw.edu.au

Abstract

Subgraph enumeration is a fundamental problem in graph analytics, which aims to find all instances of a given query graph on a large data graph. In this paper, we propose a system called HUGE to efficiently process subgraph enumeration at scale in the distributed context. HUGE features 1) an optimiser to compute an advanced execution plan without the constraints of existing works; 2) a hybrid communication layer that supports both pushing and pulling communication; 3) a novel two-stage execution mode with a lock-free and zero-copy cache design, 4) a BFS/DFS-adaptive scheduler to bound memory consumption, and 5) two-layer intra- and inter-machine load balancing. HUGE is generic such that all existing distributed subgraph enumeration algorithms can be plugged in to enjoy automatic speed up and bounded-memory execution.

CCS Concepts

• **Information systems → DBMS engine architectures**; **Join algorithms**; **Graph-based database models**.

Keywords

subgraph enumeration; distributed graph processing; join processing; cache; dynamic scheduling; load balancing

1 Introduction

Subgraph enumeration is a fundamental problem in graph analytics that aims to find all subgraph instances of a data graph that are isomorphic to a query graph. Subgraph enumeration is associated with a lot of real-world applications [4, 25, 26, 35, 41, 49, 52, 61, 82, 84]. Additionally, it is a key operation when querying graph databases such as Neo4j [55], and also plays a critical role in graph pattern mining (GPM) systems [23, 38, 50, 80].

Table 1: Results of a square query over the LJ graph in a local 10-machine cluster, including total time (T), computation time (T_R), communication time ($T_C = T - T_R$), total data transferred (C), and peak memory usage (M) among all machines.

Comm. Mode	Work	T(s)	T_R(s)	T_C(s)	C(GB)	M(GB)
Pushing	SEED	1536.6	343.2	1193.4	537.2	42.3
	BiGJoin	195.9	122.1	73.8	534.5	14.3
Pulling	BENU	4091.7	3763.2	328.5	25.3	1.3
	RADS	2643.8	2478.7	165.1	452.7	19.2
Hybrid	HUGE	52.3	51.5	0.8	4.6	2.2

With the growth in graph size nowadays [66] and the NP-hardness [27] of subgraph enumeration, researchers have devoted enormous efforts into developing efficient and scalable algorithms in the distributed context [5, 45, 46, 59, 62, 65, 70, 79, 83].

Motivations. The efficiency and scalability of distributed subgraph enumeration are jointly determined by three perspectives: *computation*, *communication* and *memory management* [47, 65]. However, existing works [5, 46, 65, 83] [1] fail to demonstrate satisfactory performance for all three perspectives. To verify, we conduct an initial experiment by running the square query (□) over the popular benchmark graph LJ [47]. The results[2] are shown in Table 1.

SEED [46] and BiGJoin [5] are join-based algorithms that adopt the *pushing* communication mode, which communicates by sending data from the host machine to remote destinations. In general, SEED processes subgraph enumeration via a series of binary joins, each joining the matches of two sub-queries via the conventional hash-join algorithm. BiGJoin [5] follows the worst-case optimal (wco) join algorithm [57], which extends the query graph one vertex at a time by intersecting the neighbours of all its connected vertices. Both algorithms are scheduled in a breadth-first-search (BFS) order [62] in order to fully utilize the parallel computation, which in turn requires materializing and transferring (via pushing) enormous intermediate results. Such design choices cause high tension on both communication and memory usage, as shown in Table 1.

While noticing the enormous cost from pushing communication, BENU [83] and RADS [65] follow a *pulling* design. BENU has been developed to pull (and cache) graph data from a distributed key-value store (e.g. Cassandra [14]). On each machine, it embarrassingly parallelises a sequential depth-first-search (DFS)-based program [81] to compute the matches. Such pulling design substantially reduces BENU's communication volume, which, however, does not shorten its communication time accordingly. The main culprit is the large overhead of pulling (and caching) data from the external key-value store. Additionally, while the use of DFS

---

[1]We mainly discuss four representative works here, while the others are in Section 8.
[2]The results may differ from the original report of BENU and RADS because we use better implementations of the join-based algorithms [47] (Section 7).

strategy results in low memory consumption, it can suffer from low CPU utilisation [64]. The above shortages jointly reduce the computing efficiency of BENU. To support a more efficient pulling design, RADS has developed its own compute engine without external dependency. Observe that the matches of a star (a tree of depth) rooted on a vertex can be enumerated from its neighbours [45]. Instead of transferring the intermediate results, the join that involves a star can be computed locally after pulling to the host machine the remote vertices with their neighbours. However, to facilitate such a pulling design, RADS is coupled with a StarJoin-like [79] execution plan that has already been shown to be sub-optimal [5, 46], which leads to poor performance of RADS in all perspectives.

Challenges. We distil three impact factors that jointly affect the three perspectives of distributed subgraph enumeration - *execution plan*, *communication mode*, and *scheduling strategy*.

Execution plan. Existing works derive their "optimal" execution plans, while none can guarantee the best performance by all means, as evidenced by [47] and the results in Table 1. The main reason is that these works achieve optimality in a rather *specific* context subject to the join algorithm and communication mode. For example, SEED is optimal among the hash-join-based algorithms [45, 46, 79], while BiGJoin's optimality follows the wco-join algorithm. The optimal plan of RADS is computed specifically for its pulling-based design. We argue that an optimal execution plan should lie in a more generic context without the constraints of existing works, making it challenging to define and compute.

Communication mode. While pulling mode can potentially reduce communication volume, it is non-trivial to make it eventually improve overall performance. Regarding design choice, it is not an option to blindly embrace the pulling design, as RADS has practised, without considering its impact on the execution plan. Regarding implementation, it is infeasible to directly utilise an external infrastructure that can become the bottleneck, as BENU has encountered.

Scheduling strategy. Although DFS strategy has small memory requirement, it can suffer from low network and CPU utilisation. To saturate CPU power (parallelism), BFS strategy is more widely used for distributed subgraph enumeration. However, it demands a large memory to maintain enormous intermediate results. *Static* heuristics such as batching [5] and region group [65] are used to ease memory tension by limiting the number of initially matched (pivot) vertices/edges. Nevertheless, such static heuristics all lack in a tight bound and can perform poorly in practice. In our experiment (Section 7), we have observed out-of-memory errors from the static heuristics, even while starting with one pivot vertex/edge.

Our Solution and Contributions. In this paper, we take on all the challenges above by presenting a system called HUGE, short for pushing/pulling-**H**ybrid s**U**b**G**raph **E**numeration system. Specifically, we make the following contributions:

(1) Advanced execution plan. We study to break down an execution plan of subgraph enumeration into the logical and physical aspects. Logically, we express all existing works [5, 45, 46, 59, 65, 79, 83] in a uniform join-based framework. As a result, these works can be readily plugged into HUGE to enjoy automatic performance improvement. Physically, we carefully consider the variances of join algorithms (hash join and wco join) and communication modes

(pushing and pulling) for better distributed join processing. As a result, we are able to search for an optimal execution plan to minimise both communication and computation cost in a more generic context without the constraints of existing works.

(2) Pushing/pulling-hybrid compute engine. As the generic execution plan may require both pushing and pulling communication, we develop a hybrid compute engine that efficiently supports dual communication mode. Communication-wise, the dual-mode communication allows the runtime to use either pushing or pulling communication based on which mode produces less cost (according to the optimal plan). As a result, HUGE can benefit from substantially reduced communication volume, as can be seen from Table 1 where HUGE renders the smallest communication volume of 4.6GB, and the lowest communication time of 0.8s. Computation-wise, we notice that cache is the key to efficient pulling-based computation. To this end, we devise a new cache structure called *least-recent-batch used* (LRBU) cache. Together with a two-stage execution strategy, we achieve *lock-free* and *zero-copy* cache access with small synchronisation cost. Additionally, a two-layer intra- and inter-machine work-stealing mechanism is employed for load balancing. Overall, these techniques contribute to HUGE's superior performance. As shown in Table 1, HUGE outperforms SEED, BiGJoin, BENU and RADS by 29.4×, 3.7×, 78.2×, 50.6×, respectively.

(3) BFS/DFS-adaptive scheduler. To manage memory usage without sacrificing computing efficiency, we introduce a BFS/DFS-adaptive scheduler to dynamically control the memory usage of subgraph enumeration. It adopts BFS-style scheduling whenever possible to fully leverage parallelism and adapts dynamically to DFS-style scheduling if the memory exceeds a constant threshold. With the scheduler, we prove that HUGE achieves a tight memory bound of $O(|V_q|^2 \cdot D_G)$ for a subgraph enumeration task, where $|V_q|$ is the number of query vertices and D_G is the maximum degree of the data graph. As a result, HUGE uses only slightly more memory than BENU (purely DFS scheduling) in Table 1, while achieving the best performance among the competitors.

(4) In-depth experiment. We conduct extensive experiments on 7 real-world graphs. Results show the effectiveness of our techniques. As a whole, HUGE outperforms previously best pulling-based algorithm by up to 105×, and the best join-based algorithm by up to 14×, with considerably much less communication and memory usage.

Paper Organization. The rest of this paper is organized as follows. Section 2 introduces preliminaries. Section 3 presents HUGE's optimiser. We present implementation details of HUGE in Section 4 and how computation is scheduled in HUGE to achieve bounded-memory execution in Section 5. We discuss the potential functionalities of HUGE in Section 6. Empirical evaluations are in Section 7, followed by related work in Section 8 and conclusion in Section 9.

2 Preliminaries

Graph Notations. We assume both the data graph and query graph are unlabelled, undirected, and connected[3]. A graph is a tuple $g = (V_g, E_g)$, where V_g is the vertex set and $E_g \subseteq V_g \times V_g$ is the edge set of g. For a vertex $\mu \in V_g$, we use $\mathcal{N}_g(\mu)$ to denote the neighbours of μ, and $d_g(\mu) = |\mathcal{N}_g(\mu)|$ to denote the degree of μ. The average and maximum degree of g is denoted as $\overline{d_g}$ and D_g, respectively. Each

---

[3]Our implementation can seamlessly support directed and labelled graph.

vertex $v \in V_g$ is assigned with an unique integer ID from 0 to $|V_g| - 1$ denoted as $\mathrm{ID}(v)$. A *star*, denoted as $(v; \mathcal{L})$, is a tree of depth 1 with v as the root and $\mathcal{L}$ as the leaf vertices. A *subgraph* g' of g, denoted $g' \subseteq g$, is a graph such that $V_{g'} \subseteq V_g$ and $E_{g'} \subseteq E_g$. A subgraph g' is an *induced subgraph* of g if and only if $\forall \mu, \mu' \in V_{g'}, e = (\mu, \mu') \in E_g$ it holds that $e \in E_{g'}$. We denote $g = g_1 \cup g_2$ for merging two graphs, where $V_g = V_{g_1} \cup V_{g_2}$ and $E_g = E_{g_1} \cup E_{g_2}$.

Subgraph Enumeration. Two graphs q and g are *isomorphic* if and only if there exists a bijective mapping $f : V_q \rightarrow V_g$ such that $\forall (v, v') \in E_q, (f(v), f(v')) \in E_g$. Given a query graph q and a data graph G, the task of *subgraph enumeration* is to enumerate all subgraphs g of G such that g is isomorphic to q. Each isomorphic mapping from q to g is called a *match*.

By representing the query vertices as $\{v_1, v_2, \ldots, v_n\}$, we can simply denote a match f as $\{u_{k_1}, u_{k_2}, \ldots, u_{k_n}\}$, where $f(v_i) = u_{k_i}$ for $1 \le i \le n$. We call a subgraph (often induced) q' of q a *partial query*, and a match of q' a *partial match*. Given a query graph q and data graph G, we denote the result set of subgraph enumeration as $\mathbb{R}_G(q)$, or $\mathbb{R}(q)$ if it is clear.

As a common practice, we apply the method of *symmetry breaking* [28] to avoid duplicated enumeration caused by automorphism (an isomorphism from a graph to itself).

Graph Storage. We *randomly* partition a data graph G in a distributed context as most existing works. For each vertex $\mu \in V_G$, we store it with its adjacency list $(\mu; \mathcal{N}(\mu))$ in one of the partitions. We call a vertex that resides in the local partition as a *local vertex*, and a *remote vertex* otherwise.

Ordered Set. An *ordered set* is a pair $\hat{S} = (S, Ord)$, where S is a set and Ord is the corresponding map of ordering, which maps each element in S to an integer. For $s_1, s_2 \in \hat{S}$, we say $s_1 \le s_2$ if and only if $Ord(s_1) \le Ord(s_2)$. Besides, we use $\min(\hat{S})$ and $\max(\hat{S})$ to denote an element in $\hat{S}$ with the smallest and largest order, respectively.

Remote Procedure Call. A *remote procedure call* (RPC) [54] is when a computer program calls a procedure to execute in a different address space. We refer to the caller as *client* and the executor as *server*. The form of request–response interaction allows RPC to be naturally adopted for pulling communication.

3 Advanced Execution Plan

In this section, we first show that existing works can fit uniformly into a *logical* join-based framework. Then we discuss two primary physical settings for distributed join processing. We eventually propose a dynamic-programming-based optimiser to compute the optimal execution plan for subgraph enumeration.

3.1 A Logical Join-based Framework

It is known that subgraph enumeration can be expressed as a multi-way join of some basic structures called join units (e.g. edges, stars) [46]. Given a query graph q and a data graph G, and a sequence of join units $\{q_1, q_2, \ldots q_k\}$, such that $q = q_1 \cup q_2 \cup \cdots q_k$, we have

$$\mathbb{R}_G(q) = \mathbb{R}_G(q_1) \bowtie \mathbb{R}_G(q_2) \bowtie \cdots \bowtie \mathbb{R}_G(q_k). \quad (1)$$

Logically speaking, existing works all solve the above join via multiple rounds of *two-way* joins, with the variances in join unit ($\mathcal{U}$) and join order (O). For simplicity, we represent a two-way join $\mathbb{R}(q') = \mathbb{R}(q'_l) \bowtie \mathbb{R}(q'_r)$ as a 3-tuple (q', q'_l, q'_r). The join order is an ordered sequence of two-way joins (q', q'_l, q'_r) (where $q', q'_l, q'_r \subseteq q$), with its last element being (q, q_l, q_r).

StarJoin [79] pioneers the idea of using stars as the join unit, as well as the left-deep join order O_{ld}, in which it requires that q'_r is a join unit for each $(q', q'_l, q'_r) \in O_{ld}$. SEED [46] further allows using clique (a complete graph), in addition to stars, as the join unit, after maintaining extra index (triangle index). Moreover, SEED replaces the prior practice of left-deep join order with bushy join, which removes the constraint that each q'_r is a join unit, and hence covers a more complete searching space for an optimal execution plan.

BiGJoin As for BiGJoin[4] [5], we uncover the connections as follows. BiGJoin is based on the wco join algorithm [57]. It matches the query graph one vertex at a time in a predefined order. Let the matching order be $V_q = \{v_1, v_2, \ldots, v_n\}$. The execution plan starts from an empty set, and computes the matches of $\{v_1, \ldots, v_i\}$ in the ith round. Let a partial match after the ith round be $p = \{u_{k_1}, u_{k_2}, \ldots, u_{k_i}\}$ for $i < n$, BiGJoin expands the results in the $(i+1)^{\text{th}}$ round by matching v_{i+1} with $u_{k_{i+1}}$ for p if and only if $\forall_{1 \le j \le i}(v_j, v_{i+1}) \in E_q, (u_{k_j}, u_{k_{i+1}}) \in E_G$. The candidate set of v_{i+1}, denoted as $\mathbb{C}(v_{i+1})$ can be computed by the following intersection

$$\mathbb{C}(v_{i+1}) = \cap_{\forall_{1 \le j \le i} \land (v_j, v_{i+1}) \in E_q} \mathcal{N}_G(u_{k_j}). \quad (2)$$

Definition 3.1. A two-way join (q', q'_l, q'_r) is a *complete star join* if and only if q'_r is a star $(v'_r; \mathcal{L})$ (w.l.o.g. [5]) and $\mathcal{L} \subseteq V_{q'_l}$.

We show how BiGJoin can be expressed in the join-based framework. Let $q'_i = q_1 \cup \cdots \cup q_i$. The procedure of BiGJoin is equivalent to the joins following the left-deep order O_{ld}, where the ith element of O_{ld} is $(q'_{i+1}, q'_i, q_{i+1})$, and it further satisfies that q'_i is an *induced* subgraph of q, and $(q'_{i+1}, q'_i, q_{i+1})$ is a complete star join with $q_{i+1} = (v_{i+1}; \mathcal{L}_{i+1})$.

Example 3.1. Given a BiGJoin's execution of a 4-clique in Figure 1a, we present its corresponding execution plan in HUGE in Figure 1b, where each vertex extension is expressed as a complete star join. As an example, the extension of v_1 from an edge (v_2, v_3) is equivalent to the complete star join of (q'_2, q_1, q_2).

3.2 Physical Join Processing

Given the join-based framework, the performance of subgraph enumeration is further determined by how the join is physically processed. We summarize two primary physical settings for distributed join processing, namely, join algorithm ($\mathcal{A}$) and communication mode (C). Let's consider an arbitrary join (q', q'_l, q'_r).

Join Algorithm. While distributed join is well studied [22, 57, 67], we focus on the algorithms adopted for subgraph enumeration. A distributed hash join algorithm is the foundation of [45, 46, 65, 70, 79]. Given $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$, hash join typically shuffles $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$ based on the join key of $V_{q'_l} \cap V_{q'_r}$. Thus, hash join needs to fully materialize both $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$, which can be wasteful as only a part of $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$ can produce feasible results. In the case that the above join is a *complete star join*, the more efficient wco join algorithm can be used by processing the intersection in Equation 2. Instead of blindly materializing the data for join, wco join can benefit from the worst-case optimal bound [57] to only materialize necessary data.

---

[4] The discussions of BENU [83] and RADS [65] are in the full paper [7].
[5] Note that as join operation is commutative, the condition also applies to q'_l. Thereafter, we always present q'_r without loss of generality (w.l.o.g.) in this paper.
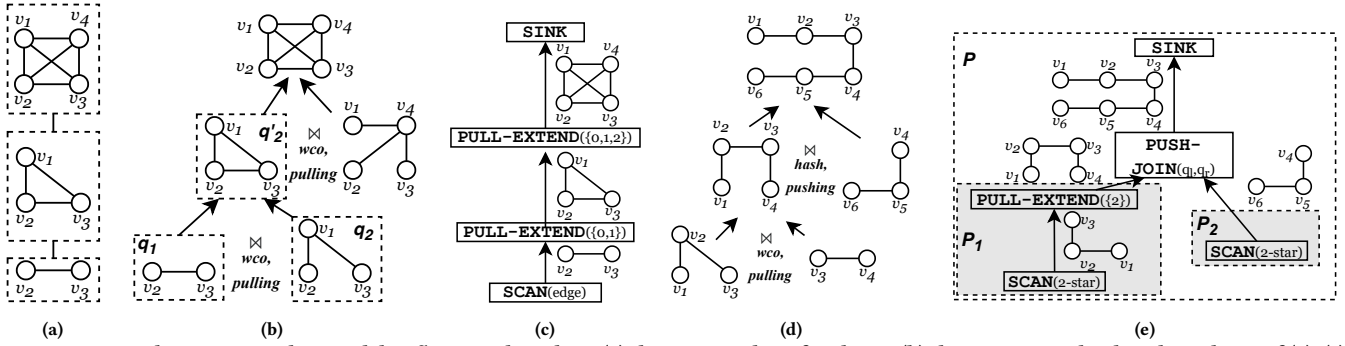
Figure 1: Example execution plans and dataflow graphs, where (a) the BiGJoin **plan of 4-clique; (b) the execution plan by Algorithm 1 of (a); (c) the dataflow graph of (b); (d) an execution plan by Algorithm 1 of 5-path; (e) the dataflow graph of (d).**

Table 2: Existing works and their execution plans.

	Logical		Physical	
Existing Work	$\mathcal{U}$	O	$\mathcal{A}$	C
StarJoin [79]	star	left-deep	hash join	pushing
SEED [46]	star & clique	bushy	hash join	pushing
BiGJoin [5]	star (limited[6])	left-deep	wco join	pushing
BENU [83]	star (limited)	left-deep	wco join	pulling
RADS [65]	star	left-deep	hash join	pulling

Communication Mode. It is straightforward to process the distributed join in the pushing mode. For hash join, we shuffle $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$ by *pushing* the results to the remote machines indexed by the join key. For wco join with $p_r = (v'_r; \mathcal{L})$, we push each $f \in \mathbb{R}(q'_l)$ to the remote machine that owns $f(v)$ continuously for each $v \in \mathcal{L}$ to process the intersection. In certain join scenario, we may leverage the pulling communication mode to process the join, in which a host machine rather *pulls* the graph data than *pushes* the intermediate results. We have the following observation:

Property 3.1. *The pulling communication can be adopted if q'_r is a star $(v'_r; \mathcal{L})$, and the join satisfies one of the following conditions: (C1) $v'_r \in V_{q_l}$; and (C2) the join is a complete star join.*

Let f be a match of q'_l, and $u'_r = f(v'_r)$. Regarding C1, after pulling $\mathcal{N}_G(u'_r)$ from the machine that owns u'_r, the join can be locally processed with the matches of q'_r (rooted on u'_r) enumerated as $|\mathcal{L}|$-combinations over $\mathcal{N}_G(u'_r)$ [45]; regarding C2, while wco join must be used, after pulling $\mathcal{N}_G(f(v))$ for all $v \in \mathcal{L}$ from a remote machine, the intersection (Equation 2) can be locally computed.

Remark 3.1. Given a join(q', q'_l, q'_r), in the pushing mode, we need to transfer data of size $|\mathbb{R}(q'_l)| + |\mathbb{R}(q'_r)|$ in the case of hash join, and $\overline{d_G}|\mathbb{R}(q'_l)|$ in the case of wco join. In the pulling mode, however, we pull *at most* the whole graph data for each machine (i.e. $|E_G|$). As the size of intermediate results (e.g. $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$) is usually order-of-magnitude larger than the data graph itself in subgraph enumeration [6, 83], pulling can potentially benefit from reduced communication.

3.3 Optimal Execution Plan

We summarize existing works and their execution plans in Table 2, it is clear that they are subject to specific settings of join algorithm and communication mode. To pursue an optimal execution plan in a more generic context, we break down an execution plan P

---

[6]It only accepts limited form of stars as discussed before.

Algorithm 1: OptimalExecutionPlan (q).

1   $M_{plan} \leftarrow \{\}, M_{cost} \leftarrow \{\}$;
2   **for** $n \leftarrow 3 \ldots |V_q|$ **do**
3   **forall** *connected subgraph* $q' \subseteq q$ *s.t.* $|V_{q'}| = n$ **do**
4   **if** q' *is a join unit* **then** $M_{cost}[q'] \leftarrow |\mathbb{R}(q')|$;
5   **else forall** *connected subgraphs* $q'_l, q'_r \subset q'$ *s.t.* $q'_l \cup q'_r = q' \wedge E_{q_l} \cap E_{q_r} = \emptyset$ **do**
6          $C \leftarrow M_{cost}[q'_l] + M_{cost}[q'_r] + |\mathbb{R}(q')|$;
7   **if** (q', q'_l, q'_r) *applies pulling by Equation 3* **then**
8            $C \leftarrow C + k|E_G|$;
9   **else** $C \leftarrow C + |\mathbb{R}(q'_l)| + |\mathbb{R}(q'_r)|$;
10   **if** $M_{cost}[q'] = \emptyset$ *or* $M_{cost}[q'] > C$ **then**
11            $M_{cost}[q'] \leftarrow C ; M_{plan}[q'] \leftarrow (q'_l, q'_r)$;

12   $O \leftarrow$ RecoverJoinOrder(M_{plan}) ;
13   $S \leftarrow$ ConfigureJoin(O) ;
14   **return** (O, S)

for subgraph enumeration into the logical settings of join unit ($\mathcal{U}$) and join order (O), as well as physical settings of join algorithm ($\mathcal{A}$) and communication mode (C). Specially, we call $L = (\mathcal{U}, O)$ as the logical plan. We detail the settings of HUGE as follows. By default, we use stars as the join unit, as our system does not assume any index data. We use the bushy join order [37] as it covers more complete searching space. Given an arbitrary join of (q', q'_l, q'_r), we configure its physical settings according to Section 3.2 as:

$$(\mathcal{A}, C) = \begin{cases} \text{(wco join, pulling),} & \text{if it is a complete star join,} \\ \text{(hash join, pulling),} & \text{if } q'_r \text{ is a star } (v'_r; \mathcal{L}) \wedge v'_r \in V_{q_l}, \\ \text{(hash join, pushing),} & \text{otherwise.} \end{cases}$$

(3)

We are now ready to present Algorithm 1 to compute the optimal execution plan for HUGE with the aim of minimizing both computation and communication cost.

The optimiser starts by initializing two empty maps M_{plan} and M_{cost}. Given a sub-query q', M_{plan} stores the mapping from q' to be best-so-far join that produces q', and M_{cost} records the corresponding cost (line 1). Note that we are only interested in the non-trivial case where q is not a join unit. The program goes through the searching space from smaller sub-queries to larger ones (line 2). For a sub-query q', no join is needed if it is a join unit, and we record its computation cost as $|\mathbb{R}(q')|$ (line 4) that can be estimated using the method such as [46, 51]. Otherwise, the optimiser enumerates all pairs of sub-queries q'_l and q'_r that can be joined to produce q' (line 5). The cost of processing the join is computed in line 7-9,

which consists of the cost of processing q'_l and q'_r, the computation cost of q' that is $|\mathbb{R}(q')|$, and the communication cost of the join. If pulling mode is configured, the communication cost is at most $k|E_G|$ (line 8), where k is the number of machines in the cluster (Remark 3.1); otherwise, the cost is equal to the shuffling cost of q'_l and q'_r, that is $|\mathbb{R}(q'_l)| + |\mathbb{R}(q'_r)|$ (line 9). If q' has not been recorded in M_{cost}, or the recorded cost is larger than the current cost C, the new cost and join will be updated to the corresponding entries (line 11). Finally, the optimiser recovers the join order O from M_{plan} and configures the physical settings according to Equation 3 for each join in O (line 12).

Example 3.2. Figure 1b illustrates the optimal execution plan for the 4-clique. In Figure 1d, we further show the optimal execution plan of a 5-path. The two joins are processed via pulling-based wco join and pushing-based hash join, respectively, which demonstrates the need of both pushing and pulling communication. Note that such a plan reflects the works [2, 51] that mix hash join and wco join in a hybrid plan space [5]. Nevertheless, these works are developed in a sequential context where computation is the only concern, while we target the distributed runtime that further considers the best communication mode. In the experiment, we show that our optimal execution plan renders better performance than [2, 51].

Remark 3.2. With the separation of logical and physical settings, we allow users to directly feed existing logical plans into the optimiser, and the optimiser will only configure the physical settings for each join. Even with the same logical plan, we shall see from the experiment (Section 7) that HUGE achieves much better performance due to the optimal settings of join algorithm and communication mode, together with the other system optimisations to be introduced. In this sense, *existing works can be plugged into* HUGE *via their logical plans to enjoy immediate speedup and bounded memory consumption.*

4 The HUGE Compute Engine

Most existing works have been developed on external big-data engines such as Hadoop [29] and Timely dataflow engine [53], or distributed key-value store such as HBase [33] and Cassandra [14]. Big-data engines typically do not support pulling communication. Distributed key-value store, however, lacks support of pushing communication, and can become the bottleneck due to large overhead. Thus, they cannot be adopted to run the execution plan in Section 3 that may require both pushing and pulling communication.

We implement our own pushing/pulling-hybrid compute engine for HUGE. In this section, we introduce the architecture of the engine, the dataflow computation model, and the distributed join processing, especially the novel pulling-based extend operator.

4.1 Architecture

HUGE adopts a shared-nothing architecture in a k-machine cluster. There launches a HUGE runtime in each machine as shown in Figure 2. We briefly discuss the following components, while leaving *Cache* and *Scheduler* to Section 4.4 and Section 5, respectively.

RPC Server: RPC server is used to answer incoming requests from other machines. The server supports two RPCs - GetNbrs and StealWork. GetNbrs takes a list of vertices as its arguments and returns their neighbours. Note that the requested vertices must reside in the current partition. StealWork can steal unprocessed tasks locally and send them to a remote machine for load balancing.
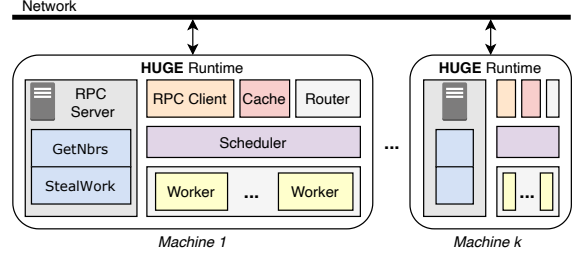

Figure 2: HUGE **Architecture**

RPC Client: An RPC client establishes connections with other machines to handle RPC communication. RPC requests will be sent through RPC client whenever RPCs are called locally.

Router: The router pushes data to other machines. It manages TCP streams connected to remote machines, with a queue for each connection. The data placed in the queue will be transferred to the corresponding machine based on its routing index (e.g. join keys).

Worker: Upon starting, the HUGE runtime initializes a worker pool containing certain number of workers. While an operator is scheduled to run, it will be assigned to the worker pool and executed by all workers to perform the de-facto computation. Each worker has access to the local partition of the graph, RPC client and the in-memory cache. If requesting a local vertex, it will return its neighbours from the local partition. Otherwise, it either returns the cached value if any, or sends an RPC request through the RPC client to obtain the neighbours, caches them, and returns the neighbours.

4.2 Dataflow Model

We adopt the popular dataflow model [1, 53] for HUGE, where computation is abstracted as a dataflow graph. A dataflow graph is a directed acyclic graph (DAG), in which each vertex is an operator, and the directed edges represent data flows. An *operator* is the basic computing unit to run in HUGE, consisted of a predefined computing instruction, a certain number of inbound channels and one outbound channel. The computation of an operator is driven by receiving data from the inbound channels. Given two operators O_1 and O_2 that are connected in the dataflow graph, the data produced from the outbound channel of O_1 will be fed to one of the inbound channels of O_2. We call O_1 the *precursor* of O_2, and O_2 the *successor* of O_1. Upon receiving a dataflow, HUGE distributes it to each machine to drive the computation.

We introduce four primitive operators necessary to understand this paper, namely SCAN, SINK, PUSH-JOIN and PULL-EXTEND. More operators can be added to HUGE to support complex analytical tasks [17, 55] (discussed in Section 6). A valid dataflow must start from a SCAN operator and end with a SINK operator. SCAN(q') accepts a join unit q' as its parameter, takes a local partition of the data graph G, iterates over the partition, and outputs the matches of q' in the partition. SINK is used to consume the results of subgraph enumeration, via either counting or writing to external I/O devices (e.g. disk). We introduce the semantics of PUSH-JOIN and PULL-EXTEND here, and leave the detailed implementation to Section 4.3 and Section 4.4.

PUSH-JOIN. PUSH-JOIN(q'_l, q'_r) processes the pushing-based hash join (Equation 3) of (q', q'_l, q'_r). It configures two inbound channels for the partial results of $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$ from the precursors. It

shuffles (via pushing) $\mathbb{R}(q'_l)$ and $\mathbb{R}(q'_r)$ based on the *join key* of $V_{q'_l} \cap V_{q'_r}$, and then compute the results using local join algorithm.

PULL-EXTEND. PULL-EXTEND(Ext) accepts a parameter of *extend index* $Ext = \{d_1, d_2, \ldots, d_j\}$. For each input data that is a partial result $f = \{u_1, u_2, \ldots, u_i\}$, the operator extends f by one more vertex as $f' = \{u_1, u_2, \ldots, u_i, u_{i+1}\}$, where the set of possible u_{i+1} is computed as $\bigcap_{k=1}^{j} \mathcal{N}_G(f[d_k])$. Each $\mathcal{N}_G(f[d_k])$, if not present in local machine, will be *pulled* from the machine that owns $f[d_k]$.

Execution Plan Translation. The HUGE engine will automatically translate an execution plan given by Algorithm 1 into a dataflow graph. We detail the translation in the full paper [7]. In brief, a SCAN operator is installed for each join unit in the execution plan, and a SINK operator is added to consume the final results. A pulling-based wco join and pushing-based hash join (Equation 3) are translated into a PULL-EXTEND and PUSH-JOIN operator, respectively. For pulling-based hash join, we will show in Section 5.2 how it will be translated via a series of PULL-EXTEND operators for bounded-memory execution.

Example 4.1. The execution plan in Figure 1b is translated into the dataflow presented in Figure 1c, in which each pulling-based wco join is directly translated to a PULL-EXTEND operator. Similarly, the dataflow of Figure 1d is given in Figure 1e, in which the top pushing-based hash join is translated into a PUSH-JOIN operator. The SCAN and SINK operators are added accordingly for computing the join units (stars) and consuming the final results.

Overview of Distributed Execution. In the distributed context, each operator's input data is partitioned to each machine and get processed in parallel. The SCAN operator directly reads from the data graph that follows the graph partitioning strategy (Section 2). The PUSH-JOIN operator takes two inputs, which will be hash-partitioned according to the join key. As for PULL-EXTEND and SINK operators, their input data are also the output data of their precursors and are hence partitioned.

As a common practice of big data engines [53, 74, 75], each operator in HUGE will process a certain number of data as a *batch* at a time. Thus, a batch of data serves as the minimum data processing unit. Without causing ambiguity, when we present "an operator processes a batch of data", we mean that each worker in a machine handles one share of the batch in parallel. A barrier is used to guarantee that all workers in a machine are running the same operator to process the same batch of data at any time. Due to load skew, different machines may run different operators unless explicit global synchronisation is enforced. We resolve such load skew via work stealing (Section 5.3). Depending on the scheduling strategy, the operator will consume certain (at least one) batches of input data in each run. If needed by a remote machine, the output data from an operator will be organised in batches and delegated to the router; otherwise, the data will be placed in the designated buffer to be further processed by the successor as the input.

4.3 PUSH-JOIN Operator

The PUSH-JOIN operator in HUGE performs distributed hash-join that shuffles the intermediate results according to the join key. Similar to [21, 29, 47], we implement a *buffered distributed hash join*. It shuffles the intermediate results (via HUGE's router) with the common join key to the same machine, buffers the received data either in memory or on the disk, and then locally compute the join.

Algorithm 2: LRBU Cache

Data: A key-value map M_{cache}, an ordered set $\hat{S}_{free}$, a set S_{sealed}

1 **Ref Method** Get(vid) $\rightarrow$ **neighbours**
2   **return** $M_{cache}[vid]$
3 **Ref Method** Contains(vid) $\rightarrow$ **Bool**
4   **return** $vid \in M_{cache}$
5 **Mut Method** Insert($vid, neighbours$)
6   **if** CacheIsFull() $\wedge \neg \hat{S}_{free}$.IsEmpty() **then**
7      $u \leftarrow \hat{S}_{free}$.PopSmallest() ; M_{cache}.Remove(u) ;
8    $M_{cache}[vid] \leftarrow neighbours$;
9 **Mut Method** Seal(vid)
10    $\hat{S}_{free}$.Remove(vid) ; S_{sealed}.Add(vid) ;
11 **Mut Method** Release()
12    $largest \leftarrow Ord(\max(\hat{S}_{free})) + 1$;
13   **while** $u \leftarrow S_{sealed}$.Pop() **do**
14      $\hat{S}_{free}$.Insert($u, Ord(u) = largest$) ;

The buffer stage can prevent the memory from being overflowed by either branch of data. We configure a constant buffer threshold, and once the in-memory buffer is full for either branch of the join, we conduct an *external merge sort* on the buffered data via the join keys, and then spill them onto the disk. For join processing, assume that the data is buffered on disk (otherwise is trivial), we can read back the data of each join key in a streaming manner (as the data is sorted,), process the join by conventional nested-loop and write out to the outbound channel. This way, the memory consumption is bounded to the buffer size, which is constant.

4.4 PULL-EXTEND Operator

As mentioned, we implement the PULL-EXTEND operator by pulling communication mode. It requires caching remote vertices for future reuse to reduce the pulling requests via network. BENU directly uses a traditional cache structure (e.g. LRU or LFU [24]) shared by all workers. We have identified two vital issues that considerably slow down cache access from such a straightforward approach.

- Memory copies: Getting a vertex from cache involves at least locating the vertex in the cache, updating the cache position, and finally copying all data (mostly neighbours) of this vertex out. Note that such memory copy is inevitable to avoid dangling pointers in the traditional cache structures, as the memory address of each entry can be changed due to potential replacement.
- Lock: Since the shared cache will be concurrently written and read by multiple workers inside a machine, lock must be imposed on the cache to avoid inconsistency caused by data racing.

To address the above issues, we target a *lock-free* and *zero-copy* cache design for HUGE. While there exist works that focus on reducing the lock contention of concurrent cache such as [88], they are not completely free from locks. For example, benchmarks [12] show that such design can only achieve about 30% reading performance compared to completely lock-free reads. Moreover, existing zero-copy techniques [40, 71, 73, 85] in distributed computation mainly work to dispatch *immutable buffer* directly to network I/O devices, which cannot be applied to our scenario where the cache structure will be frequently mutated. Hence, it requires an innovative design, coupling specifically with the execution of the PULL-EXTEND operator for lock-free and zero-copy cache access.

LRBU Cache. We present our cache structure, LRBU, short for *least recent-batch used* cache. Algorithm 2 outlines the data structure of LRBU, which consists of three members - M_{cache}, $\hat{S}_{free}$, and S_{sealed}. M_{cache} stores the IDs of remote vertices as keys and their neighbours as values. $\hat{S}_{free}$ is an ordered set (Section 2) that keeps track of the orders of remote vertices that can be safely removed from the cache, where vertices with the smallest order can be replaced when the cache is full. S_{sealed} represents a set of remote vertices that cannot be replaced at this time.

There are 5 methods in LRBU. Given a vertex, Get is used to obtain the neighbours if any and Contains checks whether the vertex presents in the cache (line 1-4). Unlike traditional cache structures, we design Get and Contains to take only *immutable* (i.e. read-only) references of the cache structure. As Get and Contains are the two methods for reading the cache, such design makes cache read fully *lock-free* when there is no concurrent writer.

Insert is used to insert a remote vertex and its neighbours into the cache. Additionally, Seal and Release are two unique methods of LRBU. Seal removes a vertex from $\hat{S}_{free}$ and adds it to S_{sealed}. Release pops all values in S_{sealed} and adds them into $\hat{S}_{free}$. The released vertices will be given an order that is larger (line 12) than all existing vertices in $\hat{S}_{free}$. In the Insert method, replacement will be triggered if the cache is full. If $\hat{S}_{free}$ is not empty, the smallest vertex will be popped out for replacement. Thus, calling Seal can prevent a particular vertex from being replaced when cache is full, while calling Release can make the certain vertices replaceable. If $\hat{S}_{free}$ is empty, the insertion will happen regardless of the capacity of the cache. This may cause the cache overflowed, but within only a limited amount as will be shown lately.

Two-stage Execution Strategy. To make full use of LRBU, we break down the execution of PULL−EXTEND into two separate stages, namely *fetch* and *intersect*. The algorithm of an PULL−EXTEND operator is given in Algorithm 3.

In the fetch stage, the PULL−EXTEND scans the input data and collects a set S_{remote} of all remote vertices that need to be fetched in the current batch (line 2-4). It then checks for each remote vertex if the vertex is in the cache already (line 7). If the vertex has been cached, the extender seals this vertex in the cache, which prevents this particular entry to be replaced while processing this batch of data. Otherwise, it puts the vertex into a fetch set S_{fetch}. At the last step of the fetch stage, all vertices in S_{fetch} will be fetched asynchronously by sending the GetNbrs RPC in batches and inserted into the shared cache one single writer (line 8-9). Note that cache write can be well overlapped with the asynchronous RPC requests. In the intersect stage, the extender performs the multiway intersections defined in Equation 2 to obtain the results and send them to the output (line 17). Finally, the sealed vertices are released by calling Release (line 20), which updates cache positions to allow them to be replaced thereafter.

In the execution, remote vertices are sealed at the beginning (line 7) and released at the end (line 20), which represents the vertices used in the very recent batch. As a result, even the cache is overflowed, the amount will not be more than the maximum number of the remote vertices in a batch. When the cache is full, LRBU replaces the vertices with the smallest order, which must be the vertices from the least-recent batch (how LRBU is named).

Algorithm 3: Algorithm of PULL−EXTEND

Input: Input channel $\mathbb{R}_i$, LRBU Cache C
Output: Output channel $\mathbb{R}_{i+1}$

1 **Procedure** Fetch()
2      $S_{remote} \leftarrow \{\}$;
3   **parallel forall** *extended vertex* $u \in \mathbb{R}_i$ **do**
4          $S_{remote}+ = \{u\}$;
5      $S_{fetch} \leftarrow \{\}$;
6   **foreach** $u \in S_{remote}$ **do**
7   **If**(C.Contains(u)) **then** C.Seal(u) **else** $S_{fetch}+ = \{u\}$;
8   **async foreach** $(u, \mathcal{N}_G(u)) \in$ GetNbrs(S_{fetch}) **do**
9   C.Insert($u,\mathcal{N}_G(u)$) ;

10 **Procedure** Intersect()
11      $\mathbb{R}_{i+1} \leftarrow \{\}$;
12   **parallel forall** $p_i \in \mathbb{R}_i$ **do**
13          $nbrs\_list \leftarrow \{\}$;
14   **foreach** *extended vertex* $u \in p_i$ **do**
15   **If** (u is remote) **then** $nbrs\_list+ = \{C.\text{Get}(u)\}$;
16   **else** $nbrs\_list+ = \{\mathcal{N}_G(u)\}$;
17          $\mathbb{C}(v_{i+1}) \leftarrow \cap_{nbrs \in nbrs\_list} nbrs$;
18   **foreach** $v \in \mathbb{C}(v_{i+1})$ **do**
19   **If**($v \notin p_i$) **then** $\mathbb{R}_{i+1}+ = \{p_i + \{v\}\}$;
20   C.Release() ;
21 **return** $\mathbb{R}_{i+1}$;

The two-stage execution strategy, together with the LRBU cache structure, eventually leads to a zero-copy and lock-free cache access in PULL−EXTEND operator:

▶ *Zero-copy.* Each vertex that will be accessed during the intersection is either in the local partition or sealed in the cache (line 15-16). As no modification will occur on the cache structure in the intersect stage (until next batch), we can access the vertex data by simply referencing the memory.

▶ *Lock-free.* Recall that the Get method of LRBU is read-only and no write operation is executed during intersection. Cache write only happens in the stage of fetch (line 7 and 9), and at the end of extend (line 20). As we allow only one cache writer in each machine, the cache access (both read and write) in HUGE is completely lock-free.

Remark 4.1. Our two-stage execution strategy separates fetch and intersect stages for lock-free and zero-copy cache access, which results in vastly improved performance. Synchronisation between fetch stage and intersect stage is necessary, but the overhead is very small as demonstrated in Exp-6 (Section 7). In addition, the initial scan in the fetch procedure can effectively aggregate RPC requests of pulling remote vertices, letting merged RPCs to be sent in bulk, which results in effective network utilisation.

5 Scheduling

We present in this section how we address the memory issue of subgraph enumeration by developing advanced scheduling techniques for HUGE. Note that there requires global synchronisation for PUSH−JOIN operator to guarantee no missing results. To ease the presentation, we first assume that the execution plan contains no PUSH−JOIN to focus on the two scheduling techniques - DFS/BFS-adaptive scheduling for bounded-memory execution and work stealing for load balancing. Then, we introduce how to plugin the PUSH−JOIN operator.

Algorithm 4: DFS/BFS-adaptive Scheduler

Input: An execution plan P
1 $O \leftarrow$ first operator in P ;
2 **while** *there are uncompleted operators* **do**
3   **if** *O has no input* $\wedge O \neq$ SCAN **then**
4      $O \leftarrow P.\text{Precursor}(O)$;
5   **else**
6   Schedule(O) ;
7   **Loop**
8   **if** $Q_O.\text{is\_full}() \vee O$ *has no input* **then**
9   Yield(O) ; **break**;
10   **If**($O = $ SINK) **then** $O \leftarrow P.\text{Precursor}(O)$;
11   **else** $O \leftarrow P.\text{Successor}(O)$;

5.1 Overview

HUGE's scheduler is a daemon thread in each machine that maintains a shared scheduling channel with all workers. Each worker can either send its status to the scheduler or receive scheduling signals. Once an operator O is scheduled (calling schedule(O)), the scheduler will broadcast a *schedule* signal to all workers to run O. The scheduler can also broadcast a *yield* signal to yield the running of O (calling Yield(O)). The workers, once received the yield signal, will complete the current batch before going to sleep.

Without PUSH-JOIN, the dataflow graph is a directed line graph. Thus, there will be at most one precursor (and successor) for each operator. Naively, there are two scheduling orders, depth-first-search (DFS) order and breadth-first-search (BFS) order. DFS scheduler will immediately yield the current operator and schedule the successor, as long as the current operator has completed *one* batch of input data. When obtaining the final results from one batch, the scheduler backtracks to the starting point to consume the next batch. On the other hand, the BFS scheduler will sequentially schedule the operators in the dataflow and not move forward to the successor until it completes computing all input data batches.

DFS scheduler may not fully utilize parallelism and network bandwidth[83], while BFS scheduler can suffer from memory crisis due to the maintenance of enormous intermediate results [45, 46, 59, 79]. Existing works use static heuristics such as region group [65] and batching [5, 47] to constrain the scheduler to only consume a portion (e.g. a batch) of input data (vertice/edges) on the SCAN operator, and will not move to the next portion until it sinks the final results of this portion. Such static heuristics lack in theoretical guarantee and can perform poorly in practice. We have observed out-of-memory errors from the experiment even while starting from one single vertex (e.g. on CW in Section 7).

5.2 DFS/BFS-adaptive Scheduler

We propose a *DFS/BFS-adaptive scheduler* for HUGE to bound the memory usage while keeping high network and CPU utilisation. Specifically, we equip a fixed-capacity output queue Q_O for each output channel of all operators in HUGE. Algorithm 4 illustrates the algorithm. Once an operator is scheduled, the scheduler tends to let it consume as much input data as possible to drive high the CPU utilisation. Meanwhile, each worker will report the number of results in Q_O to the scheduler once it completes computing one batch of data. Whenever Q_O is full, it broadcasts the "yield" signal to all workers to yield the current operator, preventing it from consuming any more batches (line 9). The successor is then scheduled
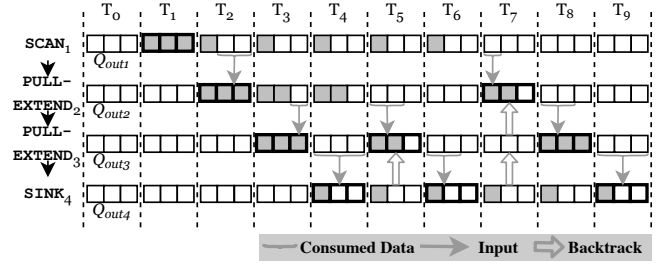

Figure 3: Running Example of DFS/BFS-adaptive Scheduler

to consume the output of the current operator (line 11). If all results in the input channel are consumed, the scheduler backtracks to the precursor (line 4) and repeats the process until the data in all operators has been consumed. Backtracking is always triggered on SINK because it consumes all input data directly (line 10).

Example 5.1. An example is shown in Figure 3 (time slot T_i), with each block represents one batch of data and the operator under schedule highlighted. Each operator has its own output queue Q_{out_i} with fixed size equals to three batches. All queues are initially empty (T_0). The SCAN$_1$ operator scans the data graph at T_1, outputting 3 batches. As the output queue is full, the scheduler yields the operator and schedules PULL-EXTEND$_2$ at T_2. The process repeats until T_4, where the input of SINK$_4$ becomes empty. Thus, the scheduler yields SINK$_4$ and triggers backtracking. It moves to the precursor PULL-EXTEND$_3$, and schedules this operator at T_5. Backtracking is also triggered at T_6 where the input of current operator becomes empty. However, when the scheduler backtracks to PULL-EXTEND$_3$, its input is also empty. So the scheduler further moves forward to PULL-EXTEND$_2$ and starts scheduling PULL-EXTEND$_2$ at T_7.

Bounded-Memory Execution. Different from the static heuristics [5, 65] that lack in a tight bound, we show how the DFS/BFS-adaptive scheduler helps bound memory consumption. Note that SINK operator directly writes data to the external devices and has no need of holding data, which is hence excluded from the memory analysis. We first present the following lemma for an PULL-EXTEND operator.

LEMMA 5.2. *The memory bound of scheduling a* PULL-EXTEND *operator is* $O(|V_q| \cdot D_G)$.

PROOF. All omitted proofs can be found in the full paper [7]. □

We discuss the other two cases in the following, namely SCAN operator and the process of pulling-based hash join.

SCAN. Note that the memory may overflow while enumerating a star (as the join unit). Thus, instead of directly computing the star, we rewrite a SCAN($q' = (v; \mathcal{L})$) operator in a dataflow, via an initial SCAN($q_e = (v, v')$) for any $v' \in \mathcal{L}$ to produce the first edge, which is then chained with $(|\mathcal{L}| - 1)$ PULL-EXTEND ($Ext = \{0\}$) operators to grow the other edges of the star.

Pulling-based Hash Join. Consider a join (q', q'_l, q'_r) that is processed via pulling-based hash join, where q'_r is a star $(v'_r; \mathcal{L})$ (must be so according to Equation 3). Similar to the SCAN operator, a pulling-based hash join may also suffer from exponential memory of computing stars. We show how such a join can be realized via a series of PULL-EXTEND operators to resolve the issue.

As a preliminary notation, given a query graph q with the vertices listed as $\{v_1, v_2, \ldots, v_n\}$ and $V'_q \subseteq V_q$, we denote $\text{Idx}(q|V'_q)$ as an

ordered indices of q w.r.t. V_q', where $i \in \mathsf{Idx}(q|V_q')$ if and only if $v_i \in V_q'$. We split $\mathcal{L}$ into two parts, namely $V_1 = \mathcal{L} \cap V_{q_l'}$ and $V_2 = \mathcal{L} \setminus V_1$, and accordingly divide the execution into a chain of PULL-EXTEND operators. Specifically,

- If $V_1 \neq \emptyset$, we deploy the PULL-EXTEND ($Ext = \mathsf{Idx}(q_l'|V_1)$) operator. Note that this extension does not actually match new query vertex, but rather verify the connection between v_r' and each $v \in V_1$ in a match. Thus, we install a hint on the operator to only preserve the result f where $f(v_r') = u_{i+1}$, and get rid of the extended u_{i+1} in the result.
- For each $v \in V_2$, we sequentially chain a new PULL-EXTEND ($Ext = \mathsf{Idx}(q_l'|\{v\})$) operator to grow the other star edges.

With the above transformation, we further have:

LEMMA 5.3. *Given q_s as a star $(v_s; \mathcal{L})$, the memory bound of scheduling a SCAN (q_s) operator and a pulling-based hash join (q', q_l', q_s) are $O(|\mathcal{L}|^2 \cdot D_G)$ and $O(|\mathcal{L}| \cdot |V_{q'}| \cdot D_G)$, respectively.*

Summarizing from Lemma 5.2 and Lemma 5.3, we finally have:

THEOREM 5.4. *HUGE schedules a subgraph enumeration task with the memory bound of $O(|V_q|^2 \cdot D_G)$.*

5.3 Load Balancing

Graph computation is usually irregular due to the power-law characteristics in real-world graphs [18, 19]. Current solutions [65, 83] often distribute load based on the firstly matched vertex, which may still suffer from load skew. In HUGE, we adopt the work-stealing technique [23, 89] to *dynamically* balance the load. We implement a two-layer intra- and inter-machine work stealing to accommodate HUGE's caching mechanism and BFS/DFS-adaptive scheduler.

For intra-machine work stealing, we maintain a *deque* [16] in each worker. Once the worker executes an operator, it injects the partial results $\mathbb{R}_i$ from the operator's input channel to its own deque. The current worker will pop out $\mathbb{R}_i$ from the back of the deque to do computation. Once a worker has completed its own job by emptying its deque, it will randomly pick one of the workers with non-empty deque, and steal half of the data from the front. For PULL-EXTEND operator, recall that its execution is separated into fetch and intersect stages. While there is barely any skew for fetching data, we only apply intra-machine work stealing to the intersect stage. Specifically, when a worker completes its computation in line 21 of Algorithm 3, it will try to steal the other worker's unprocessed data in line 12 to continue the process.

Inter-machine work stealing happens when any machine completes computing its own job. In this case, the scheduler of the machine will send the StealWork RPC to a random remote machine to steal unprocessed partial results in batches from the input channel of the *top-most unfinished operator*. If receiving data, the scheduler will schedule the corresponding operator to compute the received data; otherwise, it picks another random machine to repeat the attempt. Machines who have completed their own work will send their status to the first machine in the cluster upon completion. The first machine will then broadcast the messages to all other machines in the cluster. A list of finished machines is maintained at each machine, whose job will not be stolen. Once the computation of stolen work is done and there is no more remote work to steal (i.e. all machines have finished their own job), the machine sends the status to the first machine again to mark termination.

Note that the work stealing is applied at operator-level as described to better balance the load. This is because the exponential nature of subgraph enumeration that can cause the intermediate results to explode at any operator on certain vertices (especially large-degree vertices).

5.4 Handling Join Operator

HUGE enforces a synchronisation barrier prior to the PUSH-JOIN operator, thus the join cannot proceed until both precursors complete their computation. With PUSH-JOIN operator, the dataflow graph of HUGE becomes a directed tree.

We first consider a dataflow P with one PUSH-JOIN operator (e.g. Figure 1e), which contains a left subgraph P_1 and a right subgraph P_2. HUGE first computes P_1, and then P_2, whose results will be globally synchronized at the barrier of PUSH-JOIN. As P_1 and P_2 contains only PULL-EXTEND, they can be scheduled via the above scheduling techniques (Sections 5.2 and 5.3). HUGE computes the join after the computation of P_1 and P_2 are completed.

Given P_1 and P_2, we use $P_1 \cdot \rightarrow P_2$ to denote P_1 must be computed before P_2. In Figure 1e, we have $P_l \cdot \rightarrow P$ and $P_r \cdot \rightarrow P$. Each subgraph contains no PUSH-JOIN can be directly scheduled; otherwise, it will be recursively divided by PUSH-JOIN. By constructing a DAG of all subgraphs based on the $\cdot \rightarrow$ relations, a valid execution order can be determined via topological ordering of the DAG.

BFS/DFS-adaptive scheduling is unnecessary for PUSH-JOIN, as the buffering technique (Section 4.3) can already prevent memory from overflowing. While join may produce too many data to overflow the successors, we allow PUSH-JOIN to actively tell the scheduler to yield its execution in case that its output queue is full. Regarding work stealing, we only apply intra-machine stealing for PUSH-JOIN. For the non-trivial case that the buffered data is on disk, a worker can steal job by simply advancing the reading offsets of the other worker's buffered files.

6 Applications

HUGE is designed to be flexible for extending more functionalities. Extended systems can directly benefit from HUGE's pushing/pulling-hybrid communication and bounded-memory execution. We introduce three representative examples.

Cypher-based Distributed Graph Databases. Subgraph enumeration is key to querying graph databases using language like Cypher [56]. HUGE can thus be extended as a Cypher-based distributed graph database, by implementing more operations like projection, aggregation and ordering, and connecting it with a front-end parser (e.g. [32]) and an optimizer with cost estimation for labelled (and/or property) data graph (e.g. [51]).

Graph Pattern Mining (GPM) Systems. A GPM system [23, 38, 50, 80] aims to find all subgraph patterns of interest in a large data graph. It supports applications such as motif counting [52] and frequent subgraph mining [36]. It essentially processes subgraph enumeration repeatedly from small query graphs to larger ones, each time adding one more query vertex/edge. Thus, HUGE can be deployed as a GPM system by adding the control flow like loop in order to construct a more complex dataflow for GPM tasks.

Shortest Path & Hop-constrained Path. HUGE can also be applied to solve more general path queries, such as the classic shortest path problem or hop-constrained path enumeration [58]. Shortest path

Table 3: Table of Datasets

Dataset	$|V|$	$|E|$	d_{max}	d_{avg}
GO	875,713	4,322,051	6,332	5.0
LJ	4,847,571	43,369,619	20,333	17.9
OR	3,072,441	117,185,083	33,313	38.1
UK	18,520,486	298,113,762	194,955	16.1
EU	173,789,185	347,997,111	20	3.9
FS	65,608,366	1,806,067,135	5,214	27.5
CW	978,409,098	42,574,107,469	75,611,696	43.5

can be computed by repeatedly applying PULL-EXTEND from the source vertex until it arrives at the target. For hop-constrained path enumeration, HUGE can conduct a bi-directional BFS by extending from both ends and joining in the middle.

7 Experiments

7.1 Experimental Setup

We follow [47] to build a Rust codebase for a fair comparison. For join-based algorithms (BiGJoin and SEED), we directly adopt the Rust implementations in [47], which contains many optimisations (e.g. symmetry break and compression). For RADS, the original authors have reviewed our implementation. For BENU, we select the distributed key-value database Cassandra [14] to store the data graph as recommended by the original authors. For others, we partition and store the data graph in the compressed sparse row (CSR) format and keep them in-memory. We use the generic compression optimisation [62] whenever it is possible in all implementations, and decompress (by counting) to verify the results.

Hardware. We deploy HUGE in: (1) a local cluster of 10 machines, each with a 4-core Intel Xeon CPU E3-1220, 64GB memory, 1TB disk, connected via a 10Gbps network; (2) an AWS cluster of 16 "r5.8xlarge" instances, each with 32 vCPUs, 256GB memory, 1TB Amazon EBS storage, connected via a 10Gbps network. We run 4 workers in the local cluster and 14 workers in the AWS cluster. All experiments are conducted in the local cluster except Exp-3.

Datasets. We use 7 real-world datasets of different sizes in our experiments as in Table 3. Google (GO), LiveJournal (LJ), Orkut (OR), and Friendster (FS) are downloaded from [72]. UK02 (UK), EU-road (EU), and ClueWeb12 (CW) are obtained from [10], [15], and [60], respectively. The datasets include social graphs (LJ, OR and FS), road networks (EU), and web graphs (GO, UK and CW).

Queries. We use 7 queries according to prior works [5, 46, 47, 62, 65, 83] as shown in Figure 4. The partial orders for symmetry breaking are listed below each query.

Parameters and Metrics. If not otherwise specified, we use q_1-q_3 as the default queries, and UK as the default dataset. Note that we may omit certain results for clarity. We configure the default system parameters of HUGE as batch size: $512K$ (Section 4.1), cache capacity: 30% of the data graph (Section 4.4), and output queue size: 5×10^7 (Section 5.2). We allow 3 hours for each query. OT and OOM are used to indicate a query runs overtime and out of memory, respectively. We measure the total time T, computation time T_R and communication time $T_C = T - T_R$ according to [47]. In the bar char, we present the *ratio* of $\frac{T_C}{T}$ using grey filling, and mark the case of OOM with a ✗ on top of the bar.

7.2 Comparing Existing Solutions

Exp-1: Speed Up Existing Algorithms. We first verify that existing works can be readily plugged into HUGE via their logical
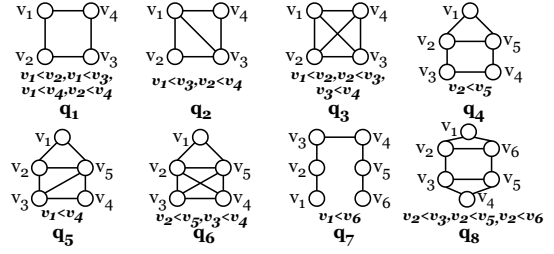
Figure 4: The Query Graphs

plans to receive automatic speedup and bounded-memory execution (Remark 3.2). We run the logical plans of BENU, RADS, SEED, and BiGJoin in HUGE, denoted as HUGE−BENU, HUGE−RADS, HUGE−SEED, and HUGE−WCO, respectively. While SEED's plan may include clique as the join unit, we let HUGE−SEED compute the clique via PULL-EXTEND instead of building the costly triangle index. Note that we use LJ instead of UK for RADS and HUGE−RADS, where both of them run OT on UK because of the poor execution plan of RADS. The results of q_1 and q_2 are presented in Figure 5, with the speedup factor marked on top of each bar.

For BENU, the huge overhead of using Cassandra makes it significantly slower than HUGE−BENU. For RADS, the speedup is less significant, mainly due to the poor execution plans of RADS, especially for q_2, where a massive number of 3-stars must be materialized. SEED runs OOM for q_1, while HUGE−SEED completes in 544 seconds because it processes the join via the more efficient pulling-based wco join according to Equation 3. Note that although SEED replies on the triangle index for querying q_2, our index-free HUGE−SEED still achieves a speedup of 2.5×. Lastly, HUGE−WCO outperforms BiGJoin by 8.5× and 4.8× on q_1 and q_2, with less memory usage (e.g. 4GB vs 12GB for q_1). Specifically, HUGE−WCO reduces the communication time by 764× and 115×, respectively, thanks to the efficient PULL-EXTEND operator.

Exp-2: All-round Comparisons. We compare HUGE (with optimal execution plan by Algorithm 1) on q_1-q_6 with the state-of-the-art algorithms using different data graphs in this experiment (Figure 6). Among all cases, HUGE has the highest completion rate of 90%, where BiGJoin, SEED, RADS, and BENU complete 80%, 50%, 30%, and 30%, respectively. Computation-wise, HUGE outperforms RADS by 54.8×, BENU by 53.3×, SEED by 5.1×, and BiGJoin by 4.0× on average. Note that with the costly triangle index, SEED can query q_3 (a clique) without any join, while the index-free HUGE only runs slightly slower for this query. Communication-wise, the communication time of HUGE takes only a very small portion (the shaded area in a bar) in all cases, due to a combination of caching, batching RPC requests, and good execution plan. In comparison, we can observe that all other algorithms (especially join-based algorithms) spend a notable portion of time communicating data in most cases. Memory-wise, due to the BFS/DFS-adaptive scheduling technique, HUGE keeps the memory usage bounded, and the peak memory usage is 16.6GB among all cases, compared to >64GB (OOM), 2.3GB, >64GB, 34.1GB for RADS, BENU, SEED and BiGJoin, respectively. This experiment shows that HUGE can perform scalable and efficient subgraph enumeration while taking into consideration of computation, communication and memory management.

Exp-3: Web-scale Data Graph. We run HUGE over the web-scale graph CW on the AWS cluster to test its ability in handling large
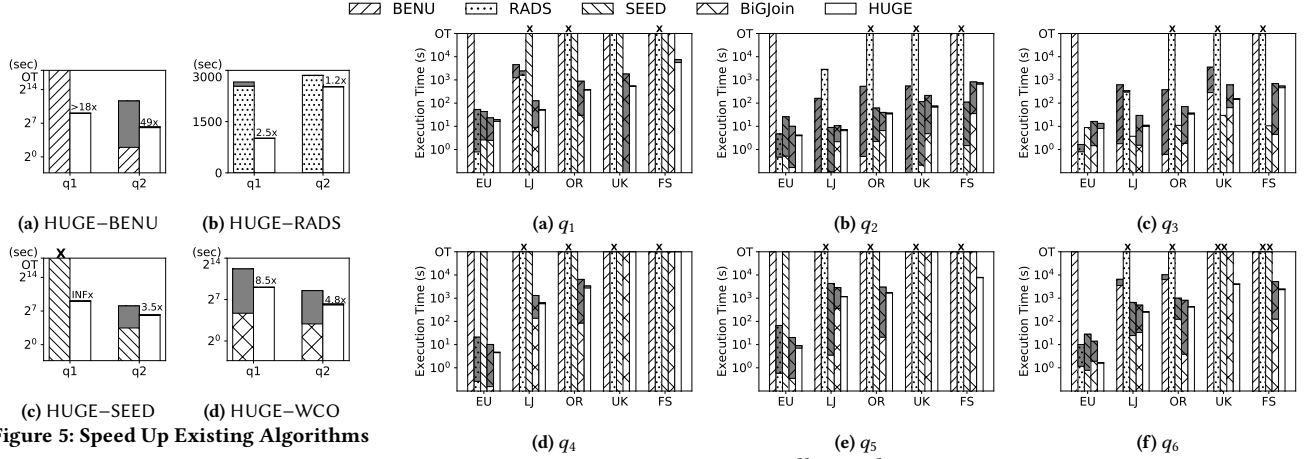
(a) HUGE−BENU **(b) HUGE−RADS**

(c) HUGE−SEED **(d) HUGE−WCO**

Figure 5: Speed Up Existing Algorithms

legend: BENU RADS SEED BiGJoin HUGE

(a) q_1 **(b) q_2** **(c) q_3**

(d) q_4 **(e) q_5** **(f) q_6**

Figure 6: All-Round Comparisons.

Table 4: Throughput on CW

	q_1	q_2	q_3
Throughput	2,895,179,286/s	354,507,087,789/s	206,696,071/s

graphs. The data graph has a raw size of about 370GB (in CSR format) which is larger than the configured memory of the machine. BENU fails to load the graph into Canssandra within one day, so as SEED that needs to build the triangle index. Both RADS and BiGJoin run OOM quickly even when we start with one single vertex in a region group (batch). However, HUGE, runs the queries with a stable memory usage of around 85G when setting the cache capacity size to 30GB and the output queue size to 5×10^8. The number of results on this graph has been estimated to be dramatically large [5]. Therefore, we run each query for 1 hour and report the average throughput ($\frac{|\mathbb{R}|}{3600}$) of HUGE in Table 4. The authors of BiGJoin [5] have used an incremental dataflow to avoid overflowing the memory. In a same-scale cluster (the machine has similar configurations), they obtain the throughput of 26,681,430/s and 46,517,875/s for q_1 and q_3, which is much lower than our results.

7.3 The Design of HUGE

legend: q1 q3

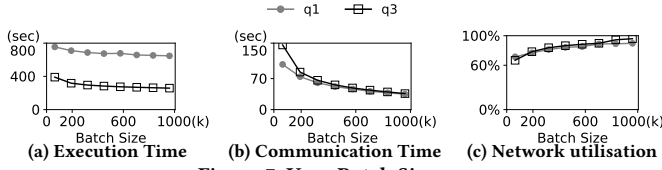**(a) Execution Time** **(b) Communication Time** **(c) Network utilisation**

Figure 7: Vary Batch Size

Exp-4: Effectiveness of Batching. We use a batch of data as the minimum data processing unit (Section 4.1). We investigate how batching affects the HUGE's performance by varying the batch sizes with cache disabled. We report the results of q_1 and q_3 in Figure 7. Let the size of data transferred via network be C (in GB). We measure the network utilisation as $\frac{8C/T_C}{10}$ (10Gbps is the network bandwidth). Increasing the batch size reduces execution and communication time. This is because HUGE's two-stage execution strategy can efficiently aggregate RPC requests within a single batch to improve network utilisation. The average network utilisation starts with 71% when the batch size is 100K, and arrives at 86% and 94% when the size is 512K and 1024K, respectively. As larger batch can make the cache and the output queue more easily overflowed, we set the default batch size to 512K (with satisfactory network utilisation).

legend: q1 q3

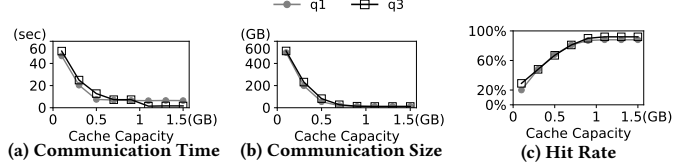**(a) Communication Time** **(b) Communication Size** **(c) Hit Rate**

Figure 8: Vary Cache Capacity

Exp-5: Cache Capacity. We evaluate the impacts of cache capacity on query performance in Figure 8, varying the cache capacity from 0.1GB to 1.5GB. As the capacity increases, the communication time and size decrease rapidly. For q_1, growing the cache capacity from 0.1GB to 0.5GB increases the average hit rate by about 3.5 times, and reduces the total communication by almost 10 times. The performance does not change after the cache capacity exceeds 1.1GB for both queries, whose capacity is enough to hold all remote vertices accessed in these two queries.

Table 5: Vary Cache Design

	LRBU	LRBU-Copy	LRBU-Lock	LRU-Inf	Cncr-LRU
q_1	**589.3s** (27.7s)	734.1s	920.1s	997.5s	2597.1s
q_2	**63.3s** (3.7s)	74.5s	98.0s	107.7s	240.5s
q_3	**200.6s** (24.8s)	314.5s	525.4s	563.4s	980.9s

Exp-6: Cache Design We evaluate the benefit of LRBU's lock-free and zero-copy cache design (Table 5). While enabling the two-stage execution strategy (Section 4.4), we first compare the performance of HUGE with LRBU, LRBU-Copy, LRBU-Lock and LRU-Inf, which represent our LRBU cache, the LRBU cache with memory copy enforced, the LRBU cache with both memory copy and lock enforced, and a LRU cache with infinite capacity[7], respectively. LRBU outperforms LRBU-Copy, LRBU-Lock and LRU-Inf by 1.3×, 1.9× and 2.0×, respectively, which reveals the effectiveness of the zero-copy and lock-free cache access. To evaluate the two-stage execution strategy, we further compare LRBU with a variant called Cncr-LRU, which disables two-stage execution, and applies advanced concurrent LRU-cache [11, 88] for resolving data contentions. The performance gain of LRBU over Cncr-LRU is 4.4× on average. The two-stage execution may bring in synchronisation cost, which is hard to measure directly. Alternatively, as indicated in the bracket of LRBU in Table 5, we measure the whole time spent on the fetch stage t_f, knowing

---

[7]The official Rust LRU library (https://doc.rust-lang.org/0.12.0/std/collections/lru_cache) is used, with the capacity set to the maximum integer.

that it must contain the time for synchronisation. Observe that t_f is merely about 7.5% of the total execution time on average, the synchronisation overhead must thus be small.
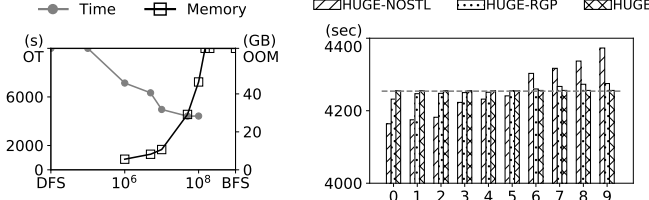

Figure 9: Dynamic Scheduling

Figure 10: Work Stealing

Exp-7: Scheduling. We evaluate the BFS/DFS-adaptive scheduling using a long-running query q_6 that can trigger memory crisis. By varying the output queue size for each operator from 0 to infinite, HUGE's scheduler essentially transforms from DFS scheduler, to BFS/DFS-adaptive scheduler, and eventually to BFS scheduler. Figure 9 shows the execution time and memory consumption for different queue sizes. When the queue size is smaller than 10^6 (including pure DFS scheduling), the query runs OT. As the size increases, the execution time decreases until 10^7, from which the curve gets flat. The execution time at the point 5×10^7 is 38% faster than that at 1×10^6. After the size goes beyond 10^8 (including BFS-style scheduling), the program encounters OOM. The results indicate that HUGE's adaptive scheduler keeps the memory usage bounded while achieving high efficiency.

Exp-8: Load balancing. To test the effectiveness of our work-stealing technique, we compare HUGE with HUGE-NOSTL (HUGE with work stealing disabled, which distributes the load on the pivot vertex as BENU) and HUGE-RGP (with the region-group technique of RADS). The results of q_6 are shown in Figure 10 (other queries in the full paper [7]). With work stealing, HUGE demonstrates the best load balancing as observed from Figure 10. Specifically, we measure the standard deviation of the execution time among all workers, and HUGE has a standard deviation of only 0.5, compared to HUGE-NOSTL's 73.4 and HUGE-RGP's 13.2. We then measure the overhead of our work-stealing technique by computing $Total$ as the aggregated CPU time among all workers. Compared to HUGE-NOSTL, HUGE only slightly increases $Total$ by 0.017%.

Table 6: Runtime for Different Execution Plans

	HUGE−WCO	HUGE−EH	HUGE−GF	HUGE
q_7	OT	7340.28s (170.02s)		
q_8	64.5s (21ms)	67.2s (15.6s)	64.4s (13.9s)	**40.1s (6.5s)**

Exp-9: Comparing Hybrid Plans. We plug into HUGE the logical plans of wco join (as HUGE−WCO), as well as the hybrid plans of EmptyHeaded (HUGE−EH) and GraphFlow (HUGE−GF), and compare them with HUGE (with the plan by Algorithm 1). We use queries q_7 and q_8 for their variances in execution plans, and the graph GO to avoid too many OT cases. For q_7, the optimiser of HUGE produces the same logical plan as EmptyHeaded and GraphFlow that joins a 3-path with a 2-path (via PUSH−JOIN operator), which is better than the wco join plan that must produce the matches of a 4-path. For q_8, HUGE's optimiser, EmptyHeaded and GraphFlow all generate their own hybrid plans, while HUGE's plan renders better performance. This is because that HUGE's optimiser takes both computation and communication into consideration,
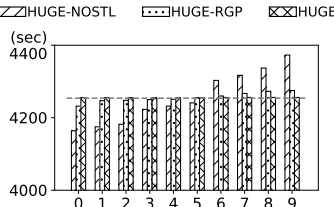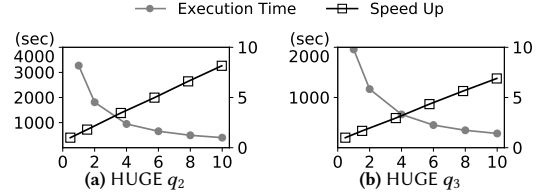

Figure 11: Vary Number of Machines.

while existing hybrid plans are developed in the sequential context where computation is the only concern (Example 3.2).

Exp-10: Scalability. We test the scalability of HUGE by varying the machine number in the cluster from 1 to 10 using the bigger data graph FS (Figure 11). HUGE demonstrates almost linear scalability for both queries. Additionally, we compare the scalability of HUGE with BiGJoin (the OT results of BENU, RADS and SEED are excluded). HUGE scales better than BiGJoin, with the average scaling factor of 7.5× compared to BiGJoin's 6.7× from 1 to 10 machines. Detailed results are in the full paper [7].

8 Related Work

Single-machine Enumeration Many sequential algorithms are proposed, mainly for labelled graphs. Most of them follows Ullmann's [81] backtracking approach [48, 78] with different matching order, pruning rules, and index structure [9, 13, 20, 30, 31, 34, 43, 69, 86, 87]. Parallel algorithms [8, 39, 44, 63, 76, 77] are later proposed to compute subgraph matching using multiple threads. Similarly, EmptyHeaded [2] and GraphFlow [42, 51] aim at computing subgraph enumeration in parallel on a single machine mixing worst-case optimal join [57] and binary join. They can be seamlessly migrated to distributed environment using HUGE (Section 3.1).

Distributed Enumeration MultiwayJoin [3] uses a one-round multiway-join to enumerate subgraphs, and QFrag [68] broadcasts the data graph, These algorithms have poor scalability for large data graphs or complex queries [47]. Another class of algorithms, including EdgeJoin [59], StarJoin [79], PSgL [70], TwinTwigJoin [45], SEED [46], CBF [62], and BiGJoin [5], follows a join-based framework that has been empirically studies and evaluated in [47]. To solve the problem of shuffling huge amount of intermediate results in join-based algorithms, [62] proposed a compression technique to reduce communication cost. BENU and RADS further introduced a pull-based scheme that pull the data graph when needed instead of shuffling the intermediate results. However, they have been demonstrated to be inefficient in this paper.

9 Conclusion

In this paper, we present HUGE, an efficient and scalable subgraph enumeration system in the distributed context. HUGE incorporates an optimiser to compute an advanced execution plan, and a novel architecture that supports pulling/pushing-hybrid communication. Together with a lock-free and zero-copy cache design, and a dynamic scheduling module, HUGE achieves high performance in both computation and communication with bounded memory.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[2] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. ACM, New York, NY, USA, 431–446.

[3] Foto N Afrati, Dimitris Fotakis, and Jeffrey D Ullman. 2013. Enumerating subgraph instances using map-reduce. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on.* IEEE, 62–73.

[4] Noga Alon, Phuong Dao, Iman Hajirasouliha, Fereydoun Hormozdiari, and S Cenk Sahinalp. 2008. Biomolecular network motif counting and discovery by color coding. *Bioinformatics* 24, 13 (2008), i241–i249.

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 691–704.

[6] Albert Atserias, Martin Grohe, and Dániel Marx. 2008. Size Bounds and Query Plans for Relational Joins. In *Proceedings of the 2008 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*. IEEE Computer Society, Washington, DC, USA, 739–748.

[7] Anonymous Authors. [n.d.]. *HUGE: An Efficient and Scalable Subgraph Enumeration System*. Technical Report. Available: https://www.dropbox.com/s/4nnl2gl0hu2xkom/HUGE_Full_Paper.pdf.

[8] Bibek Bhattarai, Hang Liu, and H. Howie Huang. 2019. CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1447–1462.

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient Subgraph Matching by Postponing Cartesian Products. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1199–1214.

[10] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proc. of the Thirteenth International World Wide Web Conference (WWW 2004)*. Manhattan, USA, 595–601.

[11] Caffeine. 2020. Caffeine Cache. https://github.com/ben-manes/caffeine.

[12] Caffeine. 2020. Caffeine Cache Benchmarks. https://github.com/ben-manes/caffeine/wiki/Benchmarks.

[13] Vincenzo Carletti, Pasquale Foggia, Alessia Saggese, and Mario Vento. 2017. Challenging the time complexity of exact subgraph isomorphism for huge and dense graphs with VF3. *IEEE transactions on pattern analysis and machine intelligence* 40, 4 (2017), 804–818.

[14] Cassandra. 2020. Apache Cassandra. https://cassandra.apache.org

[15] The DIMACS Implementation Challenge. 2006. The Challenge9 datasets. http://www.dis.uniroma1.it/challenge9.

[16] David Chase and Yossi Lev. 2005. Dynamic Circular Work-Stealing Deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28.

[17] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: An Efficient Task-Oriented Graph Mining System. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 32, 12 pages.

[18] Fan Chung, Linyuan Lu, and Van Vu. 2003. Spectra of random graphs with given expected degrees. *Proceedings of the National Academy of Sciences* 100, 11 (2003), 6313–6318.

[19] Aaron Clauset, Cosma Rohilla Shalizi, and Mark EJ Newman. 2009. Power-law distributions in empirical data. *SIAM review* 51, 4 (2009), 661–703.

[20] Luigi P Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. 2004. A (sub) graph isomorphism algorithm for matching large graphs. *IEEE transactions on pattern analysis and machine intelligence* 26, 10 (2004), 1367–1372.

[21] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 137–150.

[22] David J. DeWitt and Robert H. Gerber. 1985. Multiprocessor Hash-Based Join Algorithms. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11* (Stockholm, Sweden) *(VLDB '85)*. VLDB Endowment, 151–164.

[23] Vinicius Dias, Carlos H. C. Teixeira, Dorgival Guedes, Wagner Meira, and Srinivasan Parthasarathy. 2019. Fractal: A General-Purpose Graph Pattern Mining System. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery,

[24] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Trans. Storage* 13, 4, Article 35 (Nov. 2017), 31 pages.

[25] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association Rules with Graph Patterns. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1502–1513.

[26] Gary William Flake, Steve Lawrence, C Lee Giles, and Frans M Coetzee. 2002. Self-organization and identification of web communities. *Computer* 35, 3 (2002), 66–70.

[27] Michael R. Garey and David S. Johnson. 1979. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA. 203 pages.

[28] Joshua A. Grochow and Manolis Kellis. 2007. Network Motif Discovery Using Subgraph Enumeration and Symmetry-Breaking. In *Research in Computational Molecular Biology*, Terry Speed and Haiyan Huang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 92–106.

[29] Hadoop. 2020. Apache Hadoop. https://hadoop.apache.org.

[30] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1429–1446.

[31] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 337–348.

[32] Kongzhang Hao, Zhengyi Yang, Longbin Lai, Zhengmin Lai, Xin Jin, and Xuemin Lin. 2019. PatMat: A Distributed Pattern Matching Engine with Cypher. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management* (Beijing, China) *(CIKM '19)*. Association for Computing Machinery, New York, NY, USA, 2921–2924.

[33] HBase. 2020. Apache HBase. https://hbase.apache.org.

[34] Huahai He and Ambuj K. Singh. 2008. Graphs-at-a-Time: Query Language and Access Methods for Graph Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 405–418.

[35] S. Hu, L. Zou, J. X. Yu, H. Wang, and D. Zhao. 2018. Answering Natural Language Questions by Subgraph Matching over Knowledge Graphs. *IEEE Transactions on Knowledge and Data Engineering* 30, 5 (2018), 824–837. https://doi.org/10.1109/TKDE.2017.2766634

[36] J. Huan, W. Wang, and J. Prins. 2003. Efficient mining of frequent subgraphs in the presence of isomorphism. In *Third IEEE International Conference on Data Mining*. 549–552. https://doi.org/10.1109/ICDM.2003.1250974

[37] Yannis E. Ioannidis and Younkyung Cha Kang. 1991. Left-Deep vs. Bushy Trees: An Analysis of Strategy Spaces and Its Implications for Query Optimization. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (Denver, Colorado, USA) *(SIGMOD '91)*. Association for Computing Machinery, New York, NY, USA, 168–177. https://doi.org/10.1145/115790.115813

[38] Kasra Jamshidi, Rakesh Mahadasa, and Keval Vora. 2020. Peregrine: A Pattern-Aware Graph Mining System. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 13, 16 pages.

[39] Xin Jin, Zhengyi Yang, Xuemin Lin, Shiyu Yang, Lu Qin, and You Peng. 2021. FAST: FPGA-based Subgraph Matching on Massive Graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE.

[40] Kafka. 2020. Apache Kafka. http://kafka.apache.org/.

[41] Sanjay Ram Kairam, Dan J. Wang, and Jure Leskovec. 2012. The Life and Death of Online Groups: Predicting Group Growth and Longevity. In *Proceedings of the Fifth ACM International Conference on Web Search and Data Mining* (Seattle, Washington, USA) *(WSDM '12)*. Association for Computing Machinery, New York, NY, USA, 673–682.

[42] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1695–1698.

[43] Hyeonji Kim, Juneyoung Lee, Sourav S. Bhowmick, Wook-Shin Han, JeongHoon Lee, Seongyun Ko, and Moath H.A. Jarrah. 2016. DUALSIM: Parallel Subgraph Enumeration in a Massive Graph on a Single Machine. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1231–1245.

[44] R. Kimmig, H. Meyerhenke, and D. Strash. 2017. Shared Memory Parallel Subgraph Enumeration. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 519–529.

[45] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable Subgraph Enumeration in MapReduce. *Proc. VLDB Endow.* 8, 10 (June 2015), 974–985.

[46] Longbin Lai, Lu Qin, Xuemin Lin, Ying Zhang, Lijun Chang, and Shiyu Yang. 2016. Scalable Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 10, 3 (Nov. 2016), 217–228.

[47] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed Subgraph Matching on Timely Dataflow. *Proc. VLDB Endow.* 12, 10 (June 2019), 1099–1112.

[48] Jinsoo Lee, Wook-Shin Han, Romans Kasperovics, and Jeong-Hoon Lee. 2012. An In-Depth Comparison of Subgraph Isomorphism Algorithms in Graph Databases. *Proc. VLDB Endow.* 6, 2 (Dec. 2012), 133–144.

[49] Jure Leskovec, Ajit Singh, and Jon Kleinberg. 2006. Patterns of influence in a recommendation network. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 380–389.

[50] Daniel Mawhirter and Bo Wu. 2019. AutoMine: Harmonizing High-Level Abstraction and High Performance for Graph Mining. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 509–523.

[51] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (July 2019), 1692–1704.

[52] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[53] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. 2013. Naiad: A Timely Dataflow System. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 439–455.

[54] Bruce Jay Nelson. 1981. *Remote Procedure Call*. Ph.D. Dissertation. USA. AAI8204168.

[55] Neo4j. 2020. Neo4j. https://neo4j.com/.

[56] Neo4j. 2020. openCypher. https://www.opencypher.org.

[57] Hung Q Ngo, Christopher Ré, and Atri Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.* 42, 4 (Feb. 2014), 5–16.

[58] You Peng, Ying Zhang, Xuemin Lin, Wenjie Zhang, Lu Qin, and Jingren Zhou. 2019. Towards Bridging Theory and Practice: Hop-Constrained s-t Simple Path Enumeration. *Proc. VLDB Endow.* 13, 4 (Dec. 2019), 463–476.

[59] Todd Plantenga. 2013. Inexact subgraph isomorphism in MapReduce. *J. Parallel and Distrib. Comput.* 73, 2 (2013), 164–175.

[60] The Lemur Project. 2018. The Clubweb12 dataset. https://lemurproject.org/clueweb12.

[61] N Pržulj, Derek G Corneil, and Igor Jurisica. 2006. Efficient estimation of graphlet frequency distributions in protein–protein interaction networks. *Bioinformatics* 22, 8 (2006), 974–980.

[62] Miao Qiao, Hao Zhang, and Hong Cheng. 2017. Subgraph Matching: On Compression and Computation. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 176–188.

[63] Raghavan Raman, Oskar van Rest, Sungpack Hong, Zhe Wu, Hassan Chafi, and Jay Banerjee. 2014. PGX.ISO: Parallel and Efficient In-Memory Engine for Subgraph Isomorphism. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems* (Snowbird, UT, USA) *(GRADES'14)*. Association for Computing Machinery, New York, NY, USA, 1–6.

[64] John H. Reif. 1985. Depth-first search is inherently sequential. *Inform. Process. Lett.* 20, 5 (1985), 229 – 234.

[65] Xuguang Ren, Junhu Wang, Wook-Shin Han, and Jeffrey Xu Yu. 2019. Fast and Robust Distributed Subgraph Enumeration. *Proc. VLDB Endow.* 12, 11 (July 2019), 1344–1356.

[66] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (Dec. 2017), 420–431.

[67] Donovan A. Schneider and David J. DeWitt. 1989. A Performance Evaluation of Four Parallel Join Algorithms in a Shared-Nothing Multiprocessor Environment. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data* (Portland, Oregon, USA) *(SIGMOD '89)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/67544.66937

[68] Marco Serafini, Gianmarco De Francisci Morales, and Georgos Siganos. 2017. QFrag: Distributed Graph Search via Subgraph Isomorphism. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 214–228.

[69] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 364–375.

[70] Yingxia Shao, Bin Cui, Lei Chen, Lin Ma, Junjie Yao, and Ning Xu. 2014. Parallel Subgraph Listing in a Large-Scale Graph. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 625–636.

[71] P. Shivam, P. Wyckoff, and D. Panda. 2001. EMP: Zero-Copy OS-Bypass NIC-Driven Gigabit Ethernet Message Passing. In *SC '01: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. 49–49. https://doi.org/10.1145/582034.582091

[72] Stanford SNAP. 2020. The SNAP datasets. http://snap.stanford.edu/data/index.html.

[73] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. 2012. Revisiting software zero-copy for web-caching applications with twin memory allocation. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. 355–360.

[74] Spark. 2020. Apache Spark. https://spark.apache.org.

[75] Storm. 2020. Apache Storm. http://storm.apache.org/index.html.

[76] S. Sun, Y. Che, L. Wang, and Q. Luo. 2019. Efficient Parallel Subgraph Enumeration on a Single Machine. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 232–243.

[77] S. Sun and Q. Luo. 2018. Parallelizing Recursive Backtracking Based Subgraph Matching on a Single Machine. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)*. 1–9.

[78] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-Depth Study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1083–1098.

[79] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. 2012. Efficient Subgraph Matching on Billion Node Graphs. *Proc. VLDB Endow.* 5, 9 (May 2012), 788–799.

[80] Carlos H. C. Teixeira, Alexandre J. Fonseca, Marco Serafini, Georgos Siganos, Mohammed J. Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 425–440.

[81] J. R. Ullmann. 1976. An Algorithm for Subgraph Isomorphism. *J. ACM* 23, 1 (Jan. 1976), 31–42.

[82] Kai Wang, Xuemin Lin, Lu Qin, Wenjie Zhang, and Ying Zhang. 2019. Vertex priority based butterfly counting for large-scale bipartite networks. *Proceedings of the VLDB Endowment* 12, 10 (2019), 1139–1152.

[83] Zhaokang Wang, Rong Gu, Weiwei Hu, Chunfeng Yuan, and Yihua Huang. 2019. BENU: Distributed Subgraph Enumeration with Backtracking-Based Framework. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 136–147.

[84] Xifeng Yan, Philip S. Yu, and Jiawei Han. 2004. Graph Indexing: A Frequent Structure-Based Approach. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France) *(SIGMOD '04)*. Association for Computing Machinery, New York, NY, USA, 335–346.

[85] Bairen Yi, Jiacheng Xia, Li Chen, and Kai Chen. 2017. Towards Zero Copy Dataflows Using RDMA. In *Proceedings of the SIGCOMM Posters and Demos* (Los Angeles, CA, USA) *(SIGCOMM Posters and Demos '17)*. Association for Computing Machinery, New York, NY, USA, 28–30.

[86] Shijie Zhang, Shirong Li, and Jiong Yang. 2009. GADDI: Distance Index Based Subgraph Matching in Biological Networks. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, New York, NY, USA, 192–203.

[87] Peixiang Zhao and Jiawei Han. 2010. On Graph Query Optimization in Large Networks. *Proc. VLDB Endow.* 3, 1–2 (Sept. 2010), 340–351.

[88] Da Zheng, Randal Burns, and Alexander S. Szalay. 2013. Toward Millions of File System IOPS on Low-Cost, Commodity Hardware. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 69, 12 pages.

[89] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 301–316.